Standard Treiber Handbuch

E-LAB AVRco

Pascal Multi-Tasking für Single Chips

Version für

AVR

Blaise Pascal Mathematiker 1623-1662

23-Jun-2018

Autor Rolf Hofmann
Editor Gunter Baab

E-LAB Computers

Mikroprozessor-Technik
Industrie-Elektronik
Hard + Software
8-Bit • 16-Bit • 32-Bit

E-LAB Computers
Grombacherstr. 27
D74906 Bad Rappenau
Tel 07268/9124-0
Fax 07268/9124-24
http://www.e-lab.de
info@e-lab.de

Wichtige Information

Weltweit wird versucht fehlerfreie Software herzustellen. Die Betonung liegt dabei auf versucht, denn es besteht eine einhellige Meinung, je komplexer eine Software ist, desto grösser die Wahrscheinlichkeit, dass Fehler eingebaut sind.

Wir sind aber nicht der Meinung, dass das ein Grundgesetz ist, und dass man deshalb mit Fehlern und Problemen einfach leben muss (obwohl das bei manchen Software Giganten offensichtlich so ist ☺).

Sollten Sie Fehler feststellen, so wären wir dankbar für jede Information darüber. Wir werden uns bemühen, dieses Problem möglichst kurzfristig zu lösen.

Es ist ebenfalls internationaler Konsens, dass für Folgekosten, die aus fehlerhafter Software entstehen, der Software Hersteller jedwede Haftung ausschliesst, es sei denn es wurde etwas anderes extra vereinbart.

Mit der Benutzung jeglicher Software Produkte von E-LAB Computers schliessen wir als Hersteller sämtliche Haftung aus daraus entstehenden Kosten bei Fehlern der Software aus.

Sie als Anwender bzw. Benutzer der Software erklären Sich damit einverstanden. Sollte das nicht der Fall sein, so dürfen Sie die Software auch nicht benutzen, bzw. einsetzen.

Wie gesagt, dieser Haftungsausschluss ist international Standard und üblich.

Dieses Handbuch und die zugehörige Software ist geistiges Eigentum von E-LAB Computers und damit urheberrechtlich geschützt. Diese Produkte werden dem Erwerber zur Nutzung überlassen. Der Erwerber darf diese Produkte nicht an dritte weitergeben noch weiterveräussern. Weitergabe von Kopien dieser Produkte an Dritte, ob gegen Endgeld oder nicht, ist ausdrücklich untersagt.

Wir meinen dass Sie, als Benutzer der Software, damit Geld verdienen können und damit auch eine Pflege der Produkte erwarten. Ein Produkt, das fast ausschliesslich aus Raubkopien besteht, bringt dem Hersteller/Autor kein Geld ein. Und damit kann ein Produkt auch nicht gepflegt und weiterentwickelt werden.

Es liegt also auch im Interesse des Anwenders, dass das Urheberrecht beachtet wird.

Das wars der Autor

Inhaltsverzeichnis

1 Einleitung

1.1 Sinn und Zweck von Treibern

Wie jedes Rechner System dient auch ein Mikrocontroller System nicht dem Selbstzweck, sondern kommuniziert mit der Aussenwelt. Dies kann der Mensch sein, andere Rechner Systeme, externe Medien, Sensoren, Aktoren etc. Diese Vielfalt ist enorm.

Diese Kommunikation kann sehr einfach sein, wenn z.B. über ein Port LEDs geschaltet werden sollen. Schon etwas komplizierter wird es, wenn über ein Port mechanische Schalter, Tasten etc. gelesen werden sollen. Da ist z.B. eine Entprellung unerlässlich.

Diese Komplexität geht bis zu Graphik und Filesystem Treibern am oberen Ende. Alle diese Aufgaben werden i.A. durch sogenannte Treiber erledigt, wobei der Name Treiber etwas irreführend ist. Ein Treiber treibt etwas an, aber Tasten und Endschalter werden ja nicht angetrieben. Der engl. Ausdruck *driver* ist da schon etwas besser. Man versteht darunter so was ähnliches wie Steuerung.

Viele Jahre, ja Jahrzehnte, und z.T. bis heute, war es im embedded Bereich üblich dass man sich diese Treiber selbst erstellt hat, möglichst in Assembler. Die üblichen Entwicklungs Systeme haben max. eine Unterstützung der seriellen Schnittstelle angeboten. Viel mehr war auch i.A. nicht notwendig, da die sehr beschränkten Resourcen der Controller keine grossen Sprünge erlaubten.

Da in den letzten Jahren die Controller ihre Rechenleistung von ca. 1MIPS auf heute 20MIPS und mehr erhöht haben und der on-chip Speicher von typ. 1kByte auf bis zu 1MByte erhöht wurde, lassen sich auch mit kleinen Controller sehr komplexe Systeme erstellen. Dazu kommt noch, dass die Anforderungen von den Kunden an die Software sich ebenfalls kontinuierlich erhöht hat. Waren früher ein paar Sensoren, LEDs, Relais und Tasten ausreichend, so sind heute Bedienfelder, LCD oder sogar Graphik LCDs, Filesysteme, Netzwerke und komplexe Berechnungen fast schon Alltag.

Denkt man an den PC, so ist das alles kein grosses Problem, alles eingebaut, sowohl die zugehörige Hardware als auch die notwendigen Treiber. Ein typisches Embedded Entwicklungs System bietet hier auch noch heute fast nichts. Im günstigsten Fall kann man für viel Geld Libraries kaufen, die häufig noch sehr aufwendig angepasst werden müssen.

Da ist dann sehr oft Eigenbau angesagt. Diese selbstgeschriebenen Treiber sind dann natürlich des Entwicklers liebstes Kind und werden gehätschelt und gepflegt. Aber selten hat man dann in seiner eigenen Bibliothek jeden Treiber den man gerade braucht. Also ist wieder der Selbstbau gefragt. Mit langen Entwicklungs und Debug Zeiten.

Deshalb wurden extrem viele Treiber im AVRco System implementiert, so dass sich der Programmierer wieder auf das wesentliche beschränken kann, nämlich seine Anwendung. Da kommt dann immer wieder das Argument „*was ich selbst geschrieben habe, das kenne ich*“ und auch „*bei einem fremden Treiber weiss ich nicht genau wie der funktioniert*“.

Das Gegenargument dazu ist, dass Treiber, die mit einem System mitkommen, normalerweise schon viele 100-mal benutzt wurden und deshalb im wesentlich bug-frei sein sollten. Ausserdem ist Entwicklungs Zeit teuer und diese reduziert sich enorm mit vorhandenen, sehr gut ausgereiften Treibern.

Nicht ohne Stolz kann E-LAB behaupten das Embedded Entwicklungs System mit der grössten eingebauten Treiber Sammlung zu haben.

Nahezu alle Treiber werden durch den ***AVRco Application Wizard*** und auch durch den ***AVRco Simulator*** unterstützt.

2 Übersicht

2.1 AVRco Versionen

alle AVRco Versionen unterstützen alle AVR Controller die ein internes RAM (für den Stack) besitzen, also praktisch die gesamte Palette.

AVRco Profi Version:
die Profi Version enthält alle verfügbaren Treiber, darunter auch sehr komplexe wie z.B. ein FAT16 File System oder eine umfangreiche Library für graphische LCDs.
Weiterhin wird die professionelle Programm Erstellung durch den vollen Support von Units unterstützt.

AVRco Standard Version:
in der Standard Version sind nur die besonders komplexen Treiber *nicht* enthalten.

AVRco Demo Version:
auch die Demo Version unterstützt alle Controller und besitzt alle Treiber der Standard Version.
Die **einzige Einschränkung** ist die Limitierung des erzeugten Code auf eine Größe von 4 kByte.

2.2 Treiber und Handbuch Versionen

Dieses Handbuch bezieht sich auf Treiber die sowohl in der AVRco Standard Version, als auch in der Profi Version enthalten sind.

2.3 Gliederung der Dokumentation

..\E-Lab\DOCs\DocuCompiler.pdf:
enthält die Pascal Sprachbeschreibung und deren Erweiterungen gegenüber dem Standard Pascal

..\E-Lab\DOCs\DocuStdDriver.pdf:
enthält die Beschreibung der Treiber die sowohl in der Standard, also auch in der Profi Version vorhanden sind

..E-Lab\DOCs\DocuProfiDriver.pdf:
enthält die Beschreibung der Treiber die ausschließlich in der Profi Version vorhanden sind

..\E-Lab\DOCs\DocuReference.pdf :
enthält eine Kurzreferenz (die im wesentlichen mit der Online Hilfe identisch ist)

..\E-Lab\DOCs\DocuTools.pdf:
enthält die Beschreibung der integrierten Entwicklungsumgebung, des Simulators, ein Tutorial usw.

..\ E-LAB\IDE\DataSheets\Release-News.txt:
listet die Erweiterungen in chronologischer Reihenfolge auf.
Die Dokumentation der Erweiterungen erfolgt in den oben erwähnten .pdf Files (DocuXXX.pdf)

..\E-Lab\AVRco\Demos\ :
enthält sehr viele Test und Demo Programme

..\E-Lab\DOCs\ :
enthält die Dokumentation sowie weitere Schaltpläne und Datenblätter

3 Treiber AVRco Standard Version

3.1 SwitchPorts: SwitchPort1, SwitchPort2, SwitchPort_G

Ein SwitchPort im AVRco System ist ein entprelltes Port. Entprellung heisst hier, dass ein Portbit mindestens 2 SystemTicks seinen Wert nicht verändern darf, um als stabil erkannt zu werden. Erst dann wird ein solches Bit bzw. der Status dieses Bits in einem dem Port zugeordneten (stable) Byte abgelegt.

Durch die Entprellung wird sicher gestellt, dass z.B. mechanische Schalter und Tasten einen sicheren und eindeutigen Zustand für die Applikation hat. Fehl-Interpretationen werden damit weitgehend vermieden.

Die Entprellung erfolgt im SysTick, weshalb dieser importiert sein muss. Weiterhin kann eine Entprellung nur erfolgen, wenn der allgemeine Interrupt auch freigegeben wurde (EnableInts oder Start_Processes).

Die Applikation kann auf einzelne Bits eines SwitchPorts mit INP_STABLE(BitNum) zugreifen. Diese Funktion liefert den aktuellen Status des gewünschten Bits als true/false zurück.

Für die meisten Anwendungen ist die Funktion INP_RAISE(BitNum) interessanter. Hier wird der veränderte Status zurückgegeben. Hat das PortBit sein Status von stabil-false nach stabil-true geändert, wird ein true zurückgegeben und gleichzeitig aber auch dieser Status zurückgesetzt. D.h. eine positive Flanke wird gelatcht und kann durch diese Funktion ausgelesen werden. Nach dem Auslesen gibt jeder weitere Funktionsaufruf jetzt ein false zurück bis wieder diese Flanke aufgetreten ist. Der Status bleibt solange erhalten, bis diese Funktion aufgerufen wurde, egal wie oft sich das Portbit inzwischen geändert hat. Diese Funktion kann also nur die Aussage treffen, dass mindestens einmal seit dem letzten Funktionsaufruf eine positive Flanke aufgetreten ist. Wie oft ist nicht bekannt und was der aktuelle Status des Portbits ist, wird hierbei nicht beachtet.

Eine weitere Möglichkeit ist das Auslesen des stabilen Zustands des ganzen Ports indem das Byte PORT_STABLE gelesen wird. Es enthält alle entprellten Bits des Ports.

Das AVRco System kennt insgesamt 3 SwitchPorts: SwitchPort1, SwitchPort2 und SwitchPort_G.
Die ersten beiden beziehen sich jeweils auf ein bestimmtes Port und lesen bzw. verarbeiten dieses Port als ganzes. Damit sind total 16 Schalter etc zu erfassen, mit der Einschränkung dass jeweils acht in einem einzigen Port der CPU zusammengefasst sein müssen.

Diese Beschränkung auf ein Port entfällt beim SwitchPort_G. Die 8 Bits dieses „virtuellen“ Ports können auf beliebige Ports der CPU verteilt sein. Der Nachteil des SwitchPortG ist allerdings, dass die Verarbeitung, die im SysTick erfolgt, wesentlich aufwendiger ist (Codegrösse, Zeit) als bei einem normalen Port.

Ganz allgemein gilt, dass der SysTick nicht zu kurz sein sollte, um ein sicheres Entprellen zu gewährleisten. Wenn man bedenkt dass das Prellen mechanischer Schalter sich zwischen 1 und 20msec abspielt, kann man davon ausgehen, dass mit einem SysTick von 1..5msec eine Entprellung in den meisten Fällen nicht sicher funktionieren kann.

Aus diesem Grund ist es sinnvoll bei SysTicks < Entprellzeit nicht bei jedem SysTick zu entprellen, sondern in Intervallen von x SysTicks. Für diesen Zweck gibt es das Define **Debounce** welches einen internen Intervall Timer importiert. Hiermit kann festgelegt werden, dass z.B. nur in jedem fünften Tick die Entprellung durchgeführt wird. Dieser Timer ist dann für alle Entprellungen aktiv, z.B. auch für das MatrixPort und das KeyPort.

Alle SwitchPorts nehmen keinerlei Initialisierung der Ports vor. Es ist Sache der Applikation diese physischen Ports richtig zu initialisieren.

3.1.1 Implementation

Imports
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.

```
Import SysTick, SwitchPort1, ..;
```

Defines
Definition des Port Register, wobei hier immer das PIN-Register gemeint ist!

```
Define   ProcClock   = 8000000;          {Hertz}
         SysTick     = 10;               {msec}
         StackSize   = $0030, iData;
         FrameSize   = $0030, iData;
         SwitchPort1 = PinB;              {SwitchPort}
```

Optional kann hier noch die Polarität der Triggerflanke (Edgemask) ausgewählt werden mit:

```
Define   SwitchPort1 = PinB, $17;        {SwitchPort, edgemask}
```

Eine weitere **Option** ist die Vorgabe der Port Polarität:

```
Define   PolarityP1  = %00000110;        {Polarity SwitchPort1}
```

Der Sinn der Port **Polarität** ist dass man auch bei low-aktiven Eingängen eine positive Ergebnis Logik erhält.

```
PolarityP1  = $F0;       // upper 4bits high active, lower 4bits low active
```

In dem PolarityPx Byte können die Aktiv-Pegel der einzelnen Eingänge (8bits) angegeben werden. Jedes Bit steht für einen Eingang und gibt an, ob eine 0 oder eine 1 durch den geschlossenen Schalter produziert wird. Damit kann, egal wie die Schalter Logik ist, immer mit positiver Logik gearbeitet werden.

Die Edgemaske gibt die **Flanke** für die einzelnen Bits an. Eine binäre Null bedeutet dass eine positive Flanke erfasst wird. Oder wenn die Polarität richtig eingestellt ist, bedeutet eine 0 „onSwitchClose“ trigger und eine 1 „onSwitchOpen“ trigger.

Die Port Polarity Angabe beeinflusst natürlich auch die Wirkungsweise der Edgemask, da zuerst das gelesene physische Port mit dem Wert von PortPolarity verarbeitet wird (XOR). Das Ergebnis dieser Operation wird dann mit der Edgemask verarbeitet.

SwitchPort2 ist in der Definition identisch mit SwitchPort1. Daher gilt obiges ohne Einschränkung auch hier.

Eine weitere **Option** ist die Vorgabe des Entprell Intervalls:

```
Debounce  =  5;    {debounce every 5 SysTicks}
```

Bei sehr schnellen (kurzen) SysTicks < 10mSec und lange prellenden Schaltern oder Tasten ist ein sicheres Entprellen nicht möglich. Deshalb kann mit der Option Debounce das Entprell Intervall vorgegeben werden. Bei einem SysTick von 1mSec und einem Debounce von 20 erfolgt die Entprellung alle 20mSec, was einen guten Wert darstellt. Das Define von Debounce gilt allerdings auch für die anderen entprellten System Treiber wie KeyPort und MatrixPort.

SwitchPort_G unterscheidet sich zwangsläufig von den beiden ersteren Ports:

```
Import  SysTick, SwitchPort_G, ..;
Define  ProcClock     = 8000000;              {Hertz}
        SysTick       = 10;                   {msec}
        StackSize     = $0030, iData;
        FrameSize     = $0030, iData;
        SwitchPort_G  = [ResetKey, PinC, 4]  [StartInput, PinC, 5] [Sensor, PinB, 1], %00000001;
        PolarityP_G   = %00000110;            {Polarity SwitchPort_G}
```

Da beim SwitchPort_G jedes einzelne Bit in einem anderen Port liegen kann, muss jedes einzelne Bit auch komplett definiert werden. Dies geschieht mit bis zu 8 folgenden Pin-Defines:
[PinName, PortName, BitNum]
PinName ist ein Frei definierbarer Name, der in der Applikation als Byte Konstante benutzt werden kann.
PortName ist der Name eines PIN-Registers, kann aber auch eine beliebige numerische Adresse im RAM Bereich sein.
BitNum ist das gewünschte Bit im Port/Adresse PortName.

Die einzelnen Bit Definitionen werden durch [] voneinander getrennt, ein Komma oder Semikolon ist hier nicht zulässig. Am Ende der bis zu 8 Bit-Definitionen kann, getrennt durch ein Komma, auch hier der optionale Parameter ***EdgeMask*** angegeben werden.

Der Port Polarity Parameter ***PolarityP_G*** ist auch hier optional. Ebenfalls das ***Debounce*** Define.

3.1.2 Imports und Exports

Ein SwitchPort Import importiert diverse Bytes, die für das System reserviert sind. Der Port Handler wird in den SysTick eingeklinkt.

Jedes SwitchPort exportiert eine Variable PORT_STABLE die alle Bits des Ports in entprellter Form enthält. Diese Variable kann jederzeit gelesen werden.

Ein SwitchPort kann nach einem CPU Reset undefinierte Zustände haben, abhängig von den Port Pegeln. Deshalb gibt es die Procedure SWITCHPORTx_CLEAR die das Port aufräumt.
Da der Status eines Switchports frühestens 3 SysTicks nach der System-Interrupt Freigabe (EnableInts bzw. Start_Processes) stabil ist, macht es Sinn, erst dann SWITCHPORTx_CLEAR aufzurufen.

Der entprellte Status eines einzelnen PortBits kann mit der Funktion INP_STABLEx(bit) abgefragt werden.

Ob an einem Switchport ein bestimmtes Bit einen Flankenwechsel hatte, kann mit der Funktion INP_RAISEx(bit) abgefragt werden. Hierbei ist zu beachten, dass nur die Flanke gespeichert bzw. gelatcht wird, die mit dem Define EdgeMask festgelegt wurde

Definitionen

```
Var   PORT_STABLE1      : byte;
      PORT_STABLE2      : byte;
      PORT_STABLE_G     : byte;

Procedure SWITCHPORT1_CLEAR;                  // SWITCHPORT1
Procedure SWITCHPORT2_CLEAR                   // SWITCHPORT2
Procedure SWITCHPORT_G_CLEAR;                 // SWITCHPORT1_G

Function INP_STABLE1 (bit : byte) : boolean;  // SWITCHPORT1
Function INP_STABLE2 (bit : byte) : boolean;  // SWITCHPORT2
Function INP_STABLE_G (bit : byte) : boolean; // SWITCHPORT_G

Function INP_RAISE1 (bit : byte) : boolean;   // SWITCHPORT1
Function INP_RAISE2 (bit : byte) : boolean;   // SWITCHPORT2
Function INP_RAISE_G (bit : byte) : boolean;  // SWITCHPORT_G
```

3.1.3 Key Repeat Support

Der Treiber bietet auch eine optionale Unterstützung von externen Repeat Funktionen.
Um eine Auto-Repeat Funktion einzelner Tasten zu implementieren, muss das steuernde Programm Teil dazu wissen, wie lange eine Taste schon aktiviert ist.

Dazu gibt es zu jeder Taste einen optionalen 8bit Timer, der im Scan Intervall hochgezählt wird, solange die Taste aktiv und stabil ist. Der Timer bleibt bei 255 stehen. Wird die Taste wieder losgelassen, wird der Timer vom System auf 0 gesetzt.

Es gibt 8 Timer. Diese Timer sind in ein Array plaziert und können von der Applikation ausgewertet werden.

Soll diese Option benutzt werden muss diese importiert werden.

```
Import SysTick, SwitchPort1, ... ;          // SwitchPort2, SwitchPort_G

From System import SwitchPTimer1;    // SwitchPTimer2, SwitchPTimer_G
```

Dieser Import definiert das Timer Array in dieser Art:

```
Var SwitchTimerArray1 : array[0..7] of byte;
```

Die Applikation kann jetzt jederzeit auf das Array zugreifen mit:

```
if SwitchTimerArray1[2] > 0 then
  // key is pressed
  ...
endif;
```

oder so:

```
case SwitchTimerArray1[4] of
  0     : // key released
        |
  1..10 : // key initially activated
        |
else
  // do any repeat action
endcase;
```

Achtung:
Die Timer werden im Scan Rythmus inkrementiert. In den meisten Fällen ist das 1x pro SysTick.
Ist ***Debounce*** definiert, erfolgen die Inkrements natürlich in (SysTick * Debounce) Abständen.

Da das Switchport immer ein ganzes 8bit Port testet, können auch nicht benutzte Port Bits in die Timer einfliessen. Was jedoch ganz unerwünscht sein düfrte, ist wenn solche Pins in die Pipe geschrieben werden und dann evtl. einen Event auslösen (WaitPipe etc). Deshalb ist es möglich unerwünschte bits auszumaskieren.
Dazu gibt es dieses Define:

```
Define  SwitchPMask1 = $0F;  // SwitchPMask2, SwitchPMask_G
```

Gesetzte Bits in der Maske geben die Verarbeitung frei, ungesetzte sperren diese.

3.1.4 Full Auto Repeat

Der Treiber bietet auch eine optionale komplette interne Auto-Repeat Funktion.
Dazu wird der SwitchPTimer*x* benutzt um die Repeat Funktion zu realisieren.
Die dadurch generierten Key Ereignisse werden in eine spezielle Pipe ***SwitchKeyPipe*** geschrieben.

Die Applikation braucht jetzt normalerweise keine der standard SwitchPort Funktionen mehr. Alle Operationen können mit dieser Pipe durchgeführt werden. Es sind hierbei alle Pipe Funktionen zulässig.

Soll diese Auto-Repeat Option benutzt werden muss der SwitchPTimer importiert werden.

```
Import SysTick, SwitchPort1, ... ;                  // SwitchPort2, SwitchPort_G

From System import Pipes, SwitchPTimer1;   // SwitchPTimer2, SwitchPTimer_G
```

Die Auto-Repeat Funktion muss mit einem Define deklariert werden:

```
Define SwitchPPipe1 = PipeLen, FirstRepeat, RepeatRate; // SwitchPPipe2, SwitchPPipe_G
```

Dadurch wird diese Variable exportiert:

```
SwitchKeyPipe1 : Pipe[PipeLen] of byte;          // SwitchKeyPipe2, SwitchKeyPipe_G
```

Der erste Parameter ***PipeLen*** bestimmt die Länge der Pipe bzw. die Anzahl der Tastendrücke die in ihr gespeichert werden können.
Der zweite Parameter ***FirstRepeat*** definiert die Zeit in msec bevor das Repeat beginnt.
Der dritte Parameter ***RepeatRate*** bestimmt die Zeit in msec zwischen zwei Repeats.

Wird eine Taste gedrückt, wird diese nach dem Entprellen sofort in die Pipe geschrieben. Bleibt diese Taste aktiv, beginnt nach der Zeit ***FirstRepeat*** das Auto-Repeat in Abständen von ***RepeatRate*** msec. Dabei wird dann jedesmal die ID (0..7) dieser Taste in die Pipe geschrieben. Das auto-repeat arbeitet bis entweder diese Taste inaktiv wird oder die Pipe voll ist.

Wird eine Taste inaktiv, so wird die jeweilige Key-ID + $80 (8.bit gesetzt) in die Pipe geschrieben.

Die Applikation kann jetzt jederzeit auf die Pipe zugreifen mit:

```
If PipeStat (SwitchKeyPipe1) <> 0 then
  key:= PipeRecv (SwitchKeyPipe1);
  ...
endif;
```

3.1.5 Support Funktionen

Die Repeat Funktion kann zur Laufzeit ein bzw. ausgeschaltet werden. Dazu sind diese Prozeduren implementiert:

```
Procedure SwitchKeyRepeat1 (rept : boolean);
Procedure SwitchKeyRepeat2 (rept : boolean);
Procedure SwitchKeyRepeat_G (rept : boolean);
```

Grundsätzlich können weiterhin alle SwitchPort Funktionen weiterbenutzt werden, wie z.B. SWITCHPORT1_CLEAR, was allerdings in den seltensten Fällen nötig wird.

Ebenso können auch die Pipe spezifischen Funktionen verwendet werden wie z.B. PipeFlush oder man kann direkt wie zu MSDOS Zeiten in den Tastatur Buffer schreiben mit PipeSend.

Prozesse und Tasks können die Funktion ***WaitPipe*** benutzen.

3.1.6 Variabler Define des SwitchPort

Für das Define des SwitchPort1 und SwitchPort2 kann auch eine Byte Variable definiert werden:

```
Define SwitchPort1 = @name, ...
```

wobei "name" ein beliebig sein kann, da zu diesem Zeitpunkt noch keine Variablen deklariert sind. Das System legt dann die Variable "name" vom Typ Byte automatisch an.
Der Zweck ist, dass z.B. Prozesse oder Tasks dieses Byte mit speziellen IOs (z.B. I2C) "füttern" können und das System dieses entprellt.

3.2 KeyBoard Library Driver 2x2 ... 4x4

Bei vielen Embedded Applikationen sind Steuertasten angeschlossen, die von einem Mensch bedient werden. Solange die Anzahl dieser Tasten sich in einem Eingabe Port unterbringen lassen, ist das kein meistens Problem. Werden es aber mehr Tasten, müssen entsprechend viel Port Pins benutzt werden. Meistens reichen die Port Pins eines Prozessors gerade aus und die Tasten sind nicht mehr unterzubringen.

Hier bietet sich das Multiplex Verfahren an. Hierbei werden die Tasten auf den Kreuzungspunkten einer Matrix angeordnet. Die Matrix wird aus Spalten (Columns) und Reihen (Rows) aufgebaut. Die Anzahl der möglichen Tasten ergibt sich aus dem Produkt von Columns x Rows. Bei 4 Spalten und 4 Reihen sind das max. 16 Tasten. Man kann also davon ausgehen, dass mit einer Matrix die Anzahl der benötigten Port Pins halbiert wird.

Beim Multiplex Verfahren werden die Reihen eine nach der anderen aktiviert, die gesamten Spalten nacheinander eingelesen und ausgewertet. Das Ergebnis wird in Bitfeldern abgelegt. Bei einer 4x4 Matrix erhält man also 16bits als Ergebnis. Um Strom zu sparen, wird immer nur eine Reihe aktiviert, die anderen sind im TriState Zustand.

In der vorliegenden Implementation besitzt jede Spalte ein PullUp Widerstand von 1..10kOhm. Die internen PullUps des AVRs sind zu hochohmig (typ. 80k) dafür. Durch das schnelle scannen werden die positiven Flanken der Spalten durch die kapazitive Belastung durch CMOS-Eingänge, Kabel, Tasten usw. zu flach und es gibt falsche Ergebnisse. Das Scannen erfolgt in der SysTick Routine, weshalb diese importiert werden muss.

Innerhalb des Scannens erfolgt auch ein Entprellen der Tasten. Eine Taste muss mindestens 3 SysTicks ihren aktuellen Status stabil halten, bis eine Änderung akzeptiert wird.

Imports
Der SysTick Treiber muss importiert sein.

```
Import SysTick, MatrixPort, ... ;
```

Durch den Import des MatrixPorts wird automatisch der AufzählungsTyp (Enumeration) “Keys” impotiert:

```
Type  Keys = (Key1, Key2, ..., KeyN);
Const lastMatrixKey : Keys  = KeyN;
```

Defines
Die Funktionsweise und Parameter des Keyboard Treibers muss definiert werden.

```
Define   MatrixRow   = PortA, 4;     {use PortA, start with bit4}
         MatrixCol   = PinA, 0;      {use PinA,  start with bit0}
         MtrixType   = 3, 4          {3 Rows at PortA, 4 Columns at PinA}
         Debounce    = 4;            {optional debounce in systick counts)
```

Die Applikation kann auch die notwendigen IO-Zugriffe selbst durchführen. Dazu muss das Define geändert werden:

```
Define  MatrixPort  = UserPort;
```

MatrixRow und *MatrixCol* dürfen dazu nicht definiert werden. Die Applikation muss dafür diese Call-Back Funktion zur Verfügung stellen:

```
UserDevice MatrixPortIOS : byte;
```

Damit ist es möglich die internen Funktionen wie Entprellen und Scannen in Zusammenhang mit eigener Hardware zu benutzen.

3.2.1 Exportierte Variablen

Memory Organisation
Alle dem Keyboard Treiber zugehörige Variablen liegen aufgereiht hintereinander im Speicher.

_OLDMX2	byte
_OLDMX1	byte
_MATRIX2	byte
_MATRIX1	byte
_CHANGEMX2	byte
_CHANGEMX1	byte
KEYBOARD	semaphore

Außer KEYBOARD darf keine der obigen Variablen benutzt werden!
Die Variable “KeyBoard” ist eine 8-bit Semaphore die mit einem entsprechenden Zugriff gelesen und dekrementiert werden kann. Die Semaphore enthält als Information die Anzahl der Tasten Clicks. Sie wird mit “ClearKeyBoard” zurückgesetzt. Die Semaphore eignet sich hervorragend für einen Prozess, der sich solange schlafen legt, bis ein Tastatur Erreignis eintritt.

```
process HandleKeyBoard (20, 20 : idata);
begin
  WaitSema (KeyBoard);
  case GetKey of
    Key1 : ...; |
  endcase;
end;
```

3.2.2 Exportierte Funktionen und Prozeduren

Function *ReadKey (**const** key : Keys) : boolean;*
Der aktuelle Status einer Taste wird mit true oder false zurückgegeben. Ist die Taste gedrückt, so kommt ein true, andernfalls ein false zurück. Key muss vom Typ “Keys” sein, also Key1..Key16.

Function *KeyRaised (**const** key : Keys) : boolean;*
Der Status einer Taste wird mit true oder false zurückgegeben. Wurde die Taste mindestens einmal gedrückt, so wird ein true zurückgegeben. Eine Aussage, ob die Taste in diesem Moment gedrückt ist, kann nicht getroffen werden. Der KeyBoard Treiber hat eine sog. Memory Funktion, d.h. eine einmal aktivierte Taste wird abgespeichert, bis sie mit KeyRaised ausgelesen wurde oder ein ‘ClearKeyBoard’ ausgeführt wurde. Damit kann sich der Programmierer ein kontinuierliches Pollen der Tasten sparen.

Function *ReadKeyBoard : BitSet of Keys;*
Das Erbenis dieser Funktion ist ein “BitSet of Keys” worin jedes Bit einer Taste entspricht. Das Ergebnis ist der statische Zustand der Keys, nicht die Memory Funktion. Die Definition von “ BitSet of Keys” ist folgende:

```
Type    Keys    = (Key1, Key2, ..., KeyN);
        KeySet  = BitSet of Keys
```

Es gibt mehrere Möglichkeiten mit dem BitSet-Resultat weiterzuarbeiten:

```
bool := [Key1, Key4] in ReadKeyBoard;
bool := ReadKeyBoard in [Key1, Key4];

bool := ReadKeyBoard = [Key1, Key4];
bool := [Key1, Key4] = ReadKeyBoard;
```

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\BitSets**

Function *KeyStatRaised : boolean;*
Der Status des KeyBoards wird mit true oder false zurückgegeben. Wurde mindestens eine Taste einmal gedrückt und diese Taste inzwischen nicht mit “KeyRaised” gelesen, wird die Funktion true. In anderen Worten, die Funktion gibt ein true zurück, wenn eine Taste gedrückt war oder noch gedrückt ist. Damit kann man sich wiederum das auch das Pollen des kompletten KeyBoards mittels “KeyRaised” solange sparen, bis ein true zurück kommt.

Function *GetKeyRaised : Keys;*
Die Funktion ruft “KeyStatRaised” auf und wenn dieses Resultat true ist, wird anschliessend die erste gefundene aktive, gelatchte Taste zurück gegeben. Ist keine Taste aktiv, pollt diese Funktion solange das KeyBoard, bis eine Taste aktiv geworden ist. Das bedeutet, dass wenn keine Taste gedrückt wird, sich das Programm aufhängt.

Function *KeyStat : boolean;*
Der Status des KeyBoards wird mit true oder false zurückgegeben. Ist mindestens eine Taste gedrückt wird die Funktion true. In anderen Worten, die Funktion gibt ein true zurück, wenn eine Taste noch gedrückt ist.

Function *GetKey : Keys;*
Die Funktion ruft “KeyStat” auf und wenn dieses Resultat true ist, wird anschliessend die erste gefundene aktive Taste zurück gegeben. Ist keine Taste aktiv, pollt diese Funktion solange das KeyBoard, bis eine Taste aktiv geworden ist. Das bedeutet, dass wenn keine Taste gedrückt wird, sich das Programm aufhängt.

Procedure *ClearKeyBoard;*
Diese Prozedur setzt das komplette KeyBoard zurück, inkl. dem Gedächtnis “KeyRaised” und der Semaphore.

Procedure *KeyBoardEnable (ena : boolean);*
Diese Prozedur kann das komplette KeyBoard sperren inklusive dem Scanner, Timer etc. Somit kann eine User Eingabe verhindert werden, oder bei zeitkritischen Programm Teilen die SysTick Verarbeitungs Zeit (Interrupt Sperrzeit des SysTicks) temporär auf ein Minimum zurückgesetzt werden.

3.2.3 Zuordnung zwischen Tasten und Bits

Die Taste auf dem Kreuzungspunkt von Row1 und Col1 ist das höchstwertige bit. Dann folgt die Taste auf dem Punkt Row1/Col2 usw.

Hat die Matrix z.B. 12 Tasten, so werden auch 12 bits belegt, d.h. das Ergebnis von Tasten Abfragen kann von Key1..Key12 laufen. Die Taste auf Row1/Col1 ergibt dann Key12. Ist die Anordnung 4Col und 3Row dann hat die Taste auf Row3/Col4 den Wert Key1.

Row1/Col 1 Key12	Row2/Co l1 Key8	Row3/Co l1 Key4
Row1/Col 2 Key11	Row2/Co l2 Key7	Row3/Co l2 Key3
Row1/Col 3 Key10	Row2/Co l3 Key6	Row3/Co l3 Key2
Row1/Col 4 Key9	Row2/Co l4 Key5	Row3/Co l4 Key1

3.2.4 Entprellen

Bei SysTicks < Prellzeit ist es sinnvoll nicht bei jedem SysTick zu entprellen, sondern in Intervallen von x SysTicks. Für diesen Zweck gibt es das

```
Define Debounce = nn;   // nn = SysTicks
```

welches einen internen Intervall Timer importiert. Hiermit kann festgelegt werden, dass z.B. nur in jedem fünften Tick die Entprellung durchgeführt wird. Dieser Timer ist dann für alle Entprellungen aktiv, z.B. auch für das KeyBoard8 und die SwitchPorts.

3.2.5 Key Repeat Unterstützung

Der Treiber bietet auch eine optionale Unterstützung von externen Repeat Funktionen.
Um eine Auto-Repeat Funktion einzelner Tasten zu implementieren, muss das steuernde Programm Teil dazu wissen, wie lange eine Taste schon aktiviert ist.

Dazu gibt es zu jeder Taste einen optionalen 8bit Timer, der im Scan Intervall hochgezählt wird, solange die Taste aktiv und stabil ist. Der Timer bleibt bei 255 stehen. Wird die Taste wieder losgelassen, wird der Timer vom System auf 0 gesetzt.

Es gibt soviele Timer, wie Tasten definiert sind. Diese Timer sind in ein Array plaziert und können von der Applikation ausgewertet werden.

Soll diese Option benutzt werden muss diese importiert werden.

```
Import SysTick, MatrixPort, ... ;

From MatrixPort import MatrixTimer;
```

Dieser Import definiert das Timer Array in dieser Art:

```
var MatrixTimerArray : array[Keys] of byte;
```

Die Applikation kann jetzt jederzeit auf das Array zugreifen mit:

```
if MatrixTimerArray[Key1] > 0 then
   // key is pressed
   ...
endif;
```

oder so:

```
case MatrixTimerArray[Key1] of
  0     : // key released
          |
  1..10 : // key initially activated
          |
else
  // do any repeat action
endcase;
```

Achtung:
Die Timer werden im Scan Rythmus inkrementiert. In den meisten Fällen ist das 1x pro SysTick.
Ist ***Debounce*** definiert, erfolgen die Inkrements natürlich in (SysTick * Debounce) Abständen.

3.2.6 Full Auto Repeat

Der Treiber bietet auch eine optionale komplette interne Auto-Repeat Funktion. Dazu wird der MatrixTimer benutzt um die Repeat Funktion zu realisieren.

Die dadurch generierten Key Ereignisse werden in eine spezielle Pipe ***MatrixKeyPipe*** geschrieben.

Die Applikation braucht jetzt normalerweise keine der standard MatrixPort Funktionen mehr. Alle Operationen können mit dieser Pipe durchgeführt werden. Es sind hierbei alle Pipe Funktionen zulässig.

Soll diese Auto-Repeat Option benutzt werden muss der MatrixTimer importiert werden.

```
Import SysTick, MatrixPort, ... ;

From MatrixPort import MatrixTimer;
```

Die Auto-Repeat Funktion muss mit einem Define deklariert werden:

```
Define MatrixPipe = PipeLen, FirstRepeat, RepeatRate;
```

Dadurch wird diese Variable exportiert:

```
MatrixKeyPipe: Pipe[PipeLen] of Keys;
```

Der erste Parameter ***PipeLen*** bestimmt die Länge der Pipe bzw. die Anzahl der Tastendrücke die in ihr gespeichert werden können.
Der zweite Parameter ***FirstRepeat*** definiert die Zeit in msec bevor das Repeat beginnt.
Der dritte Parameter ***RepeatRate*** bestimmt die Zeit in msec zwischen zwei Repeats.

Wird eine Taste gedrückt, wird diese nach dem Entprellen sofort in die Pipe geschrieben. Bleibt diese Taste aktiv, beginnt nach der Zeit ***FirstRepeat*** das Auto-Repeat in Abständen von ***RepeatRate*** msec. Dabei wird dann jedesmal die ID dieser Taste in die Pipe geschrieben. Das auto-repeat arbeitet bis entweder diese Taste inaktiv wird oder die Pipe voll ist.

Wird eine Taste inaktiv, so wird die jeweilige Key-ID + $80 (8.bit gesetzt) in die Pipe geschrieben.

Die Applikation kann jetzt jederzeit auf die Pipe zugreifen mit:

```
If PipeStat (MatrixKeyPipe) <> 0 then
  key:= PipeRecv (MatrixKeyPipe);
  ...
endif;
```

3.2.7 Support Funktionen

Die Auto-Repeat Funktion des Treibers kann zur Laufzeit ein und ausgeschaltet werden mit:

```
Procedure KeyBoardRepeat (rept : boolean);
```

Grundsätzlich können weiterhin alle MatrixPort Funktionen weiterbenutzt werden, wie z.B. ClearKeyBoard, was allerdings in den seltensten Fällen nötig wird.

Ebenso können auch die Pipe spezifischen Funktionen verwendet werden wie z.B. PipeFlush oder man kann direkt wie zu MSDOS Zeiten in den Tastatur Buffer schreiben mit PipeSend.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\KeyBoardAuto**

3.2.8 UserDevice und Matrix Port

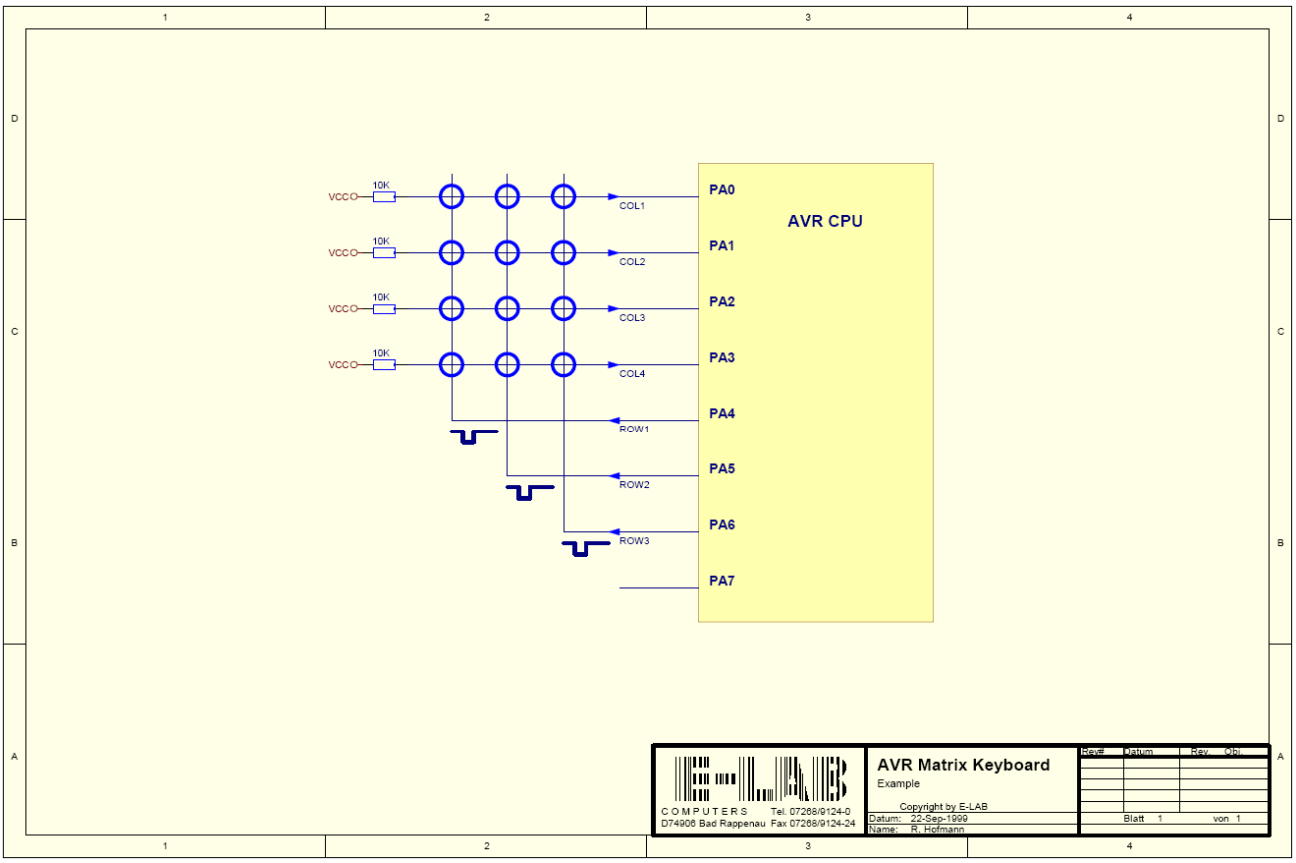
Wenn eine Anwendung die Standard Ports eines AVRs nicht für das MatrixPort-IO benutzen kann, dann besteht auch noch die Möglichkeit, die eigentliche Port Zugriffe selbst durchzuführen, wobei der Treiber das Scannen und die Entprellung übernimmt.

```
Import SysTick, MatrixPort, ... ;

Define
  ProcClock    = 16000000;      {Hertz}
  SysTick      = 10;
  StackSize    = $0064, iData;
  FrameSize    = $0064, iData;
  MatrixPort   = UserPort;
  MatrixType   = 4, 4;

Implementation

{$NORETURNCHECK}
UserDevice MatrixPortIOS : byte;
Begin
  ASM;
  ...
  ...
  EndASM;
end;
```

Der Treiber übergibt an die UserDevice Funktion im Register _ACCA (R17) die aktuelle Col Nummer in der Form 0, 1, 2, 3, 4 etc. Im Register _ACCB (R16) wird als Alternative die Spalten Nummer (COL) als Bit Position übergeben in der Form $1, $2, $4, $8, $10, $20 etc. Je nach dem wie die externe Hardware aufgebaut ist kann nun _ACCA oder _ACCB verwendet werden, um eine Reihe (ROW) einzulesen. Mit einem Multiplexer ist _ACCA vorteilhaft, mit einem Latch etc. ist _ACCB besser geeignet.

Die Applikation muss nun in _ACCA die gelesene Reihe (ROW) an den Treiber zurückliefern.

Der Programm Code sollte unbedingt in Assembler geschrieben werden um hier keine Zeit zu verschwenden, den diese Funktion wird aus dem SysTick heraus aufgerufen. Aus diesem Grund dürfen auch nur _ACCA und _ACCB zur Verarbeitung verwendet werden. Müssen zusätzliche Register benutzt werden, so müssen diese zuvor mit PUSH gerettet und später mit POP wieder zurückgeholt werden. Lokale Variablen sind hier verboten, ebenso der Aufruf Systemtreiber oder Systemfunktionen. Auch keine mathematischen Operation des Systems sind erlaubt.



Schaltplan **KeyBoard**

3.3 KeyBoard Library Driver 2x8 ... 8x8

Bei vielen Embedded Applikationen sind Steuertasten angeschlossen, die von einem Mensch bedient werden. Solange die Anzahl dieser Tasten sich in einem Eingabe Port unterbringen lassen, ist das kein meistens Problem. Werden es aber mehr Tasten, müssen entsprechend viel Port Pins benutzt werden. Meistens reichen die Port Pins eines Prozessors gerade aus und die Tasten sind nicht mehr unterzubringen.

Hier bietet sich das Multiplex Verfahren an. Hierbei werden die Tasten auf den Kreuzungspunkten einer Matrix angeordnet. Die Matrix wird aus Spalten (Columns) und Reihen (Rows) aufgebaut. Die Anzahl der möglichen Tasten ergibt sich aus dem Produkt von Columns x Rows. Bei 8 Spalten und 8 Reihen sind das max. 64 Tasten. Man kann also davon ausgehen, dass mit einer Matrix die Anzahl der benötigten Port Pins mehr als halbiert wird.

Beim Multiplex Verfahren werden die Reihen eine nach der anderen aktiviert, die gesamten Spalten nacheinander eingelesen und ausgewertet. Das Ergebnis wird in Bitfeldern abgelegt. Bei einer 8x8 Matrix erhält man also 64bits als Ergebnis. Um Strom zu sparen, wird immer nur eine Reihe aktiviert, die anderen sind im TriState Zustand.

In der vorliegenden Implementation besitzt jede Spalte ein PullUp Widerstand von 1..10kOhm. Die internen PullUps des AVRs sind zu hochohmig (typ. 80k) dafür. Durch das schnelle scannen werden die positiven Flanken der Spalten durch die kapazitive Belastung durch CMOS-Eingänge, Kabel, Tasten usw. zu flach und es gibt falsche Ergebnisse. Das Scannen erfolgt in der SysTick Routine, weshalb diese importiert werden muss. Um Code und Laufzeit zu sparen, wird vorausgesetzt, dass als COL-Input-Port ein komplettes Port benutzt wird.

Innerhalb des Scannens erfolgt auch ein Entprellen der Tasten. Eine Taste muss mindestens 3 SysTicks ihren aktuellen Status stabil halten, bis eine Änderung akzeptiert wird.

Imports
Der SysTick Treiber muss importiert sein.

```
Import SysTick, KeyPort8, ... ;
```

Durch den Import des KeyPorts wird automatisch der AufzählungsTyp (Enumeration) “Keys” impotiert:

```
Type   Keys   = (Key1, Key2, ..., KeyN);
Const lastKeyBoardKey : Keys = KeyN;
```

Defines
Die Funktionsweise und Parameter des Keyboard Treibers muss definiert werden.

```
Define   KeyB8Row  = PortA, 2;    {use PortA,  start with bit2}
         KeyB8Col  = PinD;        {use PinD complete input port}
         KeyB8Type = 4;           {4 Rows at PortA, 8 Columns at PinD}
         Debounce  = 5;           {nn = SysTicks, optional}
```

Anmerkung:
Das Row-Port muss ein bidirektionales Port sein.
Das Col-Port muss zumindest ein Input-Port sein.

Die Applikation kann auch die notwendigen IO-Zugriffe selbst durchführen. Dazu muss das Define geändert werden:

```
Define  KeyB8Type = UserPort, 3;     {3 Rows,  8 Columns}
```

KeyB8Row und *KeyB8Col* dürfen dazu nicht definiert werden. Die Applikation muss dafür diese Call-Back Funktion zur Verfügung stellen:

```
UserDevice KeyBoard8IOS : byte;
```

Damit ist es möglich die internen Funktionen wie Entprellen und Scannen in Zusammenhang mit eigener Hardware zu benutzen.

3.3.1 Exportierte Variablen

Memory Organisation
Alle dem Keyboard Treiber zugehörige Variablen liegen aufgereiht hintereinander im Speicher.

```
KEYBOARD8        .EQU    076h    ; var iData  semaphore
_KEYBOARDENA     .EQU    077h    ; var iData  boolean
_OLDKB81         .EQU    078h    ; var iData  byte
_OLDKB82         .EQU    079h    ; var iData  byte
_OLDKB83         .EQU    07Ah    ; var iData  byte
_OLDKB84         .EQU    07Bh    ; var iData  byte
_KEYB81          .EQU    07Ch    ; var iData  byte
_KEYB82          .EQU    07Dh    ; var iData  byte
_KEYB83          .EQU    07Eh    ; var iData  byte
_KEYB84          .EQU    07Fh    ; var iData  byte
_CHANGEKB81      .EQU    080h    ; var iData  byte
_CHANGEKB82      .EQU    081h    ; var iData  byte
_CHANGEKB83      .EQU    082h    ; var iData  byte
_CHANGEKB84      .EQU    083h    ; var iData  byte
```

Die Variable “KeyBoard8” ist eine 8-bit Semaphore die mit einem entsprechenden Zugriff gelesen und dekrementiert werden kann. Die Semaphore enthält als Information die ungefähre Anzahl der Tasten Clicks. Sie wird mit “ClearKeyBoard8” zurückgesetzt. Die Semaphore eignet sich hervorragend für einen Prozess, der sich solange schlafen legt, bis ein Tastatur Erreignis eintritt.

```
process HandleKeyBoard (20, 20 : idata);
begin
  WaitSema (KeyBoard8);
  case GetKey8 of
    Key1 : ...; |
  endcase;
end;
```

Der Inhalt der Semaphore (count) gibt nicht immer die tatsächliche Anzahl der Clicks wieder. Sicher ist, dass mindestens die Anzahl “KeyBoard8” vorliegen, es können jedoch auch mehr sein.

3.3.2 Memory Funktion der Tasten

Der KeyBoard Treiber hat eine sog. Memory Funktion, d.h. eine einmal aktivierte Taste wird abgespeichert, bis sie mit “KeyRaised8” oder “GetKeyRaised8” ausgelesen wurde oder ein ‘ClearKeyBoard8’ ausgeführt wurde. Damit kann sich der Programmierer ein kontinuierliches Pollen der Tasten sparen.
Zu beachten ist, dass der Speichervorgang erst dann aktiviert wird, wenn die Taste selbst wieder inaktiv wird. Alle Funktionen, die auf diese Memory Eigenschaft zugreifen, emthalten in ihrem Namen “RAISED”.

3.3.3 Exportierte Funktionen und Prozeduren

Function *ReadKey8 (**const** key : Keys) : boolean;*
Der aktuelle Status einer Taste wird mit true oder false zurückgegeben. Ist die Taste gedrückt, so kommt ein true, andernfalls ein false zurück. Key muss vom Typ “Keys” sein, also Key1..Keyx.

Function *KeyRaised8 (**const** key : Keys) : boolean;*
Der Status einer Taste wird mit true oder false zurückgegeben. Wurde die Taste mindestens einmal gedrückt und losgelassen, so wird ein true zurückgegeben. Eine Aussage, ob die Taste in diesem Moment gedrückt ist, kann nicht getroffen werden.

Function *ReadKeyBoard8 (**const** row : byte) : byte;*
Das Erbenis dieser Funktion ist ein Byte worin jedes Bit einer Taste entspricht. Der Parameter “row” gibt den 8-bit (Key) Block an, wobei “0” die Keys 1..8 und “1” die Keys 9..16 etc. zurückgeben. Das Ergebnis ist der statische Zustand der Keys, nicht die Memory Funktion.

Function *KeyStatRaised8 : boolean;*
Der Status des KeyBoards wird mit true oder false zurückgegeben. Wurde mindestens eine Taste einmal gedrückt und diese Taste inzwischen nicht mit “KeyRaised8” gelesen, wird die Funktion true. In anderen Worten, die Funktion gibt ein true zurück, wenn eine Taste gedrückt war oder noch gedrückt ist. Damit kann man sich wiederum das Pollen des kompletten KeyBoards mittels “KeyRaised8” solange sparen, bis ein true zurück kommt.

Function *GetKeyRaised8 : Keys;*
Die Funktion ruft “KeyStatRaised8” auf und wenn dieses Resultat true ist, wird anschliessend die erste gefundene gelatchte Taste zurück gegeben. Ist keine Taste gespeichert, pollt diese Funktion solange das KeyBoard, bis eine Taste aktiv und inaktiv geworden ist. Das bedeutet, dass wenn keine Taste betätigt wurde, sich das Programm aufhängt.

Function *KeyStat8 : boolean;*
Der Status des KeyBoards wird mit true oder false zurückgegeben. Ist mindestens eine Taste gedrückt wird die Funktion true. In anderen Worten, die Funktion gibt ein true zurück, wenn eine Taste noch gedrückt ist.

Function *GetKey8 : Keys;*
Die Funktion ruft “KeyStat8” auf und wenn dieses Resultat true ist, wird anschliessend die erste gefundene aktive Taste zurück gegeben. Ist keine Taste aktiv, pollt diese Funktion solange das KeyBoard, bis eine Taste aktiv geworden ist. Das bedeutet, dass wenn keine Taste gedrückt wird, sich das Programm aufhängt.

Procedure *ClearKeyBoard8;*
Diese Prozedur setzt das komplette KeyBoard zurück, inkl. dem Gedächtnis “KeyRaised8” und der Semaphore.

Procedure *KeyBoardEnable8 (ena : boolean);*
Diese Prozedur kann das komplette KeyBoard sperren inklusive dem Scanner, Timer etc. Somit kann eine User Eingabe verhindert werden, oder bei zeitkritischen Programm Teilen die SysTick Verarbeitungs Zeit (Interrupt Sperrzeit des SysTicks) temporär auf ein Minimum zurückgesetzt werden.

Ein disable (Sperrung) setzt automatisch das KeyBoard zurück. Falls Timer und Pipe importiert wurden, werden diese ebenfalls zurückgesetzt.

3.3.4 Zuordnung zwischen Tasten und Bits

Die Taste auf dem Kreuzungspunkt von Row1 und Col1 ist das niederwertigste bit. Dann folgt die Taste auf dem Punkt Row1/Col2 usw.

Hat die Matrix z.B. 32 Tasten, so werden auch 32 bits belegt, d.h. das Ergebnis von Tasten Abfragen kann von Key1..Key32 laufen. Die Taste auf Row1/Col1 ergibt dann Key1.

	Row1	Row2	Row3	Row4
Col1	Key1	Key9	Key17	Key25
Col2	Key2	Key10	Key18	Key26
Col3	Key3	Key11	Key19	Key27
Col4	Key4	Key12	Key20	Key28
Col5	Key5	Key13	Key21	Key29
Col6	Key6	Key14	Key22	Key30
Col7	Key7	Key15	Key23	Key31
Col8	Key8	Key16	Key24	Key32

3.3.5 Entprellen

Bei SysTicks < Prellzeit ist es sinnvoll nicht bei jedem SysTick zu entprellen, sondern in Intervallen von x SysTicks. Für diesen Zweck gibt es das

```
Define Debounce = nn;   // nn = SysTicks
```

welches einen internen Intervall Timer importiert. Hiermit kann festgelegt werden, dass z.B. nur in jedem fünften Tick die Entprellung durchgeführt wird. Dieser Timer ist dann für alle Entprellungen aktiv, z.B. auch für das MatrixPort und die SwitchPorts.

3.3.6 Key Repeat Unterstützung

Der Treiber bietet auch eine optionale Unterstützung von externen Repeat Funktionen.
Um eine Auto-Repeat Funktion einzelner Tasten zu implementieren, muss das steuernde Programm Teil dazu wissen, wie lange eine Taste schon aktiviert ist.

Dazu gibt es zu jeder Taste einen optionalen 8bit Timer, der im Scan Intervall hochgezählt wird, solange die Taste aktiv und stabil ist. Der Timer bleibt bei 255 stehen. Wird die Taste wieder losgelassen, wird der Timer vom System auf 0 gesetzt.

Es gibt soviele Timer, wie Tasten definiert sind. Diese Timer sind in ein Array plaziert und können von der Applikation ausgewertet werden.

Soll diese Option benutzt werden, muss diese importiert werden.

```
Import SysTick, KeyPort8, ... ;

From KeyPort8 import KeyBoardTimer;
```

Dieser Import definiert das Timer Array in dieser Art:

```
Var KeyboardTimers : array[Keys] of byte;
```

Die Applikation kann jetzt jederzeit auf das Array zugreifen mit:

```
if KeyboardTimers[Key1] > 0 then
  // key is pressed
  ...
endif;
```

oder so:

```
case KeyboardTimers[Key1] of
  0     : // key released
        |
  1..10 : // key initially activated
        |
else
  // do any repeat action
endcase;
```

Achtung:
Die Timer werden im Scan Rythmus inkrementiert. In den meisten Fällen ist das 1x pro SysTick.
Ist ***Debounce*** definiert, erfolgen die Inkrements natürlich in (SysTick * Debounce) Abständen.

3.3.7 Full Auto Repeat

Der Treiber bietet auch eine optionale komplette interne Auto-Repeat Funktion.
Dazu wird der KeyboardTimer benutzt um die Repeat Funktion zu realisieren.
Die dadurch generierten Key Ereignisse werden in eine spezielle Pipe ***KeyboardPipe*** geschrieben.

Die Applikation braucht jetzt normalerweise keine der standard KeyPort Funktionen mehr. Alle Operationen können mit dieser Pipe durchgeführt werden. Es sind hierbei alle Pipe Funktionen zulässig.

Soll diese Auto-Repeat Option benutzt werden muss der KeyPortTimer importiert werden.

```
Import SysTick, KeyPort8, ... ;

From KeyPort8 import KeyBoardTimer;
```

Die Auto-Repeat Funktion muss mit einem Define deklariert werden:

```
Define KeyB8Pipe = PipeLen, FirstRepeat, RepeatRate;
```

Dadurch wird diese Variable exportiert:

```
KeyboardPipe : Pipe[PipeLen] of Keys;
```

Der erste Parameter ***PipeLen*** bestimmt die Länge der Pipe bzw. die Anzahl der Tastendrücke die in ihr gespeichert werden können.
Der zweite Parameter ***FirstRepeat*** definiert die Zeit in msec bevor das Repeat beginnt.
Der dritte Parameter ***RepeatRate*** bestimmt die Zeit in msec zwischen zwei Repeats.

Wird eine Taste gedrückt, wird diese nach dem Entprellen sofort in die Pipe geschrieben. Bleibt diese Taste aktiv, beginnt nach der Zeit ***FirstRepeat*** das Auto-Repeat in Abständen von ***RepeatRate*** msec. Dabei wird dann jedesmal die ID dieser Taste in die Pipe geschrieben. Das auto-repeat arbeitet bis entweder diese Taste inaktiv wird oder die Pipe voll ist.

Wird eine Taste inaktiv, so wird die jeweilige Key-ID + $80 (8.bit gesetzt) in die Pipe geschrieben.
Die Applikation kann jetzt jederzeit auf die Pipe zugreifen mit:

```
If PipeStat (KeyboardPipe) <> 0 then
  key:= PipeRecv (KeyboardPipe);
  ...
endif;
```

3.3.7.1 Support Funktionen

Die Auto-Repeat Funktion des Treibers kann zur Laufzeit ein und ausgeschaltet werden mit:

```
Procedure KeyBoardRepeat8 (rept : boolean);
```

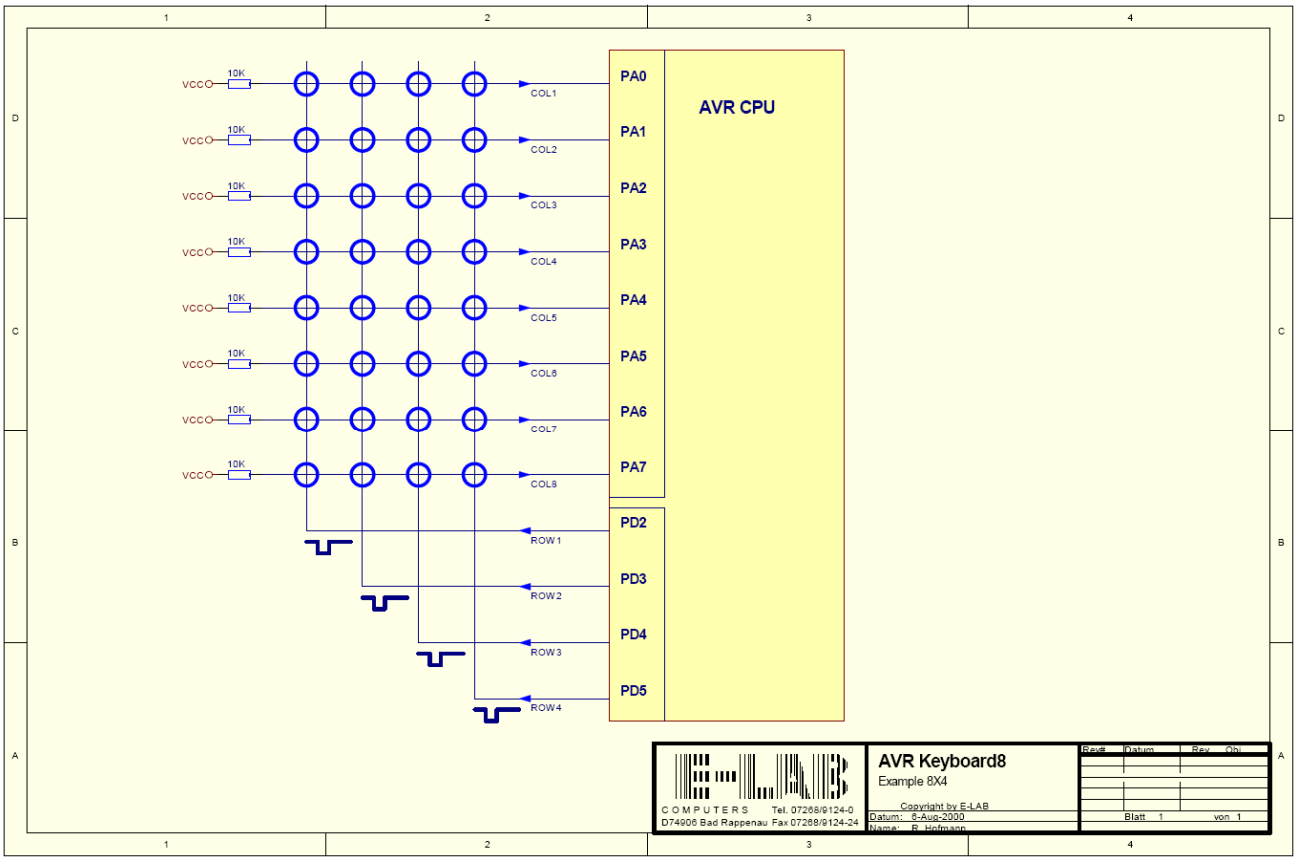
Grundsätzlich können weiterhin alle KeyPort Funktionen weiterbenutzt werden, wie z.B. ClearKeyBoard8, was allerdings in den seltensten Fällen nötig wird.

Ebenso können auch die Pipe spezifischen Funktionen verwendet werden wie z.B. PipeFlush oder man kann direkt wie zu MSDOS Zeiten in den Tastatur Buffer schreiben mit PipeSend.

Bitte beachten:
Beim Auto-Repeat besitzt jede Taste einen 8bit Timer. Das sind max. 64 Bytes Speicherbedarf.
Da alle Operationen (Scannen, Entprellen, Timer, Pipe und Repeat) im SysTick stattfinden, bedeutet das eine nicht unerhebliche Last für diesen. Pro 8 Keys muss mit ca. 18usec@16MHz Verarbeitungs Zeit gerechnet werden. Worstcase (64 keys) sind das ca. 150usec. Um diese Zeit erhöht sich auch die Interrupt Sperrzeit im SysTick wenn der Debounce Timer abgelaufen ist. Deshalb sollte die CPU möglichst mit 16MHz oder mehr laufen.

Programm Beispiel und Schaltung:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\KeyBoardAuto8**



Schaltplan **KeyBoard8**

3.3.8 UserDevice und KeyBoard8

Wenn eine Anwendung die Standard Ports eines AVRs nicht für das KeyBoard-IO benutzen kann, dann besteht auch noch die Möglichkeit, die eigentliche KeyBoard Zugriffe selbst durchzuführen, wobei der Treiber das Scannen und die Entprellung übernimmt.

```
Import SysTick, KeyPort8, ... ;

Define
  ProcClock    = 16000000;       {Hertz}
  SysTick      = 10;
  StackSize    = $0064, iData;
  FrameSize    = $0064, iData;
  KeyB8Type    = UserPort, 5;    {5 Rows,  8 Columns}

Implementation

{$NORETURNCHECK}
UserDevice KeyBoard8IOS : byte;
begin
  ASM;
  ...
  ...
  EndASM;
end;
```

Der Treiber übergibt an die UserDevice Funktion im Register _ACCA (R17) die aktuelle Row Nummer in der Form 0, 1, 2, 3, 4 etc. Im Register _ACCB (R16) wird als Alternative die Row Nummer als Bit Position übergeben in der Form $1, $2, $4, $8, $10, $20 etc. Je nach dem wie die externe Hardware aufgebaut ist kann nun _ACCA oder _ACCB verwendet werden, um eine Spalte (COL) einzulesen. Mit einem Multiplexer ist _ACCA vorteilhaft, mit einem Latch etc. ist _ACCB besser geeignet.

Die Applikation muss nun in _ACCA die gelesene Spalte an den Treiber zurückliefern.

Der Programm Code sollte unbedingt in Assembler geschrieben werden um hier keine Zeit zu verschwenden, denn diese Funktion wird aus dem SysTick heraus aufgerufen. Aus diesem Grund dürfen auch nur _ACCA und _ACCB zur Verarbeitung verwendet werden. Müssen zusätzliche Register benutzt werden, so müssen diese zuvor mit PUSH gerettet und später mit POP wieder zurückgeholt werden.

Lokale Variablen sind hier verboten, ebenso der Aufruf Systemtreiber oder Systemfunktionen. Auch keine mathematischen Operation des Systems sind erlaubt.

3.4 LCD-Display

3.4.1 LCDPORT

Import und Definition eines Ports für LCD. Es können 1-, 2- oder 4-zeilige Displays angeschlossen werden. Auch Versionen mit 2 Enables bzw. zwei Displays sind möglich. Es werden Controller vom Typ HD44780, HD66712 , KS0070 und KS0073 oder kompatible unterstützt.

Das LCD-Display wird im 4bit-Mode betrieben. Die unteren 4 Bits des gewählten Ports sind dabei die Datenbits D4..D7 des Displays, Port-Bit4 ist das Steuersignal *E,* Port-Bit5 ist das Signal *RS* und Port-Bit6 das Signal *RW.* Port-Bit7 ist das Enable2, ansonsten ist es frei.

Bei der Wahl des Ports ist darauf zu achten, dass dieses Port bi-direktional arbeiten muss.

```
Import  SysTick, LCDport;

Define  ProcClock   = 4000000;   {4Mhz clock }
        SysTick     = 10;        {10msec Tick}
        LCDtype     = 44780;     {66712,0070, 0073}
        LCDport     = PortA;     {Port Adresse}
        LCDRows     = 2;         {2-Zeiliges Display}
        LCDColumns  = 16;        {16-stelliges Display}
```

Achtung:
wird das LCDport importiert, sollte auch ein funktionierendes Display korrekt angeschlossen sein, ansonsten dauert jeder Zugriff sehr lange, da eine LCD Funktion erst nach einem Time-Out zurückkehrt.

Für Test und Debugzwecke kann das Abfragen des BUSY-Flags des Displays abgeschaltet werden mit: **{$LCDNOWAIT}** Dieser Schalter muss allerdings für das Programm im Chip deaktiviert werden, sonst läuft das Programm nicht, da der Display Controller überfahren wird.

LCDTYPE
Diese Definition kann entfallen, wenn der Displaytyp 44780 verwendet wird.
Der Typ 66712 bei “LCDxy” etwas anders adressiert und muss deshalb unbedingt angegeben werden, falls er eingesetzt wird.
Die Samsung Typen KS0070, KS0073 sind nicht absolut Timing kompatibel und müssen ebenfalls angegeben werden. Manche Display mit KS0073 können auch über SPI betrieben werden:

```
Define
    LCDtype      = 0073;                    {KS0073}
    LCDrows      = 4;                       {4 rows}
    LCDcolumns   = 20;                      {20 characters/line}
    LCDport      = SPI_Soft, PortB, 1, 2, 3, 0;   // PortX, SCK, MOSI, MISO, SS
    //  LCDport  = SPI;                    // Hardware SPI
    // LCDport   = SPI_C, PortB, 1;        // Hardware SPI, SS_port, SS_pin XMega
```

LCDROWS
Zeilenzahl des LCD-Displays (1, 2, 4). Wird zur Initialisierung benötigt. Bei grossen Displays mit zwei Enable Eingängen oder zwei separaten Modulen wird hier als zweiter Parameter eine 2 angegeben.

```
LCDRows = 4, 2;       {4-Zeiliges Display mit 2 Enables}
```

LCDCOLUMNS
Zeichenzahl des LCD-Displays (8, 12, 16, 20, 24, 40). Wird zur Adressierung (LCDxy) benötigt.

```
LCDColumns = 16;      {16-stelliges Display}
```

3.4.2 Funktionen und Prozeduren

LCDout
Schreiben ins LCD DD-Ram bzw. CG-Ram mit *RS=1* und *RW=0*.
Anzeige der Daten auf dem Display. Übergabe Parameter ist vom Typ Byte oder Char.

```
LCDout ('A');           {Char A anzeigen}
Write (LCDout, IntToStr(i));
```

LCDinp
Lesen des LCD DD-Ram bzw. CG-Ram mit *RS=1* und *RW=1*. Das Ergebnis ist vom Typ Byte.

```
c:= LCDinp;
```

LCDctrl
Schreiben auf LCD-ControlPort mit *RS=0* und *RW=0*.
Hiermit werden Steuerworte an das Display ausgegeben, wie z.B. *DispClear, ReturnHome, OnOff, CursorPos* etc.

```
LCDctrl ($01);          {Clear Display}
LCDctrl ($02);          {Cursor Home}
LCDctrl ($08);          {Display off}
LCDctrl ($41);          {CursorPos 1}
```

LCDstat
Lesen des LCD StatusPorts mit *RS=0* und *RW=1*.
Hiermit wird der Status des Displays abgefragt. Bit7 enthält das *BusyFlag* und Bit 0..6 die Cursor- bzw. DD-Ram Adresse. LCDstat ist in erster Linie dazu da um die aktuelle Cursor Adresse zu erfahren. Das *BusyFlag* braucht bei obigen LCD Operationen (LCDout, LCDinp, LCDstat) nicht abgefragt zu werden, da diese dies implizit selbst tun.

```
repeat until (LCDstat and $80) > 0;    {nicht notwendig}
if (LCDstat and $7f) > 15 then
  LCDctrl(2);                          {Cursor Home}
endif;
```

Weiterhin ist ein Time-out implementiert. Bei einem nicht vorhandenen oder defekten Display wird jeder Zugriff nach ca. 2msec abgebrochen. Die Funktion "LCDstat" gibt dabei im Resultat Bit7 gesetzt zurück, wenn das Busy-Bit des LCDports einen Pull-up besitzt.

LCDlower, LCDupper
Bei grossen, mehrzeiligen Displays ist oft ein zweites Enable Signal für den zweiten Controller auf dem Display zu finden. Ist ein solches Signal auf dem Display vorhanden, muss dies bei der Definition von **LCDrows** als zweiter Parameter mit angegeben werden. Das ansonsten freie Bit7 des Steuerports der CPU muss an das zweite Enable Signal des Displays angeschlossen werden. Werden zwei separate Displays verwendet, gilt das gleiche. Die beiden Module werden parallel geschaltet, ausgenommen die 2 Enables.

Der Compiler veröffentlicht jetzt zwei zusätzliche Prozeduren LCDupper und LCDlower. Ein Aufruf einer Prozedur bestimmt welches Enable Signal bei allen folgenden LCD-Zugriffen aktiviert wird. Nach LCDupper dann wird das Enablesignal1 (PortBit4) benutzt, nach LCDlower wird das Enablesignal2 (PortBit7) benutzt.

In der Praxis bedeuten zwei Enable Signale auch zwei komplett getrennte Controller auf dem Display. Das Display ist praktisch als zwei separate Einheiten auf einem Board zu betrachten!!

Alle Einstellungen, Positionierungen, Schreib- und Lesevorgänge etc. gelten nur für den Controller, der das Enable Signal bekommt.

Alternativ zu einem grossen Display mit 2 Enables können jedoch auch zwei separate kleinere Displays parallel angeschlossen werden. Diese müssen aber identisch sein.

```
Procedure LCDupper;
Procedure LCDlower;
```

LCDxy
Positionierung des Cursors auf Spalte[x] und Zeile[y]. Die Zählweise beider Werte beginnen mit “0”.

```
Procedure LCDxy (column, row : byte);
  LCDxy(0, 0);                  (* Cursor to line 0 and column 0 *)
  LCDxy(2, 1);                  (* Cursor to line 1 and column 2 *)
```

LCDcursor
Einstellung des Cursors Modus. Der Cursor kann komplett abgeschaltet, statisch ein oder blinkend geschaltet werden. Der erste Parameter schaltet den Cursor ein bzw. aus, der zweite Parameter schaltet das Blinken ein bzw. aus.

```
Procedure LCDcursor (on, blink : boolean);
  LCDcursor (false, false);     (* Cursor off *)
  LCDcursor (true, false);      (* Cursor on, no blink *)
  LCDcursor (true, true);       (* Cursor on, blink *)
```

LCDGetXY
liest die aktuelle Cursor Position
Im High-Byte des Ergebnis steht dann die Y-Position = Zeile und im Low-Byte die X-Position = Spalte

```
Function LCDgetXY : word;
```

LCDclr
Löscht das ganze Display und setzt den Cursor auf Position 0,0

```
Procedure LCDclr;               (* erase display, cursor to 0, 0 *)
```

LCDclrLine
Löscht die angegebene Zeile(0..n) und setzt den Cursor auf Zeile Anfang.

```
Procedure LCDclrLine (Line : byte);   (* erase line "Line", Cursor to "0,"Line" *)
```

LCDclrEOL
Löscht die aktuelle Zeile bis Zeilen Ende. Die Cursor Position bleibt unverändert.

```
Procedure LCDclrEOL;            (* erase current line to end *)
```

LCDHome
Setzt den Cursor auf Position 0,0

```
Procedure LCDhome;              (* cursor to 0, 0 *)
```

LCDoff
Schaltet das Display ab, verändert ansonsten nichts.

```
Procedure LCDoff;               (* switch display off *)
```

LCDon
Schaltet das Displays ein setzt den Cursor auf ON und BLINK

```
Procedure LCDon;                    (* switch display on *)
```

LCDCharSet
Die Prozedur lädt ein selbst definiertes Grafik-Zeichen an eine freie Stelle [0..15] im Character Generator Ram des Display Moduls. Das Zeichen kann mittels der zugehörigen Adresse dargestellt werden. Zur Definition des Sonderzeichens muss die gewünschte Adresse bzw. der ordinale Wert dieses Zeichen angegeben werden. Dann folgen 8 Bytes welche die 8 horizontalen Pixelreihen von oben nach unten darstellen. Da ein Zeichen nur 5 Pixel breit ist, sind auch nur die 5 niederwertigsten Bits relevant. Das Bit0 repräsentiert das ganz rechte Pixel, Bit4 das ganz linke.

```
Procedure LCDcharset (loc : char; b1, b2, b3, b4, b5, b6, b7, b8 : byte);
```

Beispiel für die Generierung und Darstellung des Zeichens “Grad” mit der Ordinalzahl 10.

```
LCDcharset (#10, $1C, $14, $1C, $00, $00, $00, $00, $00);

LCDout (#10);                       (* display user defined degree char *)
```

Der AVRco enthält ein spezielles Zeichengenerator Programm speziell für Sonderzeichen "LCDCharEd.exe". Hiermit können solche Zeichen graphisch erzeugt und der notwendige Code in der Zwischenablage abgelegt werden. "Shift/Insert" kopiert den Inhalt der Zwischenablage an eine beliebige Stelle der Source. Diese Funktion ist auch in der Symbolleiste der Entwicklungsumgebung verfügbar.

LCDcharSetP
Diese Prozedur ist eine mehr generellere Alternative zur obigen Version. Mit *srcArea* wird die Art der Source ausgewählt, 0=RAM, 1=EEPROM, 2=FLASH. Der Pointer muss auf eine entsprechende Struktur (array[0..7] of byte) in dem ausgewählten Speicher zeigen.

```
Procedure  LCDcharsetP (const loc : char; srcArea : byte; ptr : pointer);
```

{$LCDNOINIT}
Dieser Compilerschalter verhindert das automatische Initialisieren des LCD Controllers nach einem Programm Start. Das Anwendungsprogramm kann später entscheiden, ob und wann eine Initialisierung stattfinden soll, indem es die Procedure “LCDsetup” aufruft.

LCDsetup
Die Prozedur führt die Initialisierung des LCD-Controllers aus. Vorraussetzung ist allerdings, dass noch vor dem Define Block der Compilerschalter {$LCDNOINIT} aktiviert wird. Erst mit dem Aufruf von “LCDsetup” wird das Display auch zugreifbar.

Achtung:
Der Compilerschalter und diese Procedure gehören zusammen und müssen auch zusammen verwendet werden. Dies ist eine **Option** und nur für spezielle Fälle sinnvoll.

3.4.3 LCD-Split

Ist kein kompletter Port für alle Control und Daten Leitungen frei, bietet der Treiber auch die Möglichkeit, daß das Display auf 2 Ports verteilt werden kann. Dabei kann ein beliebiges Port die Control Lines steuern und ein anderes die 4 Datenleitungen.

Die Standard Definition ist diese:

LCDport = PortC; *// PortC 0..6(7) = D4, D5, D6, D7, E, RS, RW (, E2 opt.)*

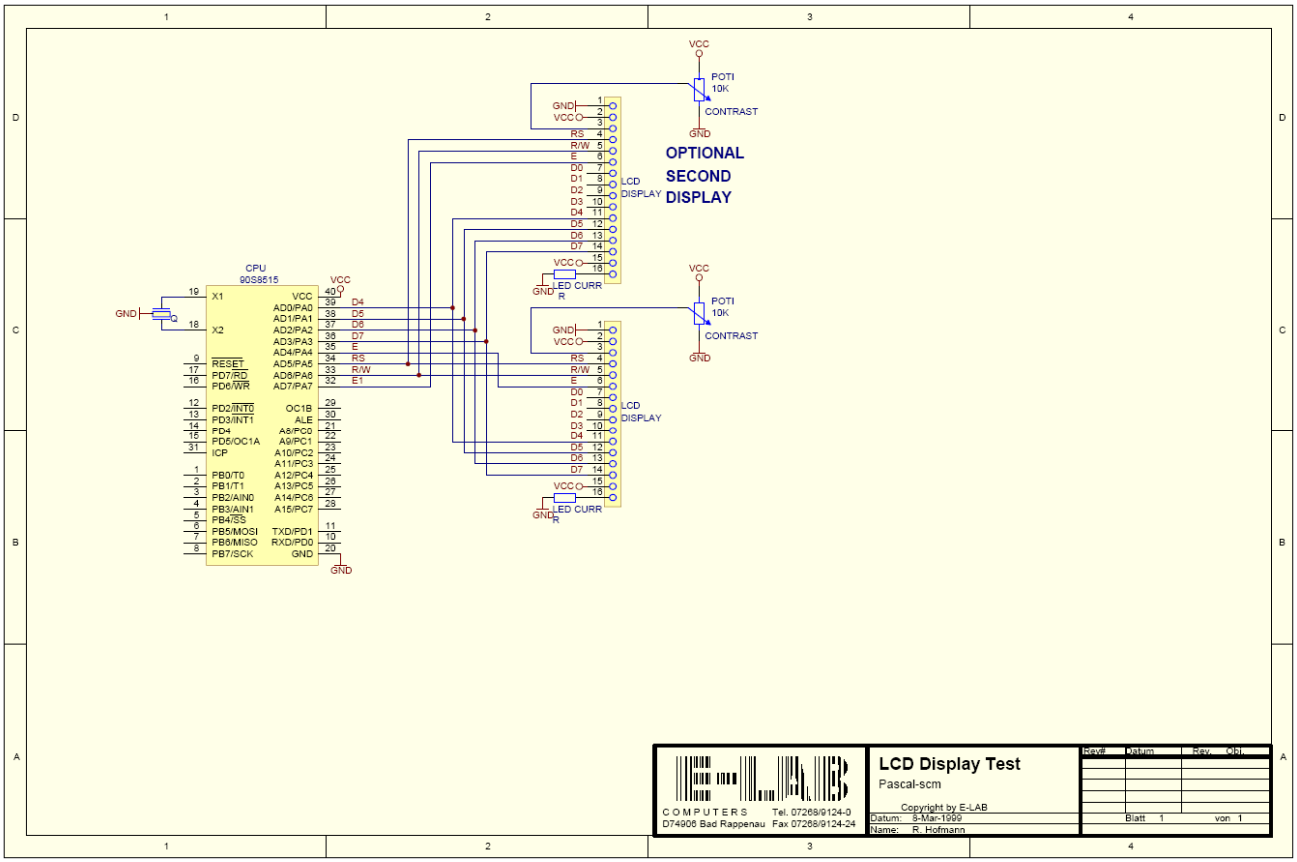
Beim Splitting muss das Define geändert werden:

Define *LCDport = PortC, 2, PortA, 3;* *// controlport, bit, Dataport, bit*

Dabei kommt zuerst das Controlport mit dem ersten benutzten Bit. Die Reihenfolge der Bits ist nicht veränderbar, jedoch **im Unterschied zum non-Split Treiber**: RS, RW, E (, Enable2 optional). Darauf folgt das Datenport mit dem ersten benutzten Bit. Die Reihenfolge liegt ebenfalls fest mit D4, D5, D6, D7.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\LCDsplit**



Schaltplan **LCD**

3.4.4 Benutzer definierte LCD Treiber (LCDUserPort)

Der Anwender kann alternativ auch einen eigenen Hardware Treiber definieren.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\LCD_IOS"**

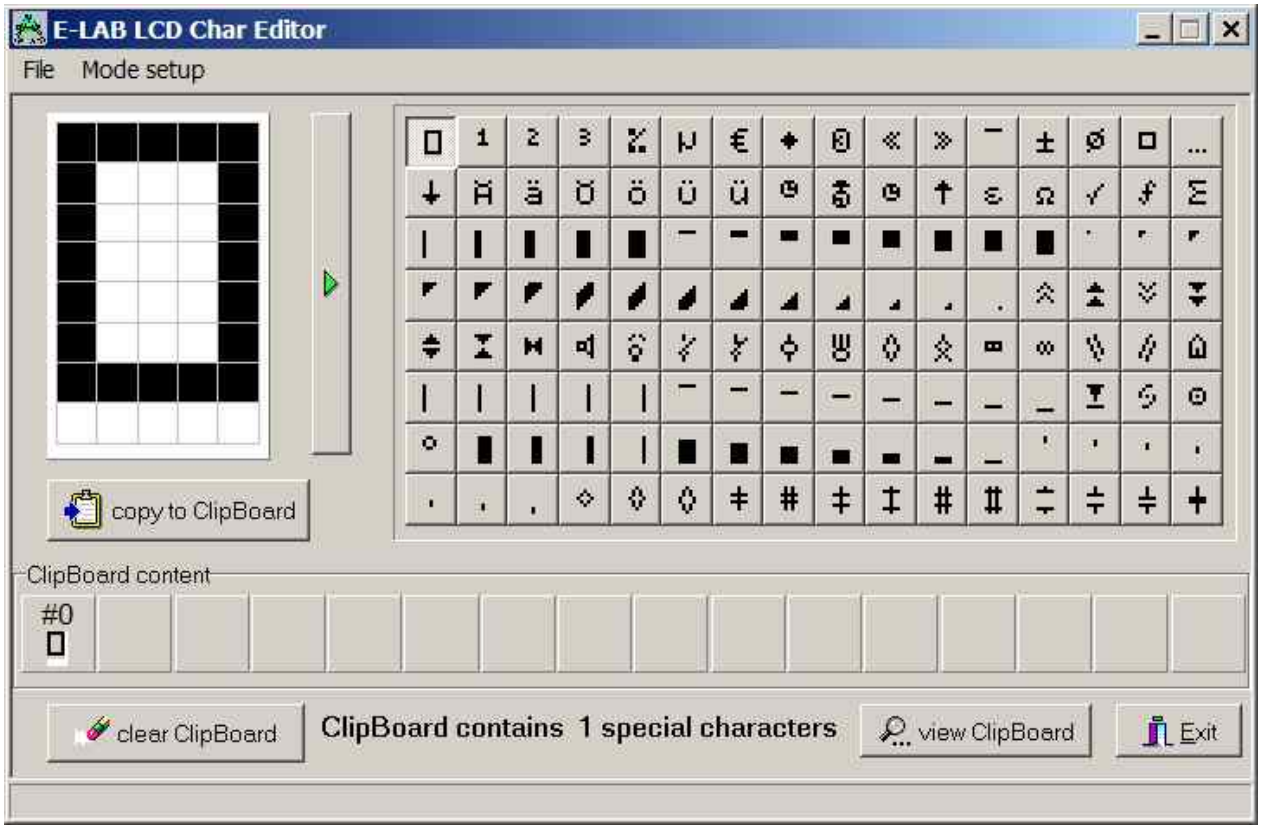
Wird das Display im 8-Bit Mode betrieben, kann die Initialisierung auch durch die System-Funktion "LCDSETUP" erfolgen. Diese initialisiert über "LCDIOS" das Display. Dazu muss der Compiler Schalter {$LCDNOINIT} aktiviert werden um die automatische Initialisierung abzuschalten.

3.4.5 Benutzer definierte LCD Zeichen

Der AVRco enthält ein spezielles Zeichengenerator Programm speziell für Sonderzeichen "LCDCharEd.exe". Hiermit können solche Zeichen graphisch erzeugt und der notwendige Code in der Zwischenablage abgelegt werden. "Shift/Insert" kopiert den Inhalt der Zwischenablage an eine beliebige Stelle der Source.

Diese Funktion ist auch in der Symbolleiste der Entwicklungsumgebung verfügbar.

Die erzeugten Sonderzeichen können mit der Procedure LCDcharset verarbeitet werden.
Die mit diesem Tool erzeugten Zeichensätze werden mit der extension ".lchr" in Files abgelegt und können damit auch wiederverwendet werden.



3.5 LCD BarGraph Treiber

Allgemeines
Die Darstellung von Werten auf einem Display (LCD, 7seg etc) erfolgt üblicherweise numerisch, d.h. in Zahlen. Damit lassen sich Messergebnisse etc. präzise darstellen. Die numerische Darstellung hat aber auch gravierende Nachteile. Die Zahl muss im Kopf des Betrachters umgesetzt und bewertet werden. Einen Trend feststellen, z.B. steigend oder fallend, fordert weitere Rechenleistungen beim Betrachter.

Eine graphische Darstellung ist sehr unpräzise aber der Betrachter hat nur ein Bruchteil der Analyse durchzuführen als bei der numerischen Darstellung. Das gilt insbesondere bei der Bewertung gross, klein, steigend oder fallend. Hier ist die graphische Darstellung weit überlegen.

Graphische Darstellungen werden meistens in der Form von Balken oder Torten Diagrammen realisiert. Bei einfachen Displays wie den LCDs ist natürlich nur die Balken Graphik (BarGraph) realistisch.

Einführung LCD BarGraph
Die vorliegende Implementation benutzt entweder den Treiber LCDport oder den Treiber LCDmultiPort. Dazu ist entweder der Treiber ***LCDport*** oder der Treiber ***LCDmultiPort*** zu importieren. Der Standard LCDport Treiber kann separat neben dem MultiPort Treiber existieren.

Es können bis zu 4 Bargraphen definiert werden, wobei jeder Graph ein eigenes „Koordinaten“ System besitzt:

Y-Position	Zeile
X-Start	Character
X-Len	Character
Scale	Byte

Jeder Graph muss per Define einem LCD Treiber zugewiesen werden. Es ist zum Beispiel möglich die Graphen 1..3 dem LCDmultiPort zuzuweisen und der Graph 4 dem LCDport.

Beim LCDmultiPort kann ein Graph zur Laufzeit auf jedes der 8 möglichen Displays gelegt werden.

Zur Darstellung des Graphen werden die ersten 5 der 8 möglichen benutzerdefinierbaren Zeichen (#0..#4) der Displays belegt. Jedes mit einem Bargraphen belegte Display muss deshalb ein spezielles BarGraph Init durchlaufen. Diese Sonderzeichen sollten dann nicht mehr durch das Userprogramm umdefiniert werden.

Der Treiber akzeptiert als Steuerwerte nur Bytes. Ein solches Byte wird anhand der Skalierungs und Koordinaten Faktoren umgerechnet und dann graphisch dargestellt.

Achtung:
Manche exotische LCD Controller (z.B. Samsung S6A0032) sind prinzipiell zwar kompatibel zum Standard Hitachi Controller 44780, bieten aber keine oder nur 2 frei definierbare Zeichen. Damit lässt sich natürlich kein Bargraph mit diesem Treiber realisieren. Hier muss die Applikation dies komplett selbst erledigen indem sie das Block Zeichen #255 ($FF) verwendet.

Imports
Wie beim AVRco System üblich, muss der Treiber definiert werden. Zusätzlich muss aber auch noch der gewünschte LCD Treiber importiert und definiert werden.

Entweder importiere den LCD Treiber **LCDmultiPort** und den I2C/TWI Treiber
Oder importiere den LCD Treiber **LCDport**
Der SysTick wird nicht benötigt.

```
Import    I2Cport, LCDmultiPort;
oder
Import    TWImaster, LCDmultiPort;
oder
Import    TWInet, LCDmultiPort;          // use Master mode
oder
Import    LCDPort;
```

Defines

Je nach gewünschtem I2C bzw. TWIport muss dieses definiert werden. Weiterhin muss, wie auch beim Standard LCDport, der verwendete LCD-Typ definiert werden.

Beispiel für *I2Cport*:

```
Define    ProcClock       = 8000000;    {8Mhz clock }
          I2Cport         = PortC;      {port used}
          I2Cdat          = 7;          {bit7-PortC}
          I2Cclk          = 6, 4;       {bit6-PortC, optional delay 4}
          LCDmultiPort    = I2C_Soft;   {use Software I2Cport}
          LCDTYPE_M       = 66712;      {LCD controller type}
          LCDrows_M       = 2;          {2 rows}
          LCDcolumns_M    = 20;         {20 chars per line}
```

Beispiel für *TWImaster*:

```
Define    ProcClock       = 8000000;    {8Mhz clock }
          TWIpresc        = TWI_BR100;  {100kBit/sec alt. TWI_BR400}
          LCDmultiPort    = I2C_TWI;    {use TWIport}
          LCDTYPE_M       = 44780;      {LCD controller type}
          LCDrows_M       = 4;          {4 rows}
          LCDcolumns_M    = 16;         {16 chars per line}
```

Beispiel für *TWInetMaster*:

```
Define    ProcClock       = 8000000;    {8Mhz clock }
          TWInode         = 05;         {default address in slave mode}
          TWIpresc        = TWI_BR400;  {400kBit/sec alt. TWI_BR100}
          TWIframe        = 4, iData;   {buffer/packet size}
          TWIframeBC      = 6;          {option broadcast buffer/packet size}
          TWInetMode      = Master;
          LCDmultiPort    = I2C_TWI;    {use TWIport}
          LCDTYPE_M       = 0070;       {LCD controller type}
          LCDrows_M       = 1;          {1 rows}
          LCDcolumns_M    = 12;         {12 chars per line}
```

Beispiel für *LCDport*:

```
Define    ProcClock       = 8000000;    {8Mhz clock }
          LCDport         = PortB;      {use Port B}
          LCDTYPE         = 0070;       {LCD controller type}
          LCDrows         = 2;          {2 rows}
          LCDcolumns      = 20;         {20 chars per line}
```

Nach dem kompletten Import und Define des LCD Treibers folgt jetzt noch das Define der LCD Bargraphen:

```
Define  ProcClock = 8000000;                {8Mhz clock }
...
...
LCDBargraph1 = LCDmultiPort;
LCDBargraph2 = LCDmultiPort;
LCDBargraph3 = LCDmultiPort;
LCDBargraph4 = LCDmultiPort;
// LCDBargraph4 = LCDPort;
```

LCDBargraph1..4
Importiert das Koordinaten System für einen BarGraphen und definiert den LCD Treiber der dazu benutzt werden muss. Dieser LCD Treiber muss ebenfalls importiert und definiert sein.

3.5.1 Funktionen

Die Definition eines Bargraphen veröffentlicht jeweils eine LCD abhängige allgemeine Init Funktion und jeweils zwei BarGraph Funktionen:

LCDbarInit_P
Init für den Standard ***LCDport***. Jedes angeschlossene Display muss einmal initialisiert werden. Dies geschieht im Treiber LCDport automatisch beim PowerOn bzw. CPU-Reset. Um die 5 Bargraph spezial Zeichen darstellen zu können, müssen diese in das Character RAM des LCD Controllers geladen werden. Dieses geschieht einmalig mit dieser Prozedur

```
Procedure LCDbarInit_P;
```

LCDbarInit_M
Init für den ***LCDmultiPort***. Jedes angeschlossene Display muss einmal initialisiert werden. Im Gegensatz zum Standard Treiber LCDport geschieht das nicht automatisch beim PowerOn bzw. CPU-Reset. Die Applikation muss hier das Display mit ***LCDsetup_M*** initialisieren. Erst dann erfolgt die BarGraph Initialisierung. Um die 5 Bargraph spezial Zeichen darstellen zu können, müssen diese in das Character RAM des LCD Controllers geladen werden. Dieses geschieht mit dieser Prozedur unmittelbar nach der Display Initialisierung, denn LCDbarInit_M hat keinen Display-num Parameter um gezielt ein Display anwählen zu können. Es wird immer das gerade aktivierte Display benutzt. Im Zweifelsfall zuvor mit der Procedure *LCDsetPort_M* das gewünschte Display anwählen

```
Procedure LCDbarInit_M;
```

LCDbarSet1..4
Bevor ein Bargraph benutzt werden kann, muss mit obiger Prozedur ***LCDbarInit*** einmal der BarGraph Zeichensatz geladen werden. Auch das Koordinaten System eines Graphen muss mindestens einmal mit LCDbarSet initialisiert werden. Die Parameter dieser Funktion bestimmen auf welcher Zeile und in welcher Grösse der Graph dargestellt wird.

```
Procedure   LCDbarSet 1 (const Line, PosA, Len, Scal : byte);
  LCDbarSet1 (0, 8, 8, 100);
```

Line bestimmt die Display Zeile 0..3
PosA bestimmt die X-Start Position, in Character
Len bestimmt die totale Graph Länge in Character
Scal bestimmt die Skalierung, d.h. den rechten Endwert des Graphen

Die Summe von PosA+Len darf nicht grösser sein als die Zeilen Länge (Spalten) des Displays.
Der Wert von Scal sollte sich im Bereich von 50..255 bewegen um eine gute Darstellung zu bekommen.

LCDbarOut1..4
Setzt den BarGraphen auf einen neuen Wert und zeigt ihn an. Der Übergabewert wird anhand des Koordinaten Systems dieses BarGraphen skaliert.
Beim LCDmultiPort wird immer das aktuelle Display beschrieben. Im Zweifelsfall zuvor mit der Procedure *LCDsetPort_M* das gewünschte Display anwählen.

```
Procedure   LCDbarOut1 (const b : byte);
  LCDbarOut1(50);
```

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\LCD_BarGraph**

3.6 LCDmultiPort Treiber für bis zu 8 LCDs

Allgemeines
Üblicherweise werden alpha-numerische LC-Displays an ein Port der CPU direkt angeschlossen und im 4- oder 8bit Mode betrieben. Bei zwei oder mehr Displays reichen dann ganz schnell die verfügbaren Port Leitungen nicht mehr aus. Zusätzlich wird die Mechanik durch die mindestens 10 benötigten Leitungen pro Display nicht mehr handhabbar.

Die Lösung des Problems ist hier der Einsatz von LCDs mit einem Bus-System, in der Regel I2C. Diese Displays sind allerdings recht teuer und die Auswahl ist nicht sehr gross.

Wenn man jetzt auf intelligente Displays mit internem I2C Bus verzichtet und „einfach“ die Anzahl der vorhandenen CPU Ports mittels I2C I/O-Expander Chips erweitert, kann an jedes solches Chip ein LCD angeschlossen werden. Dazu werden nur 2 Portleitungen der CPU benötigt, unabhängig von der Zahl der angeschlossenen Displays.

Einführung LCDmultiPort
Die vorliegende Implementation benutzt entweder den Software I2C-Treiber (I2Cport) oder den internen TWI (I2C) Port der AVR mega CPUs. Dazu ist entweder der Treiber *I2Cport* oder der Treiber *TWImaster* oder der Treiber *TWInet* im Mastermode zu importieren. Der Standard LCDport Treiber kann separat neben dem MultiPort Treiber existieren.

Bei den XMegas muss statt des TWI einer der vorhandenen TWIs angegeben werden: TWI_C, TWI_D, TWI_E oder TWI_F.

Als I2C Port-Expander muss pro LCD der Typ PCA9555 von Philips verwendet werden. Dieser Baustein kann bis zu 8 mal am Bus vorhanden sein. Die Verdrahtung zwischen PCA9555 und LCD ist durch den Treiber fest vorgegeben und dem folgenden Schaltplan entnommen werden. Der PCA9555 kann mit bis zu 400kBit/sec am I2C Bus betrieben werden. Im Gegensatz zu seinen Vorgänger Typen kann jedes der beiden 8bit Ports gezielt gelesen, geschrieben und umprogrammiert werden.

Der Treiber LCDmultiPort steuert seine remote Ports PCA9555 so, als wenn die LCDs direkt im 8bit Mode an einem Port der CPU angeschlossen wären.

Die Treiber Funktionen entsprechen den Funktionen des Treibers *LCDport* mit der Erweiterung, dass für jede Funktion auch noch die Display Nummer mit angegeben werden muss (Ausnahme: LCDout).

3.6.1 Technische Daten

I2C Port	Software I2C importiert durch I2Cport	
oder	CPU-TWI importiert durch TWImaster	// Mega
oder	CPU-TWI importiert durch TWInet im Mastermode	// Mega
oder	CPU-TWI importiert durch TWI_C, TWI_D, TWI_E, TWI_F	// XMega
Displays	bis zu 8 Stück, Controller 44780, 66712 oder KS0070 8bit Modus, nur ein Enable Signal möglich. Alle Displays müssen vom gleichen Typ sein, Controller, Anzahl Zeilen und Spalten.	
Hardware	I2C I/O-Expander Chip PCA9555 von Philips, 1 Stück pro Display	
I2C Adressen	Die PCA9555 liegen auf den Bus-Adressen $20..$27	

wobei LCD1 die Adresse $20 hat, LCD2 hat $21 etc.
Die PCA9555 haben drei Adresspins bzw. Bits die jeweils entsprechend beschaltet werden müssen.

Imports

Wie beim AVRco System üblich, muss der Treiber importiert und definiert werden. Zusätzlich muss aber auch noch der gewünschte I2C/TWI Treiber importiert werden.
Der SysTick wird nicht benötigt. Bei einem nicht vorhandenen Display kehren alle Funktionen nach einem Time-Out zurück.

```
Import  I2Cport, LCDmultiPort;
oder
Import  TWImaster, LCDmultiPort;
oder
Import  TWInet, LCDmultiPort;          // use Master mode
```

XMega

```
Import  TW_C, TWI_E, LCDmultiPort;
```

Defines

Je nach gewünschtem I2C bzw. TWIport muss dieses definiert werden. Weiterhin muss, wie auch beim Standard LCDport, der verwendete LCD-Typ definiert werden.

Beispiel für *I2Cport*:

```
Define  ProcClock       = 8000000;      {8Mhz clock }
        I2Cport         = PortC;        {port used}
        I2Cdat          = 7;            {bit7-PortC}
        I2Cclk          = 6, 4;         {bit6-PortC, optional delay 4}
        LCDmultiPort    = I2C_Soft;     {use Software I2Cport}
        LCDTYPE_M       = 66712;        {LCD controller type}
        LCDrows_M       = 2;            {2 rows}
        LCDcolumns_M    = 20;           {20 chars per line}
```

Beispiel für *TWImaster*:

```
Define  ProcClock       = 8000000;      {8Mhz clock }
        TWIpresc        = TWI_BR100;    {100kBit/sec alt. TWI_BR400}
        LCDmultiPort    = I2C_TWI;      {use TWIport}
        LCDTYPE_M       = 44780;        {LCD controller type}
        LCDrows_M       = 4;            {4 rows}
        LCDcolumns_M    = 16;           {16 chars per line}
```

Beispiel für *TWInetMaster*:

```
Define  ProcClock       = 8000000;      {8Mhz clock }
        TWInode         = 05;           {default address in slave mode}
        TWIpresc        = TWI_BR400;    {400kBit/sec alt. TWI_BR100}
        TWIframe        = 4, iData;     {buffer/packet size}
        TWIframeBC      = 6;            {option broadcast buffer/packet size}
        TWInetMode      = Master;
        LCDmultiPort    = I2C_TWI;      {use TWIport}
        LCDTYPE_M       = 0070;         {LCD controller type}
        LCDrows_M       = 1;            {1 rows}
        LCDcolumns_M    = 12;           {12 chars per line}
```

Beispiel für *XMega*:

```
Import   TWI_C;
Define  OSCtype         = int32MHz, PLLmul=4, prescB=1, prescC=1;
        TWIprescC       = TWI_BR100;    {100kBit/sec alt. TWI_BR400}
        LCDmultiPort    = TWI_C;  {use TWIport}
        LCDTYPE_M       = 44780;        {LCD controller type}
        LCDrows_M       = 4;            {4 rows}
        LCDcolumns_M    = 16;           {16 chars per line}
```

LCDmultiPort
Definiert das zu verwendende I2C-Port., entweder SoftWare-I2C mit *I2C_Soft* oder onchip TWIport mit *I2C_TWI*. Der jeweilige Treiber dazu muss importiert und definiert werden.

LCDtype_M
Definiert den LCD-Controller Typ HD44780, HD66712, KS0070 oder KS0073

LCDrows_M
Definiert die Zeilenzahl der Displays 1, 2 oder 4

LCDcolumns_M
Definiert Spalten (Zeichen/Zeile) der Displays 8, 12, 16, 20, 24 oder 40.

3.6.2 Typen und Funktionen

Der Import von LCDmultiPort veröffentlicht einen Aufzählungstyp (Enumeration), der für den Aufruf der Treiber Funktionen benutzt werden muss:

```
Type TLCD_num = (LCD_m1, LCD_m2, LCD_m3, LCD_m4, LCD_m5, LCD_m6, LCD_m7, LCD_m8);
```

LCDsetup_M
Jedes angeschlossene Display muss einmal initialisiert werden. Im Gegensatz zum Standard Treiber *LCDport* geschieht das nicht automatisch beim PowerOn bzw. CPU-Reset. Die Initialisierung belässt das Display im ausgeschalteten Zustand. Die Einschaltung erfolgt mit der Funktion *LCDcursor_M*. LCDsetup_M kehrt bei erfolgreicher Initialisierung mit einem true zurück, ansonsten mit einem false.

```
Function LCDsetup_M (const LCD_num : TLCD_num) : boolean;
  If not LCDsetup_M(LCD_m1) then ...
    Error …
  Endif;
```

LCDon_M
Schaltet das Displays ein setzt den Cursor auf ON und BLINK

```
Procedure LCDon_M (const LCD_num : TLCD_num);
  LCDon_M (LCD_m6);
```

LCDcursor_M
Einschalten des Displays und Einstellung des Cursors Modus. Der Cursor kann komplett abgeschaltet, statisch ein oder blinkend geschaltet werden. Der erste Parameter schaltet den Cursor ein bzw. aus, der zweite Parameter schaltet das Blinken ein bzw. aus.

```
Procedure LCDcursor_M (const LCD_num : TLCD_num; on, blink : boolean);
  LCDcursor_M (LCD_m2, false, false);       (* Cursor off *)
  LCDcursor_M (LCD_m3, true, false);        (* Cursor on, no blink *)
  LCDcursor_M (LCD_m4, true, true);         (* Cursor on, blink *)
```

LCDOut_M
Schreiben ins LCD DD-Ram bzw. CG-Ram und Anzeigen der Daten auf dem Display. Übergabe Parameter ist vom Typ Char. Da diese Prozedur auch mit ***Write*** funktionieren muss, gibt es hier den sonst obligatorischen Parameter *LCD_num* nicht. Das Schreiben auf das Display erfolgt daher immer auf dasjenige, auf welches zuletzt mit einer Funktion zugegriffen wurde. Das gewünschte Display kann aber auch explizit mit ***LCDsetPort_M*** vorgegeben werden.

```
Procedure LCDout_M (const c : char);
  LCDout_M ('A');                   {show char A}
  Write (LCDout_M, IntToStr (i));   {string output}
```

LCDclr_M
Löscht das ganze Display und setzt den Cursor auf Position 0,0

```
Procedure LCDclr_M (const LCD_num : TLCD_num);
```

```
LCDclr_M (LCD_m5);
```

LCDclrEOL_M
Löscht die aktuelle Zeile ab Cursor Position bis Zeilen Ende. Die Cursor Position bleibt unverändert.

```
Procedure LCDclrEOL_M (const LCD_num : TLCD_num);
  LCDclrEOL_M (LCD_m6);
```

LCDclrLine_M
Löscht die angegebene Zeile(0..n) und setzt den Cursor auf Zeilen Anfang

```
Procedure LCDclrLine_M (const LCD_num : TLCD_num; const line : byte);
  LCDclrLine_M (LCD_m6, 1);
```

LCDhome_M
Setzt den Cursor auf Position 0,0

```
Procedure LCDhome_M (const LCD_num : TLCD_num);
  LCDhome_M (LCD_m7);
```

LCDxy_M
Positionierung des Cursors auf Spalte[x] und Zeile[y]. Die Zählweise beider Werte beginnen mit “0”.

```
Procedure LCDxy_M (const LCD_num : TLCD_num; x, y : byte);
  LCDxy_M (LCD_m8, 0, 0);          (* Cursor to line 0 and column 0 *)
  LCDxy_M (LCD_m1, 2, 1);          (* Cursor to line 1 and column 2 *)
```

LCDgetXY_M
Position des Cursors auslesen. Die Zeile (y) ist im high-byte und die Spalte (x) im low-byte des Resultats.

```
Function LCDgetXY_M (const LCD_num : TLCD_num) : word;
```

LCDoff_M
Schaltet das Display ab, verändert ansonsten nichts.

```
Procedure LCDoff_M (const LCD_num : TLCD_num);
  LCDoff_M (LCD_m2);
```

LCDsetPort_M
Optionales Umschalten des Treibers auf ein bestimmtes LCD. Nur sinnvoll im Zusammenhang mit *LCDout_M*.

```
Procedure LCDsetPort_M (const LCD_num : TLCD_num);
  LCDsetPort _M (LCD_m2);
```

LCDgetPort_M
Die Funktion liefert als Ergebnis das aktuell eingestellt LCD-Port.

```
Function LCDgetPort_M : TLCD_num;
```

LCDctrl_M
Optionale Low-Level Funktion. Schreiben auf LCD-ControlPort mit *RS=0* und *RW=0*.
Hiermit werden Steuerworte an das Display ausgegeben, wie z.B. *DispClear, ReturnHome, OnOff, CursorPos* etc.

```
Procedure LCDctrl_M (const LCD_num : TLCD_num; const b : byte);
  LCDctrl_M (LCD_m1, $01);         {Clear Display}
  LCDctrl_M (LCD_m2, $02);         {Cursor Home}
  LCDctrl_M (LCD_m3, $08);         {Display off}
  LCDctrl_M (LCD_m4, $41);         {CursorPos 1}
```

LCDinp_M

Optionale Low-Level Funktion. Lesen des LCD DD-Ram bzw. CG-Ram mit *RS=1* und *RW=1*. Das Ergebnis ist vom Typ Byte.

```
Function LCDINP_M (const LCD_num : TLCD_num) : byte;
  b:= LCDinp_M (LCD_m5);
```

LCDstat_M
Optionale Low-Level Funktion. Lesen des LCD Status Ports mit *RS=0* und *RW=1*.
Hiermit wird der Status des Displays abgefragt. Bit7 enthält das *BusyFlag* und Bit 0..6 die Cursor- bzw. DD-Ram Adresse. LCDstat ist in erster Linie dazu da um die aktuelle Cursor Adresse zu erfahren. Das *BusyFlag* braucht bei den LCD Operationen (LCDout_M, LCDinp_M, LCDstat_M) nicht abgefragt zu werden, da diese dies implizit selbst tun.

```
Function LCDstat_M (const LCD_num : TLCD_num) : byte;
  repeat until (LCDstat_M(LCD_m1) and $80) > 0;          {not necessary}
  if (LCDstat_M (LCD_m1) and $7f) > 15 then
    LCDctrl_M (LCD_m1, 2);                               {Cursor Home}
  endif;
```

LCDcharset_M
Die Prozedur lädt ein selbst definiertes Grafik-Zeichen an eine freie Stelle [0..15] im Character Generator Ram des Display Moduls. Das Zeichen kann mittels der zugehörigen Adresse dargestellt werden. Zur Definition des Sonderzeichens muss die gewünschte Adresse bzw. der ordinale Wert dieses Zeichen angegeben werden. Dann folgen 8 Bytes welche die 8 horizontalen Pixelreihen von oben nach unten darstellen. Da ein Zeichen nur 5 Pixel breit ist, sind auch nur die 5 niederwertigsten Bits relevant. Das Bit0 repräsentiert das ganz rechte Pixel, Bit4 das ganz linke.

```
Procedure LCDcharset_M (LCD_num : TLCD_num;  loc : char; c1, c2, c3, c4, c5, c6, c7, c8 : byte);

  LCDcharset_M (LCD_m5, #10, $1C, $14, $1C, $00, $00, $00, $00, $00);

  LCDout_M (LCD_m5, #10);                    (* display user defined degree char *)
```

Bemerkung:
viele LCD Controller unterstützen zwar 16 private Zeichen, können aber nur 8 darstellen!

LCDcharset_MP
Diese Prozedur ist eine mehr generellere Alternative zur obigen Version. Mit *srcArea* wird die Art der Source ausgewählt, 0=RAM, 1=EEPROM, 2=FLASH. Der Pointer muss auf eine entsprechende Struktur (array[0..7] of byte) in dem ausgewählten Speicher zeigen.

```
Procedure LCDcharset_MP (LCD_num : TLCD_num; loc : char; srcArea : byte; ptr : pointer);
```

3.6.3 Nicht benutzte PortPins des LCD Control Ports

Die oberen 5 Bits des LCD Control Ports werden vom Treiber nicht benutzt. Diese Pins sind immer auf Input geschaltet und können von der Applikation gelesen werden.

LCDportInp_M
Diese Funktion liest die oberen 5 Input Pins des Control Ports. Die unteren 3 Bits (E, RW, RS) werden immer als 0 zurückgegeben. Der Parameter *LCD_num* wird nicht gespeichert und beeinflusst die anderen LCD Operationen daher nicht.

```
Function LCDportInp_M (const LCD_num : TLCD_num) : byte;
```

3.6.4 Multi-Processing und TWI Port –I2C

In einer Applikation mit Prozessen und/oder Tasks kommt es häufig vor, dass der TWI-Bus nicht nur als Netzwerk sondern auch für andere Zwecke (LCD, Ports etc) gebraucht wird. Wenn dabei die Zugriffe auf das TWI aus unterschiedlichen Prozessen heraus erfolgen, kommt es unweigerlich zu Konflikten da solche sequentiellen Treiber (I2C, TWI, UART etc) nicht re-entrant sind, d.h. sie sind nicht unterbrechbar und neu aufrufbar. Deshalb besitzt das TWI Port ganz allgemein eine Semaphore vom Typ DeviceLock.

```
TWI_DevLock : DEVICELOCK;
TWI_DevLockTN : DEVICELOCK;   // XMega
```

Der TWI Treiber beachtet und steuert diese Semaphore. Beim Eintritt in den Treiber wird geprüft ob der Treiber frei ist (Semaphore inaktiv). Ist das der Fall, dann wird die Semaphore aktiviert = Treiber gesperrt und der Auftrag wird ausgeführt. Nach dem Beenden des Auftrags wird die Semaphore wieder freigegeben.

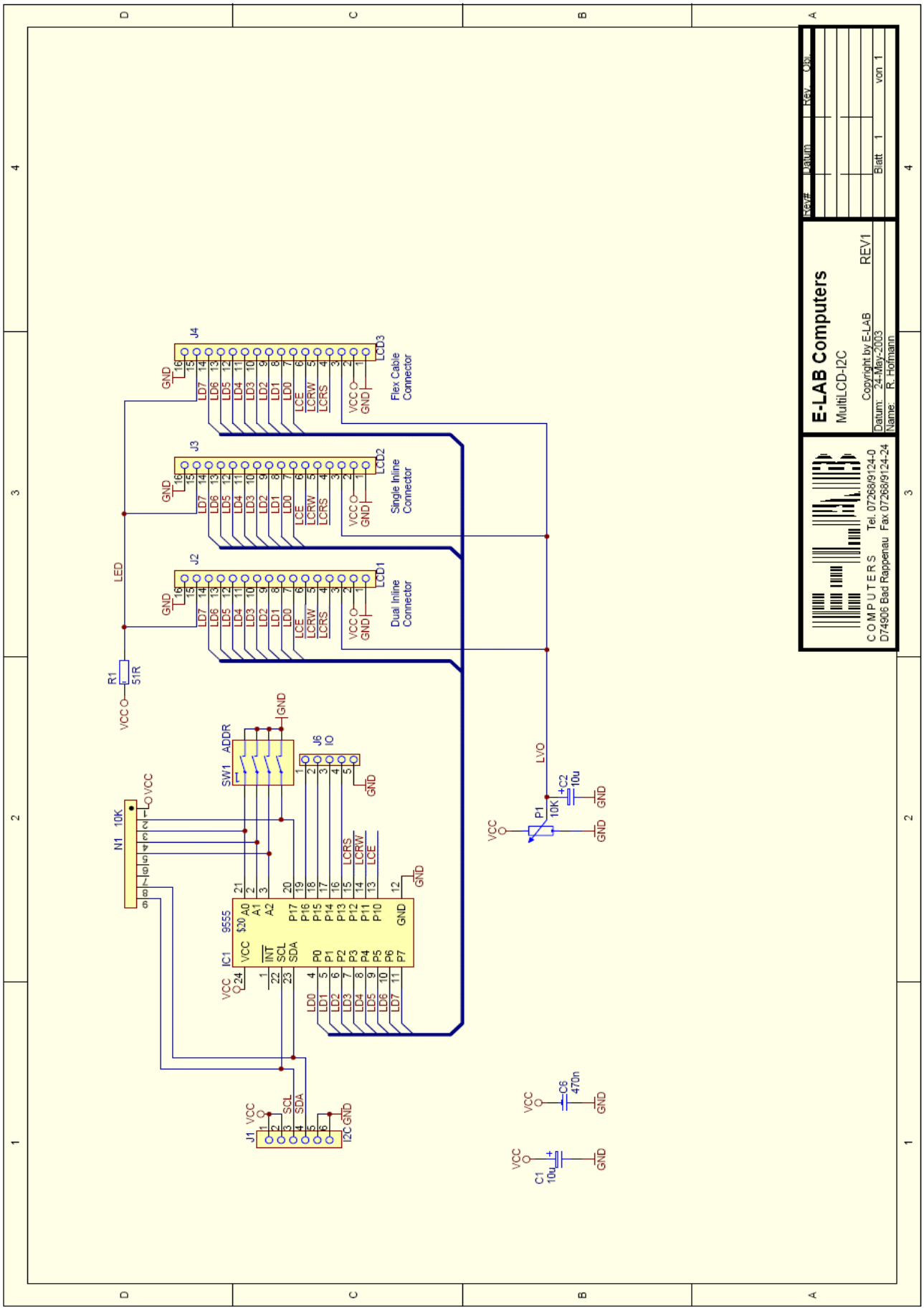
Wird beim Eintritt festgestellt, dass der Treiber belegt ist (Semaphore aktiv) dann wird ein **Schedule** durchgeführt und dieser Prozess reiht sich wieder in die Warteschlange ein. Bei einem der nächsten Prozess Wechsel wird der Prozess wieder gestartet und prüft jetzt erneut die Semaphore. Das wiederholt sich jetzt solange, bis die Semaphore wieder frei ist.

Achtung:
Weil ein TWI-Aufruf durch „Schedule“ abgebrochen werden kann, sollten für den TWI Zugriff **keine Tasks** verwendet werden, da Tasks damit komplett abgebrochen werden und der Auftrag in diesem Fall überhaupt nicht ausgeführt wird. Mann kann sich jedoch ein Flag setzen, wenn der Task mit seinem Auftrag erfolgreich war und wenn nicht, muss der Task wiederholt werden. Das bringt jedoch eine etwas komplizierte Verwaltung mit sich.

Weiterhin muss klar sein, dass dieses Verfahren nur auf Byte Ebene eine Sperrung behandelt. Werden z.B. ganze Strings zum LCD geschickt, muss auch weiterhin die Applikation dafür sorgen, dass der String Transfer nicht durch einen anderen Prozess unterbrochen wird, der auch LCD Zugriffe machen will.

Programm Beispiel und Schaltung:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\LCDmulti**
ein Xmega Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Xmega_LCDmulti**



Schaltplan **LCDmulti**

3.7 LED 7seg Display

3.7.1 Disp7sPort, Mux|NonMux

Import und Definition eines Ports für ein 7segment LED Display mit bis zu 8 Stellen/Digits.

Multiplexed:
Das Display wird im Common-Cathode Multiplex Modus betrieben. Dafür werden 5 Bits eines Output-Ports benötigt sowie zwei serielle Treiberbausteine, einen für die Segmente, z.B. UCN5895, und einen für die Digits, z.B. UCN5841.

Die On-Zeit eines Digits beträgt einen SysTick. Damit ist die Zeit für einen kompletten Refresh = Digits x SysTick (msec). Bei einem SysTick von 10msec und 4 Digits ergibt sich somit eine Refreshrate von 4 x 10msec = 40msec = 25Hz.

NonMultiplexed:
Im nonMultiplexed Mode werden nur Segment Schieberegister verwendet. Pro Digit wird ein Schieberegister und 8 Widerstände benötigt. Der Nachteil des höheren Bauteile Aufwands wird durch abstrahlungsfreien Betrieb (keine EMV Probleme) aufgehoben.

Das Bit0 des Ports wird an den jeweiligen CLOCK-Eingang und das Bit1 an den entspr. Dateneingang der beiden Treiber angeschlossen. Bit2 ist das Latch-Signal für den Digit-Treiber und das Bit3 ist das Latch-Signal für den Segment-Treiber. Bit4 ist das Output-Enable-Signal (Blank) für den Digit-Treiber (Mux).

```
Import   SysTick, Disp7sPort;

Define   ProcClock     = 4000000;              {4Mhz clock }
         SysTick       = 10;                   {10msec Tick}
         Disp7sPort    = PortA, Mux;           {Port Adresse, multiplexed}
         //Disp7sPort  = PortA, NonMux;        {Port Adresse, nicht multiplexed}
         //Disp7sPort  = PortA, Mux, startpin; {Port Adresse, multiplexed mit Startpin}
         //Disp7sPort  = UserPort;             {User defined IO-driver, multiplexed}
         DispMode      = ShiftLeft;            {links durchschieben}
         DispDigits    = 4, iData;             {4-stelliges Display}
```

Disp7Port mit Startpin
die benötigten 3..5 Bits eines Ports brauchen nicht unbedingt von Bit0 an zu beginnen.
Voraussetzung ist, dass alle benötigten Bits ab einem definierten "Startpin" aufwärts in den Port passen. Bei keiner Angabe wird mit Bit0 des Ports begonnen.
Ist ***UserPort*** angegeben, arbeitet der Treiber immer multiplexed und die UserDevice Funktion ***Disp7sIOS*** muss von der Applikation bereitgestellt werden.

DispMode
Arbeitsweise des Displays beim Anfügen eines neuen Zeichens. Mit **Wrap** wird der Einfüge Cursor nach erreichen des Display Endes auf die Stelle 0 gesetzt (ganz links). Mit **ShiftLeft** bleibt der Cursor ganz rechts stehen, das Zeichen wird hier eingefügt und der Rest des Inhalts wird um eine Position nach links verschoben. Mit **ShiftRight** ist dieser Vorgang genau umgekehrt.

Als weitere Option kann bei der Multiplex Ausführung das Display Blank unterdrückt werden. Das Blank Signal ist bei doppelt gepufferten Schieberegistern nicht notwendig, da es hier zu keinen “Geister Bildern” kommen kann. Der Blank-Pin wird nicht benutzt und kann anderweitig verwendet werden.

```
Define   DispMode = ShiftLeft, noBlank;      {links durchschieben}
```

DispDigits
Stellenzahl des Displays (1..8) und Lage des Zeichenbuffers. Wird zur Initialisierung benötigt.

3.7.2 Funktionen und Prozeduren

DispOut
Schreiben in den Display Buffer. Übergabe Parameter ist vom Typ Byte oder Char. Das in der aktuellen Stelle des Displays befindliche Zeichen wird mit dem neuen überschrieben. Die Schnittstelle kann auch im Zusammenhang mit **Write** eingesetzt werden, wie unter LCD-Display und “Write” beschrieben.

Da ein 7seg-Display nicht alle Zeichen des Alphabets darstellen kann, werden einige Zeichen als Leerzeichen dargestellt. Steuerzeichen werden bis auf ‘CR’ und ‘LF’ ignoriert.

Linefeed (LF = $0A) löscht das Display und setzt den Schreibcursor an die Stelle ‘0’.
Return (CR = $0D) setzt den Schreibcursor an die Stelle ‘0’ verändert jedoch das Display nicht.
Ist der Schreibcursor an der letzten Stelle angekommen, wird er automatisch auf ‘0’ gesetzt (**Wrap**) Mode.
Bei den **Shift** Modi wird der Display Inhalt um eine Stelle nach rechts bzw. links geschoben und das neue Zeichen angefügt.

```
DispOut ('A');                  {Char A anzeigen}
Write (DispOut, #10 + 'Hallo');
```

DispBlink
Das ganze Display blinkt bzw. das Blinken wird ausgeschaltet. Der Übergabe Parameter ist vom Typ Boolean.

```
DispBlink (true);               {Blinken ein}
```

DispDigBlink
Ein einzelnes Digit blinkt bzw. das Blinken wird ausgeschaltet. Der Übergabe Parameter ist vom Typ Byte und bezeichnet die Stelle bzw. das Digit. Ist das Digit am blinken, so wird jetzt das Blinken ausgeschaltet und umgekehrt

```
DispDigBlink (1);               {Blinken ein/aus}
```

DispPos
Der Schreibcursor wird an eine bestimmte Stelle positioniert. Der Übergabe Parameter ist vom Typ Byte und bezeichnet die Stelle bzw. das Digit. Die Zählreihenfolge der Digits ist von ‘0’ (ganz links) bis ‘Disp7sPort’ -1 (ganz rechts);

```
DispPos (0);                    {Cursor Home}
```

DispClear
Das ganze Display wird gelöscht Der Schreibcursor wird an die Stelle ‘0’ positioniert. Alle aktuelle Blinkfunktionen werden zurückgesetzt.

```
DispClear;                      {Löschen + Cursor Home}
```

Disp7Test
Testfunktion. Schaltet alle Segmente ein

```
Procedure Disp7Test;
```

DispClEol
Das Display wird von der aktuellen Cursorposition bis Zeilenende gelöscht Die Position des Schreibcursors bleibt erhalten.

```
DispClEol;                      {Löschen bis Zeilen Ende}
```

Disp7Buff
Müssen Sonderzeichen dargestellt werden, so kann direkt auf den Refreshbuffer zugegriffen werden und dieses Zeichen abgelegt werden indem dieser als array[0..DispDigits-1] of byte adressiert wird.

```
Disp7Buff[3] := $63;          {Grad-Zeichen darstellen}
```

DispRefresh
Wird direkt in den Disp7Buff geschrieben, hat das im NonMux Mode keine Auswirkung, da die externen Latches nicht upgedatet werden. Nach einem abgeschlossenen Schreibvorgang auf den Buffer sollte daher diese Prozedur aufgerufen werden, um die Änderung auch sichtbar zu machen.

```
DispRefresh;                  {physisches Update der Treiber}
```

_Disp7sTab
Falls spezielle Zeichen oder gar ein anderes Alphabet benötigt werden, kann der interne Zeichensatz überschrieben werden. Dazu muss ein array[0..60] of byte angelegt werden. Der erste Eintrag ist das Leerzeichen ($20). Das System subtrahiert daher $20 von jedem mit DispOut übergebenen Zeichen.
Es bestehen 2 Möglichkeiten diesen Zeichensatz zu implementieren.

Erzeugen eines Arrays mit Konstanten durch direkte Eingabe:

```
_Disp7sTab : array[0..60] of byte = (0, 23, 53, 12, .., .., );
```

Erzeugen eines Arrays mit Konstanten durch einlesen einer Datei:

```
_Disp7sTab : array[0..60] of byte = 'Disp7table.bin';
```

Hier muss die Datei alle 60 möglichen Zeichen in binärer Form enthalten

UserPort
Wurde über das define der UserPort ausgewählt, so muss die Applikation diesen Device Treiber bereitstellen:

```
UserDevice Disp7sIOS(digit, segment : byte);
begin
  ...
end;
```

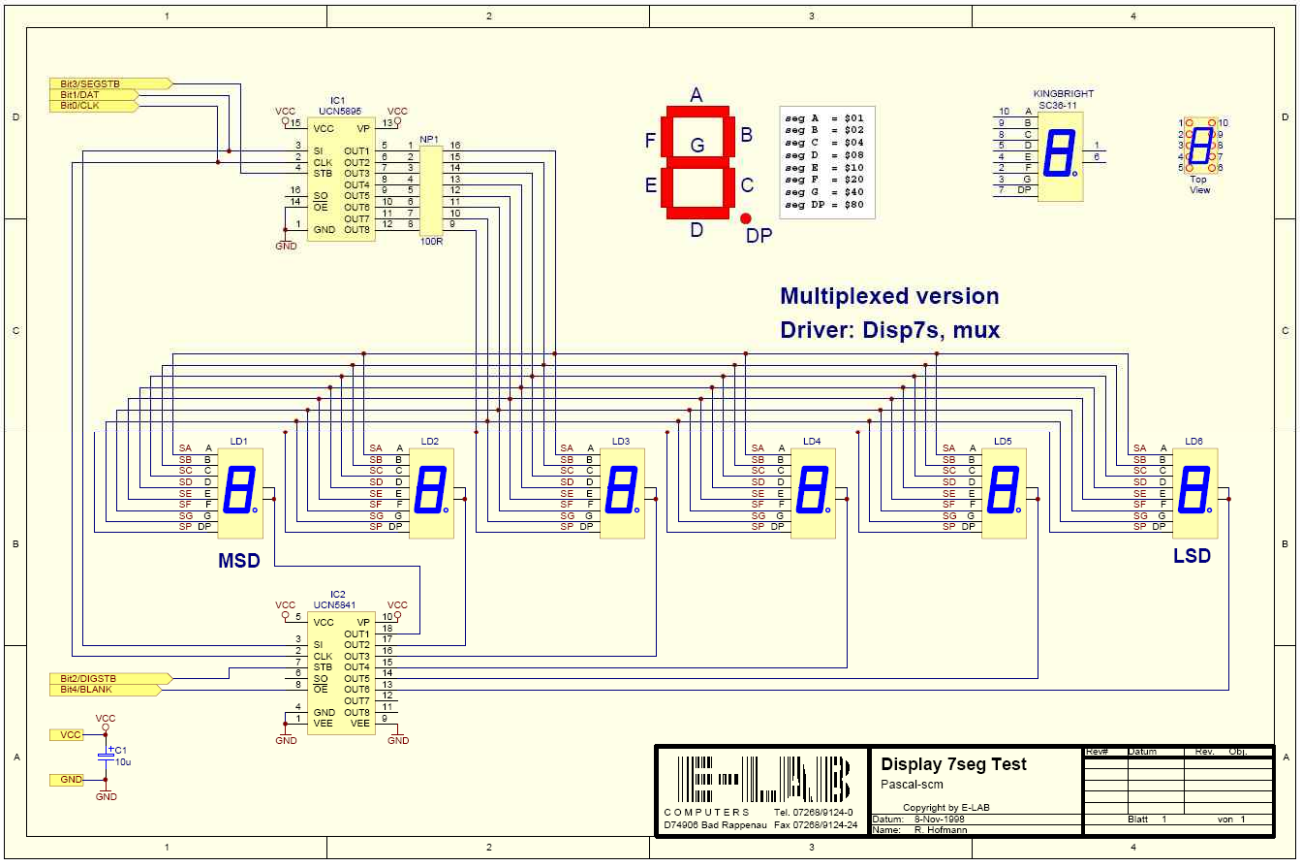
Der Parameter *digit* übergibt dabei die Digit Nummer (Position) und der Parameter *segment* die zu aktivierenden Segmente dieser Stelle. Die Digit Bits bezeichnen jeweils eine der acht möglichen Digits. Dabei korrespondiert das Bit0 mit Digit0 = ganz linke Stelle etc. Gesetzte/aktive Bits schalten das jeweilige Digit bzw. Segment an.

Die Segmente folgen den hier üblichen Zuordnungen (Zeichnungen weiter unten):

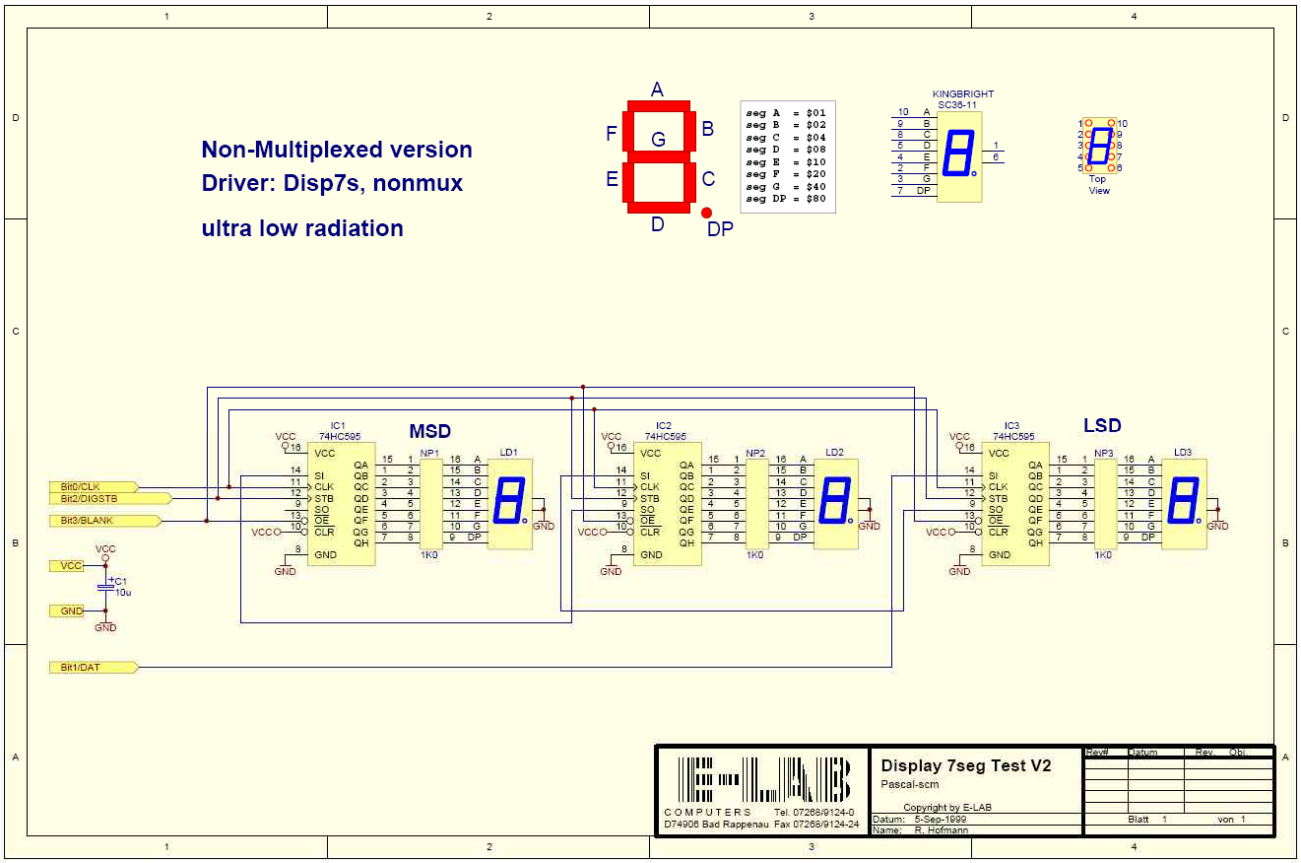
SegmentA	= $01
SegmentB	= $02
SegmentC	= $04
SegmentD	= $08
SegmentE	= $10
SegmentF	= $20
SegmentG	= $40
DecimalPnt	= $80

Der Buchstabe “C” ergibt deshalb den Wert $39 im Parameter Segment. Daraus resultiert z.B.
Digit = $01
Segment = $39
Die erste (linke) Stelle soll ein „C“ anzeigen.

Beispiel Programm ***AVR Disp7sUser*** in der Demos Directory unter ***Disp7sUser***.



Schaltplan **Disp7s** (multiplexed)



Schaltplan **Disp7s2** (non multiplexed)

3.8 LED 14seg Display

Allgemeines
7 und 14segment Displays werden, trotz ihrer eingeschränkten Informations Möglichkeiten, weiterhin da eingesetzt, wo es entweder auf grosse Zeichen (bis zu 20cm Höhe) oder grosse Helligkeit oder gute Lesbarkeit auf grössere Entfernung ankommt. Oft kommen auch mehrere dieser Anforderungen zusammen. Hier kann man dann keine der eigentlich flexibleren LCDs mehr einsetzen. Besonders die 14segment Typen erlauben eine wesentlich flexiblere und lesbarere Darstellung als ihre kleinen Brüder mit 7segmenten.

14segment-Displays werden entweder mit Schieberegister oder mit speziellen Display Treiber Chips angeschlossen. Der vorliegende Treiber unterstützt nur gemultiplexte Anordnungen mit Schieberegistern.

Durch den Einsatz von ultrahellen low current Typen halten sich die durch das multiplexen hervorgerufenen EMV Probleme in Grenzen, da hier mit relativ niedrigen Impuls Strömen gearbeitet werden kann >= 15mA

Einführung Disp14sPort
Der Treiber steuert ein 14segment LED Display mit bis zu 8 Stellen/Digits. Das Display wird im Common-Anode Multiplex Modus betrieben. Dafür werden 5 Bits eines Output-Ports benötigt sowie zwei serielle Treiberbausteine, zwei für die Segmente, z.B. UCN5841 und einen für die Digits, z.B. 74HC595. Der HC595 kann natürlich nicht direkt die Digits treiben, er benötigt einen nachgeschalteten Leistungstreiber.

Die On-Zeit eines Digits beträgt einen SysTick. Damit ist die Zeit für einen kompletten Refresh = Digits x SysTick (msec). Bei einem SysTick von 10msec und 4 Digits ergibt sich somit eine Refreshrate von 4 x 10msec = 40msec = 25Hz.

Das Bit0 des Ports wird an den jeweiligen CLOCK-Eingang und das Bit1 an den entspr. Dateneingang der beiden Treiber angeschlossen. Bit2 ist das Latch-Signal für den Digit-Treiber und das Bit3 ist das Latch-Signal für den Segment-Treiber. Bit4 ist das Output-Enable-Signal (Blank) für den Digit-Treiber.

Die Verdrahtung zwischen CPU, den Treiber Chips und dem 14seg Display ist fest vorgegeben durch den Software Treiber.

Die Treiber Funktionen entsprechen den Funktionen des Treibers ***Disp7sPort***.

Imports
Wie beim AVRco System üblich, muss der Treiber importiert und definiert werden.
Der SysTick wird auch benötigt.

```
Import   SysTick, Disp14sPort;
```

Defines
Wie auch beim Standard 7seg-Port muss der Display Aufbau definiert werden.

```
Define   ProcClock    = 8000000;          {8Mhz clock }
         SysTick      = 5;                {5msec}
         Disp14sPort  = PortB, 0;         {Port, start Portbit}
         Disp14Mode   = ShiftRight, Blank;
         Disp14Digits = 6, iData;
```

Disp14sPort
Definiert das zu verwendende Port. Der Parameter StartBit gibt an, mit welchem Bit des Ports die Steuerbits beginnen sollen. Die Reihenfolge bleibt aber konstant.

Disp14sDigits
Definiert die Anzahl der Digits des Treibers 1..8 und den Speicherbereich seines Buffers.

Disp14Mode
Arbeitsweise des Displays beim Anfügen eines neuen Zeichens. Mit **Wrap** wird der Einfüge Cursor nach erreichen des Display Endes auf die Stelle 0 gesetzt (ganz links). Mit **ShiftLeft** bleibt der Cursor ganz rechts stehen, das Zeichen wird hier eingefügt und der Rest des Inhalts wird um eine Position nach links verschoben. Mit **ShiftRight** ist dieser Vorgang genau umgekehrt.
Mit **Blank** wird ein blanking Signal generiert, das Geisterbilder während des Digit-Switch unterdrückt.
Mit **NoBlank** kann dieses Portbit u.U. eingespart werden.

3.8.1 Variablen

Alle Schreib Operationen erfolgen nach der Umsetzung von ASCII in die Segment Darstellung zuerst in den lokalen Buffer. Dieser Buffer wird dann über die Schieberegister an die beteiligten Segment Treiber verteilt. Die Applikation kann direkt in den Display Buffer schreiben und damit Segmente manipulieren oder Sonderzeichen darstellen.

Der Buffer ist als **Array[0..Disp14Digits -1] of word** definiert.

```
Var Disp14Buff : array[0.. Disp14Digits-1] of word;
```

3.8.2 Funktionen

Disp14Pos
Der Schreibcursor wird an eine bestimmte Stelle positioniert. Der Übergabe Parameter ist vom Typ Byte und bezeichnet die Stelle bzw. das Digit. Die Zählreihenfolge der Digits ist von ‘0’ (ganz links) bis I2C_7sDIGn -1 (ganz rechts);

```
Procedure Disp14Pos (const digit : byte);
I2C_Disp7Pos (2);                    (* position 2 *)
```

Disp14Out
Schreiben in den Display Buffer. Übergabe Parameter ist vom Typ Char. Das in der aktuellen Stelle des Displays befindliche Zeichen wird mit dem neuen überschrieben.

Da ein 14seg-Display nicht alle ASCII Zeichen darstellen kann, werden einige Zeichen als Leerzeichen dargestellt. Steuerzeichen werden bis auf ‘CR’ und ‘LF’ und #0 .. #9 ignoriert.

Linefeed (LF = $0A) löscht das Display und setzt den Schreibcursor an die Stelle ‘0’.
Return (CR = $0D) setzt den Schreibcursor an die Stelle ‘0’ verändert jedoch das Display nicht.
Ist der Schreibcursor an der letzten Stelle angekommen, wird er automatisch auf ‘0’ gesetzt (**Wrap** Mode).
Bei den **Shift** Modi wird der Display Inhalt um eine Stelle nach rechts bzw. links geschoben und das neue Zeichen angefügt.

```
Procedure Disp14Out (const ch : char);
  Disp14Out ('A');                   {show char A}
  Write (Disp14Out, IntToStr(i));    {string output}
```

Disp14Clear
Das ganze Display wird gelöscht Der Schreibcursor wird an die Stelle ‘0’ positioniert. Alle aktuelle Blinkfunktionen werden zurückgesetzt.

```
Procedure Disp14Clear;
  Disp14Clear;
```

Disp14CLREOL
Das Display wird von der aktuellen Cursorposition bis Zeilenende gelöscht. Die Position des Schreibcursors bleibt erhalten.

```
Procedure Disp14CLREOL;
  Disp14CLREOL;
```

Disp14Blink
Das ganze Display blinkt bzw. das Blinken wird ausgeschaltet. Der Übergabe Parameter ist vom Typ Boolean.

```
Procedure Disp14Blink (const blink : boolean);
  Disp14Blink (true);
```

Disp14DigBlink
Ein einzelnes Digit blinkt bzw. das Blinken wird ausgeschaltet. Der Übergabe Parameter ist vom Typ Byte und bezeichnet die Stelle bzw. das Digit. Mit „blink“ wird das blinken ein- bzw. ausgeschaltet.

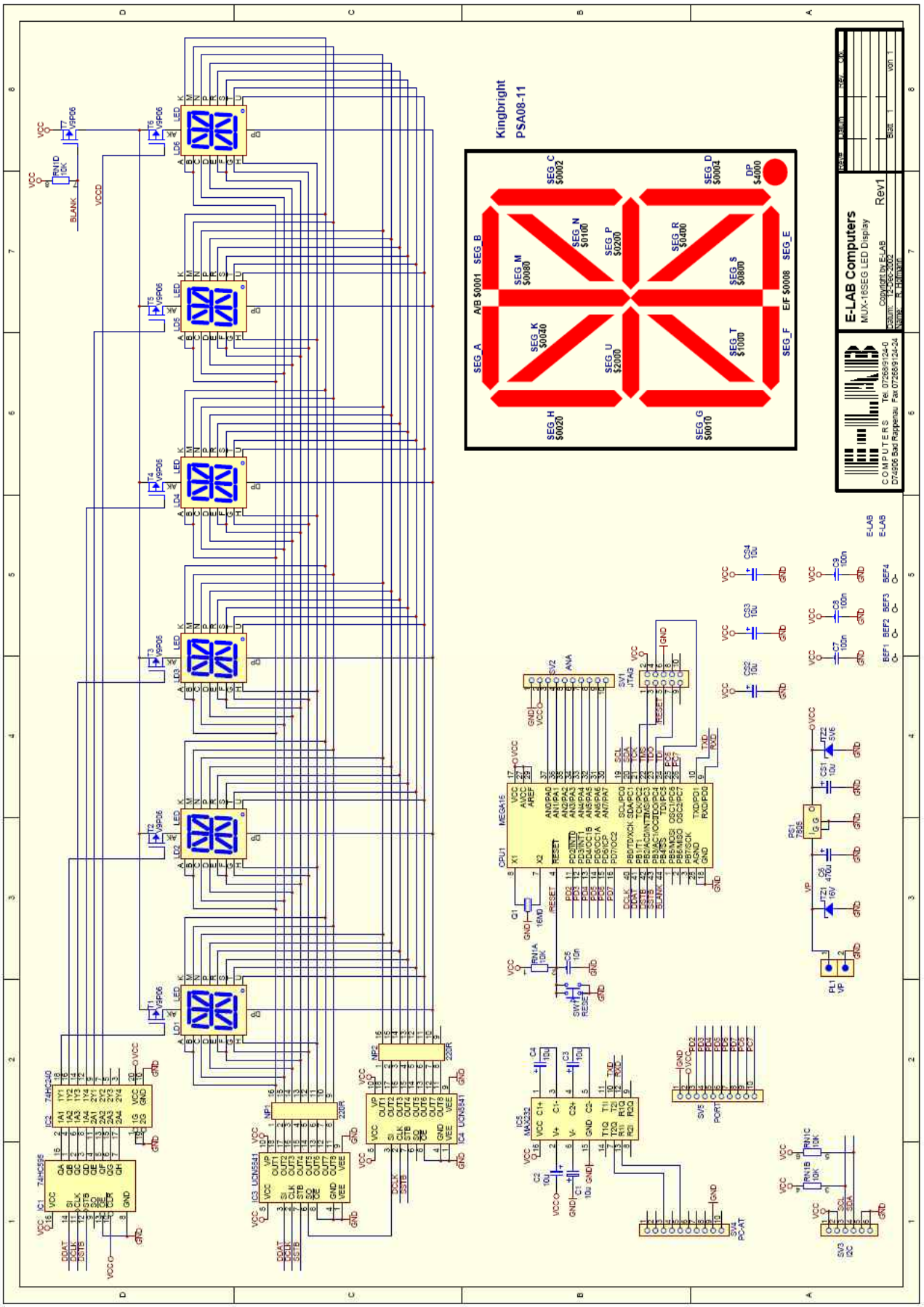
```
Procedure Disp14DigBlink (digit: byte; blink: boolean);
  Disp14DigBlink (1, true);
```

Disp14Test
Alle Segmente des Displays werden eingeschaltet. Der Schreibcursor wird an die Stelle ‘0’ positioniert. Alle aktuelle Blinkfunktionen werden zurückgesetzt.

```
Procedure Disp14Test;
  Disp14Test;
```

Programm Beispiel und Schaltung:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Disp14s**



Schaltplan **Disp14seg**

3.9 LED 7seg Display Treiber für bis zu 4 Displays

Allgemeines
7segment Displays werden, trotz ihrer eingeschränkten Informationsmöglichkeiten, weiterhin da eingesetzt, wo es entweder auf grosse Zeichen (bis zu 20cm Höhe) oder grosse Helligkeit oder gute Lesbarkeit auf grössere Entfernung ankommt. Oft kommen auch mehrere dieser Anforderungen zusammen. Hier kann man dann keine der eigentlich flexibleren LCDs mehr einsetzen.

7segment-Displays werden entweder mit Schiebergister oder mit speziellen Display Treiber Chips angeschlossen. Der Nachteil beider Versionen ist einerseits der grosse Code und Rechenzeit Aufwand und bei mehreren separaten Displays auch noch der zusätzliche CPU Pin Verbrauch.

Die Lösung des Problems ist hier der Einsatz von 7seg-Displays mit einem Bus-System, in der Regel I2C. Fertige 7seg-Displays mit Bus Anschluss gibt es allerdings nicht.

Es bietet sich hier an standard I2C LED Treiber Bausteine zu benutzen die z.B. 16 LEDs bzw. 2 Digits direkt treiben können. Dazu werden nur 2 Portleitungen der CPU benötigt, unabhängig von der Zahl der angeschlossenen Displays.

Moderne Chips dieser Art besitzen einen internen Blinker und/oder Dimmer, so dass die CPU hier nur Befehle erteilen muss, das Blinken, Dimmen und Multiplexen per kontinuierlicher Software Schleife, z.B. im SysTick, entfällt dadurch und die CPU wird wesentlich entlastet.

Einführung I2C_Disp7
Die vorliegende Implementation benutzt entweder den Software I2C-Treiber (I2Cport) oder den internen TWI (I2C) Port der AVR mega CPUs. Dazu ist entweder der Treiber *I2Cport* oder der Treiber *TWImaster* oder der Treiber *TWInet* im Mastermode zu importieren. Der Standard 7SegPort Treiber kann separat neben dem I2C_Disp7 Treiber existieren.

Als I2C Port-Expander muss für jeweils 2 Digits der Typ PCA9532 von Philips verwendet werden. Dieser Baustein kann bis zu 8 mal am Bus vorhanden sein. Dadurch ergibt sich die maximale Summe aller Digits von 16. Diese 16Digits können auf 4 separate Displays verteilt werden, wobei wiederum ein einzelnes Display nicht mehr als 8 Digits besitzen kann.

Der Treiber arbeitet im Non-multiplex Mode, was zwar zu einem grösseren Bauteile Aufwand führt, dafür gibt es relativ wenig EMV Probleme. Diese sind bei Multiplex-LEDs nicht zu unterschätzen, da ja das Display durch eine Glass bzw. Kunstoffscheibe ungehindert abstrahlen kann.

Die Verdrahtung zwischen PCA9532 und den 7seg-Displays ist durch den Treiber fest vorgegeben und kann dem folgenden Schaltplan entnommen werden. Der PCA9532 kann mit bis zu 400kBit/sec am I2C Bus betrieben werden.

Die Treiber Funktionen sind ähnlich den Funktionen des Treibers *Disp7sPort* mit der Erweiterung, dass für jede Funktion auch noch die Display Nummer mit angegeben werden muss (Ausnahme: I2C_Disp7Out).

3.9.1 Technische Daten

I2C Port oder oder	Software I2C importiert durch I2Cport CPU-TWI importiert durch TWImaster CPU-TWI importiert durch TWInet im Mastermode
Displays	bis zu 4 Stück, max. 8 Digits/Display, total 16 Digits
Hardware	I2C LED-Treiber Chip PCA9532 von Philips, 1 Stück für 2 Digits
I2C Adressen	die PCA9532 liegen auf den Bus-Adressen $60..$67 MSD...LSD Die PCA9532 haben drei Adresspins bzw. Bits die jeweils entsprechend beschaltet werden müssen, d.h. sie müssen fortlaufende Adressen haben. Das linke (MSD) Digit hat die niederwertige Adresse.

Imports
Wie beim AVRco System üblich, muss der Treiber importiert und definiert werden. Zusätzlich muss aber auch noch der gewünschte I2C/TWI Treiber importiert werden.
Der SysTick wird nicht benötigt.

```
Import  I2Cport, I2C_Disp7;
oder
Import  TWImaster, I2C_Disp7;
oder
Import  TWInet, I2C_Disp7;              // use Master mode
```

Defines
Je nach gewünschtem I2C bzw. TWIport muss dieses definiert werden. Weiterhin muss, wie auch beim Standard 7seg-Port, der Display Aufbau definiert werden.

Beispiel für *I2Cport*:

```
Define  ProcClock     = 8000000;        {8Mhz clock }
        I2Cport       = PortC;          {port used}
        I2Cdat        = 7;              {bit7-PortC}
        I2Cclk        = 6, 4;           {bit6-PortC, optional delay 4}
        I2C_Disp7     = I2C_Soft, iData;{use Software I2Cport, buffer loc}
            // use 2 Displays
        I2C_7sDig1    = 8;              {first  display 8digits}
        I2C_7sDig2    = 6;              {second display 8digits}
        I2C_7Mode     = wrap;
```

Beispiel für *TWImaster*:

```
Define  ProcClock     = 8000000;        {8Mhz clock }
        TWIpresc      = TWI_BR100;      {100kBit/sec alt. TWI_BR400}
        I2C_Disp7     = I2C_TWI, xData; {use TWIport, buffer location}
            // use 4 Displays
        I2C_7sDig1    = 4;              {first  display 4digits}
        I2C_7sDig2    = 4;              {second display 4digits}
        I2C_7sDig3    = 4;              {third  display 4digits}
        I2C_7sDig4    = 4;              {foutrh display 4digits}
        I2C_7Mode     = shiftLeft;
```

Beispiel für *TWInetMaster*:

```
Define  ProcClock     = 8000000;        {8Mhz clock }
        TWInode       = 05;             {default address in slave mode}
        TWIpresc      = TWI_BR400;      {400kBit/sec alt. TWI_BR100}
        TWIframe      = 4, iData;       {buffer/packet size}
        TWIframeBC    = 6;              {option broadcast buffer/packet size}
        TWInetMode    = Master;
        I2C_Disp7     = I2C_TWI, iData; {use TWIport, buffer location}
            // use 1 Display
        I2C_7sDig1    = 4;              {4digits}
        I2C_7Mode     = wrap;
```

I2C_Disp7
Definiert das zu verwendende I2C-Port. Entweder SoftWare-I2C mit ***I2C_Soft*** oder onchip TWIport mit ***I2C_TWI***. Der jeweilige Treiber dazu muss importiert und definiert werden.

I2C_7sDIGn
Definiert den Import eines Displays und die Anzahl dessen Digits

I2C_7Mode
Arbeitsweise des Displays beim Anfügen eines neuen Zeichens. Mit **Wrap** wird der Einfüge Cursor nach erreichen des Display Endes auf die Stelle 0 gesetzt (ganz links). Mit **ShiftLeft** bleibt der Cursor ganz rechts stehen, das Zeichen wird hier eingefügt und der Rest des Inhalts wird um eine Position nach links verschoben

3.9.2 Typen

Der Import von I2C_Disp7 veröffentlicht einen Aufzählungstyp (Enumeration), der für den Aufruf der Treiber Funktionen benutzt werden muss:

```
Type TI2C_DISP7 = (Disp7_1, Disp7_2, Disp7_3, Disp7_4);
```

Zum Testen, Ein- und Ausschalten eines Display muss für die zugehörige Funktion eine dieser Enumerationen verwendet werden:

```
Type TI2C_Ctrl7 = (Disp7_On, Disp7_Off, Disp7_Test);
```

3.9.3 Variablen

Alle Schreib Operationen erfolgen nach der Umsetzung von ASCII in die Segment Darstellung zuerst in lokale Buffer. Dieser Buffer wird dann über den I2C Bus an die beteiligten Segment Treiber verteilt. Die Applikation kann direkt in den Display Buffer schreiben und damit Segmente manipulieren oder Sonderzeichen darstellen.

Jedes mit Define definierte Display hat einen eigenen Buffer, der als Array[0..I2C_7sDIGn -1] definiert ist.

```
Var I2C_7Buff1, I2C_7Buff2, I2C_7Buff3, I2C_7Buff4;
```

3.9.4 Funktionen

Alle Funktionen mit der Ausnahme von *I2C_Disp7Out* benötigen als ersten Parameter das gewünschte Display. Dieser Parameter wird intern gespeichert und von I2C_Disp7Out benötigt.

I2C_Disp7Init
Jedes angeschlossene Display (Treiber Chips) muss zumindest einmal initialisiert werden. Der Buffer Inhalt wird dabei nicht verändert. Die Blinkrate gibt den Blink Rhythmus vor. Die zugehörige On/Off Zeit wird mit DutyCycle vorgegeben.

```
Function I2C_Disp7Init (const Disp : TI2C_Disp7; BlinkRate, DutyCycle : byte);
  I2C_Disp7Init (Disp7_1, 50, 127);  // 3Hz rate, 50/50 onoff
```

I2C_Disp7Ctrl
Jedes angeschlossene Display kann ein- und ausgeschaltet und getestet werden, ohne den Buffer Inhalt zu verändern. Die Test-Funktion schaltet alle Segmente incl. Dezimal Punkt ein. Mit der ON-Funktion wird der Testmode wieder verlassen.

```
Function I2C_Disp7Ctrl (const Disp : TI2C_Disp7; const ctrl : TI2C_Ctrl7);
  I2C_Disp7Ctrl (Disp7_1, Disp7_Test);
  I2C_Disp7Ctrl (Disp7_2, Disp7_Off);
  I2C_Disp7Ctrl (Disp7_3, Disp7_On);
```

I2C_Disp7Test
Testfunktion. Testfunktion. Schaltet alle Segmente ein

```
Procedure I2C_Disp7Test (const Disp: TI2C_Disp7);
```

I2C_Disp7Pos
Der Schreibcursor wird an eine bestimmte Stelle positioniert. Der Übergabe Parameter ist vom Typ Byte und bezeichnet die Stelle bzw. das Digit. Die Zählreihenfolge der Digits ist von '0' (ganz links) bis I2C_7sDIGn -1 (ganz rechts);

```
Procedure I2C_Disp7Pos (const Disp : TI2C_Disp7; const digit : byte);
  I2C_Disp7Pos (Disp7_1, 2);            (* position 2 *)
```

I2C_Disp7Set
Optionales Umschalten des Treibers auf ein bestimmtes Display. Nur sinnvoll im Zusammenhang mit ***I2C_Disp7Out***.

```
Procedure I2C_Disp7Set (const disp : TI2C_Disp7);
  I2C_Disp7Set (Disp7_1);
```

I2C_Disp7Get
Die Funktion liefert als Ergebnis das aktuell eingestellt Display.

```
Function I2C_Disp7Get : TI2C_Disp7;
```

I2C_Disp7Out
Schreiben in den Display Buffer. Übergabe Parameter ist vom Typ Char. Das in der aktuellen Stelle des Displays befindliche Zeichen wird mit dem neuen überschrieben.

Da diese Prozedur auch mit ***Write*** funktionieren muss, gibt es hier den sonst obligatorischen Parameter *Disp* nicht. Das Schreiben auf das Display erfolgt daher immer auf dasjenige, auf welches zuletzt mit einer Funktion zugegriffen wurde. Das gewünschte Display kann aber auch explizit mit ***I2C_Disp7Set*** vorgegeben werden.

Da ein 7seg-Display nicht alle Zeichen des Alphabets darstellen kann, werden einige Zeichen als Leerzeichen dargestellt. Steuerzeichen werden bis auf 'CR' und 'LF' ignoriert.

Linefeed (LF = $0A) löscht das Display und setzt den Schreibcursor an die Stelle '0'.
Return (CR = $0D) setzt den Schreibcursor an die Stelle '0' verändert jedoch das Display nicht.
Ist der Schreibcursor an der letzten Stelle angekommen, wird er automatisch auf '0' gesetzt (**Wrap**) Mode.
Bei den **Shift** Modi wird der Display Inhalt um eine Stelle nach links geschoben und das neue Zeichen angefügt.

```
Procedure I2C_Disp7Out (const ch : char);
  I2C_Disp7Out ('A');                   {show char A}
  Write (I2C_Disp7Out, IntToStr (i));          {string output}
```

I2C_Dipp7Clear
Das ganze Display wird gelöscht. Der Schreibcursor wird an die Stelle '0' positioniert. Alle aktuelle Blinkfunktionen werden zurückgesetzt.

```
Procedure I2C_Disp7Clear (const Disp : TI2C_Disp7);
  I2C_Disp7Clear (Disp7_1);
```

I2C_Disp7CLEOL
Das Display wird von der aktuellen Cursorposition bis Zeilenende gelöscht Die Position des Schreibcursors bleibt erhalten.

```
Procedure I2C_Disp7CLEOL (const Disp : TI2C_Disp7);
  I2C_Disp7CLEOL (Disp7_1);
```

I2C_Disp7DigitBlink
Ein einzelnes Digit blinkt bzw. das Blinken wird ausgeschaltet. Der Übergabe Parameter ist vom Typ Byte und bezeichnet die Stelle bzw. das Digit. Mit „blink“ wird das blinken ein- bzw. ausgeschaltet.

```
Procedure I2C_Disp7DigitBlink (Disp : TI2C_Disp7; digit: byte; blink: boolean);
  I2C_Disp7DigitBlink (Disp7_1, 1, true);
```

I2C_Disp7Dim
Die Displays können gedimmt werden. Die Helligkeit kann von 0..255 eingestellt werden. Beim Dimmen sind zwei wichtige Punkte zu beachten:

1. blinkendes Display schaltet immer zwischen 0% und 100% Helligkeit um. Dimmwerte werden hierbei nicht beachtet.
2. Dimmen erfolgt mit Puls-Pausen Modulation (PWM). Dadurch kann es zu erhöhter Störabstrahlung kommen, die aber nur ein Bruchteil eines Multiplex Displays beträgt.

```
Procedure I2C_Disp7Dim (const Disp : TI2C_Disp7; const dim : byte);
  I2C_Disp7Dim (Disp7_1, 127);          (* brightness to 50% *)
```

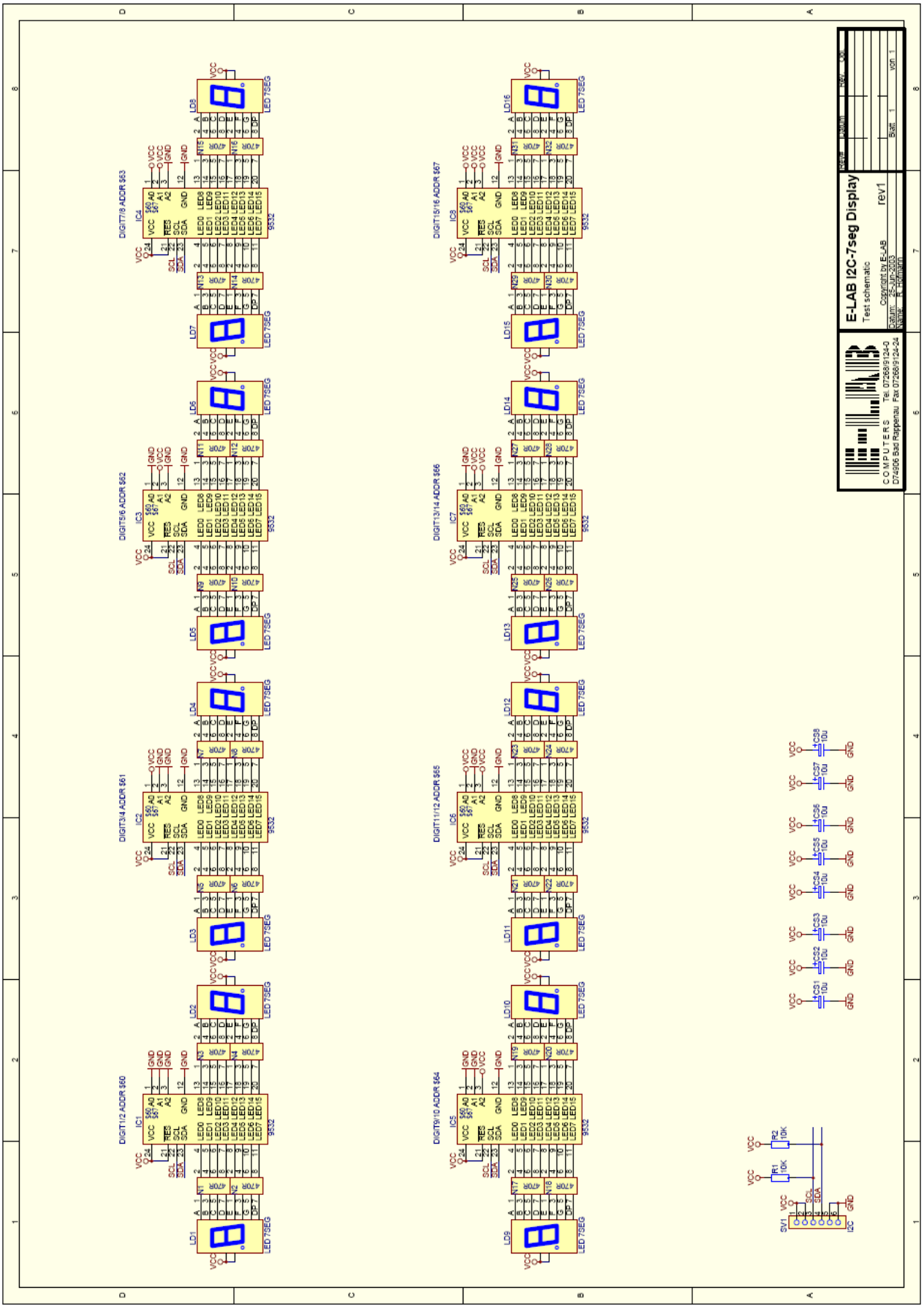
I2C_Disp7Refresh
Alle Funktionen und Prozeduren führen automatisch einen Refresh/Update vom Display Buffer in die Digit Treiber aus, wenn dies erforderlich ist.

Wird jedoch direkt in den ***I2C_7Buffx*** geschrieben, hat das keine direkte Auswirkung, da die externen Treiber dadurch nicht upgedated werden. Nach einem abgeschlossenen Schreibvorgang auf den Buffer sollte daher diese Prozedur aufgerufen werden, um die Änderung auch sichtbar zu machen.

```
Procedure I2C_Disp7Refresh (const Disp : TI2C_Disp7);
  I2C_Disp7Refresh (Disp7_1);
```

Programm Beispiel und Schaltung:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\I2C_Disp7s**



Schaltplan **I2C_7seg**

3.10 IOexpand Treiber für bis zu 128 digitale IOs

Allgemeines
Bei manchen Steuerungs Anwendungen reichen die bei einer bestimmten CPU zur Verfügung stehenden IOs bzw. Port Pins nicht aus, vor allem wenn zwei Ports komplett durch die externe Speicheransteuerung wegfallen.

Hier hilft nur eine noch grössere CPU oder die Erweiterung der möglichen Port Bits durch zusätzliche Hard- und Software. Hierbei gibt es mehrere Möglichkeiten eine solche Erweiterung zu implementieren. Echte IO-Chips wie z.B. 8255 z.B. wegen Platz oder Ansteuer Gründen nicht verwendet werden, ausserdem bieten diese max. 20 zusätzliche Bits. Standard Latches z.B. müssen ebenfalls parallel angesteuert werden und benötigen dadurch doch erheblich Port Bits der CPU, so dass die Einsparung and CPU Resourcen doch nicht so gross ist.

Wenn es nicht auf allzu grosse Geschwindigkeit bei der Port Bearbeitung ankommt, und die Anzahl der notwendigen zusätzlichen gewünschten Bits erheblich ist, kommt eigentlich nur das Schieberegister Vefahren in Betracht.

Jedes Schieberegister bedient entweder 8 input oder 8 output bits. Die Ansteuerung erfolgt typischerweise mit 3 Steuerpins. Da Schieberegister kaskadierbar (hintereinander schaltbar) sind, bleibt die Anzahl der notwendigen Steuerpins konstant, egal wieviele Register in Serie geschaltet sind. Der Nachteil dieser Anordnung besteht darin, dass pro gelesenem bzw. geschriebenen Bit ein Schieberegister Zyklus gebraucht wird (Clock, Daten).
Durch geschickte Programmierung des Treibers kann bei Input und Output Registern sowohl Port Pins der CPU als auch Maschinen Befehle bei der Ausführung eingespart werden.

Einführung IOexpander
Die vorliegende Implementation benutzt 3 bidirektionale Port Bits der CPU wenn nur ein Mode gebraucht wird, entweder nur Input oder nur Output. Wenn Input und Output kombiniert werden, müssen 5 bi-direktionale Port Pins bereitgestellt werden. Alle Port Pins müssen in einer ununterbrochenen Reihe auf dem gleichen Port liegen.

Der Treiber ist völlig passiv, d.h. er muss von der Applikation aufgerufen werden (zyklisch durch Task, Prozess, Interrupt oder nach Bedarf). Der Treiber stellt seine Resultate in Variablen zur Verfügung, die dann gelesen werden können. Ausgabe Werte die sich ändern sollen, müssen vor dem Treiber Aufruf in entsprechende Variablen geschrieben werden, die der Treiber dann ausliest und an die Schieberegister ausgibt.

Jeder Treiber Aufruf liest also die Output Variablen und schiebt deren Inhalt in die Output Register. Gleichzeitig mit jeder Bit-Schiebe Operation liest er die Input Schieberegister und legt deren Werte im Speicher ab.

Der Treiber arbeitet Byte-orientiert, d.h. es müssen immer 8bit Schieberegister angeschlossen werden. Hierbei sind maximal 8 Register zulässig was 64 bits entspricht. Das gilt sowohl für den Input Bereich als auch für den Output. Die 8bit Register müssen nicht symmetrisch sein. Es können z.B. 8 Input Register mit 1 Output Register kombiniert werden.

3.10.1 Technische Daten

Input Register	0..8	Bits	0, 8, 16, 24, 32, 40, 48, 56, 64
Output Register	0..8	Bits	0, 8, 16, 24, 32, 40, 48, 56, 64
Input Datenbytes	0..8	im iData oder xData Bereich	
Output Datenbytes	0..8	im iData oder xData Bereich	
Benutzte Port Bits	Input only	3	
	Output only	3	
	kombiniert	5	

Imports
Wie beim AVRco System üblich, muss der Treiber importiert und definiert werden.
Der SysTick wird nicht benötigt.

```
Import  IOexpand;
```

Defines
Es muss spezifiziert werden, welches Port der CPU und welche bits davon benutzt werden sollen, wo die notwendigen Variablen liegen und ebenfalls wieviel Input Bytes und Output Bytes benötigt werden.

```
Define   ProcClock   = 8000000;          {8Mhz clock }
         IOexpand    = PortD, 2, iData;  {Port, Port-startbit, memory loc}
         IOexpInp    = 4;                (4x8 = 32 bit input}
         IOexpOutp   = 4;                (4x8 = 32 bit output}
```

IOexpand
Definiert das zu verwendende Port, das erste benutzbare bit und den Datenbereich des Treibers. Das Port muss bidirektional sein. Das Startbit im Port muss so gewählt werden, dass die 3 bzw. 5 notwendigen Steuerbits auch noch in das Port passen. Der Datenbereich bestimmt, wo die Arbeitsvariablen des Treibers liegen.

IOexpInp
Definiert die Zahl der Input Bytes und bestimmt dadurch auch den internen Speicherbedarf.

IOexpOutp
Definiert die Zahl der Output Bytes und bestimmt dadurch auch den internen Speicherbedarf.

3.10.2 Funktionen und Variable

IOexpInpArr
Der Import des Treibers stellt diese Array zur Verfügung, wenn Input Pins/Bytes deklariert wurden. Die Grösse dieses Arrays hängt von der Zahl der Input Bytes ab. Der Treiber legt in diesem Array seine Lese Ergebnisse ab und zwar in IOexpInpArr[0] das erste Byte, in IOexpInpArr[1] das zweite etc.

Die Definition sieht deshalb so aus:

```
Var IOexpInpArr : array[0.. IOexpInp-1] of byte;
```

IOexpInp*x*
Die einzelnen Bytes dieses Arrays haben auch ein Synonym, so dass auf die indizierte Array Adressierung verzichtet werden kann. Die einzelnen Bytes, von denen es natürlich nur [IOexpInp] Stück gibt, sind so definiert:

```
Var   IOexpInp0 : byte;
      IOexpInp1 : byte;
      IOexpInp2 : byte;
      ...
```

IOexpOutpArr
Der Import des Treibers stellt diese Array zur Verfügung, wenn Output Pins/Bytes deklariert wurden. Die Grösse dieses Arrays hängt von der Zahl der Output Bytes ab. Der Treiber liest aus diesem Array seine Ausgabe Vorgabe Werte und zwar aus IOexpOutpArr[0] das erste Byte, in IOexpOutpArr[1] das zweite etc.

Die Definition sieht deshalb so aus:

```
Var IOexpOutpArr : array[0.. IOexpOutp-1] of byte;
```

IOexpOutp*x*
Die einzelnen Bytes dieses Arrays haben auch ein Synonym, so dass auf die indizierte Array Adressierung verzichtet werden kann. Die einzelnen Bytes, von denen es natürlich nur [IOexpOutp] Stück gibt, sind so definiert:

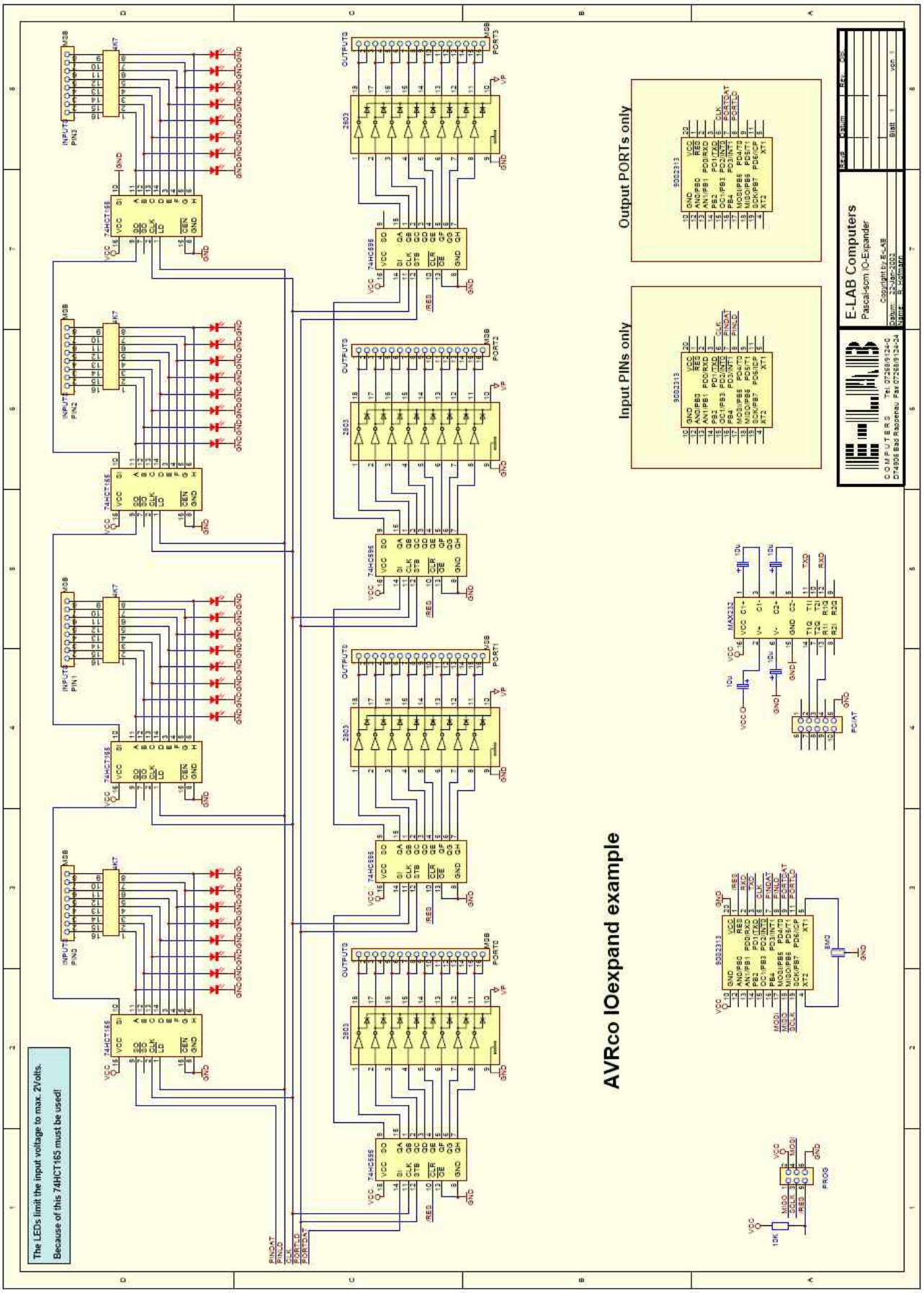
```
Var   IOexpOutp0 : byte;
      IOexpOutp1 : byte;
      IOexpOutp2 : byte;
      ...
      ...
```

IOexpUpdate
Die Applikation führt diesen Prozedur Aufruf durch, der den Treiber veranlasst, sämtliche Input Schieberegister einzulesen und in dem Input Array bzw. Bytes abzulegen. Im gleichen Vorgang wird das Output Array bzw. Bytes gelesen und an die Output Schieberegister ausgegeben.
Die Definition von IOexpUpdate ist:

```
Procedure IOexpUpdate;
```

Programm Beispiel und Schaltung:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\IOexpand**



Schaltplan **IOexpander**

3.11 RS232/V24 Treiber SerPort, SerPort2 und SerPort3, -4

Die größeren AVR Typen enthalten eine oder mehrere serielle Schnittstellen. Bei den älteren Typen (AT90Sxxxx) ist ein UART implementiert, die neueren enthalten sogenannte USARTs.
Die USARTs sind wesentlich flexibler als der frühere UART.

Das AVRco System kennt, abhängig vom CPU Typ, bis zu 4 SerPorts:
SerPort (SerPort1), SerPort2, SerPort3, SerPort4,
Die **XMegas** bieten bis zu 8 Schnittstellen, **SerPortC0, SerPortC1, SerPortD0, SerPortD1** etc.

Hat eine CPU USARTs statt UARTs, stellt das AVRco System diverse zusätzliche System Funktionen zur Verfügung, welche die erweiterten Funktionen dieser USARTs unterstützen, wie z.B. verändern der Databit Länge, Paritymode und Stopbit Anzahl. Diese Funktionen sind zur Laufzeit aufrufbar.

Weiterhin können die internen **Defaultwerte** (2 Stopbits, noParity, 8 Databits) zur Design Zeit über die Defines überschrieben werden, so dass das System schon mit den gewünschten Einstellungen startet und eine Runtime Umstellung normalerweise nicht notwendig ist.

3.11.1 Grundlegende Funktionen (UART und USART)

Imports
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.

```
Import SysTick, SerPort, SerPort2, ..;    // SerPort = SerPort1
```

Defines
Die Baudrate, optionale Rx und/oder TxBuffer, optionale Databits, Stopbits und Parity

```
Define   ProcClock   = 4000000;          {4Mhz clock }
         SysTick     = 10;               {10msec Tick}
         SerPort     = 9600, Stop1;      {9600 Baud, 1Stopbit}
         //SerPort   = 9600, parEven;    {9600 Baud, gerade Parität}
         //SerPort   = 9600, parOdd;     {9600 Baud, ungerade Parität}
         // SerPort1 = 9600, parOdd;     {9600 Baud, odd parity}
         TxBuffer    = 8;                {8 Byte Buffer and Int}
         // TxBuffer1 = 8;               {8 Byte Buffer and Int}
         RxBuffer    = 8, iData;         {8 Byte Buffer and Int}
         // RxBuffer1 = 8, iData;        {8 Byte Buffer and Int}
```

Für die weiteren seriellen Schnittstellen lauten die Definitionen analog:

```
Define   SerPort2    = 9600, Stop1;      {9600 Baud, 1Stopbit}
         //SerPort2  = 9600, parEven;    {9600 Baud, gerade Parität}
         //SerPort2  = 9600, parOdd;     {9600 Baud, ungerade Parität}
         TxBuffer2   = 8;                {8 Byte Buffer und Int}
         RxBuffer2   = 8, iData;         {8 Byte Buffer und Int}

Define   SerPort3    = 9600, Stop1;      {9600 Baud, 1Stopbit}
         //SerPort3  = 9600, DataBit5    { DataBit5, DataBit6, DataBit7, DataBit9}
         TxBuffer3 ...
         etc.
```

XMega

```
Define   ProcClock   = 4000000;          {4Mhz clock }
         SysTick     = 10;               {10msec Tick}
         SerPortC0   = 9600, Stop1;      {9600 Baud, 1Stopbit}
         //SerPortC0 = 9600, parEven;    {9600 Baud, gerade Parität}
         //SerPorC0  = 9600, parOdd;     {9600 Baud, ungerade Parität}
         //SerPorC0  = 9600, SyncMaster; {9600 Baud, synchron mode, XCK output}
         //SerPorC0  = 9600, SyncSlave;  {9600 Baud, synchron mode, XCK input}
         TxBufferC0  = 8;                {8 Byte Buffer und Int}
         RxBufferC0  = 8, iData;         {8 Byte Buffer und Int}
```

Für die weiteren seriellen Schnittstellen lauten die Definitionen analog:

```
Define   SerPortC1   = 9600, Stop1;      {9600 Baud, 1Stopbit}
         //SerPortC1 = 9600, parEven;    {9600 Baud, gerade Parität}
         //SerPortC1 = 9600, parOdd;     {9600 Baud, ungerade Parität}
         TxBufferC1  = 8;                {8 Byte Buffer und Int}
         RxBufferC1  = 8, iData;         {8 Byte Buffer und Int}

Define   SerPortD0   = 9600, Stop1;      {9600 Baud, 1Stopbit}
         //SerPortD0 = 9600, DataBit5    { DataBit5, DataBit6, DataBit7, DataBit9}
         TxBufferD0 ...
```

etc.

Durch den Import der XMega SerPorts wird dieser Typ (Enumeration) durch das System deklariert:

```
Type
  tUSARTenum = (UsartC0, UsartC1, UsartD0, UsartD1, UsartE0, UsartE1, UsartF0, UsartF1, UsartCDC);
```

Bemerkungen:

Da beim Standard UART das 9. Bit entweder als 2. Stopbit verwendet wird oder als Paritybit, kann entweder mit 2 Stopbits gearbeitet werden oder mit Parity, beides zusammen geht nicht! Es gibt dabei aber Ausnahmen, wenn z.B. der UART dieses durch eigene Hardware unterstützt.

Bitte beachten, dass -wenn eine Parity Funktion angegeben ist- die Parity für jedes einzelne gesendete und empfangene Byte **vom System selbst errechnet** werden muss, sofern der UART das nicht durch eigene Hardware unterstützt. Parameter Änderungen zur Laufzeit sind möglich.

RXBUFFER (RXBUFFER1), RAMpage

Definition der RX-Buffer Länge (Ringbuffer).
RXBUFFER erfüllt zwei Funktionen: zum einen werden die Speicherstellen in der gewünschten Anzahl (4..254) bereitgestellt, zum anderen wird, der Receiver im Interrupt betrieben, ansonsten im Polling (wenn RxBuffer nicht definiert ist). Bei der Definition der Bufferlänge mit RxBuffer muss beachtet werden, dass die meisten Prozessoren nur einen relativ kleinen Ram Speicher haben.

Der Buffer wird als Ringbuffer betrieben ohne Überwachung. Das bedeutet, dass u.U. Zeichen verloren gehen, wenn sie nicht rechtzeitig mit mittels *SerInp* abgeholt werden. Das kann bis zum Programmabsturz führen. Also sorgfältige Programmierung auch bei Buffer Betrieb!

TXBUFFER (TXBUFFER1), RAMpage

Definition der TX-Buffer Länge. Das unter RxBuffer ausgeführte gilt auch hier uneingeschränkt, mit der Ausnahme dass der TxBuffer nicht überlaufen kann.

3.11.1.1 Funktionen und Prozeduren

SerStat (SerStat1, SerStat2, SerStat3, SerStat4)

XMega

SerStatC0, SerStatC1, SerStatD0, SerStatD1, SerStatE0, SerStatE1, SerStatF0, SerStatF1

Liefert den Status der seriellen Schnittstelle. Sollte zumindest vor dem Aufruf von SerInp aufgerufen werden, um unnötige Wartezeiten innerhalb der Prozedur zu vermeiden.
SerStat liefert TRUE zurück, wenn ein Zeichen vorhanden ist. Bei Prozesse oder Tasks sollte auf SerStat verzichtet werden und dafür WaitPipe(RxBuffer) verwendet werden.

Weiterhin kann die Funktion "PipeFull" auf die seriellen Schnittstellen angewendet werden:

Function *PipeFull (RxBuffer): boolean;*

SerStatP (SertStatP1, SertStatP2, SertStatP3, SertStatP4)
XMega
SerStatPC0, SerStatPC1, SerStatPD0, SerStatPD1, SerStatPE0, SerStatPE1, SerStatPF0, SerStatPF1

Mit
Define *SerPort = 9600, parEven;*
XMega
Define *SerPortC0 = 9600, parEven;*
wird eine gerade Parität festgelegt.

Mit
Define *SerPort = 9600, parOdd;*
XMega
Define *SerPortC1 = 9600, parOdd;*
wird eine ungerade Parität festgelegt.

Tritt ein Paritäts Fehler beim Empfang auf, bleibt das Byte trotzdem gültig und kann gelesen werden. Nach dem Lesen kann mit der Funktion

SerStatP : boolean;
XMega
SerStatPC0 : boolean;

das Ergebnis der Paritätsprüfung festgestellt werden. War die Parität falsch, kommt ein true (= Error) ansonsten ein false zurück. Wird mit Rx-Interrupt gearbeitet, so kann nur festgestellt werden, dass seit dem letzten Aufruf von "SerStatP" ein Parity Fehler aufgetreten ist, welches Byte im RxBuffer davon betroffen war, kann nicht mehr festgestellt werden.
Der Funktions Aufruf von "SerStatP" setzt auch gleichzeitig das systeminterne
Parity Error Flag zurück (false).

SerBaud (SerBaud1, SerBaud2, SerBaud3, SerBaud4)
XMega
SetSerBaud
Stellt die BaudRate der seriellen Schnittstelle zur Laufzeit ein.

SerBaud(19200);
XMega
SetSerBaud(UsartC0, 19200);

SerInp (SerInp1, SerInp2, SerInp3, SerInp4)
XMega
SerInpC0, SerInpC1, SerInpD0, SerInpD1, SerInpE0, SerInpE1, SerInpF0, SerInpF1
Liest die serielle Schnittstelle bzw. Buffer.
Im Pollingbetrieb (RxBuffer nicht definiert) wird beim Aufruf von SerInp nicht der Buffer gelesen sondern das Empfangsregister des UART's. In beiden Fällen wartet die Prozedur, bis ein Zeichen zum Abholen bereit ist. Um unnötige Wartezeiten oder gar ein Blockieren des Programms zu vermeiden, ist es normalerweise sinnvoll zuerst den Schnittstellenstatus mittels SerStat abzuholen und dann ggf. das empfangene Zeichen.

```
repeat until SerStat;          {warte auf RxReady}
c := SerInp;
XMega
repeat until SerStatC0;        {warte auf RxReady}
c := SerInpC0;
```

Sind Prozesse importiert, ist es besser, die Schnittstelle wird in einem Prozess verarbeitet. Der Prozess wartet dann mit **WaitPipe**(RxBuffer); auf ein Zeichen und verschwendet damit überhaupt keine Rechenzeit. Der RxBuffer kann mit **FlushBuffer**(RxBuffer) zurückgesetzt werden.

XMega
WaitPipe (RxBufferC0) **FlushBuffer**(RxBufferC0)

SerOut (SerOut1, SerOut2, SerOut3, SerOut4)
XMega
SerOutC0, SerOutC1, SerOutD0, SerOutD1, SerOutE0, SerOutE1, SerOutF0, SerOutF1
Schreibt in die serielle Schnittstelle bzw. Buffer
Im Pollingbetrieb (TxBuffer nicht definiert) wird beim Aufruf von SerOut nicht in den Buffer geschrieben sondern direkt in das Senderegister des UART's. In beiden Fällen wartet die Prozedur, bis (polling) das letzte Zeichen gesendet wurde und das Senderegister frei ist, bzw. (Interrupt) bis der Sendebuffer ein neues Zeichen aufnehmen kann.

```
SerOut ('x');
Write (SerOut, ByteToStr (100:6));      {-> '   100'}
XMega
SerOutC0 ('x');
Write (SerOutC0, ByteToStr (100:6));  {-> '   100'}
```

SerInpBlock (SerInpBlock1, SerInpBlock2, SerInpBlock3, SerInpBlock4)
SerInpBlock_P (SerInpBlock1_P, SerInpBlock2_P, SerInpBlock3_P, SerInpBlock4_P)
```
Procedure SerInpBlock(var location: type);
Procedure SerInpBlock_P(ptr : pointer; len : word);
```
XMega
SerInpBlock
SerInpBlock_P
```
Procedure SerInpBlock (Usart : tUSARTenum; var location: type);
Procedure SerInpBlock_P(Usart : tUSARTenum; ptr : pointer; len : word);
```

Diese Funktionen ermöglichen es eine beliebige Datenstruktur zu empfangen. Die möglichen Speicherbereiche sind Prozedur/Funktions-lokal, RAM, EEProm und Flash. Mit SerInpBlock_P, SerInpBlock2_P etc. kann das Ziel aber nur das RAM sein. Der Typ des Ziels (location) im RAM wird nur zur Berechnung der Block Grösse benötigt. Die eigentliche Struktur wird nicht beachtet.

SerOutBlock (SerOutBlock1, SerOutBlock2, SerOutBlock3, SerOutBlock4)
SerOutBlock_P (SerOutBlock1_P, SerOutBlock2_P, SerOutBlock3_P, SerOutBlock4_P)
```
Procedure SerOutBlock(const location: type);
Procedure SerOutBlock_P(p : pointer; count : word);
```
XMega
```
Procedure SerOutBlock(Usart : tUSARTenum; const location: type);
Procedure SerOutBlock_P(Usart : tUSARTenum; p : pointer; count : word);
```

ermöglicht es, eine beliebige Datenstruktur zu senden. Die möglichen Speicherbereiche sind Prozedur/Funktions-lokal, RAM, EEProm und Flash

3.11.1.2 Interrupt Betrieb

In den meisten Fällen werden serielle Schnittstellen im Interrupt betrieben. Das hat den Vorteil, dass das Senden und Empfangen von Zeichen im Hintergrund abläuft. SerPort Interrupts benötigen immer einen Ringbuffer indem sich die Empfangs/Sende Daten befinden. Durch das Define eines Rx- oder TxBuffers wird gleichzeitig auch das zugehörige Interrupt System importiert.

```
Define   RxBuffer    = 24,  iData;
         TxBuffer    = 16,  iData;
         RxBuffer2   = 40,  xData;
         TxBuffer2   =  8,   xData1;
         RxBuffer3   = 40,  xData;
         TxBuffer3   =  8,   xData1;
         RxBuffer4   = 40,  xData;
         TxBuffer4   =  8,   xData1;
```

XMega

```
Define   RxBufferC0  = 24,  iData;
         TxBufferC0  = 16,  iData;
         RxBufferF0  = 40,  xData;
         TxBufferF1  =  8,  xData1;
```

SerInp Timeout

Eine Alternative beim Empfang von seriellen Daten zu "if SerStat then SerInp" ist der Aufruf von SerInp mit einem Timeout Parameter. Damit lassen sich manche Aufgaben besser erledigen. Der TimeOut Mode beim Zeichen Empfang muss deklariert werden:

```
Define   SysTick     = 10;                    // necessary for timeout handling
         SerPort     = 9600, Stop2, timeout;  // Stop2 and timeout are optional
         SerPort2    = 9600, Stop2, timeout;  // Stop2 and timeout are optional
         SerPort3    = 9600, Stop2, timeout;  // Stop2 and timeout are optional
         SerPort4    = 9600, Stop2, timeout;  // Stop2 and timeout are optional
XMega
Define   SysTick     = 10;                    // necessary for timeout handling
         SerPortC0   = 9600, Stop2, timeout;  // Stop2 and timeout are optional
         SerPortD1   = 9600, Stop2, timeout;  // Stop2 and timeout are optional
         SerPortE1   = 9600, Stop2, timeout;  // Stop2 and timeout are optional
         SerPortF0   = 9600, Stop2, timeout;  // Stop2 and timeout are optional
```

Damit werden diese Systemfunktionen importiert:

```
Function SerInp_TO (var rx : char|byte; const timeout : byte) : boolean;
Function SerInp_TO1 (var rx : char|byte; const timeout : byte) : boolean;
Function SerInp_TO2 (var rx : char|byte; const timeout : byte) : boolean; // 2.chan
Function SerInp_TO3 (var rx : char|byte; const timeout : byte) : boolean; // 3.chan
Function SerInp_TO4 (var rx : char|byte; const timeout : byte) : boolean; // 4.chan
XMega
Function SerInp_TOC0(var rx : char|byte; const timeout : byte) : boolean; // SerPortC0
Function SerInp_TOD1(var rx : char|byte; const timeout : byte) : boolean; // SerPortD1
Function SerInp_TOE1(var rx : char|byte; const timeout : byte) : boolean; // SerPortE1
Function SerInp_TOF0(var rx : char|byte; const timeout : byte) : boolean; // SerPortF0
```

Der Parameter "rx" muss eine Variable vom Typ byte oder char sein, EEprom ist hier nicht möglich. TimeOut ist ein Byte. Nach Rückkehr enthält die variable "rx" den empfangener Character oder Byte, wenn das Ergebnis der Funktion true ist. Ist das Ergebnis false, bleibt die Variable "rx" unverändert, denn es wurde kein Zeichen empfangen.
Der Parameter "TimeOut" zählt in SysTicks. Wird hier eine null übergeben kehrt die Funktion sofort zurück, mit oder ohne Zeichen (true/false). Ist der Parameter > 0 dann wird solange gewartet bis entweder ein Zeichen angekommen ist oder die eingestellte Zahl SysTicks abgelaufen sind.

SerInp_TO ist nicht mit Read und ReadLn verwendbar!

SerInpBlock Timeout

Die bisherigen Prozeduren "SerInpBlock" wurden durch TimeOut gesteuerte Varianten erweitert. Hier gilt das gleiche wie oben unter "SerInp TimeOut" ausgeführt wurde. Die Funktion kehrt mit einem true zurück, wenn "location" komplett mit den empfangenen Daten aufgefüllt wurde. Tritt ein TimeOut auf, so ist der Rückgabe Wert false und die übergebene Variable ist gar nicht oder nur zum Teil überschrieben. Das TimeOut wird für jedes einzelne Zeichen verwendet. TimeOut muss importiert werden:

```
Define   SysTick     = 10;                    // necessary for timeout handling
         SerPort     = 9600, Stop2, timeout;  // Stop2 and timeout are optional
         // SerPort1 = 9600, Stop2, timeout;  // Stop2 and timeout are optional
         SerPort2    = 9600, Stop2, timeout;  // Stop2 and timeout are optional
         SerPort3    = 9600, Stop2, timeout;  // Stop2 and timeout are optional
         SerPort4    = 9600, Stop2, timeout;  // Stop2 and timeout are optional
```

```
XMega
Define   SysTick     = 10;                    // necessary for timeout handling
         SerPortC0  = 9600, Stop2, timeout;  // Stop2 and timeout are optional
         SerPortD1 = 9600, Stop2, timeout;   // Stop2 and timeout are optional
         SerPortE1 = 9600, Stop2, timeout;   // Stop2 and timeout are optional
         SerPortF0 = 9600, Stop2, timeout;   // Stop2 and timeout are optional
```

Damit werden diese Systemfunktionen importiert:

```
Function SerInpBlock_TO(var location : type; const TimeOut : byte) : boolean;
Function SerInpBlock_TO1(var location : type; const TimeOut : byte) : boolean;
Function SerInpBlock_TO2(var location : type; const TimeOut : byte) : boolean;
Function SerInpBlock_TO3(var location : type; const TimeOut : byte) : boolean;
Function SerInpBlock_TO4(var location : type; const TimeOut : byte) : boolean;
XMega
Function SerInpBlock_TO(Usart : tUSARTenum; var location : type; const TimeOut : byte) : boolean;

Function SerInpBlockP_TO(ptr : pointer; len : word; const TimeOut : byte) : boolean;
Function SerInpBlockP_TO1(ptr : pointer; len : word; const TimeOut : byte) : boolean;
Function SerInpBlockP_TO2(ptr : pointer; len : word;  const TimeOut : byte) : boolean;
Function SerInpBlockP_TO3(ptr : pointer; len : word; const TimeOut : byte) : boolean;
Function SerInpBlockP_TO4(ptr : pointer; len : word; const TimeOut : byte) : boolean;
XMega
Function SerInpBlockP_TO(Usart : tUSARTenum; ptr : pointer; len : word; const TimeOut : byte) : boolean;
```

Als Ziel der Operation ist nur das RAM zulässig, EEprom ist nicht möglich. Der Typ des Ziels (location) im RAM wird nur zur Berechnung der Block Grösse benötigt. Die eigentliche Struktur wird nicht beachtet.

3.11.1.3 Handshake Betrieb

Handshake für Tx (DTR)

Manchmal steuert eine externe Einheit den Datenfluss "SerOut" der seriellen Schnittstelle. Das ist kein Problem, solange ohne TxBuffer und damit ohne Tx-Interrupts gearbeitet wird. Mit TxBuffer wird das zum Problem da die Applikation zwar das DTR-Signal erkennen und den Tx-Interrupt des UARTS abschalten kann, aber dann kann es schon spät sein und der Empfänger wird überfahren. Ausserdem muss die Anwendung zu jeder Zeit das DTR Signal pollen, was in der Realität nicht möglich ist.

Deshalb wurde speziell für den Tx-Buffer Betrieb ein DTR Handshake implementiert. Deaktiviert das angeschlossene Gerät sein DTR Signal, stoppt der Tx-Buffer Treiber sofort den Sendebetrieb. Eine erneute Freigabe des Sendens erfolgt ***automatisch*** im SysTick wenn die Stop Bedingung entfallen ist. Dazu muss der SysTick importiert sein. Wenn das nicht der Fall ist, dann muss dies allerdings die Applikation vornehmen. Das kann z.B. dadurch erfolgen, dass der DTR Pin ein INT-Pin ist, der bei der Flanke von Stop nach Freigabe einen Interrupt erzeugt. In dieser Interrupt Service Routine kann jetzt die erneute Freigabe erfolgen. In den meisten Fällen reicht es jedoch aus, wenn in der Hauptprogramm Schleife der Wechsel von Sperrung nach Freigabe (DTR-Pin Pegel Wechsel) erfasst wird und hier die erneute Freigabe erfolgt.

Die DTR-Option der seriellen Schnittstelle muss definiert werden mit:

```
Define   SerPortDTR     = PinC, 2, Positive;   // first serport
         SerPortDTR1    = PinC, 2, Positive;   // first serport
         SerPortDTR2    = PinC, 3, Negative;   // second serport if available
         SerPortDTR3    = PinC, 4, Positive;   // third serport if vailable
         SerPortDTR4    = PinC, 5, Negative;   // fourth serport if available
XMega
Define   SerPortDTRC0 = PinC, 2, Positive;    // first serport
         SerPortDTRC1 = PinC, 3, Negative;    // second serport if available
         SerPortDTRD0 = PinC, 4, Positive;    // third serport if available
         SerPortDTRD1 = PinC, 5, Negative;    // fourth serport if available
```

Mit dem Define wird das DTR Handshake importiert und gleichzeitig der zuständige
Port Pin festgelegt. Die Pegel Angabe bedeutet Freigabe bei "Pegel".
Für die Freigabe bzw. Aufhebung der Sperrung durch die Applikation sind diese Prozeduren implementiert:

```
Procedure SerPort_Send;    oder    Procedure SerPort_Send1;
Procedure SerPort_Send2;
Procedure SerPort_Send3;
Procedure SerPort_Send4;
XMega
Procedure SerPort_Send (Usart : tUSARTenum);
```

Handshake für Rx (DSR)
In umgekehrter Richtung, d.h. beim Empfang kann auch das Problem des „überfahrens“ des RxBuffers entstehen. Bei reinem Text Transfer kann das XON/XOFF Protokoll verwendet werden. Das ist aber bei binären Daten nicht möglich. Deshalb wird hier ebenfalls ein Hardware Handshake verwendet. Der Receiver signalisiert damit dem Sender, dass sein Buffer voll ist oder dass er wieder empfangsbereit ist. Das Stop erfolgt dabei bei ca. ¾ Buffer Füllung, das ready wenn der Empfänger den Buffer auf ¼ geleert hat

Deshalb wurde speziell für den Rx-Buffer Betrieb ein DSR Handshake implementiert. Besteht beim Rx-Buffer die Gefahr des Überlaufs, deaktiviert der Treiber sein DSR Signal und stoppt damit den Sender. Eine erneute Freigabe des Sendens erfolgt automatisch, wenn wieder genug Platz im Rx-Buffer ist.

Die DSR-Option der seriellen Schnittstelle muss definiert werden mit:

```
Define   SerPortDSR     = PortC, 2, Positive;   // first serport
         SerPortDSR1    = PortC, 2, Positive;   // first serport
         SerPortDSR2    = PortC, 3, Negative;   // second serport if available
         SerPortDSR3    = PortC, 4, Positive;   // third serport if available
         SerPortDSR4    = PortC, 5, Negative;   // fourth serport if available
XMega
Define   SerPortDSRC0 = PortC, 2, Positive;    // first serport
         SerPortDSRC1 = PortC, 3, Negative;    // second serport if available
         SerPortDSRD0 = PortC, 4, Positive;    // third serport if available
         SerPortDSRD1 = PortC, 5, Negative;    // fourth serport if available
```

Mit dem Define wird das DSR Handshake importiert und gleichzeitig der zuständige Port Pin festgelegt. Die Pegel Angabe bedeutet Freigabe bei "Pegel".

Die Freigabe und Sperrung erfolgt automatisch, es ist keinerlei Support Funktion vorhanden.

Handshake für Rx (XON/XOFF)
Als Alternative zum Leitungs-HandShake (DSR) kann das Problem des „überfahrens“ des RxBuffers auch per Software mit Steuerzeichen gelöst werden. Dabei ist das sog. XON/XOFF Protokoll nur bei reinem Text Transfer anwendbar. Übertragung binärer Daten ist nicht möglich. Der Receiver signalisiert dem Sender mit einem „XOFF“ (#19), dass sein Buffer voll ist oder dass er wieder empfangsbereit ist mit dem „XON’ (#17). Das Stop erfolgt dabei wie beim DSR-Handshake bei ca. ¾ Buffer Füllung, das ready wenn der Empfänger den Buffer auf ¼ geleert hat
Das XON/XOFF wurde speziell für den Rx-Buffer Betrieb implementiert. Besteht beim Rx-Buffer die Gefahr des Überlaufs, schickt der Treiber das XOFF Zeichen und stoppt damit den Sender. Eine erneute Freigabe des Sendens erfolgt automatisch, wenn wieder genug Platz im Rx-Buffer ist mit dem Senden des XON Zeichens.

Die XON/XOFF-Option der seriellen Schnittstelle (Rx) muss definiert werden mit:

```
Define   RxBuffer     = 20, iData, XON;   // fist serport
         RxBuffer1    = 20, iData, XON;   // fist serport
         RxBuffer2    = 30, iData, XON;   // second serport if available
         RxBuffer3    = 20, iData, XON;   // third serport
         RxBuffer4    = 30, iData, XON;   // fourth serport if available
```

XMega

```
Define    RxBufferC0  = 20, iData, XON;   // fist serport
          RxBufferC1  = 30, iData, XON;   // second serport if available
          RxBufferD0  = 20, iData, XON;   // third serport
          RxBufferD1  = 30, iData, XON;   // fourth serport if available
```

Mit dem Define wird das XON/XOFF Handshake importiert.

Die Freigabe und Sperrung erfolgt automatisch, es ist keine Support Funktion notwendig.
Wenn **XON/XOFF** oder **DSR Handshake** importiert ist, dann kann die Applikation auch über die automatische Sperrung bzw. Freigabe des SerPort Receive informiert werden. Dazu sind zwei Callback Funktionen implementiert:

Procedure *onSerRxStopped;*
Das System ruft diese Funktion wird immer dann auf, wenn der SerPort Treiber den RxBuffer durch senden eines XOFF oder aktivieren der DSR Handshake Leitung sperrt.

Procedure *onSerRxResumed;*
Diese Funktion wird immer dann aufgerufen, der SerPort Treiber den RxBuffer durch senden eines XON oder de-aktivieren der DSR Handshake Leitung wieder freigibt

Implementierte Funktionen für die 4 möglichen SerPorts:

```
Procedure onSerRxStopped;          //SerPort
Procedure onSerRxResumed;          //SerPort
Procedure onSerRxStopped2;         //SerPort2
Procedure onSerRxResumed2;         //SerPort2
Procedure onSerRxStopped3;         //SerPort3
Procedure onSerRxResumed3;         //SerPort3
Procedure onSerRxStopped4;         //SerPort4
Procedure onSerRxResumed4;         //SerPort4
```

XMega

```
Procedure onSerRxStoppedC0;        //SerPortC0
Procedure onSerRxResumedC0;        //SerPortC0
Procedure onSerRxStoppedC1;        //SerPortC1
Procedure onSerRxResumedC1;        //SerPortC1
Procedure onSerRxStoppedD0;        //SerPortD0
Procedure onSerRxResumedD0;        //SerPortD0
Procedure onSerRxStoppedD1;        //SerPortD1
Procedure onSerRxResumedD1;        //SerPortD1
Procedure onSerRxStoppedE0;        //SerPortE0
Procedure onSerRxResumedE0;        //SerPortE0
Procedure onSerRxStoppedE1;        //SerPortE1
Procedure onSerRxResumedE1;        //SerPortE1
Procedure onSerRxStoppedF0;        //SerPortF0
Procedure onSerRxResumedF0;        //SerPortF0
Procedure onSerRxStoppedF1;        //SerPortF1
Procedure onSerRxResumedF1;        //SerPortF1
```

Achtung: wie bei allen CallBack Funktionen gilt auch hier dass der CallBack aus einem Interrupt heraus erfolgt und deshalb die gleichen Regeln wie beim Interrupt gelten: keine grösseren Operationen wie z.B. Float um eine zu lange Sperrung des globalen Interrupts zu vermeiden. Evtl. Registersicherung nicht vergessen!

3.11.1.4 RS485

Wird der USART/UART mit RS485 Treibern im halb-duplex Mode betrieben, so ist es sehr oft notwendig, den Leitungstreiber zu steuern (Tristate und/oder Richtung). Dabei darf der Treiber nur umgesteuert werden, wenn das letzte Bit des letzten Bytes den UART verlassen hat. Dazu wurde ein Define und eine Prozedur implementiert:

```
Define  TxBuffer    = 8, iData;
        SerCtrl     = PortD, 2, positive;     {control line for RS485 driver}
        //SerCtrl1  = PortD, 2, positive;     {control line for RS485 driver}
        //SerCtrl2  = PortD, 2, positive;     {control line for RS485 driver}
        //SerCtrl3  = PortD, 2, positive;     {control line for RS485 driver}
        //SerCtrl4  = PortD, 2, positive;     {control line for RS485 driver}
XMega
Define  TxBuffer    = 8, iData;
        SerCtrlC0   = PortD, 2, positive;     {control line for RS485 driver}
        //SerCtrlC1 = PortD, 2, positive;     {control line for RS485 driver}
        //SerCtrlD0 = PortD, 2, positive;     {control line for RS485 driver}
        //SerCtrlD1 = PortD, 2, positive;     {control line for RS485 driver}
```

Da eine RS485 Unterstützung nur bei TxBuffer Betrieb sinnvoll ist, muss der TxBuffer definiert sein. Das Define "SerCtrl" bestimmt das Steuerport, den Steuerpin und die Polarität für enable.

```
Procedure Ser_Enable (const ena : boolean);
Procedure Ser_Enable1 (const ena : boolean);
Procedure Ser_Enable2 (const ena : boolean);
Procedure Ser_Enable3 (const ena : boolean);
Procedure Ser_Enable4 (const ena : boolean);
XMega
Procedure SetSerEnable(Usart : tUSARTenum; const ena : boolean);
```

Wird zur Steuerung des Leitungstreibers benutzt. Mit ena = true wird der Treiber sofort freigeschaltet. Anschliessend kann gesendet werden. Mit ena = false wird das System informiert, dass wenn das letzte Byte aus dem TxBuffer gesendet wurde, der Leitungstreiber abgeschaltet werden soll. Eine Statusabfrage des Leitungs-Treibers ist nicht vorgesehen. Dazu kann das zuständige Port Bit zurückgelesen werden.

```
Ser_Enable (true);
write (SerOut, 'hello');
Ser_Enable (false);
XMega
SetSerEnable(UsartC0, true);
write (SerOutC0, 'hello');
SetSerEnable(UsartC0, false);
```

3.11.1.5 TxComplete Callback

Wird nur die Info gebraucht, dass der TxBuffer komplett leer ist und auch der UART alle Bits gesendet hat, dann kann der SerPort Treiber die Applikation darüber informieren. Dazu ist das Define/Import ähnlich bzw. alternativ zum RS485 Mode aufzubauen. Es wird das SerCtrl und TxBuffer Define und die Funktion Ser_Enable benötigt. Diese Info erfolgt über eine Call Back Funktion. RS485 Treiber sind dabei nicht möglich!

```
Define  TxBuffer    = 16, iData;
        SerCtrl     = onSerTxComplete1;     {necessary callback function}
        //SerCtrl2  = onSerTxComplete2;     {necessary callback function}
        //SerCtrl3  = onSerTxComplete3;     {necessary callback function}
        //SerCtrl4  = onSerTxComplete4;     {necessary callback function}
```

XMega

```
Define  TxBuffer     = 32, iData;
        SerCtrlC0    = onSerTxComplete_C0; {necessary callback function}
        //SerCtrlC1  = onSerTxComplete_C1; {necessary callback function}
        //SerCtrlD0  = onSerTxComplete_D0; {necessary callback function}
        //SerCtrlD1  = onSerTxComplete_C0; {necessary callback function}
```

Die jeweilige Callback Funktion muss von der Applikation bereitgestellt werden:

```
Procedure onSerTxCompleteXX;
```

Support Funktionen:

```
Procedure Ser_Enable (const ena : boolean);
Procedure Ser_Enable1 (const ena : boolean);
Procedure Ser_Enable2 (const ena : boolean);
Procedure Ser_Enable3 (const ena : boolean);
Procedure Ser_Enable4 (const ena : boolean);
```

XMega

```
Procedure SetSerEnable(Usart : tUSARTenum; const ena : boolean);
```

Wird zur Steuerung des Callbacks benutzt. Mit ena = true wird der das Verfahren gestartet. Anschliessend kann gesendet werden. Mit ena = false wird das System informiert, dass wenn das letzte Byte aus dem TxBuffer gesendet wurde, die Callback Funktion aufgerufen werden soll..

```
Ser_Enable (true);
write (SerOut, 'hello');
Ser_Enable (false);
```

XMega

```
SetSerEnable(UsartC0, true);
write (SerOutC0, 'hello');
SetSerEnable(UsartC0, false);
```

Muss zusätzlich noch ein RS485 Treiber gesteuert werden, so muss die Applikation diesen beim SerEnable selbst freigeben und im Callback wieder abschalten.

Achtung: wie bei allen CallBack Funktionen gilt auch hier dass der CallBack aus einem Interrupt heraus erfolgt und deshalb die gleichen Regeln wie beim Interrupt gelten: keine grösseren Operationen wie z.B. Float um eine zu lange Sperrung des globalen Interrupts zu vermeiden. Evtl. Registersicherung nicht vergessen!

3.11.2 Portumschaltung

Für AVRs mit mehreren UARTs kann dynamisch zwischen den SerPorts umgeschaltet werden. Wenn richtig importiert, gibt es dazu das Byte **SerPortSelect**. Es steuert die Wahl der Schnittstelle für die Funktionen SerOut, SerInp, SerStat und FlushBuffer. Wenn z.B. SerPortSelect den Wert 1 hat gehen alle SerOut Calls nach SerPort2, ebenso SerInp nach SerInp2 und SerStat nach SerStat2.

Diese Umschaltung betrifft natürlich auch die mit den SerPorts verbundenen Funktionen wie Write, WriteLn und Read etc.

Grundsätzlich gilt: wird hier SerOut, SerInp, SerStat etc. benutzt bestimmt SerPortSelect das entsprechende Port. Wird gezielt ein Port angesprochen , z.B. SerInp2, dann wird auch immer dieses Port benutzt.

Um diese dynamische Umschaltung zu ermöglichen, müssen alle beteiligten SerPorts importiert sein. Dann kann auch "SerPortSelect" importiert werden.

```
Device = mega2560, VCC = 5;
Import SysTick, SerPort, SerPort2, SerPort3;
From SerPort import SerPortSelect;
```

Damit steht das Byte "SerPortSelect" zur Verfügung.

```
SerPortSelect:= 1;      // switch to serport2
SerOut ('x');           // write 'x' to serport2
if SerStat then         // check serport2
  ch:= SerInp;          // read from Serport2
endif;

SerPortSelect:= 2;      // switch to serport3
SerOut ('x');           // write 'x' to serport3
if SerStat then         // check serport3
  ch:= SerInp;          // read from Serport3
endif;

SerPortSelect:= 0;      // switch to serport
SerOut ('x');           // write 'x' to serport
if SerStat then         // check serport
  ch:= SerInp;          // read from Serport
endif;
SerPortSelect:= 0;      // switch to serport
FlushBuffer (RxBuffer);// clear the selected RxBuffer
```

XMega

Hier ist die Variable *SerPortSelect* kein Byte sondern vom Typ tUSARTenum

```
SerPortSelect:= UsartC1;      // switch to serportC1
SerOut ('x');                 // write 'x' to serportC1
if SerStat then               // check serportC1
  ch:= SerInp;                // read from SerportC1
endif;

SerPortSelect:= UsartD0;      // switch to serportD0
SerOut ('x');                 // write 'x' to serportD0
if SerStat then               // check serportD0
  ch:= SerInp;                // read from SerportD0
endif;

SerPortSelect:= UsartC0;      // switch to serportC0
SerOut ('x');                 // write 'x' to serportC0
if SerStat then               // check serportC0
  ch:= SerInp;                // read from SerportC0
endif;
```

Diese Umschaltung betrifft natürlich auch die mit den SerPorts verbundenen Funktionen wie Write, WriteLn und Read etc.

Grundsätzlich gilt: wird hier SerOut, SerInp, SerStat etc. benutzt bestimmt SerPortSelect das entsprechende Port. Wird gezielt ein Port angesprochen , z.B. SerInpC0, dann wird auch immer dieses Port benutzt.

3.11.3 UART enable und disable für XMegas

In speziellen Fällen kann es notwendig sein entweder den Rx Teil, den Tx Teil oder beide eines UARTs abzuschalten und wieder freizugeben. Dafür vorgesehen sind diese Funktionen:

```
procedure UartEnableRxXX( Ena : boolean);    // Rx hardware enable/disable
procedure UartEnableTxXX( Ena : boolean);    // Tx hardware enable/disable
```

wobei ***XX*** für C0, C1, D0, D1 etc. steht.

3.11.4 IRDA für XMegas UARTs (IRcom)

Die XMega UARTs bieten einen IRDA Support. IRDA = **I**nfra**R**ed**D**ata**A**ssociation. Mit IRDA können Daten über Infrarot übertragen werden. Es gibt dafür mehrere low-Level Protokolle und darüber auch mehrere Schichten Protokoll Stack. Diese Implementation benutzt nur die einfachste und unterste Stufe von IRDA. Dieses SIR Protokoll (**S**erial**I**nfra**R**ed) ist ein asynchrones Protokoll wie es auch von UARTs verwendet wird. Allerdings werden die Impulsbreiten hier auf 1/6 begrenzt um Strom zu sparen.

Das SIR arbeitet grundsätzlich mit 115.2kBaud, 1 Startbit, 8 Databits, 1 Stopbit und no parity.
Alle diese Voraussetzungen unterstützt die IRDA Hardware in den XMega UARTs. Allerdings kann diese Hardware nur einmal im XMega an einen beliebigen UART angebunden werden.
IRDA/SIR arbeitet exakt wie ein MAX232 ausser dass die Daten nicht über Leitungen geschickt werden sondern über unsichtbares Licht. Dadurch gibt es die Einschränkung dass eine solche Verbindung nur halb-duplex ist, d.h. es kann immer nur ein Teilnehmer zu einer Zeit senden. Normalerweise ist die nutzbare Entfernung bei IRDA >1m. Mit entsprechender höherer Sende Leistung sind auf Distanzen >10m erreichbar.

Durch den Hardware Support der UARTs ist das IRDA dem RC5 weit überlegen! Weiterhin können im Gegensatz zu RS232 mehrere Teilnehmer vorhanden sein. Ein einfaches Bus System.

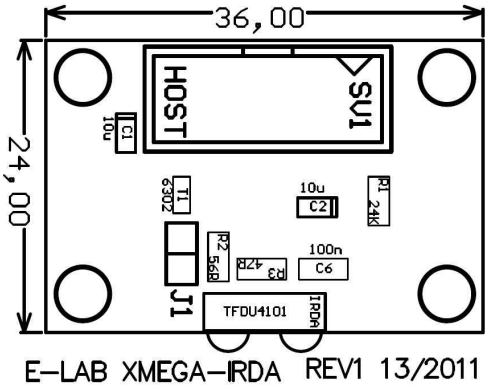
Soll ein IRDA Transceiver betrieben werden, dann muss das dem Treiber beim Define mitgeteilt werden:
Define *SerPortC1 = 115200, IRDA;*

Es können fast alle SerPort Funktionen hier auch benutzt werden.

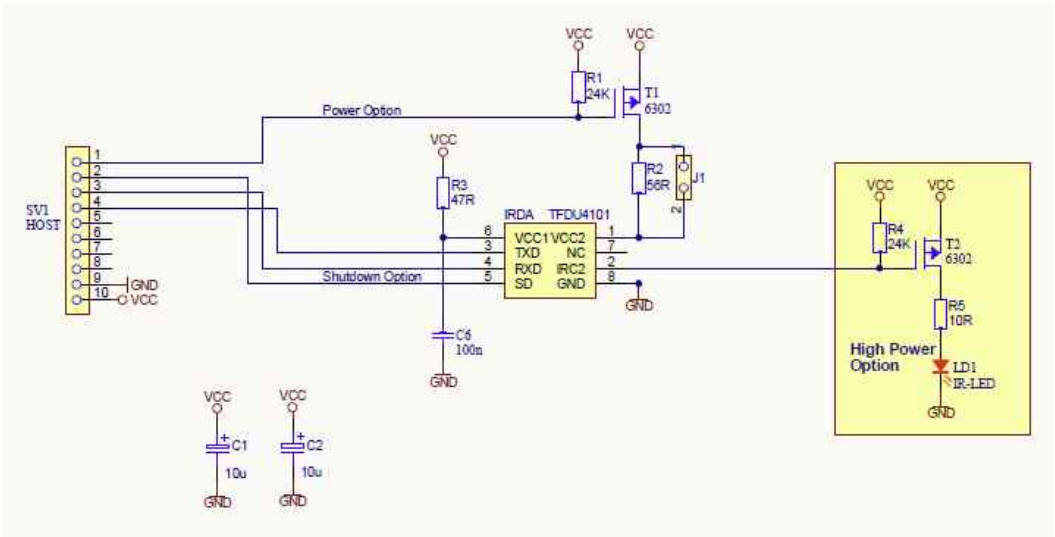
Beim bi-direktionalen Betrieb ist aber nur Halb-Duplex möglich. D.h. dass nur ein Teilnehmer zu einer Zeit Senden kann. Weiterhin empfangen die üblichen IRDA Transceiver auch ihre eigenen Sende Daten, was natürlich unerwünscht ist. Deshalb muss beim Transceiver Betrieb dafür Sorge getragen werden, dass der eigene Rx beim Senden abgeschaltet wird. Ist das Telegramm komplett raus, dann muss der Receiver wieder aktiviert werden. Dazu sind zwei Funktionen vorhanden:

Procedure *IRDAtxStartXX;*
Procedure *IRDAtxWaitIdleXX;*

Das ***XX*** steht dabei für den IRDA UART (C0..F1).



Programm Beispiel: zwei Beispiele befinden sich im Verzeichnis
..\E-Lab\AVRco\Demos\XMega_IRDA



3.11.5 Erweiterte Funktionen für Controller mit USART

Bei den Controllern mit USART gibt es im wesentlichen folgende Erweiterungen:

DatenBits : 5/6/7 oder 8 Bits. Die Funktion des Multiprozessor-Bit (Bit9) ist auch wesentlich erweitert worden
StopBits : 1 oder 2. Ist mit UART nur über das 9. bit machbar.
Parity : none/even/odd. Ist mit UART ebenfalls nur über das 9. bit möglich und muss dann per Software generiert werden .

3.11.5.1 Typen, Prozeduren und Funktionen

Besitzt eine AVR CPU einen USART, so werden diverse Typen und Funktionen vom AVRco System bereitgestellt, die es bei einem Standard UART so nicht gibt.

Typen

Zur besseren und eindeutigen Handhabung der zusätzliche Funktionen beim USART exportiert das System folgende Typen:

```
Type tParity      = (parNone, parEven, parOdd);
Type tDataBits    = (DataBit5, DataBit6, DataBit7, DataBit8);
Type  tStopBits   = (StopBit1, StopBit2);
```

Prozeduren und Funktionen

Bei den USARTs ist es jetzt problemlos möglich zur Laufzeit diverse Parameter zu ändern. Dazu dienen die folgenden Prozeduren:

```
Procedure SerStopBits (bits : tStopBits);                 // Stopbits (1/2)     COM1
Procedure SerStopBits1 (bits : tStopBits);                // Stopbits (1/2)     COM1
Procedure SerStopBits2 (bits : tStopBits);                // Stopbits (1/2)     COM2
Procedure SerStopBits3 (bits : tStopBits);                // Stopbits (1/2)     COM3
Procedure SerStopBits4 (bits : tStopBits);                // Stopbits (1/2)     COM4
Procedure SerDataBits (bits : tDataBits);                 // Databits (5/6/7/8) COM1
Procedure SerDataBits1 (bits : tDataBits);                // Databits (5/6/7/8) COM1
Procedure SerDataBits2 (bits : tDataBits);                // Databits (5/6/7/8) COM2
Procedure SerDataBits3 (bits : tDataBits);                // Databits (5/6/7/8) COM3
Procedure SerDataBits4 (bits : tDataBits);                // Databits (5/6/7/8) COM4
Procedure SerParity (par : tParity);                      // Parity (N/E/O)     COM1
Procedure SetParity1 (par : tParity);                     // Parity (N/E/O)     COM1
Procedure SerParity2 (par : tParity);                     // Parity (N/E/O)     COM2
Procedure SerParity3 (par : tParity);                     // Parity (N/E/O)     COM3
Procedure SerParity4 (par : tParity);                     // Parity (N/E/O)     COM4
Procedure SerBaud (var baud : word);                      // Baudrate var Parameter COM1
Procedure SerBaud (const baud : byte..longword);          // Baudrate const Parameter COM1
Procedure SerBaud1 (var baud : word);                     // Baudrate var Parameter COM1
Procedure SerBaud1 (const baud : byte..longword);         // Baudrate const Parameter COM1
Procedure SerBaud2 (var baud : word);                     // Baudrate var Parameter COM2
Procedure SerBaud2 (const baud : byte..longword);         // Baudrate var Parameter COM2
Procedure SerBaud3 (var baud : word);                     // Baudrate var Parameter COM3
Procedure SerBaud3 (const baud : byte..longword);         // Baudrate var Parameter COM3
Procedure SerBaud4 (var baud : word);                     // Baudrate var Parameter COM4
Procedure SerBaud4 (const baud : byte..longword);         // Baudrate var Parameter COM4
XMega
Procedure SetSerStopBits(Usart : tUSARTenum; bits : tStopBits);  // Stopbits (1/2)
Procedure SetSerDataBits(Usart : tUSARTenum; bits : tDataBits); // Databits (5/6/7/8)
Procedure SetSetParity(Usart : tUSARTenum; par : tParity);       // Parity (N/E/O)
Procedure SetSerBaud(Usart : tUSARTenum; const baud : byte..longword);  // Baudrate const Parameter
```

Bitte beachten, dass diese Parameter Änderungen zur Laufzeit u.U. 1 bis 2 falsche Zeichen beim Empfang auslösen können, wenn während der Änderung gerade Zeichen empfangen werden.

3.11.6 SLIP packet orientiertes Protokoll

Netzwerke arbeiten grundsätzlich mit sog. Packeten oder Frames. Das ist ein Datenblock der durch eine Anfangs und Ende Kennung begrenzt wird. Dies ist absolut notwendig um eine sichere Übertragung zu gewährleisten. Diese Frame Begrenzung kann ganz unterschiedlich ausgeführt sein, z.B. durch eine Pause mit einer bestimmten minimalen Länge wie bei ModBUS RTU oder auch bestimmte Steuerbytes wenn die Daten ansonsten immer nur ASCII Zeichen (A..Z, a..z, 0..9) sind (ModBUS ASCII).

Bei komplexeren Protokollen wie CAN, Ethernet, I2C/TWI etc. ist die Frame Begrenzung explizit schon in der Treiber Hardware implementiert. Ohne dieses Framing ist es ausserordentlich problematisch auf der Empfängerseite einen einkommenden Frame eindeutig zu identifizieren. Aber auch bei den Timeout gesteuerten Verfahren ist das ganze nicht trivial. Auf beiden Seiten muss mit Timern und Interrrupts gearbeitet werden um ungewollte Lücken/Timeouts zu vermeiden. Bei den ASCII Protokollen hat man wiederum den grossen Aufwand auf beiden Seiten binäre Daten in ASCII Zeichen (z.B. HEX) zu wandeln und wieder zurück zu konvertieren. Speziell bei bit-seriellen Schnittstellen wie UARTs und SPI stellt sich das Problem eindeutige Frames zu erstellen und zu identifizieren. Für diese Zwecke gibt es seit vielen Jahren das SLIP Verfahren.

Serial **L**ine **I**nternet **P**rotocol

Das Wort “Internet“ kann man auch umdeuten als “Interchange“ oder „Interface“ weil SLIP eigentlich nur das Frame-Start und Frame-End Handling wirklich definiert. Alles andere muss von einem übergeordneten Level aus gehandhabt werden.

SLIP stellt also sicher, dass ein Packet eindeutig mit seiner Start Bedingung erkannt wird und die darin enthaltenen Daten bis zur Ende-Kennung der Applikation zur Verfügung gestellt werden. Die darin enthaltenen Daten können beliebig sein und werden weder verändert noch analysiert oder ausgewertet.

Das AVRco System stellt eine SLIP Implementation zur Verfügung und zwar für die seriellen Schnittstellen 1..4 soweit in der CPU vorhanden. Dazu sind jeweils eine Sende Funktion und eine Empfangs Funktion vorhanden.

```
Procedure SerOutSLIP(src : pointer; count : word);      // UART1
Procedure SerOutSLIP1(src : pointer; count : word);     // UART1
Procedure SerOutSLIP2(src : pointer; count : word);     // UART2
Procedure SerOutSLIP3(src : pointer; count : word);     // UART3
Procedure SerOutSLIP4(src : pointer; count : word);     // UART4
```

XMega

```
Procedure SerOutSLIPC0(src : pointer; count : word);     // UARTC0
Procedure SerOutSLIPC1(src : pointer; count : word);     // UARTC1
Procedure SerOutSLIPD0(src : pointer; count : word);     // UARTD0
Procedure SerOutSLIPD1(src : pointer; count : word);     // UARTD1
Procedure SerOutSLIPE0(src : pointer; count : word);     // UARTE0
Procedure SerOutSLIPE1(src : pointer; count : word);     // UARTE1
Procedure SerOutSLIPF0(src : pointer; count : word);     // UARTF0
Procedure SerOutSLIPF1(src : pointer; count : word);     // UARTF1
```

Diese Prozeduren senden ein Daten Packet über die gewählte serielle Schnittstelle. Mit ***src*** wird die Quelle der Daten als Pointer angegeben. Der Parameter ***count*** bestimmt die Anzahl der zu sendenden Daten. Die Prozedur kehrt zurück, wenn sie alle Daten in den Sende Buffer des UART geschrieben hat. Der Parameter count darf nicht 0 sein.

```
Function SerInpSLIP(dst : pointer; tmo : byte; count : word) : word;      // UART1
Function SerInpSLIP1(dst : pointer; tmo : byte; count : word) : word;     // UART1
Function SerInpSLIP2(dst : pointer; tmo : byte; count : word) : word;     // UART2
Function SerInpSLIP3(dst : pointer; tmo : byte; count : word) : word;     // UART3
Function SerInpSLIP4(dst : pointer; tmo : byte; count : word) : word;     // UART4
```

XMega

```
Function SerInpSLIPC0(dst : pointer; tmo : byte; count : word) : word;      // UARTC0
Function SerInpSLIPC1(dst : pointer; tmo : byte; count : word) : word;      // UARTC1
Function SerInpSLIPD0(dst : pointer; tmo : byte; count : word) : word;      // UARTD0
Function SerInpSLIPD1(dst : pointer; tmo : byte; count : word) : word;      // UARTD1
Function SerInpSLIPE0(dst : pointer; tmo : byte; count : word) : word;      // UARTE0
```

Function *SerInpSLIPE1(dst : pointer; tmo : byte; count : word) : word;* *// UARTE1*
Function *SerInpSLIPF0(dst : pointer; tmo : byte; count : word) : word;* *// UARTF0*
Function *SerInpSLIPF1(dst : pointer; tmo : byte; count : word) : word;* *// UARTF1*

Diese Funktionen warten solange auf ein Daten Packet von der gewählten serielle Schnittstelle bis dieses komplett empfangen wurde oder ein Timeout aufgetreten ist. Mit ***dst*** wird das Ziel der Daten als Pointer angegeben. Der Parameter ***tmo*** bestimmt das Timeout in Systicks für jedes einzelne zu empfangende Byte. Der Parameter ***count*** bestimmt die maximale Anzahl der zu empfangenden Daten. Sollte das Packet grösser sein als dieser Wert kehrt die Funktion mit einem Fehler zurück, Result = 0. Sollte ein Timeout auftreten kehrt die Funktion ebenfalls mit einer 0 zurück. Im Erfolgsfall enthält das Ergebnis die Packetgrösse in Bytes.

Zur korrekten Funktion dieser Treiber muss das zugehörige SerPort korrekt importiert und definiert werden. Weiterhin muss das Timeout für dieses Port aktiviert sein. Zu beachten ist weiterhin, dass ein Serport Buffer nur max. 254 Bytes gross sein kann. Das bedeutet dass ein Packet max. 252 Datenbytes haben kann, da ja auch noch die beiden Frame Bytes dazu kommen. Durch den SLIP Algorithmus müssen bestimmte Datenbytes durch zwei Ersatz Bytes ersetzt werden. Das bedeutet im schlimmsten Fall, dass mit einem 254 bytes grossen Rx/Tx Buffer nur 126 effektive Bytes übertragen werden können.
Beim Senden spielt das keine grosse Rolle, da der Treiber solange wartet bis er alle Bytes in den TxBuffer loswerden konnte.

Beim Empfangen grosser Packete wird es hierbei problematischer. Wenn die Applikation nicht kontinuierlich die zugehörige RxStat Funktion abfragt und im Erfolgsfall sofort SerInpSLIP aufruft, weist der RxBuffer Interrupt Treiber überzählige Rx-Bytes ab und dieser Frame wird fehlerhaft werden. Keine Probleme sind zu erwarten, wenn zumindest der RxBuffer doppelt so gross ist wie der grösste Rx-Frame.

RS485 Line Driver
Wurde für das zuständige SerPort der RS485 Modus definiert, so handhabt der SerOutSLIP Treiber dieses automatisch ohne weiteres zutun der Applikation.

SLIP Beispiel:

```
program SLIPtest;
Device = mega128, VCC = 5;
Import SysTick, SerPort;
Define
  ProcClock              = 16000000;       {Hertz}
  SysTick                = 10;             {msec}
  StackSize              = $040, iData;
  FrameSize              = $060, iData;
  SerPort                = 115200, Stop2, TimeOut; {Baud, StopBits|Parity}
  RxBuffer               = 64, iData;
  TxBuffer               = 64, iData;
Implementation
{$IDATA}
Var
  Test    : word;
  ptrRx   : pointer;
  ptrTx   : pointer;
  BuffRx  : array[0..15] of byte;
  BuffTx  : array[0..15] of byte;
Begin  //Main
  EnableInts;
  BuffTx[2]:= 192;
  BuffTx[6]:= 219;
  ptrTx:= @BuffTx;
  ptrRx:= @BuffRx;
  SerOutSLIP(ptrTx, 8);
  mDelay(20);
  // if RxStat then …
  test:= SerInpSLIP(ptrRx, 10, 8);
end SLIPtest.
```

3.12 Serial Network LAN

Nicht für XMegas, statt dessen SlipPorts benutzen.
Distributed Intelligence ist nicht nur ein Schlagwort. Verteilte Rechenleistung innerhalb eines Systems bringt enorme Vorteile: weniger Verkabelung, mehr Rechenleistung pro Zeiteinheit und einfachere Programme. Setzt man ein Rechner resp. CPU an den Ort, wo er gebraucht wird und gibt es in einem System mehrere solcher Orte, so muss man zwangsweise eine Kommunikation zwischen den einzelnen Programmen/Rechnern/CPUs aufbauen.

Dieser Datenaustausch erfolgt normalerweise über ein Local Area Network (LAN). Es gibt diverse Möglichkeiten ein solches Netzwerk zu realisieren. Die meisten davon benötigen einen speziellen Netzwerk Controller (Ethernet, ArcNet, InterBus-S, ProfiBus, CAN-BUS etc.) oder zumindest eine CPU, in welcher der Controller enthalten ist (z.B. CAN-Bus). Diese Controller bieten meistens eine sehr komfortable Anbindung der CPU ans Netz.

Leider fehlt oft der Platz für einen zusätzlichen Controller Baustein oder der Preis ist nicht “drin”. CPUs mit eingebauten Controllern sind z.Zt. noch etwas rar bzw. die Gehäuse Form (zu gross, zu viele Pins) ist nicht geeignet.

Da die meisten CPUs inzwischen eine serielle Schnittstelle besitzen, bietet es sich an, diese für Netzwerk Zwecke einzusetzen. Alte Mainframes haben das schon vor vielen Jahren praktiziert.

Alle in Frage kommenden Atmel AVR-CPUs haben einen UART. Ein besonderes Feature ist, dass die Übertragung neun Datenbits haben kann. Dieses neunte Bit bietet sich an als Kennung für die Adresse eines Telegrams. Manche 8051 Derivate haben das gleiche Feature.

Die vorliegende Implementation ist in erster Linie für solche CPU-CPU Verbindungen gedacht. Die Strategie besteht aus einem Single-Master und Multi-Slave System. Zur sicheren Kommunikation wird ein entsprechendes Protokoll gefahren.

Prinzipielles
Es wurde ein Single-Master / Multi-Slave System implementiert, das, wie der Name sagt, grundsätzlich aus einem Master und mindestens einem Slave besteht.

Der Master kontrolliert alle angeschlossenen Slaves. Kein Slave sollte von sich aus die BUS-Kontrolle übernehmen, obwohl das jederzeit möglich ist. Jeglicher Transfer, ob vom oder zum Slave wird vom Master aus initiiert. Damit entfallen die sonst erheblichen Probleme mit Kollisionen, Prioritäten etc.

Netzwerke arbeiten allgemein mit Telegrammen, auch Frames genannt. Durch die Verwendung von Frames wird zwar ein gewisser Overhead betrieben, im Gegenzug dafür aber die Betriebssicherheit erheblich gesteigert. Um weiteren Problemen aus dem Weg zu gehen, wurde auf den Full Duplex Transfer verzichtet. Damit lassen sich die preiswerten und extrem sicheren RS485 Treiber verwenden.

Im Halb-Duplex Betrieb kann immer nur ein Teilnehmer senden, alle anderen hören mit. Der Master fordert z.B. ein Frame von einem Slave. Dazu schickt er einen Frame an den Slave mit dem entsprechenden Inhalt (Adresse, Daten, Kommandos). Der Slave erkennt seine Adresse und stellt in seinem Frame-Buffer den Frame bereit. Wenn das Anwender Programm im Slave eine Antwort an den Master schicken muss (Vereinbarungssache), schickt es einen Frame los, dessen Adresse seine eigene ist. Alle anderen Slaves ignorieren diesen Frame. Der Master stellt den Frame in seinem Buffer bereit und informiert die Application darüber.

Was der Master mit seinem Frame beim Slave bezwecken will und ob der Slave eine Antwort schicken muss liegt in der Hand des Programmierers. Erwartet der Master bzw. dessen Programm eine Antwort vom Slave, sollte der Slave diese umgehend abschicken, da der Master normalerweise keine weiteren Frames mehr schickt, bis die Antwort da ist, um eine BUS Kollision zu vermeiden.

Funktionsprinzip
Der **Master** empfängt alle Frames die auf dem BUS abgeschickt werden und stellt sie dem Anwendungsprogramm zur Verfügung. Eine Auswertung irgendwelcher Art im Master erfolgt nicht.

Der **Slave** hört zwar immer auf dem BUS mit, fängt aber nur Frames ab bzw. liest sie ein, wenn die Adresse innerhalb des Frames identisch ist mit seiner eigenen (Node Adr). Eine Auswertung des Frames innerhalb des Slave Devices erfolgt nicht. Es bleibt dem Anwendungsprogramm überlassen, dies zu tun.

Der Slave schickt auch kein Acknowledge oder eine Antwort an den Master. Dies ist Sache der Anwendung. Eine Ausnahme davon ist, wenn im Slave mit **LANrxAutoAck(true)** ein automatisches ACK freigegeben. Hierbei schickt der Slave automatisch ein kurzes Telegram an den Master ohne Inhalt, d.h. nur Slave-Adresse, FrameLength 0 und ggf. eine Checksumme. Das geschieht allerdings nur, wenn der Slave den Frame korrekt empfangen hat.

Die Slave Adressen müssen > 0 sein. Die Adresse “0” gilt als Broadcast Adresse für alle Slaves. Nur der Master sollte ein Broadcast senden (Kollisionsgefahr). Alle Slaves werten das Telegramm aus, reagieren aber nicht darauf mit einem Antwortframe, auch das AutoAck ist in diesem Fall abgeschaltet.

Die aktuelle Datenlänge (Bytes) eines Frame kann von 0 Bytes bis LANframe –1 laufen. Die Adresse kann 8 oder 16 Bits sein (über Define eingestellt). Die Checksumme (über define eingestellt) kann nicht vorhanden, 8 Bit Check, 16 Bit Check oder 16 Bit CRC-Check sein.

Variabel ist dabei nur die aktuelle Datenlänge, Adresslänge und Checktype sind durch Defines festgelegt und während der Laufzeit nicht änderbar.

Aufbau eines Telegramms:

| ADDRESS | FRAMELENGTH | DATA, DATA ... | CHECKSUM |

Adresse:
Die Adresse kann ein Byte (1..255 Slaves) oder ein Wort (1..65535 Slaves) sein. Normalerweise reicht eine Byte Adresse aus, da die üblichen RS485 Bausteine im günstigsten Fall 255 Teilnehmer Lasten treiben können. Mit dem Define LANadr wird die Adressbreite eingestellt.

Jeder Slave sollte eine eigene Adresse haben, die im ganzen Netzwerk sonst nicht mehr vorkommt. Ausnahmen sind allerdings denkbar. Der Master hat keine eigene Adresse, da er alle Frames empfangen muss. Das bzw. die Adressbytes werden mit gesetztem 9. Bit verschickt. Bei 2 Byte Adressen wird das LoByte zuerst geschickt.

FrameLength:
Die Frame Länge wird mit define LANframe = nn eingestellt. Ist der Parameter nn < 256, wird ein Längenbyte übertragen, ist nn >= 256, werden 2 Längenbytes übertragen. Die aktuelle Framelänge kann zwischen 0 und nn bytes sein. Allerdings müssen alle BUS Teilnehmer die gleiche Längen Art (Byte/Word) besitzen, so dass die Frames bei allen identisch sind. In den Bereichen eines Bytes oder Words können die implementierten Buffergrössen allerdings variieren.

DATA:
Der Datenbereich innerhalb eines Frames wird durch das Längenbyte bzw. Wort angegeben. Es brauchen keine Daten im Frame sein, dann ist das Längenbyte/Wort auch null.

CHECKSUM:
Die Checksumme ist eine Setup Option. Sie wird über den kompletten Frame gebildet. Es kann eine 8bit, eine 16bit Checksumme, oder ein 16bit CRC gebildet werden.

Achtung:
Die Adressbreite, die max. FrameLänge und die Art der Checksumme muss zur Design Zeit mittels den Defines festgelegt werden. Ebenso der Modus Master/Slave. Eine Änderung zur Laufzeit ist nicht möglich. Die implementierten Frame Buffer Längen können in den einzelnen Teilnehmern variieren, aber nur soweit, wie die Framelänge dies zulässt. Entweder haben alle Teilnehmer eine Buffer Grösse kleiner 256 Bytes (FrameLength in einem Byte) oder grösser 255 Bytes (FrameLength in einem Wort).

Transfer
Der Datentransfer erfolgt grundsätzlich im Interrupt Verfahren. Damit wird sichergestellt, dass wenn ein Frame fertig zusammengestellt ist, dieser auch in kürzester Zeit sein Ziel erreicht. Wichtig ist, dass Interrupts nicht zu lange gesperrt bleiben. Das gilt vor allen dingen bei hohen Baudraten. Bei 9600Bd wird bei einem Transfer jede msec ein Receiver Interrupt ausgelöst, bei 56kBd alle 160usec.

Die eingestellte Baudrate muss selbstverständlich bei allen Beteiligten identisch sein. Sie braucht sich allerdings nicht an das herkömmliche Raster (9600, 19200 etc.) zu halten.

Steuerung
Ob nun ein Funkmodem, eine RS485 Leitung oder TTL Treiber zum Einsatz kommen, alle Arten haben die gleiche Eigenschaft, sie sind nicht full duplex. D.h. es kann entweder gerade nur Empfangen oder Gesendet werden. Aber niemals gleichzeitig. Daher muss jeder BUS Teilnehmer die Möglichkeit haben, seine Leitungstreiber abzuschalten und nur zuzuhören. Dies geschieht mit einem Port Pin, der durch Define LANctrl = Port, Bit; definiert wird. Die Ab- bzw. Umschaltung erfolgt durch das System selbst, das Anwendungsprogramm braucht sich darum nicht zu kümmern.

3.12.1 Implementation

Imports:
Das LANport muss, wie beim AVRco System üblich, durch eine Import Anweisung importiert werden

```
Import   LANport;
```

Defines

LANport
Festlegung des benutzten UARTs.

```
Define   LANport  = SerPort;      {SerPort2}
```

LANctrl
Festlegung des Steuer Ports und Pins für die Umschaltung des Leitungstreibers. Optional statt eines Ports auch **None** möglich.

```
Define   LANctrl  = PortA, 5;      {PortName, bit nummer}
```

LANmode
Die Betriebsart der CPU, Master bzw. Slave muss mittels Define festgelegt werden. Ein Wechsel der gewählten Betriebsart während des Betriebs, also zur Laufzeit ist nicht möglich.

```
Define   LANmode  = Master;     {Master/Slave}
```

LANbaud
Die Arbeitsweise des LAN muss bei Master und Slave **identisch** sein und wird durch folgende Defines bestimmt:

```
Define   LANbaud  = 57600;      {Baudrate}
```

Die Baudrate kann beliebig sein, allerdings muss sie bei allen Teilnehmer gleich sein bzw. die RS232 typische Toleranz von max. +/- 5% einhalten.

LANadr
Die Adressbreite (Byte/Word) bestimmt die max. Teilnehmer Zahl. Obwohl mehr als 255 Slaves möglich sind (LANadr = 16), ist das nicht realistisch. Einfache Netzwerk Interfaces (z.B. RS485 Treiber) können in der Regel nicht mehr als 128 Anschlüsse treiben.

```
Define   LANadr   = 8 [,Mask];    {8 oder 16 bits, [maskiert von LANADRMASK] }
```

Wenn "Mask" weggelassen wird, ist keine Maske definiert. Ist LANPORT importiert, wird eine Variable "LANADRMASK" vom Typ Byte oder Word, abhängig von der Adress Größe (8/16 Bit) angelegt.
Auf diese Variable kann zugegriffen werden:

LANADRMASK := $0F;

Alle gesetzten Bits werden als DON'T CARE behandelt, das heisst, daß diese Bits im Adress Komparator beim Empfang eines Frames nicht beachtet werden.

LANframe
Master und Slave haben beide einen Receive- und einen Transmit-Buffer. Alle Slaves sollten die gleiche Buffergrösse wie der Master haben. Die angegebene Buffergrösse bestimmt den internen Speicherbedarf. Die transferierten Frames können kleiner aber nie grösser sein als hier angegeben.
Da die aktuelle Frame Länge als Parameter mit übertragen wird, müssen alle Frames entweder mit einem Byte als Längen Information auskommen oder mit einem Word. Ein mix ist nicht möglich. Diese Einschränkung vereinfacht das Protokoll und sein Handling wesentlich und ist in der Praxis eigentlich keine. Alle Frame Size Definitionen müssen daher entweder kleiner oder grösser 256 Bytes sein.
Die Länge dieser Buffer und der Speicherbereich wird durch folgendes Define bestimmt:

Define *LANframe = 16, iData; {Framesize max. 16 bytes in iData}*

LANcheck
Eine Prüfung des empfangenen Frames ist sehr wichtig bei Netzwerken. Der 8bit Check ist normalerweise ausreichend. Sind die Frames relativ lang (>50 bytes), sollte der 16bit Check eingestellt werden. Der CRC check ist nur bei sehr langen Frames und bei schwieriger Umgebung (Störungen) sinnvoll. CRC checks kosten sehr viel Zeit und Code.

Define *LANcheck = ChkSum8; {ChkSum8, ChkSum16, CRC16}*

3.12.2 Exportierte Variablen

3.12.2.1 Memory Organisation

Alle dem LANtreiber zugehörige Variable liegen aufgereiht hintereinander im Speicher. Die Reihenfolge ist, beginnend mit der niederwertigen Adresse, folgende:

LANNODEADR	Byte bei < 256 LAN Nodes, Word bei > 255 LAN Nodes. Nur beim **SLAVE**
_LANRXPTR	Byte bei einer Framelänge < 256 bytes, word bei Framelänge > 255 bytes
LANRXSTATREG	Byte
LANRXADR	Byte bei < 256 LAN Nodes, Word bei > 255 LAN Nodes
LANRXLEN	Byte bei einer Framelänge < 256 bytes, word bei Framelänge > 255 bytes
LANRXBUFF	Array[0..LANframe-1] of Byte;
_LANRXCHK	byte wenn ChkSum8, word wenn ChkSum16/CRC16
_LANTXPTR	Byte bei einer Framelänge < 256 bytes, word bei Framelänge > 255 bytes
LANTXSTATREG	Byte
LANTXADR	Byte bei < 256 LAN Nodes, Word bei > 255 LAN Nodes
LANTXLEN	Byte bei einer Framelänge < 256 bytes, word bei Framelänge > 255 bytes
LANTXBUFF	Array[0..LANframe-1] of Byte;
_LANTXCHK	Byte wenn ChkSum8, Word wenn ChkSum16/CRC16

LANnodeAdr
Diese Variable gibt es nur beim Slave. Der Typ ist ein Byte bei < 256 LAN Nodes und ein Word bei > 255 LAN Nodes. Der Slave vergleicht, wenn er eine Adresse in einem Frame erkennt (9.bit gesetzt), ob sie identisch ist mit der Adresse, die momentan gesendet wird. Ist sie identisch, wird dieser Frame im komplett empfangen und abgelegt. Adressen mit dem Wert „0“ werden als Broadcast Frames erkannt und grundsätzlich abgelegt, aber auch grundsätzlich nicht beantwortet.

Es sollte vermieden werden, dass mehr als ein Slave die gleiche Adresse hat. Gibt es dennoch mehrere Slaves mit der gleichen Adresse, darf keiner von ihnen auf einen Frame vom Master antworten. Andernfalls sind Bus Kollisionen vorprogrammiert. Es existiert keine einfache Möglichkeit deratiges abzufangen.

_LANRxPtr, _LANTxPtr
Diese beiden Variablen (Byte bzw. Word) dürfen nur gelesen werden! Sie sind für den internen Gebrauch bestimmt und werden als Pointer/Counter während des Empfangs bzw. Sendens benutzt. Diese Variablen sind ein Byte wenn „Define LANframe“ < 256 bytes, und ein Word bei „Define LANframe“ > 255 bytes

LANRxStatReg, LANTxStatReg
Diese beiden Variablen (Byte) dürfen nur gelesen werden! Beide Status Register werden beim Empfang bzw. Senden eines Frames automatisch upgedated. Im Ruhe Zustand sind alle bits 0. Bei fortschreitender Aktion werden die Bits von rechts nach “1”. Nach erfolgreichem Abschluss der Aktion sind alle Bits “1’ und das byte hat den Wert $FF. Zwischenwerte bedeuten, dass ein Fehler aufgetreten ist oder zumindest der Frame noch nicht komplett empfangen bzw. gesendet wurde. Ein $9F bedeutet z.B. Checksumme Fehler. Die Applikation sollte immer das 8.Bit prüfen, um festzustellen, ob ein Frame komplett ist.

```
(* bit0,1      :  00 = idle            *)
(*                01 = first adr       *)
(*                11 = second adr      *)
(* bit2,3      :  00 = idle            *)
(*                01 = first len       *)
(*                11 = second len      *)
(* bit4        :   0 = rx/tx frame     *)
(*                 1 = rx/tx frame     *)
(* bit5,6      :  00 = idle            *)
(*                01 = first check     *)
(*                11 = second check    *)
(* bit7        :   0 = processing      *)
(*                 1 = rx/tx complete  *)
```

LANRxAdr, LANTxAdr
Diese Variablen sind ein Byte wenn „Define LANadr = 8“, und ein Word bei „Define LANadr = 16“
Master:
Der Master empfängt alle Frames, die von einem anderen BUS-Teilnehmer verschickt werden. Das Programm kann den Absender anhand der Variablen “LANrxAdr” indentifizieren.
Verschickt der Master einen Frame, muss er den gewünschten Empfänger in die Variable “LANtxAdr” eintragen. Die Adresse “0” bezeichnet hierbei eine Broadcast Message, d.h. alle Teilnehmer empfangen den Frame, werten ihn aus, schicken aber keine Antwort (Kollisions Gefahr).
Slave:
Ein Slave empfängt nur diejenigen Frames, deren interne Adresse mit seiner Node-Adresse übereinstimmt. Diese interne Adresse des ankommenden Frames wird in “LANrxAdr” abgelegt. Die Adresse ist entweder mit der Node Adresse des Slave identisch oder es ist eine Broadcast Adresse. Das Programm muss bei einem empfangenen Frame diese Adresse prüfen, ob es eine Broadcast Adresse ist, und sich entsprechend verhalten.
In “LANtxAdr” wird normalerweise die Node Adresse des Slave abgelegt. Der Slave sollte nur Frames mit seiner eigenen Adresse verschicken. Es ist denkbar, dass ein Slave Broadcast Messages verschicken kann oder sogar einen anderen Slave adressieren kann. Das sollte aber unbedingt vorher mit dem Master “abgesprochen” werden, so dass es nicht zu Kollisionen auf dem BUS kommt.

LANRxLen, LANTxLen
Diese Variablen sind ein Byte wenn „Define LANframe“ < 256 bytes, und ein Word bei „Define LANframe“ > 255 bytes. Die verschickten Frames können die Länge „0“ haben. Die maximale Länge kann nicht grösser sein als in „Define LANframe“ angegeben. Grundsätzlich sollten alle Slaves und der Master gleiche „LANframe“ Grössen haben. Es ist jedoch denkbar, dass bestimmte Slaves eine kürzere „LANframe“ haben. Der Master muss aber dem Rechnung tragen und an diese Nodes nur Frames schicken, die auch in deren Buffer passen.
LANrxLen wird beim empfangen eines Frames automatisch mit dem empfangegen Wert belegt.
LANtxLen wird durch die Funktion „LANtxFrame(Node, len)“ gesetzt. LANtxLen enthält beim Senden und Empfang die **effektive Telegrammlänge**. Adresse, Checksumme, etc. werden nicht mitgezählt.
LANtxLen wird beim Empfang eines Frames mit dem im Frame enthaltenen Wert aktualisiert. LANtxLen wird durch die Funktion „LANtxFrame(Node, len)“ gesetzt und auch innerhalb des Frames übertragen.

LANRxBuff, LANTxBuff
Die Länge dieser beiden Buffer werden durch das Define LANframe bestimmt. Die Buffer können als Array of byte angesehen werden und damit auch gelesen und geschrieben werden.
Die Indize laufen von 0 bis LANframe -1.

```
LANtxBuff[0]:= $56;
X:= LANrxBuff[6];
```

Es ist empfehlenswert und sehr nützlich, eine Struktur (record) über den RxTeil und den Tx Teil des LAN Speichers zu legen. Damit ist ein symbolischer Zugriff auf die kompletten Daten möglich.

```
type    tLANRec = record
                    LANstate   : byte;       // LAN state size and loc fixed
                    LANnode    : byte;       // rx/tx address size and loc fixed
                    LANlen     : byte;       // rx/tx framelen size and loc fixed
                    LANusr1    : byte;       // user defined
                    LANusr2    : word;       // user defined
                    LANdata    : array[0..LANframe-4] of char;      // user
                  end;

var
  LANRxRec[@LANrxStatReg]  : tLANRec;
  LANTxRec[@LANtxStatReg]  : tLANRec;

// Frame belegen
  LANTxRec.LANusr1:= $30;
  LANTxRec.LANusr2:= $3231;
  LANTxRec.LANdata[0]:= '3';
```

_LANRxChk, _LANTxChk
Der Typ ist ein Byte wenn ChkSum8 und ein Word wenn ChkSum16/CRC16 eingestellt ist. Die Checksumme wird über den ganzen Frame gerechnet (LANRXADR + LANRXLEN + LANRXBUFF). Das gleiche gilt auch für den TxFrame. Die Checks werden automatisch beim Senden und Empfangen durchgeführt.

3.12.3 Exportierte Funktionen und Prozeduren

Function *LANRxStat : boolean;*
Diese Funktion gibt ein true zurück, wenn ein Frame empfangen wurde und noch nicht mit „LANrxClear“ ungültig gemacht wurde. Ob der Frame fehlerfrei ist, muss mit **LANrxStatReg** fest-gestellt werden. Solange ein Frame gültig ist, wird kein weiterer Frame akzeptiert, d.h. dieser wird ignoriert. Es ist Aufgabe des Programms einen empfangenen Frame so schnell wie möglich zu lesen und mit LANrxClear ungültig zu machen, so dass ein neuer empfangen werden kann.
Arbeiten die einzelnen Programme im Master und Slave mit einem Acknowledge, d.h. wenn der Master einen Frame losgeschickt hat, wartet das übergeordnete Programm, bis der Slave einen Antwortframe geschickt hat, gehen keine Frames verloren, aber der Durchsatz wird u.U. erheblich reduziert. Es ist also Aufgabe der einzelnen Programme bzw. des Programmierers ein entsprechendes Verfahren anzuwenden.

Procedure *LANRxClear;*
Das Statusbyte des RxBuffers wird zurückgesetzt.

Function *LANTxStat : boolean;*
Diese Funktion prüft, ob ein zu sendender Frame gesendet ist oder nicht. Ist der Frame gesendet, kehrt die Funktion mit true zurück, ansonsten mit false. Ein neuer Frame kann nur gesendet werden, wenn der aktuelle komplett übertragen ist oder der Status mit „LANtxClear“ zurückgesetzt wurde.

Procedure *LANTxClear;*
Das Statusbyte des TxBuffers wird zurückgesetzt.

Function *LANTxFrame (node : byte[word]; len : byte[word]) : boolean;*
Die Funktion prüft, ob der letzte Frame erfolgreich gesendet wurde. Wenn nicht, kehrt die Funktion mit false zurück. Mit LANtxClear kann der Frame zurückgesetzt werden, ohne die eigentlichen Daten zu löschen, und LANtxFrame kann noch einmal aufgerufen werden. Wenn der Frame gesendet werden kann, wird der „len“ Parameter in „LANtxLen“ abgelegt und node in „LANtxAdr“, wobei beim Slave der Wert in NODE seine eigene Adresse sein sollte, die auch in „LANnodeAdr“ abgelegt ist.

Procedure *LANrxAutoAck (**const** OnOff : boolean);*
Diese Prozedur existiert nur beim Slave. Ist OnOff = true, antwortet der Slave automatisch mit einem leeren Frame, wenn der empfangene Frame ohne Fehler war, jedoch nicht bei Broadcast Messages.

3.12.4 Multi-Processing

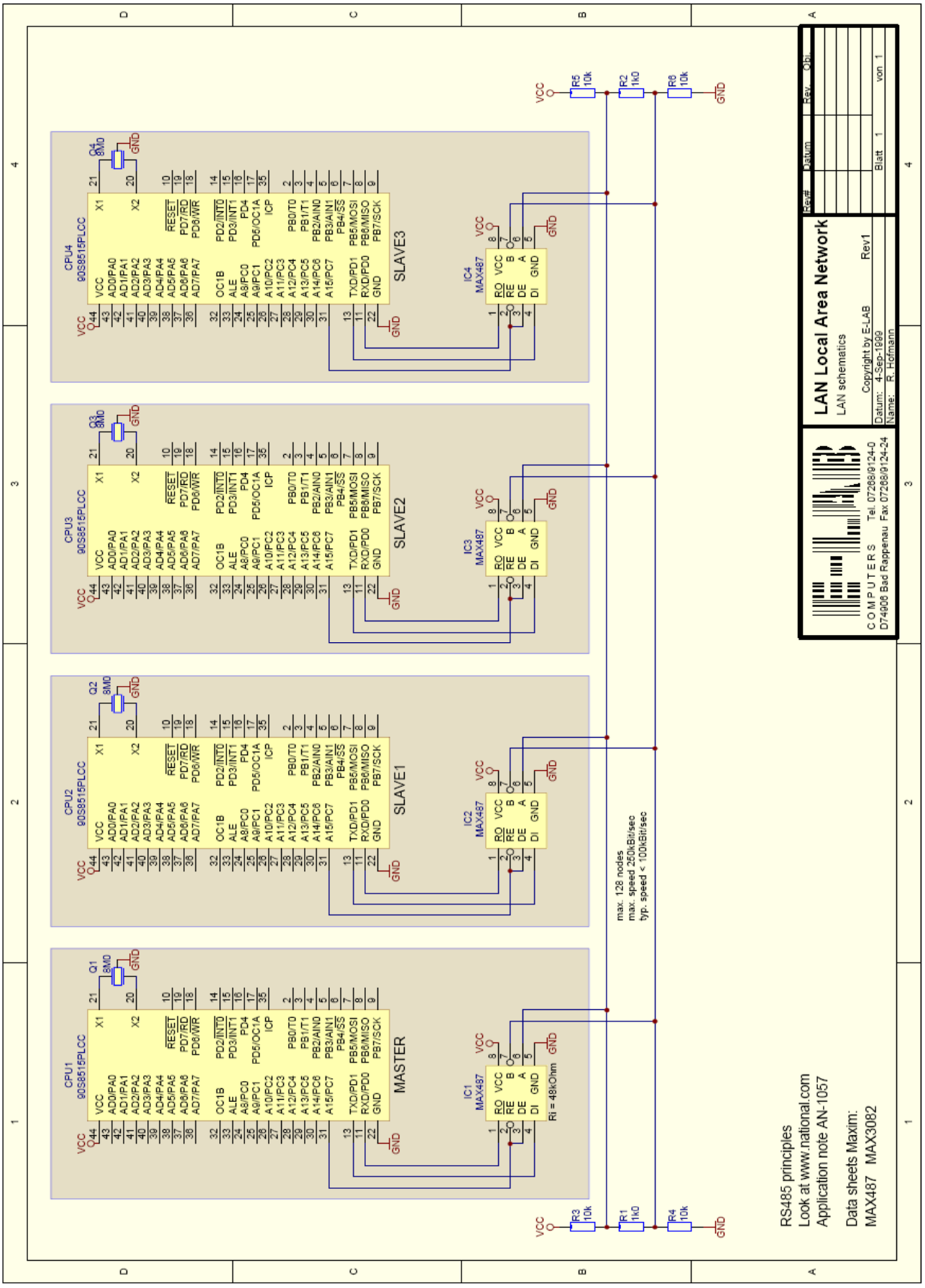
Wenn Prozesse in Betrieb sind, ist es u.U. sinnvoll, einen Prozess auf einen Frame warten zu lassen. Das kann mit “WaitPipe(LANrxBuff)” erreicht werden. Der Prozess wartet bis das erste Byte eines Frames empfangen wurde. Dann muss der Prozess pollen via LANrxStat und/oder LANrxStatReg.

3.12.5 Leitungstreiber

Das LAN Interface benutzt grundsätzlich die CPU internen seriellen Schnittstellen. Diese muss ein steuerbares 9. Bit haben um zwischen Daten und Adressen unterscheiden zu können.

V24/RS232 Teiber (z.B.. MAX232) sind nicht brauchbar, weil sie nicht abschaltbar sind. Alle Treiber, auch die des Masters sind im Ruhezustand abgeschaltet. Dadurch hören alle Teilnehmer auf dem BUS mit. Nur wenn ein Frame gesendet wird, wird der Treiber aktiviert und steuert den BUS.

Wenn eine elektrische Verbindung zwischen den Teilnehmern existiert, werden RS485 Treiber eingesetzt. Diese sind billig, robust, sehr schnell und sind für lange Leitungen geeignet. Weiterhin sind sie prinzipiell tristate fähig. Eine einfache Zwei-Draht Verbindung kann benutzt werden. Gemeinsame Masse ist normalerweise nicht notwendig. Twisted pair ist sinnvoll, aber nicht zwingend., ebenso ist eine Schirmung normalerweise nicht notwendig, auch nicht bei Übertragungsraten > 100kBaud.



Schaltplan **LAN**

3.13 USBport Treiber USBsmart XMega

Embedded Systeme brauchen sehr oft eine Anbindung an einen Host, z.B. PC. Lange Zeit war das die serielle Schnittstelle über das COMport. Moderne PCs und vor allen Dingen Laptops und Notebooks besitzen aber oft kein COMport. Es wird erwartet dass praktisch jedes Peripherie Gerät über eine USB Schnittstelle verfügt.

Es gibt mehrere Lösungs Möglichkeiten für eine USB Anbindung mit einem AVR:

1. ein FTDI Chip on Board welches mit dem entsprechenden Treiber auf dem PC ein virtuelles COMport bildet. Auf der XMega Seite ist der FTDI wie ein MAX232 an den XMega anzuschliessen. Nachteile: Platz Bedarf, Kosten und der Konflikt mit anderen COMports auf der PC Seite.
2. HID und USB-XMegas. Nachbildung einer Maus oder Keyboard auf der AVR Seite. Vorteile: preiswert, minimaler Aufwand auf der PC Seite. Nachteil: eingeschränkte Möglichkeiten des Systems.
3. CDC und USB-XMegas. Das System stellt für den PC einen virtuellen COMport dar, ähnlich FTDI. Vorteile: auf der PC Seite muss nur das INF-File erstellt werden, aber nicht ganz trivial. Nachteile: Konflikt mit anderen COMports, da man sich nie darauf verlassen kann, dass das Gerät immer auf dem gleichen COMport liegt und damit evtl. Konflikt mit anderen COMports, z.B. Bluetooth.
4. Unique USB und USB-XMegas. Der eleganteste Weg für einen USB Anschluss. Der XMega stellt damit für den PC ein völlig neues USB Gerät dar. Nachteile: es muss für den PC eine DLL, ein SYS und ein INF File erstellt werden. Die Kommunikation auf dem PC ist aufwändiger als bei allen anderen Verfahren. Vorteile: Es stehen alle Möglichkeiten offen: Kommandos über die Controlpipe senden (Endpoint0). Eine Daten Pipe (Endpoint1), Rx und Tx. Flexible Packet Grössen. Hohe Geschwindigkeit, bis zu 500kByte/sec! Alle Treiber und Files für den PC werden via Mausclick generiert!

Die vorliegende Implementation benutzt das unique Interface wie unter 4. beschrieben. Auf der AVR Seite bietet der XMega-USBsmart Treiber des AVRco System leistungsfähige aber trotzdem einfache Funktionen. Es werden alle XMegas mit interner USB Schnittstelle unterstützt. Es wurde bewusst auf Interupts verzichtet und die Zahl der Endpoints auf eins begrenzt. Dadurch wurde ein sehr kleiner (<2kB) aber trotzdem schneller Treiber bereitgestellt. Da keine Interrupts benutzt werden müssen die USB Funktionen immer gepollt werden.

Control Pipe: Über die Control Pipe (Endpoint0) kann ein echter Hardware Reset vom PC aus ausgelöst werden. Weiterhin können über die Control Pipe private Kommandos an den XMega abgesetzt werden. Dort werden diese Daten über eine Callback Funktion an die Applikation weitergeleitet. Um Timeouts auf dem PC zu vermeiden und den Anforderungen einer USB Schnittstelle gerecht zu werden muss die Applikation kontinuierlich die Control Pipe (Endpoint0) pollen.

Simple Interface. Es wird eine 64 Byte grosse Pipe (Endpoint1 Rx/Tx) zur Verfügung gestellt. Der eigentliche Datentransfer erfolgt durch den Endpoint1. Um Timeouts auf dem PC zu vermeiden und den Anforderungen einer USB Schnittstelle gerecht zu werden muss die Applikation kontinuierlich die beiden Endpoints pollen.

3.13.1 Import des USB Treibers

Wie sonst auch im AVRco System muss der Treiber importiert werden:

```
Import SysTick, USBsmart, ...;        // smart version oft he XMega USB driver
```

Der SysTick wird hier nicht unbedingt gebraucht.

3.13.2 Definition des USB Treibers

Der USB Treiber benötigt ein paar notwendige Defines die bestimmen wie sich der Treiber beim Host anmeldet.

```
Define
  // The XMega USB occupies the internal 32MHz source, so the internal 2MHz osc must be used here
  OSCtype          = int2MHz,
                     PLLmul = 16,
                     prescB = 1,
                     prescC = 1;          // CPU runs with 32MHz
  SysTick          = 10;                // msec
  StackSize        = $0064, iData;
  FrameSize        = $0064, iData;

  USBmanufact      = 'E-Lab Computers'; // max 31 bytes
  USBprodName      = 'xMiniUSBApp';     // "          "
  USBpid           = 30;                // product ID
  USBvid           = $9124;             // vendor ID
  USBprodRel       = 201;               // product release
  USBcurrent       = 200;               // current consumption, max 500mA
  USBvBUS          = PortB.7;           // port and pin to check the USB power connection, optional
```

SysTick ist optional.
USBvid ist die Vendor/Hersteller ID. Diese wird vom USB.org erteilt (kostet).
USBpid ist die Product ID. Diese kann beliebig sein.
USBprodRel ist aktuelle Version des Produkts. Beliebig.
USBmanufact ist der Hersteller im Klartext.
USBprodName ist der Name des Produkts im Klartext.
USBcurrent ist die zu erwartende Stromaufnahme über das USB Kabel in mA.
USBvBUS ist der Port Pin an dem die Applikation einen angeschlossenen PC erkennt (5Volt). Optional

USBsernum ist die Serien Nummer des Geräts und wird hier **nicht** angegeben. Der Treiber nimmt dafür die Chip-interne Seriennummer des XMegas. Jedoch kann eine Serien Nummer **optional** angegeben werden. Dann bekommen alle Geräte die gleiche Serien Nummer.

Alle Angaben sind Pflicht, wobei vom Host in der Regel nur USBvid und USBsernum wirklich beachtet wird. Ist die Serien Nummer aller Geräte dieses Hersteller und Typs konstant, dann erfolgt beim Anschliessen eines solchen Geräts an ein USB Port nur jeweils einmal die Nachfrage nach dem Treiber, und dann nie wieder. Wechselt die Serien Nummer so wird mit jeder neuen Serien Nummer wieder nach dem Treiber gefragt.

3.13.1 Callback Funktion

Das Host Programm hat die Möglichkeit über die Control Pipe (Endpoint0) diverse Kommandos abzusetzen. Dazu muss eine Callback Funktion im Main erstellt werden. Diese wird dann an den Treiber übergeben:

```
  //  User function to receive EP0 ControlRequests from the Host
Function myUSB_ControlRequest(bRequest : Byte; wValue : word) : boolean;
Begin
  // ...
  // this function returns with user defined commands in bRequest and optional parameters
  // in wValue send by the host
end;
...
  USB_SetControlCallback (@myUSB_ControlRequest);
```

Wird *myUSB_ControlRequest* nicht erstellt oder diese mit USB_SetControlCallback nicht an den Treiber übergeben, dann erfolgt auch kein Callback und die evtl. ankommenden Kommandos werden ignoriert. Wenn die Applikation den Request Typ definiert durch “bRequest” behandeln kann muss die Funktion ein true zurückgeben, ansonsten ein false. Die Requests 0..15 sollten für das System reserviert bleiben.

3.13.2 Exportierte Funktionen und Prozeduren

Procedure *USB_Init(pFirstRXBuf : pointer);*
Diese Prozedur initialisiert die USB Hardware im XMega. Der Pointer wird benötigt da ja ein Host unmittelbar nach dem Init schon Daten senden kann. Der Pointer muss auf einen 64byte Speicher Bereich zeigen.

Procedure *USB_SetControlCallback(myUserProc : tUserSetupProc);*
Mit dieser Prozedur wird dem Treiber eine optionale Call-Back Funktion zugewiesen die der Treiber aufruft wenn Vendor Requests (private Kommandos) in der Control Pipe (Endpoint0) ankommen.

Procedure *USB_ControlJob;*
Die Prozedur muss kontinuierlich aufgerufen werden (Polling). Hier wird abgeprüft ob über den Endpoint0 Aufträge oder Abfragen ankommen. Werden die Pausen hierbei zu lang (>50msec), kann es im Host zu time-outs kommen. Es macht Sinn diesen Job in einem Task ausführen zu lassen.

Procedure *USB_ControlSend(pData : Pointer; size : word); // send Controlrequest Data*
Wird in einem ControlCallBack ein bRequest abgearbeitet dann muss dieser immer mit dieser Prozedur beantwortet werden. Dabei kann der Pointer auf einen Datenblock zeigen und in „size“ die Anzahl der zu sendenden Daten angegeben werden. Werden keine Daten zurückgegeben, wird der Pointer NIL und size 0.

Function *USB_RxDataAvail : boolean; // check if Data on Endpoint1 received*
Dies ist die Rx Poll Funktion. Sie muss kontinuierlich aufgerufen werden. Werden die Pausen hierbei zu lang (>1sec), kann es im Host zu time-outs kommen, die zum kompletten Abbruch der Kommunikation führen. Bei einem true wird dann mit USB_RxCount die Anzahl der Bytes im RxBuffer abgefragt und der Inhalt des RxBuffers dann entsprechend verarbeitet. Ist dies geschehen muss der RxBuffer mit USB_RxSetBuf freigegeben werden. Danach akzeptiert der Treiber wieder neue Daten vom Host über den Endpoint1.

Function *USB_RXcount : byte;*
Sind Daten angekommen dann muss mit dieser Funktion die Blockgrösse/Bytecount festgestellt werden.

Procedure *USB_RxSetBuf(Buf : Pointer); // connect Endpoint1 do your buffer*
Wurden über Endpoint1 (USB_RxDataAvail) Daten empfangen, dann muss nach deren Verarbeitung der Empfang mit dieser Prozedur wieder freigegeben werden. Der Pointer muss auf einen 64byte grossen Buffer zeigen. Dieser muss global sein und auf gerader Adresse liegen, **Align2**

Procedure *USB_TxSend(pData : Pointer; size : byte);*
Wenn der Host Daten erwartet dann muss der Pointer auf den TxBuffer zeigen und in size muss die Anzahl der Bytes angegeben werden. Kann nichts gesendet werden, dann wird der Pointer NIL und size 0.
Der Buffer muss global sein und auf gerader Adresse liegen, **Align2**

Function *USB_TXcomplete : boolean;*
Nachdem USB_TxSend ausgeführt wurde, sollte durch Pollen dieser Funktion solange gewartet werden, bis der Host diese Daten sicher abgeholt hat. Jede weitere Aktion bis dahin ist nicht sinnvoll. Die Application kann ein Time-out in dieser Poll-Loop implementieren.

Procedure *USB_Detach;*
Meldet den USB Treiber vom Host (PC) ab.

Typischer Ablauf:

```
USB_Init …
USB_RxSetBuf …
Loop
   USB_ControlJob;
   if USB_RxDataAvail then
     Count:= USB_RxCount;
        // do something with RXData .....
     USB_RxSetBuf(@Rxbuf);                  // we need more data ..
  endif;
  // fill TxBuf with data …
  USB_TxSend(@TXbuf,64);
  Repeat until USB_TxComplete;
```

Ein Demo Programm befindet sich in der Demo Directory in “XMega_USBsmart” und ”XMega_BootUSB”.

3.13.3 Host/PC Implementation

USB Interfaces sind reine Master/Slave Systeme wie z.B. I2C oder SPI. Der PC ist immer der Host/Master und das Gerät ist immer der Slave. Ein Slave kann keinerlei Daten zum Host senden, sondern der Host muss Daten vom Slave abholen. Dies ist bei jeder Kommunikation zwischen Host und Slave im Auge zu behalten!

Ein USB Gerät braucht auf der Host/PC Seite immer einen passenden Treiber. Das sind u.U. schon vorhandene wie HID für Maus oder Tastatur. Hier werden keinerlei zusätzliche Treiber etc. gebraucht. Dann gibt es noch das relativ unbekannte bzw. wenig benutzte CDC Interface (virtual comport) das nur ein passendes INF-File benötigt. Proprietäre Geräte wie z.B. Drucker oder die E-LAB Programmer brauchen immer einen kompletten Treibersatz, bestehend aus einen SYS-File, INF-File und evtl. noch einer DLL.

Die AVRco USBsmart Implementation benötigt ebenfalls diese drei Treiber Teile. Da die Erstellung vor allem des SYS Files enormes Wissen verlangt benutzt das AVRco System einen sogenannten generic driver, der sich fast beliebig parametrisieren lässt, das ist das **libUSB** System. Dieses bietet neben einem kompletten, konfigurierbaren Treibersatz auch die automatische Erstellung des INF Files. Dieses Text File enthält die Beschreibung des Treibers und wesentliche Daten des daurch unterstützten Gerätes. Die manuelle Erstellung dieses INF-Files setzt sehr grosses knowhow voraus und ist sicher nicht jedermanns Sache.

Bei der Erstellung einer Windows Applikation für USB gesteuerte AVRco Geräte wird die durch das libUSB System erstellte DLL benutzt. Diese DLL bietet alle notwendigen Interface Funktionen. Die von der beiliegenden Delphi Applikation „USBtester“ benutzten DLL Funktionen (und viele mehr) sind in der Unit „LibUSB.pas“ enthalten. Benutzte Funktionen sind:

3.13.3.1 Initialisierung etc.

```
Procedure usb_init;
Function usb_find_busses : longword;
Function usb_find_devices : longword;
Function  usb_get_busses : pusb_bus;
```

3.13.3.2 Device spezifisch

```
Function usb_open(dev : pusb_device) : pusb_dev_handle;
Function usb_close(dev : pusb_dev_handle) : longword;
Function usb_set_configuration(dev : pusb_dev_handle; configuration : longword) : longword;
Function usb_claim_interface(dev : pusb_dev_handle; iinterface : longword) : longword;
Function usb_release_interface(dev : pusb_dev_handle; iinterface : longword) : longword;
```

3.13.3.3 Support

```
Function usb_get_descriptor_by_endpoint(udev : pusb_dev_handle; ep: longword; ttype : byte; index : byte;
                                        var buf; size : longword) : longword;
Function usb_get_descriptor(udev: pusb_dev_handle; ttype: byte; index: byte; var buf; size: longword):
longword;
Function usb_get_string_simple(dev: pusb_dev_handle; index: longword; var buf; buflen: longword) :
longword;
```

3.13.3.4 Data Transfer

Function *usb_control_msg(dev : pusb_dev_handle; requesttype, request, value, index : longword; var bytes; size, timeout : longword) : longword;*
Diese Funktion wird benutzt um über die Control-Pipe (endpoint0) diverse Kommandos abzusetzen. Folgende Parameter sind zwingend um „private“ Kommandos und Daten an den Slave zu schicken:

requesttype = $40
request = $00
value ist das eigentliche Kommando
index ist ein optionaler word parameter
bytes und *size* wird nur für das lesen der controlpipe benötigt
timeout ist in msec

Die Kommandos (value) 0..15 sind für das AVRco System reserviert.
Über die ControlMessage/endpoint0 können diverse Kommandos (Vendor Requests) geschickt werden.

Kommandos 16..255 (*msgUser+x*) sind für die User Applikation und werden im AVR Treiber durch eine Callback Funktion ausgewertet. Der index Parameter ist optional. Ein Beispiel ist im RESET Kommando in der Delphi Source zu finden.

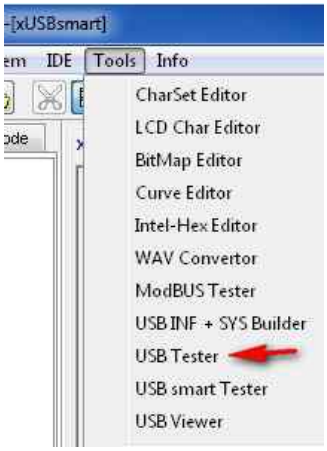
Function *usb_bulk_write(dev: pusb_dev_handle; ep : longword; var bytes; size, timeout:longword): longword;*
Das ist die eigentlich Sende Funktion. Daten werden zu der endpipe1 geschickt.
Parameter:

ep = endpoint1
bytes = Buffer der die Sende Daten enthält
size = Anzahl der zu sendenden Bytes. Bei endpoint1 beliebig, der Slave muss aber auch diese Daten am Stück verarbeiten können, d.h. dessen Rx-Buffer muss auch gross genug sein. Auf der PC Seite lassen sich hier beliebig grosse Packete angeben. Diese werden dann intern in 64 Byte grosse Blöcke/Packete gestückelt und gesendet. Das Resultat ist die Anzahl der effektiv gesendeten Bytes. Wenn der Slave dieses Packet wegen Speichergrenzen nicht komplett unterbringen kann dann wird trotzdem z.Zt. hier der Wert von „size“ zurückgegeben.

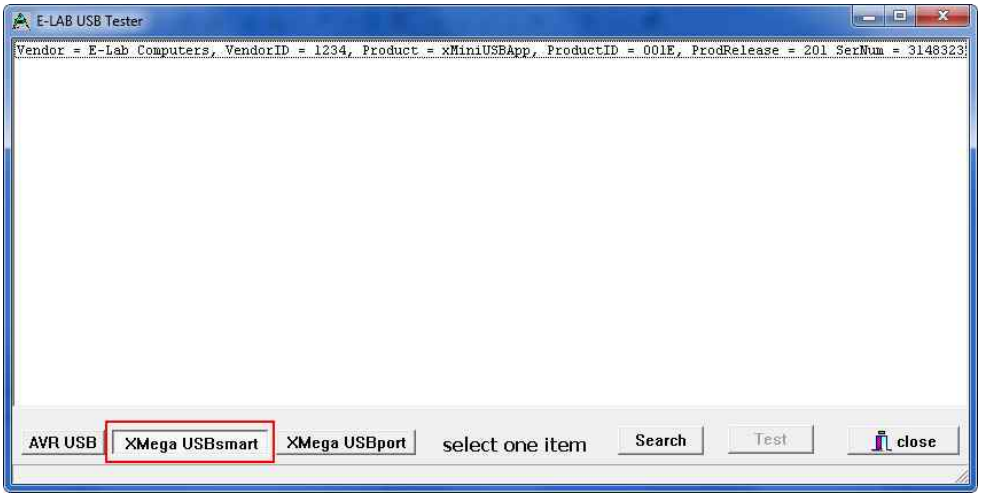
Function *usb_bulk_read(dev: pusb_dev_handle; ep: longword; var bytes; size, timeout:longword): longword;*
Das ist die eigentlich Empfangs Funktion. Daten werden von der endpipe1 abgeholt.
Parameter:

ep = endpoint1.
bytes = Buffer der die Empfangs Daten enthält
size = Anzahl der zu empfangenden Bytes. Bei endpoint1 beliebig, der Slave muss aber auch diese Daten am Stück liefern können, d.h. dessen Tx-Buffer muss auch gross genug sein. Auf der PC Seite lassen sich hier beliebig grosse Packete angeben. Diese werden dann intern in 64 Byte grossen Blöcke abgeholt und als ein Block/Packet bereitgestellt. Das Resultat ist die Anzahl der effektiv empfangenen Bytes.

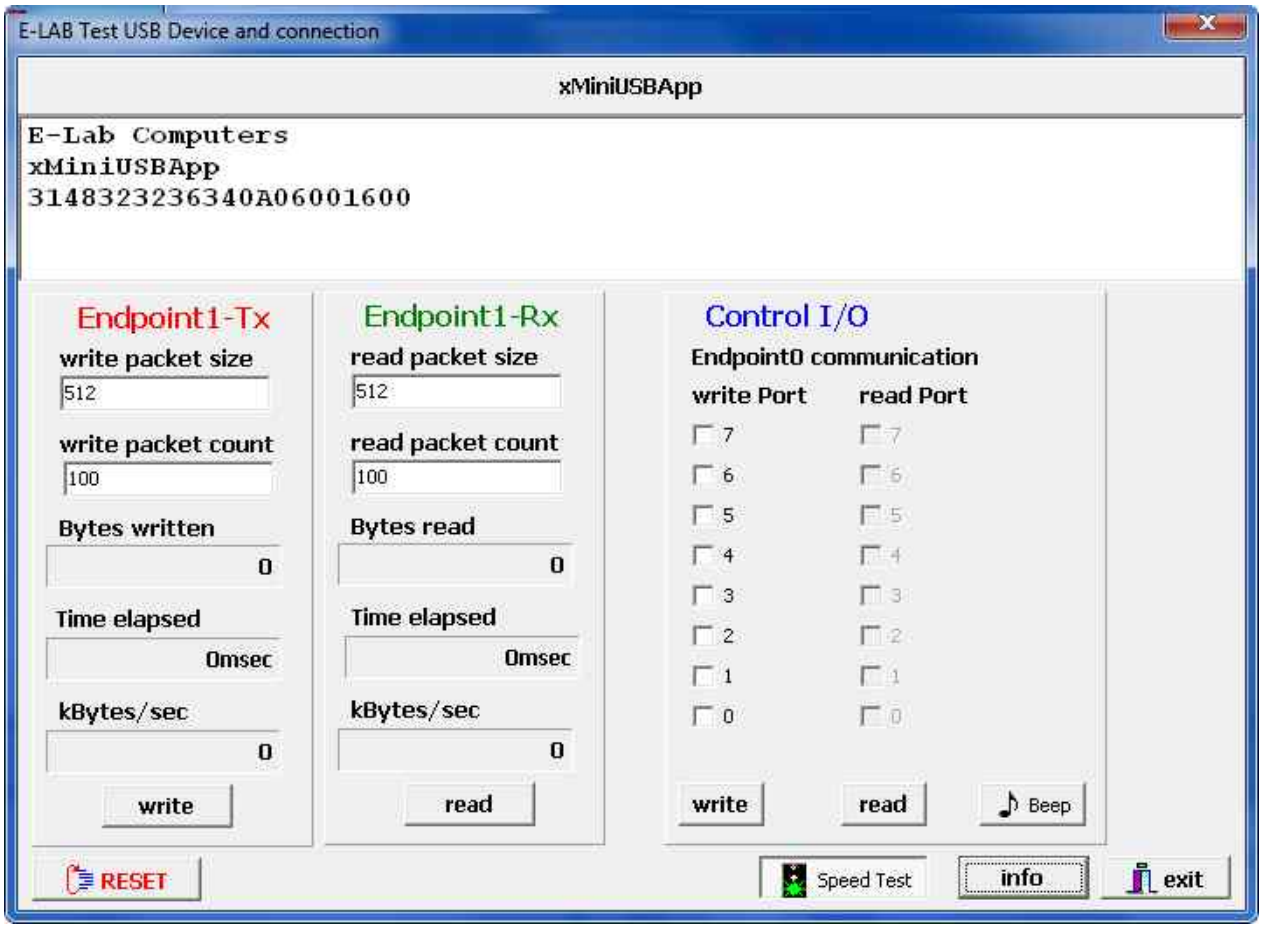
3.13.4 Testprogramm in der IDE PED32



Die AVRco Installation enthält ein Testprogram „USBtester“ das über nebenstehendes Menü gestartet werden kann. Dadurch wird untenstehendes Programm angezeigt:



Indem ein Device ausgewählt wird, wird der Arbeits Dialog angezeigt.



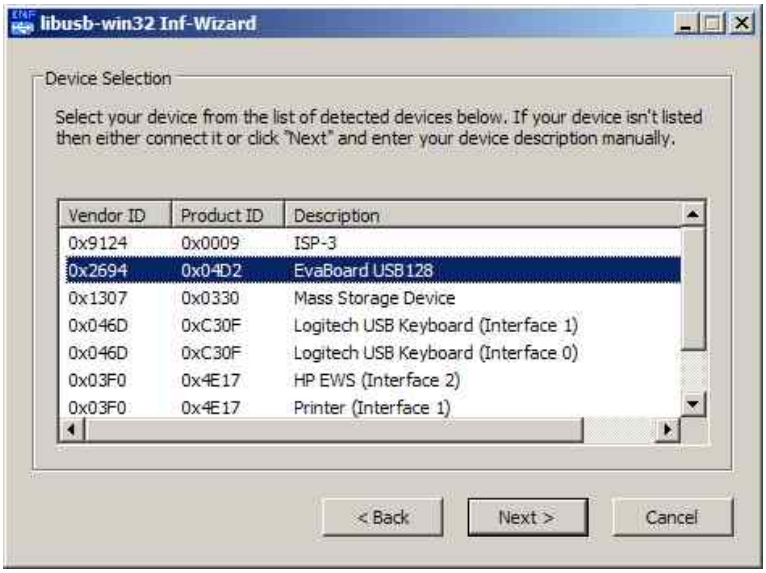
Hier können Transfers gestartet werden und Control Kommandos abgesetzt werden.

Die Source dieses Delphi Programms befindet sich in der Installations Directory unter

..\AVRco\IDE\USBtester\

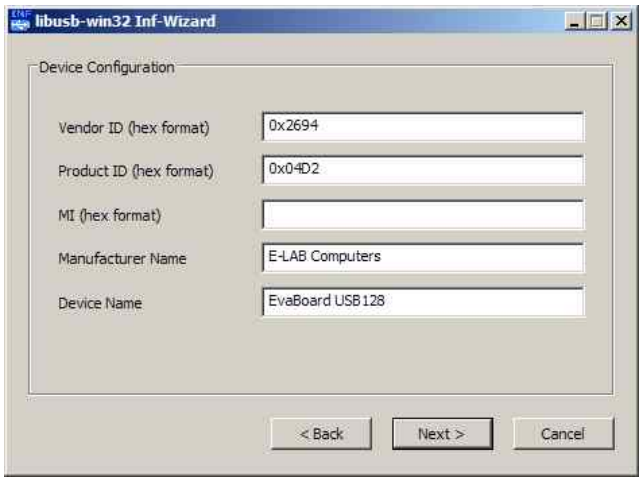
3.13.5 Support Tools

Inf-Wizard Start



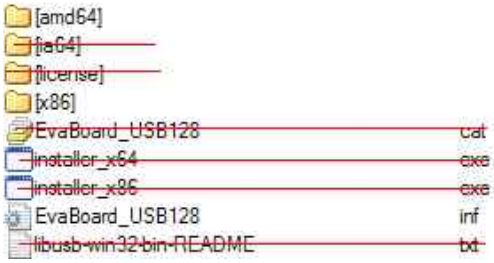
Die Erstellung der notwendigen SYS, INF und DLL Files für den Host/PC stellt eine enorme Aufgabe für den Entwickler dar. Für viele praktisch unlösbar ohne das richtige knowhow und Erfahrung. Das AVRco System enthält deshalb ein Tool das genau diese Files erstellt. Es wird mit obenstehenden Menu gestartet.

Achtung:
Das AVR Device muss angeschlossen sein und auch funktionieren! Ein Treiber braucht zu diesem Zeitpunkt noch nicht installiert sein. Wählen Sie nur das von Ihnen erstellte Gerät aus! Sollten Sie aus Versehen ein anderes Gerät gewählt haben und den generierten Treiber installieren, dann wird dieses (falsche) Gerät mit Sicherheit nicht mehr funktionieren!



Als nächster Schritt erfolgt die Überprüfung der Daten des gewählten Treibers. Mit „Next“ werden die Treiber Files erstellt und abgelegt.
Jetzt kann noch (optional) dieser Treiber direkt installiert werden. Damit ist dieses Gerät arbeitsbereit!

Der Wizard generiert mehrere Files:



Für den Enduser **relevant** sind nur diese:

Nebenstehende Files und Directories sind genau so an den Enduser weiter zu geben.

Wird das Gerät zum ersten mal an einen PC angeschlossen erfolgt eine Abfrage nach dem passenden Treiber. Hier ist dann das **INF-File** anzugeben. Der Rest folgt dann automatisch.

Niemals den Inf-Wizard mitgeben. Unbedarfte User können sich damit das komplette PC System lahmlegen.

3.14 PWMports

3.14.1 PWMport1A 1B 1C PWMport3A 3B 3C PWMport4A 4B 4C PWMport5A 5B 5C

Alle AVRs haben zumindest einen 16bit Timer (Timer1) der bis zu 3 PWM Kanäle anbietet. Grössere CPUs haben auch noch einen zweiten 16bit Timer (Timer3) der ebenfalls bis zu 3 PWM Kanäle haben kann. Welche Timer jeweils vorhanden sind, muss dem entspr. Datenblatt entnommen werden. Wieviel PWMs solch ein Timer wirklich besitzt, ist nicht ganz so leicht festzustellen. Am besten ist man nimmt sich das Pinout des Datenblatts und sucht nach Pin Namen „OC1A, OC1B und OC1C“ für den Timer1. Beim Timer3 heissen diese Pins „OC3A, OC3B, OC3C“. Sind diese Pins vorhanden, kann auch der zugehörige PWM importiert werden. Wenn nur ein Kanal vorhanden ist, fehlt der abschliessende Buchstabe (z.B. OC2).

Da ein Timer bis zu 3 PWMs steuern kann, ist dessen Setup (prescaler, resolution) natürlich für alle seine drei PWMs gemeinsam. PWMport4 und 5 nur in der Rev und höher.

Imports
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.
Die zu verwendenden Timer (Timer1 und/oder Timer3) und PWMports werden durch den Import der PWMports festgelegt.

Import *SysTick, PWMport1A, ..., PWMport3C, ...;*

Defines
Die Werte der Vorteiler und die Auflösung der PWMs.

```
Define   ProcClock     = 8000000;        {Hertz}
         SysTick       = 10;             {msec}
         StackSize     = $0030, iData;
         FrameSize     = $0030, iData;
         PWMpresc1     = $2;             {prescaler  timer1}
         PWMres1       = $8;             {resolution timer1}
         PWMmode1      = fast, negative  {optional define}
         PWMpresc3     = $4;             {prescaler  timer3}
         PWMres3       = $10;            {resolution timer3}
         PWMmode3      = slow, positive  {optional define }
```

PWMres1 (Timer1),
PWMres3 (Timer3)
PWMres4 (Timer4)
PWMres5 (Timer5)
definiert die Auflösung (Resolution) für jeweils einen der beiden möglichen PWMport Gruppem.
Beim AVR kann hier 8, 9 oder 10 Bits eingestellt werden.

PWMpresc1 (Timer1),
PWMpresc3 (Timer3)
PWMpresc4 (Timer4)
PWMpresc5 (Timer5)
definiert den gemeinsamen Vorteiler für jeweils eine der beiden möglichen PWMport Gruppen.
Der Vorteiler bestimmt die Wiederholrate bzw. Frequenz der PWM’s. Werte für AVR 1..5.

PWMmode1 (Timer1),
PWMmode3 (Timer3)
PWMmode4 (Timer4)
PWMmode5 (Timer5)
Optionale Defines, bestimmen den standard oder fast mode des Timers und die Puls Polarität.

PWMport1A, PWMport1B, PWMport1C (Timer1)
PWMport3A, PWMport3B, PWMport3C (Timer3)
Der Import von PWMPORTxx importiert automatisch die zugehörige Variable *PWMPORTxx*
Die Variablen sind immer vom Typ Word, unabhängig von der gewählten Auflösung.
Sie sind das einzige Interface zwischen Applikation und Treiber.

```
PWMport1A:= 127;     {50% Duty Cycle at 8bit res}
PWMport3B:= 511;     {50% Duty Cycle at 10bit res}
```

PWMport4A, PWMport4B, PWMport4C (Timer4)
PWMport5A, PWMport5B, PWMport5C (Timer5)
Timer4 und Timer5, falls vorhanden. Diese PWMs werden ab Compiler vRevision 4.xx unterstützt.

```
PWMport4A:= 127;     {50% Duty Cycle at 8bit res}
PWMport5B:= 511;     {50% Duty Cycle at 10bit res}
```

Bemerkung:
Evtl. vorhandene 8bit PWMs der 8bit Timer werden in REV3.xx von AVRco nicht unterstützt
Die alten AVRco PWMport Namen ***PWMport1*** und ***PWMport2*** sind zwar weiterhin gültig, sollten aber nicht für Neu-Designs verwendet werden.

3.14.2 PWMport2A, PWMport2B

neuere AVRs haben teilweise einen Timer2, der bis zu 2 PWM Kanäle haben kann (mega644 / 2560).
Auch diese können, sofern vorhanden und PWM fähig, benutzt werden.
Die Ausführungen für Timer1/Timer3 gelten dann analog.

```
Import SysTick, PWMport2A, PWMport2B;

Define
...
        PWMpresc2   = $8;           {prescaler  timer2}
        PWMres2     = $8;           {resolution timer2}
        PWMmode2    = fast, negative    {optional define}
...

PWMport2A:= 84;
PWMport2B:= 10;
```

3.14.3 Software PWM

Eine Software PWM ist implementiert. Es können bis zu 8 Kanäle eingestellt werden. Dazu wird ein 8bit Timer (Timer0 oder Timer2) verwendet, der im Interrupt läuft. Die Auflösung lässt sich zwischen 10 und 255 einstellen. Die PWM Frequenz (Cycle Time) kann zwischen 5msec/200Hz und 50msec/20Hz eingestellt werden. Diese PWMs sind interessant für Helligkeits Steuerung von Lampen und LEDs, zur Drehzahlregelung von Motoren und zur Temperatur Regelung.

Bei diesem Treiber ist zu beachten, dass bei einer 8MHz CPU und 10msec cycletime (100Hz) der Update auf alle 8 Kanäle 15usec in dem Timer Interrupt benötigt. Die Wiederhol Rate des Interrupts ist 100usec wenn eine Auflösung von 100 eingestellt ist. Daher benötigt der SoftPWM in diesem Fall 15% der CPU Leistung.

Mit einer Auflösung von 254 wird die Wiederhol Rate des Timer Interrupt auf 40usec erhöht und der SoftPWM benötigt 40% der CPU Leistung.

Mehr Kanäle benötigen mehr CPU Leistung. Höher Auflösung braucht zusätzliche CPU Leistung und kürzere PWM Zyklen brauchen ebenfalls mehr CPU Leistung.

Imports

```
Import SysTick, SoftPWM;
```

Defines

```
Define
    ProcClock       = 8000000;          {Hertz}
    SysTick         = 10;               {msec}
    StackSize       = $0032, iData;
    FrameSize       = $0032, iData;
    SoftPWMport     = PortA;            {use PortA for PWM output}
    SoftPWMchans    = 4, 0;             {4 channels, bit0/PortA is the first}
    //SoftPWMchans  = 4, 2, negative;   {4 channels, bit2/PortA 1.bit, low pulsed}
    SoftPWMtimer    = timer2, 10;       {use timer2, PWM cycle time 10msec = 100Hz}
    SoftPWMres      = 16;               {PWM resolution is 16 points}
```

SoftPWMport
Bestimmt das zur Ausgabe notwendige PORT

SoftPWMchans
Bestimmt die Anzahl der gewünschten PWM Kanäle und das erste Bit des Ports = PWM1 Bit
Optional kann die Ausgangspolarität *positive/negative* vorgegeben werden.

SoftPWMtimer
Bestimmt den 8-bit Timer (Timer0 oder Timer2) und die Zyklus Zeit in 10mSec Schritten.
XMega Hier muss einer der Timer (Timer_C0.. Timer_F1) bestimmt werden.

SoftPWMres
Bestimmt die Auflösung der PWMs, 10..254 Punkte.

Es gibt für diesen Treiber zwei Funktionen:
Procedure *SoftPWMstop;*
Der zugehörige Timer wird angehalten. Die PWM Pins werden auf idle/inaktiv gesetzt.

Procedure *SoftPWMstart;*
Der zugehörige Timer wird neu gestartet und alle PWM Werte werden auf Null gesetzt.

Für die PWMports gibt es nur Byte-Speicherstellen die vom System exportiert werden und die gelesen und geschrieben werden können.

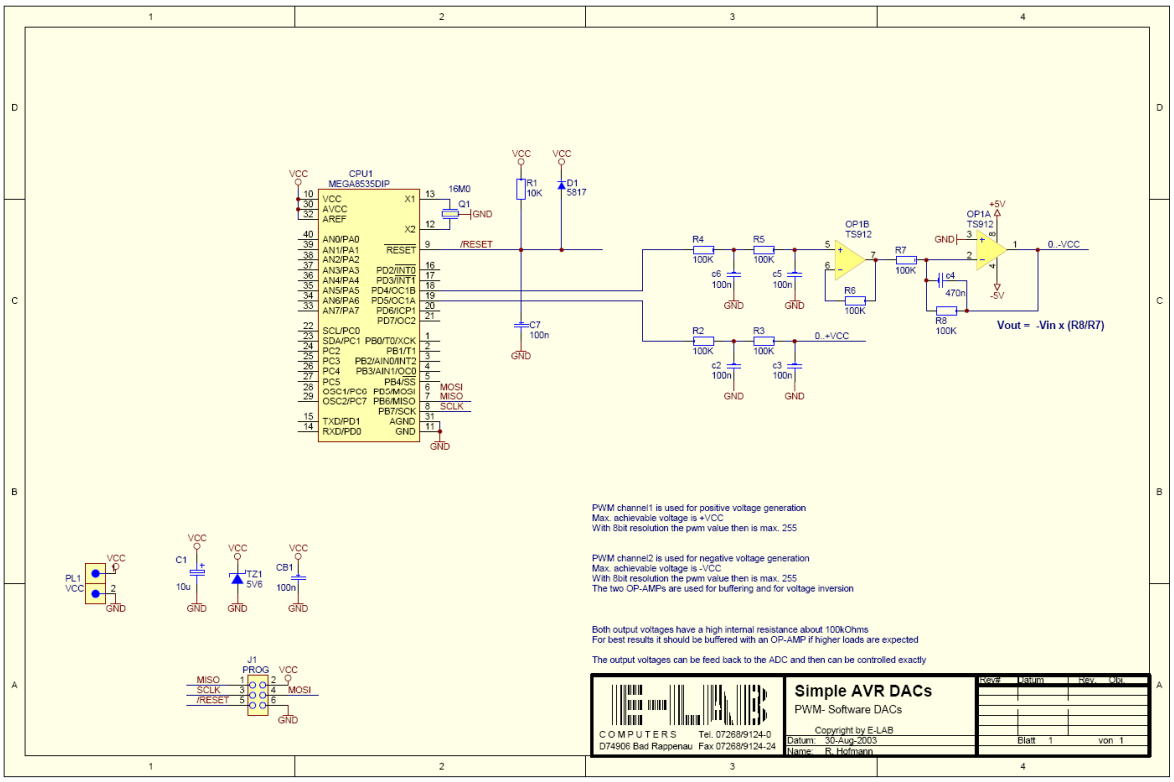
```
Var
  SoftPWM0 : byte;
  SoftPWM1 : byte;
  SoftPWM2 : byte;
  SoftPWM3 : byte;
```

Es ist auch möglich die PWMs als ein Array of Byte anzusprechen. Das Array ist definiert mit:

```
var
  SoftPWMarr  : array[0..maxChan-1] of byte;

  SoftPWM2:= 8;
  SoftPWMarr[1]:= 5;
```

Programm Beispiel und Schaltung:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\SoftPWM**
ein XMega Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_SoftPWM**



Schaltplan **SoftDAC**

3.14.4 Software-PWM SoftPWM8 XMega

Für die XMegas ist eine weitere Software PWM implementiert. Es können bis zu 16 Kanäle eingestellt werden. Die Kanäle können auf beliebigen Ports und beliebigen Port-Bits liegen. Der Vorteil dieses PWMs ist eine extrem geringe CPU Last und sehr kurze Interrupt Latenz Zeit (Interrupt Sperrung).
~2.5usec total @ 32MHz 8 Kanäle.
Der Nachteil ist die Auflösung von 3.5bits. Das bedeutet es können 8 Stufen eingestellt werden plus komplett aus. Für viele Anwendungen ist das vollkommen ausreichend.

Imports
```
Import SysTick, SoftPWM8;
```

Defines
```
Define
  // The XMegas don't provide any Oscillator fuses.
  // So the application must setup the desired values
  // possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz
  //>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
  OSCtype   = int32MHz,
              PLLmul=4,
              prescB=1,
              prescC=1;
  SysTick    = 10, adj;         // msec, correct the RTC32K timer for exact mSec timing
  StackSize  = $80, iData;
  FrameSize = $100, iData;

                      // Chan0    Chan1   Chan2   Chan3  Chan4   Chan5 ...
  SoftPWM8chans  = PortR.1, PortF.0, PortF.1, PortF.2, PortF.3, PortF.4,
                     PortF.5, PortF.6, PortF.7; // 8 channels, mixed ports
  SoftPWM8timer   = Timer_D0, 10;      // use Timer_D0, PWM cycle time 10msec = 100Hz
```

SoftPWM8chans
Hier können bis zu 16 Ports mit Pins angegeben werden.

SoftPWM8timer
Das Define bestimmt den XMega Timer der für den PWM gebraucht wird. Der zusätzliche Parameter bestimmt die Zyklus Zeit. (1..100msec)

Exportierte Funktionen
```
procedure SoftPWM8set(chan, pwm : byte);
```
Diese Prozedur setzt die gewünschte on-time für den angegebenen Kanal. Die Zeit muss im Bereich 0..8 liegen. 0 = off, 8 = full on. Die Kanal Nummer beginnt mit 0 und läuft bis zur Anzahl des Defines -1.

```
procedure SoftPWM8start;
```
Startet den PWM. Timer Interrupt wird freigegeben.

```
procedure SoftPWM8stop;
```
Stoppt den PWM. Timer Interrupt wird gesperrt. Die vorgegeben Werte werden aber nicht verändert.

```
procedure SoftPWM8clear;
```
Die eingestellten Werte aller Kanäle werden auf 0 gesetzt. Timer Interrupt wird nicht verändert.

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_SoftPWM8**

3.15 XMega PWM

Jeder XMega 16bit Timer hat bietet bis zu 4 PWM Ausgänge die alle von diesem Treiber unterstützt werden. Da ein Timer bis zu 4 PWMs steuern kann, ist dessen Setup (prescaler, resolution) natürlich für alle seine PWMs gemeinsam.

Imports
Der Treiber muss wie gewöhnlich beim AVRco importiert werden. Die zu verwendenden Timer (TimerC0, TimerC1 etc) und PWMports werden durch den Import der PWMports festgelegt. Im folgenden bezeichnet ***TT*** den Timer (C0, C1, D0, D1 etc) und ***C*** den Kanal (A, B, C, D).

```
Import SysTick, PWM_TTC;
Import SysTick, PWM_C0A, PWM_C0B, PWM_C0C, PWM_C0D, PWM_C1A, PWM_C1B;
```

Defines
Bestimmen die Werte der Vorteiler und die Auflösung der PWMs sowie die Ausgangspolarität der Kanäle.

*PWMpresc_**TT***
definiert den Vorteiler für jeweils einen der Timer (1..7).
Der Vorteiler bestimmt die Wiederholrate bzw. Frequenz der PWM's.

*PWMres_**TT***
definiert die Auflösung (Resolution) für jeweils einen der Timer (8..16bits).

*PWMpol_**TTC***
definiert die Ausgangs-Polarität eines Kanals (positive oder negative).

```
Define
        //>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
        OSCtype         = int32MHz,
                          PLLmul=4,
                          prescB=1,
                          prescC=1;
        StackSize       = $0030, iData;
        FrameSize       = $0060, iData;
        PWMpresc_C0     = $1;           // prescaler timerC0
        PWMres_C0       = 8;            // pwm resolution timerC0
        PWMpol_C0A      = negative;     // output polarity PWM_C0A
```

3.15.1 Funktionen und Prozeduren

***Procedure** EnablePWM_**TTC**(ena : boolean);*
Startet oder stoppt den PWM Kanal. Bei disable geht der Ausgang auf den 0-Wert.

```
EnablePWM_C0A(true);
```

***Procedure** SetPWM_**TTC**(pw : byte|word);*
Stellt das Tastverhältnis ein. Bei Auflösungen > 8bit wird ein Word erwartet, ansonsten ein Byte.

```
SetPWM_C0A(123);
```

Zu beachten ist dass ein 100% Wert nicht erreicht werden kann. Es bleibt dann immer noch ein minimaler peak. Die Timer C0, D0, E0 und F0 haben 4 PWM Kanäle, Timer C1, D1, E1 und F1 haben 2 Kanäle. Das ergibt bei den grösseren XMegas bis zu 24 PWMs. Deshalb ist ein SoftPWM bei den XMegas nicht sinnvoll.

Die PWM Ausgänge (Pins) sind fest bestimmten Port Pins zugeordnet und lassen sich nicht ändern.
*OC**TC*** outout-compare-timer-chan pin

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_PWM**

3.16 XMega CRC

Wenn Datenfehler bei Übertragungen etc. erkannt werden muss, dann bietet sich eine Checksumme an, die einem Datenblock angehängt werden müssen. Der sicherste Weg dazu ist eine CRC Checksumme, die mit einem bekannten Polynom über das komplette Paket gebildet wird. Wenn ein Fehler beim Empfänger aufgetreten ist muss die Operation wiederholt oder verworfen werden. Eine Fehler Korrektur findet nicht statt.

Es gibt zwei CRCs:
CRC16 (CRC-CCITT, Polynom *x*^16+*x*^12+*x*^5+1)
CRC32 (IEEE 802.3, Polynom *x*^32+*x*^26+*x*^23+*x*^22+*x*^16+*x*^12+*x*^11+*x*^10+*x*^8+*x*^7+*x*^5+*x*^4+*x*^2+*x*+1)

Beide werden durch interne Hardware in den meisten XMegas unterstützt. Wobei das CRC32 natürlich die bessere Sicherheit bietet.

Imports
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.

Import *SysTick, XMega_CRC , ..;*

3.16.1 Funktionen und Prozeduren

Function *CRC16_Block(seed : word; source : pointer; count : word) : word;*

Function *CRC32_Block(seed : longword; source : pointer; count : word) : longword;*

Diese beiden Funktionen sind identisch bis auf das Ergebnis, 16 oder 32bit.
Seed bestimmt den Initial Wert für den CRC
Source muss auf einen Speicher im internen SRAM zeigen.
Count gibt die Anzahl der zu prüfenden Bytes an.
Die Checksumme wird als word oder longword zurückgegeben.

Streamen
Obige Funktionen arbeiten mit Datenblöcken. Manchmal muss man auch mit Daten Strömen arbeiten. Dazu sind untenstehende Funktionen vorgesehen. Mit ***StreamInit*** wird der CRC gestartet. Dann werden mit ***StreamAdd*** kontinuierlich Bytes dem CRC hinzugefügt. Ist das streamen beendet dann kann das Ergebnis mit ***StreamGet*** abgeholt werden.

Procedure *CRC16_StreamInit(seed : word);*
Startet bzw. Initialisiert den CRC Stream.

Procedure *CRC16_StreamAdd(b : byte);*
Fügt jeweils ein Byte dem CRC hinzu.

Function *CRC16_StreamGet : word;*
Hiermit wird das Ergebnis (CRC) obiger Operationen abgeholt.

Die 32bit Funktionen entsprechen obigen 16bit Funktionen
Function *CRC32_StreamInit(seed :longword);*
Procedure *CRC32_StreamAdd(b : byte);*
Function *CRC32_StreamGet : longword;*

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_CRC**

3.17 SPI onBoard Netzwerk

Nicht für XMegas, statt dessen serielle SlipPorts benutzen.

Das SPI-Interface der Atmel AVR Familie eignet sich gut zur Kopplung von zwei oder mehreren CPUs. Die Kommunikation zwischen CPU und speziellen SPI-Bausteinen ist auch möglich.

Die vorliegende Implementation ist in erster Linie für CPU-CPU Verbindungen gedacht. Simple Byte Transfers zwischen CPU und SPI-Bausteinen sind nicht implementiert, diese können mit wenigen Maschinen Befehlen selbst implementiert werden. Die Strategie besteht aus einem Single-Master und Multi-Slave System. Zur sicheren Kommunikation wird ein entsprechendes Protokoll gefahren.

3.17.1 Mini Netzwerk

Wie schon o.a. ist bei einem SPI-Interface ein einzel-Byte Transfer nicht das Problem. Schwierig und aufwendig wird es erst bei CPU-CPU Kopplungen. Hier setzt die Implementation an. Es wurde ein Single-Master / Multi-Slave System implementiert, das, wie der Name sagt, grundsätzlich aus einem Master und mindestens einem Slave besteht.

Der Master kontrolliert alle angeschlossenen Slaves. Kein Slave kann von sich aus die BUS-Kontrolle übernehmen. Jeglicher Transfer, ob vom oder zum Slave wird vom Master aus initiiert und gesteuert. Damit entfallen die sonst erheblichen Probleme mit Kollisionen, Prioritäten etc.

Die Implementation kann als On-Board Netzwerk bezeichnet werden. Man spricht hier auch von einem “Micky Maus” Netzwerk.

Netzwerke arbeiten allgemein mit Telegrammen, auch Frames genannt. Durch die Verwendung von Frames wird zwar ein gewisser Overhead betrieben, im Gegenzug dafür aber die Betriebssicherheit erheblich gesteigert. Um weiteren, dem SPI anhaftende Probleme, aus dem Weg zu gehen, wurde auf den dem SPI eigenen Full Duplex Transfer verzichtet. Statt dessen wird mit einem quasi Halb-Duplex Betrieb gearbeitet. Der Master fordert z.B. ein Frame von einem Slave. Der Slave schickt ein Acknowledge und der Master holt den Frame beim Slave ab. Umgekehrt, wenn der Master ein Frame zu einem Slave schicken will, schickt der Master zuerst ein Request, der Slave antwortet wiederum mit einem Acknowledge und der Master schickt daraufhin seinen Frame zum Slave.

Das Request/Acknowledge Verfahren kann man auch als ein Polling des Masters betrachten. Der Master arbeitet grundsätzlich ohne SPI-Interrupts, während der Slave grundsätzlich mit SPI-Interrupts arbeitet. Wenn das Requ-Ack Handshake fertig ist, sind alle Interrupts des Masters gesperrt und der Slave befindet sich in der SPI-Interrupt-Service Routine, die er erst wieder verlässt, wenn der Frame komplett übertragen ist. Damit wird sichergestellt, dass der Frame in kürzester Zeit übertragen ist (Master) und kein Byte verloren geht (Slave). Byte Verluste sind ein Hauptproblem bei SPI CPU-CPU Kopplungen, denn ein Interrupt im Slave während der Übertragung führt bei den hohen Bitraten mit Sicherheit zum Byte Verlust beim Slave. Andererseits wenn der Slave während des Frames seinen Interrupt Service nicht verlässt, kann das nicht passieren. Der Nachteil ist, dass alle Interrupts im Slave während des Transfers gesperrt sind. Um diese Zeit möglichst kurz zu halten, darf auf der anderen Seite der Master während des Transfers auch nicht durch eigene Interrupts (z.B. Task-Wechsel) gestört werden, um den Slave nicht allzulange warten zu lassen.

Die Frames Längen können beliebig variieren, allerdings in den Grenzen 1..SPIbufferLen.

Jeder Slave benötigt zum aktivieren des Empfangs eine Steuerleitung vom Master, die an den **SS**-pin des Slave angeschlossen wird. Damit kann auf die Slave Adresse innerhalb des Frame verzichtet werden. Die Anwendungssoftware bestimmt beim Master wie bzw. welches Port diese Select Leitungen erzeugen. Der AVRco SPI Treiber bietet hier verständlicherweise keine Unterstützung. Es ist Aufgabe des Anwendungsprogramms vor der Aufnahme der Kommunikation mit einem Slave diesen auch durch die zugeordnete Steuerleitung zu aktivieren.

Ist bei einem Slave der SS-Eingang inaktiv (log. 1) so schaltet dieser seine SPI Logik ab und die beiden Leitungen MISO und MOSI werden ge-tristatet. Damit können eine grössere Anzahl Slaves an den BUS angeschaltet werden, vorausgesetzt, dass immer nur ein Slave ein aktives SS-Signal hat.

Prinzipielles
Das SPIport muss, wie beim AVRco System üblich, durch eine Import Anweisung importiert werden

Imports

```
Import   SPIport;
```

Die Betriebsart der CPU, Master bzw. Slave muss mittels Define festgelegt werden. Ein Wechsel der gewählten Betriebsart während des Betriebs, also zur Laufzeit ist nicht möglich.

Defines

```
Define   SPIport = Master;
```

oder

```
Define   SPIport = Slave;
```

Master und Slave haben beide einen Receive- und einen Transmit-Buffer. Alle Slaves sollten die gleiche Buffergrösse wie der Master haben. Die Länge dieser Buffer und der Speicherbereich (min1, max. 254) wird durch folgendes Define bestimmt:

```
Define   SPIbuffer = nn, iData;
```

Die Arbeitsweise des SPI muss bei Master und Slave identisch sein und wird durch folgende Defines bestimmt:

```
Define   SPIOrder   = MSB;      {LSB}
         SPICPOL    = 0;        {0/1}
         SPICPHA    = 0;        {0/1}
         SPIpresc   = 0;        {0/1/2/3}
```

Wenn der Master vom Programm beauftragt wird, einen Frame vom Slave abzuholen oder an ihn zu senden, pollt er den Slave auf dessen Acknowledge. Damit dies nicht zur Blockade des Masters führt, muss mit dem Define SPIretry vorgegeben werden, wie oft der Master pollt bevor er mit einer Timeout Fehlermeldung (SPIrxFrame, SPItxFrame) zurück kommt:

```
Define   SPIretry    = 10;
```

3.17.2 Exportierte Variablen

SpiTxBuff, SpiRxBuff
Die Länge dieser beiden Buffer werden durch das Define SPIbuffer bestimmt. Die Buffer können als Array of byte angesehen werden und damit auch gelesen und geschrieben werden. Die Indize laufen von 0 bis SPIbufferLen -1.

```
SPItxBuff[0]:= $56;
X:= SPIrxBuff[6];
```

Mit der Funktion SPIout(x) kann fortlaufend in den Tx-Buffer geschrieben werden. Ein Schreibzeiger wird automatisch mitgeführt. Der Rx-Buffer kann mit SPIinp gelesen werden, ein Lesezeiger wird fortlaufend mitgeführt.

```
for i:= 0 to 5 do
  SPIout ($ff);
endfor;
```

```
for i:= 1 to SPIRXLEN do
  bb:= SPIinp;
endfor;
```

SpiRxLen, SpiTxLen
Diese beiden Byte Variablen dürfen nur gelesen werden! SPIrxLen wird beim Empfang eines Frames automatisch mit dem korrekten Wert besetzt. SPItxLen wird durch das Kommando “SPItxFrame(n)” bestimmt.

3.17.3 Exportierte Funktionen und Prozeduren

Function *SpiRxStat : boolean;*
Diese Funktion gibt ein true zurück, wenn ein Frame empfangen wurde und der Lesezeiger (durch SpiInp bewegt) noch nicht auf das Frame Ende zeigt. (SPIrxLen > 0) and (SPIrxOutP < SPIrxLen)

Function *SpiInp : byte;*
Diese Funktion liest ein Byte aus dem RxBuffer an der Stelle SPIrxBuff[SPIrxOutP] und inkrementiert den Lesezeiger (SPIrxOutP). Sie sollte nur in Verbindung mit SPIrxStat verwendet werden.

Function *SpiRxFrame : boolean;*
Diese Funktion hat beim Master und Slave unterschiedliche Bedeutung. Beim **Slave** ist sie identisch mit „SPIrxStat“.

Beim **Master** wird ein true zurückgegeben, wenn ein Frame empfangen wurde und dieser noch nicht oder nicht ganz ausgelesen wurde (identisch mit SPIrxStat).
Ist der RxBuffer aber leer oder mit „SPIrxClear“ ungültig gemacht, geht der Master in Polling Betrieb und versucht vom aktuellen Slave ein Telegram zu erhalten. Dazu schickt er kontinuierlich (max. SPIretry) ein $FF an den Slave. Dieser antwortet entweder gar nicht, d.h. ein $FF kommt wieder zurück, oder er hat nichts zu senden (Antwort = $00), oder er schickt ein Byte > $00 und kleiner $FF.

Das bedeutet der Slave hat einen Frame zu senden und dieses Byte ist das Längenbyte des folgenden Frames. Der Master holt jetzt diese Anzahl von Bytes vom Slave ab, stellt sie in den RxBuffer, setzt den Lesepointer zurück und legt die Framelänge in SPIrxLen ab. Die Funktion kehrt darauf mit einem true zurück. Wurde beim Polling ein Retry Timeout erreicht, kehrt die Funktion mit einem false zurück.

Während des Empfangs des Frames vom Slave muss der Master jeweils ein Byte schicken, damit der Slave eines senden kann. Diese Bytes vom Master werden vom Slave ignoriert, da sie nur Dummies sind.
Während des ganzen Vorgangs befindet sich der Slave in seiner Interrupt Service Routine, die er erst wieder verlässt wenn alle Bytes übergeben sind oder der SS-Pin inaktiv (high) wird.

Function *SpiRxClear : boolean;*
Die Lese und Schreib Pointer sowie die Framelänge des RxBuffers werden zurückgesetzt. War der Frame schon leer wird ein false zurückgegeben, ansonsten ein true.

Function *SpiOnLine : boolean;*
Der **Master** schickt eine $00-Anfrage an den aktivierten Slave (SS-pin aktiv!). Der Slave antwortet mit einem $00. Hiermit kann festgestellt werden, ob der selektierte Slave online ist, d.h. bereit ist Daten zu empfangen oder zu senden. Das Ergebnis ist true, wenn der Slave aktiv ist ansonsten false. Das Resultat sagt nichts darüber aus, ob der Slave Daten zu schicken hat.

Es wird weder im Slave, noch im Master ein Buffer oder Status verändert.

Function *SpiTxStat : boolean;*
Diese Funktion prüft, ob ein zu sendender Frame gesendet ist oder nicht. Ist der Frame gesendet, kehrt die Funktion mit true zurück, ansonsten mit false.

Function *SpiOut (**const** b : byte) : boolean;*
Diese Funktion schreibt ein Byte in den TxBuffer an die Stelle SPIrxBuff[SPItxInP] und inkrementiert den Schreibzeiger (SPItxInP). Geschrieben wird nur, wenn der TxBuffer bereit ist, d.h. ein vorheriger Sende Vorgang mit „SPItxFrame“ wurde erfolgreich abgeschlossen. Weiterhin muss im TxBuffer noch Platz sein. Nach erfolgreichem Schreiben wird ein true zurückgegeben, ansonsten false.

Function *SpiTxFrame (**const** len : byte) : boolean;*
Diese Funktion hat beim Master und Slave unterschiedliche Bedeutung.

Beim **Slave** wird abgeprüft, ob der letzte Frame erfolgreich gesendet wurde. Wenn nicht, kehrt die Funktion mit false zurück. Wenn ja, wird der „len“ Parameter in „SPItxLen“ abgelegt. Das Ergebnis der Funktion ist jetzt true.

Wenn der Master jetzt einen Frame abfordert, prüft der Slave in seinem SPI-Interrupt Service den Parameter SPItxLen. Der Slave sendet als Acknowledge dieses Byte an den Master. Ist dieses Byte $00, verlässt der Slave den Interrupt, da nichts zu senden ist. Ansonsten wartet der Slave auf die Dummy Bytes des Masters und antwortet auf jedes Byte mit einem neuen Byte aus dem TxBuffer. Wurde der komplette Frame übertragen, setzt der Slave sein SPItxLen auf 0 zurück und verlässt den Interrupt Service.

Beim **Master** wird abgeprüft, ob der letzte Frame erfolgreich gesendet wurde. Wenn nicht, kehrt die Funktion mit false zurück. Die mögliche Ursache war ein TimeOut während des vorhergehenden SPItxFrame. Mit SPItxClear kann der Frame zurückgesetzt werden, ohne die eigentlichen Daten zu löschen, und SPItxFrame kann noch einmal aufgerufen werden. Wenn der Frame gesendet werden kann, wird der „len“ Parameter in „SPItxLen“ abgelegt.

Zum senden des Frames pollt der Master solange den aktivierten Slave mit $00, bis entweder ein TimeOut erfolgt (SPIretry) oder der Slave mit einem Byte > $00 und kleiner $FF antwortet. Dieses Byte ist die max. mögliche Frame Länge, die der Slave empfangen kann. Eine weitere Auswertung dieses Wertes erfolgt allerdings nicht. Es wird vorausgesetzt, dass der Master und alle Slaves die gleichen Buffer Längen haben.

Bei einem Timeout (SPIretry) kehrt die Funktion mit einem false zurück.

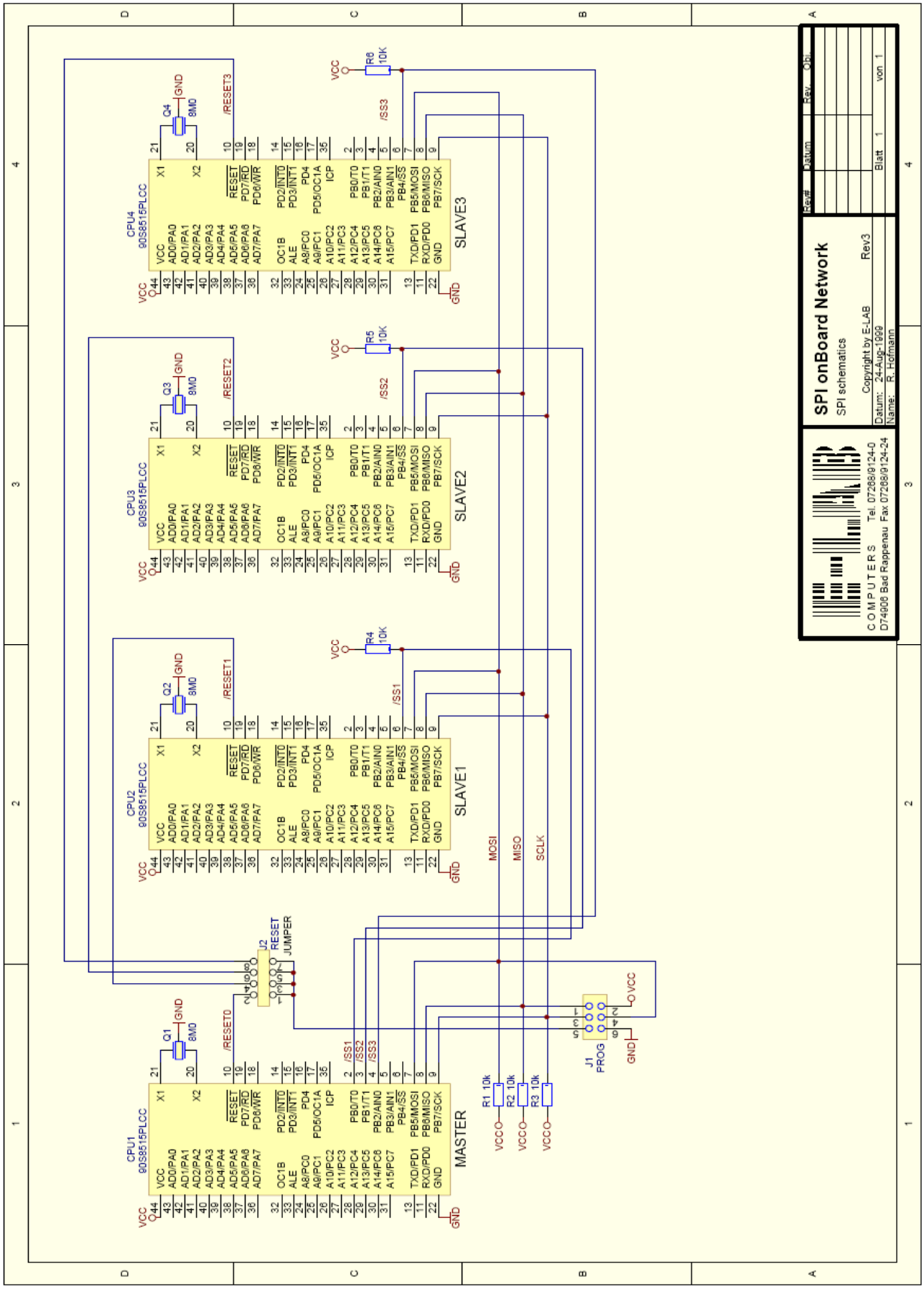
Hat der Slave mit einem Acknowledge geantwortet, sendet der Master zuerst die Frame Länge (SPItxLen) und dann die entsprechende Anzahl von bytes aus dem TxBuffer. Das Echo des Slaves wird dabei ignoriert, da der Slave sich jetzt in seiner Interrupt Service Routine befindet, und diese erst wieder verlässt wenn die korrekte Anzahl von bytes übertragen wurde oder sein SS-pin inaktiv wird.

Procedure *SpiTxClear;*
Die Lese und Schreib Pointer sowie die Framelänge des TxBuffers werden zurückgesetzt. War der Frame schon leer, wird ein false zurückgegeben, ansonsten ein true.

Bemerkungen

Der Slave beachtet während eines Transfers kontinuierlich seinen SS-Pin. Wird dieser inaktiv, so macht er während eines Frame Empfangs diesen Frame ungültig. Sendet der Slave gerade einen Frame, bricht er das Senden ab und setzt die Telegramm Länge auf die ursprüngliche Länge zurück. Das bedeutet, dass der Frame nicht gesendet wurde und ein neuer Versuch gestartet kann.

Alle SPI Pins, MISO, MOSI und SCLK sowie alle SS-Pins der Slaves sollten einen leichten PullUp Widerstand von 10k haben. Bei Leitungsbruch etc. kann dann zumindest der Master durch „SPIonLine“ feststellen, welche Slaves noch bereit sind .



Schaltplan **SPI**

3.18 SPI Low Level Treiber SPIdriver, SPI_C…SPI_F Hardware Version

Das AVRco System unterstützt neben dem Mini-Netzwerk ***SPIport*** auch direkt den Daten Transfer über die on-Chip SPI Schnittstelle des AVRs im Master Mode. Der hier beschriebene Treiber stellt die low-level Funktionen zum Lesen und Schreiben von Datenblöcken über die SPI Schnittstelle im Master Mode zur Verfügung. Es werden auch Änderungen des SPI Mode zur Laufzeit unterstützt.

Durch den Import des Treibers und die zugehörigen Defines wird auch das Setup des SPI vorgegeben. Es stehen drei unterschiedliche Funktionen zur Verfügung. Das für den angeschlossenen Slave immer notwendige Chip Select wird wie üblich durch den Treiber selbst gehandhabt. Hierzu wird das dafür vorgesehene PortBit ***SS*** des SPI Ports benutzt. Es können natürlich auch weitere Chipselects durch die Applikation selbst vor den Input und Output Funktionen aktiviert und anschliessend wieder deaktiviert werden.

Daten Quellen und Ziele der Lese und Schreib Block-Operationen können nur das RAM sein, EEprom und Flash sind nicht möglich.

Imports
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.

```
Import SysTick, SPIdriver, ..;
XMega
Import SysTick, SPI_C, ..;     // SPI_D, SPI_E, SPI_F
```

Defines
Die Bitrate, ClockPhase, ClockPolarity und MSB/LSB first müssen über das Define eingestellt werden, abhängig von den Vorgaben des SPI-Slaves.

```
Define   ProcClock   = 8000000;          {Hertz}
         SysTick     = 10;               {msec}
         StackSize   = $0030, iData;
         FrameSize   = $0030, iData;
         SPIorder    = MSB;
         SPIcpol     = 1;
         SPIcpha     = 1;
         SPIpresc    = 1;                // presc = 0..3 -> 4/16/64/128
         SPI_SS      = false;            // don't use SS pin as chipselect, not for XMega
```

Alternativ zu *SPIcpol* und *SPIcpha* kann der SPI Mode auch generell definiert werden:

```
Define   SPImode    = 0;                // 0, 1, 2, 3
```

```
XMega
Define   OSCtype    = int32MHz, PLLmul=4, prescB=1, prescC=1;  // all 32MHz
         SysTick    = 10;               {msec}
         StackSize  = $0030, iData;
         FrameSize  = $0030, iData;
         SPIorderC  = MSB;
         SPImodeC   = 0;                // Clock Phase and Polarity
         SPIprescC  = 1[, double];      // presc = 0..3 -> 4/16/64/128. Double speed optional
         SPI_SSC    = PortB, 3;         // use this pin as SS chipselect
   //    SPI_SSC    = none;             // application generates the SS chipselect
```

Bei den XMegas gibt es bis zu 4 separate SPI Ports. Diese sind jeweils einem I/O Port zugeordnet. PortC, PortD, PortE, PortF. Deshalb heissen diese SPIs hier SPI_C, SPI_D, SPI_E und SPI_F.
Bitte auch den Schalter *{$REUTILIZE xxx}* im Compiler Handbuch beachten.

Achtung:
Grundsätzlich muss bei allen AVRs und XMegas der eigentliche SS-Pin als Ausgang programmiert werden. Das System stellt dies sicher. Wird dieser Pin nicht als SS verwendet, so kann er als normaler Output verwendet werden.

Funktionen

Diese SPI Defines können zur Laufzeit geändert werden:

```
  Procedure SetSPIorder(msb : boolean);
  Procedure SetSPIclkPol(pol : byte);          // pol   = 0/1
  Procedure SetSPIclkPha(phase : byte);        // phase = 0/1
  Procedure SetSPIpresc(presc : byte);         // presc = 0..3 -> 4/16/64/128
  Procedure SetSPImode(mode : byte);           // mode = 0..3
XMega
  Procedure SetSPIorderC(msb : boolean);
  Procedure SetSPIorderD(msb : boolean);
  Procedure SetSPIorderE(msb : boolean);
  Procedure SetSPIorderF(msb : boolean);

  Procedure SetSPIclkPolC(pol : byte);         // pol   = 0/1
  Procedure SetSPIclkPolD(pol : byte);         // pol   = 0/1
  Procedure SetSPIclkPolE(pol : byte);         // pol   = 0/1
  Procedure SetSPIclkPolF(pol : byte);         // pol   = 0/1

  Procedure SetSPIclkPhaC(phase : byte);       // phase = 0/1
  Procedure SetSPIclkPhaD(phase : byte);       // phase = 0/1
  Procedure SetSPIclkPhaE(phase : byte);       // phase = 0/1
  Procedure SetSPIclkPhaF(phase : byte);       // phase = 0/1

  Procedure SetSPIprescC(presc : byte);        // presc = 0..3 -> 4/16/64/128
  Procedure SetSPIprescD(presc : byte);        // presc = 0..3 -> 4/16/64/128
  Procedure SetSPIprescE(presc : byte);        // presc = 0..3 -> 4/16/64/128
  Procedure SetSPIprescF(presc : byte);        // presc = 0..3 -> 4/16/64/128

  Procedure SetSPImodeC(mode : byte);          // mode = 0..3
  Procedure SetSPImodeD(mode : byte);          // mode = 0..3
  Procedure SetSPImodeE(mode : byte);          // mode = 0..3
  Procedure SetSPImodeF(mode : byte);          // mode = 0..3

  Procedure SPIout(source : pointer; count : word);
XMega
  Procedure SPIoutC(source : pointer; count : word);
  Procedure SPIoutD(source : pointer; count : word);
  Procedure SPIoutE(source : pointer; count : word);
  Procedure SPIoutF(source : pointer; count : word);
```

Schreibt einen Datenblock von der Stelle ***source*** mit der Länge ***count*** in den SPI-Slave.

```
  SPIout(@ArrXY, sizeOf (ArrXY));
XMega
  SPIoutC(@ArrXY, sizeOf (ArrXY));

  Procedure SPIinp(dest : pointer; count : word);
XMega
  Procedure SPIinpC(dest : pointer; count : word);
  Procedure SPIinpD(dest : pointer; count : word);
  Procedure SPIinpE(dest : pointer; count : word);
  Procedure SPIinpF(dest : pointer; count : word);
```

Liest einen Datenblock aus dem SPI-Slave in die Stelle ***dest*** mit der Länge ***count***

```
  SPIinp(@ArrXY, sizeOf (ArrXY));
XMega
  SPIinpC(@ArrXY, sizeOf (ArrXY));
```

Procedure *SPIinOut(source, dest : pointer; count : word);*
XMega
Procedure *SPIinOutC(source, dest : pointer; count : word);*
Procedure *SPIinOutD(source, dest : pointer; count : word);*
Procedure *SPIinOutE(source, dest : pointer; count : word);*
Procedure *SPIinOutF(source, dest : pointer; count : word);*
Schreibt einen Datenblock von der Stelle ***source*** mit der Länge ***count*** in den SPI-Slave und liest gleichzeitig einen Datenblock aus dem SPI-Slave in die Stelle ***dest*** mit der Länge ***count***
SPIinOut(@ArrXY, @RecordAB, 24);
XMega
SPIinOutC(@ArrXY, @RecordAB, 24);

Procedure *SPIoutByte(b : byte);*
XMega
Procedure *SPIoutByteC(b : byte);*
Procedure *SPIoutByteD(b : byte);*
Procedure *SPIoutByteE(b : byte);*
Procedure *SPIoutByteF(b : byte);*
Schreibt ein Byte in den SPI-Slave.
SPIoutByte($40);
XMega
SPIoutByteC($40);

Procedure *SPIoutWord(w : word);*
XMega
Procedure *SPIoutWordC(w : word);*
Procedure *SPIoutWordD(w : word);*
Procedure *SPIoutWordE(w : word);*
Procedure *SPIoutWordF(w : word);*
Schreibt ein Word in den SPI-Slave.
SPIoutWord($4080);
XMega
SPIoutWordC($4080);

Procedure *SPIoutLong(L : LongWord);*
XMega
Procedure *SPIoutLongC(L : LongWord);*
Procedure *SPIoutLongD(L : LongWord);*
Procedure *SPIoutLongE(L : LongWord);*
Procedure *SPIoutLongF(L : LongWord);*
Schreibt ein LongWord in den SPI-Slave.
SPIoutLong($12345678);
XMega
SPIoutLongC($12345678);

Function *SPIinpByte : byte;*
XMega
Function *SPIinpByteC : byte;*
Function *SPIinpByteD : byte;*
Function *SPIinpByteE : byte;*
Function *SPIinpByteF : byte;*
Liest ein Byte aus dem SPI-Slave.
bb:= SPIinpByte;
XMega
bb:= SPIinpByteC;

Function *SPIinpWord : word;*
XMega
Function *SPIinpWordC : word;*
Function *SPIinpWordD : word;*
Function *SPIinpWordE : word;*
Function *SPIinpWordF : word;*
Liest ein Word aus dem SPI-Slave.
ww:= SPIinpWord;
XMega
ww:= SPIinpWordC;

Function *SPIinpLong : longword;*
XMega
Function *SPIinpLongC : longword;*
Function *SPIinpLongD : longword;*
Function *SPIinpLongE : longword;*
Function *SPIinpLongF : longword;*
Liest ein LongWord aus dem SPI-Slave.
Lw:= SPIinpLong;
XMega
Lw:= SPIinpLongC;

Function *SPIinOutByte(b : byte) : byte;*
XMega
Function *SPIinOutByteC(b : byte) : byte;*
Function *SPIinOutByteD(b : byte) : byte;*
Function *SPIinOutByteE(b : byte) : byte;*
Function *SPIinOutByteF(b : byte) : byte;*
Schreibt ein Byte in den SPI-Slave und liest gleichzeitig ein Byte aus dem SPI-Slave.
bb:= SPIinOutByte($33);
XMega
bb:= SPIinOutByteC($33);

Procedure *SetSPIdoubleSpeed(ds : boolean);*
XMega
Procedure *SetSPIdoubleSpeedC(ds : boolean);*
Procedure *SetSPIdoubleSpeedD(ds : boolean);*
Procedure *SetSPIdoubleSpeedE(ds : boolean);*
Procedure *SetSPIdoubleSpeedF(ds : boolean);*
Schaltet die SPI Datenrate zwischen doppelt und einfach.
SetSPIdoubleSpeed(true); // switch to double speed
XMega
SetSPIdoubleSpeedC(true); // switch to double speed

Xmega only
Function *SPIinpLong64C : longword;*
Function *SPIinpLong64D : longword;*
Function *SPIinpLong64E : longword;*
Function *SPIinpLong64F : longword;*
Liest ein 64bit Word aus dem SPI-Slave.
W64:= SPIinpLong64C;

Procedure *SPIoutLong64C(L : Word64);*
Procedure *SPIoutLong64D(L : Word64);*
Procedure *SPIoutLong64E(L : Word64);*
Procedure *SPIoutLong64F(L : Word64);*
Schreibt ein LongWord in den SPI-Slave.
SPIoutLong64($123456780ABCD);

Der SPI-SS Pin wird per Default als Chipselect für den Slave benutzt. Mit dem SPI_SS Define kann das unterbunden werden. Dann muss die Applikation selbst die Chipselects steuern.

3.19 MSPI Low Level SPI Treiber MSPI_0..MSPI_3 *AVR*

Nicht für XMegas!

Verschiedene AVR Typen erlauben es den/die UARTs als SPI Master zu betreiben. Dies wird in den Datenblättern unter „USART in SPI Mode“ näher beschrieben. Dort wird das als ***MSPIM*** Mode bezeichnet.

Das AVRco System unterstützt neben dem Standard Hardware/Software ***SPI*** auch direkt den Daten Transfer über die on-Chip UART Schnittstelle des AVRs im ***Master Mode***. Der hier beschriebene Treiber stellt die low-level Funktionen zum Lesen und Schreiben von Datenblöcken über die SPI-USART Schnittstelle im Master Mode zur Verfügung. Es werden auch Änderungen des SPI Mode zur Laufzeit unterstützt. Die Treiber werden als MSPI_0, MSPI_1, MSP_2 etc. bezeichnet, abhängig davon welcher USART benutzt werden soll.

Da bis zu 4 USARTS in einem AVR vorhanden sein können und diese alle, falls auch so implementiert, in den MSPIM Mode geschaltet werden können, bietet das System auch bis zu 4 MSPI Treiber die praktisch identisch sind und eine angehängte Ziffer 0..3 unterschieden werden. Z.B MSP0, MSP1 oder MSPIout0 oder MSPIout1 etc. Im folgenden werden diese möglichen Indizes 0..3 durch ein ***n*** deutlich gemacht.

Durch den Import des Treibers und die zugehörigen Defines wird auch das Setup des SPI vorgegeben. Es stehen drei unterschiedliche Funktionen zur Verfügung. Das für den angeschlossenen Slave immer notwendige Chip Select wird **nicht** durch den Treiber gehandhabt, da hier kein Pin zugeordnet ist. Die Applikation muss dass selbst handhaben, indem sie vor jedem IN oder OUT Kommando ein Chipselect aktiviert bzw. deaktiviert.
Der TxD***n*** Pin wird zu MOSI***n***, der RXD***n*** Pin zu MISO***n*** und XCK***n*** Pin zu SCK***n***.

Daten Quellen und Ziele der Lese und Schreib Block-Operationen können nur das RAM sein, EEprom und Flash sind nicht möglich.

Imports
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.

```
Import SysTick, MSPI_0, MSPI_1,  ..;
```

Defines
Die Bitrate, ClockPhase, ClockPolarity und MSB/LSB first müssen über das Define eingestellt werden, abhängig von den Vorgaben des SPI-Slaves. Sie können auch zur Laufzeit geändert werden.

```
Define   ProcClock    = 8000000;          {Hertz}
         SysTick      = 10;               {msec}
         StackSize    = $0030, iData;
         FrameSize    = $0030, iData;
         MSPIorder0 = MSB;
         MSPIcpol0   = 1;
         MSPIcpha0  = 1;
         MSPIpresc0 = 1;                  // presc = 0..255

         MSPIorder1 = MSB;
         MSPIcpol1   = 1;
         MSPIcpha1  = 1;
         MSPIpresc1 = 1;                  // presc = 0..255
         ....
```

Alternativ zu *MSPIcpolx* und *MSPIcphax* kann der SPI Mode auch generell definiert werden:

```
Define   MSPImode0  = 0;                  // 0, 1, 2, 3
```

3.19.1 Funktionen

Diese MSPI Defines können zur Laufzeit geändert werden:

Procedure *MSetSPIorder**n** (msb : boolean);*
Procedure *MSetSPIclkPol**n** (pol : byte);* *// pol = 0/1*
Procedure *MSetSPIclkPha**n** (phase : byte);* *// phase = 0/1*
Procedure *MSetSPIpresc**n** (presc : byte);* *// presc = 0..255*
Procedure *SetMSPI0mode(mode : byte); // mode = 0..3*
Procedure *SetMSPI1mode(mode : byte); // mode = 0..3*
Procedure *SetMSPI2mode(mode : byte); // mode = 0..3*
Procedure *SetMSPI3mode(mode : byte); // mode = 0..3*

Procedure MSPIout***n*** (source : pointer; count : word);
Schreibt einen Datenblock von der Stelle ***source*** mit der Länge ***count*** in den MSPI-Slave.
MSPIout0 (@ArrXY, sizeOf (ArrXY));

Procedure MSPIinp***n*** (dest : pointer; count : word);
Liest einen Datenblock aus dem MSPI-Slave in die Stelle ***dest*** mit der Länge ***count***
MSPIinp1 (@ArrXY, sizeOf (ArrXY));

Procedure MSPIinOut***n*** (source, dest : pointer; count : word);
Schreibt einen Datenblock von der Stelle ***source*** mit der Länge ***count*** in den MSPI-Slave und liest gleichzeitig einen Datenblock aus dem MSPI-Slave in die Stelle ***dest*** mit der Länge ***count***
MSPIinOut2 (@ArrXY, @RecordAB, 24);

Procedure MSPIoutByte***n*** (b : byte);
Schreibt ein Byte in den MSPI-Slave.
MSPIoutByte3 ($40);

Procedure MSPIoutWord***n*** (w : word);
Schreibt ein Word in den MSPI-Slave.
MSPIoutWord0 ($4080);

Procedure MSPIoutLong***n*** (L : longword);
Schreibt ein LongWord in den MSPI-Slave.
MSPIoutLong1 ($12345678);

Function MSPIinpByte***n*** : byte;
Liest ein Byte aus dem MSPI-Slave.
bb:= MSPIinpByte2;

Function MSPIinpWord***n*** : word;
Liest ein Word aus dem MSPI-Slave.
ww:= MSPIinpWord3;

Function MSPIinpLong***n*** : longword;
Liest ein LongWord aus dem MSPI-Slave.
Lw:= MSPIinpLong0;

Function MSPIinOutByte***n*** (b : byte) : byte;
Schreibt ein Byte in den MSPI-Slave und liest gleichzeitig ein Byte aus dem MSPI-Slave.
bb:= MSPIinOutByte1 ($33);

3.20 MSPI Low Level SPI Treiber MSPI_C0..MSPI_F1 *XMega*

Alle XMega Typen erlauben es den/die UARTs als SPI Master zu betreiben. Dies wird in den Datenblättern unter „USART in SPI Mode“ näher beschrieben. Dort wird das als ***MSPIM*** Mode bezeichnet.

Das AVRco System unterstützt neben dem Standard Hardware/Software ***SPI*** auch direkt den Daten Transfer über die on-Chip UART Schnittstelle der XMegas im ***Master Mode***. Der hier beschriebene Treiber stellt die low-level Funktionen zum Lesen und Schreiben von Datenblöcken über die SPI-USART Schnittstelle im Master Mode zur Verfügung. Es werden auch Änderungen des SPI Mode zur Laufzeit unterstützt. Die Treiber werden als MSPI_C0, MSPI_C1, MSP_D0 etc. bezeichnet, abhängig davon welcher USART benutzt werden soll. Die Endung C0 etc. wird hier im weiteren der Einfachheit halber als ***xx*** bezeichnet. MSPI_***xx***

Alle XMega USARTS können, falls diese den MSPIM Mode unterstützen, auch als MSPI benutzt werden. Damit bietet das System auch bis zu 8 MSPI Treiber an, die praktisch identisch sind und durch eine angehängte Endung C0..F1 unterschieden werden. Z.B MSPI_C0, MSPI_C1 oder MSPIout_C0 oder MSPIout_C1 etc. Im folgenden werden diese möglichen Indizes durch ein ***xx*** deutlich gemacht.

Durch den Import des Treibers und die zugehörigen Defines wird auch das Setup des MSPI vorgegeben. Es stehen drei unterschiedliche Funktionen zur Verfügung. Das für den angeschlossenen Slave immer notwendige Chip Select wird **immer** durch den Treiber gehandhabt. Dazu muss im Define das gewünschte Port und der Pin vorgegeben werden.
Der TxD***xx*** Pin wird zu MOSI***xx***, der RXD***xx*** Pin zu MISO***xx*** und XCK***xx*** Pin zu SCK***xx***.

Daten Quellen und Ziele der Lese und Schreib Block-Operationen können nur das RAM sein, EEprom und Flash sind nicht möglich. Weiterhin können DMA Kanäle bei Block-IOs benutzt werden.

Imports
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.
Import *SysTick, MSPI_C0, MSPI_C1, ..;*

Defines
Die Bitrate, den SPImode (ClockPhase, ClockPolarity) und MSB/LSB first müssen über das Define eingestellt werden, abhängig von den Vorgaben des SPI-Slaves. Sie können auch zur Laufzeit geändert werden. Der Pin und das Port für das /CS (SS) müssen definiert werden. Wenn hier “**none**” vorgegeben wird dann muss die Applikation das **SS** selbst bedienen.
Define

```
// The XMegas don't provide any Oscillator fuses.
// So the application must setup the desired values

//>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
        OSCtype         =  int32MHz,
                           PLLmul=4,
                           prescB=1,
                           prescC=1;
        SysTick         = 10;              {msec}
        StackSize       = $0030, iData;
        FrameSize       = $0030, iData;
        MSPIorder_C0    = MSB;
        MSPIMode_C0     = 1;               // 0, 1, 2, 3
        MSPIpresc_C0    = 1;               // presc = 0..255
        MSPI_SS_C0      = PortE, 4;

        MSPIorder_C1    = MSB;
        MSPImode_C1     = 0;               // 0, 1, 2, 3
        MSPIpresc_C1    = 3;               // presc = 0..255
        MSPI_SS_C1      = none;

        Optional
        MSPIDMA_OUT_C1  = DMAch0;   // DMAch0 for BlockOut functions
        ....
```

3.20.1 Funktionen

Diese MSPI Defines können zur Laufzeit geändert werden:
Procedure *SetMSPIorder_xx(msb : boolean);*
Procedure *SetMSPIclkPol_xx(pol : byte);* *// pol = 0/1*
Procedure *SetMSPIclkPha_xx(phase : byte);* *// phase = 0/1*
Procedure *SetMSPIpresc_xx(presc : byte);* *// presc = 1..255*
Procedure *SetMSPImode_xx(mode : byte);* *// mode = 0..3*

DMA:
Für OutBlock kann einer der 4 möglichen DMA Kanäle benutzt werden
Für InpBlock kann einer der 2 möglichen DMA Paare (0+1 oder 2+3) benutzt werden.
Der benutzte DMA muss im Define mit angegeben werden.
Define MSPIDMA_OUT_ xx = DMAch0; DMAch0 für BlockOut Funktionen
Define MSPIDMA_INP_ xx = DMAch2; DMAch2+3 für BlockInp Funktionen

Procedure MSPIout_xx (source : pointer; count : word); // DMA can be used
Schreibt einen Datenblock von der Stelle ***source*** mit der Länge ***count*** in den MSPI-Slave.
MSPIout_C0(@ArrXY, sizeOf (ArrXY));

Procedure MSPIinp_xx (dest : pointer; count : word); // DMA can be used
Liest einen Datenblock aus dem MSPI-Slave in die Stelle ***dest*** mit der Länge ***count***
MSPIinp_C1(@ArrXY, sizeOf (ArrXY));

Procedure MSPIinOut_xx(source, dest : pointer; count : word); // no DMA possible
Schreibt einen Datenblock von der Stelle ***source*** mit der Länge ***count*** in den MSPI-Slave und liest gleichzeitig einen Datenblock aus dem MSPI-Slave in die Stelle ***dest*** mit der Länge ***count***
MSPIinOut_D0(@ArrXY, @RecordAB, 24);

Procedure MSPIoutByte_xx(b : byte);
Schreibt ein Byte in den MSPI-Slave.
MSPIoutByte_D1($32);

Procedure MSPIoutWord_xx(w : word);
Schreibt ein Word in den MSPI-Slave.
MSPIoutWord_E0($4080);

Procedure MSPIoutLong_xx(L : LongWord);
Schreibt ein LongWord in den MSPI-Slave.
MSPIoutLong_E1($12345678);

Function MSPIinpByte_xx : byte;
Liest ein Byte aus dem MSPI-Slave.
bb:= MSPIinpByte_E1;

Function MSPIinpWord_xx : word;
Liest ein Word aus dem MSPI-Slave.
ww:= MSPIinpWord_F0;

Function MSPIinpLong_xx : LongWord;
Liest ein LongWord aus dem MSPI-Slave.
Lw:= MSPIinpLong_F1;

Function MSPIinOutByte_***xx***(b : byte) : byte;
Schreibt ein Byte in den MSPI-Slave und liest gleichzeitig ein Byte aus dem MSPI-Slave.
bb:= MSPIinOutByte_E1($24);

DMA:
Ein DMA Transfer kann nur auf Block Funktionen angewendet werden.

Wurde ein DMAout channel definiert
Define *MSPIDMA_OUT_ xx = DMAchX*
So wird der DMA nur für die Funktion
Procedure *MSPIout_xx(..)*
angewendet.

Wurde ein DMAinp channel definiert
Define *MSPIDMA_INP_ xx = DMAchX*
So wird der DMA nur für die Funktion
Procedure *MSPIinp_xx(…);*
angewendet.

Beide Funktionen kehren nur zurück wenn der komplette Transfer abgewickelt wurde. Dieses Warten kann abgeschaltet werden indem eine separate (optionale) Busy Funktion aufgerufen werden:
Function *MSPI_DMAready_xx : boolean;*
Diese Funktion kehrt mit einem false zurück solange der Transfer noch läuft. Ist der Transfer abgeschlossen, dann kommt ein true zurück.

Achtung:
Da der DMA sich den Adress- und Datenbus mit der CPU teilen muss, wird der Transfer gebremst.
Die Transfer Zeit beim DMA ist dadurch Faktor 3 länger als ohne DMA!

Beispiel Programme:
Beispiele sind in den Directories **..\E-Lab\AVRco\Demos\XMega_MSPI**
und **..\E-Lab\AVRco\Demos\XMega_MSPI_DMA**

3.21 SPI Low Level Treiber SPIdriver Software Version

Zusätzlich zum direkten SPIdriver, der das SPI Port des AVR benutzt stellt das AVRco System auch zwei reine Software SPI Schnittstellen im Master Mode zur Verfügung. Der hier beschriebene Treiber stellt die low-level Funktionen zum Lesen und Schreiben von Datenblöcken über die beiden Software SPI Schnittstellen im Master Mode zur Verfügung.

Durch den Import des Treibers und die zugehörigen Defines wird auch das Setup der Software SPI vorgegeben. Es stehen drei unterschiedliche Funktionen zur Verfügung. Das für den angeschlossenen Slave immer notwendige Chip Select wird wie üblich durch den Treiber selbst gehandhabt.

Daten Quellen und Ziele der Lese und Schreib Block-Operationen können nur das RAM sein, EEprom und Flash sind nicht möglich.

Import
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.

```
Import SysTick, SPIdriver1, ..;          // oder SPIdriver2 oder beide
```

Define
Die ClockPhase, ClockPolarity und MSB/LSB first müssen über das Define eingestellt werden, abhängig von den Vorgaben des SPI-Slaves.

```
Define
    ProcClock    = 16000000;          {Hertz}
    SysTick      = 10;                {msec}
    StackSize    = $0030, iData;
    FrameSize    = $0040, iData;

    SPIdriver1   = PortA, 0, 1, 2, 3;   // SCK, MOSI, MISO, SS
    SPIorder1    = MSB;
    SPIcpol1     = 1;
    SPIcpha1     = 1;

    SPIdriver2   = PortC, 7, 4, 1, 5;   // SCK, MOSI, MISO, SS
    SPIorder2    = MSB;
    SPIcpol2     = 0;
    SPIcpha2     = 0;
```

Alternativ zu *SPIcpolx* und *SPIcphax* kann der SPI Mode auch generell definiert werden:

```
Define   SPImode1    = 0;                  // 0, 1, 2, 3
```

Das Define SPIdriverX beschreibt die Bit-Funktionen des zu verwendenden IO-Ports.
Zuerst kommt der Port Name. Dann folgt die Position des Clock Pins, des MOSI Pins und des MISO Pins, dann die Position des SS-Pins (Chip select). Dieser ist optional.

Alle Pins bzw. Bits müssen in einem Port sein, splitting ist nicht möglich. Die Bits können jedoch beliebig über das Port verteilt sein.

Die Bedeutung der Defines SPIcpol (SPI Clock Polarity) und SPIcpha (SPI Clock Phase) bzw. des SPI Modes kann einem AVR Datenblatt entnommen werden.

Die Definition des SS Pins (chip select) kann auch weggelassen werden. Dann erzeugt der Treiber keine Chip Selects. Das ist dann Aufgabe der Applikation. Ist der SS Pin definiert, dann aktiviert der Treiber vor jedem Slave Zugriff den SS Pin und deaktiviert ihn nach dem Zugriff wieder.

Die SPI Datenrate ist nicht einstellbar und beträgt ca. 1/16 des Prozessor Clocks. Bei 16MHz ist die Bitrate deshalb ca. 1Mbit/sec

3.21.1 Funktionen

Diese SPI Defines können zur Laufzeit nicht geändert werden.

Procedure SPIout1 (source : pointer; count : word);
Procedure SPIout2 (source : pointer; count : word);
Schreibt einen Datenblock von der Stelle ***source*** mit der Länge ***count*** in den SPI-Slave.
SPIout1 (@ArrXY, sizeOf (ArrXY));

Procedure SPIinp1 (dest : pointer; count : word);
Procedure SPIinp2 (dest : pointer; count : word);
Liest einen Datenblock aus dem SPI-Slave in die Stelle ***dest*** mit der Länge ***count***
SPIinp2 (@ArrXY, sizeOf (ArrXY));

Procedure SPIinOut1 (source, dest : pointer; count : word);
Procedure SPIinOut2 (source, dest : pointer; count : word);
Schreibt einen Datenblock von der Stelle ***source*** mit der Länge ***count*** in den SPI-Slave und liest gleichzeitig einen Datenblock aus dem SPI-Slave in die Stelle ***dest*** mit der Länge ***count***
SPIinOut1 (@ArrXY, @RecordAB, 24);

Procedure SPIoutByte1 (b : byte);
Procedure SPIoutByte2 (b : byte);
Schreibt ein Byte in den SPI-Slave.
SPIoutByte2 ($40);

Procedure SPIoutWord1 (w : word);
Procedure SPIoutWord2 (w : word);
Schreibt ein Word in den SPI-Slave.
SPIoutWord1 ($4080);

Procedure SPIoutLong1 (L : longword);
Procedure SPIoutLong2 (L : longword);
Schreibt ein LongWord in den SPI-Slave.
SPIoutLong2 ($12345678);

Function SPIinpByte1 : byte;
Function SPIinpByte2 : byte;
Liest ein Byte aus dem SPI-Slave.
bb:= SPIinpByte1;

Function SPIinpWord1 : word;
Function SPIinpWord2 : word;
Liest ein Word aus dem SPI-Slave.
ww:= SPIinpWord2;

Function SPIinpLong1 : longword;
Function SPIinpLong1 : longword;
Liest ein LongWord aus dem SPI-Slave.
Lw:= SPIinpLong1;

Function SPIinOutByte1 (b : byte) : byte;
Function SPIinOutByte2 (b : byte) : byte;
Schreibt ein Byte in den SPI-Slave und liest gleichzeitig ein Byte aus dem SPI-Slave.
bb:= SPIinOutByte2 ($33);

Single-Bit Ausgaben:
Procedure SPIoutBit1(b : boolean);
Procedure SPIoutBit2(b : boolean);
Schreibt ein Bit in einen Slave. Zur Verwendung von z.B. 9bit Ausgabe.

Beispiel in XMega_SoftSPI

3.22 Serial SPI Flash AT25DF (XMega)

Manche Applikationen benötigen sehr grosse Konstanten, z.B. das EVE Graphic System wo grosse Bitmaps und riesige Fonts gebraucht werden. Auch HTML Seiten können viele kBs gross sein. Speichern und Zurücklesen von sehr grossen Datenmengen in einem sehr preiswerten SPI serial Flash Speicher ist eine Lösung dieses Problems.
Das hier unterstützte Device ist der AT25DF321 32MBit = 4MByte und der AT25DF641 64Mbit = 8MByte serial flash Baustein. Beide chips sind praktisch identische 8pin SOIC Bauteile. Optional können 2 ROMs gesteuert werden. Wie bei allen Flash Bauteilen ist der Lese Vorgang random und erfolgt mit very high speed. Einzelne Bytes und Blocks bis zu 64kBytes können mit einem Zugriff gelesen warden mit bis zu 50Mbit/sec. Der XMega hier ist aber limitiert auf 16MBit/sec.

Der Nachteil bei Flash Bausteinen ist dass non-empty blocks zuerst gelöscht werden müssen bevor sie wieder beschrieben werden können. Deshalb ist die Haupt Anwendung für solche Speicher ein read-only System. Der Inhalt des Flash wird einmal geschrieben und oft gelesen.

Import
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.
Import *SysTick, AT25DFxx, ..;*

Define
Der Treiber benötigt das Define eines Hardware oder Software SPI:
Define
AT25DFxx = SPI_E, PortE.3; // hardware SPI, /CS
oder Software SPI
// SS SCLK MOSI MISO
AT25DFxx = SPI_Soft, PortE.3, PortE.7, PortE.5, PortE.6;
*AT25DF_ROM2 = PortE.2; // **optional** 2nd chip select*

Low Level Funktionen
Diese Funktionen werden normalerweise von der Applikation selbst nicht gebraucht. Sie dienen nur für ganz bestimmte Fälle.

procedure *AT25_CS_Hi;*
Der SPI-Chipselect Ausgang wird auf log1 (inaktiv) gesetzt.

procedure *AT25_CS_Lo;*
Der SPI-Chipselect Ausgang wird auf log0 (aktiv) gesetzt.

procedure *AT25_outByte(b : byte);*
Das Byte b wird über SPI-MOSI ausgegeben.

function *AT25_inpByte : byte;*
Die Funktion liest ein Byte über SPI-MISO ein.

procedure *AT25_outAddr(a : longword);*
Die Flash-Adresse a wird an den AT25 ausgegeben

procedure *AT25_write_enable;*
Das Write-enable im AT25 wird aktiviert.

procedure *AT25_write_disable;*
Das Write-enable im AT25 wird deaktiviert.

Support Funktionen
procedure *AT25_SelectROM(rom : byte);*
Wenn das optionale 2te ROM definiert wurde, muss das gewünschte ROM damit vorgegeben werden.

procedure *AT25_checkBusy;*
Das Busy Flag im AT25 wird solange abgefragt bis es inaktiv wird.

procedure *AT25_Reset;*
Diese Funktion dient dazu alle noch evtl. im AT25 laufende Operationen (write, erase etc) abzubrechen.

function *AT25_readStatus : word;*
Der Status des AT25 wird gelesen. Weitere Infos dazu im AT25 Datenblatt.

procedure *AT25_protect_sector(sect : longword);*
Ein 64kb grosser Flash Sector im AT25 wird schreib- und löschgeschützt. “sect“ muss in einen 64kB grossen Bereich zeigen. Die Anfangs Adresse des Blocks ist nicht erforderlich.

procedure *AT25_unprotect_sector(sect : longword);*
Ein 64kb grosser Flash Sector im AT25 wird zum schreiben und löschen freigegeben. “sect“ muss in einen 64kB grossen Bereich zeigen. Die Anfangs Adresse des Blocks ist nicht erforderlich.

procedure *AT25_protect_all;*
Das komplette Flash im AT25 wird schreib- und löschgeschützt.

procedure *AT25_unprotect_all;*
Das komplette Flash im AT25 wird zum schreiben und löschen freigegeben.

function *AT25_ GetDeviceDensity : byte;*
Die Funktion gibt die Flashgrösse des AT25 in Mbits zurück. 32..64

Haupt Funktionen
procedure *AT25_chip_erase;*
Diese Funktion führt einen kompletten Chip-Erase durch, d.h. alle Flash Zellen werden auf $FF gesetzt. Dies kann bis zu 25sec dauern.

procedure *AT25_4kb_erase(addr : longword);*
Ein 4kByte grosser Block im Flash wird gelöscht. “addr“ muss in den gewünschten Block zeigen.

procedure *AT25_32kb_erase(addr : longword);*
Ein 32kByte grosser Block im Flash wird gelöscht. “addr“ muss in den gewünschten Block zeigen.

procedure *AT25_64kb_erase(addr : longword);*
Ein 64kByte grosser Block im Flash wird gelöscht. “addr“ muss in den gewünschten Block zeigen.

procedure *AT25_read_array(addr : longword; buf : pointer; count : word);*
Diese Funktion liest einen bis zu 64kB grossen Flash Inhalt aus dem AT25. “addr“ bezeichnet eine beliebige Start Adresse. “buf“ ist ein Pointer auf einen Bereich im iData wohin geschrieben werden soll. “count“ ist die Anzahl der zu lesenden Bytes.

procedure *AT25_write_page(addr : longword; buf : pointer);*
Diese Funktion schreibt einen 256 Byte grossen Bereich aus iData in den AT25. “addr“ ist die Ziel Addresse im Flash und muss auf den Beginn eines 256byte Blocks im AT25 zeigen (ein vielfaches von 256).

procedure *AT25_write_bytes(addr : longword; buf : pointer; count : byte);*
Diese Funktion schreibt einen bis 256 Byte grossen Bereich aus iData in den AT25. “addr“ ist die Ziel Addresse im Flash. Es kann nur in einen 256byte grossen Bereich geschrieben werden. Ist “count“ = 256 dann muss “addr“ auf den Beginn eines 256byte blocks im Flash zeigen. Ist “count“ = 1 dann kann mit “addr“ jedes beliebige Byte in dem betroffen 256byte Block geschrieben werden. “buf“ zeigt auf die Quelle im iData Bereich.
Wie bei allen Flash Schreib Operationen muss sichergestellt werden, dass dass die zu schreibende Speicherstelle(n) schon gelöscht sind ($FF). Das kann z.B. mit Chip-Erase oder einem Block-Erase gemacht werden. Soll gepatcht werden, also nur ein paar Bytes in einem schon beschriebenen Bereich mit neuem Inhalt überschrieben werden, so muss z.B. der dadurch betroffene 4kB Block ausgelesen werden. Dann wird dieser im iData verändert und nachdem dieser Block im AT25 gelöscht wurde, kann er jetzt neu geschrieben werden. Es müssen dann wieder alle 4kB zurückgeschrieben werden.

Beispiel Programm:
Ein Beispiel ist in der Directory **..\E-Lab\AVRco\Demos\XMega_SerFlash**
Ein zweites Beispiel ist in der Directory **..\E-Lab\AVRco\Demos\XMega_SerFlash2**

3.23 TWI (I2C) Treiber Master und Slave

Die I2C (TWI) Schnittstelle ist eine international standardisierte, synchrone serielle Schnittstelle zum Anschluss von Peripheriebausteinen an einen Prozessor und auch zur Kommunikation von Prozessoren untereinander. Die hier vorliegende Implementation unterstützt den Master Mode und den Slave Mode. Vorraussetzung für beide Modi ist, dass die verwendete AVR-CPU ein TWI Interface besitzt (z.B. mega16).

Es können mehrere Slaves an einen TWI-Master angeschlossen werden, wobei jeder Slave eine spezielle, interne Adresse besitzt, die im sog. Netz nur einmal vorkommen sollte. Da theoretisch über 100 Bausteine betrieben werden können, kann man auch von einem Netzwerk sprechen. Die Variante mit 10-Bit Adressen wird nicht unterstützt.

Die Verbindung untereinander geschieht mittels zwei Steuerleitungen: Daten und Takt. Diese beiden Leitungen sind bi-direktional, arbeiten also in beiden Richtungen. Beide Leitungen müssen mit jeweils einem Pull-Up Widerstand von 1..10kOhm gegen +5Volt versehen sein.

Slave Adressen
Zu beachten ist bei den Adressangaben immer, dass in den Datenblättern die Baustein Adresse um 1 nach links geschoben ist, da das R/W bit in das Bit0 kopiert wird. Ein Slave der die Adresse R/W $80..$81 hat, hat jedoch in der AVRco Implementation die Adresse $40, da das R/W bit nicht mit angegeben werden darf.

Die Slave Adresse $00 kommt in keinem Baustein vor. Sie ist eine sogenannte **Global Call** Adresse, was mit einer Broadcast Message in Netzwerken zu vergleichen ist. Angeschlossen Slave CPUs können jedoch meistens diese Adresse auswerten. Weiteres darüber in dem Slave Kapitel.

3.23.1 Master Betrieb

In dieser Implementation arbeitet ein Master im Polling verfahren, d.h. es werden keine Interrupts benutzt. Es werden keine Retries ausgeführt, wenn ein Bus-Error auftritt.

Lock-up im Master Betrieb
Die interne TWI Hardware der mega CPUs reagiert sehr empfindlich auf externe Störungen, vor allem der CLK Leitung. Das kann u.U. zum kompletten Ausfall führen. Deshalb ist an diversen Stellen ein Timeout in den Treiber eingebaut. Tritt so ein TimeOut auf, wird die TWI Hardware neu initialisiert und die Library Funktion kehrt mit einem false zurück. Ist die externe Störung beseitigt, arbeitet das System ohne weitere Intervention der Applikation weiter. Um diese TimeOut Funktion nützen zu können, muss der ***SysTick*** importiert sein.

3.23.2 Slave Betrieb

In dieser Implementation arbeitet ein Slave immer im Interrupt Verfahren.

TWI (I2C) Master
Die durch **TWImaster** importierte und durch **TWIpresc** definierte TWI-Schnittstelle bietet die drei High-Level Funktionen TWIstat, TWIinp und TWIout. Diese Funktionen sind generalisiert, d.h. sie sind auf keinen speziellen Slave Chip angepasst. Der Programmierer kann bzw. muss durch entsprechende Verwendung und Parameter das Software-Protokoll des Slave einhalten, wobei das Telegramm-Protokoll (Hardware-Adresse, Clock-Generierung etc) durch den Bibliothekstreiber realisiert wird.

Durch das vorgeschriebene Hardware-Protokoll erkennen alle Funktionen, ob der Slave vorhanden und bereit oder nicht vorhanden oder BUSY ist. Ist die Funktion fehlgeschlagen wird ein FALSE zurückgeliefert, ansonsten ein TRUE.

Die zwei Transfer-Funktionen bieten die Möglichkeit Blöcke zu lesen und zu schreiben, wobei der Typ des Parameters "Data" automatisch die Blocklänge bestimmt. Weiteres unter TWIinp und TWIout.

Imports

TWImaster
Definiert und importiert die Master TWI-Bus Schnittstelle. Bei den XMegas muss eines der TWI Ports importiert werden.

Der TWI-Bus benutzt den onChip TWI Kanal. Die Schnittstelle arbeitet dabei nur als Bus-Master.
Eine Umschaltung vom Master in den Slave Betrieb ist nicht implementiert. Dafür sollte der AVRco TWI-LAN Treiber benutzt werden.

Wird mittels Import der TWImaster importiert, so muss auch mittels Define der zugehörige onChip Prescaler definiert werden, der die Bitrate des TWI Kanals bestimmt.

Der Import von TWImaster importiert auch automatisch die Bibliotheksfunktionen TWIstat, TWIinp und TWIout. Diese Funktionen unterstützen neben Einzelbyte-Transfers auch Block-Transfers. Die Funktionen liefern entsprechend dem Ergebnis ein True oder False zurück.

```
Import   SysTick, TWImaster;
Define   ProcClock = 4000000;  {4Mhz clock }
         SysTick   = 10;       {10msec Tick}
         TWIpresc  = 32        (TWI speed}
XMega
Import   SysTick, TWI_C;                          // TWI_C, TWI_D, TWI_E, TWI_F
Define   OSCtype       = int32MHz, PLLmul=4, prescB=1, prescC=1;
         SysTick       = 10;                      {10msec Tick}
         TWIprescC     = TWI_BR400;               (TWI speed}
         optional
         TWInoTimeOutC  = false;                        // or  true, default = false
```

TWIpresc
Definiert den Wert des internen TWI Vorteilers, der die Bitrate des TWIbus bestimmt. Ein hoher Wert reduziert die Geschwindigkeit. Ein niedriger Wert erhöht diese. Zu beachten ist, dass die meisten I2C Slaves nach Datenblatt eine Bitrate von max. 100kBit/sec erreichen, in der Praxis aber diese Bitrate von der Leitungslänge, den Pullups etc. eventuell eingeschränkt wird.

Moderne I2C Chips können bis zu 400kBit/sec erreichen. Dann wird aber u.U. die Leitungslänge kritisch. Zumindest muss man hier mit Pullups von 1..2kOhm arbeiten wobei hier die parallele Summe aller Widerstände gemeint ist.

Das System exportiert im Master Mode auch noch die Konstanten

```
Const  TWI_BR100 : byte = nn;      // nn = prescaler value for 100kBits/sec
       TWI_BR400 : byte = nn;      // nn = prescaler value for 400kBits/sec
       TWI_BR500 : byte = nn;      // nn = prescaler value for 500kBits/sec
       TWI_BR600 : byte = nn;      // nn = prescaler value for 600kBits/sec
       TWI_BR800 : byte = nn;      // nn = prescaler value for 800kBits/sec
```

TWInoTimeOut ***XMega***
Normalerweise bestimmt die Variable *TWI_TimeOut**tn*** den Timeout bei den TWI Operationen. Bei sehr schnellen Slaves kann auf den TimeOut im x-SysTick Bereich verzichtet werden. Mit diesem optionalen Define (true) wird das TimeOut auf ca. 4msec heruntergesetzt werden.

Die **AVR+Mega** Treiber exportieren die Byte variable *TWI_TimeOut* im Bereich von 0..255 Systicks.

Achtung:
Bei den XMegas sind die MasterSlave und Slave Modes nicht unterstützt!
Bei den folgenden Funktionen wird das Suffix **tn** benutzt um das gewünschte TWI-port TWI_C, TWI_D, TWI_E, TWI_F zu beschreiben.
tn kann dann C, D, E oder F sein.
Beim XMega TWI wird auch die Variable *TWI_TimeOut**tn*** exportiert. Diese wird vom System auf 100msec gesetzt, kann aber vom User auf eine beliebige Anzahl von SysTicks (1..255) überschrieben werden.

3.23.3 Funktionen

3.23.3.1 TWIstat

Die Funktion TWIstat stellt fest, ob der gewählte Slave vorhanden ist, bzw. sich meldet. Im Fehlerfall kehrt die Funktion mit dem Ergebnis FALSE zurück, ansonsten ist das Ergebnis TRUE. Ein EEprom z.B. ignoriert nach dem Beschreiben für eine gewisse Zeit jeden weiteren Zugriff und würde in diesem Fall ein false liefern. Ein echter TWI-Slave liefert immer ein false zurück, falls er intern gesperrt bzw. busy ist. In diesem Fall bedeutet ein false nicht, dass der Slave nicht vorhanden ist.

Die Definition von TWIstat ist:

```
Function TWIstat(SlaveAdr : byte) : boolean;
Function TWIstatTN(SlaveAdr : byte) : boolean;   // XMega
```

Der Parameter **SlaveAdr** ist die physikalische Baustein-Adresse auf dem TWI-Bus.

3.23.3.2 TWIinp

Die Funktion TWIinp liest mindestens ein Byte aus dem angewählten Slave. Schlägt der Versuch fehl kehrt die Funktion mit dem Ergebnis FALSE zurück, ansonsten ist das Ergebnis TRUE.

Die Definition von TWIinp ist:

```
Function TWIinp(SlaveAdr : byte; Var Data) : boolean;
Function TWIinpTN(SlaveAdr : byte; var Data) : boolean;   // XMega
```

Der Parameter **SlaveAdr** ist die physikalische Baustein-Adresse auf dem TWI-Bus. Baustein-interne Adressen (z.B. EEprom) müssen vorher mit einem Write durch TWIout eingestellt werden.

Die Variable **Data** muss als Variable definiert sein, wobei der Typ der Variablen fast beliebig sein kann. Das SizeOf(Data) bzw. die Anzahl der durch diese Variable belegte Speicherstellen bestimmen auch die Anzahl der Bytes, die im Block gelesen und in die Variable Data abgelegt werden. Bei einem Byte, Char oder Boolean sind das 1 byte, bei einem Word oder Integer sind das 2 Bytes, wobei das niederwertige Byte zuerst gelesen wird und dann das High-Byte.

Ist Data ein Array-Typ, so wird die Länge des Arrays festgestellt und eine entsprechende Anzahl Bytes gelesen, beginnend mit der niederwertigsten Adresse (kleinstem Index) des Arrays.

Bei einem String-Typ als Data wird die max. mögliche Länge des Strings festgestellt und eine entsprechende Anzahl Bytes gelesen, beginnend mit dem Index string[0]. Das bedeutet, dass ein String immer mit seinem Längenbyte gelesen und geschrieben wird. Beim Lesen bestimmt das Längenbyte die Anzahl der zu lesenden Bytes. Die max. Länge des Ziel Strings wird aber nicht überschritten.

3.23.3.3 TWIinpP

Eine Funktion mit einem Pointer als Ziel steht ebenso zur Verfügung:

```
Function TWIinpP (SlaveAdr : byte; dst : pointer; len : word) : boolean;
Function TWIinpPTN(SlaveAdr : byte; dst : pointer; len : word) : boolean;   // XMega
```

Achtung:
Bei der Wahl der Data-Variable ist darauf zu achten, dass der Slave auch einen entsprechend langen Blocktransfer beherrscht. Ein TWI Baustein der nur aus einem 8-bit Parallel-Port besteht, kann in der Regel auch immer nur ein Byte übertragen.

3.23.3.4 TWIout

Die Funktion TWIout schreibt mindestens ein Byte in den angewählten Slave. Schlägt der Versuch fehl kehrt die Funktion mit dem Ergebnis FALSE zurück, ansonsten ist das Ergebnis TRUE.
Die Definition von TWIout ist:

```
Function TWIout(SlaveAdr : byte;  Command : byte|word [; Data]) : boolean;
Function TWIoutTN(SlaveAdr : byte;  Command : byte|word [; Data]) : boolean;   // XMega
```

Der Parameter **SlaveAdr** ist die physikalische Baustein-Adresse auf dem TWI-Bus.
Der Parameter **Command** ist Slave-abhängig, muss aber spezifiziert sein. Er kann eine Byte-Variable, eine 8-bit Konstante oder Word sein. Sehr oft ist dieses Kommando eine Baustein-interne Adresse (z.B. EEprom).

Bei **Slave CPUs** darf das Command immer nur **ein Byte** sein!!
Ist Command eine literale Konstante, z.B. 1, so wird angenommen, dass der Typ ein Byte ist. Sollte ein Word gewünscht sein, muss das dem Treiber mitgeteilt werden mit (addr, word(1), data);

Grössere EEproms (>256 bytes) besitzen eine 16-bit Adresse. Da es nicht sinnvoll
wäre, dafür spezielle Bibliotheksfunktionen zu implementieren, muss die Funktion **TWIout** informiert werden, ob der Adressparameter (2. Parameter) aus einem oder zwei Bytes besteht, also 8bit oder 16bit lang ist. Deshalb muss dieser Parameter eindeutig sein und damit entsprechend deklariert werden.

3.23.3.5 TWIoutP und TWIoutWP

Eine Funktion mit einem Pointer als Quelle steht ebenso zur Verfügung:

```
Function TWIoutP(SlaveAdr : byte; Command : byte; src : pointer; len : word) : boolean;
Function TWIoutPTN(SlaveAdr : byte; Command : byte; src : pointer; len : word) : boolean;  // XMega

Function TWIoutWP(SlaveAdr : byte; Command : word; src : pointer; len : word) : boolean;
Function TWIoutWPTN(SlaveAdr : byte; Command : word; src : pointer; len : word) : boolean;  // XMega
```

```
{ eeprom 8kBytes 24C65 }
Var w  : word;                  { 16 bit adr }

  w:=  0000;
  b2:= 0;
  bool:= TWIout ($50, w);       { set adr for read = 0000 }
while b2 < $FF do               { read eeprom with adr auto incr}
   bool:= TWIinp ($50, b1);     { result into b1 }
   inc(b2);
endwhile;
```

Bei Komponenten, die eine weitere interne Adresse besitzen, wie z.B. EEproms, AD-Wandler und auch Slave-CPUs, muss, bevor gelesen wird, über einen Schreibzugriff die gewünschte Lese Adresse ausgegeben werden. Der nächste Lesezugriff geht dann auf diese Adresse. Besitzt der Baustein ein auto-inkrement der Adresse (ADC, EEprom, SlaveCPU), kann jetzt auch kontinuierlich gelesen werden, ohne eine neue Adresse auszugeben.

```
{ eeprom 256*8bit PCF8582E }
Var b2 : byte;                  { 8 bit adr }

B2:=  00;
bool:= TWIout ($53, b2);        { set adr for read = 00 }
while b2 < $FF do               { read eeprom with adr auto incr}
   bool:= TWIinp ($53, b1);     { result into b1 }
   inc(b2);
endwhile;
```

Die Variable **Data** ist optional und kann eine 8 oder 16-bit Konstante oder eine fast beliebige Variable sein. Auch Rom und EEprom-Konstante (Arrays, Strings und Stringliterale) sind möglich. Dieser Parameter ist optional und kann, muss aber nicht vorhanden sein. Der Slave bzw. die gewünschte Slave-Aktion bestimmt den Typ bzw. das weglassen von Data. Das SizeOf(Data) bzw. die Anzahl der durch diese Variable bzw. Konstante belegte Speicherstellen bestimmen auch die Anzahl der Bytes, die im Block aus der Variablen gelesen und in den Slave geschrieben werden. Bei einem Byte, Char oder Boolean sind das 1 byte, bei einem Word oder Integer sind das 2 Bytes, wobei das niederwertige Byte zuerst geschrieben wird und dann das High-Byte.

Ist Result ein Array-Typ, so wird die Länge des Arrays festgestellt und eine entsprechende Anzahl Bytes geschrieben, beginnend mit der niederwertigsten Adresse (kleinstem Index) des Arrays.

Bei einem String-Typ als Data wird die Länge des Strings festgestellt SizeOf(String) und eine entsprechende Anzahl Bytes geschrieben, beginnend mit dem Index string[0]. Das bedeutet, dass ein String immer mit seinem Längenbyte gelesen und geschrieben wird. Beim Schreiben bestimmt das Längenbyte die Anzahl der zu schreibenden Bytes:

Count := length(Source).

Normalerweise wird **Data** als ein Block behandelt, egal ob es sich um ein Byte, Word, String, Array oder Record handelt. Manchmal möchte man jedoch auch das Ergebnis eines Ausdrucks übergeben, ohne dieses vorher in einem Byte, Word etc. abspeichern zu müssen. Ausdrucke sind jedoch nicht direkt möglich in Zusammenhang mit **Data**. Um dies trotzdem mit gewissen Einschränkungen zuzulassen muss dies dann dem Compiler mitgeteilt werden, indem **Data** speziell gekennzeichnet wird. Dazu ist **Data** in Klammern zu setzen.

```
bool:= TWIout ($53, b2, (a and b or c));
```

Das Ergebnis des geklammerten Ausdrucks muss ein Byte, Word, Integer etc sein. Chars, Boolean, Strings etc sind nicht zulässig.

Wurde der Parameter Data nicht angegeben, so wird nur das Command-Byte übertragen.

Achtung:
Bei der Wahl der Data-Variable/Konstante ist darauf zu achten, dass der Slave auch einen entsprechend langen Blocktransfer beherrscht. Ein I2C Baustein der nur aus einem 8-bit Parallel-Port besteht, kann in der Regel auch immer nur ein Byte übertragen.

Die Slave Adresse 00 ist die sogenannte **General Call** Adresse. Normale I2C Bausteine reagieren nicht darauf. TWI-Slaves können, wenn freigegeben, darauf reagieren. Allerdings ist ein Lesezugriff nicht sinnvoll, da alle Slave darauf reagieren.

3.23.4 Multi-Processing beim Master

In einer Applikation mit Prozessen und/oder Tasks kommt es häufig vor, dass der TWI-Bus nicht nur als Netzwerk sondern auch für andere Zwecke (LCD, Ports etc) gebraucht wird. Wenn dabei die Zugriffe auf das TWI aus unterschiedlichen Prozessen heraus erfolgen, kommt es unweigerlich zu Konflikten da solche sequentiellen Treiber (I2C, TWI, UART etc) nicht re-entrant sind, d.h. sie sind nicht unterbrechbar und neu aufrufbar. Deshalb besitzt das TWI Port ganz allgemein eine Semaphore vom Typ DeviceLock.

```
TWI_DevLock : DEVICELOCK;
TWI_DevLockTN : DEVICELOCK;
```

Der TWI Treiber beachtet und steuert diese Semaphore. Beim Eintritt in den Treiber wird geprüft ob der Treiber frei ist (Semaphore inaktiv). Ist das der Fall, dann wird die Semaphore aktiviert = Treiber gesperrt und der Auftrag wird ausgeführt. Nach dem Beenden des Auftrags wird die Semaphore wieder freigegeben.

Wird beim Eintritt festgestellt, dass der Treiber belegt ist (Semaphore aktiv) dann wird ein ***Schedule*** durchgeführt und dieser Prozess reiht sich wieder in die Warteschlange ein. Bei einem der nächsten Prozess Wechsel wird der Prozess wieder gestartet und prüft jetzt erneut die Semaphore. Das wiederholt sich jetzt solange, bis die Semaphore wieder frei ist.

Achtung:
Durch den Abbruch eines TWI-Aufrufs durch „Schedule“ sollten für den TWI Zugriff ***keine Tasks*** verwendet werden, da Tasks damit komplett abgebrochen werden und der Auftrag in diesem Fall nicht ausgeführt wird. Mann kann sich jedoch ein Flag setzen, wenn der Task mit seinem Auftrag erfolgreich war und wenn nicht, muss der Task wiederholt werden. Das bringt jedoch eine etwas komplizierte Verwaltung mit sich.

Programm Beispiel:

ein Beispiel **TWImaster** befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\TWItest**
ein Xmega Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Xmega_TWI**

3.23.5 TWI (I2C) Slave

XMega: die Slave Modes sind nicht implementiert.

Die durch **TWIslave** importierte und durch **TWIbuffer** und **TWImode** definierte TWI-Schnittstelle bietet die mehrere High-Level Funktionen: TWIgetRxStat, TWIgetTxStat, TWIsetRDY, TWIgetRDY, TWIsetBUSY, TWIgetBUSY und TWIgetCMD, abhängig von der Betriebsart. Diese Funktionen sind generalisiert, d.h. sie sind auf keinen speziellen Master Chip angepasst. Der Programmierer kann bzw. muss durch entsprechende Verwendung der Funktionen einen Handshake Betrieb sicherstellen.

Ein TWI/I2C Slave kann von sich aus niemals Daten schicken oder empfangen. Alle Transfers vom und zum Slave werden vom Master durchgeführt.

Handshake Betrieb
Der Slave sorgt durch jeweilige Status Operationen dafür, dass hereinkommende Daten nicht durch weitere überschrieben werden und dass zu sendende Daten nur einmal abgeholt werden. Funktionen: TWIgetRDY und TWIsetRDY.

Transparent Betrieb
Hier arbeiten der RxBuffer und der TxBuffer als Dualport Memory. Der Slave führt kontinuierlich die Daten bzw. Befehle im RxBuffer aus und stellt fortlaufend neue Daten in den TxBuffer. Der Master liest bzw. schreibt ebenfalls beide Buffer. Ein weiterer Informations Austausch findet nicht statt. Der Slave kann auch nicht feststellen, ob der Master neue Daten geschickt bzw. abgeholt hat. Funktionen: TWIsetBUSY und TWIgetBUSY

Support Funktionen sind TWIsetGC und TWIsetSlaveAddr.

Imports

TWIslave
Definiert und importiert die Slave TWI-Bus Schnittstelle.

Der TWI-Bus benutzt den onChip TWI Kanal. Die Schnittstelle arbeitet dabei nur als Bus-Slave.
Eine Umschaltung vom Slave in den Master Betrieb ist nicht implementiert. Dafür sollte der AVRco TWI-net Treiber benutzt werden.

Wird mittels Import der TWIslave importiert, so muss auch mittels Define die Rx/Tx Buffergrösse bestimmt werden, über die das Senden und Empfangen abgewickelt wird.
Die Default Slave Adresse ist ebenfalls vorzugeben. Diese kann jedoch zur Laufzeit geändert werden.
Der Import des TWIslave importiert automatisch auch alle zugehörige Bibliotheks Funktionen.

```
Import   SysTick, TWIslave;
```

Defines

```
Define   ProcClock  = 4000000;     {4Mhz clock }
         SysTick    = 10;          {10msec Tick}
         TWIaddr    = 2;           (TWI slave address}
         TWIbuffer  = 8, iData;    (TWI rx/tx buffersize, location}
         TWImode    = Handshake;   (TWI handshake or transparent}
```

TWIaddr
Definiert die Default BUS-Adresse des Slave. Die Adresse 0 ist nicht zulässig, ebensowenig Adressen grösser 127. Diese Adresse kann zur Laufzeit geändert werden.

TWIbuffer
Die Datentransfers zwischen Master und Slave werden über einen RxBuffer und einen TxBuffer abgewickelt, die beide gleich gross sind und durch dieses Define bestimmt werden. Die Buffergrösse kann zwischen 2

und 255 bytes eingestellt werden. Die Lage der Buffer im Speicher (iData, xData) muss ebenfalls vorgegeben werden. Es werden die Buffer **TWIrxBuffer** und **TWItxBuffer** generiert.

TWImode
Diese Define legt die prinzipielle Arbeitsweise des Slaves fest. Im Transparent Mode findet kein Handshake statt. Master und Slave haben praktisch unbegrenzten Zugriff auf die Buffer des Slaves.
Im Handshake Mode bestimmt ausschliesslich der Slave wann Daten durch den Master gelesen oder geschrieben werden dürfen.

3.23.6 Allgemeine Funktionen

TWIsetGC
Die Prozedur TWIsetGC bestimmt ob der Slave auf Broadcast Telegramme (Slave Addr = 0) empfangen soll oder nicht. Die Behandlung eines solchen Telegrams ist Sache der Applikation.
Der Master sollte nur Write Zugriffe machen.

Procedure *TWIsetGC (BroadcastEnable : boolean);*

TWIsetSlaveAddr
Die Prozedur TWIsetSlaveAddr legt eine neue Slave Adresse fest. Nur die Adressen 1..127 sind erlaubt.

Function *TWIsetSlaveAddr (SlaveAddr : byte);*

3.23.7 Funktionen im Handshake Mode

Das Define TWImode legt die grundsätzliche Arbeitsweise des Slave fest. Im Handshake Verfahren bestimmt immer der Slave, ob der Master senden bzw. empfangen kann.

Automatische Sperrung
Wenn der Master einen Lese- oder Schreibzugriff auf den Slave durchgeführt hat, sperrt der Treiber die TWI Schnittstelle komplett und der Status wird entsprechend gesetzt. Ab sofort wird **jeder** weitere Zugriff des Masters mit einem NACK beantwortet.

Die Slave Applikation muss deshalb immer den RxStatus und den TxStatus prüfen. Gibt eine der beiden Funktionen ein true zurück, dann ist die Schnittstelle gesperrt und das Slave Programm muss zumindest jetzt „TWIsetRDY(true)“ aufrufen, um die Sperrung wieder aufzuheben. Normalerweise sollte aber vorher der Receive Buffer ausgewertet werden, wenn „TWIgetRxStat“ ein true geliefert hat.

Hat der Slave in seinen Transmit Buffer Daten bereitgestellt, so sollte er warten, bis die Funktion „TWIgetTxStat“ ein true geliefert hat. Erst dann hat der Master Daten abgeholt und es können neue Daten in den TxBuffer gestellt werden. Anschliessend muss mit „TWIsetRDY(true)“ die Sperrung aufgehoben werden, so dass der Master wieder zugreifen kann.

Die Funktion „TWIgetRDY“ gibt allgemein Auskunft über den Sperr-Zustand der Schnittstelle.

Es ist Aufgabe des Programmierers eine übergeordnete Kommunikation zu etablieren, die eindeutig festlegt, ob der Slave Daten schicken oder empfangen soll. Das Command Byte bietet sich hierfür an.

Manuelle Sperrung/Entsperrung
Sperrt der Slave das TWI-Interface indem er „TWIsetRDY(false)“ aufruft, so bekommt der Master beim nächsten Zugriff ein **NACK** von seiner I2C/TWI Schnittstelle und kehrt zu seiner Applikation mit einem false zurück. Eine gesperrte TWI-Schnittstelle wird vom Slave mit „TWIsetRDY(true)“ wieder freigegeben.

Interne Adressen
Jeder neue Lese/Schreib Zugriff des Masters setzt den internen Adress-Pointer des Slave auf 00. Der Slave inkrementiert automatisch die interne Adresse mit jedem Byte Transfer bis entweder das Buffer Ende erreicht wurde oder ein neuer Zugriff des Masters erfolgt.

Das Command Byte des Masters beim “TWIout“ wird gesondert abgelegt und kann mit der Funktion „TWIgetCMD“ jederzeit gelesen werden. Ein gezieltes Schreiben auf das Command byte des Slave ohne den RxBuffer des Slave zu verändern, sieht so aus:

Master
TWIout (SlaveAdr, $AA);

Das Command des Slave enthält $AA. Zu beachten ist jedoch, dass der Slave auch bei einem Schreiben des Command Bytes sein Interface sperrt, um der Applikation anzuzeigen, dass ein Zugriff erfolgte.

TWIgetRxStat
Die Funktion TWIgetRxStat stellt fest, ob seit dem letzten Aufruf von TWIsetRDY(true) ein neues Packet/Daten hereingekommen ist oder nicht. Sind neue Daten da, wird ein true zurückgegeben, ansonsten ein false. Ist das Ergebnis ein true, ist auch die TWI-Schnittstelle komplett gesperrt.

Function *TWIgetRxStat : boolean;*

TWIgetTxStat
Die Funktion TWIgetTxStat stellt fest, ob seit dem letzten Aufruf von TWIsetRDY(true) ein Packet/Daten abgeholt worden ist oder nicht. Sind die Daten abgeholt, wird ein true zurückgegeben, ansonsten ein false. Ist das Ergebnis ein true, ist auch die TWI-Schnittstelle komplett gesperrt.

Function *TWIgetTxStat : boolean;*

TWIgetRDY
Die Funktion TWIgetRDY gibt Auskunft über den Sperrzustand der TWI-Schnittstelle. Das Ergebnis ist ein false wenn gesperrt, ansonsten ein true. Liegt keine manuelle Sperrung vor, so war ein Master Zugriff die Ursache für eine evtl. Sperrung. Welche Art dieser Zugriff war, lässt sich mit TWIgetRxStat und TWIgetTxStat feststellen.

Function *TWIgetRDY : boolean;*

TWIsetRDY
Die Funktion TWIsetRDY(true) setzt den Status des RxBuffers auf Buffer leer und den TxBuffer auf voll, so dass der Master ein neues Packet/Daten senden bzw. empfangen kann. Die Sperrung des TWI-Interfaces wird aufgehoben.

Mit TWIsetRDY(false) wird das TWI Interface komplett gesperrt, der Status bleibt jedoch unverändert.

Diese Funktion wird aber nur durchgeführt, wenn gerade kein Master Zugriff am Laufen ist. War die Funktion erfolgreich, gibt sie ein true zurück, ansonsten ein false.

Function *TWIsetRDY (ready : boolean) : boolean;*

TWIgetCMD
Die Funktion TWIgetCMD übergibt als Ergebnis das zuletzt vom Master mit **TWIout** geschickte Kommando. Der Inhalt dieses Kommandos hat im Handshake Mode keine weitere Bedeutung und ist Vereinbarungssache.

Function *TWIgetCMD : byte;*

3.23.8 Funktionen im Transparent Mode

Das Define TWImode legt die grundsätzliche Arbeitsweise des Slave fest. Ist der Transparent Mode eingestellt können der Master und der Slave unbeschränkt die Slave Buffer lesen und schreiben.

Werden Strukturen, z.B. words, integer etc. übertragen, so kann es jedoch zu Problemen kommen, wenn gerade byte-weise ein integer geschrieben wird, während der andere diesen integer liest. Deshalb gibt es hier zwei Sperrfunktionen.

TWIsetBusy
Die Funktion TWIsetBusy sperrt oder gibt TWI-Schnittstelle wieder frei. Wenn der Slave in die Buffer schreibt oder liest, und er möchte verhindern, dass in diesem Moment der Master auf einen Buffer zugreift, dann sperrt er die Buffer und der Master bekommt ein **NACK**. Das ist allerdings nur sinnvoll, wenn komplexe Typen übertragen werden sollen. Die Funktion kehrt mit einem false zurück, wenn die Sperrung nicht erfolgreich war, d.h. der Master ist zu diesem Zeitpunkt schon am übertragen.

Function *TWIsetBusy (busy : boolean) : boolean;*

TWIgetBusy
Die Funktion gibt den Status der TWI-Schnittstelle zurück, der mit TWIsetBusy vorgegeben wurde. Die Funktion kehrt mit einem true zurück, wenn die Schnittstelle gesperrt ist, ansonsten mit einem false.

Function *TWIgetBusy : boolean;*

Achtung:

Der **command** Parameter der Master-Funktion **TWIout** hat im Transparent Mode eine besondere Bedeutung. Er bezeichnet die Start Adresse des folgenden Lese oder Schreibzugriffs in den jeweiligen Buffer.

Master
TWIout (SlaveAdr, 2, $AA);
Slave
Das Datum $AA wird im Slave RxBuffer in der Speicherstelle 2 abgelegt. RxBuffer[2]

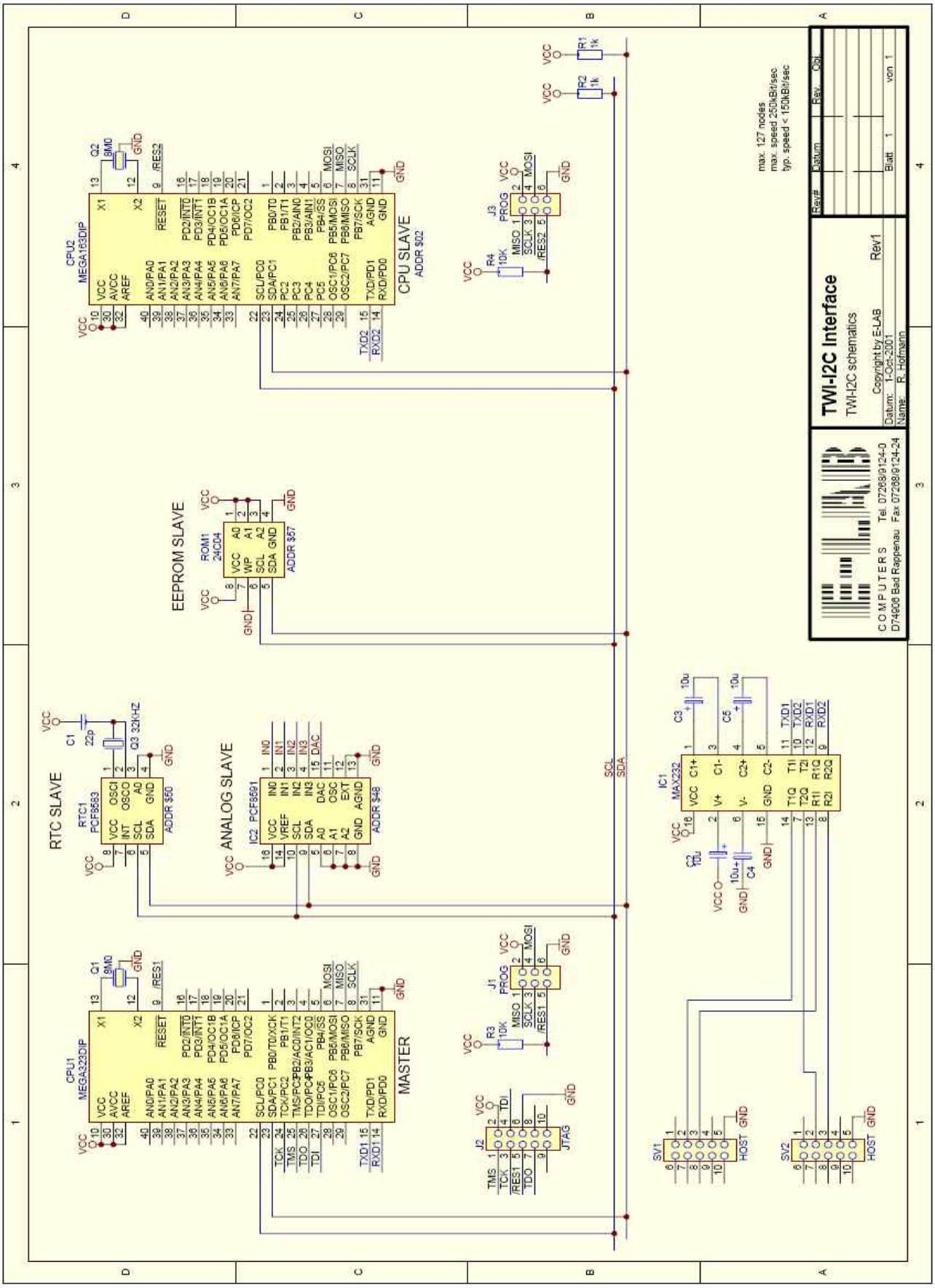
Master
TWIout (SlaveAdr, 3); // set read addr
TWIinp (SlaveAdr, bb);
Slave
Das Datum im Slave TxBuffer in der Speicherstelle 3 wird abgeholt. TxBuffer[3]

Der Slave inkrementiert automatisch das command bis entweder das Buffer Ende erreicht wurde oder ein neues command empfangen wird.

Für den TWIslave Mode ohne Handshake gibt es zwei Status bytes:
TWIslvTxCount und *TWIslvRxCount*. Bei jedem empfangen bzw. gesendeten Byte wird das zugehörige Tx/Rx Byte um 1 inkrementiert.

Programm Beispiele:

Ein Beispielprogramm **TWImaster** für eine Master Implementation,
ein Beispielprogramm **TWIslaveTrn** für eine Slave-Transparent Implementation und
ein Beispielprogramm **TWIslaveHsk** für eine Slave-Handshake Implementation befinden sich in
..\E-Lab\AVRco\Demos\TWItest



Schaltplan **TWI**

3.24 TWI-Net Library Driver

Es gibt viele Wege wie bei einem Multiprozessor System die Kommunikation zwischen den einzelnen CPUs installiert werden kann. Es bietet sich hier z.B. DualPort Memory, SPI, RS232, CAN-Bus und auch Ethernet an.

Einige der Möglichkeiten scheiden oft wegen Platz oder Preis Gründen aus. Eine sehr elegante Version ist die Vernetzung über eine I2C Schnittstelle. Alle neueren Atmel megas haben eine I2C, von Atmel zu **TWI** umgetauft.

I2C ist eine relativ sichere und schnelle Verbindung. Allerdings kommen hier nur Bausteine in Frage, die I2C/TWI als Hardware auf dem Chip implementiert haben. Bei reinen onBoard Netzen ist das TWI-Netzwerk allen anderen vorzuziehen. Bei inSystem Netzwerken zur Verbindung von Boards untereinander ist das TWI besser als die SPI Version und der RS485/LAN Version zumindest ebenbürtig. Wenn längere Distanzen zu überbrücken sind und es auf Geschwindigkeit ankommt, ist das LAN Netzwerk besser geeignet. Kommt es auf die Geschwindigkeit nicht zu sehr drauf an, kann auch hier das TWI Netzwerk verwendet werden (mehrere Meter)

Der einzige Nachteil des TWI/I2C ist die open-collector Arbeitsweise, die hier in Abhängigkeit von der Leitungslänge die max. Bitrate bestimmt.

Die vorliegende Implementation ist in erster Linie für CPU-CPU Verbindungen gedacht. Die Strategie besteht aus einem Multi-Master-Multi-Slave System. Zur sicheren Kommunikation wird ein entsprechendes Protokoll gefahren.

3.24.1 TWI Netzwerk

Es wurde ein Multi-Master / Multi-Slave System implementiert, das, wie der Name sagt, grundsätzlich aus mindestens einem Master und mindestens einem Slave besteht. Jeder Slave kann zum Master werden und jeder Master zum Slave. Mehr als ein Master gleichzeitig aktiv führt jedoch zur Kollisionsgefahr und sollte vermieden werden, oder zumindest mit grosser Sorgfalt geplant sein.

Der Master kontrolliert alle angeschlossenen Slaves. Kein Slave sollte von sich aus die BUS-Kontrolle übernehmen und Master werden, obwohl das jederzeit möglich ist. Jeglicher Transfer, ob vom oder zum Slave wird vom Master aus initiiert und durchgeführt. Damit entfallen die sonst erheblichen Probleme mit Kollisionen, Prioritäten etc. Der Wechsel eines Slaves zum Masterbetrieb wird durch die Treiber unterstützt, muss aber von der Applikation sehr sorgfältig und kontrolliert vorgenommen werden. Sinnvoll ist hierbei, dass der aktuelle Master die Master Betriebsart an einen Slave weitergibt und sich selbst zum Slave macht, oder er bleibt zumindest inaktiv. Damit werden Kollisionen vermieden, die hier kaum handhabbar sind.

Netzwerke arbeiten allgemein mit Telegrammen, auch Frames oder Packets genannt. Durch die Verwendung von Frames wird zwar ein gewisser Overhead betrieben, im Gegenzug dafür aber die Betriebssicherheit erheblich gesteigert. Die TWI Schnittstelle bietet zusätzlich ein Broadcast an. Das wird hier als zusätzlicher Info- oder Kommando Kanal benutzt.

Beim TWI/I2C Betrieb kann immer nur ein Teilnehmer senden, alle anderen hören mit oder sind deaktiviert. Der Master fordert z.B. ein Frame von einem Slave. Dazu schickt er einen Frame oder Command an den Slave mit dem entsprechenden Inhalt (Adresse, Daten, Kommandos). Der Slave erkennt seine Adresse und teilt der Applikation den Empfang eines Frames oder Commands mit. Wenn die Slave Applikation ein Antwortpacket schicken will (das ist kein muss), stellt diese in ihrem Frame-Buffer den Frame bereit. Der Master holt den Frame vom Slave ab und stellt den Frame in seinem Buffer bereit und informiert seine Application darüber.

Was der Master mit seinem Frame beim Slave bezwecken will und ob der Slave eine Antwort bereithalten muss liegt in der Hand des Programmierers. Erwartet der Master bzw. dessen Programm eine Antwort vom Slave, sollte der Slave diese umgehend bereithalten, da das Master Programm normalerweise den Slave jetzt solange pollt, bis die Antwort bereit ist. Dieses Handshaking ist jedoch Sache der Applikation und nicht des Treibers.

Der Slave kann auch von sich aus einen Frame bereitstellen. Der Master erkennt dies durch Pollen der Slaves und teilt dies der Anwendung mit. Der Master kann auch jederzeit von sich aus ein Frame an einen Slave schicken, vorausgesetzt der Empfangsbuffer des Slave ist leer.

Die Slaves sind komplett im Interrupt betrieben. Der Master arbeitet im polling mode.

Grundsätzlich gilt:

Der **Master** sendet und empfängt alle Frames auf dem BUS und stellt sie dem Anwendungsprogramm zur Verfügung. Eine Auswertung irgendwelcher Art im Master erfolgt nicht.

Ein **Slave** hört zwar immer auf dem BUS mit, fängt aber nur Frames ab bzw. liest sie ein, wenn die gesendete Hardware Adresse identisch ist mit seiner eigenen (Node Adr). Eine Auswertung des Frames innerhalb des Slave Devices erfolgt nicht, ausgenommen der Checksum Prüfung. Es bleibt seinem Anwendungsprogramm überlassen, den Frame auszuwerten.

Der Slave Treiber selbst stellt auch kein Acknowledge oder eine Antwort bereit, ausgenommen der TWI/I2C internen Prüfungen bzw. Handshakes. Es ist Sache der Anwendung auf eine Frame zu reagieren.
Die Slave Adressen müssen > 0 und <128 sein. Die Adresse “0” gilt als Broadcast Adresse für alle Slaves. Alle aktiven Slaves empfangen das Broadcast Command (2 bytes, + optionaler Frame), reagieren aber nicht direkt darauf.
Die aktuelle Datenlänge (Bytes) eines Frame kann von 0 Bytes bis TWIframe –1 laufen. Die Adresse ist 7 Bits lang was 126 Nodes zulässt. Die Checksumme ist grundsätzlich eine 8 Bit Checksumme. Variabel ist dabei nur die aktuelle Datenlänge, die Adresslänge ist immer ein Byte und ebenso die Checksumme.

Aufbau eines Telegramms:

| ADDRESS | FRAMELENGTH | DATA, DATA ... | CHECKSUM |

Adresse:
Die Adresse ist ein Byte (1..127 Slaves).

Jeder Slave muss eine eigene Adresse haben, die im ganzen Netzwerk sonst nicht mehr vorkommt. Der Master hat immer eine eigene Adresse, da er auch zum Slave umgeschaltet werden kann. Wenn der Master ein Packet schickt, fügt er seine eigene Slave Adresse auch in das Telegramm selbst ein. Wenn ein Slave ein Frame zum Abholen bereit stellt, fügt er seine eigene Adresse ins Telegramm ein. D.h. die Adresse ist immer die Adresse des Slave bzw. Absenders des Packets!

FrameLength:
Die Frame Länge wird mit define TWIframe = nn eingestellt. Ist der Parameter nn < 256, wird ein Längenbyte übertragen, ist nn >= 256, werden 2 Längenbytes übertragen. Die aktuelle Framelänge kann zwischen 0 und nn bytes sein. Allerdings müssen alle BUS Teilnehmer die gleiche Längen Art (Byte/Word) besitzen, so dass die Frames bei allen identisch sind. In den Bereichen eines Bytes oder Words können die implementierten Buffergrössen allerdings variieren.

DATA:
Der Datenbereich innerhalb eines Frames wird durch das Längenbyte bzw. Wort angegeben. Es brauchen keine Daten im Frame sein, dann ist das Längenbyte/Wort auch null.

CHECKSUM:
Die Checksumme wird über den kompletten Frame gebildet. Es wird eine 8bit Checksumme gebildet.

Achtung:
Die max. FrameLänge muss zur Design Zeit mittels den Defines festgelegt werden. Ebenso der Modus Master, Slave oder MasterSlave. Eine Änderung zur Laufzeit ist nur für den MasterSlave Mode möglich. Die implementierten Frame Buffer Längen können in den einzelnen Teilnehmern variieren, aber nur soweit, wie die Framelänge dies zulässt. Entweder haben alle Teilnehmer eine Buffer Grösse kleiner 256 Bytes (FrameLength in einem Byte) oder grösser 255 Bytes (FrameLength in einem Wort). Eine Mischung ist nicht zulässig.

Sicherheit
Die Applikationen im Master und Slave können mit einem Acknowledge arbeiten, d.h. wenn der Master einen Frame losgeschickt hat, wartet das übergeordnete Programm im Master, bis der Slave einen Antwortframe bereitgestellt hat. Das gibt zwar sehr grosse Sicherheit, aber der Durchsatz wird u.U. erheblich reduziert. Es ist also Aufgabe der einzelnen Programme bzw. des Programmierers ein entsprechendes Verfahren anzuwenden.

Aufbau eines Broadcast Commands:

| SUBADDR | COMMAND |
oder optional:
| SUBADDR | COMMAND | FRAMELENGTH | DATA, DATA ...

SubAddress:
Die Adresse ist ein Byte und wird beim Slave zur Validierung des Commands herangezogen.

Command:
Das Command ist ein Byte und wird beim Slave nur unter bestimmten Bedingungen (SubAdr, Mask etc) ausgewertet.

FrameLength:
Der Parameter FrameLength ist hier immer ein Byte. Diese Option ist nur möglich, wenn beim Import/Definition des TWI Netzwerks ein Broadcast Frame mit **Define TWIframeBC = nn** definiert wurde. Dadurch kann auch noch ein Datenframe mit bis zu 255 bytes angefügt werden. FrameLength kann hierbei den Wert 0..255 annehmen.

Die wesentliche Funktion von Broadcasts ist es, bestimmten oder allen Slaves Kommandos zu erteilen, die unabhängig von seinem aktuellen TxFrame oder RxFrame Status empfangen und bearbeitet werden. Damit können z.B. Blockaden, hervorgerufen durch unvollständige und zerstörte Frames, aufgehoben werden.

Das ist wichtig, wenn das System mit Prozessen arbeitet und diese auf Semaphoren warten.

Transfer:
Der Datentransfer erfolgt beim Slave grundsätzlich im Interrupt Verfahren. Damit wird sichergestellt, dass wenn ein Frame fertig zusammengestellt ist, dieser auch in kürzester Zeit sein Ziel erreicht. Wichtig ist, dass im Slave die Interrupts nicht zu lange gesperrt bleiben. Das gilt vor allen Dingen bei hohen Bitraten. Bei 100kBit/sec wird bei einem Byte Transfer jede 100usec ein TWI Interrupt ausgelöst.

Die eingestellte Bitrate muss der langsamsten CPU, der Leitungslänge und den PullUp Widerständen angepasst sein. Realistisch ist ein Wert von 100..150kBit/sec.

Leitungen:
Der I2C/TWI Bus ist ein Open Collector Bus, d.h. alle Treiber schalten gegen 0Volt. Eine logische 1 wird ausschliesslich durch PullUps erzeugt. Der Grund dafür ist, dass die beiden Leitungen SCL und SDA immer bidirektional arbeiten. Der Slave kann jederzeit durch pulldown ein Clockstretching einfügen. Ausserdem werden die einzelnen Phasen immer durch einen low Ack Impuls des Slaves bestätigt. Das ganze bringt eine sehr einfache Bus Verdrahtung und trotzdem viel Komfort und Sicherheit.

Der Nachteil ist die eingeschränkte Geschwindigkeit durch den Open Collector bzw. Pullup Betrieb.
Je niederohmiger der Pullup ist, desto grösser kann die Geschwindigkeit und/oder Leitungslänge sein. Umso mehr steigt allerdings auch der Stromverbrauch an. Die Pullups dürfen 500Ohm nicht unterschreiten, da sonst die Portpins Treiber der CPUs beschädigt werden (Überstrom).

Statt Pullups können auch Konstantstrom Quellen benutzt werden, die bei gleichem Stromverbrauch eine wesentlich steilere Flanke erzeugen und damit eine höhere Bitrate zulassen.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\TWInet**

Moderne I2C Chips können bis zu 400kBit/sec erreichen. Dann wird aber u.U. die Leitungslänge kritisch. Das System exportiert im Master Mode auch noch die Konstanten

```
Const   TWI_BR100 : byte = nn;   // nn = prescaler value for 100kBits/sec
        TWI_BR400 : byte = nn;   // nn = prescaler value for 400kBits/sec
```

Diese Konstanten können auch schon im Define verwendet werden:

```
Define  ProcClock = 16000000;      {16Mhz clock }
        TWIpresc  = TWI_BR100;     {TWI speed 100kBit/sec}
```

3.24.2 Implementation

Imports

Das TWInet muss, wie beim AVRco System üblich, durch eine Import Anweisung importiert werden

```
Import  TWInet;
```

Defines

```
Define  TWInode  = $12;          {Node Addr, always necessary}
```

Festlegung der Node Adresse, auch für den Master zwingend. Kann zur Laufzeit geändert werden.

```
Define  TWInetMode = Master;     {Master, Slave or MasterSlave}
```

Die Betriebsart der CPU, Master bzw. Slave muss mittels Define festgelegt werden. Ein Wechsel der gewählten Betriebsart während des Betriebs, also zur Laufzeit, ist nur möglich wenn MasterSlave definiert wurde.

```
Define  TWIpresc = 32;           {Prescaler, bitrate for Master mode}
```

Bestimmt die Bitrate des Masters. Sollte nie kleiner 8 sein.

Der Frame Typ des TWI-net muss bei Master und Slave **identisch** sein und wird durch folgendes Define bestimmt:

```
Define  TWIframe = 16, iData;    {Framesize max. 16 bytes in iData}
```

Master und Slave haben beide jeweils einen Receive- und einen Transmit-Buffer. Alle Master/Slaves sollten die gleiche Buffergrösse haben, müssen aber nicht. Die angegebene Buffergrösse bestimmt den internen Speicherbedarf. Die transferierten Frames können kleiner aber nie grösser sein als hier angegeben.

Da die aktuelle Frame Länge als Parameter mit übertragen wird, müssen alle Frames entweder mit einem Byte als Längen Information auskommen oder mit einem Word. Ein mix ist nicht möglich. Diese Einschränkung vereinfacht das Protokoll und sein Handling wesentlich und ist in der Praxis eigentlich keine. Alle Frame Size Definitionen müssen daher entweder kleiner oder grösser 256 Bytes sein.

Die Länge dieser Buffer und der Speicherbereich wird durch obiges Define bestimmt.

Für Broadcast Telegramme kann neben den beiden Parametern ***SubAddr*** und ***Cmd*** auch optional ein Datenframe mitgeschickt werden. Diese Option ist mit diesem Define zu importieren:

```
Define   TWIframeBC = 64;          {Broadcast Framesize max. 16 bytes}
```

Auch hier gilt, entweder alle oder keine Bus Teilnehmer (Master und Slaves) müssen diesen Import haben, und wenn vorhanden, dann muss die Framegrösse überall identisch sein.

3.24.3 Exportierte Typen

```
Type tTWInetmode  = (TWInetSlave, TWInetMaster);

Type tTWIStates   = (TWIRxEmpty, TWIRxBusy, TWIRxFull, TWITxEmpty,  TWITxBusy, TWITxFull,
                     TWIbcCMD, TWIstatFail);

Type tTWINetState = set of tTWIStates;
```

3.24.4 Exportierte Variablen

3.24.4.1 Memory Organisation

Alle dem TWItreiber zugehörige Variablen, die für die Anwendung sichtbar sind, liegen aufgereiht hintereinander im Speicher. Die Reihenfolge ist, beginnend mit der niederwertigen Adresse, folgende:

TWIrxStatReg	Byte
TWIrxADR	Byte
TWIrxLEN	Byte bei einer Framelänge < 256 bytes, word bei Framelänge > 255 bytes
TWIrxBUFF	Array[0..TWIframe-1] of Byte;
_TWIrxCHK	Byte

TWItxStatReg	Byte
TWItxADR	Byte
TWItxLEN	Byte bei einer Framelänge < 256 bytes, word bei Framelänge > 255 bytes
TWItxBUFF	Array[0..TWIframe-1] of Byte;
_TWItxCHK	Byte

TWIrxStatReg, TWItxStatReg
Diese beiden Variablen (Byte) dürfen nur gelesen werden! Beide Status Register werden beim Empfang bzw. Senden eines Frames automatisch upgedated. Im Ruhe Zustand sind alle bits 0. Bei fortschreitender Aktion werden die Bits von rechts nach “1”. Nach erfolgreichem Abschluss der Aktion sind alle Bits “1’ und das byte hat den Wert $FF. Zwischenwerte bedeuten, dass ein Fehler aufgetreten ist oder zumindest der Frame noch nicht komplett empfangen bzw. gesendet wurde. Ein $9F bedeutet z.B. Checksumme Fehler. Die Applikation sollte immer das 8.Bit prüfen, um festzustellen, ob ein Frame komplett ist. Ein korrekt und komplett empfangener Frame erzeugt hier ein „FF“. Beim Master und Slave wird nach einem erfolgreichen Sendevorgang das Byte auf „00“ gesetzt.

```
(* bit0,1    :   00 = idle            *)
(*               01 = start           *)
(*               11 = node addr       *)
(* bit2,3    :   00 = idle            *)
(*               01 = first len       *)
(*               11 = second len      *)
(* bit4      :    0 = rx/tx frame     *)
(*                1 = rx/tx frame     *)
(* bit5,6    :   00 = idle            *)
(*               01 = start check     *)
(*               11 = check ready     *)
(* bit7      :    0 = processing      *)
(*                1 = rx/tx complete  *)
```

Verlauf der Status Register während des Empfangens und Versenden eines Frames, vorausgesetzt es traten keine Fehler auf

Master sendet:	TWItxStatReg
00h	Grundzustand.
01h	Frame ist komplett assembliert
02..FFh	Sendevorgang läuft
00h	Übertragung ist abgeschlossen

Master empfängt:	TWIrxStatReg
00h	Grundzustand.
01..7Fh	Empfangsvorgang läuft
FFh	Übertragung ist abgeschlossen

Wenn die Applikation den empfangenen Frame ausgewertet hat, muss sie durch „TWIRXCLEAR“ den empfangenen Frame ungültig machen. Das Status Register enthält dann 00h

Slave sendet:	TWItxStatReg
00h	Grundzustand.
01h	Frame ist komplett assembliert
02..FFh	Sendevorgang läuft, Master holt Frame ab
00h	Übertragung ist abgeschlossen

Slave empfängt:	TWIrxStatReg
00h	Grundzustand.
01..7Fh	Empfangsvorgang läuft
FFh	Übertragung ist abgeschlossen

Wenn die Applikation den empfangenen Frame ausgewertet hat, muss sie durch „TWIRXCLEAR“ den empfangenen Frame ungültig machen. Das Status Register enthält dann 00h

TWIrxAdr, TWItxAdr
Diese Variablen sind ein Byte Typ und enthalten immer die Node Adresse des Senders des Packets

Master:
Der Master initiiert, sendet und pollt alle Frames. Da beim I2C/TWI Bus der Master sowohl Frames zum Slave sendet, als auch Antwort Frames vom Slave abholt, weiss der Master bzw. die Applikation jederzeit, wohin ein Frame gegangen ist bzw. woher ein Frame abgeholt wurde.

Verschickt der Master einen Frame, trägt der Treiber seine eigene Node Adresse in die Variable “TWItxAdr” ein.

Die Adresse „0“ ist Broadcast Commands reserviert und darf nicht durch einen Master oder Slave als eigene Node Adresse verwendet werden.

Slave:
Ein Slave empfängt nur diejenigen Frames, deren Hardware Adresse mit seiner Node-Adresse übereinstimmt.
Die Node Adresse des Absenders (Master) wird in “TWIrxAdr” abgelegt. Broadcasts mit der ZielAdresse „0“ werden separat behandelt. Siehe weiter unten.

In “TWItxAdr” wird immer die Node Adresse des Slave abgelegt. Dies geschieht automatisch beim Senden.

TWIrxLen, TWItxLen
Diese Variablen sind ein Byte wenn „Define TWIframe“ < 256 bytes, und ein Word bei „Define TWIframe“ > 255 bytes. Die verschickten Frames können die Länge „0“ haben. Die maximale Länge kann nicht grösser sein als in „Define TWIframe“ angegeben. Grundsätzlich sollten alle Slaves und der Master gleiche „TWIframe“ Grössen haben. Es ist jedoch denkbar, dass bestimmte Slaves eine kürzere „TWIframe“ haben. Der Master muss aber dem Rechnung tragen und an diese Nodes nur Frames schicken, die auch in deren Buffer passen.

TWIrxLen wird beim empfangen eines Frames automatisch mit dem empfangenen Wert belegt.
TWItxLen wird durch die Funktion „TWItxFrame(Node, len)“ gesetzt und mit dem Frame übertragen.

TWIrxBuff, TWItxBuff
Die Länge dieser beiden Buffer werden durch das Define TWIframe bestimmt. Die Buffer können als Array of byte gesehen und damit auch gelesen und geschrieben werden. Die Indize laufen von 0 bis TWIframe -1.

```
TWItxBuff[0]:= $56;
X:= TWIrxBuff[6];
```

Es ist empfehlenswert und sehr nützlich, eine Struktur (record) über den RxTeil und den Tx Teil des TWI Speichers zu legen. Damit ist ein symbolischer Zugriff auf die kompletten Daten möglich.

```
type   tTWIRec = record
                   TWIstate : byte;          // TWI state, size and loc fixed
                   TWInode : byte;           // node, size and loc fixed
                   TWIlen   : byte/word;     // framelen, size and loc fixed
                   TWIusr1  : byte;          // user defined
                   TWIusr2  : word;          // user defined
                   TWIdata  : array[0..TWIframe-4] of char;     // user
                 end;

var
{$NOOVRCHECK}
  TWIRxRec[@TWIrxStatReg]  : tTWIRec;
{$NOOVRCHECK}
  TWITxRec[@TWItxStatReg]  : tTWIRec;

// Frame belegen
  TWITxRec.TWIusr1:= $30;
  TWITxRec.TWIusr2:= $3231;
  TWITxRec.TWIdata[0]:= '3';
```

_TWIrxChk, _TWItxChk
Der Typ ist ein Byte. Die Checksumme wird über den ganzen Frame gerechnet (TWIRXADR + TWIRXLEN + TWIRXBUFF). Das gleiche gilt auch für den TxFrame. Die Checks werden automatisch beim Senden und Empfangen durchgeführt. Die beide Variablen sind nur zur Information bzw. zur Hilfe bei evtl. auftretenden Fehlern.

3.24.4.2 Variable nur im MasterSlave und Slave Mode

Diese Variablen sind nur im MasterSlave Mode und im Slave Mode vorhanden:

```
Define   TWInetMode  = MasterSlave;
```
oder
```
Define   TWInetMode  = Slave;
```

Ausserdem haben sie nur eine Funktion, wenn der Slave Mode auch aktiv ist.

Mit den Broadcast Variablen wird das Verhalten des Slaves bei Broadcast Commands eines Masters gesteuert.
I2C/TWI Broadcasts werden durch die Hardware Adresse 00 initiiert. Alle Slaves empfangen alle Broadcasts. Um hier weitere Unterscheidungen treffen zu können, ist hier eine Sub-Adresse und Maske implementiert. Diese benutzt der Slave Treiber um zu erkennen, ob dieser Broadcast Command auch ihn betrifft. Der Master, der den Broadcast sendet, schickt als erstes Datenbyte das Kommando und dann die Sub-Adresse. Optional folgt dann noch ein Datenframe. Alle Slaves führen dann intern folgende Operation aus:

```
if (SubAddr or TWIadrMask) = (NodeAdr or TWIadrMask) then
  TWIbroadcastAdr:= SubAdr;
  TWIbroadcastCMD:= CMD;
  //Receive optional Broadcast frame;
  Inc (TWIBROADCASTSEMA);
endif;
```

TWIadrMASK
Der Typ ist ein Byte. Der Slave benutzt diese Maske um bei einem Broadcast zu erkennen, ob die Sub-Adresse auch für ihn gilt.

TWIbroadcastADR
Der Typ ist ein Byte. Diese Variable enthält die Sub-Adr des zuletzt gesendeten Broadcast Commands.

TWIbroadcastCMD
Der Typ ist ein Byte. Diese Variable enthält das Kommando des zuletzt gesendeten Broadcast Commands.

Broadcast Option
Wurde durch das Define „TWIframeBC“ ein Broadcast Frame importiert gibt es beim Master und bei den Slaves die zusätzlichen Informationen bzw. Parameter

TWIBroadCastCount
Der Typ ist ein Byte. Diese Variable enthält die im Datenframe verschickten Bytes.

TWIBCStatReg
Der Typ ist ein Byte. Diese Variable enthält die tatsächlich im Datenframe empfangenen Bytes.

TWIBroadCastBuffer
Der Typ ist ein Array[0.. TWIframeBC-1] of Byte. Diese Variable enthält die als Datenframe empfangenen Bytes.

Ein Check des BroadCast Frames kann durch Vergleich der beiden Variablen TWIBCStatReg und TWIBroadCastCount stattfinden. Wenn eine Differenz auftritt ist der Frame unvollständig oder defekt.

3.24.5 Multi-Processing beim Master

In einer Applikation mit Prozessen und/oder Tasks kommt es häufig vor, dass der TWI-Bus nicht nur als Netzwerk sondern auch für andere Zwecke (LCD, Ports etc) gebraucht wird. Wenn dabei die Zugriffe auf das TWI aus unterschiedlichen Prozessen heraus erfolgen, kommt es unweigerlich zu Konflikten da solche sequenziellen Treiber (I2C, TWI, UART etc) nicht re-entrant sind, d.h. sie sind nicht unterbrechbar und neu aufrufbar. Deshalb besitzt das TWI Port ganz allgemein eine Semaphore vom Typ DeviceLock.

```
TWI_DevLock  : DEVICELOCK;
```

Der TWI Treiber beachtet und steuert diese Semaphore. Beim Eintritt in den Treiber wird geprüft ob der Treiber frei ist (Semaphore inaktiv). Ist das der Fall, dann wird die Semaphore aktiviert = Treiber gesperrt und der Auftrag wird ausgeführt. Nach dem Beenden des Auftrags wird die Semaphore wieder freigegeben.

Wird beim Eintritt festgestellt, dass der Treiber belegt ist (Semaphore aktiv) dann wird ein **Schedule** durchgeführt und dieser Prozess reiht sich wieder in die Warteschlange ein. Bei einem der nächsten Prozess Wechsel wird der Prozess wieder gestartet und prüft jetzt erneut die Semaphore. Das wiederholt sich jetzt solange, bis die Semaphore wieder frei ist.

Achtung:
Durch den Abbruch eines TWI-Aufrufs durch „Schedule“ sollten für den TWI Zugriff **keine Tasks** verwendet werden, da Tasks damit komplett abgebrochen werden und der Auftrag in diesem Fall nicht ausgeführt wird. Mann kann sich jedoch ein Flag setzen, wenn der Task mit seinem Auftrag erfolgreich war und wenn nicht, muss der Task wiederholt werden. Das bringt jedoch eine etwas komplizierte Verwaltung mit sich.

3.24.6 Multi-Processing beim Slave

TWIbroadcastSEMA
Der Typ ist eine Semaphore. Bei jedem akzeptierten Broadcast Command eines Masters wird diese Semaphore um eins inkrementiert. Der Inhalt der Variablen „TWIbroadcastAdr“ und „TWIbroadcastCMD“ wird dabei mit neuen Werten überschrieben. Das gleiche gilt auch für den optionalen „TWIBroadCastBuffer“.

TWIrxSEMA
Der Typ ist eine Semaphore. Bei jedem komplett empfangenen Frame von einem Masters wird diese Semaphore im Slave um eins inkrementiert. Auch ein Checksummen Fehler führt zu einem Inkrement, da der Frame ja komplett ist.

TWItxSEMA
Der Typ ist eine Semaphore. Bei jedem komplett gesendeten Frame an einen Masters wird diese Semaphore im Slave um eins inkrementiert.

Wenn Prozesse in Betrieb sind, ist es u.U. sinnvoll, einen Prozess auf einen Frame warten zu lassen. Das kann mit “WaitSema(TWIrxSema)” erreicht werden. Der Prozess wartet bis ein Frame komplett empfangen wurde. Da auch ein Checksummen Fehler diese Semaphore inkrementiert, muss der Prozess mit „TWIRXSTAT“ oder mit „GetTWIslaveSTAT(0)“ den tatsächlichen Status des empfangenen Frames abfragen. Das „TWIrxStatReg“ enthält dann 00h, oder im Fehlerfall einen Fehlercode.

Ein Prozess kann auch auf ein Broadcast Command warten mit “WaitSema(TWIbroadcastSema)”.

Ebenso ist es für den Slave möglich, einen Prozess warten zu lassen, bis der Master einen zum senden bereitgestellten Frame abgeholt hat. Dies geschieht mit “WaitSema(TWIrxSema)”. Der Prozess wird aufgeweckt, wenn der Frame komplett gesendet wurde. Das „TWItxStatReg“ enthält dann 00h.

Wenn Frames durch Störungen nur zum Teil übertragen wurden, kann es allerdings dazu kommen, dass eine bestimmte Semaphore nie inkrementiert wird und damit der zugehörige Prozess nie geweckt wird.

3.24.7 Exportierte Funktionen und Prozeduren

Procedure *SetTWInodeAddr (sAddr : byte);*
Jeder TWI Master und Slave muss eine eigene und im Netzwerk nur einmal vorkommende Node Adresse haben. Diese wird normalerweise durch das **Define TWInode = nn;** zugewiesen. Es ist jedoch möglich mit dieser Prozedur die Node Adresse zu ändern.

Diese Adresse wird als Hardware Adresse beim Lesen oder Schreiben eines TWI Slaves verwendet. Die Frame internen Adressen TWIrxAdr und TWItxAdr werden beim Versenden von Frames vom Sender mit dessen Hardware Adresse beschrieben. Damit kann der Empfänger immer feststellen wer der Absender eines Frames war.

Function *GetTWIslaveSTAT (node : byte) : tTWINetState;*
Diese Funktion dient zum feststellen des Status der lokalen Rx und Tx Buffer bzw. Frames und bei dem Master auch für den Status von Frames eines Slaves. Ist das System im Slave Mode bzw. als reiner Slave deklariert, wird der Parameter „node“ ignoriert und immer nur der lokale Status zurückgegeben.

Ist das System im Master Mode bzw. als reiner Master deklariert, so gibt die Funktion mit dem Parameter „node“ = 0 den lokalen Status zurück und mit „node“ <> 0 den Status des mit der Hardware Adresse „node“ arbeitenden Slave zurück.

Das Resultat der Funktion ist ein Bitset, das so konstruiert ist:

```
tTWIStates     = (TWIRxEmpty, TWIRxBusy, TWIRxFull, TWITxEmpty, TWITxBusy, TWITxFull,
                  TWIbcCMD, TWIstatFail);
tTWINetState   = set of tTWIStates;
```

Ist das System ein Master und kommt das Resultat von einem Slave, so haben die einzelnen Werte folgende Bedeutung:

TWIRxEmpty	der RxBuffer des Slave ist leer und empfangsbereit.
TWIRxBusy	der RxBuffer ist belegt und fehlerhaft
TWIRxFull	der RxBuffer ist voll und noch nicht abgearbeitet
TWITxEmpty	der TxBuffer des Slave ist leer
TWITxBusy	der Slave assembliert gerade ein Frame, oder der letzte Frame war fehlerhaft
TWITxFull	der Slave hat einen Frame bereitgestellt zum abholen
TWIbcCMD	der Slave hat das letzte Broadcast Command noch nicht ungültig gemacht
TWIstatFail	der Slave hat auf die Status Anfrage nicht reagiert

Die Auswertung der Funktion kann so aussehen:

```
TWIstate:= GetTWIslaveStat(02);
If TWIstatFail in TWIstate then
  ..
else if [TWIrxEmpty, TWItxFull] in TWIstate then
  // either send a frame or receive a frame
else ..
```

Wenn ein TWI knoten sich selbst checkt, kann natürlich der Wert „TWIstatFail“ nicht vorkommen.

***Function** TWIrxStat : boolean;*
Diese Funktion gibt ein true zurück, wenn ein Frame empfangen wurde und noch nicht mit „TWIrxClear“ ungültig gemacht wurde. Ob der Frame fehlerfrei ist, muss mit **TWIrxStatReg** festgestellt werden. Solange ein Frame gültig ist, wird kein weiterer Frame akzeptiert, d.h. dieser wird ignoriert. Es ist Aufgabe des Programms einen empfangenen Frame so schnell wie möglich zu lesen und mit TWIrxClear ungültig zu machen, so dass ein neuer empfangen werden kann. Es gehen zwar keine Frames verloren, aber das System kann blockiert werden.

***Procedure** TWIrxClear;*
Das Statusbyte des RxBuffers wird zurückgesetzt (00h). Beim Slave kann jetzt ein neuer Frame empfangen werden. Wenn MasterSlave oder Slave definiert ist, wird auch die Semaphore „TWIrxSema“ zurückgesetzt.

***Function** TWItxStat : boolean;*
Diese Funktion prüft, ob ein zu sendender Frame gesendet ist oder nicht. Ist der Frame gesendet, kehrt die Funktion mit true zurück, ansonsten mit false. Ein neuer Frame kann nur gesendet werden, wenn der aktuelle komplett übertragen ist **oder** der Status mit „TWItxClear“ zurückgesetzt wurde.

***Procedure** TWItxClear;*
Das Statusbyte des TxBuffers wird zurückgesetzt. Ein neuer Frame kann gesendet werden. Wenn MasterSlave oder Slave definiert ist, wird auch die Semaphore „TWItxSema“ zurückgesetzt.

***Function** TWItxFrame (node : byte; len : byte[word]) : boolean;*
Die Funktion prüft den Parameter „node“ auf Plausibilität (0 < node < 128). Ebenso wird geprüft, ob der letzte Frame erfolgreich gesendet wurde (TWItxStatReg = 00h). Wenn nicht, kehrt die Funktion mit false zurück. Mit TWItxClear kann der Frame zurückgesetzt werden, ohne die eigentlichen Daten zu löschen, und TWItxFrame kann noch einmal aufgerufen werden. Wenn der Frame gesendet werden kann, wird der „len“ Parameter in „TWItxLen“ abgelegt. Der Parameter „node“ bezeichnet die Hardware Adresse des Empfängers. Die Node Adresse des Senders wird in „TWItxAdr“ abgelegt.

1. Beim Master beginnt jetzt das Senden. Wenn der Slave reagiert, wird der Frame komplett übertragen. Im Erfolgsfall enthält das „TWItxStatReg“ jetzt 00h. Reagiert der Slave nicht oder kommt es zu einem Übertragungsfehler ist „TWItxStatReg“ jetzt ein Wert > 00h und < FFh enthalten. Wenn z.B. der Slave nicht reagiert, steht hier 80h. Die Master Applikation kann auch vor dem Senden mit der Funktion „GetTWIslaveStat(slavenode)“ prüfen, ob der Slave aktiv und empfangsbereit ist bevor er die Sende Funktion aufruft.

2. Wenn der Slave TWItxFrame aufruft, initialisiert diese Funktion den Frame und setzt „TWItxStatReg“ auf 01h und kehrt mit einem true zurück. Der Frame ist aber noch nicht verschickt. Erst wenn ein Master diesen Frame mit „TWIrxFrame“ liest, wird das Statusregister auf 00h gesetzt und die Semaphore „TWItxSema“ inkrementiert. Der Parameter „node“ ist beim Slave hier ohne Bedeutung, wird jedoch auch auf Gültigkeit geprüft.

3.24.8 Funktionen und Prozeduren nur im MasterSlave Mode

***Procedure** SetTWImode (**const** twimode : tTWInetmode);*
Diese Prozedur schaltet den Arbeitsmode im Betrieb zwischen Master und Slave um. Der Master sollte sich erst in Slave Mode schalten, wenn sicher ist, dass ein Slave anschliessend zum Master wird. Ein Slave sollte sich erst zum Master schalten, wenn er vom aktuellen Master die Erlaubnis dazu bekommt. Bei zwei Mastern auf dem BUS kommt es unweigerlich zu Kollisionen, die sich kaum handhaben lassen.

tTWInetmode = (TWInetSlave, TWInetMaster);

Der aktuelle Modus kann durch Lesen der System Variablen _**TWInetMode** festgestellt werden.

Achtung:
Im MasterSlave Mode startet ein TWI System immer im Slave Mode. Soll dieses System Master werden, muss zuerst diese Procedure aufgerufen werden.

Diese Prozedur setzt alle Frames, Buffers, Status Register und Semaphoren zurück.

3.24.9 Funktionen und Prozeduren nur im MasterSlave und Master Mode

Diese Funktionen dürfen nur verwendet werden, wenn das System im Master Mode ist bzw. als reiner Master definiert ist.

***Function** TWItxBroadcast (cmd : byte; subnode : byte) : boolean;*
Diese Funktion führt einen I2C Broadcast an alle Slaves durch. Jeder I2C Port eines aktiven Slaves empfängt grundsätzlich alle Broadcasts per Definition. I2C Broadcasts sind reine Schreibzugriffe. Der Broadcast besteht aus der Hardware Adresse 00h einem angefügten Command Byte und der Sub-Node Adresse. Das Telegramm besteht daher aus 2 Bytes.

Falls „TWIframeBC“ definiert/importiert wurde, wird daran noch ein variabler Frame angehängt. Dazu ist dann diese Funktion zu verwenden:

***Function** TWItxBroadcast (cmd, subnode, count : byte): boolean;*

Der Master füllt dazu das Array TWIBroadCastBuffer mit Daten und übergibt in obiger Funktion im Parameter „count“ die Anzahl der zu schickenden Bytes. Der Slave liest diese Bytes aus seinem TWIBroadCastBuffer.

Mit den Broadcast Variablen wird das Verhalten des Slaves bei Broadcast Commands eines Masters gesteuert.
I2C/TWI Broadcasts werden durch die Hardware Adresse 00 initiiert. Alle Slaves empfangen alle Broadcasts. Um hier weitere Unterscheidungen treffen zu können, ist hier eine Sub-Adresse und Maske implementiert. Diese benutzt der Slave Treiber um zu erkennen, ob dieser Broadcast Command auch ihn betrifft. Der Master, der den Broadcast sendet, schickt als erstes Datenbyte diese Sub-Adresse und dann das Kommando. Alle Slaves führen dann intern folgende Operation aus:

```
If (SubAddr or TWIadrMask) = (NodeAdr or TWIadrMask) then
  TWIbroadcastAdr:= SubNode;
  TWIbroadcastCMD:= CMD;
  //Optional receive data frame;
  Inc (TWIbroadcastSema);
Endif;
```

Bei jedem akzeptierten Broadcast Command eines Masters wird die Semaphore „TWIbroadcastSema“ um eins inkrementiert. Der Inhalt der Variablen „TWIbroadcastAdr“ und „TWIbroadcastCMD“ wird dabei mit neuen Werten überschrieben. Das gleiche gilt auch für den optionalen „TWIBroadCastBuffer“.

Ein Broadcast wird empfangen, egal wie der interne Rx oder Tx Status aussieht. Er verändert diesen auch nicht, da ein Broadcast beim Master und bei den Slaves absolut unabhängig von den üblichen Frames behandelt wird.

Function *TWIrxFrame (node : byte) : boolean;*
Beim Master liest diese Funktion einen Frame von einem Slave aus. Die Funktion prüft zuerst, ob der Parameter „node“ <> 00h und < 128 ist, denn die Slave Adresse 00 ist für Broadcasts reserviert und Adressen > 127 sind nicht zulässig. Im Fehlerfall kehrt die Funktion mit einem False zurück. Dann wird geprüft, ob der zuletzt empfangene Frame von der Applikation verarbeitet wurde, d.h. ob „TWIrxStatReg“ 00h ist. Wenn nicht, kehrt die Funktion mit false zurück. Mit TWIrxClear kann der Frame zurückgesetzt werden und TWIrxFrame kann noch einmal aufgerufen werden. Der Parameter „node“ bestimmt welcher Slave adressiert werden soll.

1. Beim Master beginnt jetzt das Lesen des Slaves. Wenn der Slave reagiert, wird der Frame komplett übertragen. Im Erfolgsfall enthält das „TWIrxStatReg“ jetzt FFh. Reagiert der Slave nicht oder kommt es zu einem Übertragungsfehler ist „TWIrxStatReg“ jetzt ein Wert > 00h und < FFh enthalten. Wenn z.B. der Slave nicht reagiert, steht hier 80h. Die Master Applikation kann auch vor dem Empfangen mit der Funktion „GetTWIslaveStat(slavenode)“ prüfen, ob der Slave aktiv und sendebereit ist bevor er die Empfangs Funktion aufruft.
   Wenn der Frame empfangen werden konnte, enthält „TWIrxLen“ die Anzahl der Bytes im RxBuffer. Die Node Adresse des Slaves wird in „TWIrxAdr“ abgelegt.

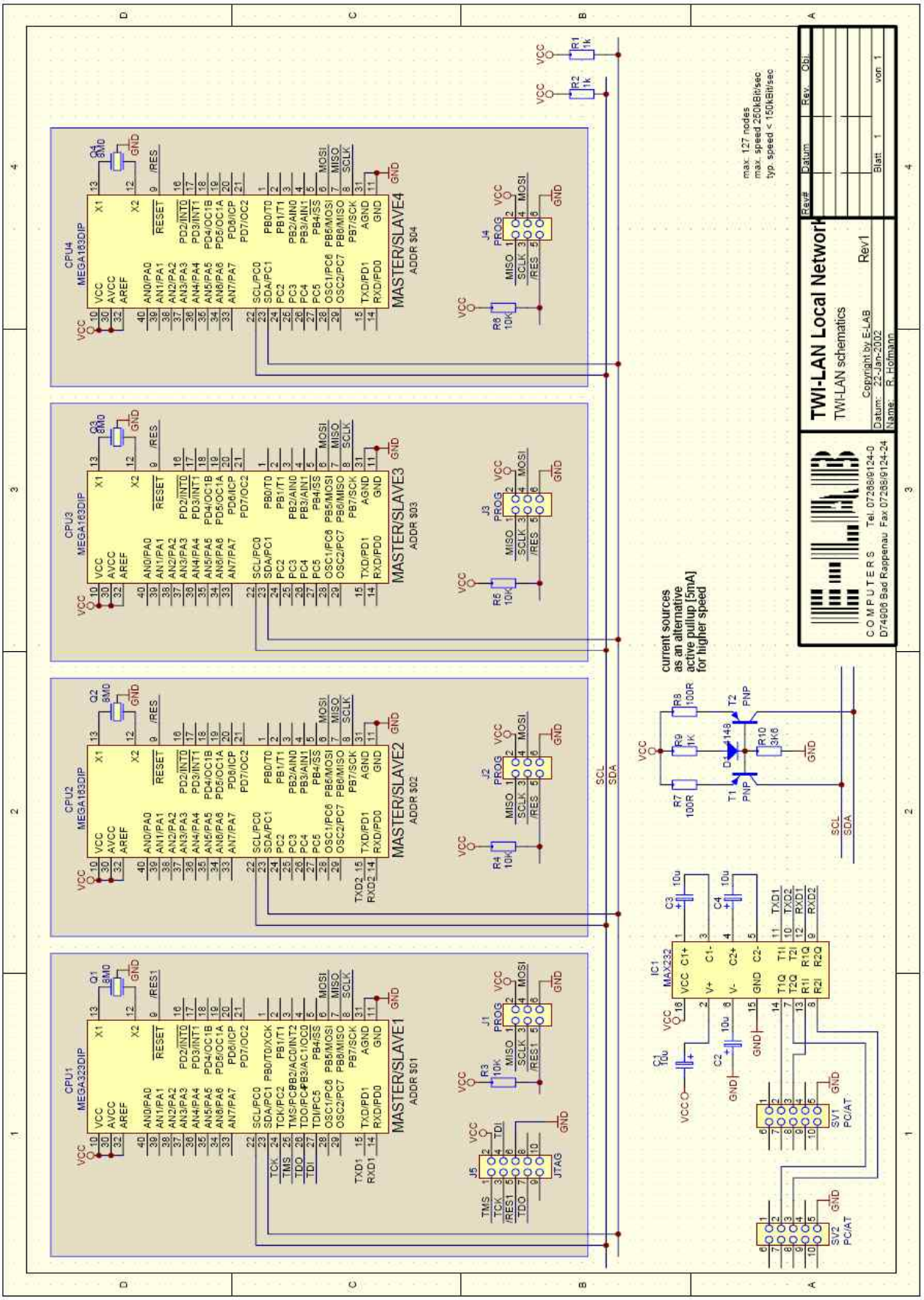
2. Wenn der Master einen Frame mit „TWIrxFrame“ komplett gelesen hat, wird im Slave das Statusregister “TWItxStatReg” auf 00h gesetzt und die Semaphore „TWItxSema“ wird inkrementiert.

Programm Beispiele und Schaltplan:

3 Beispiel Programme

TWInetMastr,
TWIslave und
TWInetMsSl

befinden sich im Verzeichnis **..\E-Lab\AVRco\Demos\TWInet**



Schaltplan **TWI_Net**

3.24.10 Zusätzliche Funktionen

Um im TWInet auch Standard I2C-Bausteine ansprechen zu können, sind die Basis Routinen für I2C Chips auch hier implementiert. In diesem Zusammenhang bedeutet hier „Slave“ ein normales I2C-chip und kein Netzwerk Slave! Niemals mit diesen Funktionen auf einen Netzwerk Slave zugreifen und ebenfalls nie mit Netzwerk Funktionen auf Standard I2C Chips zugreifen!

TWIstat
Die Funktion TWIstat stellt fest, ob der gewählte Slave vorhanden ist, bzw. sich meldet. Im Fehlerfall kehrt die Funktion mit dem Ergebnis FALSE zurück, ansonsten ist das Ergebnis TRUE. Ein EEprom z.B. ignoriert nach dem Beschreiben für eine gewisse Zeit jeden weiteren Zugriff und würde in diesem Fall ein false liefern. Ein echter TWI-Slave liefert immer ein false zurück, falls er intern gesperrt bzw. busy ist. In diesem Fall bedeutet ein false nicht, dass der Slave nicht vorhanden ist.

Die Definition von TWIstat ist:

Function *TWIstat (SlaveAdr : byte) : boolean;*

Der Parameter **SlaveAdr** ist die physikalische Baustein-Adresse auf dem TWI-Bus.

TWIinp
Die Funktion TWIinp liest mindestens ein Byte aus dem angewählten Slave. Schlägt der Versuch fehl kehrt die Funktion mit dem Ergebnis FALSE zurück, ansonsten ist das Ergebnis TRUE.

Die Definition von TWIinp ist:

Function *TWIinp (SlaveAdr : byte; Var Data) : boolean;*

Der Parameter **SlaveAdr** ist die physikalische Baustein-Adresse auf dem TWI-Bus. Baustein-interne Adressen (z.B. EEprom) müssen vorher mit einem Write durch TWIout eingestellt werden.

Die Variable **Data** muss als Variable definiert sein, wobei der Typ der Variablen fast beliebig sein kann. Das SizeOf(Data) bzw. die Anzahl der durch diese Variable belegte Speicherstellen bestimmen auch die Anzahl der Bytes, die im Block gelesen und in die Variable Data abgelegt werden. Bei einem Byte, Char oder Boolean sind das 1 byte, bei einem Word oder Integer sind das 2 Bytes, wobei das höherwertige Byte zuerst gelesen wird und dann das Low-Byte.

Ist Data ein Array-Typ, so wird die Länge des Arrays festgestellt und eine entsprechende Anzahl Bytes gelesen, beginnend mit der niederwertigsten Adresse (kleinstem Index) des Arrays.

Bei einem String-Typ als Data wird die max. mögliche Länge des Strings festgestellt und eine entsprechende Anzahl Bytes gelesen, beginnend mit dem Index string[0]. Das bedeutet, dass ein String immer mit seinem Längenbyte gelesen und geschrieben wird. Beim Lesen bestimmt aber nicht das Längenbyte die Anzahl der zu lesenden Bytes, sondern die Länge des Ziel-Strings:

Count := sizeOf(Destination).

Achtung:
Bei der Wahl der Data-Variable ist darauf zu achten, dass der Slave auch einen entsprechend langen Blocktransfer beherrscht. Ein TWI Baustein der nur aus einem 8-bit Parallel-Port besteht, kann in der Regel auch immer nur ein Byte übertragen.

TWIout
Die Funktion TWIout schreibt mindestens ein Byte in den angewählten Slave. Schlägt der Versuch fehl kehrt die Funktion mit dem Ergebnis FALSE zurück, ansonsten ist das Ergebnis TRUE.

Die Definition von TWIout ist:

```
Function TWIout (SlaveAdr, Command : byte|word [; Data]) : boolean;
```

Der Parameter **SlaveAdr** ist die physikalische Baustein-Adresse auf dem TWI-Bus.

Der Parameter **Command** ist Slave-abhängig, muss aber spezifiziert sein. Er kann eine Byte-Variable, eine 8-bit Konstante oder Word sein. Sehr oft ist dieses Kommando eine Baustein-interne Adresse (z.B. EEprom).

Bei **Slave CPUs** darf das Command immer nur **ein Byte** sein!!

Ist Command eine literale Konstante, z.B. 1, so wird angenommen, dass der Typ ein Byte ist. Sollte ein Word gewünscht sein, muss das dem Treiber mitgeteilt werden mit (addr, word(1), data);

Grössere EEproms (>256 bytes) besitzen eine 16-bit Adresse. Da es nicht sinnvoll
wäre, dafür spezielle Bibliotheksfunktionen zu implementieren, muss die Funktion **TWIout** informiert werden, ob der Adressparameter (2. Parameter) aus einem oder zwei Bytes besteht, also 8bit oder 16bit lang ist. Deshalb muss dieser Parameter eindeutig sein und damit entsprechend deklariert werden:

```
{ eeprom 8kBytes 24C65 }
Var    w  : word;                  { 16 bit adr }

w:=  0000;
b2:= 0;
bool:= TWIout ($50, w);            { set adr for read = 0000 }
while b2 < $FF do                  { read eeprom with adr auto incr}
  bool:= TWIinp ($50, b1);         { result into b1 }
  inc(b2);
endwhile;
```

Bei Komponenten, die eine weitere interne Adresse besitzen, wie z.B. EEproms, AD-Wandler und auch Slave-CPUs, muss, bevor gelesen wird, über einen Schreibzugriff die gewünschte Lese Adresse ausgegeben werden. Der nächste Lesezugriff geht dann auf diese Adresse. Besitzt der Baustein ein auto-inkrement der Adresse (ADC, EEprom, SlaveCPU), kann jetzt auch kontinuierlich gelesen werden, ohne eine neue Adresse auszugeben.

```
{ eeprom 256*8bit PCF8582E }
Var    b2 : byte;                  { 8 bit adr }
B2:=  00;
bool:= TWIout ($53, b2);           { set adr for read = 00 }
while b2 < $FF do                  { read eeprom with adr auto incr}
  bool:= TWIinp ($53, b1);         { result into b1 }
  inc(b2);
endwhile;
```

Die Variable **Data** ist optional und kann eine 8 oder 16-bit Konstante oder eine fast beliebige Variable sein. Rom-Konstante (Arrays, Strings und Stringliterale sind jedoch nicht möglich). Dieser Parameter ist optional und kann, muss aber nicht vorhanden sein. Der Slave bzw. die gewünschte Slave-Aktion bestimmt den Typ bzw. das weglassen von Data. Das SizeOf(Data) bzw. die Anzahl der durch diese Variable bzw. Konstante belegte Speicherstellen bestimmen auch die Anzahl der Bytes, die im Block aus der Variablen gelesen und in den Slave geschrieben werden. Bei einem Byte, Char oder Boolean sind das 1 byte, bei einem Word oder Integer sind das 2 Bytes, wobei das höherwertige Byte zuerst geschrieben wird und dann das Low-Byte.

Ist Result ein Array-Typ, so wird die Länge des Arrays festgestellt und eine entsprechende Anzahl Bytes geschrieben, beginnend mit der niederwertigsten Adresse (kleinstem Index) des Arrays.

Bei einem String-Typ als Data wird die Länge des Strings festgestellt SizeOf(String) und eine entsprechende Anzahl Bytes geschrieben, beginnend mit dem Index string[0]. Das bedeutet, dass ein String immer mit seinem Längenbyte gelesen und geschrieben wird. Beim Schreiben bestimmt aber nicht das Längenbyte die Anzahl der zu schreibenden Bytes, sondern die Länge des Source-Strings:

Count := sizeOf(Source).

Wurde der Parameter Data nicht angegeben, so wird nur das Command-Byte übertragen.

Achtung:
Bei der Wahl der Data-Variable/Konstante ist darauf zu achten, dass der Slave auch einen entsprechend langen Blocktransfer beherrscht. Ein I2C Baustein der nur aus einem 8-bit Parallel-Port besteht, kann in der Regel auch immer nur ein Byte übertragen.

Lock-up im Master Betrieb
Die interne TWI Hardware der mega CPUs reagiert sehr empfindlich auf externe Störungen, vor allem der CLK Leitung. Das kann u.U. zum kompletten Ausfall führen. Deshalb ist an diversen Stellen ein Timeout in den Treiber eingebaut. Tritt so ein TimeOut auf, wird die TWI Hardware neu initialisiert und die Library Funktion kehrt mit einem false zurück. Ist die externe Störung beseitigt, arbeitet das System ohne weitere Intervention der Applikation weiter.

Um diese TimeOut Funktion nützen zu können, muss der ***SysTick*** importiert sein.

3.25 AD-Wandler

3.25.1 ADCPORT

Definiert und importiert den AD-Wandler. (nur wenn im Chip vorhanden).

Der AD-Wandler wird im Hintergrund im Interrupt-Service *SysTick* gelesen und neu gestartet. Der Grund dafür ist, dass ein freilaufender ADC je nach Prozessor alle 10 bis 40usec einen Interrupt verursachen würde. Durch die notwendige Verwaltungsarbeit wäre das System mit diesen Interrupts 'dicht'. Polling des Wandlers hätte im Endeffekt das gleiche Resultat. Eine ADC Verwaltung durch SysTick z.B. alle 10msec fällt dagegen kaum ins Gewicht.

```
Import   SysTick, ADCPort;

Define   ProcClock   = 4000000;    {4Mhz clock }
         SysTick     = 10;         {10msec Tick}
         ADCchans    = 2, iData;   {2 Kanäle benutzen}
         ADCpresc    = 16;         {Vorteiler 16}
```

3.25.2 ADCchans, RAMpage

Gewünschte Anzahl der ADC-Kanäle.
Muss grösser 0 sein und maximal die im Prozessor Description File (xxx.dsc) definierte Kanalzahl. Die erfassten Werte werden in dafür reservierte Speicherstellen abgelegt (Achtung: Resourcen) und können jederzeit mittels GetAdc ausgelesen werden. Der Wert von z.B. 4 bedeutet, dass die Kanäle 1..4 benutzt bzw. gescannt werden.
Manchmal ist es notwendig, dass z.B. nur die Kanäle 4 und 7 benutzt werden. Um Resourcen zu sparen, können ein oder zwei beliebige Kanäle definiert werden:

```
ADCchans = [4, 7], iData;
```

oder auch nur ein Kanal

```
ADCchans = [5], iData;
```

oder mehrere Kanäle

```
ADCchans = [2..5], iData;
```

Ergebnis Integration:

```
ADCchans = 2, iData, int2;          {2-fach integrieren}
```

Mit "Int2" wird das vorhergehende Ergebnis zu dem neuen addiert und diese Summe durch 2 geteilt.
Result:= (prevValue + newValue) div 2;

ADCpresc
Gewünschter Clock Vorteiler des AD-Wandlers. Dieser Wert bestimmt die Wandlungszeit des ADCs und indirekt auch die Empfindlichkeit gegen Spikes und Rauschen des Signals.

3.25.3 Funktionen und Prozeduren

GetADC
Liefert Daten des ADC's. Funktion zum auslesen der ADC Kanäle.
ADCport und SysTick **müssen** importiert werden. Der Übergabe Parameter bezeichnet den gewünschten Kanal (1..*ADCport*). Parameter läuft von 1 bis n. Das Ergebnis der Funktion ist immer ein Word.

```
a:= GetAdc (1);          {ersten Kanal lesen}
```

Wenn nur **ein** Kanal definiert ist, muß der entsprechende Aufruf so sein:

```
a:= GetAdc;              {einzigen Kanal lesen}
```

SetADCfixed
Wurden beim ADC-Import mehr als ein Kanal definiert, so wird der ADC-Multiplexer bei jedem SysTick um eins weitergeschaltet.
Bei z.B. 6 importierten Kanälen kommt der Kanal jeden 6. Tick einmal zur Wandlung.
Das kann zu bestimmten Zeitpunkten innerhalb eines Programms zu lange dauern (Regelschleife).

Mit

```
Procedure SetAdcFixed (fix : boolean; chan : byte);
```

kann jetzt ein bestimmter Kanal kontinuierlich konvertiert werden (fix = true).
Die anderen Kanäle werden nicht weiter konvertiert, bis diese Prozedur mit fix = false noch einmal aufgerufen wird. Dabei spielt dann der Parameter "chan" keine Rolle mehr.

3.25.4 Call-Back Funktion onADCread

Für spezielle AD-Wandlungen ist es oft notwendig dass die Applikation informiert wird, wenn im SysTick ein ADC Kanal gelesen wird. Dafür wurde die Prozedur "OnACDread" implementiert.

```
Procedure onADCread;
```

Wenn im Anwendungsprogramm diese Prozedur implementiert ist

```
Procedure onADCread;
begin
...
end;
```

dann wird "onADCread" bei jedem SysTick (Timer Interrupt) aufgerufen und zwar an der Stelle, wo das ADC Ergebnis gelesen wird.
ACCB/ACCA (R16/R17) (lo/hi) enthalten dann das ADC Ergebnis und ACCALO (R18) enthält den aktuellen gelesenen Kanal (0..7). Man kann hier zum Beispiel nur einen ADC Kanal verwenden und mit einem externen analog Multiplexer arbeiten. Da der ADC Treiber aber das ADC Resultat immer in der gleichen Speicherstelle (Variable) ablegt, muss die Applikation das jetzt selbst handhaben und das Messergebnis abhängig von der Einstellung des Multiplexers selbst abspeichern und den ext. Multiplexer verändern.
Auch hier gilt das gleiche wie bei allen anderen Call-Back Funktionen aus dem SysTick heraus:

keine grosse Datenverarbeitung, Unterprogramm Aufrufe. Möglichst alles in Assembler schreiben.

Achtung:

1. Es dürfen keine lokalen Parameter definiert werden
2. Die Register SREG, _ACCA/R17, _ACCB/R16, _ACCALO/R18, _ACCCLO/R30 und _ACCCHI/R31 werden automatisch gesichert. Wenn Pascal Statements oder andere Register benutzt werden, müssen evtl. zusätzlich benutzte Register vorher gerettet werden.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\NonLinear**

3.26 AD-Wandler XMega

3.26.1 ADC_A ADC_B

Definiert und importiert die AD-Wandler. (soweit im Chip vorhanden).

Die XMegas besitzen bis zu 2 getrennte AD-Konverter. Diese sind jeweils fest den Ports A und B zugeordnet und werden deshalb als ADC_A und ADC_B bezeichnet.

Diese Konverter können vollkommen autark arbeiten. D.h. sie scannen selbstständig jeweils bis zu 8 Kanäle und stellen diese Ergebnisse in Registern bereit. Da ein Port bis zu 8 ADC Inputs haben kann muss die Applikation entscheiden welche von den 8 Pins gescannt werden sollen. Es werden bei dieser Implementation keine Interrupts und kein DMA verwendet. Der SysTick wird nur gebraucht wenn mehr als 4 Kanäle benutzt werden.

Die eingestellte Auflösung ist 12bit, single-ended, unsigned und right justified.

Import *SysTick, ADC_A, ADC_B;*

Define

```
 // The XMegas don't provide any Oscillator fuses.
 // So the application must setup the desired values
// possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz

//>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
OSCtype     = int32MHz,
              PLLmul=4,
              prescB=1,
              prescC=1;
SysTick     = 10;                 // {msec
StackSize   = $0032, iData;
FrameSize   = $0064, iData;

ADCrefA     = REF100;             // intern 1.0V reference, input pins at PortA, 4 channels
ADCprescA   = 256;                // prescaler 256
ADCchansA = [0..7];               // 8 channels in use, using SysTick

ADCrefB     = REFextB;            // extern reference, input pins at PortB, 3 channels
ADCprescB  = 32;                  // prescaler 32
ADCchansB = [1];                  // single channel, no SysTick used
```

3.26.2 Defines

ADCrefA, ADCrefB
Dieses Define bestimmt die Referenz Spannungs Quelle.

REF100	= interne 1.0Volt Referenz
REFintVCC	= interne Referenz VCC div 1.6
REFintVCC2	= interne Referenz VCC div 2
REFextA	= externe Referenz an PortA
REFextB	= externe Referenz an PortB

ADCprescA, ADCprescB
Gewünschter Clock Vorteiler des AD-Wandlers. Dieser Wert bestimmt die Wandlungszeit des ADCs und indirekt auch die Empfindlichkeit gegen Spikes und Rauschen des Signals (4, 8, 16, 32, 64, 128, 256, 512). Optional kann auch noch die Auflösung mit angegeben werden:

```
ADCprescA   = 4, 8;               // prescaler 4, resolution 8bits (8 oder 12)
```

Ohne diese Angabe ist die Auflösung immer 12bit.

ADCchansA, ADCansB
Bestimmt die Anzahl der gewünschten Kanäle und auch der jedem Kanal zugeordnete Port-Pin.
Mögliche Defines sind z.B.:

```
ADCchansA = [0..7];          // 8 channels in use PortA.0 .. PortA.7
ADCchansA = [1..3];          // 3 channels in use PortA.1 .. PortA.3
ADCchansA = [1, 3, 5, 7];    // 4 channels in use PortA.1, PortA.3, PortA.5, PortA.7
ADCchansA = [0];             // 1 channel in use PortA.0
ADCchansA = 1;               // 1 channel in use PortA.1
```

Zu beachten ist dass wenn mehr als 4 Kanäle/Pins definiert sind dass dann der SysTick benötigt wird!

3.26.3 Funktionen und Prozeduren

GetADC
Liefert Daten des ADC's. Funktion zum auslesen der ADC Kanäle.
ADC_A oder ADC_B oder beide **müssen** importiert werden. Der Übergabe Parameter bezeichnet den gewünschten Kanal (0..*7*). Das Ergebnis der Funktion ist immer ein Word.

```
Function GetAdcA(chan : byte) : word;
Function GetAdcB(chan : byte) : word;
```

Der Parameter chan muss im Bereich 0..7 sein. Wenn aber z.B. nur 2 Pins definiert wurden, dann läuft chan von 0..1. Wurden z.B. die Pins 2,4,5,7 als Input gewählt, dann gibt es diese Zuordnung:

Port.Pin2 > chan0
Port.Pin4 > chan1
Port.Pin5 > chan2
Port.Pin7 > chan3

Falsche Werte von chan haben als Resultat $0000 zur Folge.

```
 w:= GetAdcA(0);       {read first channel}
w:= GetAdcB(3);
```

Einschränkungen bei den XMegaE5 Typen:
Mehr als ein Kanal lässt sich nur mit [m..n]

```
ADCchansA = [1..3];          // 3 channels in use PortA.1 .. PortA.3
```

Definieren.
ADC_B existiert nicht und damit auch keine Defines dazu.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMEGA_ADC**

3.27 DA-Wandler XMega

3.27.1 DAC_A DAC_B

Definiert und importiert die DA-Wandler. (soweit im Chip vorhanden).

Die XMegas besitzen bis zu 2 getrennte DA-Konverter. Diese sind jeweils fest den Ports A und B zugeordnet und werden deshalb als DAC_A und DAC_B bezeichnet. Die eingestellte Auflösung ist 12bit und right justified.

```
Import SysTick, DAC_A, DAC_B;

From System Import ;

Define
  // The XMegas don't provide any Oscillator fuses.
  // So the application must setup the desired values
  // possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz

  //>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
  OSCtype       = int32MHz, PLLmul=4, prescB=1, prescC=1;
  SysTick       = 10;          {msec}
  StackSize     = $0032, iData;
  FrameSize     = $0064, iData;

  DAC_A         = chan01, REF100;      // DAC_A channel 0 + 1 used
  DAC_B         = chan0, REFextB;      // DAC_B channel 0 used
XMega "U"  Typen:
  DAC_A         = chan1, REFextB;      // DAC_A channel 1 only used
  DAC_B         = chan1, REFextB;      // DAC_B channel 1 only used
```

3.27.2 Defines

DAC_A, DAC_B
Jeder DAC hat bis zu 2 Kanäle, chan0 und chan1. Ist nur ein Kanal vorhanden oder soll nur ein Kanal benutzt werden, muss chan0 vorgegeben werden. Sind zwei Kanäle vorhanden und sollen beide benutzt werden, so wird chan01 angegeben. Der zweite Parameter bestimmt die Referenz Spannungs Quelle.

REF100	= interne 1.0Volt Referenz
REFaVCC	= externe Referenz an AVCC Pin
REFextA	= externe Referenz an PortA
REFextB	= externe Referenz an PortB

3.27.3 Funktionen und Prozeduren

SetDAC
Schreibt einen neuen Ausgabewert (12bit) in das DAC Register. DAC_A oder DAC_B oder beide **müssen** importiert und definiert sein.

```
Function SetDacA_chan0(val : word);
Function SetDacA_chan1(val : word);
Function SetDacB_chan0(val : word);
Function SetDacB_chan1(val : word);

SetDacA_chan0(0);           // write to DACA chan0
SetDacB_chan1(ww);          // write to DACB chan1
```

Programm Beispiel:
ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMEGA_DAC**

3.28 Real Time Clock

Manche AVRs (4434, 8535, mega83, mega161, mega163, mega103 etc) besitzen einen sogenannten Real Time Counter, was allerdings nicht ganz vergleichbar ist zu eine RTC-Chip. Allerdings lässt sich damit ohne allzu grossen Software und Hardware Aufwand eine Echtzeit Uhr bilden. Voraussetzung ist, dass die CPU im IO-Bereich ein „ASSR“ Register besitzt und die Möglichkeit, einen externen Uhrenquarz anzuschliessen. Das ist zumindest bei den oben aufgeführten CPU-Typen gegeben. Bei CPU-Typen, die keine RTC besitzen (z.B. 8515) oder wo die Timer anderweitig benötigt werden, lässt sich ohne allzu grossen zusätzlichen Aufwand im SysTick eine recht genaue Uhr nachbilden.

Die **RTC** bietet die Stellmöglichkeit von Zeit (hh:min:sec) und optional Datum (yy.mm.dd) und die entsprechenden Rücklese Funktionen. Für die RTC-Zeit sind sogenannte CallBack Funktionen vorgesehen. Diese werden, falls implementiert, bei jeder neuen Sekunde, Minute oder Stunde aufgerufen (Ticks).

Durch zusätzliche (optionale) Imports können auch einer oder mehrere **Timer** (DownCount Alarm) und ein **Alarm** (Vorgabe Zeit/Datum) definiert werden. Beide Zusatz Funktionen besitzen ebenfalls einen CallBack der die entsprechende Prozedur im Erreignisfall aufruft.

Die RTC benötigt das ASSR Register, den zugehörigen Timer (Timer0 oder Timer2) und einen externen 32768Hz Uhrenquarz. Wenn mit dem SysTick gearbeitet wird, entfallen der Timer und der Quarz.

Wie bei allen RTCs besteht nicht nur das Problem, möglichst präzise zu sein, sondern das Setup Problem der Uhr nach PowerUp muss auch gelöst werden. Sehr hilfreich ist hier der AVRco DCF77 Treiber. Bitte lesen Sie dazu die zugehörige Dokumentation.

Imports
Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import SysTick, RTclock, ..;
```

Die optionalen Ereignis Funktionen müssen, falls benötigt, ebenfalls importiert werden.

```
From RTclock Import RTCtimer, RTCalarm;
```

Die Art der Uhr (nur Zeit oder Datum+Zeit) sowie die Lage der Software Uhren Register werden mit Define vorgegeben:

Defines (mega103)

```
Define   ProcClock = 6000000;      {Hertz}
         SysTick   = 10 , Timer2;  {msec}
         RTclock   = iData, Time;  {Time or DateTime}
```

Optional können auch mehrere Timer Kanäle mit Define vorgegeben werden

```
Define   RTCtimer  = 4;                  {1..8 Channels}
```

Als Alternative zum internen RTC Kanal:

```
Define   RTCsource = SysTick[, adj];     {optional}
```

Achtung:
Wird als RTC-source der SysTick eingestellt, so muss der SysTick bestimmte Bedingungen erfüllen. Er darf nicht kleiner 1 sein, er muss ganzzahlig sein und 1000 / SysTick darf keinen Rest ergeben (1000 mod SysTick = 0).

3.28.1 RTC-Funktionen/Prozeduren

Grundsätzliche und Implementations unabhängige Prozedur und Funktionen.

Procedure *RTCsetSecond (sec : byte);*
Setzt das Sekunden Register mit dem Wert „sec“

Procedure *RTCsetMinute (min : byte);*
Setzt das Minuten Register mit dem Wert „min“

Procedure *RTCsetHour (hour : byte);*
Setzt das Stunden Register mit dem Wert „hour“

Procedure *RTCsetDay (day : byte);*
Setzt das Tages Register mit dem Wert „day“. Nur möglich, wenn das Define mit „DateTime“ erfolgte.

Procedure *RTCsetWeekDay (wday : byte); {0 = Monday}*
Setzt das Wochentag Register mit dem Wert „wday“. Nur möglich, wenn das Define mit „DateTime“ erfolgte.

Procedure *RTCsetMonth (month : byte);*
Setzt das Monats Register mit dem Wert „month“. Nur möglich, wenn das Define mit „DateTime“ erfolgte.

Procedure *RTCsetYear (year : byte);*
Setzt das Jahres Register mit dem Wert „year“. Nur möglich, wenn das Define mit „DateTime“ erfolgte.

Achtung: Das Stellen der Uhr sollte nur bei gesperrtem Interrupt erfolgen!

Function *RTCgetSecond : byte;*
Liest das Sekunden Register

Function *RTCgetMinute : byte;*
Liest das Minuten Register

Function *RTCgetHour : byte;*
Liest das Stunden Register

Function *RTCgetDay : byte;*
Liest das Tages Register. Nur möglich, wenn das Define mit „DateTime“ erfolgte.

Function *RTCgetWeekDay : byte; {0 = Monday}*
Liest das Wochentags Register. Nur möglich, wenn das Define mit „DateTime“ erfolgte.

Function *RTCgetMonth : byte;*
Liest das Monats Register. Nur möglich, wenn das Define mit „DateTime“ erfolgte.

Function *RTCgetYear : byte;*
Liest das Jahres Register. Nur möglich, wenn das Define mit „DateTime“ erfolgte.

Achtung: Das Lesen der Uhr sollte nur bei gesperrtem Interrupt erfolgen!

Die folgenden drei Prozeduren sind sogenannte ***CallBack*** Prozeduren. Das heisst, dass diese, wenn vorhanden, bei jedem Sekunden, Minuten oder Stunden Übertrag automatisch aufgerufen werden. Hierbei ist aber unbedingt zu beachten, dass diese Funktionen aus dem zugehörigen Timer-Hardware-Interrupt heraus aufgerufen werden.
Das bedeutet, wie bei allen anderen Interrupt Prozeduren auch, dass nur die wesentlichen Register (_ACCA/R16, _ACCB/R17 und das Z-Register) gerettet wurden. Deshalb ist darauf zu achten, dass innerhalb dieser Prozeduren auch nur diese Register benutzt werden können. Andernfalls sind zusätzliche Register manuell zu sichern.

Diese Prozeduren müssen in der Source erstellt werden!

Procedure *RTCtickSecond; {CallBack procedure}*
Wird bei jedem Sekunden Übertrag aufgerufen.

Procedure *RTCtickMinute; {CallBack procedure}*
Wird bei jedem Minuten Übertrag aufgerufen.

Procedure *RTCtickHour; {CallBack procedure}*
Wird bei jedem Stunden Übertrag aufgerufen.

3.28.2 Alarm-Prozeduren

Wurde **RTCalarm** importiert, so steht eine Alarmfunktion zur Verfügung, die beim Erreichen der Vorgabezeit oder Zeit+Datum die CallBack Prozedur „RTCalarm“ aufruft.

Procedure *RTCalarm_Time (hour, min, sec : byte);*
Setzt die Zeit-Compare Register von RTCalarm. Compare wird gestoppt.

Procedure *RTCalarm_Date (year, month, day : byte);*
Setzt die Datum-Compare Register von RTCalarm. Nur wenn bei dem Define „DateTime“ angegeben wurde. Compare wird gestoppt.

Procedure *RTCalarm_Start (mode : byte); {mode 0= stop, 1= Time, 2= DateTime}*
Startet oder stoppt RTCalarm.

Procedure *RTCalarm_Stop;*
Stoppt RTCalarm.

Procedure *RTCalarm; {CallBack procedure}*
Wird bei einem Compare-match aufgerufen, d.h. maximal einmal.
Diese Prozedur muss in der Source erstellt werden!

3.28.3 Timer-Prozeduren

Wurde **RTCtimer** importiert, so stehen eine oder bis zu 8 Timerfunktionen zur Verfügung, die bei jedem Sekunden Tick den Vorgabe Wert um eins dekrementiert. Ist dieser Wert 0 so wird die CallBack Prozedur „RTCtimer“ aufgerufen.

Procedure *RTCtimer_Load (seconds : word[longword]); // single timer defined*
Procedure *RTCtimer_Load (chan : byte; seconds : word[longword]); // multiple timer defined*

Setzt die Zeit-Compare Register von RTCtimerX. Compare wird gestoppt.
Wurden LongWord oder LongInts im System importiert, so ist seconds ein LongWord (32bits), ansonsten ein Word (16bits).

Procedure *RTCtimer_Start; // single timer imported*
Procedure *RTCtimer_Start (chan : byte); // multiple timer imported*
Startet RTCtimer.

Procedure *RTCtimer_Stop; // single timer imported*
Procedure *RTCtimer_Stop (chan : byte); // multiple timer imported*
Stoppt RTCtimer.

Procedure *RTCtimer; {CallBack procedure}*
Procedure *RTCtimer (chan : byte); {CallBack procedure}*

Wird bei Downcounter = 0 aufgerufen, d.h. einmal. Der Timer wird gestoppt.
Diese Prozedur muss in der Source erstellt werden!

Bemerkungen:

Die RTC muss im Timer0/2 **Interrupt** laufen. Wird der Interrupt länger als eine Sekunde gesperrt, so ist mit faschen Werten zu rechnen.

CallBack Prozeduren werden grundsätzlich aus dem Timer Interrupt heraus aufgerufen, mit minimaler Register Sicherung. Deshalb sorgfältig den in der Prozedur generierten Assembler Code untersuchen. Im Zweifelsfall Register sichern (ASM) oder den Prozedur Code in Assembler schreiben.

CallBack Prozeduren müssen selbst erstellt werden.

```
Procedure RTCtimer;              {CallBack procedure, only one timer imported}
Begin
  Elapsed:= true;
End;

Procedure RTCtimer(chan : byte);     {CallBack procedure, more than one timer imported}
Begin
  Elapsed:= true;
End;
```

Stellen und **Auslesen** der RTC Register sollte nur bei gesperrtem Interrupt erfolgen! Ansonsten kann es zu fehlerhaften Resultaten kommen, wenn während eines solchen Vorgangs der Timer Interrupt zuschlägt.

```
DisableInts;
RTCsetSecond (x);
RTCsetMinute (y);
RTCsetHour (z);
EnableInts;
```

Bei allen anderen Operationen (auch RTCtimer_Load, RTCalarm_Time und RTCalarm_Date) ist ein Sperren nicht notwendig.

SysTick benutzt einen 8bit Timer und die RTC auch. Der Timer für die RTC ist in der CPU fest vorgeben. Daher muss für den SysTick der jeweils andere 8bit Timer benutzt werden (Timer0 oder Timer2).

Wird die RTC mittels „Define RTCsource = SysTick“ durch den SysTick betrieben, wird kein Timer benötigt. Der optionale Parameter ***adj*** bei der Definition der RTCsource = SysTick dient zum fein Justieren der RTC. Hiermit kann durch ausprobieren des Wertes (min –100, max +100) der Korrektur-Wert bestimmt werden, mit dem die RTC am genauesten läuft. Damit ist eine Genauigkeit von +/- 1sec/Tag erreichbar. Zum Erfassen des Wertes wird eine präzise Uhr, z.B. DCF77 Uhr benötigt.

Programm Beispiele:

sind unter **..\E-Lab\AVRco\Demos\RTclock** und **..\E-Lab\AVRco\Demos\RTC8564** zu finden

3.29 I2C-Bus

3.29.1 I2CPORT

Definiert und importiert die I2C-Bus Schnittstelle.

Der I2C-Bus wird softwaremässig implementiert. Die Schnittstelle kann dabei nur als Bus-Master arbeiten. Eine evtl. vorhandene Hardware-I2Cbus-Schnittstelle wird dabei nicht benutzt und auch nicht unterstützt.

Wird mittels Import das I2Cport importiert, so muss auch mittels Define das zugehörige Parallel-Port und die beiden benötigten Bits I2Cclk und I2Cdat definiert werden. Die zwei Bits müssen im gleichen Port sein. Das Port muss bidirektional arbeiten können.

Der Import von I2Cport importiert auch automatisch die Bibliotheksfunktionen I2Cstat, I2Cinp und I2Cout. Diese Funktionen unterstützen neben Einzelbyte-Transfers auch Block-Transfers. Die Funktionen liefern entsprechend dem Ergebnis ein True oder False zurück.
I2Cout akzeptiert auch Records und Arrays aus dem RAM, Flash und EEProm als Parameter.

```
Import   SysTick, I2Cport;

Define   ProcClock   = 4000000;    {4Mhz clock }
         SysTick     = 10;         {10msec Tick}
         I2Cport     = PortB;      {benutze port B}
         I2Cclk      = 0;          {clock-pin = port B bit 0}
         //I2Cclk    = 0,NOPs      {NOPs1, NOPs2…NOPs10}
         //I2Cclk    = 0, 2        {2x sDelay}
         //I2Cclk    = 0, @myDel   {x sDelay}  // myDel = variable which can be changed at runtime
         I2Cdat      = 3;          {data-pin  = port B bit 3}
```

I2Cclk
Definiert den Portpin 'Clock' des Ports, der mit I2Cport für den I2Cbus Betrieb spezifiziert wurde. Das angegebene Bit muss als Ein- und Ausgang schaltbar sein. Siehe auch I2Cdat
Bereich 0..7.

Die Taktrate kann über das optionale
```
Define   I2Cclk  = PortBit, NOPs
```

eingestellt werden. "NOPs" ist entweder ein Wert (Konstante) zw. 1..255 oder ein Variablen-Namen. Die Konstante / Variable bestimmt die Anzahl der CPU-Zyklen zw. jedem Status Wechsel der Daten- und Clock-Leitung des I2C. Die Variable kann zu Laufzeit geändert werden und erlaubt dadurch eine flexible Anpassung an unterschiedliche I2C Chips. In der Praxis werden diese NOPs durch ein **sDelay** ausgeführt. Dieses Delay braucht ca. 3 CPU Zyklen für einen Durchlauf. Wird hier also ein Wert von 10 angegeben, sind das dann ca. 30 CPU Zyklen effektiv. Anstatt der Konstante oder Variablen können auch direkt die Anzahl der NOPs angegeben werden:

```
Define  I2Cclk  = 0, NOPs3;    // PortBit, 3 NOPs  [0..10]
```

I2Cdat
Definiert den Portpin 'Data' des Ports, der mit I2Cport für den I2Cbus Betrieb spezifiziert wurde.
Das angegebene Bit muss als Ein- und Ausgang schaltbar sein. Siehe auch I2Cclk
Bereich 0..7

Alternativ kann auch das komplette Define in einer Zeile angegeben werden:

```
Define   I2Cport  = Port, DataPin, ClockPin[, NOPs];        // NOPs = optional parameter
```

Beispiel:
```
Define   I2Cport  = PortC, 2, 3, NOPs2;   // 2 nops into the clock delay
Define   I2Cport  = PortC, 2, 3, 2;        // 2x sDelays into the clock delay, XMegas
```

3.29.2 Funktionen und Prozeduren

I2Cstat
Die Funktion I2Cstat stellt fest, ob der gewählte Slave vorhanden ist, bzw. sich meldet. Im Fehlerfall kehrt die Funktion mit dem Ergebnis FALSE zurück, ansonsten ist das Ergebnis TRUE. Ein EEprom z.B. ignoriert nach dem Beschreiben für eine gewisse Zeit jeden weiteren Zugriff und würde in diesem Fall ein false liefern. Es werden max. 255 Versuche bis zum Abbruch unternommen. Im Fehlerfall kehrt die Funktion nach ca. 6msec mit false zurück.

***Function** I2Cstat (SlaveAdr : byte) : boolean;*

Der Parameter **SlaveAdr** kann eine Byte-Variable oder eine 8-bit Konstante sein. Laut Definition ist das 8. Bit der Adresse das Read-Write Bit und wird deshalb von der Funktion überschrieben bzw. ignoriert. Diese Adresse ist die physikalische Baustein-Adresse auf dem I2C-Bus.

I2CIinp
Die Funktion I2Cinp liest mindestens ein Byte aus dem angewählten Slave. Schlägt der Versuch fehl (TimeOut) kehrt die Funktion mit dem Ergebnis FALSE zurück, ansonsten ist das Ergebnis TRUE. Es werden max. 255 Versuche bis zum Abbruch unternommen. Im Fehlerfall kehrt die Funktion nach ca. 6msec mit false zurück.

***Function** I2Cinp (SlaveAdr : byte; **var** Data) : boolean;*

Der Parameter **SlaveAdr** kann eine Byte-Variable oder eine 8-bit Konstante sein. Laut Definition ist das 8. Bit der Adresse das Read-Write Bit und wird deshalb von der Funktion überschrieben bzw. ignoriert. Diese Adresse ist die physikalische Baustein-Adresse auf dem I2C-Bus. Baustein-interne Adressen (z.B. EEprom) müssen vorher mit einem Write durch I2Cout eingestellt werden.
Die Variable **Data** muss als Variable definiert sein, wobei der Typ der Variablen fast beliebig sein. Das SizeOf(Data) bzw. die Anzahl der durch diese Variable belegte Speicherstellen bestimmen auch die Anzahl der Bytes, die im Block gelesen und in die Variable Data abgelegt werden. Bei einem Byte, Char oder Boolean sind das 1 byte, bei einem Word oder Integer sind das 2 Bytes, wobei das niederwertige Byte zuerst gelesen wird und dann das High-Byte.

Ist Data ein Array-Typ, so wird die Länge des Arrays festgestellt und eine entsprechende Anzahl Bytes gelesen, beginnend mit der niederwertigsten Adresse (kleinstem Index) des Arrays.
Bei einem String-Typ als Data wird die max. mögliche Länge des Strings festgestellt und eine entsprechende Anzahl Bytes gelesen, beginnend mit dem Index string[0]. Das bedeutet, dass ein String immer mit seinem Längenbyte gelesen und geschrieben wird. Beim Lesen bestimmt das gelesene Längenbyte die Anzahl der zu lesenden Bytes. Die max. Länge des Ziel Strings wird aber nicht überschritten.

Achtung:
Bei der Wahl der Data-Variable ist darauf zu achten, dass der Slave auch einen entsprechend langen Blocktransfer beherrscht. Ein I2C Baustein der nur aus einem 8-bit Parallel-Port besteht, kann in der Regel auch immer nur ein Byte übertragen.

I2Cout
Die Funktion I2Cout schreibt mindestens ein Byte in den angewählten Slave. Schlägt der Versuch fehl (TimeOut) kehrt die Funktion mit dem Ergebnis FALSE zurück, ansonsten ist das Ergebnis TRUE. Es werden max. 255 Versuche bis zum Abbruch unternommen. Im Fehlerfall kehrt die Funktion nach ca. 6msec mit false zurück.

***Function** I2Cout (SlaveAdr : byte; Command : byte|word [; Data]) : boolean;*

Der Parameter **SlaveAdr** kann eine Byte-Variable oder eine 8-bit Konstante sein. Laut Definition ist das 8. Bit der Adresse das Read-Write Bit und wird deshalb von der Funktion überschrieben bzw. ignoriert. Diese Adresse ist die physikalische Baustein-Adresse auf dem I2C-Bus.

Der Parameter **Command** ist Slave-abhängig, muss aber spezifiziert sein. Er kann eine Byte-Variable, eine 8-bit Konstante oder word sein. Sehr oft ist dieses Kommando eine Baustein-interne Adresse (z.B. EEprom).

Grössere EEproms (>256 bytes) besitzen eine 16-bit Adresse. Da es nicht sinnvoll wäre, dafür spezielle Bibliotheksfunktionen zu implementieren, muss die Funktion **I2Cout** informiert werden, ob der Adressparameter (2. Parameter) aus einem oder zwei Bytes besteht, also 8bit oder 16bit lang ist. Deshalb muss dieser Parameter eindeutig sein und damit entsprechend deklariert werden.

```
{ eeprom 8kBytes 24C65 }
var w  : word;                  { 16 bit adr }

w:=  0000;
b2:= 0;
bool:= I2Cout ($50, w);         { set adr for read = 0000 }
while b2 < $FF do               { read eeprom with adr auto incr}
  bool:= I2Cinp ($50, b1);      { result into b1 }
  inc(b2);
endwhile;
```

Bei Komponenten, die eine weitere interne Adresse besitzen, wie z.B. EEproms und AD-Wandler, muss, bevor gelesen wird, über einen Schreibzugriff die gewünschte Lese Adresse ausgegeben werden. Der nächste Lesezugriff geht dann auf diese Adresse. Besitzt der Baustein ein auto-inkrement der Adresse (ADC, EEprom), kann jetzt auch kontinuierlich gelesen werden, ohne eine neue Adresse auszugeben.

```
{ eeprom 256*8bit PCF8582E }
var b2 : byte;                  { 8 bit adr }

B2:=  00;
bool:= I2Cout ($53, b2);        { set adr for read = 00 }
while b2 < $FF do               { read eeprom with adr auto incr}
  bool:= I2Cinp ($53, b1);      { result into b1 }
  inc(b2);
endwhile;
```

Die Variable **Data** ist optional und kann eine 8 oder 16-bit Konstante oder eine fast beliebige Variable oder Konstante sein. Dieser Parameter ist optional und kann, muss aber nicht vorhanden sein. Der Slave bzw. die gewünschte Slave-Aktion bestimmt den Typ bzw. das weglassen von Data. Das SizeOf(Data) bzw. die Anzahl der durch diese Variable bzw. Konstante belegte Speicherstellen bestimmen auch die Anzahl der Bytes, die im Block aus der Variablen gelesen und in den Slave geschrieben werden. Bei einem Byte, Char oder Boolean sind das 1 byte, bei einem Word oder Integer sind das 2 Bytes, wobei das niederwertige Byte zuerst geschrieben wird und dann das High-Byte.

Ist Result ein Array-Typ, so wird die Länge des Arrays festgestellt und eine entsprechende Anzahl Bytes geschrieben, beginnend mit der niederwertigsten Adresse (kleinstem Index) des Arrays.

Bei einem String-Typ als Data wird die Länge des Strings festgestellt length(String) und eine entsprechende Anzahl Bytes geschrieben, beginnend mit dem Index string[0]. Das bedeutet, dass ein String immer mit seinem Längenbyte gelesen und geschrieben wird. Beim Schreiben bestimmt auch das Längenbyte die Anzahl der zu schreibenden Bytes:

Count := length(Source).

Wurde der Parameter Data nicht angegeben, so wird nur das Command-Byte übertragen.

Achtung:
Bei der Wahl der Data-Variable/Konstante ist darauf zu achten, dass der Slave auch einen entsprechend langen Blocktransfer beherrscht. Ein I2C Baustein der nur aus einem 8-bit Parallel-Port besteht, kann in der Regel auch immer nur ein Byte übertragen.

3.29.3 Die I2C-BUS Schnittstelle

Die I2C Schnittstelle ist eine international standardisierte, synchrone serielle Schnittstelle zum Anschluss von Peripheriebausteinen an einen Prozessor und auch zur Kommunikation von Prozessoren untereinander. Die hier vorliegende Implementation unterstützt die hauptsächlich verwendete Master-Slave Konfiguration, wobei die CPU der Master ist und das angeschlossene Bauteil der Slave.

Es können mehrere Slaves angeschlossen werden, wobei jeder Slave eine spezielle, interne Adresse besitzt, die im sog. Netz nur einmal vorkommen sollte. Da theoretisch über 100 Bausteine betrieben werden können, kann man auch von einem Netzwerk sprechen. Die Variante mit 10-Bit Adressen wird nicht unterstützt.

Die Verbindung untereinander geschieht mittels zwei Steuerleitungen: Daten und Takt. Diese beiden Leitungen sind bi-direktional, arbeiten also in beiden Richtungen. Deshalb muss das dafür vorgesehene Port bzw. zumindest die dafür benutzten Pins in der Datenrichtung umschaltbar sein oder eine TriState Funktion haben. Beide Leitungen müssen mit jeweils einem Pull-Up Widerstand von 10..50kOhm gegen +5Volt versehen sein.

Die durch I2Cport importierte und durch I2Cclk und I2Cdat definierte I2C-Schnittstelle bietet die drei High-Level Funktionen I2Cstat, I2Cinp und I2Cout. Diese Funktionen sind generalisiert, d.h. sie sind auf keinen speziellen Chip angepasst. Der Programmierer kann bzw. muss durch entsprechende Verwendung und Parameter das Software-Protokoll des Slave einhalten, wobei das Telegramm-Protokoll (Hardware-Adresse, Clock-Generierung etc) durch den Bibliothekstreiber realisiert wird.

Durch das vorgeschriebene Hardware-Protokoll erkennen die alle Funktionen, ob der Slave vorhanden und bereit oder nicht vorhanden oder BUSY ist. Ist die Funktion fehlgeschlagen wird ein FALSE zurückgeliefert, ansonsten ein TRUE.

Die zwei Transfer-Funktionen bieten die Möglichkeit Blöcke zu lesen und zu schreiben, wobei der Typ des Parameters "Data" automatisch die Blocklänge bestimmt. Weiteres unter I2Cinp und I2Cout.

3.29.4 Multi-Processing und I2C

In einer Applikation mit Prozessen und/oder Tasks kommt es häufig vor, dass der I2C-Bus nicht nur als Netzwerk sondern auch für andere Zwecke (LCD, Ports etc) gebraucht wird. Wenn dabei die Zugriffe auf das I2C aus unterschiedlichen Prozessen heraus erfolgen, kommt es unweigerlich zu Konflikten da solche sequentiellen Treiber (I2C, TWI, UART etc) nicht re-entrant sind, d.h. sie sind nicht unterbrechbar und neu aufrufbar. Deshalb besitzt das I2C Port ganz allgemein eine Semaphore vom Typ DeviceLock.

I2C_DevLock : DEVICELOCK;

Der I2C Treiber beachtet und steuert diese Semaphore. Beim Eintritt in den Treiber wird geprüft ob der Treiber frei ist (Semaphore inaktiv). Ist das der Fall, dann wird die Semaphore aktiviert = Treiber gesperrt und der Auftrag wird ausgeführt. Nach dem Beenden des Auftrags wird die Semaphore wieder freigegeben.

Wird beim Eintritt festgestellt, dass der Treiber belegt ist (Semaphore aktiv) dann wird ein ***Schedule*** durchgeführt und dieser Prozess reiht sich wieder in die Warteschlange ein. Bei einem der nächsten Prozess Wechsel wird der Prozess wieder gestartet und prüft jetzt erneut die Semaphore. Das wiederholt sich jetzt solange, bis die Semaphore wieder frei ist.

Achtung:
Durch den Abbruch eines I2C-Aufrufs durch „Schedule“ sollten für den I2C Zugriff ***keine Tasks*** verwendet werden, da Tasks damit komplett abgebrochen werden und der Auftrag in diesem Fall nicht ausgeführt wird. Mann kann sich jedoch ein Flag setzen, wenn der Task mit seinem Auftrag erfolgreich war und wenn nicht, muss der Task wiederholt werden. Das bringt jedoch eine etwas komplizierte Verwaltung mit sich.

```
Import    SysTick, I2Cport;
Define    ProcClock  = 4000000;     {4Mhz clock }
          SysTick    = 10;          {10msec Tick}
          2Cport     = PortB;       {benutze port $05}
          I2Cclk     = 0;           (clock-pin = port B bit 0}
          I2Cdat     = 3;           (data-pin  = port B bit 3}

Var       ar1     : array[1..4] of byte;
          b       : byte;
          bool    : boolean;

 {adr 0 zum lesen einstellen}
 if not I2Cout(%01010000, $0) then   {EEprom select, intern adr 0}
   Error;
 else  {lesen}
   bool:= I2Cinp(%01010000, b);      {EEprom select, read adr 0 -> b}
 endif;

{einen Block einlesen, EEprom hat adr autom. inkrementiert}
bool:= I2Cinp (%01010000, ar1);      {EEprom select, read adr 1..4 -> ar1}

{EEprom Schreib-Operation}
if not I2Cout ($50, $0, b) then      {EEp select, intern adr 0, Byte b prog}
  Error;
else   {Block schreiben}
  bool:= I2Cout ($50, 1, ar1);       {EEp select, ar1 -> EEp-adr 1..4 prog}
endif;

{warten auf ok Beispiele}
while not I2Cstat(...) do
  ...;
  ...;
endwhile;

repeat
   ...;
   ...;
until I2Cstat(...);
```

3.30 I2Cexpand Treiber für bis zu 8 bidirektionale Ports

Allgemeines
Bei manchen Steuerungs Anwendungen reichen die bei einer bestimmten CPU zur Verfügung stehenden IOs bzw. Port Pins nicht aus, vor allem wenn zwei Ports komplett durch die externe Speicheransteuerung wegfallen.
Hier hilft nur eine noch grössere CPU oder die Erweiterung der möglichen Port Bits durch zusätzliche Hard- und Software. Hierbei gibt es mehrere Möglichkeiten eine solche Erweiterung zu implementieren. Echte IO-Chips wie z.B. 8255 z.B. wegen Platz oder Ansteuer Gründen nicht verwendet werden, ausserdem bieten diese max. 20 zusätzliche Bits. Standard Latches z.B. müssen ebenfalls parallel angesteuert werden und benötigen dadurch doch erheblich Port Bits der CPU, so dass die Einsparung and CPU Resourcen doch nicht so gross ist.
Wenn es nicht auf allzu grosse Geschwindigkeit bei der Port Bearbeitung ankommt, und die Anzahl der notwendigen zusätzlichen gewünschten Bits erheblich ist, kommt neben dem Schieberegister Vefahren auch das I2C Bussystem in Betracht.
Für den I2C Bus (TWI) gibt es eine ganze Anzahl von frei programmierbaren remote IO-Ports. Am besten für diesen Zweck geeignet ist der Philips Baustein PCA9554 bzw. PCA9554A. Dieser enthält ein PORT-Register, ein PIN-Register und ein DDR-Register und ist damit vollkommen gleichwertig zu einem AVR Port. Es können bis zu 8 solche Bausteine an den Bus angeschlossen werden.

Einführung I2Cexpander
Die vorliegende Implementation benutzt entweder den Software I2C-Treiber (I2Cport) oder den internen TWI (I2C) Port der AVR mega CPUs. Dazu ist entweder der Treiber ***I2Cport*** oder der Treiber ***TWImaster*** oder der Treiber ***TWInet*** im Mastermode zu importieren. Bei den XMegas muss einer dessen TWIs benutzt bzw. importiert werden: TWI_C, TWI_D, TWI_E or TWI_F.

Als I2C Port-Expander muss pro Port der Typ PCA9554A von Philips verwendet werden. Dieser Baustein kann bis zu 8 mal am Bus vorhanden sein. Der PCA9554A kann mit bis zu 400kBit/sec am I2C Bus betrieben werden. Im Gegensatz zu seinen Vorgänger Typen kann jedes Port gezielt gelesen, geschrieben und umprogrammiert werden.

Die Basis Adresse des PCA9554A ist $38..$3F. Alternativ zum PCA9554A kann auch der PCA9554 eingesetzt werden, dessen Basis Adresse auf $20..$27 liegt. Diese Adressen sind im AVRco System für 16bit I2C-Bausteine des Typs PCA9555 reserviert. Deshalb sollte der PCA9554 nur in Ausnahmefällen eingesetzt werden.

Die möglichen Ports haben die Namen PORT0…PORT7, PIN0…PIN7, DDR0…DDR7
Hierbei hat PORT0/PIN0/DDR0 die I2C-Adresse $38, PORT1/PIN1/DDR1 die I2C Adresse $39 etc.

Als Besonderheit haben diese I2C Chips die Möglichkeit Input Pins zu invertieren. Dazu werden die Spezial Ports INP_POL0..INP_POL7 exportiert. Eine log1 invertiert das zugehörige Input Bit.

3.30.1 Technische Daten

I2C Port	Software I2C importiert durch I2Cport	
oder	CPU-TWI importiert durch TWImaster	// Mega
oder	CPU-TWI importiert durch TWInet im Mastermode	// Mega
oder	CPU-TWI importiert durch TWI_C, TWI_D, TWI_E oder TWI_F	// XMega
Hardware	I2C I/O-Expander Chip PCA9554A von Philips, 1 Stück pro Port	
I2C Adressen	Die PCA9554A liegen auf den Bus-Adressen $38..$3F wobei PORT0 die Adresse $38 hat, PORT1 hat $39 etc.	

Die PCA9554 liegen auf den Bus-Adressen $20..$27, wobei PORT0 die Adresse $20 hat, PORT1 hat $21 etc.

Die PCA95554(A) haben drei Adresspins bzw. Bits die jeweils entsprechend beschaltet werden müssen.

Imports
Wie beim AVRco System üblich, muss der Treiber importiert und definiert werden. Zusätzlich muss aber auch noch der gewünschte I2C/TWI Treiber importiert werden.
Der SysTick wird nicht benötigt.

```
Import  I2Cport, I2Cexpand;
oder
Import  TWImaster, I2Cexpand;
oder
Import  TWInet, I2Cexpand;          // use Master mode
XMega
Import  TWI_C, I2Cexpand;           // use TWI_C
```

Defines
Je nach gewünschtem I2C bzw. TWIport muss dieses definiert werden.

Beispiel für *I2Cport*:
```
Define  ProcClock     = 8000000;        {8Mhz clock }
        I2Cport       = PortC;          {port used}
        I2Cdat        = 7;              {bit7-PortC}
        I2Cclk        = 6, 4;           {bit6-PortC, optional delay 4}
        I2Cexpand     = I2C_Soft, $38;  {use Software I2Cport, 9554A}
        I2CexpPorts   = Port0, Port4;   {use Port0 and Port4}
```

Beispiel für *TWImaster*:
```
Define  ProcClock     = 8000000;        {8Mhz clock }
        TWIpresc      = TWI_BR100;      {100kBit/sec alt. TWI_BR400}
        I2Cexpand     = I2C_TWI, $38;   {use TWIport, 9554A}
        I2CexpPorts   = Port1, Port2;   {use Port1 and Port2}
```

Beispiel für *TWInetMaster*:
```
Define  ProcClock     = 8000000;        {8Mhz clock }
        TWInode       = 05;             {default address in slave mode}
        TWIpresc      = TWI_BR400;      {400kBit/sec alt. TWI_BR100}
        TWIframe      = 4, iData;       {buffer/packet size}
        TWIframeBC    = 6;              {option broadcast buffer/packet size}
        TWInetMode    = Master;
        I2Cexpand     = I2C_TWI, $20;   {use TWIport, 9554}
        I2CexpPorts   = Port7;          {use Port7}
```

Beispiel für *XMega*:
```
Import  TWI_C, I2Cexpand;           // use TWI_C, TWI_D, TWI_E or TWI_F
Define
        OSCtype       = int32MHz, PLLmul=4, prescB=1, prescC=1;
        TWIpresc      = TWI_BR100;      {100kBit/sec alt. TWI_BR400}
        I2Cexpand     = TWI_C, $38;     {use TWIportC, 9554A}
        I2CexpPorts   = Port1, Port2;   {use Port1 and Port2}
```

I2Cexpand
Definiert das zu verwendende I2C-Port., entweder SoftWare-I2C mit ***I2C_Soft*** oder onchip TWIport mit ***I2C_TWI***. Der jeweilige Treiber dazu muss importiert und definiert werden.
Der zweite Parameter bestimmt das I2C-chip: $20 beim PCA9554 oder $38 beim PCA9554A

I2CexpPorts
Definiert welche und wieviel Ports unterstützt werden sollen. Zulässig sind die Angaben von Port0 ... Port7

3.30.2 Typen und Funktionen

Der Import von I2Cexpand veröffentlicht eine spezial Type:

```
Type TI2CPORT = internal;
```

Dieser Typ kann dazu benutzt werden, um den einzelnen Ports eine aussage kräftigeren Namen zu geben.

```
Var myName[@Port1] : TI2Cport;
```

Mit *myName* kann jetzt das Port1 angesprochen werden.

I2CexpStat
Bei Power-Up ist es sinnvoll festzustellen ob alle Ports auch wirklich reagieren. Dazu kann der I2C-Status eines Ports abgefragt werden.

```
Function I2CexpStat (Port: TI2Cport) : boolean;
```

Diese Funktion gibt ein true zurück, wenn das Selektieren des PCA9554 Chips erfolgreich war.

Im Programm Verlauf kann ein solches Port wie ein normales Port des AVRs behandelt werden, allerdings mit Einschränkungen. Das Port kann nicht durch Pointer adressiert werden, es darf nicht Teil eines Konstrukts sein, wie z.B. Array oder Record. Es kann nicht Prozedur-lokal sein und kann auch nicht als Übergabe Parameter für Prozeduren/Funktionen verwendet werden. Die einzige zulässige spezial Operation ist *inc(port)* und *dec(port)*.

Die bitweise Adressierung der Port-Bits ist zulässig. Dazu müssen Bits als Overlays über ein Port gelegt werden.

```
Var iBit1[@Port2, 1] : bit;
```

Damit können die einzelnen Bits eines Port gezielt angesprochen werden:

```
if not iBit1 then
  iBit1:= true;
  iBit1:= false;
  toggle (iBit1);
endif;
```

Das gleiche bzw. ähnliches gilt übrigens auch für die Standard AVR Ports.

3.30.3 Multi-Processing und TWI Port

In einer Applikation mit Prozessen und/oder Tasks kommt es häufig vor, dass der TWI-Bus nicht nur als Netzwerk sondern auch für andere Zwecke (LCD, Ports etc) gebraucht wird. Wenn dabei die Zugriffe auf das TWI aus unterschiedlichen Prozessen heraus erfolgen, kommt es unweigerlich zu Konflikten da solche sequenziellen Treiber (I2C, TWI, UART etc) nicht re-entrant sind, d.h. sie sind nicht unterbrechbar und neu aufrufbar. Deshalb besitzt das TWI Port ganz allgemein eine Semaphore vom Typ DeviceLock.

```
TWI_DevLock : DEVICELOCK;
TWI_DevLockTN  : DEVICELOCK;      // XMega TN = C, D, E or F
```

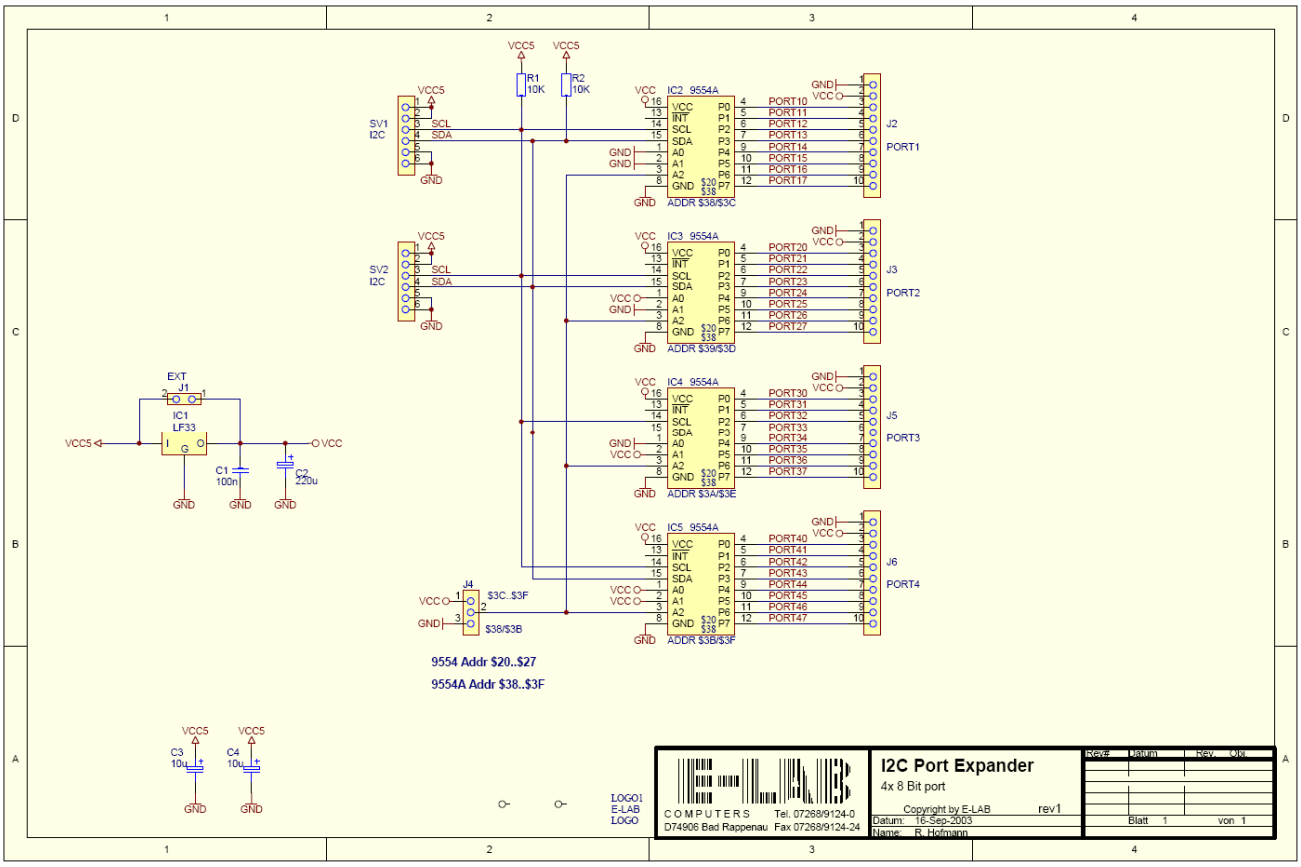
Der TWI Treiber beachtet und steuert diese Semaphore. Beim Eintritt in den Treiber wird geprüft ob der Treiber frei ist (Semaphore inaktiv). Ist das der Fall, dann wird die Semaphore aktiviert = Treiber gesperrt und der Auftrag wird ausgeführt. Nach dem Beenden des Auftrags wird die Semaphore wieder freigegeben.

Wird beim Eintritt festgestellt, dass der Treiber belegt ist (Semaphore aktiv) dann wird ein ***Schedule*** durchgeführt und dieser Prozess reiht sich wieder in die Warteschlange ein. Bei einem der nächsten Prozess Wechsel wird der Prozess wieder gestartet und prüft jetzt erneut die Semaphore. Das wiederholt sich jetzt solange, bis die Semaphore wieder frei ist.

Achtung:
Durch den Abbruch eines TWI-Aufrufs durch „Schedule“ sollten für den TWI Zugriff ***keine Tasks*** verwendet werden, da Tasks damit komplett abgebrochen werden und der Auftrag in diesem Fall nicht ausgeführt wird. Mann kann sich jedoch ein Flag setzen, wenn der Task mit seinem Auftrag erfolgreich war und wenn nicht, muss der Task wiederholt werden. Das bringt jedoch eine etwas komplizierte Verwaltung mit sich.

Programm Beispiel und Schaltung:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\ I2Cexpand**
ein Xmega Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\ XMega_I2Cexpand**



Schaltplan **I2Cexpand**

3.31 Impuls Zähler Treiber PulseCount

Nicht für XMegas
Einfache und langsame Ereignis/Impuls Zähler kann man durch kontinierliches beobachten eines Port Pins z.B. in einem Task aufbauen. Kommen diese externen Ereignisse in schneller Folge so macht es Sinn dazu einen Interrupt Pin zu benutzen und im Interrupt die Pulse zu zählen. Bei hoher Impulsfolge oder häufiger Interrupt Sperrung besteht jedoch einerseits die Gefahr das System durch diese Interrupts „dicht“ zu machen oder sogar Impulse zu verlieren.

Im allgemeinen aber ganz im besonderen für hohe Impuls Raten bieten sich interne 16bit Counter für diese Aufgabe an. Der Vorteil dabei ist dass diese Zähler vollkommen autark laufen und das System nur minimal mit Interrupts belasten. Die meisten der 16bit Counter haben einen externen Clock Input Pin. An diesen wird die Impuls Quelle angeschlossen. Die Applikation muss dabei sicher stellen dass der zugehörige Port Pin immer auf Input steht. Zur korrekten Funktion des Treibers muss der globale Interrupt freigegeben sein.

Der Counter zählt mit jeder high Flanke am Clock Pin um eins weiter. Kommt ein Überlauf des 16bit Counters zustande, springt der Counter auf 0 und löst einen Overflow Interrupt aus. In diesem Interrupt wird dann eine 16bit Variable um 1 inkrementiert. Beim Lesen des PulseCount Stands wird deshalb ein Longword (32bit) zurückgegeben, das aus dem Counter Inhalt und der Variable besteht. Es können 2 Counter definiert werden.

Imports
Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import  SysTick, PulseCount, { PulseCount2} ..;
```

Der notwendige 16bit Timer/Counter wird mit Define vorgegeben:

Defines

```
Define
    ProcClock    = 8000000;        {Hertz}
    SysTick      = 10;             {msec}
    StackSize    = $0010, iData;
    FrameSize    = $0010, iData;
    PulseCount   = Timer1;         {or Timer3..5 if present}
    // PulseCount2 = Timer3;       {or Timer3..5 if present}
```

3.31.1 Funktionen

Es werden 4 verschiedene Funktionen pro Kanal zur Verfügung gestellt:

```
Function GetPulseCount : longword;
Function GetPulseCount2 : longword;
```
Diese Funktion liefert den aktuellen internen Zählerstand. Der interne Wert wird nicht verändert und der Counter wird nicht angehallten. Ein kontinuierliches Pollen sollte vermieden werden um durch die implizite Interrupt Sperrung das System nicht lahmzulegen.

```
Procedure PulseCountStart;
Procedure PulseCountStart2;
```
Diese Prozedur startet den Zähler. Notwendig beim Programmstart, Stop oder Clear.

```
Procedure PulseCountStop;
Procedure PulseCountStop2;
```
Diese Prozedur stopt den Zählvorgang ohne den internen Zähler zu verändern.

```
Procedure PulseCountClear;
Procedure PulseCountClear2;
```
Diese Prozedur stoppt den internen Zähler und setzt ihn auf 0.

3.32 Impuls Zähler Treiber PulseCount XMega

Im allgemeinen aber ganz im besonderen für hohe Impuls Raten bieten sich interne 16bit Counter für das Zählen externer Ereignisse an. Der Vorteil dabei ist dass diese Zähler vollkommen autark laufen und das System nur minimal mit Interrupts belasten. Alle der bis zu acht 16bit Counter haben einen externen Clock Input Pin. An diesen wird die Impuls Quelle angeschlossen. Zur korrekten Funktion des Treibers muss der globale Interrupt freigegeben sein.

Der Counter zählt mit jeder low oder high Flanke am Clock Pin um eins weiter. Kommt ein Überlauf des 16bit Counters zustande, springt der Counter auf 0 und löst einen Overflow Interrupt aus. In diesem Interrupt wird dann eine 16bit Variable um 1 inkrementiert. Beim Lesen des PulseCount Werts wird deshalb ein Longword (32bit) zurückgegeben, das aus dem Counter Inhalt und der Variable besteht. Es können bis zu 8 Counter definiert werden.

Imports

```
Import  SysTick, PulseCount_C0, { PulseCount_C1, PulseCount_D0} ..; // C0..F1 counters
```

Defines

```
Define
  // The XMegas don't provide any Oscillator fuses.
  // So the application must setup the desired values
  // possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz

  //>> CPU       = 32MHz, PeripherX4=32MHz, PeripherX2=32MHz
        OSCtype             = int32MHz,
        PLLmul              = 4,
        prescB              = 1,
        prescC              = 1;
        SysTick             = 10;          {msec}
        StackSize           = $0060, iData;
        FrameSize           = $0080, iData;
        PulseCount_C0       = PortA, 0, PullUp;     // clock input Port, Pin, Pullup/PullDown/None
        PulseCountEv_C0 = 0;                        // event channel to be used
```

Jedes der Ports A..F und jeder Pin dieser Ports kann verwendet werden. Dabei muss einer der 8 Event Channel benutzt werden.

3.32.1 Funktionen

Der Treiber bietet 4 Funktionen per Kanal wobei ***XX*** für den importierten Timer steht, C0..F1:

Function *PulseCountRead_XX : longword;*
Diese Funktion liefert den aktuellen internen Zählerstand. Der interne Wert wird nicht verändert und der Counter wird nicht angehallten.

Procedure *PulseCountStart_XX;*
Diese Prozedur startet den Zähler. Notwendig beim Programmstart, Stop oder Clear.

Procedure *PulseCountStop_XX;*
Diese Prozedur stopt den Zählvorgang ohne den internen Zähler zu verändern.

Procedure *PulseCountClear_XX;*
Diese Prozedur stopt den internen Zähler und setzt ihn auf 0.

Beispiel Programm:
Ein Beispiel Programm ist in der Directory **..\E-Lab\AVRco\Demos\XMega_PulseCount**

3.33 Inkremental Encoder Treiber IncrPort

Nicht für XMegas, anstatt dessen QDEC verwenden

Positionen, Stellvorgänge oder Drehrichtungen werden in der Elektronik oft durch sog. Inkremental Encoder überwacht. Das sind mechanische oder optoelektronische Bauelemente, die zwei (drei) digitale um 90(120)grad versetzte Kurvenzüge ausgeben. Anhand dieser zwei (drei) Signale kann einerseits die Drehrichtung festgestellt werden und andererseits durch Zählen der Flanken auch eine relative Position. Besteht auch eine Null-Punkt Erkennung, so lässt sich die Position auch absolut ermitteln.

Die vorliegende Implementation setzt ein (zwei) XOR-Gatter voraus, das die beiden (drei) Signale verarbeitet und einen Eingang des **Analog Comparators** des AVR speist. Der andere Eingang des ACOMP wird auf ca. 50% VCC gehalten. Diese Schaltung gestattet es dass jede Flanke der beiden (drei) Signale einen Interrupt auslöst. Damit ist eine Vervierfachung des Eingangssignals sichergestellt (Quadratur). Mit diesem Signal allein lässt sich schon eine sehr schnelle Impuls Zählung erreichen. Eine Vor-Rückwärts Erkennung ist dadurch allerdings noch nicht möglich.

Die Drehrichtung des Gebers wird durch die Phasenlage der beiden (drei) Signale festgestellt. Dazu müssen diese an ein beliebiges **Eingangsport** des AVRs angeschlossen werden. Bedingung ist, dass alle Signale am selben Port anliegen, ohne Lücken dazwischen . Die Port Pins ansonsten können beliebig sein.

Imports

Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import  SysTick, IncrPort, ..;
```

Die Auflösung (16 oder 32bit, 2 oder 3Phasen) und der gewünschte Port und die beiden (drei) PortPins werden mit Define vorgegeben:

Defines

```
Define
    ProcClock    = 8000000;        {Hertz}
    SysTick      = 10;             {msec}
    StackSize    = $0010, iData;
    FrameSize    = $0010, iData;
    IncrCounter  = 16, 2;          {16 bit integer, 2 Phasen}
    IncrPort     = PinD, $C0;      {PinD, Portpin 6 + 7}
```

3.33.1 Funktionen

Es werden 4 verschiedene Funktionen zur Verfügung gestellt:

```
Function GetIncrementVal : integer [longint];
```

Diese Funktion liefert den aktuellen internen Zählerstand als Integer, wenn ein 16bit Zähler definiert wurde. Das Ergebnis ist ein longint, wenn ein 32bit Zähler gewählt wurde. Der interne Wert wird nicht verändert.

```
Function GetIncrementRel : integer [longint];
```

Diese Funktion liefert den aktuellen internen relativen Zählerstand als Integer, wenn ein 16bit Zähler definiert wurde. Das Ergebnis ist ein longint, wenn ein 32bit Zähler gewählt wurde. Der Wert ist relativ zum letzten Aufruf dieser Funktion. War z.B. seither kein Zählimpuls, ist das Ergebnis 0. Wird diese Funktion in konstanten Abständen aufgerufen, lässt sich damit auch eine Geschwindigkeit bzw. Drehzahl ableiten.

```
Procedure ClearIncrementVal;
```

Diese Prozedur setzt den absoluten und relativen internen Zähler auf null.

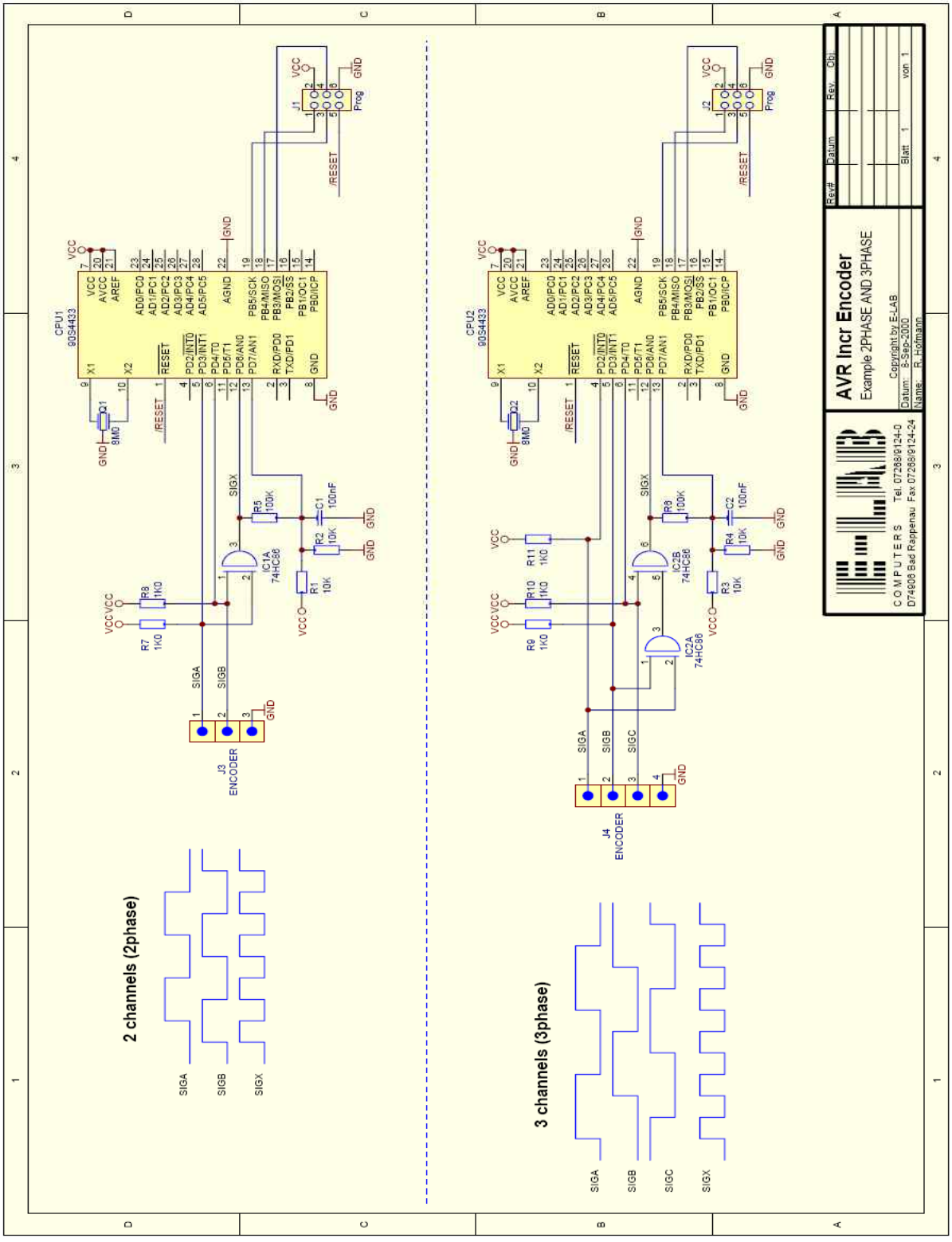
```
Procedure SetIncrementVal (val : integer [longint]);
```

Diese Prozedur setzt den absoluten internen Zähler auf den Wert „val“.

Bemerkung:
Dieser Treiber muss im Interrupt laufen. Wird der Interrupt gesperrt so ist mit falschen Ergebnissen zu rechnen.

Beispiel und Schaltung:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Increment**



Schaltplan **Increment**

3.34 Inkremental Encoder Treiber IncrPort4

Der weiter oben beschriebene Treiber benötigt den Analog Comparator und unterstützt deswegen nur einen Geber. Eine universellere Implementation ist der *IncrPort4* Treiber, der bis zu 4 Geber lesen kann und als Resource nur einen Timer und ein 8bit Port benötigt. Eine externe Logik wird nicht gebraucht. Es wird aber keine Nullpunkt Erkennung unterstützt.

Diese Implementation benutzt ein beliebiges Input-Port, an dem bis zu 4stk 2-Phasen Geber angeschlossen werden können. Alle Geber müssen an das gleiche Port angeschlossen werden. Geber-0 an Portx.0 und Portx.1, Geber-1 an Portx.2 und Portx.3 etcetc. Die verwendeten Portpins müssen auf Input stehen. Die Vervierfachung des Eingangssignals (Quadratur) ist immer aktiv.

Unbenutzte Pins können beliebig verwendet werden.

Der Treiber benutzt einen zyklischen Timer Interrupt um die Kanäle zu scannen und auszuwerten. Um sichere Ergebnisse zu erhalten muss die Scan Rate mindestens um Faktor 4 höher liegen als die max. zu erwartende Impuls Rate der Geber. Eine zu hoch gewählte Scanrate kann u.U. das System durch die Timer Interrupts „dicht“ machen. Ein kontinuierliches Pollen der Kanäle durch die Applikation kann zu Problemen mit dem Interrupt System führen, da jeder Lesezugriff auch immer eine Interrupt Sperrung beinhaltet.

Imports
Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import  SysTick, IncrPort4, ..;
```

Die Auflösung (16 oder 32bit), der gewünschte Port (PIN), die Anzahl der Encoder, der Timer und die Scanrate werden mit Define vorgegeben:

Defines

```
Define
    ProcClock   = 16000000;         {Hertz}
    StackSize   = $0020, iData;
    FrameSize   = $0040, iData;
    IncrPort4   = PinA, 2, 32;      // pin-reg used, channels, 16 or 32bit integer
    IncrScan4   = Timer3, 10;       // timer used, scan rate 10kHz (1..100)
```

3.34.1 Funktionen

Es werden 7 verschiedene Funktionen zur Verfügung gestellt. Der Parameter *chan* zählt von 0 an (0..3).

***Procedure** IncrCount4start;*
Nach einem Reset oder PowerOn läuft der Scan Timer noch nicht. Er muss mit dieser Prozedur gestartet werden. Diese Prozedur verändert die Zählerstände nicht.

***Procedure** IncrCount4stop;*
Der Scan Timer wird hiermit gestoppt und es werden keine Interrupts mehr generiert. Die Zählerstände werden nicht verändert.

***Function** GetIncrVal4 (chan : byte) : integer [longint];*
Diese Funktion liefert den aktuellen internen Zählerstand von *chan* als Integer, wenn ein 16bit Zähler definiert wurde. Das Ergebnis ist ein longint, wenn ein 32bit Zähler gewählt wurde. Der interne Wert wird nicht verändert.

Function *GetIncrRel4 (chan : byte) : integer [longint];*
Diese Funktion liefert den aktuellen internen relativen Zählerstand von *chan* als Integer, wenn ein 16bit Zähler definiert wurde. Das Ergebnis ist ein longint, wenn ein 32bit Zähler gewählt wurde. Der Wert ist relativ zum letzten Aufruf dieser Funktion. War z.B. seither kein Zählimpuls, ist das Ergebnis 0. Wird diese Funktion in konstanten Abständen aufgerufen, lässt sich damit auch eine Geschwindigkeit bzw. Drehzahl ableiten.

Procedure *ClearIncrVal4 (chan : byte);*
Diese Prozedur setzt den absoluten und relativen internen Zähler von *chan* auf null.

Procedure *ClearIncrAll4;*
Diese Prozedur setzt alle absoluten und relativen internen Zähler auf null.

Procedure *SetIncrVal4 (chan : byte; val : integer [longint]);*
Diese Prozedur setzt den absoluten internen Zähler *chan* auf den Wert „val“.

Bemerkungen:
Dieser Treiber muss im Interrupt laufen. Wird der Interrupt zu lange gesperrt so ist mit falschen Ergebnissen zu rechnen.

Wenn der Parameter *chan* von einer Funktion erwartet wird, wird dieser immer mit der Vorgabe durch das Define *IncrPort4* verglichen. Wird ein falscher Kanal Wert übergeben führt die Funktion nichts aus und kehrt ggf. mit dem Wert 0 zurück. Der Kanal Parameter *chan* zählt von 0 an.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Increment4**

3.35 UP/DOWN Counter Treiber XMega

Eine vereinfachte Methode um mechanische Positionen zu erfassen ist ein simpler Up-Down Counter. Im Gegensatz zum Inkremental Geber/Counter gibt es hier einen Puls Eingang und einen Richtungs Eingang.

Die vorliegende Implementation benutzt einen der bis zu 8 XMega Timer. Die zwei benötigten Eingänge (Clock und DIR) können dabei in beliebigen Ports liegen (PortA..PortF). Bei einer Auflösung von 16bit (Integer) wird nur einer der 8 Event Channels benötigt. Bei 32bit (LongInt) wird zusätzlich ein Interrupt generiert.
Bedingung ist, dass alle Signale am selben Port anliegen. Die Port Pins ansonsten können beliebig sein.

Imports
Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import  SysTick, UPDWN_C0, ..;        // UPDWN _C1, UPDWN _D0, UPDWN _D1, UPDWN _E0
```

Wobei C0, C1, D0, D1 etc. den gewünschten Timer auswählen. Das "*XX*" steht für den gewählten Timer.
Die Auflösung (16 oder 32bit), der gewünschte Port, die zwei PortPins und der Event Kanal werden definiert:

Defines

```
Define
  // The XMegas don't provide any Oscillator fuses.
  // So the application must setup the desired values
  // possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz

  //>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
  OSCtype           =  int32MHz,
                       PLLmul=4,
                       prescB=1,
                       prescC=1;
  SysTick           = 10, adj;        // msec, correct the RTC32K timer for exact mSec timing
  StackSize         = $0064, iData;
  FrameSize         = $0064, iData;

UPDwnClkDir_C0    = PortD, 0, 1;     // Port, clock, dir input pin
UpDwnRes_C0       = 16;              // Resolution UPDWN_C0, 16 or 32
UpDwnEvChan_C0    = 0;               // 0..6
```

Im folgenden werden die Defines, Funktionen und Prozeduren mit *XX* gekennzeichnet. *XX* steht dabei für den jeweiligen QDEC bzw. Timer Kanal (C0..F1).

UPDwnClkDir _XX	definiert das zu verwendende Port (PortA..PortF) und dessen zwei Pins.
UpDwnRes _XX	definiert die Auflösung, 16 oder 32 bits.
UpDwnEvChan_XX	definiert den notwendigen Event Channel, 0..6. Es werden zwei aufeinander folgende Channel benutzt!

3.35.1 Funktionen

Es werden 4 verschiedene Funktionen zur Verfügung gestellt:

Procedure *UPDWNenable_XX(ena : boolean);*
Diese Prozedur startet bzw. stoppt den internen Zähler. Der aktuelle Zähler Stand wird nicht verändert.

Function *UPDWNgetPos_XX : integer [longint];*
Diese Funktion liefert den aktuellen internen Zählerstand als Integer, wenn ein 16bit Zähler definiert wurde. Das Ergebnis ist ein longint, wenn ein 32bit Zähler gewählt wurde. Der interne Wert wird nicht verändert.

Procedure *UPDWNclearpos_XX;*
Diese Prozedur setzt den internen Zähler auf null. Der Zähler wird angehalten.

Procedure *UPDWNsetPos_XX(val : integer [longint]);*
Diese Prozedur setzt den internen Zähler auf den Wert „val“. Der Zähler wird angehalten.

Die Drehrichtung und die Änderungs Geschwindigkeit kann ermittelt werden, wenn der aktuelle Zählerstand mit dem vorherigen verglichen wird. Das muss allerdings in einem festen Zeit Intervall geschehen.

Bemerkung:
Der 32bit Treiber muss im Interrupt laufen. Wird der Interrupt zu lange gesperrt so ist mit falschen Ergebnissen zu rechnen.

Beispiel:
ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_UpDownCount**

3.36 QDEC Inkremental Encoder Treiber XMega

Positionen, Stellvorgänge oder Drehrichtungen werden in der Elektronik oft durch sog. Inkremental Encoder überwacht. Das sind mechanische oder optoelektronische Bauelemente, die zwei digitale um 90grad versetzte Kurvenzüge ausgeben. Anhand dieser zwei Signale kann einerseits die Drehrichtung festgestellt werden und andererseits durch Zählen der Flanken auch eine relative Position.

Die vorliegende Implementation benutzt einen (zwei) der bis zu 8 XMega Timer. Die zwei benötigten Phasen Eingänge können dabei in beliebigen Ports liegen (PortA..PortF). Bei einer Auflösung von 16bit (Integer) werden zwei der 8 Event Channels benötigt. Bei 32bit (LongInt) wird ein weiterer Timer und weitere zwei Event Channels benötigt.
Der Treiber arbeitet mit Vervierfachung (Quadratur). Bedingung ist, dass alle Signale am selben Port anliegen, ohne Lücken dazwischen. Die Port Pins ansonsten können beliebig sein.

Imports
Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import  SysTick, QDEC_C0,  QDEC32_F1, ..;        // QDEC_C1, QDEC_D0, QDEC_D1, QDEC_E0
```

Wobei C0, C1, D0, D1 etc. den gewünschten Timer auswählen. Der Import *QDEC_xx* oder *QDEC32_xx* bestimmt die Auflösung, 16 oder 32 bits.
Der gewünschte Port, die zwei PortPins und die Event Kanäle werden definiert:

Defines

```
Define
  // The XMegas don't provide any Oscillator fuses.
  // So the application must setup the desired values
  // possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz

  //>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
  OSCtype          =  int32MHz,
                      PLLmul=4,
                      prescB=1,
                      prescC=1;
  SysTick          = 10;              // msec
  StackSize        = $0064, iData;
  FrameSize        = $0064, iData;

// 16 bit resolution
   QDECphase_C0    = PortD, 0, 1;      // Port, Phase0, Phase90 input pin
  QDECevChan_C0    = 0;                // 0, 2, 4 or 6

// 32 bit resolution
   QDECphase_F1    = PortE, 4, 5;      // Port, Phase0, Phase90 input pin
   QDECevChan_F1   = 2;                // event channel 0, 2, 4 or 6
   QDECcntHi_F1    = TCC1, 5, 6;       // Timer, two event channels
```

Im folgenden werden die Defines, Funktionen und Prozeduren mit *XX* gekennzeichnet. *XX* steht dabei für den jeweiligen QDEC bzw. Timer Kanal (C0..F1).

QDECphase_XX	definiert das zu verwendende Port (PortA..PortF) und dessen zwei Pins.
QDECevChan_XX	definiert den notwendigen Event Channel, 0, 2, 4 oder 6
QDECcntHi_XX	32bits, definiert den notwendigen zweiten Timer und zwei Event Channels

3.36.1 Funktionen

Es werden 4 verschiedene Funktionen zur Verfügung gestellt:

Procedure *QDECenable_XX(ena : boolean);*
Diese Prozedur startet bzw. stoppt den internen Zähler. Der aktuelle Zähler Stand wird nicht verändert.

Function *QDECgetPos_XX : integer [longint];*
Diese Funktion liefert den aktuellen internen Zählerstand als Integer, wenn ein 16bit Zähler definiert wurde. Das Ergebnis ist ein longint, wenn ein 32bit Zähler gewählt wurde. Der interne Wert wird nicht verändert.

Procedure *QDECclearpos_XX;*
Diese Prozedur setzt den internen Zähler auf null.

Procedure *QDECsetPos_XX(val : integer [longint]);*
Diese Prozedur setzt den internen Zähler auf den Wert „val“.

Die Drehrichtung und die Änderungs Geschwindigkeit kann ermittelt werden, wenn der aktuelle Zählerstand mit dem vorherigen verglichen wird. Das muss allerdings in einem festen Zeit Intervall geschehen.

Bemerkung:
Die Event Channels des Basis Timers müssen auf geradzahligen Adressen beginnen. Es werden zwei Kanäle belegt. Die Event Channels des weiteren (32bit) Timers sollten aufeinander folgen.

Beispiel:
ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_QDEC**

3.37 Stepper-Motor

3.37.1 Prinzipielles

Mit Schrittmotoren lassen sich recht problemlos präzise und auch schnelle Positionier- und Stellantriebe realisieren. Insgesamt gesehen ist der Schrittmotor der preiswerteste Antrieb dieser Art. Allerdings hat dieser Antrieb auch einen entscheidenden Nachteil: Schrittmotore sind Synchron Motore und diese haben das Problem, dass wenn sie ausser Synchronisation kommen, sie entweder stehen bleiben oder, was auch nicht viel besser ist, dass sie Schritte verlieren. Das bedeutet in der Praxis, dass die Anfahrdrehzahl, die Beschleunigung und die Endfrequenz extrem wichtig sind. Diese Parameter sind vom Motortyp, der Betriebsart (Full/Halfstep etc), von der Art der Endstufe, von der Massenträgheit des Systems sowie ganz allgemein von der Last bzw. Reibungsverluste abhängig.

Es werde nur Schrittmotore mit 2 Wicklungen unterstützt (2-Phasen Motore).

Der Entwickler muss sich über die Vor- und Nachteile der möglichen Ansteuerungen im klaren sein. Tutorials u.ä. über dieses Thema können von den Homepages von Ericsson und ST heruntergeladen werden. Es kann nicht Ziel dieses Manuals sein, einen Schrittmotor Lehrgang zu bieten. Trotzdem eine kurze Erläuterung:

3.37.1.1 Konstantspannungs Betrieb

Dies ist die simpelste Art und lässt sich mit 4 Transistoren bewältigen. Der Motor muss jedoch dafür ausgelegt sein (hochohmig). Als Betriebsart sind nur Voll- und Halbschritt möglich. Die Anfahr Steprate ist relativ niedrig und ebenso die zu erreichende Endfrequenz. Motor und Systemresonanzen bilden ein grosses Problem.

3.37.1.2 Konstantstrom Betrieb

Diese Betriebsart ist aufwendiger und teuerer, bietet jedoch, je höher die Schrittauflösung ist (z.B. Sinus Strom Modus mit “StepMicro8”) eine enorme Auflösung (0.2Winkelgrad) und fast keine Resonanzprobleme mehr. Ausserdem läuft der Motor fast so rund wie ein Gleichstrom Motor, auch bei langsamen Drehzahlen. Ein Nachteil ist, dass die Software und auch die Hardware mit der 6..8fachen Geschwindigkeit (StepMini6, StepMicro8) laufen muss. Weiterhin muss die benötigte Beschleunigungstabelle im ROM Tab auch die um diesen Faktor höhere Endfrequenz abdecken.

Das bedeutet, dass die Lookup Tabelle statt 1..2kByte (Full/Halfstep) hier 8..10kByte gross werden kann. Bei 10Khz Steprate läuft der verwendete 16bit Timer mit 100usec Interrupt. Bei einem AVR mit 8Mhz werden ca. 30usec für die Berechnung verbraucht. Das sind ca. 30% der Rechenleistung und auch eine ebenso lange Zeit, während der andere Interrupts gesperrt bleiben. Sind weitere Interrupts implementiert, so sollte deren Rechenzeit möglichst klein sein, um ein Frequenz-Jitter und damit evtl. Schrittverlust zu vermeiden.

3.37.1.3 Einschränkungen

Aus den obigen Ausführungen ist zu ersehen, dass durch den ROM Bedarf (Lookup Table) und den Rechenzeit Bedarf (Interrupt) die CPU keine grösseren zusätzlichen Aufgaben erledigen kann, **denn:**
Je feiner die Schrittauflösung, desto höher die notwendige Steprate, desto grösser die Lookup Table, desto grösser der Rechenzeit Bedarf.

Beim AVR sollte die CPU mit 8MHz laufen, damit die Parameter Werte bzw. Frequenzen auch in der Praxis stimmen. Hier wird durch den fest eingestellten Vorteiler (div 8) vom benutzten **Timer1** eine interne Auflösung von 1usec erzielt, was die Basis für alle weiteren Berechnungen bildet

.

3.37.2 Beschleunigungsrampe

Ein Schrittmotor muss, um keine Schritte zu verlieren oder gar stehen zu bleiben, mit möglichst geradliniger Beschleunigung (Hz/sec) hochgefahren werden (Rampe). Diese Rampe kann entweder in Echtzeit gerechnet werden oder über eine Lookup Table abgefragt werden. Eine Echtzeit Berechnung ist erst ab einer Rechenleistung >> 20Mips sinnvoll. Die hier verwendete Lookup Table hat den Nachteil wesentlich grösserer Speicherkosten im ROM, ist aber unabdingbar wenn mit beliebigen Beschleunigungen gearbeitet werden soll und der Prozessor nicht im Highend Bereich angesiedelt ist.

Eine Standard Lookup Table fängt normalerweise mit 100Hz (**StepMinFreg**) an, da dieser Wert mit allen Motoren und Betriebsarten erreicht werden kann. Das Ende der Tabelle (**StepMaxFreg**) wird fast ausschliesslich durch die Betriebsart des Motors bestimmt. Im Voll- und Halbschrittbetrieb sind 1kHz möglich. Im Mini- oder Microschritt Betrieb können Werte bis zu 10kHz erreicht werden.

Diese beiden Werte bestimmen die Grösse der Tabelle als auch die im Rampen Betrieb **mögliche** Start- und End Steprate.

3.37.3 Antriebs Modus

Der Antriebs Mode (**StepType**) stellt die Betriebsart des Motors ein (Full, Half, Micro Step etc) als auch die Bauart der Schrittmotor Endstufe. Es sind 8 mögliche Kombinationen vorhanden:

Name	Treiber	Schritte		Port pins	ICs
StepFull 2	StepF2	Full Step	⎍⎍	2	PBL3717..3777, L6219
StepFull 4	StepF4	Full Step	⎍⎍	4	L6203, L6204
StepHalf 4	StepH4	Half Step	⎍⎍	4	PBL3717..3777, L6219
StepHalf 6	StepH6	Half Step	⎍⎍	6	L6203, L6204
StepMini 4	StepM4	Quart Step	Trapez	6	L6219, TEA3738
StepMini 6	StepM6	Hexa Step	Trapez	6	L6219, TEA3738
StepMicro 2	StepM2	8 Micro Steps	Sinus	2	A3955, IMT901
StepMicro 8	StepM8	8 Micro Steps	Sinus	8	PBL3717..3777, L6219, TLE5250
StepS4	StepS4	32 Micro Steps	Sinus	10	LMD 18245

Mehr über Trapez-Betrieb im Siemens Datenblatt TCA3727
Mehr über Sinus-Betrieb im Allegro Datenblatt A3955

StepFull 2
Vollschrittbetrieb mit 2 Phasen Ausgängen

StepFull 4
Vollschrittbetrieb mit 4 Phasen Ausgängen

SteppHalf 4
Halbschrittbetrieb mit 2 Phasen Ausgängen und 2 Enable (low active!)

StepHalf 6
Halbschrittbetrieb mit 4 Phasen Ausgängen und 2 Enable (high active!)

StepMini 4
Viertelschrittbetrieb (Trapez) mit 2 Phasen Ausgängen und 2*2 Binär Wert Ausgängen

StepMini 6
Sechstelschrittbetrieb (Trapez) mit 2 Phasen Ausgängen und 2*2 Binär Wert Ausgängen

StepMicro 2
Ein Puls- und 1 Richtungsausgang für Sinus Betrieb bzw. intelligente Endstufe

StepMicro 8
Sinus Betrieb (8 Microsteps) mit 2 Phasen Ausgängen und 2*3 Binär Wert Ausgängen

StepS4
Sinus Betrieb (32 Microsteps)

3.37.4 Import des Steppers

Um den Stepper Treiber benutzen zu können, muss dieser in der allgemeinen Import Klausel aufgeführt werden (Import **StepPort**;). Da der Treiber intern mit LongWord rechnet, und die Angabe der zu fahrenden Schritte auch als LongWord erfolgt, muss der Typ LongWord importiert werden (From System Import **LongWord**;).

Das zu verwendende Ausgabe **Port** muss im Define-Block angegeben werden. Zum Aufbau der Beschleunigungs Tabelle müssen die Parameter **StepMinFreq** und **StepMaxFreq** angegeben werden. Der anzuwendende Treiber wird mit **StepType** ebenfalls unter Define angegeben.

```
Device = 90S8515;

Import StepPort;

From System Import longword;

Define
    ProcClock     = 8000000;      {Hertz}
    StackSize     = $0030, iData;
    FrameSize     = $0010, iData;
    StepPort      = PortA;
    StepMinFreq   = 100;
    StepMaxFreq   = 5000;
    StepType      = StepM6;
```

XMega
Bei den XMegas muss auch noch ein Timer_C0…Timer_F1 definiert werden, soweit vorhanden.

```
    StepTimer     = Timer_C0;              // use Timer_C0
```

3.37.5 Parameter des Steppers

Für Beschleunigungs und Bremsrampen werden zur Laufzeit diverse Parameter benötigt, die vom System beim Import des Steppers automatisch veröffentlicht werden. **Achtung:** ein Parameter darf nur im Grundzustand verändert werden!!

```
Var StepStartFreq : word;
```
bezeichnet die Start Steprate beim Anfahren und die Steprate, bei welcher der Bremsvorgang beendet ist. Der Wert dieser Variablen sollte nicht unter dem Tabellenwert “StepMinFreq” liegen und kleiner sein als “StepEndFreq”.

```
Var StepEndFreq : word;
```
bezeichnet die End Steprate bei der die Beschleunigung zu Ende ist und die Steprate, bei welcher der Bremsvorgang eingeleitet wird. Der Wert dieser Variablen sollte nicht über dem Tabellenwert “StepMaxFreq” liegen und grösser sein als “StepStartFreq”.

Var *StepAccValue : word;*
Bezeichnet die Beschleunigung in Hz/sec. Realistische Werte liegen zwischen 1000 und 10000.

Um eine bestimmte Anzahl von Schritten mit Hilfe von Beschleunigungs und Bremsrampen zu fahren (Positionierung), wird zur Laufzeit der zusätzliche Parameter

Var *StepCount : longword; {max 2^32 Steps}*
benötigt. StepCount wird mit dem gewünschten Wert besetzt, um eine bestimmte Anzahl von Schritten mit Beschleunigen und Bremsen (Rampen) fahren zu können. StepCount ist der wesentlichste Parameter für Positionierungen. Nach Beendigung des Vorgangs enthält StepCount den Wert 0.

Wird nicht auf ein Ziel gefahren wird sondern nur eine bestimmte Zeit (StepRampCW oder StepRampCCW), so enthält StepCount die Anzahl der insgesamt mit diesem Befehl gefahrenen Schritte.

Type *TStepMode = (StepStop, StepUp, StepRun, StepDown);*
Var *StepMode : TStepMode;*
Die Variable StepMode enthält immer den aktuellen Status des Treibers. Diese Variable ist zwar auch vom Anwendungsprogramm änderbar, das kann jedoch zu erheblichen Problemen führen. Also nur lesen bitte! Im Ruhezustand enthält StepMode den Wert StepStop. Während der Beschleunigungsphase ist der Wert auf StepUp gesetzt, während der linearen Phase (konstante StepRate) auf StepRun und während der Bremsphase auf StepDown.

Damit ist es der Anwendung z.B. möglich während der Beschleunigungs- und/oder Bremsphase mittels eines weiteren separaten Portpins eine Stromüberhöhung (Boost) einzuschalten.

Da sämtliche Parameter nur im Stillstand verändert werden dürfen, ist **vor** einer **Änderung** die Variable **StepMode** auf **StepStop** zu prüfen. In der Regel brauchen die Parameter nur einmal eingestellt werden und sie bleiben gültig bis zur nächsten Änderung, mit der Ausnahme von StepCount.

3.37.6 Kommandos/Prozeduren des Steppers

Der Stepper Treiber kennt drei logische Betriebsarten mit den zugehörigen Kommandos bzw. Prozeduren. Das sind **SingleStep, Rampe** und **Positionierung**.
Achtung: eine Prozedur darf nur im Grundzustand aufgerufen werden!! Ausnahme bildet das Kommando “StepRampStop” im Rampenmodus.

StepperOn
Schaltet die Endstufe aktiv und legt die zuletzt benutzte Schrittkombination an den Ausgang an. Dieses Kommando kann vor jeder Aktion gegeben werden, muss aber nicht.

StepperOn; {Schaltet Endstufen aktiv}

StepperOff
Schaltet die Endstufe inaktiv bzw. stromlos. Dieses Kommando kann nach jeder Aktion gegeben werden, um eine Überhitzung der Endstufen bzw. des Motors zu vermeiden. Wenn eine Stromabsenkung der Endstufen möglich ist, ist dieser Weg dem “StepperOff” vorzuziehen.

StepperOff; {Schaltet Endstufen inaktiv}

StepOneCW
Ein Schritt im Uhrzeigersinn. Es werden keine Parameter benötigt oder verändert. StepCount wird nicht verändert.

StepOneCW; {Ein Schritt vor}

StepOneCCW
Ein Schritt gegen den Uhrzeigersinn. Es werden keine Parameter benötigt oder verändert. StepCount wird nicht verändert.

StepOneCCW; {Ein Schritt zurück}

StepRampCW
Rampe hochlaufen im Uhrzeigersinn. Es werden die Parameter StepStartFreq, StepEndFreq, und StepAccValue benötigt aber nicht verändert. StepCount wird zuerst auf 0 gesetzt und dann mitgezählt. Die Beschleunigungsrampe wird solange hochgelaufen, bis die maximale Steprate (StepEndFreq) erreicht ist oder der Befehl “StepRampStop” gegeben wurden. Im ersten Fall wird mit der Endfrequenz solange weitergefahren, bis der Befehl “StepRampStop” kommt, dann erfolgt eine entsprechende Abbremsung bis zum Stillstand. Kommt das Kommando “StepRampStop” während der Hochlaufphase, wird umgehend mit der Bremsphase begonnen. Ein StepRampXX benötigt also immer auch ein StepRampStop.

```
StepRampCW;          {bis auf Widerruf laufen}
```

StepRampCCW
Rampe hochlaufen gegen den Uhrzeigersinn. Es werden die Parameter StepStartFreq, StepEndFreq, und StepAccValue benötigt aber nicht verändert. StepCount wird zuerst auf 0 gesetzt und dann mitgezählt. Die Beschleunigungsrampe wird solange hochgelaufen, bis die maximale Steprate (StepEndFreq) erreicht ist oder der Befehl “StepRampStop” gegeben wurden. Im ersten Fall wird mit der Endfrequenz solange weitergefahren, bis der Befehl “StepRampStop” kommt, dann erfolgt eine entsprechende Abbremsung bis zum Stillstand. Kommt das Kommando “StepRampStop” während der Hochlaufphase, wird umgehend mit der Bremsphase begonnen. Ein StepRampXX benötigt also immer auch ein StepRampStop.

```
StepRampCCW;         {bis auf Widerruf laufen}
```

StepRampStop
Ist eines der beiden Rampenkommandos am laufen so kann es hiermit abgebrochen werden. Kommt das Kommando “StepRampStop” während der Hochlaufphase, wird umgehend mit der Bremsphase begonnen. Ein StepRampXX benötigt also immer auch ein StepRampStop.

```
StepRampStop;        {Rampe down}
```

StepDestCW
Rampe hochlaufen im Uhrzeigersinn und eine bestimmte Anzahl von Schritten ausführen. Es werden die Parameter StepStartFreq, StepEndFreq, und StepAccValue benötigt aber nicht verändert. StepCount muss vorgegeben werden und wird heruntergezählt. Die Beschleunigungsrampe wird solange hochgelaufen, bis die maximale Steprate (StepEndFreq) erreicht ist oder die halbe Wegstrecke erreicht wurde. Wurde die halbe Schrittzahl erreicht, bevor das Beschleunigungsende erreicht wurde, so wird erfolgt sofort eine entsprechende Abbremsung bis zum Stillstand. Wurde das Rampenende erreicht bevor der halbe Weg zurückgelegt wurde, wird solange mit dieser Steprate gefahren, bis noch so viele Schritte übrigbleiben, dass sicher eine Bremsrampe heruntergefahren werden kann, bis die Schrittzahl 0 ist.

```
StepDestCW;          {eine Anzahl Schritte fahren}
```

StepDestCCW
Rampe hochlaufen gegen den Uhrzeigersinn und eine bestimmte Anzahl von Schritten ausführen. Es werden die Parameter StepStartFreq, StepEndFreq, und StepAccValue benötigt aber nicht verändert. StepCount muss vorgegeben werden und wird heruntergezählt. Die Beschleunigungsrampe wird solange hochgelaufen, bis die maximale Steprate (StepEndFreq) erreicht ist oder die halbe Wegstrecke erreicht wurde. Wurde die halbe Schrittzahl erreicht, bevor das Beschleunigungsende erreicht wurde, so wird erfolgt sofort eine entsprechende Abbremsung bis zum Stillstand. Wurde das Rampenende erreicht bevor der halbe Weg zurückgelegt wurde, wird solange mit dieser Steprate gefahren, bis noch so viele Schritte übrigbleiben, dass sicher eine Bremsrampe heruntergefahren werden kann, bis die Schrittzahl 0 ist.

```
StepDestCCW;         {eine Anzahl Schritte fahren}
```

3.38 Stepper-Motor im UserMode

Prinzipielles
Der Standard Schrittmotor Treiber generiert up- und Down Rampen und gibt Schritt- und Phasen Informationen auf einem wählbaren Port der CPU aus. Diese Signale dienen zum direkten Ansteuern von H-Brückenschaltungen, evtl. mit zusätzlichen DA-Wandlern für MicroSchritte.

Diese Art des Treibers ist jedoch ungeeignet für intelligente Treiber Bausteine, die z.B. ein spezielles Kommando Protokoll benötigen. Deshalb wurde der Treiber um den Modus **UserPort** erweitert.
Hier werden keinerlei Phasen oder Schritt Informationen generiert und auch ein Port für deren Ausgabe ist nicht notwendig. Es erfolgt nur die Rampen Generierung die Schrittzählung.

Die Übergabe an die Applikation erfolgt durch eine spezielle Call-Back Funktion vom Typ **UserDevice** *StepperIOS*. Diese Funktion muss von der Applikation bereitgestellt werden. Bei jedem Schritt des Treibers wird diese Funktion aufgerufen. In der Funktion muss die Applikation die Phasen und evtl. die DAC Signale selbst generieren und ausgeben. Damit lässt sich z.B. ein Schrittmotor Controller Trinamic TMC239 mit seinem SPI-Interface steuern.

Treiber-intern werden natürlich die Vorwärts und Rückwärts Schritte korrekt addiert.

Imports
Um den Stepper Treiber benutzen zu können, muss dieser in der allgemeinen Import Klausel aufgeführt werden (Import **StepPort**;).

Da der Treiber intern mit LongWord rechnet, und die Angabe der zu fahrenden Schritte auch als LongWord erfolgt, muss der Typ LongWord importiert werden (From System Import **LongWord**;).

Da der UserPort Modus kein reales Port benötigt, darf kein Ausgabe Port im Define-Block angegeben werden. Zum Aufbau der Beschleunigungs Tabelle müssen die Parameter **StepMinFreq** und **StepMaxFreq** angegeben werden. Der anzuwendende Treiber wird mit **StepType** ebenfalls unter Define angegeben.

```
Device = mega16;

Import StepPort;

From System Import longword;

Define
    ProcClock     = 16000000;     {Hertz}
    StackSize     = $0050, iData;
    FrameSize     = $0020, iData;
    StepMinFreq   = 100;
    StepMaxFreq   = 10000;
    StepType      = UserPort;
```

Wenn ein Treiber Chip nur Step und Dir Signale braucht, dann kann auf die *StepperIOS* Funktion komplett verzichtet werden wenn die zwei dazu notwendigen Pins im Define angegeben werden.

```
Define
    …
    StepType      = UserPort;
    StepPort      = PortE.0, PortE.1;     // Clock, Dir, no StepperIOS
```

Ein Beispiel für diesen speziellen Mode ist in der Demos Directory in *XMEGA_StepperU.*

3.38.1 Parameter des Steppers

Für Beschleunigungs und Bremsrampen werden zur Laufzeit diverse Parameter benötigt, die vom System beim Import des Steppers automatisch veröffentlicht werden. **Achtung:** ein Parameter darf nur im Grundzustand verändert werden!!

Var *StepStartFreq : word;*
bezeichnet die Start Steprate beim Anfahren und die Steprate, bei welcher der Bremsvorgang beendet ist. Der Wert dieser Variablen sollte nicht unter dem Tabellenwert “StepMinFreq” liegen und kleiner sein als “StepEndFreq”.

Var *StepEndFreq : word;*
bezeichnet die End Steprate bei der die Beschleunigung zu Ende ist und die Steprate, bei welcher der Bremsvorgang eingeleitet wird. Der Wert dieser Variablen sollte nicht über dem Tabellenwert “StepMaxFreq” liegen und grösser sein als “StepStartFreq”.

Var *StepAccValue : word;*
Bezeichnet die Beschleunigung in Hz/sec. Realistische Werte liegen zwischen 1000 und 10000.

Um eine bestimmte Anzahl von Schritten mit Hilfe von Beschleunigungs und Bremsrampen zu fahren (Positionierung), wird zur Laufzeit der zusätzliche Parameter

Var *StepCount : longword; {max 2^32 Steps}*
benötigt. StepCount wird mit dem gewünschten Wert besetzt, um eine bestimmte Anzahl von Schritten mit Beschleunigen und Bremsen (Rampen) fahren zu können. StepCount ist der wesentlichste Parameter für Positionierungen. Nach Beendigung des Vorgangs enthält StepCount den Wert 0.

Wird nicht auf ein Ziel gefahren wird sondern nur eine bestimmte Zeit (StepRampCW oder StepRampCCW), so enthält StepCount die Anzahl der insgesamt mit diesem Befehl gefahrenen Schritte.

Type *TStepMode = (StepStop, StepUp, StepRun, StepDown, RampDown);*
Var *StepMode : TStepMode;*
Die Variable StepMode enthält immer den aktuellen Status des Treibers. Diese Variable ist zwar auch vom Anwendungsprogramm änderbar, das kann jedoch zu erheblichen Problemen führen. Also nur lesen bitte!

Im Ruhezustand enthält StepMode den Wert *StepStop*. Während der Beschleunigungsphase ist der Wert auf *StepUp* gesetzt, während der linearen Phase (konstante StepRate) auf *StepRun* und während der Bremsphase auf *StepDown*.

Damit ist es der Anwendung z.B. möglich während der Beschleunigungs- und/oder Bremsphase mittels eines weiteren separaten Portpins eine Stromüberhöhung (Boost) einzuschalten.

Da sämtliche Parameter nur im Stillstand verändert werden dürfen, ist **vor** einer **Änderung** die Variable **StepMode** auf **StepStop** zu prüfen. In der Regel brauchen die Parameter nur einmal eingestellt werden und sie bleiben gültig bis zur nächsten Änderung, mit der Ausnahme von StepCount.

Im UserMode muss die Applikation die CallBack Funktion **StepperIOS** zur Verfügung stellen.
Diese Funktion muss das gesamte IO und die Phasengenerierung durchführen.

```
UserDevice StepperIOS (cw : boolean);
begin
  doTheJob;
end;
```

Der Parameter ***CW*** definiert die aktuelle Drehrichtung, clockwise oder counterclockwise.

3.38.2 Kommandos/Prozeduren des Steppers

Im **UserMode** kennt der Stepper Treiber zwei logische Betriebsarten mit den zugehörigen Kommandos bzw. Prozeduren. Das sind **Rampe** und **Positionierung**.

Achtung:
eine Prozedur darf nur im Grundzustand aufgerufen werden!! Ausnahme bilden die Kommandos "StepRampStop" und "StepVelocity" im Rampenmodus.

StepperOn
StepperOff

Diese zwei Funktionen sind im ***UserMode*** nicht sinnvoll und deshalb auch nicht implementiert.

StepRampCW
Rampe hochlaufen im Uhrzeigersinn. Es werden die Parameter StepStartFreq, StepEndFreq, und StepAccValue benötigt aber nicht verändert. StepCount wird zuerst auf 0 gesetzt und dann mitgezählt. Die Beschleunigungsrampe wird solange hochgelaufen, bis die maximale Steprate (StepEndFreq) erreicht ist oder der Befehl "StepRampStop" gegeben wurden. Im ersten Fall wird mit der Endfrequenz solange weitergefahren, bis der Befehl "StepRampStop" kommt, dann erfolgt eine entsprechende Abbremsung bis zum Stillstand. Kommt das Kommando "StepRampStop" während der Hochlaufphase, wird umgehend mit der Bremsphase begonnen. Ein StepRampXX benötigt also immer auch ein StepRampStop.

StepRampCW; {bis auf Widerruf laufen}

StepRampCCW
Rampe hochlaufen gegen den Uhrzeigersinn. Es werden die Parameter StepStartFreq, StepEndFreq, und StepAccValue benötigt aber nicht verändert. StepCount wird zuerst auf 0 gesetzt und dann mitgezählt. Die Beschleunigungsrampe wird solange hochgelaufen, bis die maximale Steprate (StepEndFreq) erreicht ist oder der Befehl "StepRampStop" gegeben wurden. Im ersten Fall wird mit der Endfrequenz solange weitergefahren, bis der Befehl "StepRampStop" kommt, dann erfolgt eine entsprechende Abbremsung bis zum Stillstand. Kommt das Kommando "StepRampStop" während der Hochlaufphase, wird umgehend mit der Bremsphase begonnen. Ein StepRampXX benötigt also immer auch ein StepRampStop.

StepRampCCW; {bis auf Widerruf laufen}

StepRampStop
Ist eines der beiden Rampenkommandos am laufen so kann es hiermit abgebrochen werden. Kommt das Kommando "StepRampStop" während der Hochlaufphase, wird umgehend mit der Bremsphase begonnen. Ein StepRampXX benötigt also immer auch ein StepRampStop.

StepRampStop; {Rampe down}

StepVelocity(v : word) : boolean;
Im Rampen Modus kann zu jederzeit diese Funktion aufgerufen werden. Der Parameter ***V*** ändert dabei temporär den Wert von ***StepEndFreq***. Ist das System am Beschleunigen oder Bremsen hat das keinen sofortigen Einfluss auf das Verhalten. Ist das System in der Phase der konstanten Geschwindigkeit, so wird diese jetzt entsprechend angepasst. Das erfolgt immer mit einer neuen Up oder down Rampe.
Achtung: der Wert von "v" muss grösser sein als ***StepMinFreq***!

StepVelocity (1000);

StepDestCW
Rampe hochlaufen im Uhrzeigersinn und eine bestimmte Anzahl von Schritten ausführen. Es werden die Parameter StepStartFreq, StepEndFreq, und StepAccValue benötigt aber nicht verändert. StepCount muss vorgegeben werden und wird heruntergezählt. Die Beschleunigungsrampe wird solange hochgelaufen, bis die maximale Steprate (StepEndFreq) erreicht ist oder die halbe Wegstrecke erreicht wurde. Wurde die halbe Schrittzahl erreicht, bevor das Beschleunigungsende erreicht wurde, so wird erfolgt sofort eine entsprechende Abbremsung bis zum Stillstand.

Wurde das Rampenende erreicht bevor der halbe Weg zurückgelegt wurde, wird solange mit dieser Steprate gefahren, bis noch so viele Schritte übrigbleiben, dass sicher eine Bremsrampe heruntergefahren werden kann, bis die Schrittzahl 0 ist.

StepDestCW; {eine Anzahl Schritte fahren}

StepDestCCW
Rampe hochlaufen gegen den Uhrzeigersinn und eine bestimmte Anzahl von Schritten ausführen. Es werden die Parameter StepStartFreq, StepEndFreq, und StepAccValue benötigt aber nicht verändert. StepCount muss vorgegeben werden und wird heruntergezählt. Die Beschleunigungsrampe wird solange hochgelaufen, bis die maximale Steprate (StepEndFreq) erreicht ist oder die halbe Wegstrecke erreicht wurde. Wurde die halbe Schrittzahl erreicht, bevor das Beschleunigungsende erreicht wurde, so wird erfolgt sofort eine entsprechende Abbremsung bis zum Stillstand. Wurde das Rampenende erreicht bevor der halbe Weg zurückgelegt wurde, wird solange mit dieser Steprate gefahren, bis noch so viele Schritte übrigbleiben, dass sicher eine Bremsrampe heruntergefahren werden kann, bis die Schrittzahl 0 ist.

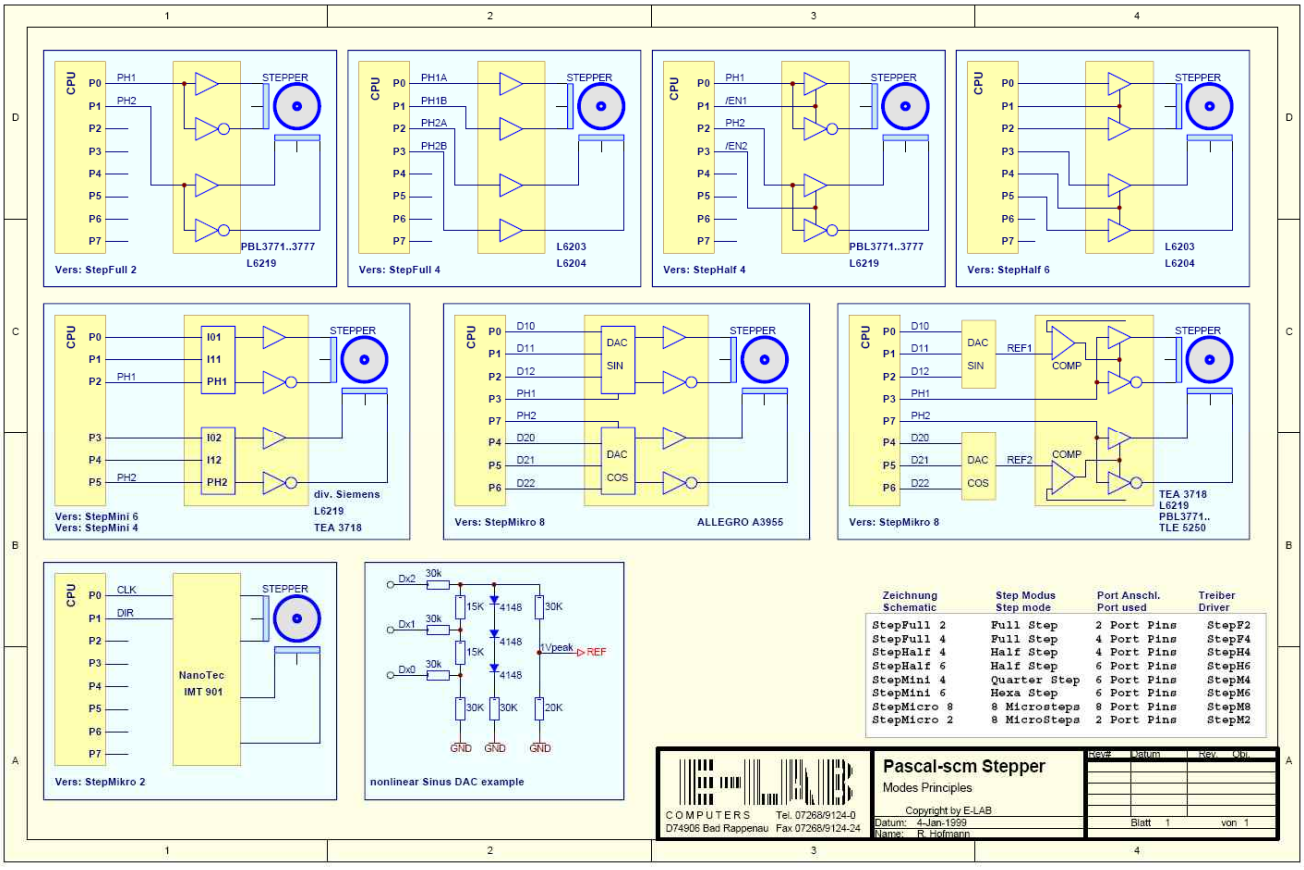
StepDestCCW; {eine Anzahl Schritte fahren}

StepPanicStop
erlaubt es während einer Rampen oder Zielfahrt sofort anzuhalten. In den meisten Fällen ist hier mit Schrittverlusten zu rechnen, so dass "StepCount" hier 0 enthält.

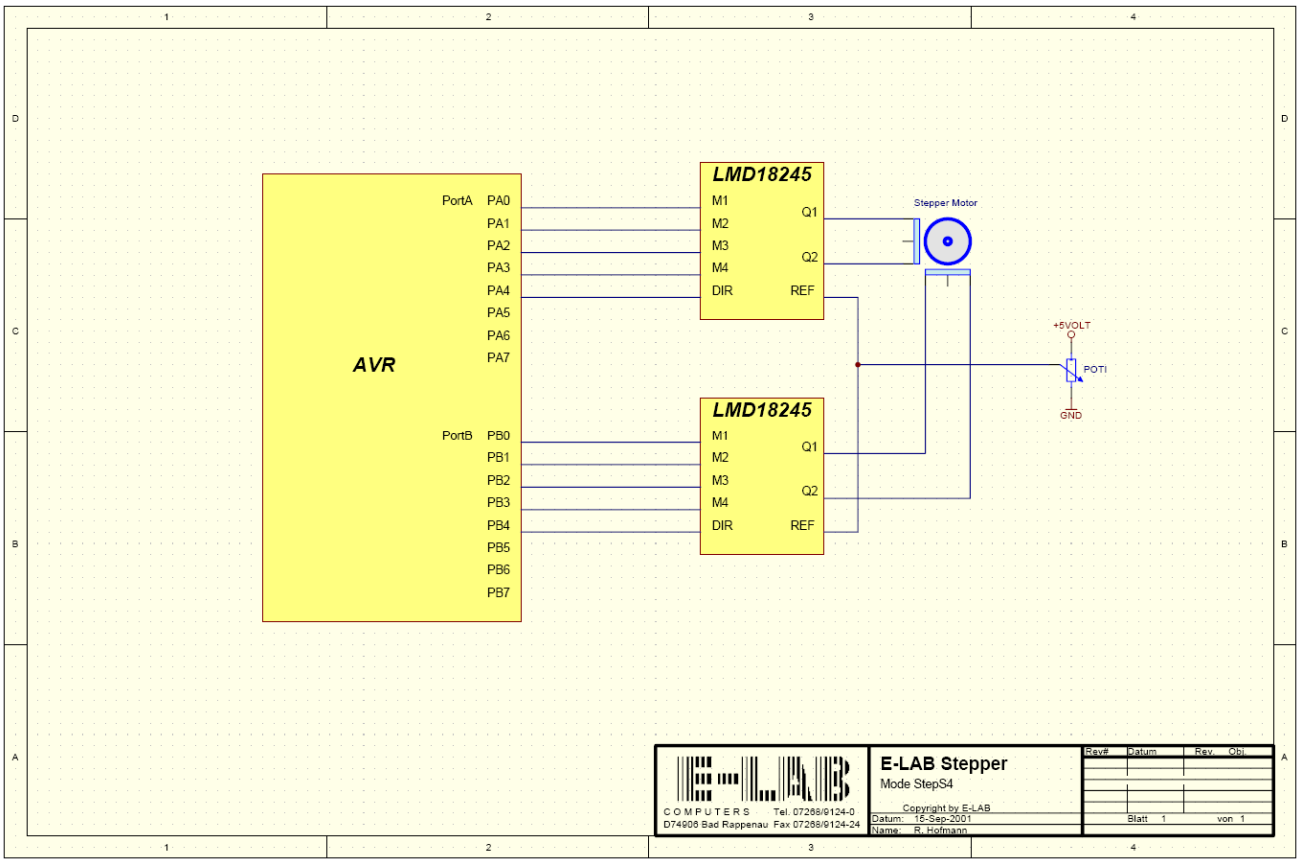
StepperSema
Der Stepper Treiber kann auch eine Semaphore verwalten. Diese wird importiert mit:

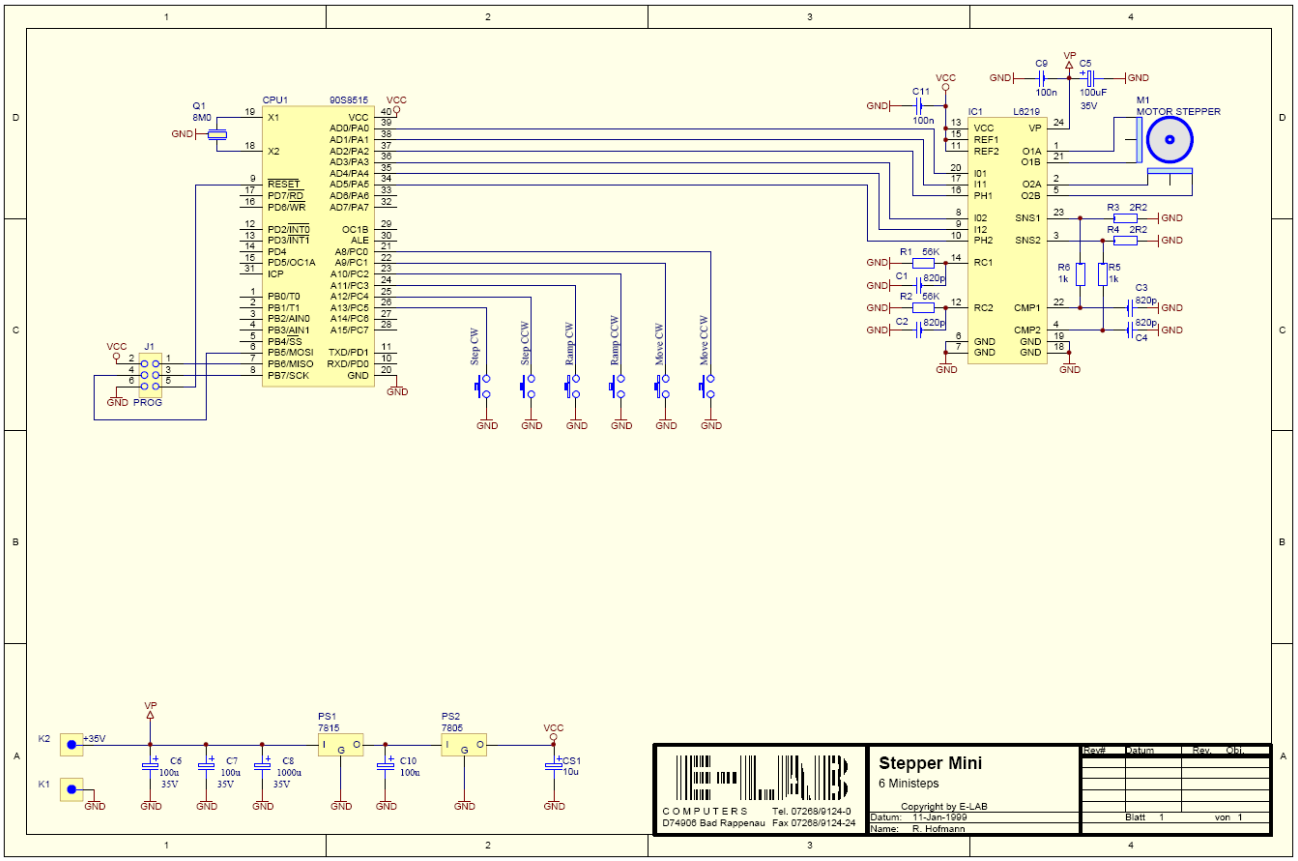
***From** StepPort **Import** StepperSema;*

Nach dem Erreichen des Stillstands bei den Funktionen StepDest oder StepRampStop wird dann diese Semaphore auf 1 gesetzt. So können Prozesse oder Tasks mit WaitSema auf den Stillstand des Motors warten.



Zeichnung Schematic	Step Modus Step mode	Port Anschl. Port used	Treiber Driver
StepFull 2	Full Step	2 Port Pins	StepF2
StepFull 4	Full Step	4 Port Pins	StepF4
StepHalf 4	Half Step	4 Port Pins	StepH4
StepHalf 6	Half Step	6 Port Pins	StepH6
StepMini 4	Quarter Step	6 Port Pins	StepM4
StepMini 6	Hexa Step	6 Port Pins	StepM6
StepMicro 8	8 Microsteps	8 Port Pins	StepM8
StepMicro 2	8 MicroSteps	2 Port Pins	StepM2


AVR
PortA
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
PortB
PB0
PB1
PB2
PB3
PB4
PB5
PB6
PB7
LMD18245
M1
M2
M3
M4
DIR
Q1
Q2
REF
Stepper Motor
+5VOLT
POTI
GND
E-LAB Stepper
Mode StepS4
Copyright by E-LAB
Datum: 16-Sep-2001
Name: R. Hofmann
Blatt 1 von 1


CPU1
90S8515
Q1
8M0
J1
PROG
Step CW
Step CCW
Ramp CW
Ramp CCW
Move CW
Move CCW
IC1
L6219
M1
MOTOR STEPPER
PS1
7815
PS2
7805
Stepper Mini
6 Ministeps
Copyright by E-LAB
Datum: 11-Jan-1999
Name: R. Hofmann
Blatt 1 von 1

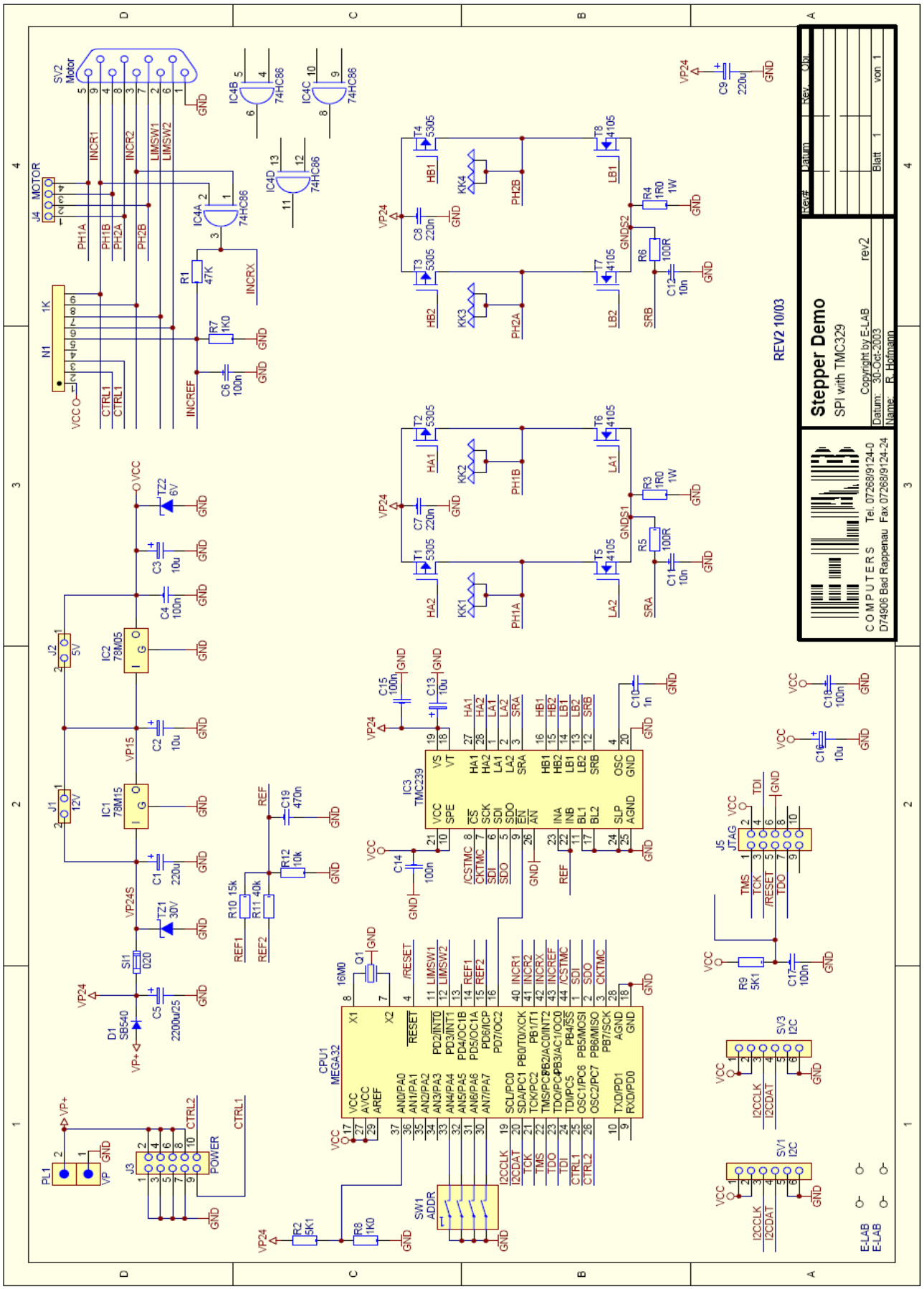
3.38.3 StepperIOS

Da das UserDevice StepperIOS aus dem Timer Interrupt heraus aufgerufen wird, müssen verwendete Register gerettet werden. Das System unterstützt die Applikation indem es folgende Register rettet, die dann in der Device Treiber Funktion ohne Einschränkung benutzt werden dürfen:

_ACCGLO R0
_ACCGHI R1
_ACCB R16
_ACCA R17
_ACCALO R18
_ACCAHI R19
_ACCDLO R20
_ACCDHI R21
_ACCELO R22
_ACCEHI R23
_ACCFLO R24
_ACCFHI R25
_ACCBLO R26
_ACCBHI R27
_ACCCLO R30
_ACCCHI R31

Beispiel und Schaltung:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\StepperDemo**
ein Xmega Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Xmega_Stepper**



Schaltplan **StepDemo** mit Trinamic Chip TMC239

3.39 Servo Treiber für bis zu 8 digitale Servos

Allgemeines
Bei kleinen Robotern, Manipulatoren und Handling Systemen werden oft die aus dem Modellbau bekannten Stellmotoren, auch Servos genannt, eingesetzt. Diese gibt es mit einem Drehmoment bis zu 100Ncm und mehr.
Durch Kugellager und Metallgetriebe werden sie auch industriellen Anwendungen gerecht, aber vor allem ist der Preis das Hauptargument für diese Antriebe.

Diese Servos werden in den meisten Fällen über Impulse gesteuert, wobei die Impulsbreite die Stell Information enthält. Dabei hat sich im Lauf der Jahre ein Standard herauskristallisiert:

Zyklus Zeit	: 20msec
Neutral Impuls Breite	: 1.5msec
Vollausschlag +	: 2msec
Vollausschlag -	: 1msec
Impuls Polarität	: positive

Die Zyklus Zeit kann bei exotischen Typen länger oder kürzer sein. Das gleiche gilt auch für die Neutral Position. Die Impulsbreiten Variation ist bei allen Typen jedoch immer +/- 0.5msec
Die Servos erlauben auch ein Trimmen der Neutral Position in gewissen Grenzen. Das geschieht zum Beispiel so dass der Impulsgenerator (Servo Controller) seine Variation von 1.5msec +/- 0.5msec auf 1.6msec +/- 0.5msec ändert.

Durch die Zykluszeit von 20msec und der maximalen Impulsbreite von 2msec (+ evtl. Neutral offset) lassen sich bis zu 8 Servos steuern, da bei Funk Fernsteuerungen die Impulse nacheinander seriell abgegeben werden und eine Pause von mehreren msec für die Synchronisation des Impuls Packets gebraucht wird.

Einführung ServoPort
Die vorliegende Implementation generiert kein serielles sequenzielles Impuls Packet sondern benutzt für jeden Servo Kanal einen separaten Portpin. Die Impulse jedes Kanals kommen trotzdem zeitlich versetzt an die Servos um das gleichzeitige Anlaufen der Motoren (Stromspitzen) zu vermeiden. Der Vorteil dieses Verfahrens ist, dass alle Impulse mit nur einem Hardware Timer der CPU generiert werden können, der im Interrupt betrieben wird.

Der Treiber arbeitet vollkommen autark im Hintergrund. Mit zwei Funktionen lassen sich die Stellposition und der Offset jedes Kanals separat einstellen. Die durch das System umgerechneten und skalierten Parameter werden in einem Array abgelegt und durch den Timer Interrupt zyklisch abgearbeitet.

Der Timer wird dabei für den Servo Treiber reserviert und kann für keine anderen Dienste herangezogen werden. Beim AVR kommen hierbei nur der Timer1 oder der Timer3 (mega64, mega128) in Betracht.

Achtung:

Da der Timer im Interrupt arbeitet, schlägt jede Interrupt response Verzögerung sich in einem Jitter (wackeln) der generierten Impulse nieder, was evtl. zu einem Knarren der Servos führen kann.
Solche Interrupt Verzögerungen werden hervorgerufen durch

1. Sperren des Globalen Interrupts durch die Applikation. Abhilfe schafft hier: vermeiden der Sperrung.
2. System Interrupts, UART, Timer etc. Das System selbst vermeidet lange Verweilzeiten in System eigenen Interrupts. Bei dem SystemTick mit vielen Aufgaben, SystemTimer, RTC, MultiTasking Scheduling kann es jedoch zu Schwankungen im 10..30usec Bereich kommen. Es ist im Einzelfall zu prüfen, ob der Jitter hierbei noch tolerabel ist.
3. Applikations Interrupts. Hier gilt was bei Interrupts grundsätzlich immer zu beachten ist, je länger die Rechenzeit innerhalb eines Interrupts, desto mehr werden andere Interrupts zeitlich verschoben oder fallen evtl. einmal komplett aus.

3.39.1 Technische Daten

Zyklus Zeit	20msec fest eingestellt
Neutral Impuls Dauer	einstellbar zwischen 1.0msec und 2.0msec
Impuls Variation	+/- 0.5msec default, definierbar zwischen +/-0.5 und 1.0msec
Neutral Offset	+/- 30% der Impuls Variation
Kanäle	1..8 chan0..chan7
Timer	Timer1 oder Timer3 (XMegas Timer_C0…Timer_F1)

Imports
Wie beim AVRco System üblich, muss der Treiber importiert und definiert werden.

Der SysTick wird nicht benötigt.

```
Import  ServoPort;
```

Defines
Es muss spezifiziert werden, welches Port der CPU und welche bits davon benutzt werden sollen, wo die notwendigen Variablen liegen, wieviel Kanäle benötigt werden, deren Polarität, der Neutral Impuls und der zu verwendende Timer.

```
Define  ProcClock    = 8000000;          {8Mhz clock }
        ServoPort    = PortB, 2, iData;  {Port, Startbit im Port, Datenbereich}
        ServoChans   = 4, Positive;      (4 channels, positive pulse}
        ServoNeutral = 1.5, Timer1;      (1.5msec neutral position, use Timer1}
        ServoSwing   = 1.0;              (adjustable between 0.5 and 1.0msec, Resolution 100 pts}
Alternative Auflösung von 1000 Punkten
        ServoSwing   = 1.0, 1000;        (adjustable between 0.5 and 1.0msec, Resolution 1000 pts}
```

ServoPort
Definiert das zu verwendende Port, das erste benutzbare bit und den Datenbereich des Treibers. Das Port kann ein output-only Port sein. Das Startbit im Port muss so gewählt werden, dass die notwendigen Output Pins auch noch in das Port passen. Der Datenbereich bestimmt, wo die Arbeitsvariablen des Treibers liegen.

ServoChans
Definiert die Zahl der Servo Kanäle (1..8) und bestimmt dadurch auch den internen Speicherbedarf.
Die Impuls Polarität wird für alle Kanäle mit „Positive“ oder „Negative“ eingestellt.

ServoNeutral
Definiert die Neutral Stellung für alle Kanäle (1.0 … 2.0). Es wird ein 16bit Timer gebraucht, der beim AVR zumindest immer der Timer1 ist. Bei neueren Typen (mega64, mega128) ist auch noch der Timer3 möglich. Bei den XMegas muss einer der Timer_C0…Timer_F1 verwendet werden, soweit vorhanden.

ServoSwing
Optional, default 0.5msec. Definiert den max. Ausschlag für alle Kanäle (+/-0.5 … +/-1.0msec).
Als weitere Option kann hier auch noch die Auflösung (Präzision) auf 1000 Punkte heraufgesetzt werden.

3.39.2 Funktionen

SetServoChan
Prozedur zum Verändern der Servo Position.
Der Parameter chan gibt den Kanal an (0..ServoChans-1)
Mit der Auflösung von 100 gibt der Parameter pulse die Position an in % Vollausschlag (-100...+100).
Mit der Auflösung von 1000 gibt der Parameter pulse die Position an in 1/1000 vom Vollausschlag (-1000...+1000).
Falsche Kanal Nummern werden nicht ausgeführt, zu grosse Werte für pulse werden gekappt auf +/-100 bzw. +/-1000.

```
Procedure SetServoChan(chan : byte; pulse : integer);
```

SetServoOffs
Verändert die Servo 0-Position. Der Parameter chan gibt den Kanal an (0..ServoChans-1)
Der Parameter offs gibt die 0-Position Abweichung an in % (Vollausschlag div 2) an (max -30...+30). Falsche Kanal Nummern werden nicht ausgeführt, zu grosse Werte für offs werden gekappt auf +/-30

Procedure *SetServoOffs (chan : byte; offs : integer);*

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Servo**
ein **XMega** Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Xmega_Servo**

3.40 DCF-77 Dekoder

Echtzeituhren werden bei mittleren und grösseren Systemen immer wichtiger. Die AVRco Bibliothek enthält deshalb einen Echtzeit-Uhr Treiber, der sowohl mit der internen Echtzeituhr diverser AVRs als auch mit dem SysTick als Grundlage auskommt. Die interne RTC ist dabei etwas genauer als die Version mit dem SysTick, wobei für die meisten Fälle die Genauigkeit der SysTick Version ausreicht.

Mit der Präzision gibt es also keine Probleme. Anders sieht es aber mit der Initialisierung der RTC aus. Hier muss nach jedem Power-Up die Uhr neu gestellt werden. Neben dem zusätzlichen Hard- und Software Aufwand, den man benötigt, um die Uhr stellen zu können, stellt sich auch noch die Gefahr einer falschen Eingabe durch den Benutzer oder sogar ein Vergessen des Stellvorgangs.

Um diesen Problemen und Kosten aus dem Weg zu gehen bieten sich zwei Möglichkeiten.

1. Einsatz eines zusätzlichen Uhrenchips mit Akkupufferung. Hier erledigt sich das Stellen der Uhr von selbst bzw. es bleibt beim einmaligen Stellen. Allerdings sind dafür zumindest zusätzliche Portpins (6..8) bzw. bei einer I2C-Uhr wird ein weiterer Treiber gebraucht. Weiterhin ist langfristig mit einer Abweichung der Uhr zu rechnen, ganz abgesehen von Schaltjahr und Winter/Sommerzeit Umstellungen.

2. Einsatz eines DCF77 Empfängers. Hier gibt es absolut keine Stell- oder Genauigkeits Probleme. Einzig die erforderliche Setup-Zeit von bis zu 2 Minuten kann manchmal etwas stören. Je nach Umgebungs Bedingungen kann u.U. der Platz bzw. Position der Ferrit Antenne etwas kritisch sein. Der DCF77 Treiber klinkt sich in den SysTick ein.

Wenn man die Wahl hat, sollte man der DCF77 Uhr den Vorzug geben. Bei einigermassen erträglichen Empfangsbedingungen ist mit einem problemlosen Betrieb zu rechnen.

Die DCF77 Uhr ist jedoch keine Echtzeit Uhr im herkömmlichen Sinn. Sie muss immer in Verbindung mit einer anderen Uhr gesehen werden. Das AVRco System bietet hier drei Möglichkeiten:

1. DCF77 steuert die RTC einer AVR CPU. Hierzu ist der RTC Treiber zu importieren
2. DCF77 steuert die RTC, welche vom SysTick angetrieben wird.
3, DCF77 steuert einen externen Uhren Chip

Die Punkte 1 und 2 sind aus der Sicht des DCF77 Treibers identisch. Wird eine fehlerfreie Zeit empfangen, so wird bei jeder vollen Minute die RTC neu gestellt. Das geschieht automatisch ohne weiteres Zutun. Der DCF77 Treiber erkennt die RTC und handelt entsprechend.

Der Punkt 3 (Uhren Chip) wird durch eine **CallBack** Prozedur unterstützt. Ist diese Prozedur vorhanden, wird sie bei jeder vollen Minute vom Treiber aufgerufen, wenn die Zeit fehlerfreie empfangen wurde. Die CallBack Prozedur, die der Programmierer bereitstellen muss, kann jetzt das Uhrenchip stellen, indem die jeweiligen Speicherstellen der DCF77 in das Chip geschrieben wird. Diese Prozedur wird aus dem SysTick heraus aufgerufen. Hierbei ist die Register Rettung zu beachten, weiterhin sollte die Prozedur sehr schnell sein, um den hier gesperrten globalen Interrupt nicht zu lange zu sperren.

Der Uhrenchip ist aber ein Sonderfall. In den allermeisten Fällen genügt der RTC Treiber, um alle Aufgaben erledigen zu können.

Imports
Der Treiber muss, wie beim AVRco üblich, importiert werden.

Import *SysTick, DCFclock, ..;*

Defines
Der als DCF-Eingang verwendete Portpin sowie die Lage der DCF Uhren Register werden mit Define vorgegeben:

```
Define  {mega103}
    ProcClock      = 8000000;          {Hertz}
    SysTick        = 10 , Timer2;      {msec}
    DCFclock       = iData;
    DCFport        = PinD, 2, negative;  {Port, bitnummer, Polarität}
    DCFfieldMode   = reset;             // decrement, optional "reset"
```

Achtung:
Der SysTick muss bestimmte Bedingungen erfüllen. Er darf nicht kleiner 1msec oder grösser 100msec sein, er muss ganzzahlig sein und 1000 / SysTick darf keinen Rest ergeben
(1000 mod SysTick = 0).

3.40.1 DCF77-Funktionen/Prozeduren

Grundsätzliche und Implementations unabhängige Prozeduren und Funktionen.

Function *DCFready : boolean;*
Das Ergebnis der Funktion wird wahr, wenn der DCF77 Treiber zum ersten mal ein komplettes Telegramm erhalten hat und im Falle der RTC-Implementation, die RTC komplett initialisiert hat. Ab diesem Moment kann man davon ausgehen, dass die Daten der RTC absolut stimmen. Dieser Vorgang dauert nach dem Einschalten mindestens 1 Minute und wenn der Empfang ok ist, maximal 2 Minuten, ansonsten kann es mehrere Minuten dauern.

Function *DCFfield : byte;*
Das Ergebnis der Funktion gibt einen ungefähren Wert für die Empfangsqualität wieder. Null ist kein Empfang, 255 ist optimaler Empfang. Es dauert mehrere Minuten bis sich dieser Wert stabilisiert hat.

Mit jedem gültigen Frame (1sec) wird der Wert um 1 inkrementiert. Bei fehlerhaftem Frame gibt es zwei Möglichkeiten:
Ist das Define **DCFfieldMode** auf „decrement“ gesetzt, wird bei einem Fehler der Wert um 1 dekrementiert.
Ist das Define **DCFfieldMode** auf „reset“ gesetzt, wird bei einem Fehler der Wert auf 0 gesetzt.

Function *DCFDayLightSave : boolean;*
Das Ergebnis der Funktion gibt die Sommer/Winterzeit zurück. Sommerzeit = true.

Procedure *DCFupdate;* *// Callback Procedure*
Findet der DCF77 Treiber den Import der RTC nicht, so ruft er jede Minute die Prozedur „DCFupdate“ auf, falls diese vorhanden ist. Diese Prozedur kann jetzt z.B. ein externes Uhrenchip beschreiben. Dazu können die Speicherstellen/Variablen

DCF_ SECOND, DCF_MINUTE, DCF_HOUR, DCF_DAY, DCF_MONTH, DCF_YEAR, DCF_WEEKDAY

benutzt werden. Hierbei ist unbedingt zu beachten, dass der Aufruf dieser Funktion aus dem SysTick heraus erfolgt. Evtl. benutzte Register retten. Möglichst sehr schnellen Assembler Code verwenden.

3.40.2 Hardware

DCF77 Empfänger gibt es als fertige kleine Boards als auch als Wandmontage Gehäuse. Die Wandmontage ist vorzuziehen, da der Standort bzw. der Abstand von einer Störquelle, z.B. PC eminent wichtig ist. Ein gestörter Empfang kann dazu führen, dass der DCF77 Treiber viele Minuten braucht, bis er zum ersten mal ein fehler freies Telegramm empfangen hat. Bis dahin arbeitet die angebundene RTC mit Hausnummern.

Ein DCF77 Empfangsmodul schickt ein bitserielles Telegramm zur CPU. Bei der CPU genügt ein einziger Input Pin zum Empfang.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\DCF77**

3.41 AVR Timer Low Level Treiber TickTimer

Sehr häufig wird ein zyklischer Interrupt oder Signalzug benötigt, der sich sehr fein und genau einstellen lässt. Die vorhandenen System Timer sind für diese Zwecke weniger gut geeignet, vor allen Dingen dann, wenn solch ein Timer Tick immer konstant ausgeführt werden muss. Hierzu ist ein Hardware Timer unbedingt notwendig.

Der hier beschriebene Treiber benutzt einen 16bit Timer der AVR CPU, entweder TIMER1 oder, wenn vorhanden, den Timer3, 4 oder 5. Der gewählte Timer ist dann für andere Aufgaben (Stepper etc) nicht mehr zu verwenden. Bei beiden Timern wird der COMPARE-MATCH-A benutzt. Es ist auch möglich den 8bit Timer2 zu benutzen, falls vorhanden.

Das Timing kann zwischen 10usec und 4sec in 1usec Inkrements eingestellt werden. Beim Timer2 ist die Auflösung stark vom Prozessorclock abhängig. Der Treiber stellt optional einen Interrupt bereit. Optional kann auch der Ausgang des Timers an den zugehörigen Compare-Match Pin durchgeschaltet werden.

Durch den Import des Treibers und die zugehörigen Defines wird auch das Setup des Timers vorgegeben. Es stehen mehrere unterschiedliche Funktionen zur Verfügung. Die Tick-Zeit kann eingestellt werden. Der Timer kann angehalten und neu gestartet werden. Der optionale Output Pin kann gesperrt und freigegeben werden.

Imports

```
Import SysTick, TickTimer, ..;          // Xmega TickTimer and/or TickTimer2

From System Import LongWord;  // needed for calculations
```

Defines

```
Define  ProcClock    = 16000000;        {Hertz}
        SysTick      = 10;              {msec}
        StackSize    = $0030, iData;
        FrameSize    = $0030, iData;
        TickTimer    = Timer1;          // use Timer1.COMPA and no PortPin
        //TickTimer  = Timer1, pinout;  // use Timer1.COMPA and its PortPin
        //TickTimer  = Timer2;          // use Timer2.COMPA and no PortPin
        //TickTimer  = Timer2, pinout;  // use Timer2.COMPA and its PortPin
        //TickTimer  = Timer3;          // use Timer3.COMPA and no PortPin
        //TickTimer  = Timer3, pinout;  // use Timer3.COMPA and its PortPin
```

Mit dem TickTimer Define wird ein 8/16bit Timer ausgewählt. In den meisten AVRs ist mindestens der Timer1 vorhanden. In den neueren und grösseren Typen gibt es als Alternative auch den Timer3, den 8bit Timer2 und evtl. die Timer4 und Timer5 die ebenfalls unterstützt werden, falls vorhanden.

Mit der optionalen Define Erweiterung ***pinout*** wird der COMPA Output Pin des Timers freigegeben und der Timer toggelt diesen Pin während des Laufs. Die Funktion ***TickTimerOutpEnable*** ist nur in mit diesem Define nutzbar, ebenso die Variable ***TickTimerPin***.

XMega

Bei den XMegas muss ein Timer_C0…Timer_F1 definiert werden, soweit vorhanden. Zusätzlich kann statt “PinOut“ das gewünschte Output Port und Pin angegeben werden. PinOut selbst ist hier nicht zulässig.

```
TickTimer    = Timer_C0;              // use Timer_C0 and no PortPin
TickTimer2   = Timer_C1, PortE, 5;    // use Timer_C1 and PortE, pin 5
```

3.41.1 Variable

```
Var TickTimerPin : bit;          { TickTimer2Pin}
```

Diese Variable kann dazu benutzt werden um „zu Fuss“ den Timer Pin zu schalten, wenn der Timer angehalten ist. Sie ist aber nur vorhanden, wenn ***pinout*** definiert wurde.

3.41.2 Funktionen

```
Function  TickTimerTime (time : longword) : boolean;                 // XMega also TickTimer2Time
Function  TickTimerReload (time : longword) : boolean;               // XMega also TickTimer2Reload
Procedure TickTimerRawVal (presc : byte; cmp : word|byte);           // XMega also TickTimer2RawVal
Procedure TickTimerStart;                                            // XMega also TickTimer2Start
Procedure TickTimerStop;                                             // XMega also TickTimer2Stop
Procedure TickTimerOutpEnable (enable : boolean; Level : byte);      // XMega also TickTimer2OutpEnable
```

Function *TickTimerTime (time : longword) : boolean; { TickTimer2Time}*
Stoppt den evtl. laufenden Timer, berechnet die neue Einstellung und schreibt diese in den Prescaler und Compare Register des Timers. Das Argument ***time*** gibt die Tick-Zeit (Pause zwischen zwei Interrupts oder Pin toggles) in usec an. Der (sinnvolle) Minimalwert des Ticks ist 10usec. Die mögliche Obergrenze hängt absolut vom Prozessor Clock ab. Bei 16MHz sind das ca. 4sec, bei 8MHz ca. 8sec. Beim 8bit Timer2 erhöht sich diese Untergrenze entsprechend. Der Prozessor Clock darf nicht kleiner 1MHz sein. Das Ergebnis wird false wenn das time Argument zu illegalen Timer Werten führt.

Function *TickTimerReload (time : longword) : boolean; { TickTimer2Reload}*
Im Gegensatz zu TickTimerTime wird der Timer nicht gestoppt oder gestartet. Ansonsten gilt das gleiche wie oben stehend. Diese Funktion dient dazu dass man während der Timer aktiv ist, das Timing verändern kann ohne den Timer zu stoppen.

Procedure *TickTimerRawVal (presc : byte; cmp : word|byte); { TickTimer2RawVal}*
Das ist die Alternative zur obigen Funktion TickTimerTime. Hier kann man direkt den Prescaler und den Compare Wert vorgeben. Für die Timer1 und 3 muss der Prescaler Wert im Bereich 1..5 liegen. Für den 8bit Timer2 muss der Wert zwischen 1..7 liegen. Der Compare Wert ist ein word für die beiden 16bit Timer und ein byte für den 8bit Timer2.

Procedure *TickTimerStart; { TickTimer2Start}*
Startet den Timer und seinen Interrupt, wenn vorhanden. Diese Funktion muss nach einer Einstellungs Änderung durch obige erste zwei Funktionen aufgerufen werden. Nach einem Programm Start/Reset steht der Timer ebenfalls auf Stop.

Procedure *TickTimerStop;*
Stoppt den Timer und seinen Interrupt, wenn vorhanden. Restart des Timer mit TickTimerStart.

Procedure *TickTimerOutpEnable (enable : boolean; Level : byte); { TickTimer2OutpEnable}*
Wurde der Timer Ouitput mit dem Define **pinout** importiert bietet das System auch diese Funktion an. Ein Timer Stop oder Start hat keinen Einfluss auf den Status des COMPA Pins. Im Stop-Zustand behält dieser Pin seinen zuletzt vom Timer vorgegeben Zustand. Ist der Timer gestartet, toggelt der Pin mit der vorgegebenen Rate. Mit dieser Funktion kann der Output Pin entweder mit **enable = true** dem Timer zugewiesen werden oder mit **enable = false** dem Timer entzogen werden. Im letzteren Fall hat der Timer zwar keinen Einfluss mehr auf den Pin, dieser behält jedoch seinen letzten Status bei. Deshalb kann beim Abschalten des Pins mit dem zweiten Argument **level** der statische Pegel des Pins vorgegeben werden. Mit level <> 0 wird der Pin log 1. Ist der Pin für den Timer freigegeben hat das Argument level keine Bedeutung.

3.41.3 Interrupt

Der Timer generiert auf Wunsch auch einen Interrupt. Dazu muss die zugehörige Callback Prozedur implementiert werden. Diese kann dann zu weiteren Diensten herangezogen werden. Zu beachten gilt dabei, dass dies ein Callback aus einem Interrupt heraus ist.

```
Procedure onTickTimer;     // onTickTimer(SaveAllRegs);          { onTickTimer2}
begin
 ...
end;
```

Programm Beispiel:
ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\TickTimer**
ein XMega Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_TickTimer**

3.42 FreqCount Treiber - Frequenzzähler/Timer

Nicht für XMegas

In manchen Applikationen wird eine Funktion gebraucht, die eine externe Frequenz feststellen bzw. auszählen kann. Oder die Länge eines Impulses muss fest gestellt werden. Für einen erfahrenen Programmierer kein Problem. Man nehme einen 16bit Hardware Zähler, ein geeignetes Zeit-Tor und fertig ist der Frequenz Zähler oder Timer. Aber wie so oft, die kleinen Details machen die Probleme.

Es bietet sich daher an, diese Funktion komplett als Bibliotheks Treiber zu implementieren, so dass sich der Programmierer um seine eigentliche Aufgabe, die Applikation, kümmern kann.

3.42.1 Einführung Frequenz Zähler

Die vorliegende Implementation benutzt einen 16bit Timer (Timer1 oder auch Timer3 beim mega128) als Impuls Zähler. Beim Frequenz Zähler wird die Torschaltung wird durch den SystemTick erledigt. Das spart Resourcen in der CPU. Die Torschaltung lässt sich auf 10sec, 1sec, 100msec und 10msec einstellen. Das Ergebnis der Messung ist ein 16/32bit Wert.

Der Nachteil bei der Verwendung des SystemTicks ist die etwas reduzierte Genauigkeit des Resultats und gewisse Einschränkungen bei der Wahl des SysTicks. Die Genauigkeit liegt hier bei 0.1%..0.2% abhängig vom Prozessor Clock und dem SysTick Wert. Hierbei gilt: Je höher der Clock und je kleiner die Zeit des SysTicks desto besser, in gewissen Grenzen. Optimale Ergebnisse erzielt man mit einem Clock >= 8MHz und einem Tick von 2msec. Weiterhin ist mit einem Jitter bei Interrupts zu rechnen.

Einschränkungen gibt es beim SysTick. Durch die notwendigen Torzeiten gilt folgendes:

<u>100Hz Bereich</u>

Auflösung	0.01Hz		
Max Frequenz	655.35Hz	=	0.65535kHz
Torzeit	100sec		
SysTick	1msec, 2msec, 2.5msec, 4msec, 5msec, 8msec, 10msec		

<u>1kHz Bereich</u>

Auflösung	0.1Hz		
Max Frequenz	6553.5Hz	=	6.5535kHz
Torzeit	10sec		
SysTick	1msec, 2msec, 2.5msec, 4msec, 5msec, 8msec, 10msec		

<u>10kHz Bereich</u>

Auflösung	1Hz		
Max Frequenz	65535Hz	=	65.535kHz
Torzeit	1sec		
SysTick	1msec, 2msec, 2.5msec, 4msec, 5msec, 8msec, 10msec		

<u>100kHz Bereich</u>

Auflösung	10Hz		
Max Frequenz	655350Hz	=	655.35kHz
Torzeit	100msec		
SysTick	1msec, 2msec, 2.5msec, 4msec, 5msec, 10msec		

<u>1MHz Bereich</u>

Auflösung	100Hz		
Max Frequenz	6553500Hz	=	6.5535MHz
Torzeit	10msec		
SysTick	1msec, 2msec, 2.5msec, 5msec, 10msec		

Im MHz Bereich wird die maximale Frequenz auch noch durch den Prozessorclock beschränkt. Der externe Takt eines Timers sollte ProcClock/4 nicht überschreiten. Bei 8MHz CPU clock sind das dann 2MHz.

Die zu messende Frequenz muss an dem jeweiligen Clock Eingang des verwendeten Timers angeschlossen werden. Bei Timer1 ist das der PIN T1 und bei dem Timer3 ist es PIN T3. Die Applikation muss sicherstellen, dass der entsprechende Port Pin auf Input programmiert ist.

3.42.2 Einführung Puls Timer

Die Implementation des Impuls Messers benutzt den gleichen 16bit Timer wie der Frequenz Messer Teil. Der SysTick wird hierbei nicht benötigt und unterliegt bei der Timer Funktion daher keiner Beschränkung.

Die erzielbare Auflösung und die Genauigkeit hängt beim Timer Mode von dem CPU-Clock, und der Abstufung der verfügbaren Vorteiler des benutzten 16bit Timers ab. Hier gilt: je höher der CPU-Clock desto kürzere Zeiten können gemessen werden, desto kürzer wird aber auch der längste mögliche Impuls. Je niedriger der CPU Clock desto längere Zeiten können gemessen werden.

Im Gegensatz zum Frequenz Zähler Teil, der absolut vom SysTick abhängig ist, ist der Timer Teil absolut vom Prozessor Clock abhängig. Der Treiber läuft optimal zwischen 4 und 16MHz.

Die minimale Auflösung lässt sich in vier Stufen einstellen: 10usec, 100usec, 1msec und 10msec.
Die typische, maximale Impuls Zeit beträgt dabei: 100msec, 1sec, 10sec und 100sec.

<u>100msec Bereich</u>
Auflösung 10usec
Max Zeit min 100msec max 500msec

<u>1sec Bereich</u>
Auflösung 100usec
Max Zeit min 1sec max 4sec

<u>10sec Bereich</u>
Auflösung 1msec
Max Zeit min 4sec max 10sec

<u>100sec Bereich</u>
Auflösung 10msec
Max Zeit min 40sec max 100sec

Der zu messende Impuls muss an dem jeweiligen Capture Eingang des verwendeten Timers angeschlossen werden. Bei Timer1 ist das der PIN ICP1 und bei dem Timer3 ist es PIN ICP3. Die Applikation muss sicherstellen, dass der entsprechende Port Pin auf Input programmiert ist.

Der Treiber misst entweder einen kompletten Zyklus (Periode) oder einen positiven Puls, abhängig vom eingestellten Mode.
TTimeBasexx = Perioden Messung
TPulseBasexx = Pulse Messung (positiver Puls).

Der Timer Mode benutzt den InputCapture Mode eines 16bit Timers. Das Auslesen des Timers erfolgt im InputCaptureInterrupt und belastet die CPU nur unwesentlich. Allerdings kann es hier bei zu langen Interrupt Sperrzeiten durch andere Teile des Programms sporadisch zu Ergebnis Verfälschungen kommen, die das System nicht erkennen kann. Die Applikation sollte in solchen Fällen das Messergebnis auf Plausibilität prüfen.

Es sind zwei Timer bzw. Zähler Kanäle möglich, FreqCounter und FreqCounter2.
Der zweite Kanal FreqCounter2 ist fest an den Timer3 der CPU gebunden. Deshalb kann er nur mit CPUs benutzt werden, die diesen Timer onChip haben, z.B. mega64 und mega128.

Imports
Wie beim AVRco System üblich, muss der Treiber importiert und definiert werden. Zur kompletten Implementation gehört auch der Import des SysTicks.

FreqCount und/oder FreqCount2
Importiert die Frequenz Zähler/Timer **FreqCount** oder **FreqCount2**

Der Import von FreqCount importiert auch automatisch diverse Bibliotheksfunktionen.
Es muss auch der SysTick importiert werden.

Import *SysTick, FreqCount [, FreqCount2];*

Defines
Der Treiber benutzt einen internen 16bit Timer, entweder Timer1 oder Timer3, falls vorhanden.

```
Define   ProcClock   = 8000000;    {8Mhz clock }
         SysTick     = 2.0;         {2msec Tick}
         FreqTimer   = Timer1;      (used 16bit Timer}
         FreqTimer2  = Timer3;      (restricted to Timer3}
```

FreqTimer, FreqTimer2
Definiert den zu verwendenden 16bit Timer. FreqTimer2 muss auf den Timer3 gelegt werden. Der SysTick sollte nur folgende Werte haben: 1msec, 2msec, 2.5msec, 4msec, 5msec, 8msec, 10msec. Alle anderen können zu grösseren Messfehler führen. Weiteres weiter oben in der Einführung.

3.42.3 Funktionen, Prozeduren, Typen

tFreqCountMode
Der Import des Treibers stellt den Typ tFreqCountMode zur Verfügung. Dieser Aufzählungstyp wird zur Einstellung der Torzeit des Zähler bzw. des prescalers benötigt. Folgende Torzeiten bzw. Auflösungen sind zugeordnet:

TFreqBaseNone		
TFreqBase100Hz	Frequ = 0.01Hz...655.35Hz	gate time = 100sec
TFreqBase1kHz	Frequ = 0.1Hz...6.5535kHz	gate time = 10sec
TFreqBase10kHz	Frequ = 1Hz...65.535kHz	gate time = 1sec
TFreqBase100kHz	Frequ = 10Hz...655.35kHz	gate time = 100msec
TFreqBase1MHz	Frequ = 100Hz...6.5535MHz	gate time = 10msec
TTimeBase100s/ TPulseBase100s	Time = 10msec…100sec	periode/pulse
TTimeBase10s/ TPulseBase10s	Time = 1msec…10sec	periode/pulse
TTimeBase1s/ TPulseBase1s	Time = 100usec…1sec	periode/pulse
TTimeBase100ms/ TPulseBase100ms	Time = 10usec…100msec	periode/pulse

Die Definition von tFreqCountMode ist:

```
Type tFreqCountMode = (TFreqBaseNone, TFreqBase100Hz, TFreqBase1kHz, TFreqBase10kHz,
                       TFreqBase100kHz, TFreqBase1MHz,   TTimeBase100s, TTimeBase10s,
                       TTimeBase1s, TTimeBase100ms, TPulseBase100s, TPulseBase10s,
                       TPulseBase1s, TPulseBase100ms);
```

FreqCountSema, FreqCountSema2
Diese Semaphore wird bei jedem neuen Messergebnis um eins inkrementiert. Nützlich nicht nur bei MultiProzessing sondern auch um bei normalen Anwendungen Rechenzeit zu sparen.

SetFreqCountMode, SetFreqCountMode2
Mit dieser Prozedur wird die Torzeit/Auflösung des Zählers oder der Vorteiler des Timers eingestellt. Diese Einstellung sollte als erstes erfolgen.

Procedure *SetFreqCountMode (mode : tFreqCountMode);*
Der Parameter **mode** ist die gewünschte Auflösung. Achtung: zumindest der erste Messzyklus nach diesem Funktionsaufruf liefert falsche Daten.

GetFreqCountMode, GetFreqCountMode2
Die Funktion GetFreqCountMode liefert als Ergebnis den aktuellen Modus des Zählers zurück

Function *GetFreqCountMode : tFreqCountMode;*

GetFreqCounter, GetFreqCounter2
Die Funktion GetFreqCounter liefert als Ergebnis (16bit Wert) der letzten Frequenz Messung zurück. Das Ergebnis muss, abhängig von dem eingestellten Modus, entsprechend interpretiert werden, d.h. das Ergebnis ist eigentlich ein Festkomma Wert, dessen Kommastelle abhängig von dem eingestellten Modus ist. Im Overflow Fall wird der Wert $FFFF zurückgegeben. Der Aufruf dieser Funktion setzt auch die Semaphore auf 0.
Function *GetFreqCounter: word;*

Achtung:
Diese Funktion sollte nur verwendet werden, wenn auch der Frequenz Zähler Modus eingestellt ist. Ansonsten muss die Funktion **GetTimeCounter** benutzt werden.

GetFreqCounterL, GetFreqCounter2L
Die Funktion GetFreqCounterL liefert als Ergebnis (32bit Wert) der letzten Frequenz Messung zurück. Das Ergebnis muss, abhängig von dem eingestellten Modus, entsprechend interpretiert werden, d.h. das Ergebnis ist eigentlich ein Festkomma Wert, dessen Kommastelle abhängig von dem eingestellten Modus ist. Im Overflow Fall wird der Wert $FFFFFFFF zurückgegeben. Der Aufruf dieser Funktion setzt auch die Semaphore auf 0.
Function *GetFreqCounterL: longword;*

Achtung:
Diese Funktion sollte nur verwendet werden, wenn auch der Frequenz Zähler Modus eingestellt ist. Ansonsten muss die Funktion **GetTimeCounter** benutzt werden.

GetFreqCountOvrFlow, GetFreqCountOvrFlow2
Die Funktion GetFreqCountOvrFlow liefert als Ergebnis einen evtl. vorhandenen Überlauf der letzten Messung zurück. Dieser Wert ist normalerweise NULL. Wenn nicht, ist beim Frequenz Zähler Modus die Torzeit zu lang und die Frequenz zu hoch für die aktuelle Einstellung. Wird mit 32bit Werten gearbeitet, d.h. das Ergebnis wird mit **GetFreqCounterL** ausgelesen, gibt es keinen Overflow und diese Funktion wird nicht benötigt.
Beim Pulse Timer ist im Overflow Fall der Zähler (Prescaler) zu schnell und es kommt zum Überlauf.
Function *GetFreqCountOvrFlow: byte;*

FreqCountRestart, FreqCountRestart2
Diese Prozedur setzt den Frequenzzähler zurück und stellt damit sicher, dass das nächste Resultat einen gültigen Wert hat. Sie kann oder sollte z.B. nach SetFreqCountMode benutzt werden.
Procedure *FreqCountRestart;*

GetTimeCounter, GetTimeCounter2
Die Funktion GetTimeCounter liefert als Ergebnis (16bit Wert) der letzten Zeit Messung zurück. Das Ergebnis muss, abhängig von dem eingestellten Modus, entsprechend interpretiert werden, d.h. das Ergebnis ist eigentlich ein Festkomma Wert, dessen Kommastelle abhängig von dem eingestellten Modus ist. Im Overflow Fall wird der Wert $FFFF zurückgegeben. Der Aufruf dieser Funktion setzt auch die Semaphore auf 0.
Function *GetTimeCounter: word;*

GetTimeCounterP, GetTimeCounterP2
Die Funktion GetTimeCounterP liefert als Ergebnis (16bit Wert) die reinen Counter Werte der letzten Zeit Messung zurück. Die Werte Count1 und Count2 sind die gezählten Impulse während der „high“ bzw. „low“ Zeit der Periode. Damit lässt sich z.B. ein PWM Signal ausmessen. Im Overflow Fall wird ein false zurückgegeben. Der *FreqCountMode* muss auf *PulseMode* eingestellt sein.
Der Aufruf dieser Funktion setzt auch die Semaphore auf 0.
Function *GetTimeCounterP(**var** Count1, Count2 : word): boolean;*

Achtung:
Diese beiden Funktionen sollten nur verwendet werden, wenn auch der Zeitmesser Modus eingestellt ist. Ansonsten muss die Funktion **GetFreqCounter** benutzt werden.

Programm Beispiel:
ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\FreqCounter**

3.43 FreqCount Treiber – Frequenzzähler XMega

In manchen Applikationen wird eine Funktion gebraucht, die eine externe Frequenz feststellen bzw. auszählen kann. Diese Implementation benutzt einen der bis zu acht 16bit Timer (TimerC0 .. TimerF1) als Impuls Zähler. Beim Frequenz Zähler wird die Torschaltung wird durch den SystemTick erledigt. Das spart Resourcen in der CPU. Die Torschaltung lässt sich auf !00sec, 10sec, 1sec, 100msec oder 10msec einstellen. Das Ergebnis der Messung ist ein 16 Wert.

Der Nachteil bei der Verwendung des SystemTicks ist die etwas reduzierte Genauigkeit des Resultats und gewisse Einschränkungen bei der Wahl des SysTicks. Die Genauigkeit liegt hier bei 0.1%..0.2% abhängig vom Prozessor Clock und dem SysTick Wert. Hierbei gilt: Je höher der Clock und je kleiner die Zeit des SysTicks desto besser, in gewissen Grenzen. Optimale Ergebnisse erzielt man mit einem Clock >= 16MHz und einem Tick von 5msec und der ***ADJ*** Option. Weiterhin ist mit einem Jitter bei Interrupts zu rechnen.

Einschränkungen gibt es beim SysTick. Durch die notwendigen Torzeiten gilt folgendes:

Der SysTick muss 2msec, 5msec or 10msec sein. 5msec bevorzugt!

<u>100Hz Bereich</u>			
Auflösung	0.01Hz		
Max Frequenz	655.35Hz	=	0.65535kHz
Torzeit	100sec		
<u>1kHz Bereich</u>			
Auflösung	0.1Hz		
Max Frequenz	6553.5Hz	=	6.5535kHz
Torzeit	10sec		
<u>10kHz Bereich</u>			
Auflösung	1Hz		
Max Frequenz	65535Hz	=	65.535kHz
Torzeit	1sec		
<u>100kHz Bereich</u>			
Auflösung	10Hz		
Max Frequenz	655350Hz	=	655.35kHz
Torzeit	100msec		
<u>1MHz Bereich</u>			
Auflösung	100Hz		
Max Frequenz	6553500Hz	=	6.5535MHz
Torzeit	10msec		

Die zu messende Frequenz muss an jedem Pin der PortsA..PortF angeschlossen werden.
Einer der 8 Event Channels wird gebraucht.

Im folgenden bezeichnet ein ***XX*** einen der Timer C0..F1.

3.43.1 Import & Define FreqCount

Import *SysTick, FreqCount_C0, FreqCount_C1, FreqCount_D0, …;*

Importiert einen oder mehrere Frequenz Zähler. Der Import von FreqCount_*XX* importiert auch automatisch diverse Bibliotheksfunktionen. Es muss auch der SysTick importiert werden.

Der Treiber benutzt einen der internen 16bit Timer, TimerC0..TimerF1, falls vorhanden.

Define

```
//>> CPU          = 32MHz, PeripherX4=32MHz, PeripherX2=32MHz
OSCtype          = int32MHz,
PLLmul           = 4,
prescB           = 1,
prescC           = 1;
SysTick          = 5, adj;              {msec} // ,adj
StackSize        = $0032, iData;
FrameSize        = $0064, iData;

FreqCount_D0     = PortD, 0, PullUp;    // clock input port/pin, PullUp, PullDown, none
FreqCountEv_D0   = 0;                   // event channel
```

SysTick
Der SysTick muss folgende Werte haben: 2msec, 5msec, oder 10msec. Alle anderen Werte werden zurückgewiesen.

FreqCount_XX
Bestimmt den Eingang des Timers, Clock input Port, Pin und pullup. Die PortA..PortF können verwendet werden und jeder Pin (0..7) in diesen Ports.

FreqCountEv_XX
Bestimmt den Event Channel für diesen Timer (0..7).

3.43.2 Funktionen, Prozeduren, Typen

Type *tFreqCountRange*
Der Import des Treibers stellt den Typ tFreqCountRange zur Verfügung. Dieser Aufzählungstyp wird zur Einstellung der Torzeit des Zähler benötigt. Folgende Torzeiten bzw. Auflösungen sind zugeordnet:

TFreqBaseNone		
TFreqBase100Hz	Frequ = 0.01Hz...655.35Hz	gate time = 100sec
TFreqBase1kHz	Frequ = 0.1Hz...6.5535kHz	gate time = 10sec
TFreqBase10kHz	Frequ = 1Hz...65.535kHz	gate time = 1sec
TFreqBase100kHz	Frequ = 10Hz...655.35kHz	gate time = 100msec
TFreqBase1MHz	Frequ = 100Hz...6.5535MHz	gate time = 10msec

Die Definition von tFreqCountRange ist:

```
Type tFreqCountRange = (TFreqBaseNone, TFreqBase100Hz, TFreqBase1kHz, TFreqBase10kHz,
                        TFreqBase100kHz, TFreqBase1MHz);
```

Var *FreqCountSema_XX*
Diese Semaphore wird bei jedem neuen Messergebnis um eins inkrementiert. Nützlich nicht nur bei MultiProzessing sondern auch um bei normalen Anwendungen Rechenzeit zu sparen.

Procedure *SetFreqCountRange_XX(range : tFreqCountRange);*
Mit dieser Prozedur wird die Torzeit des Zählers eingestellt. Diese Einstellung sollte als erstes erfolgen. Der Parameter **range** ist die gewünschte Auflösung. Achtung: zumindest der erste Messzyklus nach diesem Funktionsaufruf liefert falsche Daten. Es werden alle internen Daten zurückgesetzt.

Function *GetFreqCountRange_XX : tFreqCountRange;*
Die Funktion GetFreqCountMode liefert als Ergebnis den aktuellen Bereich des Zählers zurück

Function *GetFreqCounter_XX: word;*
Die Funktion GetFreqCounter liefert als Ergebnis (16bit Wert) der letzten Frequenz Messung zurück. Das Ergebnis muss, abhängig von dem eingestellten Modus, entsprechend interpretiert werden, d.h. das Ergebnis ist eigentlich ein Festkomma Wert, dessen Kommastelle abhängig von dem eingestellten Modus ist. Im Overflow Fall wird meistens der Wert $FFFF zurückgegeben. Der Aufruf dieser Funktion setzt auch die Semaphore auf 0.

Function *GetFreqCountOvrFlow_XX: byte;*
Die Funktion GetFreqCountOvrFlow liefert als Ergebnis einen evtl. vorhandenen Überlauf der letzten Messung zurück. Dieser Wert ist normalerweise NULL. Wenn nicht, ist beim Frequenz Zähler Modus die Torzeit zu lang und die Frequenz zu hoch für die aktuelle Einstellung. Der Aufruf dieser Funktion setzt den Overflow zurück.

Procedure *FreqCountRestart_XX;*
Diese Prozedur setzt den Frequenzzähler zurück und stellt damit sicher, dass das nächste Resultat einen gültigen Wert hat. Es werden alle internen Daten zurückgesetzt.

Ein Beispiel ist in der Directory **..\E-Lab\AVRco\Demos\XMega_FreqCount**

3.44 RFID125 Receiver (*z.Zt. nur XMega*)

RFID **R**adio**F**requency**ID**entification ist sehr weit verbreitet um Waren, Tiere, Personen etc. eindeutig identifieren zu können. Auch für Zugangskontrollen, elektronischen SchLössern, mit z.B. Schlüssel Anhängern(Keyfob) oder Plastik Karten wird RFID häufig verwendet. Es gibt hierbei mehrere Arten bzw. Verfahren, die sich im wesentlichen durch die verwendete Trägerfrequenz unterscheiden. Im vorliegenden Treiber wird die am meisten genutzte 125kHz Variante unterstützt.

Die in den passiven 125kHz Transpondern gespeicherte binäre Nummer ist insgesamt 64bit lang. Die nutzbare Datenlänge ist dabei nur 40bit bzw. 5 Byte gross, wobei das MSB die ID des Transponder Hersteller ist. Die verbleibenden 4 Bytes eine einzigartige Nummer die weltweit nur einmal vorhanden ist. Es bestehen somit total 2^40 IDs zur Verfügung.

Ein RFID125 Receiver benötigt eine extrem gute Antenne (Spule) und einen empfindlichen Empfänger Chip. Beide Teile sind preiswert im Handel erhältlich. Es lohnt sich allerdings nur bei grossen Stückzahlen den Receiver komplett selbst zu bauen. Der AVRco Treiber basiert deshalb auf der aktiven Antenne TowiTek TWT2021 (Conrad Electronic). Dieses Bauteil hat ein async/serial Ausgang mit 9600Bd, 8N1 und arbeitet von 3.3V bis 5V. Bei einem gültigen Transponder Signal werden alle 200msec ein 5Byte Packet abgegeben.

Die Empfindlichkeit bzw. die mögliche maximale Distanz zwischen Transponder und Antenne hängt stark von der Betriebsspannung der Antenne ab. Bei 3.3V sind das ca. 1cm und bei 5V sind es ca. 5cm.

Da nur der Rx Kanal des UARTs benutzt wird, kann der Tx-Pin als normaler Input-Output Pin verwendet werden.

Imports
Wie beim AVRco System üblich, muss der Treiber importiert und definiert werden.
Der SysTick wird fúr Timeouts benötigt.

```
Import     SysTick, RFID125;
```

Defines
Die ser. Schnittstelle, die Baudrate und die Datenlänge muss bestimmt werden:

```
Define   …
  SysTick       =  10;                    // 5msec
  RFID125       = SerPort0, 9600, 5;      // SerPort, Baudrate, Buffersize
XMega
  RFID125       = SerPortD1, 9600, 5;
```

Treiber Variablen
Der Treiber exportiert ein Array mit den 5 ID Bytes:

```
Var
  RFIDbuff     : array[0..Buffersize-1] of byte;
```

Funktionen
Es gibt nur eine Funktion. Diese zeigt an ob eine gültige ID gelesen wurde. Durch Aufruf dieser Funktion wird das interne Ready zurückgesetzt.

```
function  RFIDready   : boolean;
```

Ein Beispiel ist in der Directory **..\E-Lab\AVRco\Demos\XMega_RFID**

3.45 RC5 Decoder/Encoder Treiber

Es gibt eine Vielzahl Möglichkeiten, wie zwei Einheiten (Prozessoren, Steuerungen etc) miteinander kommunizieren können. Wenn keine Kabelverbindung hergestellt werden kann oder darf, so gibt es eigentlich nur noch die Funkstrecke, Ultraschall und die Infrarot Strecke. Funk scheidet normalerweise aus Kosten Gründen aus, Ultraschall wird nicht mehr verwendet.
Also bleibt im Normalfall eigentlich nur noch Infrarot übrig. Infrarot ist relativ störsicher und mit entsprechender Sendeleistung braucht oft auch nicht einmal eine direkte Sichtverbindung zwischen Sender und Empfänger bestehen. Es genügen auch Reflexionen an den Wänden.

Das hier verwendete Protokoll **RC5** kommt aus dem TV-Fernsteuer Bereich und arbeitet mit einer Träger Frequenz von 36kHz. Im **Standard Mode** besteht ein Telegram aus 2 Startbits, einem sog. Toggle Bit, 5Bits Adresse und 6Bits Kommando. Im **Extended Mode** besteht das Telegramm aus 1 Startbit, CMDbit6, toggle bit, 5Bits Adresse und weiteren 6Bits Kommando, wobei hier das Kommando 7bits hat. Die Übertragung eines Telegramms dauert ca. 25msec.

Standard Mode
S1 S2 TG A4 A3 A2 A1 A0 C5 C4 C3 C2 C1 C0

Extended Mode
S1 C6 TG A4 A3 A2 A1 A0 C5 C4 C3 C2 C1 C0

3.45.1 Receiver

Der Receiver kann an jeden beliebigen Input-Pin angeschlossen werden. Der Treiber arbeitet hier im Polling Verfahren. Mit dem Define kann die Polarität der Rx-Impulse eingestellt werden.
Negative = Ruhezustand log. 1, Impulse log. 0 default Einstellung
Positive = Ruhezustand log. 0, Impulse log. 1

Defines für den Receiver

```
Define   RC5RXPORT = PinReg, PinNum, polarity    //polarity ist optional
         RC5mode   = rc_7bit;                    //rc_6bit = default
```

3.45.2 Transmitter

Der Transmitter muss die 36kHz exakt einhalten, deswegen hier ein Timer zwingend ist. Da in fast allen AVRs nur der Timer1 geeignet ist, wird dieser verwendet. Wenn vorhanden, kann auch der Timer3 gewählt werden. Bei ***XMegas*** kann jeder der 16bit Timer verwendet werden. Die Sendediode bzw. deren Leistungstreiber muss immer an den OC1A (OC3A), beim XMega an den gewählten OCxx Timer Ausgang angeschlossen werden. Als Option kann die Träger Frequenz auch zwischen 30kHz und 40kHz eingestellt werden.

Defines für den Transmitter

```
Define
     RC5TXPORT   =  Timer1, polarity, carrier;      //Timer und carrier Frequenz ist optional
     RC5TXPORT   =  positive;
     RC5TXPORT   =  negative;
     RC5TXPORT   =  positive, 38;
     RC5TXPORT   =  Timer1, positive;
     RC5TXPORT   =  Timer3, negative, 36;
     RC5mode     =  rc_7bit;                        //rc_6bit = default
XMega
   RC5TXPORT    = Timer_C0, OCA, negative;      // Timer, Output Comp, pulse polarity, carrier
   Timer_C0, Timer_C1…, OCA, OCB, OCC or OCD
    Hier ist die Angabe von Timer und OCxx Pflicht!
```

Es wird kein Interrupt verwendet.

Imports
Der Treiber (Rx, Tx oder beide) muss, wie beim AVRco üblich, importiert werden.

```
Import SysTick, RC5Rxport, ..;
oder
Import SysTick, RC5Txport, ..;
oder
Import SysTick, RC5Txport, RC5Rxport, ..;
```

Für den Receiver Teil muss das gewünschte InputPort (PINx) und der PortPin definiert werden.
Für den Transmitter muss die Treiber Polarität angegeben werden:

Defines

```
Define   ProcClock     = 8000000;          {Hertz}
         SysTick       = 10;               {msec}
         StackSize     = $0010, iData;
         FrameSize     = $0010, iData;
         RC5TXPORT     = negative, 36;     { opt Timer, pulse polarity, opt carrier freq}
         RC5RxPort     = PinD, 7;          {PinD, Portpin 7, opt polarity}
         // RC5RxPort  = PinD, 7, negative; {negative or positive pulse}
         RC5mode       = rc_7bit;          // rc_6bit = default
```

3.45.3 Funktionen Receiver

***Function** RecvRC5 (**var** rxAdr : byte; **var** rxCmd : byte) : boolean;*
Diese Funktion liefert ein false nach einem Timeout von 130msec. Die beiden Variablen werden dabei nicht verändert. War die Funktion erfolgreich, wird ein true zurückgegeben und die beiden Variablen sind mit den empfangenen Werten besetzt. Das Bit6 (ext Mode Bit7) in rxCMD enthält dabei den sog. Toggle Wert.

RxAdr = 0 0 0 A4 A3 A2 A1 A0

RxCmd im Standard Mode
0 TOG CMD5 CMD4 CMD3 CMD2 CMD1 CMD0

RxCmd im Extended Mode
TOG CMD6 CMD5 CMD4 CMD3 CMD2 CMD1 CMD0

3.45.4 Funktionen Transmitter

***Procedure** SendRC5 (**const** txAdr : byte; **const** txCmd : byte);*
Diese Prozedur schickt die beiden Bytes über den IR-Sender ab. Das Toggle Bit6 (ext mode bit7) in txCMD muss die Anwendung nach Bedarf setzen.

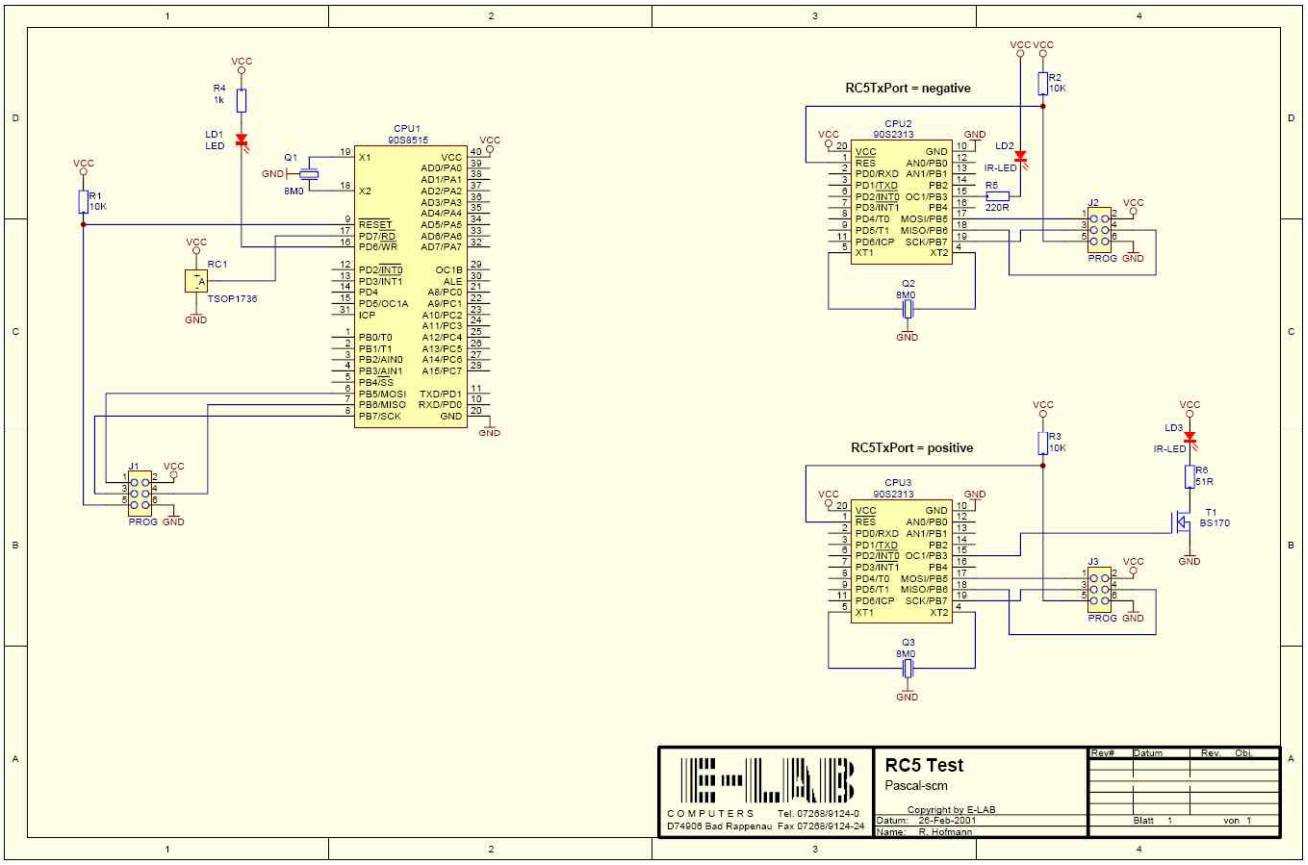
TxAdr = 0 0 0 A4 A3 A2 A1 A0

TxCmd im Standard Mode
0 TOG CMD5 CMD4 CMD3 CMD2 CMD1 CMD0

TxCmd im Extended Mode
TOG CMD6 CMD5 CMD4 CMD3 CMD2 CMD1 CMD0

Achtung:
Zu viele oder zu lange Interrupts während des Sendens oder Empfangens können das Protokoll stören, so dass evtl. das Telegramm ungültig wird. Bei ***MultiTasking*** muss unbedingt der Scheduler gesperrt werden.

Beispiele und Schaltung:
zwei AVR Beispiele befinden sich im Verzeichnis **..\E-Lab\AVRco\Demos\RC5**
zwei Xmega Beispiele befinden sich im Verzeichnis **..\E-Lab\AVRco\Demos\Xmega_RC5**



Schaltplan **RC5**

3.46 SHT11 Temperatur und Feuchte Sensor Treiber

Allgemeines
Raumtemperatur und Luftfeuchtigkeit sind die zwei wesentlichen Faktoren des Raumklimas. Die Erfassung dieser Werte sind z.B. für Klima Anlagen und Heizungsregelungen sehr wichtig. Aber auch reine Messwert Anzeige ist ein wichtiges Thema.

Probleme bereitet hierbei immer die relative Luftfeuchtigkeit. Bisherige Sensoren waren extrem ungenau und waren sehr Bauteile aufwändig. Vor allem die Genauigkeit und die Ansprechzeit liessen sehr zu wünschen übrig.

Weiterhin benötigen herkömmliche Sensoren Op-Amps und Referenz Bauteile sowie eine Linearisierung entweder in Hardware oder Software.

Mit den neuen intelligenten Sensoren von Sensirion lassen sich die beiden Faktoren mit extrem guter Präzision preisgünstig messen. Die Linearisierung findet im Sensor selbst statt. Bei der Temperatur ist der Systemfehler bei Raumtemperatur ca. +/- 0.5grad Celsius. Der Feuchte Fehler beträgt +/-3.5%, was ein sehr guter Wert ist. Die Messergebnisse sind sehr stabil. Durch enger tolerierte Versionen ist eine noch bessere Genauigkeit erreichbar. Die Präzision lässt sich durch bestimmte linearisierungs Algorithmen noch weiter verbessern.

Der Temperatur Bereich des Sensors reicht von –40grad bis +120grad Celsius.
In den Endbereichen ist der Fehler max +/-3grad Celsius

Der Feuchte Bereich geht von 0% bis 100%RH. In den Endbereichen ist der Fehler max. +/-5%RH

Einführung SHT11drvr
Der SHT11 Sensor arbeitet zwischen 2.5 und 5.5Volt. Die Verbindung zwischen Sensor und CPU ist eine Zweidraht Leitung, ähnlich dem I2C Interface, allerdings nicht kompatibel dazu.

Die Kommunikation erfolgt durch zwei Steuerleitungen (SCK und DATA), die an beliebige Port Pins der CPU angeschlossen werden können. Die DATA Leitung muss allerdings bi-direktional arbeiten können (DDR Register) und muss mit einem Pull-Up Widerstand von 2..3kOhm versehen sein.

Die Daten werden durch ein Bitserielles Protokoll gelesen. Nach dem Start Kommando erfolgt die Wandlung im Sensor, dessen Ende durch ein Low Signal auf der Datenleitung vom Sensor signalisiert wird.

Die Konvertierungszeit für ein Signal beträgt je nach Betriebsspannung und gewählter Auflösung zwischen 10 und 250msec. Es ist deshalb in der Praxis nicht sinnvoll nach dem Kommando bis zu 250msec die Datenleitung zu pollen. Es gibt in der Zwischenzeit bestimmt besseres zu tun. Da der Sensor voll statisch aufgebaut ist hält er seine Datenleitung so lange low, bis der erste Read-Clock Impuls vom den CPU kommt.

Dadurch ist es möglich, die Wandlung zu starten, dann andere Jobs zu erledigen und nach einer gewissen Zeit den Status der Datenleitung auf READY abzufragen (SHT11ConvState). In MultiTasking Applikationen setzt der Treiber eine Semaphore auf „1“, wenn das Ergebnis bereitsteht. Dadurch kann sich z.B. der verarbeitende Prozess mit „WaitSema“ schlafen legen. Er wird dann durch den Scheduler wieder aufgeweckt, wenn das Ergebniss ausgelesen werden kann.

Die Temperatur und die Feuchte Lese Funktionen sind deshalb jeweils in zwei Teilen implementiert, einem Konvertierungs Start-Teil (SHT11startTemp und SHT11startHum) und einem Lese-Teil (SHT11getTemp und SHT11getHum). Bevor z.B. „SHT11getTemp“ aufgerufen wird, muss entweder immer der Status der Konvertierung abgefragt werden mit „SHT11ConvState“ oder es muss mit „WaitSema“ gearbeitet werden.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\SHT11**
In diesem Verzeichnis befindet sich auch ein Helper File, das zusätzliche Linearisierungs Funktion enthält.

Imports
Wie beim AVRco System üblich, muss der Treiber importiert und definiert werden.
Der SysTick wird nicht benötigt.

```
Import    SysTick, SHT11drvr;
```

Defines
Der Status Mode und die zwei Port Pins müssen definiert werden.

```
Define  ProcClock     = 8000000;              // 8Mhz clock
        SysTick       = 5;                    // 5msec
        SHT11drvr     = polled [, Delay][, crc];  // polling only
        // SHT11drvr  = SysTickChecked;       // polling + semaphore
        SHT11clk      = PortD, 5;             // port and pin for clock
        SHT11dat      = PortD, 6;             // port and pin for data
```

SHT11drvr
Definiert den Polling Mode. Mit „SysTickChecked“ wird automatisch eine Semaphore implementiert, die mit WaitSema einen Prozess steuern. Der Polling mode mit „SHT11ConvState” ist trotzdem möglich.. Der optionale Parameter „Delay“ legt die Datentransfer Rate fest (default 0, min 0, max 255).
Der optionale Parameter „CRC“ bestimmt ob das CRC Byte aus dem SHT11 ausgelesen und gespeichert werden soll.

SHT11clk
Definiert das für den Clock benutzte Port und Pin. Das Port muss nicht bi-direktional sein.

SHT11dat
Definiert das für DATA benutzte Port und Pin. Das Port **muss** bi-direktional sein.

3.46.1 Variablen

Wenn der „SysTickChecked” Mode definiert wurde, wird eine Semaphore implementiert, die bei Prozessen in Zusammenhang mit WaitSema benutzt werden kann. Die Conversion Start Funktionen setzen diese Semaphore zurück. Im SysTick wird die Semaphore auf „1“ gesetzt, wenn der Sensor mit der Wandlung fertig ist.

```
Var SHT11sema : Semaphore;
```

Wenn die Option “CRC“ im Define aktiviert wurde, schliesst eine Lese Operation aus dem SHT11 immer auch das CRC Byte mit ein. Dieses Byte wird in der Speicherstelle SHT11crc abgelegt.

```
Var SHT11crc : byte;
```

Achtung:
Wenn eine Wandlung läuft, darf kein weiterer Zugriff auf den Sensor erfolgen ausser **„SHT11ConvState“.**

3.46.2 Funktionen

SHT11synchronize
Setzt das 2-Draht Interface zurück. Wird automatisch beim Programm start aufgerufen.

```
Procedure SHT11synchronize;
  SHT11synchronize;
```

SHT11ConvState
Gibt ein true zurück, wenn nach einem Conversion Start der Sensor die Daten bereit hat.

```
Function SHT11ConvState : boolean;
  repeat
  until SHT11ConvState;
```

SHT11startTemp
Startet die Temperatur Wandlung.

```
Procedure SHT11startTemp;
  SHT11startTemp;
```

SHT11getTemp
Liest nach erfolgreicher Wandlung das Messergebnis aus dem Sensor.

```
Function SHT11getTemp : word;
  Ww:= SHT11getTemp;
```

SHT11startHum
Startet die Feuchte Wandlung.

```
Procedure SHT11startHum;
 SHT11startHum;
```

SHT11getHum
Liest nach erfolgreicher Wandlung das Messergebnis aus dem Sensor.

```
Function SHT11getHum : word;
  Ww:= SHT11getHum;
```

SHT11getStatus
Liest das Status/Steuer Register des Sensors aus. Kommt hier ein $FF zurück, ist der Sensor defekt oder nicht angeschlossen. Diese Prüfung sollte unbedingt erfolgen.

```
Function SHT11getStatus : byte;
  If SHT11getStatus = $ff then
          SensorFailed ...
  endif;
```

SHT11setStatus
Diverse Sonderfunktionen können ausgeführt werden, z.B. Umschaltung der Auflösung, Heizung ein etc.

```
Procedure SHT11setStatus (s : byte);
  SHT11setStatus($01);                  // reduce solution to 8/12bit
```

SHT11softReset
Das Sensor interne Status/Steuer Register auf seine Default Werte zurückgesetzt.

```
Procedure SHT11softReset;
  SHT11softReset;                       // clear status reg to default values
```

3.47 Sound Generator und Treiber

Die Kommunikation zwischen Steuerung/Elektronik und Umgebung ist oft ein wichtiger Teil. Neben Displays, Tastaturen und Computerschnittstellen ist auch manchmal ein akustisches Signal notwendig und sinnvoll.

Diesem Bedarf kommt der hier vorgestellte Sound Treiber entgegen. Er kann zwar keine Musik direkt abspielen, ist aber in der Lage fest vorgegebene Töne und Tonfolgen auszugeben. Weiterhin kann er variable Frequenzen eine vorgegebene Zeit ausgeben. Als Wiedergabe Gerät genügt oft ein kleiner Piezo Beeper, der über einen ausreichend grossen Kondensator (>1uF) an den gewählten Port Pin angeschlossen werden kann. Wird eine grössere Lautstärke gewünscht, so ist ein externer Audio Verstärker erforderlich.

Imports
Der Treiber muss, wie beim AVRco üblich, importiert werden. Er benötigt dazu auch den Import von SysTick:

```
Import SysTick, BeepPort, ..;
```

Defines
Der gewünschte Port und der PortPin wird mit Define vorgegeben:

```
Define
    ProcClock   = 8000000;        {Hertz}
    SysTick     = 10;             {msec}
    StackSize   = $0020, iData;
    FrameSize   = $0030, iData;
    BeepPort    = PortB, 0;
```

3.47.1 Funktionen

Es werden verschiedene Sound Funktionen zur Verfügung gestellt:

***Procedure** BeepSiren (**const** mode : byte; repTimes : byte);*
Der Aufruf dieser Prozedur erzeugt einen Sirenzyklus. Die Art der Sirenen wird durch den Parameter mode eingestellt (0 oder 1). Die Anzahl der Sirenentöne wird mit repTimes angegeben.

***Procedure** BeepClick;*
Erzeugt ein kurzes Klicken.

***Procedure** BeepOutLH;*
***Procedure** BeepStepLH;*
Diese Prozeduren erzeugen eine kurze aufsteigende Tonfolge mit 3 bzw. 5 Tönen

***Procedure** BeepOutHL;*
***Procedure** BeepStepHL;*
Diese Prozeduren erzeugen eine kurze absteigende Tonfolge mit 3 bzw. 5 Tönen.

***Procedure** BeepChirpH (repTimes : byte);*
Diese Prozedur erzeugt eine sehr kurze absteigende hohe Tonfolge (Chirp). Die Anzahl der Tonfolgen wird mit repTimes angegeben.

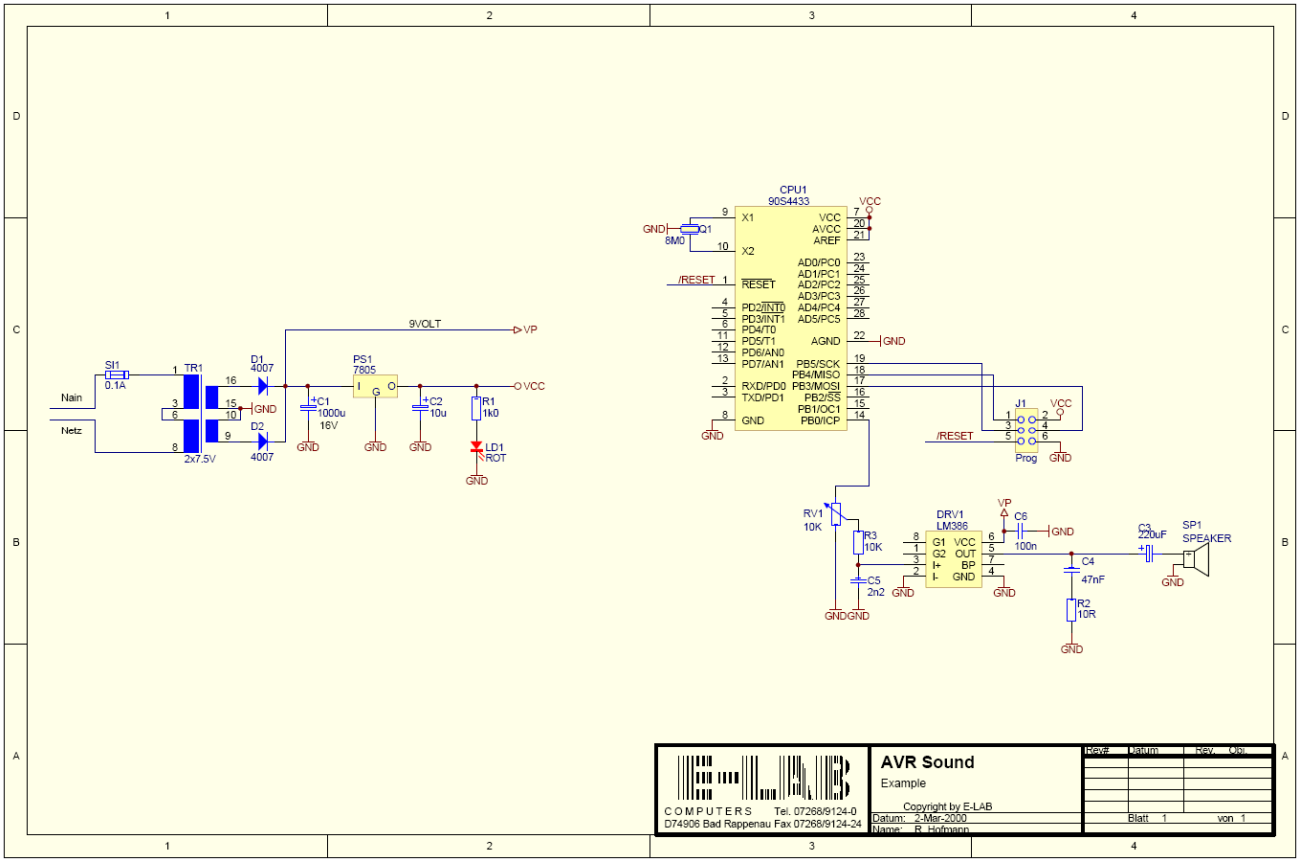
***Procedure** BeepChirpL (repTimes : byte);*
Diese Prozedur erzeugt eine sehr kurze absteigende niedrige Tonfolge (Chirp). Die Anzahl der Tonfolgen wird mit repTimes angegeben.

Procedure *BeepOutErr;*
Diese Prozedur erzeugt einen kurzen schnarrenden Ton.

Procedure *BeepOut (Frequ : word; ticks : byte);*
Diese Prozedur erzeugt einen Ton mit der angegeben Frequenz. Die minimale Frequenz ist 40Hz, die maximale ist 10kHz. Je nach Prozessor Clock kann es sein, dass die maximale oder minimale Frequenz nicht erreicht werden kann. Mit 16MHz sind beide Grenzen legal. Die Dauer des Tons wird den Wert Ticks bestimmt (Dauer in SysTicks).

Beispiel und Schaltung:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Sound**



Schaltplan **Sound**

3.48 SysLEDblink

Allgemeines
Ansteuerung von LEDs (Leuchtdioden) ist eine typische Aufgabe bei embedded Systemen. Normalerweise ist das sogar die erste Übung eines Anfängers, da es ein simples Programm ist und sofort Resultate sichtbar werden. Ein Port wird auf Ausgang programmiert und einzelne Bits dieses Ports werden auf low oder high gesetzt. Fertig.

Sollen LEDs blinken, kommt sofort ein grosses Problem auf, das auch dem Profi manchmal Schwierigkeiten bereitet. Da die LEDs auch während des Programm Ablaufs immer rhythmisch blinken müssen und eine verlängerte oder verkürzte On oder OFF Phase sehr störend wirkt, benutzt man dazu meistens einen freien Timer. In dessen Interrupt wird jetzt der Blink-Job erledigt. Aber was ist, wenn kein Timer mehr dafür frei ist? Oder wenn der Timer für zukünftige Erweiterungen reserviert bleiben soll? Dann wird es sehr schwierig.

Da jedoch im AVRco System meistens auch der ***SysTick*** importiert werden muss, bietet sich dieser dafür an. Man kann in der CallBack Funktion ***OnSysTick*** das LED Handling einbauen. Allerdings sollte dies in Assembler geschehen, um eine unerwünschte Verlängerung der Interrupt Sperrzeiten durch den Handler zu vermeiden. Weiterhin ist hier auch das Problem „Register Rettung“ nicht zu unterschätzen.

Die o.a. geführten Argumente machen es deshalb sinnvoll einen LED Blinker in das System zu integrieren. Dieser Treiber wird in den SysTick eingehängt und erledigt von dort aus völlig transparent seinen Job.
Bitte beachten, dass dieser Treiber ein „heavy duty job“ im SysTick ist.

Einführung SysLEDblink
Der Treiber dient zur Kontrolle von bis zu 8 LEDs, wobei jede LED separat gesteuert werden kann.
Die Funktionen sind:

1. LED ein
2. LED aus
3. alle LED ein
4. alle LED aus
5. LED blink ein
6. LED blink aus
7. alle LED blink ein
8. alle LED blink aus
9. Blinktimer Setup in SysTicks
10. LED x-mal blinken
11. Treiber On/Off
12. LED 1xblink

Die einzelnen LEDs können auf beliebige Ports und Pins verteilt sein, auch der allgemeine Speicher ist möglich, wenn dieser R/W ist. Ein simples Latch als Port ist nicht möglich.

Der Blink Rhythmus ist in weiten Grenzen wählbar. Die Polarität der LEDs ist einstellbar, z.B. aktiv LOW oder aktiv HIGH.

Ein alternatives bzw. Wechsel-Blinken einzelner LEDs ist möglich indem der Grundzustand (On/Off) dieser LEDs unterschiedlich eingestellt wird. Wenn z.B. LED1 auf ON steht und LED2 auf OFF und bei beiden wird das Blinken eingeschaltet, so blinken diese beiden LEDs jetzt wechselseitig.

Bemerkung:
Eine Initialisierung der beteiligten Ports (DDRx etc) findet nicht statt. Es ist Aufgabe der Applikation die beteiligten Ports zu initialisieren. Dazu gehört auch, dass zur Laufzeit diese Einstellung nicht geändert werden sollte, da der Treiber hier keinerlei Überprüfung bzw. Unterstützung bietet.

Wenn die Funktion **SysLEDflashMsg** benutzt wird, muss auch der Message Mode importiert werden:
From *SysLEDblink* ***Import*** *LEDmessage;*

Wenn die Funktion **SysLEDflashOnce** benutzt wird, muss dies auch importiert werden:
From *SysLEDblink* ***Import*** *FlashOnce;*

3.48.1 Implementation

Imports
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.

```
Import  SysTick, SysLEDblink, ..;
From  SysLEDblink Import LEDmessage, FlashOnce, FastBlink;  // these are options
```

Defines
Den Blink Rhythmus in SysTick Einheiten
Die bis zu 8 Port Register und Portpins PORT, BIT, ACTIVE_LEVEL

```
Define   ProcClock      = 8000000;          {Hertz}
         SysTick        = 10;               {msec}
         StackSize      = $0030, iData;
         FrameSize      = $0030, iData;
         SysLEDblink    = 30;               {30*SysTick = 300msec}
         // alternative
         //SysLEDblink  = mSec300;          {10..1000 msec}
         SysLEDBlink0   = PortA, 0, high;   {LEDon = high level}
         SysLEDBlink1   = PortA, 1, low;    {LEDon = low level}
         SysLEDBlink2   = PortA, 2, low;
         SysLEDBlink3   = PortA, 3, low;
         SysLEDBlink4   = PortA, 4, low;
         SysLEDBlink5   = PortA, 5, low;
         SysLEDBlink6   = PortA, 6, low;
         SysLEDBlink7   = PortA, 7, low;
```

Achtung: zumindest *SysLEDBlink0* muss definiert werden! Es sollten auch keine Lücken in den Defines sein.

Eine Alternative zu dem SysLEDblinkX Defines ist auch eine Byte Variable im RAM statt der Port Bits:

```
SysLedPort     = @LEDram, $00;       // byte-var, polarity
```

Die dadurch erzeugte Variable enthält dann die Blink Bits, die z.B. an virtuelle oder auch I2C/TWI Ports ausgegeben werden kann.

Imports und Exports
Ein SysLEDblink Import importiert diverse Bytes, die für das System reserviert sind. Der Blink Handler wird in den SysTick eingeklinkt.

3.48.2 Funktionen

```
Procedure SysLEDon (b : byte);                    // b= 0..7 LED number

Procedure SysLEDonOff(b : byte; on : boolean);    //  b= 0..7 LED number

Procedure SysLEDallOn;

Procedure SysLEDoff (b : byte);                   // b= 0..7 LED number

Procedure SysLEDallOff;

Procedure SysLEDflashOn (b : byte);               // b= 0..7 LED number

Procedure SysLEDflashOnOff(b : byte; on : boolean); // b= 0..7 LED number

Procedure SysLEDflashAllOn;

Procedure SysLEDflashOff (b : byte);              // b= 0..7 LED number

Procedure SysLEDflashAllOff;
```

Procedure *SysLEDflashMsg (led, msg, rept : byte);* *// expects LEDmessage Import*

Procedure *SetSysBlinkTimer (t : byte);* *// t= 2..255*

Procedure *SysLEDfastBlink (fast : boolean);* *// enable/disable fast mode*

Procedure *SysLEDenable (ena : boolean);* *// enable/disable the driver*

Procedure *SysLEDflashOnce (b : byte);* *// b= 0..7 LED number*

Details

Procedure *SetSysBlinkTimer (t : byte);* *// t= 2..255 in SysTicks*
Normalerweise wird die Blinkrate (on/off Time) durch das Define ***SysLEDblink*** vorgegeben.
Die Blinkrate kann aber auch zur Laufzeit mit dieser Funktion geändert werden.

Procedure *SysLEDfastBlink (fast : boolean);* *// enable/disable fast mode*
Die Blinkrate kann hiermit auf schnell oder normal umgestellt werden ohne den Timer Wert zu ändern.

Procedure *SysLEDenable (ena : boolean);* *// enable/disable the driver*
Die Verwaltung der LEDs und deren Timer wird im SysTick durchgeführt. Das kostet natürlich Rechenzeit. Wenn keine Blink Funktionen gebraucht werden, kann das Blink Handling temporär komplett abgeschaltet werden.

Procedure *SysLEDflashMsg (led, msg, rept : byte);* *// expects LEDmessage Import*
Diese Funktion unterstützt z.B. das Blinken einer Fehlermeldung. Im Parameter ***led*** wird die zu benutzende LED bestimmt. Der Parameter ***msg*** bestimmt die Anzahl der Blinks pro Telegramm, wobei hier die Obergrenze bei 126 liegt. Der Parameter ***rept*** bestimmt, wie oft dieses Telegramm wiederholt werden soll. Der Wert “255“ bedeutet unendliche Wiederholung. Dann muss dieses Blinken bei Bedarf mit der Funktion ***SysLEDflashOff*** oder ***SysLEDflashAllOff*** abgebrochen werden.

Achtung:

es kann immer nur eine LED eine Message zu gleicher Zeit ausgeben werden.

Tip:

Man kann den einzelnen LEDs auch Namen geben.

Type *myLEDs = (LEDenter, LEDexit, LEDpower);*

Diese Enumeration kann jetzt mit den Funktionen verwendet werden, z.B.:

SysLEDflashOn (byte (LEDpower));

Es geht auch so:

```
Const
   LEDenter    : byte   = 0;
   LEDexit     : byte   = 1;
   LEDpower    : byte   = 2;
```

Diese Konstanten können jetzt mit den Funktionen verwendet werden, z.B.:

SysLEDflashOn (LEDpower);

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\SystemLEDs**

3.49 Line Printer Treiber LPTport

Auch bei Embedded Anwendungen muss manchmal ein Drucker angeschlossen werden um darauf Informationen, Messergebnisse oder andere Daten auf Papier ausgeben zu können. In den meisten Fällen besitzen solche Drucker ein serielles Interface. Wo das nicht der Fall ist, kommt dann oft das etwas betagte aber noch immer aktuelle Parallel-Interface, auch Centronics Interface genannt, zum Zuge.

Das Centronics Interface besteht aus einem 8bit Datenport und einem 8bit Kontrollport. Normalerweise steht das Datenport immer auf Output, während das Kontrollport sowohl Statusleitungen (Input) als auch Steuerleitungen (Input/Output) besitzt. Der LPTport Treiber implementiert diese Standard Konfiguration. Es werden aber auch Hilfsfunktionen zur Verfügung gestellt, die einen bi-direktionalen Betrieb erlauben.

Zum Betrieb des Treibers werden also immer 2 komplette 8bit Ports benötigt. Der Treiber unterstützt dazu sowohl die Verwendung von zwei bi-direktionalen Ports des AVRs als auch den Philips I2C Dual-Port Baustein PCA9555. Dieser kann sowohl mit dem Software I2Cport als auch mit dem TWIport des AVRco betrieben werden.

Die Betriebsart des LPTport Treibers wird durch dessen Import und Define festgelegt.

Imports
Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import SysTick, LPTport, ..;
```

Die Betriebsart AVRport, I2C oder TWI wird mit Define vorgegeben.

Beispiel für den Port Modus:

```
Define
    ProcClock   = 8000000;      {Hertz}
    SysTick     = 10;           {msec}
    StackSize   = $0010, iData;
    FrameSize   = $0010, iData;
    LPTport     = PortA, PortB; // DataPort, ControlPort
```

Wird der TWI Mode benutzt, dann muss auch der TWI Treiber importiert werden:

```
Import SysTick, TWImaster, LPTport, ..;     // TWInetMaster is also possible

Define
    ProcClock   = 8000000;      {Hertz}
    SysTick     = 10;           {msec}
    StackSize   = $0010, iData;
    FrameSize   = $0010, iData;
    TWIpresc    = TWI_BR400;
    LPTport     = TWI_I2C, $24; // $24 = TWIaddr
```

```
XMega
Import SysTick, TWI_C, LPTport, ..;     // TWI_D, TWI_E, TWI_F are also possible
Define
    OSCtype     = int32MHz, PLLmul=4, prescB=1, prescC=1;
    SysTick     = 10;               {msec}
    StackSize   = $0020, iData;
    FrameSize   = $0040, iData;
    TWIprescC   = TWI_BR400;
    LPTport     = TWI_C, $24;       // $24 = TWIaddr
```

Wird der Soft-I2C Mode benutzt, dann muss auch der I2Cport Treiber importiert werden:

```
Import SysTick, I2Cport, LPTport, ..;

Define
    ProcClock   =  8000000;        {Hertz}
    SysTick     =  10;             {msec}
    StackSize   =  $0010, iData;
    FrameSize   =  $0010, iData;
    I2Cport     = PortA;
    I2Cclk      =  0;              // bit0, porta
    I2Cdat      = 1;               // bit1, porta
    LPTport     = Soft_I2C, $24;   // $24 = I2Caddr
```

3.49.1 Funktionen und Typen

Der Treiber exportiert 2 Typen:

```
Type
  tLPTlines   = (lpStrobe, lpError, lpInit, lpSelect, lpACK, lpBusy, lpSelected, lpPaper);

  tLPTlineSet = BitSet of tLPTlines;
```

Der Aufzählungstyp ***tLPTlines*** entspricht exakt der Portbelegung des Control Ports.

Name	Bit #	dir	active	connector
lpStrobe	0	output	low	1
lpError	1	input	low	15
lpInit	2	output	low	16
lpSelect	3	output	low	17
lpACK	4	input	low	10
lpBusy	5	input	high	11
lpSelected	6	input	high	13
lpPaper	7	input	high	12

databit #	connector
0	2
1	3
2	4
3	5
4	6
5	7
6	8
7	9

Das BitSet ***tLPTlineSet*** beinhaltet obige 8 bits in einem Byte.

Funktionen

Procedure *LPTinit;*
Initialisiert den Printer durch die Init und die Select Leitung. Sollte vor jedem Druckvorgang aufgerufen werden.

Procedure *LPTreset;*
Gibt einen kurzen Impuls auf die Init-Leitung und setzt dadurch den Printer zurück.

Function *LPTstat : tLPTlineset;*
Gibt den Status des Control Ports zurück. Zu beachten ist dabei, dass das Ergebnis den Aktive Modus der einzelnen Leitungen berücksichtigt. Ist das ACK Leitung z.B. aktiv (=0) dann ist das lpAck Bit im Ergebnis gesetzt. Ist die lpBusy Leitung aktiv (=1) dann ist das lpBusy Bit im Ergebnis gesetzt.
Die Applikation sollte erst anfangen zu Drucken, wenn das ***lpSelected*** Bit aktiv und das ***lpPaper*** Bit inaktiv ist.

Procedure *LPTout (dat : byte);*
Dies ist die Druck Funktion. Die Funktion kehrt sofort ohne weitere Aktion zurück, wenn die ***lpSelected*** Leitung inaktiv ist. Ansonsten wartet sie bis ***lpBusy*** inaktiv ist und übergibt dann das Byte dem Drucker.

LPTinit;
Repeat until *(lpSelected* ***in*** *LPTstat);*
WriteLn *(LPTout, 'Hallo');*

Hilfs Funktionen

Die folgenden Funktionen und Prozeduren sind für den Printer Betrieb nicht notwendig und sind deshalb Spezial Implementationen wie z.B. bi-direktionalem Betrieb vorbehalten.

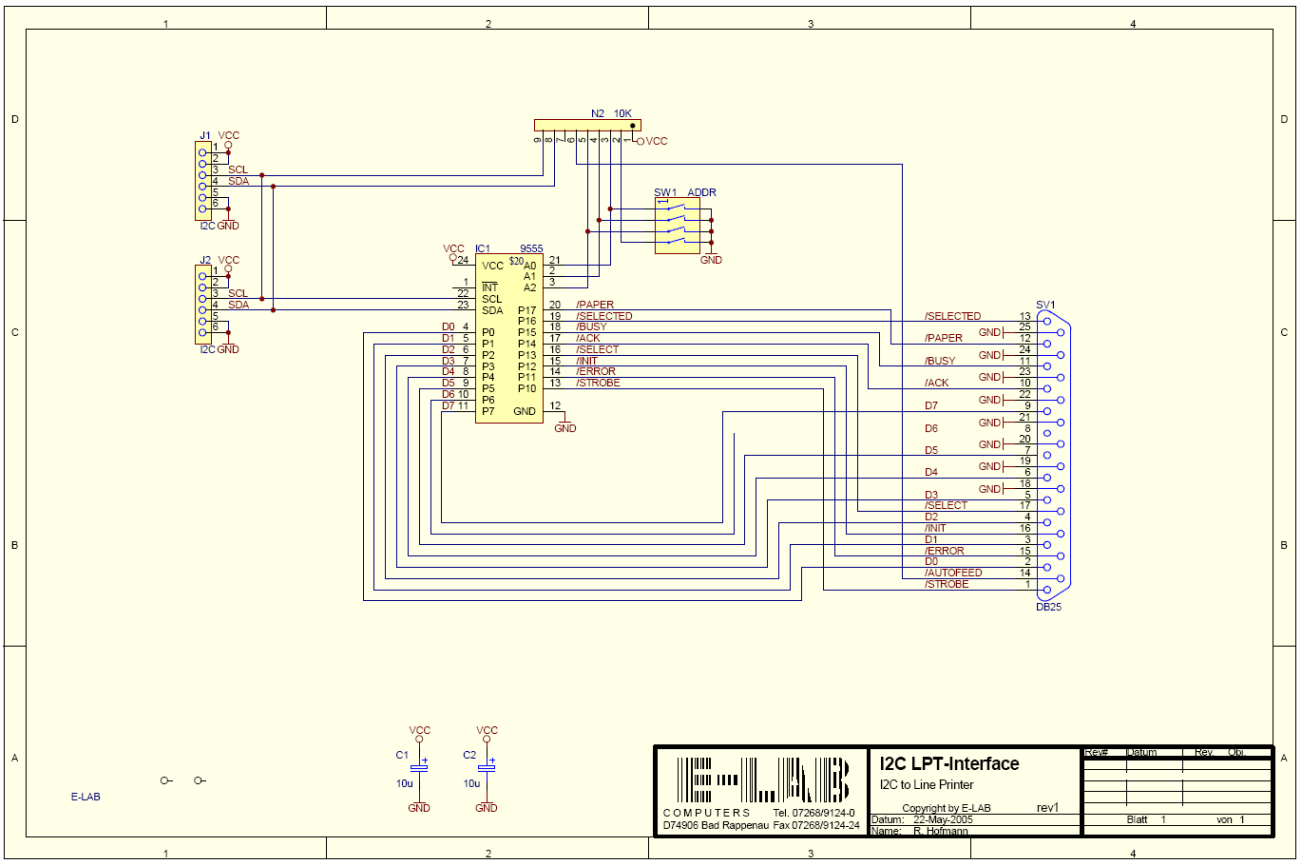
Procedure *LPTctrl (ctrl : tLPTlineSet);*
Steuert das Controlport bzw. die Steuerleitungen für den Printer.

Procedure *LPTdir (inp : boolean);*
Steuert die Datenrichtung des Datenports. True = output, false = input.

Function *LPTinp : byte;*
Liest das Datenport.

Beispiel und Schaltung:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\I2C_LPT**
ein Xmega Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_LPT**



Schaltplan **I2C_LPT**

3.50 Banking Port

In manchen Anwendungen wird es notwendig eine grosse Menge von Variablen in einem konstanten, schnellen Zugriff zu haben. Wenn die Datenmenge den normalen Adressraum der CPU überschreitet, hilft nur noch ein Filesystem. Dessen Komplexität, Geschwindigkeit und Implementation ist jedoch nicht ganz trivial. Ein einfacher Zugriff auf Variablen Namen ist in der Regel besser und schneller.

Zwischen dem linearen Adressraum des AVR und dem Filesystem liegt von den Möglichkeiten her gesehen, ein sogenanntes Banking System.

Hiermit kann eine theoretisch fast unbegrenzte Speicher Erweiterung implementiert werden, indem kleinere Teilbereiche des grossen externen Speichers in den normalen Adress Raum der CPU gemappt werden. Der externe Speicher wird dabei in gleich grosse Teile aufgeteilt. Eine Hardware schaltet ein Teil (bank) des Zusatz Speicher programmgesteuert in einen freien Speicherbereich der CPU.

Der Anwender kann alle Variablen Typen in solch eine Bank legen und kann voll transparent darauf zugreifen, ohne sich um die Bank selbst kümmern zu müssen
(Ausnahme: pointers to bank vars, siehe unten).

Um die verschiedenen Banks mit Variablen zu besetzen muss der User angeben in welcher Bank eine bestimmte Variable angesiedelt werden soll. Dies geschieht durch den vorangestellten Compiler Switch {$BDATA #} wobei # die gewünschte Bank Nummer bezeichnet.

In der vorliegenden Implementation können bis zu 16 banks mit jeweils 32kB definiert und verwaltet werden. Dies ergibt eine zusätzliche Speichergrösse von bis zu 512kBytes.

Das externe Device ist nicht auf ein Hardware Banking beschränkt. Durch das Device Treiber Konzept ist jedes externe Speichermedium adaptierbar. Der User muss 2 Hardware Driver implementieren um ein Byte zu lesen und zu schreiben, wobei das System eine relative Adresse (0..$7fff) und eine Banknummer (0..15) liefert.

Die Treiber müssen einen passenden Zugriff auf den Speicher generieren, zu dem diese beiden Parameter in geeigneter Weise miteinander kombiniert werden. Adress mapping und kombinieren der Bank Nummer mit der relativen Adresse in jeder denkbaren Art möglich. Deshalb könnte das Banking auch für Flashcards oder Floppies etc. eingesetzt werden.

Einschränkungen
Komplexe Arrays und Records werden nicht unterstützt (Arrays of records, arrays in records etc).

Strings:

Kein string **Concat** mit banked Strings in Source oder Destination wie:
st:= banked_string + 'abc';

Keine StringConversion mit banked Strings in Source oder Destination wie
int:= StrToInt (banked_string);
oder
banked_string:= IntToStr (num);

Imports
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.

Import *SysTick, BankPort, ..;*

Defines
die Anzahl BankPort banks in 32kByte pages

```
Define   ProcClock   = 8000000;      {Hertz}
         SysTick     = 10;           {msec}
         StackSize   = $0030, iData;
```

```
FrameSize   = $0030, iData;
BankPort    = 3;                {Bank Ports}
```

Belegung der Bank Pages
besetzte die Bank Pages mit Variablen mit dem vorangestellten Compilerswitch **{$BDATA x}**, wobei x die Bank Nummer für die nachfolgenden Definitionen vorgibt.

```
{$BDATA 0}
var
  bb0      : byte;
  ww0      : word;

{$BDATA 1}
var
  bb       : byte;
  ww1      : word;
```

3.50.1 Implementation

die "UserDevice BankDevIni" ist eine spezial Prozedur. Sie kann bestimmte Initialisierungen vornehmen.

```
UserDevice BankDevIni;        // Bank Device init
begin
  (* is called at System Init Time *)
  (* initialize Device Hardware *)
end;
```

die "UserDevice BankDevInp" ist eine spezial Funktion. Die relative Data Adresse wird in einem Word Argument und die Bank Nummer in einem Byte übergeben. Die Funktion muss das gelesene Byte zurückgeben.

```
UserDevice BankDevInp (bank : byte; adr : word) : byte;
begin
  ...
  return(xxx);
end;
```

die "UserDevice BankDevOut" ist eine spezial Prozedur. Die Bank Nummer wird im 1. Parameter als Byte übergeben. Die relative Data Adresse ist im 2. Argument = word. Der zu schreibende Byte Wert befindet sich im 3. Parameter.

```
UserDevice BankDevOut (bank : byte; adr : word; arg : byte);
begin
  ...
end;
```

Sonder Fälle:
Gebankte Variablen in Ausdrücken etc. die durch ihren Namen referenziert werden, werden genauso behandelt wie die Standard Variablen. Der Compiler weiss wo sie sich befinden und generiert einen „banked“ Zugriff. Da aber **pointer** immer 16bits breit sind, gibt es da keinerlei Information in welchen Speichertyp dieser Pointer zeigen soll. Deshalb impliziert ein Pointer grundsätzlich einen Zugriff in den linearen Adressraum der CPU von 0..$FFFF.

Der User kann einem Pointer die Adresse einer “banked” Variable zuweisen.
ptr:= @banked_var;
Wenn aber jetzt der Pointer de-referenziert wird wie z.B.
ptr^:= xxx;
gibt es keinerlei Information dass ein Bankbereich adressiert werden soll. Der User muss dem Compiler dies mitteilen durch die System Funktion
```
Function BankDevPtr (bank : byte; ptr : pointer) : pointer;
```

z.B.
BankDevPtr (bank#, ptr)^:= xxx;
xxx:= BankDevPtr (bank#, ptr)^;
Nun wird der Pointer umgeleitet in die Bank, die durch "bank#" angegeben wird und der Zugriff geht jetzt in diese Bank.

Ein anderer Sonderfall ist, wenn eine “banked” Variable an eine Prozedur oder Funktion übergeben wird mittels ihrer **Adresse**.

***Procedure** func_name (**var** bbbb : byte);*

wird aufgerufen mit:

Func_name (banked_var);

Die aufgerufene Prozedur/Funktion erwartet ein die Adresse eines Bytes und nicht eine Kopie des Bytes. Wenn das aufrufende Statement nun eine “banked” Variable übergibt, kann die gerufene Function nicht wissen, dass die übergebene Adresse in den „banked“ Bereich zeigt, wo die originale Variable sich befindet. Die übergebene Adresse kann als ein Pointer to Byte interpretiert werden. Im Gegensatz aber zu einem Pointer, der de-referenziert wird: **ptr^**, wird ein Var Parameter nur mit seinem Namen angesprochen, das "^" wird nicht gebraucht. Im Pascal weiss der Compiler diese Zusammenhänge.

Deshalb ist die obige Konstruktion "BankDevPtr" hier nicht anwendbar! Also muss !innerhalb! solch einer Funktion der User solch ein VAR-Parameter zu einem Banked-Var-Parameter umwandeln mit:

^Bank[bank#].var_name:= xxx;

Der Treiber selbst exportiert nur eine Funktion:
***Function** GetBankNum(bankedVar) : byte;*
Die Funktion liefert die Bank Nummer der gebankten Variablen „bankedVar“ zurück.

Zur Verdeutlichung lesen und analysieren Sie bitte sorgfältig das **Test Programm** im Verzeichnis
..\E-Lab\Demos\Banking

3.50.2 Hardware Beispiel

In dem im **Test Programm** und im untenstehenden Schaltplan enthaltenem Beispiel liegen die Banks alle im Bereich $8000..$FFFF. Es wurde ein externes 512kByte SRAM (16 banks) implementiert. Die erste physische Bank wird hier für den XDATA Bereich benutzt und ist deshalb für das Banking nicht verwendbar. Der Vorteil dieser Anordnung ist, dass Standard Speicher wie z.B. XDATA keinen Overhead erzeugt wie z.B. das Banking.

Wenn also die XDATA Page benutzt wird, so entfällt die Bank0 für das Banking. Aus diesem Grund wird die übergebene Banknummer in diesem Beispiel im Treiber um eins inkrementiert. Dadurch wird die logische Bank0 auf die physische Bank1 umgemapped etc. etc.
Ein anderer einfacher Weg mit XDATA ist, die logische Bank0 ($BDATA 0} einfach nicht zu benutzen.

Wenn XDATA benutzt wird, beachten Sie bitte dass u.U. **Interrupts** den XDATA Bereich adressieren können. Ist das der Fall, muss der globale Interrupt gesperrt werden für die Dauer des Bank Zugriffs. Beim Verlassen der Treiber muss weiterhin sichergestellt werden, dass immer Bank0 aktiviert ist.

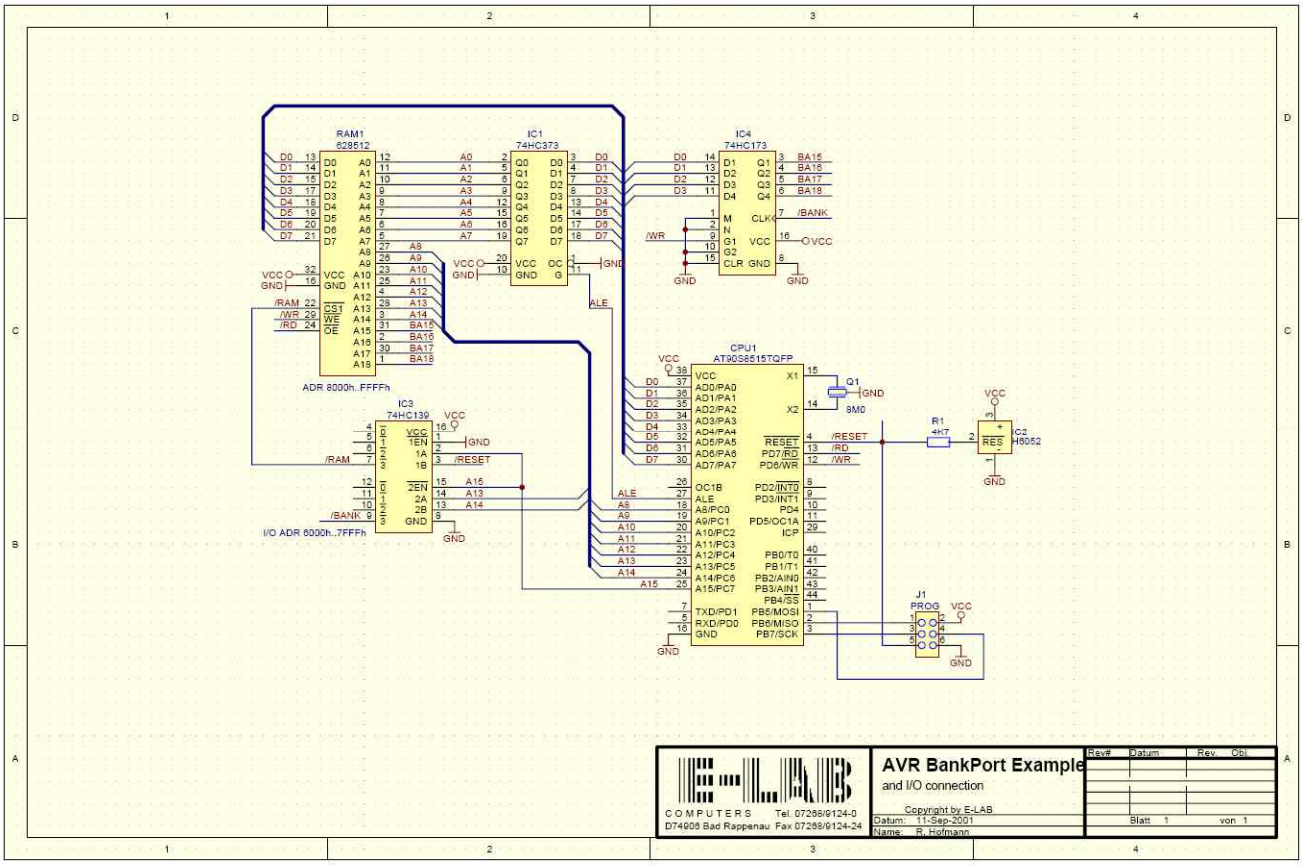
Um 512kB in den Banks zu Adressieren, müssen die notwendigen zusätzlichen Adressen für das 512kB SRAM generiert werden. Dazu ist ein Banking Port implementiert auf Adresse $7FFF (memory mapped). Das kann z.B. eine einfache Latch oder ein PLD sein. Wenn die CPU noch ein paar Portpins frei hat, können auch diese dazu verwendet werden, die zusätzlichen Adressleitungen zu erzeugen.

Im folgenden Beispiel beginnt jede Bank auf $8000 weshalb diese Konstante zu dem Adressparameter addiert werden muss.

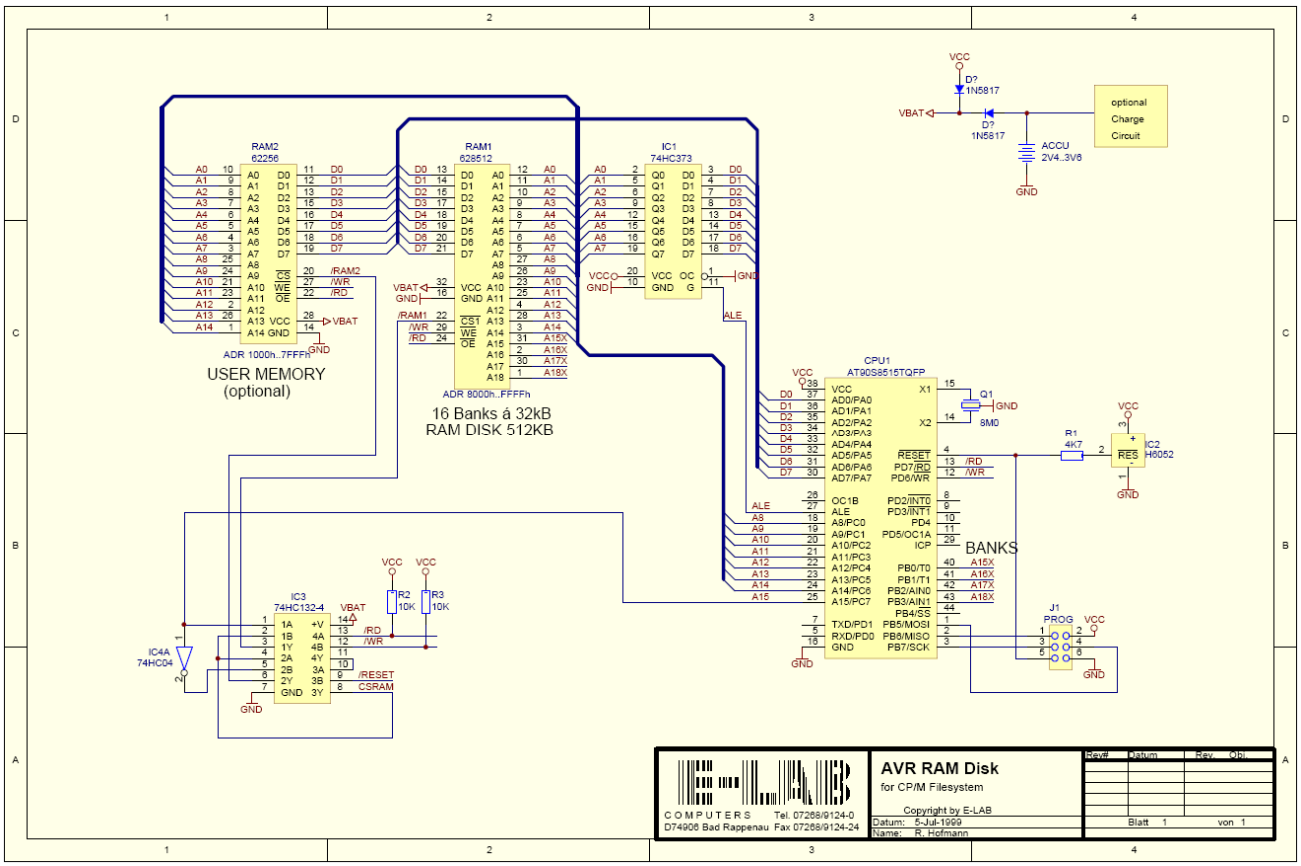
```
program BankTest;

Device = 90S8515, VCC=5;

Import SysTick, BankPort;

From System Import ;

Define
      ProcClock   = 6000000;        {Hertz}
      SysTick     = 10;             {msec}
      StackSize   = $0064, iData;
      FrameSize   = $0064, iData;
      BankPort    = 3;              {Bank Ports}

Implementation

{-----------------------------------------------------------------}
{ Var Declarations }
{$IDATA}
var   bb    : byte;

{$BDATA 0}                    // variables into bank 0
var   bb0   : byte;

{$IDATA}
{-----------------------------------------------------------------}
{ functions }
(* for a BankDevice import the following 3 function must be implemented *)

UserDevice BankDevIni;        // Bank Device init
begin
  (* is called at System Init Time *)
  (* initialize Device Hardware *)
end;

UserDevice BankDevInp (bank : byte; adr : word) : byte;
begin
  ...
  return(xxx);
end;

UserDevice BankDevOut (bank : byte; adr : word; arg : byte);
begin
  ...
end;

{-----------------------------------------------------------------}
{ Main Program }

begin
  EnableInts;
  loop
    NOP;
  endloop;
end BankTest.
```

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Banking**



Schaltplan **BankPort**



Schaltplan **AVRRamDisk**

3.51 Flash DownLoader/Writer

3.51.1 Übersicht

Neuere AVR CPUs (mega8, mega16, mega128 etc) gestatten es, dass das Anwendungsprogramm zur Laufzeit selbst in das onChip Flash schreiben kann. Das eröffnet dem Programmierer viele neue Möglichkeiten vom einfachen ändern von ROM-Konstanten über Code-Modifizierung bis zum Download einer geänderten bzw. neuen Applikation.

Für diese Zwecke besitzen diese CPUs den Maschinenbefehl **SPM** (StoreProgramMemory) und einen speziellen Bereich im Flash, die sog. **BootLoader Section**. Mit dem SPM Befehl wird in das Flash geschrieben. Der SPM Befehl selbst kann in der Regel nur im BootLoader Bereich angesiedelt sein und nur von dort auch ausgeführt werden.

Unbedingt beachtet werden muss bei allen diesen Operationen, dass der Schreibvorgang immer Blockweise erfolgt. Bei grösseren CPUs sind das 128Byte Blöcke. Weiterhin muss jeder Zugriff in einen solchen Block auf Word-Basis erfolgen.

AVRco unterstützt diese Eigenschaften der neuen CPUs durch die Import Anweisung **Import FlashWrite** und die Compilerschalter **{$BOOTRST $nnnn}**, **{$PHASE BOOTBLOCK $nnnn}** und **{$DEPHASE BOOTBLOCK}**.
Zusätzlich wurden zwei neue Bibliotheks Gruppen (Treiber) implementiert: FlashWrite zur Laufzeit und FlashDownLoad zur Laufzeit.

Imports
Um die Bibliotheks Funktionen nutzen zu können, muss die Gruppe **FlashWrite** importiert werden:

```
Import SysTick, FlashWrite, ..;
```

Bitte beachten Sie, dass dieser Import **spätestens an zweiter Stelle** der Importliste kommen muss.
Dieser Import generiert eine Word Variable **FLASH_ADDR** am Anfang von iData. Ist das Flash grösser als 64kB, wird auch noch zusätzlich eine Byte Variable **FLASH_PAGE** generiert. Diese Variablen werden von allen internen FlashWrite Funktionen benutzt und sollten während dieser Zeit normalerweise nicht verändert werden. Ansonsten können Sie vom Programm selbst beliebig verwendet werden.

3.51.2 Compiler Schalter

{$BOOTRST $nnnn}
Dieser Schalter dient nur zur Information des Debuggers/Simulators E-LAB AVRsim, ob der Hardware Reset mit den Vektor Addr 0000 ausgeführt wird, oder in den Bootblock geht. Der Parameter $nnnn bestimmt bei aktivem Schalter die Einsprung Adresse für den Simulator bei einem Reset. Mit der realen CPU wird mit dem Fusebit *BOOTRST* während des Programmierens durch den InCircuit Programmer bestimmt, wie die CPU sich bei einem solchen Reset verhalten soll. Der Schalter hat also keinerlei Auswirkungen auf das generierte Programm und damit auch nicht auf das Verhalten der CPU selbst. Der Adress Parameter kann beliebig gewählt werden, hat aber nur einen Sinn, wenn er mit den möglichen Einsprung Adressen der jeweiligen CPU übereinstimmt. Der Schalter muss nach der Device Deklaration platziert werden.

```
mega8 $nnnn = $0C00, $0E00, $0F00, $0F80;
mega16 $nnnn = $1C00, $1E00, $1F00, $1F80;
```

{$PHASE BOOTBLOCK $nnnn}
Dieser Schalter bestimmt, dass der nachfolgende Code Block auf Adresse $nnnn beginnt. Er dient weiterhin dazu, dass der Simulator Stack- und Frame Pointer Manipulationen zulässt, ohne eine Fehlermeldung auszugeben. Ist der Schalter $BOOTRST aktiv, so sollten i.A. für beide Schalter die Adressangaben identisch sein. Die o.a. Adressangaben gelten hier in gleicher Weise. Mit aktiviertem FuseBit BOOTRST erfolgt damit der Reset Einsprung auf die angegebene Adresse und das erste Statement nach diesem Compilerschalter ist auch das erste das nach einem Reset ausgeführt wird.

{$DEPHASE BOOTBLOCK}
Dieser Compilerschalter schliesst den Bootblock ab und ist zwingend erforderlich. Hiermit werden die FlashWrite Treiber im BootBlock installiert. Der Nachfolgende Code bzw. Statements werden wie üblich abgelegt bzw. schliessen an den Code an, der vor dem Schalter {$PHASE ...} generiert wurde.

3.51.3 BootApplication

{$BootApplication $nnnn} und {$BootApplication} für XMegas
Als eine Alternative zum platzieren des BootBlocks in die Applikation erlaubt dieser Schalter das Erstellen einer Applikation, die direkt in den BootBlock platziert wird. Der Schalter muss nach der Device Deklaration platziert werden.

3.51.4 FlashWrite zur Laufzeit

Die zugehörigen Funktionen erlauben es zur Laufzeit einzelne Bytes oder ganze Codeblöcke im Flash Speicher zu modifizieren. Hier ist zu beachten, dass die aktuellen CPUs nur komplette Blöcke (32..128 words) schreiben können. Weiterhin können nur words in diesen Block abgestellt werden und zwar auf gerade Adressen.
XMegas: Diese Funktionen sind nicht empfohlen für XMegas. Benutzen Sie *FlashDownloader* stattdessen.

3.51.4.1 Funktionen

FlashInitPage
Mit *FlashInitPage(const addr : LongWord)* wird ein Block vorbereitet. Der Parameter addr ist eine Byte Adresse und muss eine Konstante sein. Eine Manipulation dieser Konstanten zur Aufrufzeit (z.B. addr + 2) ist nicht möglich. Der Parameter muss nicht unbedingt auf eine Page Grenze zeigen. Im allgemeinen werden hier Adressen von ROM-Konstanten angegeben. Der Parameter wird in der Variablen *FLASH_ADDR* und falls vorhanden in *FLASH_PAGE* abgelegt. Diese Funktion sollte vor jeder neuen Page aufgerufen werden.

```
Procedure FlashInitPage (const addr : LongWord);
```

FlashReadPage
Liest die Flashpage, auf die *FLASH_ADDR* zeigt, aus dem ROM in den temporären CPU Buffer. Damit befindet sich in diesem Buffer ein Abbild der Flashpage. Der Inhalt des Buffers kann jetzt mit *FlashWritePage* verändert werden. Achtung: Ein *FlashProgPage* oder *FlashErasePage* löscht automatisch diesen Buffer.

```
Procedure FlashReadPage;
```

FlashClearPage
Löscht den temporären CPU Buffer mit $FF.

```
Procedure FlashClearPage;
```

FlashWritePage
Schreibt ein Wort in den temporären Buffer der CPU. Als Adresse (Pointer) wird die Variable *FLASH_ADDR* benutzt. Bit0 wird ignoriert. Durch diese Variable wird die Ziel Adresse dieses Schreibbefehls bestimmt. Eine Operation wie:

```
inc (FLASH_ADDR, 2);
```

ist daher zulässig und auch notwendig um diesen Buffer Pointer zu manipulieren. Aber unbedingt darauf achten, dass nur innerhalb der Page Grenzen adressiert werden darf, andernfalls ist ein Crash vorprogrammiert.
Diese Funktion inkrementiert den Pointer *FLASH_ADDR* anschliessend immer um 2. Daher ist nach Abschluss aller Buffer Manipulationen die Variable *FLASH_ADDR* wieder auf ihren Ausgangswert zu setzen.

```
Procedure FlashWritePage (const parm : word);
```

FlashErasePage
Diese Prozedur löscht die mit *FLASH_ADDR* bestimmte Page im Flash und löscht auch automatisch den temporären Buffer. Bevor der Temporäre Buffer mit FlashProgPage ins Flash geschrieben werden kann, sollte im Normalfall diese Page mit FlashErasePage gelöscht werden. Falls die Zielpage schon gelöscht ist (freier Flash Speicher) oder die zu veränderten Speicherstellen sind auf $FF, so kann darauf verzichtet werden.

```
Procedure FlashErasePage;
```

FlashProgPage
Der Temporäre Buffer der CPU wird in das Flash programmiert. Die Variable *FLASH_ADDR* bestimmt wiederum die Zielpage. Der temporäre Buffer wird anschliessend mit $FF gelöscht.

```
Procedure FlashProgPage;
```

FlashReadFuses
Die Fusebits des AVR können zur Laufzeit mit dieser Funktion ausgelesen werden. Der Parameter *FuseGroup* bestimmt dabei welches Byte gelesen werden soll. Welche Werte für welches Byte gelten, ist dem jeweiligen AVR Datenblatt zu entnehmen. Der Parameter *FLASH_ADDR* wird dazu nicht benötigt.

```
Function FlashReadFuses (FuseGroup : byte) : byte;
```

FlashWriteFuses
Die Fusebits des AVR können zur Laufzeit mit dieser Funktion gesetzt werden. Der Parameter *FuseGroup* bestimmt dabei welches Byte gesetzt werden soll. Welche Werte für welches Byte gelten, ist dem jeweiligen AVR Datenblatt zu entnehmen. Zur Zeit lässt sich nur das LockByte schreiben und hier können Bits auch nur auf 0 programmiert werden aber nicht auf 1. Der Parameter *FLASH_ADDR* wird dazu nicht benötigt.

```
Procedure FlashWriteFuses (FuseGroup, fsBits : byte);
```

FlashCopyF2R
Müssen komplett beschriebene pages nur teilweise verändert werden, z.B. nur ein Byte, dann geht das nur über das Page Erase. Deshalb muss diese Page gerettet werden, die Page gelöscht werden, die Änderungen in der geretteten Page vorgenommen und diese Page dann ins Flash zurückgeschrieben werden.

Die Funktion *FlashCopyF2R* kopiert die Flash Page, auf die *FLASH_ADDR* zeigt, in den Buffer im RAM auf den der Parameter *p* zeigt. Der RAM Buffer muss von der Grösse her in der Lage sein, eine Flash Page aufzunehmen (mega128 = 256bytes).

Die Applikation kann jetzt beliebige Änderungen in dem RAM Buffer vornehmen.

```
Procedure FlashCopyF2R (p : pointer);
```

FlashCopyR2F
Diese Funktion kopiert die zuvor mit *FlashCopyF2R* gerettete Flash Page in den temporären Buffer der CPU. Dieser Buffer kann jetzt mit *FlashProgPage* in das Flash programmiert werden.

```
Procedure FlashCopyR2F (p : pointer);

const
  StringC[$10200] : string = 'AF_ZKG';

{$IDATA}
var
  ar  : array[0..255] of byte;

{$Validate FLASHERASEPAGE *)
{$Validate FLASHPROGPAGE *)
{$Validate FlashCopyF2R }
{$Validate FlashCopyR2F }
```

```
{$PHASE BootBlock $0F000}
Procedure FlashProg;    // at $0F000
Begin
  // dummy
end FlashProg;
{$DEPHASE BootBlock}


// main
begin
  FlashInitPage (ADDR(StringC));
  FlashCopyF2R (@ar);
  // ... do the changes in “ar”
  FlashErasePage;
  FlashCopyR2F (@ar);
  FlashProgPage;
  ...
```

3.51.5 Verwendung von FlashWrite

3.51.5.1 Implementation

Die Aufrufe von **FlashWrite** Funktionen können an jeder Stelle des Programms erfolgen. Einzige Bedingung ist jedoch, dass **erst nach** dem letzten Aufruf von einer FlashWrite Funktion der Compilerschalter

```
{$DEPHASE BOOTBLOCK}
```

erfolgt. Daher gibt es drei Möglichkeiten zur Implementation von FlashWrite:

1. FlashWrite im normalen Code Bereich
Aufruf der Treiber Funktionen vor der Definition des BootBlocks:

```
Procedure WriteTest;
begin
  FlashInitPage (Addr(KeyLookUp));
  FlashReadPage;
  FlashErasePage;
  FlashWritePage ($5857);
  inc (FLASH_ADDR, 2);
  FlashWritePage ($5A59);
  FlashProgPage;
end WriteTest;

Procedure ABC;
begin
  ...
end;

{$PHASE BootBlock $01F80}
{$DEPHASE BootBlock}                    // inserts Flash drivers at $1F80
```

2. FlashWrite innerhalb des BootBlocks

```
{$PHASE BootBlock $01F80}
Procedure WriteTest;
begin
  FlashInitPage (Addr (KeyLookUp));
  FlashErasePage;
  FlashWritePage ($5857);
```

```
  inc (FLASH_ADDR, 2);
  FlashWritePage ($5A59);
  FlashProgPage;
End WriteTest;
{$DEPHASE BootBlock}
```

3. Validation

```
// validate the used driver functions before the BootBlock is defined
{$Validate FLASHREADPAGE }
{$Validate FLASHERASEPAGE }
{$Validate FLASHPROGPAGE }
{$Validate FLASHWRITEPAGE }

{$PHASE BootBlock $0F000}
{$DEPHASE BootBlock}

(* main *)
FlashInitPage (Addr  (KeyLookUp));
FlashErasePage;
```

Die 3. Art dürfte die eleganteste sein, denn die Treiber Aufrufe können an fast beliebiger Stelle erfolgen.

Bemerkungen
Wie oben zu ersehen ist, kann ein Schreibzugriff in den temporären Buffer der CPU nur auf gerade Adressen und auch nur mit einem Word Argument erfolgen. Dieser Buffer liegt immer auf Page Grenzen (mega8..mega16) 128Bytes bzw. 64 Worte.

Für alle Operationen wird intern die Variable *FLASH_ADDR* benutzt. Bei Manipulationen dieses Wertes ist immer darauf zu achten, dass die Page Grenzen nicht verletzt werden. Mit der Prozedur FlashInitPage kann der Ausgangs Zustand wieder hergestellt werden.

Die korrekte Einstellung der BOOTSZx Fuses ist essentiell. Eine falsche Einstellung verhindert jeden Flash Zugriff des Treibers.

3.51.5.2 Page komplett erneuern

Kein Flash Bit oder Byte kann von 0 nach 1 programmiert werden. Wenn das gebraucht wird, dann muss die gesamte Flash Page zuvor auf $FF gelöscht werden.
Soll eine Flash Page komplett erneuert werden oder in einer komplett gefüllten Page auch nur ein Byte geändert werden, so muss zu einem Trick gegriffen werden. War es früher (Mega163) möglich, eine Page in den temporären Buffer zu lesen, die Flash Page zu löschen, den Buffer manipulieren und dann den Buffer ins Flash zurückzuschreiben, so geht das heute so nicht mehr. Alle neueren AVRs löschen ihren temporären Buffer mit einem Page Erase oder Page Program Vorgang. Der zuvor ausgelesene Buffer wird geleert, bevor man ihn zurückschreiben kann.

Abhilfe schafft da nur, dass man die Flash Page nicht in den temporären Buffer liest, sondern ins RAM. Dann erfolgt das Page Erase, das kopieren des RAM Buffers in den temporären Buffer und anschliessend das Page Program. Der Treiber unterstützt dies durch die Funktionen *FlashCopyF2R* und *FlashCopyR2F*

```
program WriteTest;

{$BOOTRST $0F000}        {Reset Jump to $0F000}

Device = mega128, VCC=5;

Import FlashWrite;

From System Import longword;
```

```
Define
  ProcClock     =  16000000;       {Hertz}
  StackSize     =  $40, iData;
  FrameSize     =  $80, iData;

Implementation

{$IDATA}
{-----------------------------------------------------------------}
{ Type Declarations }
type

{-----------------------------------------------------------------}
{ Const Declarations }
const
  StringC[$10200]       : string = 'AF_ZKG';
  // example how a fixed const can be
  // placed into the bootblock below
  BootConst[$1FFFC]  : word   = $1234;

{-----------------------------------------------------------------}
{ Var Declarations }
{$IDATA}
var
  bb    : byte;
  ww    : word;

(*****************************************************************)
(* Because all system Flash Write functions must be    *)
(* placed into the BootBlock part of the Flash we must *)
(* import the used functions by the "validate" switch     *)
(*****************************************************************)

{$Validate FLASHREADPAGE }
{$Validate FLASHERASEPAGE }
{$Validate FLASHPROGPAGE }
{$Validate FLASHWRITEPAGE }
{$Validate FlashReadFuses }
{ $Validate FlashWriteFuses }

// The definition of the correct BootBlock location
// is a must because all FlashWrite operation can only
// be executed from this area.

// "FlashProg" is an unused dummy function.
// It is not necessary but shows that there
// can be also user functions in the bootblock

{$PHASE BootBlock $0F000}           // !!!!!!!!!!!!!!
Procedure FlashProg;                // at $0F000
begin
end FlashProg;
{$DEPHASE BootBlock}                // !!!!!!!!!!!!!!

{-----------------------------------------------------------------}
{ Main Program }
begin
  // read the fuse and lockbits
  bb:= FlashReadFuses (1);       // lock bits
  bb:= FlashReadFuses (0);       // Fusebits 0
  bb:= FlashReadFuses (3);       // Fusebits 1
  bb:= FlashReadFuses (2);       // Fusebits 2
```

```
  // Flash manipulations
  FlashInitPage (ADDR (StringC));

  // either do a FlashReadPage
  FlashReadPage;
  // or do a FlashErasePage
  FlashErasePage;
  // but not both at the same time

  ww:= FLASH_ADDR;
  FlashWritePage ($4203);
  inc (FLASH_ADDR, 32);
  FlashWritePage ($4443);
  FLASH_ADDR:= ww;
  FlashProgPage;

  loop
    Nop;
  endloop;
end WriteTest.
```

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\FlashWrite**

3.51.5.3 FlashDownLoad zur Laufzeit

Im Gegensatz zu *FlashWrite* werden hier keine Funktionen zur Verfügung gestellt, sondern ein DownLoad Monitor. Mit Hilfe dieses Monitors ist es möglich das komplette Flash, ausgenommen den Boot Bereich, über eine beliebige Hardware von aussen herunterzuladen und in das Flash zu programmieren. Damit sind ferngesteuerte Updates der Firmware über PC, Modem etc. problemlos möglich.

Um FlashDownLoad nutzen zu können, muss die Gruppe **FlashWrite** importiert werden:

Import *SysTick, FlashWrite, ..;*

Bitte beachten Sie, dass dieser Import **spätestens an zweiter Stelle** der Importliste kommen muss.

Der Anwender muss dazu 4 Funktionen selbst zur Verfügung stellen:
FlashLoaderInit, FlashLoaderRecv, FlashLoaderTransm und (optional) *FlashLoaderExit*.

Anwender Funktionen

FlashLoaderInit
Diese Prozedur muss vom Anwendungsprogramm zur Verfügung gestellt werden. Ihre Aufgabe ist es die Schnittstelle zu initialisieren, die vom DownLoad Monitor benutzt werden soll. Die Art der Schnittstelle ist vollkommen frei gestellt. Sie muss nur in der Lage sein ein Byte zu empfangen und zu versenden.
Diese Funktion ist ein CallBack, daher gilt: **keine lokalen Variablen**.

Procedure *FlashLoaderInit;*

FlashLoaderRecv
Diese Prozedur muss vom Anwendungsprogramm zur Verfügung gestellt werden. Ihre Aufgabe ist es ein Byte von der Schnittstelle abzuholen und als Resultat zurückzugeben. Da hier normalerweise mit Assembler Statements gearbeitet wird, erfolgt die Rückgabe im Register _ACCA (R17). Ein RETURN Statement ist nicht vorgesehen. Diese Funktion ist ein CallBack, daher gilt: **keine lokalen Variablen**.

Procedure *FlashLoaderRecv;*

Achtung:
Werden zusätzliche Register benötigt, so sollten _ACCB/R16, _ACCALO/R18, _ACCAHI/R19, _ACCCLO/R30 und _ACCCHI/R31 nicht benutzt bzw. gerettet werden.

FlashLoaderTransm
Diese Prozedur muss vom Anwendungsprogramm zur Verfügung gestellt werden. Ihre Aufgabe ist es ein Byte an die Schnittstelle zu übergeben bzw. zu versenden. Da hier normalerweise mit Assembler Statements gearbeitet wird, erfolgt die Übergabe im Register _ACCA (R17).
Diese Funktion ist ein CallBack, daher gilt: **keine lokalen Variablen**.

Procedure *FlashLoaderTransm;*

Achtung:
Werden zusätzliche Register benötigt, so sollten _ACCB/R16, _ACCALO/R18, _ACCAHI/R19, _ACCCLO/R30 und _ACCCHI/R31 nicht benutzt bzw. gerettet werden.

FlashLoaderExit
Diese Prozedur kann vom Anwendungsprogramm zur optional Verfügung gestellt werden. Ihre Aufgabe ist es nach erfolgreichem Download einen Restart durchzuführen. Dies geschieht normalerweise mit einem JMP/RJMP auf den Reset Vektor auf Adresse SYSTEM.VectTab. Wird diese Prozedur nicht gefunden, beendet der Flashloader selbst mit einem JUMP zur Adresse 0000h = SYSTEM.VectTab.
Diese Funktion ist ein CallBack, daher gilt: **keine lokalen Variablen**.

Procedure *FlashLoaderExit;*

Anmerkungen:
Diese Anwender Prozeduren können an beliebiger Stelle im ROM/Flash liegen. Sinnvoll ist allerdings die Platzierung in den BOOTBLOCK, so dass der Download diese Treiber nicht überschreiben kann.

Als Schnittstellen bieten sich die CPU-internen I/Os an, z.B. UART, SPI, CAN, I2C oder Ports. Auch externe Schnittstellen sind ohne weiteres möglich.

Diese Treiber dürfen aber keinesfalls im Interrupt Modus laufen, da die Vektor Tabelle oder zumindest die zugehörigen Interrupt Treiber während des Downloads überschrieben werden.

Loader Identifikation
Das Loader Programm wird in aller Regel nur einmal in die CPU programmiert und bleibt dort unverändert. Es bietet sich deshalb an, hier auch die Hardware Revision o.ä. des Boards/Systems unterzubringen. Diese 16bit Nummer kann vor dem Download abgefragt werden, um sicherzustellen, dass die Download Firmware auch auf dieser Hardware lauffähig ist. Diese Nummer wird bei der BootLoader Generierung mit in den Maschinen Code des Loaders selbst als immediate Konstante eingebracht. Das geschieht durch die

Definition einer globalen Konstante mit dem Namen „DownLoaderID“:

```
Const DownLoaderID : word = 10213;
```

Wird diese Konstante nicht definiert, wird dieser Wert als $0000 im Loader abgespeichert. Die ID-Nummer kann mit dem Befehl ‚i’ vom Loader abgefragt werden.

Es sind bis zu 4 ID-Words möglich, die dann alle sequentiell mit dem i-Kommando gesendet werden:

```
const
  DownLoaderID : word = $1234;
  DownLoaderID1 : word = $5678;
  DownLoaderID2 : word = $9ABC;
  DownLoaderID3 : word = $DEF0;
```

Sicherheiten
Wenn verhindert werden soll, dass das Flash oder EEprom über den Loader von unbefugten ausgelesen werden kann, muss noch eine Passwort Konstante definiert werden:

```
Const DownLoaderPWD : word = $1A2B;
```

Damit muss das Upload Tool mit dem Upload Kommando für das Flash und das EEprom auch immer dieses Passwort bereitstellen.

3.51.5.4 FlashDownLoader Funktionen

FlashDownLoader
Hinter dieser Prozedur versteckt sich der Download Monitor. Mit dem Aufruf dieser Prozedur wird der globale Interrupt abgeschaltet, der StackPointer wird an das Ende des Rams gelegt, ebenso der vom Monitor selbst nicht benutzte FramePointer. Die Variable **FLASH_ADDR** und evtl. **FLASH_PAGE** wird initialisiert. Daran anschliessend befindet sich ein Zwischenspeicher für Up- und Downloads. Die Grösse dieses Bereichs ist ein Array[0..PageSize-1] of byte; Der Bereich nach diesem Array und dem Frame bleibt frei und wird nicht benutzt.

Nach der internen Initialisierung des Loaders erfolgt der Aufruf der Anwender Prozedur *FlashLoaderInit*. Kehrt diese zum Loader zurück läuft der Loader in seiner Kommando Schleife. D.h. er springt kontinuierlich die Funktion *FlashLoaderRecv* an und wertet dann die empfangenen Kommandos und Daten aus. Auf jedes Kommando gibt der Loader eine Antwort oder Quittung über die Anwender Prozedur *FlashLoaderTransm* aus.

Erhält der Loader den Exit Befehl, so springt er die Anwender Prozedur *FlashLoaderExit* an und beendet sich damit selbst. Wird diese Prozedur nicht gefunden, beendet der Flashloader mit einem JUMP zur Adresse 0000h.

***Procedure** FlashDownLoader;*

FlashDownLoader Kommandos
Der Loader kennt eine Anzahl von Kommandos. Diese bestehen i.A. aus einem ASCII Zeichen und evtl. nachfolgenden Parametern. Jedes Kommando wird durch ein CR ($0D bzw. #13) nach der Ausführung quittiert. Ausnahmen davon sind die Block Transfer Kommandos. Kommandos werden immer vom Host gesendet. Der Loader führt immer nur aus. Ausser der Initialisierung muss jede Aktion vom Host initiiert werden.

Die Kommandos des Hosts können eine Page Adresse setzen, eine Page hoch oder runterladen, eine Page im Flash löschen, eine Page vom Flash in den Zwischenbuffer lesen und den Buffer in das Flash programmieren.

Ein wesentlicher Teil ist die Abfrage des Loader Status/Parameter mit dem ? Kommando. Hier erfährt der Host, welche CPU-ID das Target hat, wie gross seine Flash Pages sind und wo der BootBlock beginnt. Alle Parameter, die Grössen und Adressen anbelangen sind als Wort-Parameter ausgeführt. Das heisst, dass solch ein Parameter selbst nicht unbedingt ein Wort sein muss. Der Parameter selbst zählt in Worten. Wenn z.B. der Loader eine Page Grösse von 64 angibt, so sind das 64 Worte bzw. 128 Bytes. Das gleiche gilt z.B. für das Adress Kommando des Host. Wenn dem Kommando A der Wert 256 folgt, bedeutet das für den Loader, dass er die nächste Page die Wort Adresse 256 einstellen soll, dies ist die Adresse 512 in Bytes gerechnet. Das Host Programm muss das unbedingt beachten.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\SelfProg**

3.51.6 Host Programm

In der Installation von AVRco ist ein PC-gestütztes DownLoader Programm **FlashLoader.exe** enthalten. Dieses ist in **..E-Lab\DOCs\DocuTools.pdf** im Kapitel "*Flash Down Loader / Writer"* näher beschrieben. An dieser Stelle nur nochmals die Protokoll Beschreibung:

FlashLoader Kommando Liste

? Host fordert Loader Kennung an.
Loader antwortet mit
FD FlashDownLoader

A Host schickt page adr in word Darstellung. Alle Flash Aktionen beziehen sich auf diese Page
aa1 page adr loByte
aa2 page adr hiByte
aa3 wenn die CPU mehr als 128kBytes Flash hat muss noch dieses Page Extend Byte folgen
Loader antwortet mit
CR Befehl akzeptiert

B Host schickt EEprom adr in byte Darstellung und EEprom Data (byte).
aa1 EEprom adr loByte
aa2 EEprom adr hiByte
data 1 Byte ins EEprom
Loader programmiert dieses Byte in das EEprom und antwortet mit
CR Befehl akzeptiert
C Host schickt EEprom adr in byte Darstellung.
aa1 EEprom adr loByte
aa2 EEprom adr hiByte
Wenn ein (optionales) Passwort gesetzt ist (siehe weiter unten), muss der Host das Passwort schicken:
pw1 Passwort loByte
pw2 Passwort hiByte
Loader liest das EEprom byte an dieser Adresse im EEprom und schickt es als Antwort zurück

D Host lädt eine Page Daten zum Loader herunter
XMega 2bytes Blocksize in Bytes lo/hi byte
data es folgen (***ps*** x 2) Bytes = Pagesize x 2 (mega8..meg16 = 128bytes)
Loader speichert diese Page in sein Array und bildet eine 8Bit Checksumme durch Addition aller Bytes.
Loader schickt die errechnete Checksumme als Ergebnis der Operation zum Host:
cc Checksumme

E Host fordert löschen der aktuellen Page
Loader löscht die aktuelle Page im Flash und antwortet mit
CR Befehl akzeptiert

I Host fordert Loader Info an.
Loader antwortet mit
I Info Kennung = Loader-ID siehe weiter unten
id1 hiByte Prozessor ID
id2 midByte Prozessor ID
id3 loByte Prozessor ID
ps words pro page
bs1 Bootblock start adr lobyte \
bs2 Bootblock start adr hibyte / = Bootblock start adr in word count
bs3 Bootblock start adr extbyte / = Bootblock start adr in word count wenn Flash > 128kB

P Host fordert Programmieren des Zwischenspeichers ins Flash
Loader überschreibt die aktuelle Page im Flash mit dem Inhalt des Zwischenspeichers. Ein Verify findet nicht statt.
Der Loader antwortet mit:
CR Befehl akzeptiert

U **XMega** UserRow programmieren.
CR Befehl akzeptiert
aa1 page adr loByte
aa2 page adr hiByte
aa3 Page Extend Byte
Loader antwortet mit
CR Befehl akzeptiert
data es folgen (***ps*** x 2) Bytes = Pagesize x 2
Loader speichert diese Page in sein Array und bildet eine 8Bit Checksumme durch Addition aller Bytes.
Loader schickt die errechnete Checksumme als Ergebnis der Operation zum Host:
cc Checksumme
R Gespeicherte Page in das UserRow schreiben..
Der Loader antwortet mit:
CR Befehl akzeptiert

W Host schickt eine relative page adr in word Darstellung. Darauf folgt ein word, das in den Zwischenspeicher in diese Adresse geschrieben werden soll.
aa page adr relativ
ww1 loByte
ww2 hiByte
Loader antwortet mit
CR Befehl akzeptiert

X Ende der Kommunikation. Der Loader Monitor springt die Anwender Prozedur *FlasLoaderExit* an. Wird diese Prozedur nicht gefunden, beendet der Flashloader mit einem JUMP zur Adresse 0000h. Keine Antwort vom Loader.

3.51.7 Boot Bereich, Optimiser und Neu-Compile

Wird der Optimiser verwendet oder im Boot Bereich sind grössere Treiber angesiedelt, so kann es zu dem Problem kommen dass nach einem Neu-Compile die von der Applikation erwarteten Einsprung Adressen in den Boot sich geändert haben. Das betrifft die Applikation, da der Boot ja bei einem Selbst Flashen nicht verändert werden kann. Dies führt dann dazu dass die Applikation auf jetzt neu errechnete Adressen im Boot springt, die sich im Boot allerdings nicht geändert haben, da diese dort ja konstant sind.

Abhilfe schafft hier die Einführung von Boot Traps wie im Compiler Manual beschrieben. Dazu werden die Boot Traps importiert:

```
From System Import Traps;
```

Dieser Import legt eine Sprung Tabelle im Boot Bereich an. Jetzt müssen noch die von der Applikation erreichbaren Funktionen im Boot Bereich als Traps im Boot Block definiert werden. Da normalerweise die erste Funktion im Boot Block immer eine Boot Test Funktion sein sollte (Procedure BootTest) muss die erste Trap Funktion dahinter plaziert werden:

```
{$PHASE BootBlock $01F00}
Procedure BootTest;
begin
  …
end;

Procedure BootEntry; Trap;
begin
  FlashDownLoader;
end;
…
{$DEPHASE BootBlock}
```

Da zumindest ein Boot Einsprung zum neu-Flashen für die Applikation möglich sein sollte, aber die Boot Funktion “FashDownLoader“ möglicherweise nicht mehr sicher erreicht werden kann, sollte diese über die Trap Funktion “BootEntry“ jetzt sicher erreicht werden. Applikation:

```
  if xxx then
    BootEntry;
```

```
endif;
```

3.52 BootApplication und MainApplication

Als eine Alternative zum platzieren des BootBlocks in die Applikation erlaubt der Schalter **{$BootApplication $nnnn}** bei XMegas **{$BootApplication}** das Erstellen einer kompletten Applikation, die direkt in den BootBlock platziert wird. Der Schalter muss nach der Device Deklaration platziert werden.

Bei normalen AVR Applikationen steht die Interrupt Vektor Tabelle immer ab Adresse $0000 im Code Bereich (Flash). Das gleiche gilt auch für den Codestart, d.h. auch der generierte Code beginnt offiziell auf Adresse $0000 bzw. exakt hinter der Vektor Tabelle.

Mit diesem Schalter kann nun das System angewiesen werden, die Adress Generierung und die Vector Tabelle in den Boot Bereich zu verschieben. Damit wird es möglich eine Applikation zu erstellen, die nur im Boot Bereich des Controllers läuft und trotzdem alle System Resourcen, Treiber und Interrupts zur Verfügung hat da diese auch alle in den Boot Bereich plaziert werden. Der Adress Parameter für den Schalter ist immer eine Word Adresse!

Eine Main Applikation kann mit der Boot Applikation in dem ***define_fuses*** block zusammengeführt werden:
AddApp zwingt den Programmer ein weiteres Hexfile eines anderen Projekts zu laden:

```
AddApp = 'pathname\projectname';
```

Bei den **XMegas** wird **nur** die Boot Applikation unterstützt. Die XMega Treiber sind wesentlich komplexer als die der AVRs und es macht daher Sinn das komplette Runtime System mit den notwendigen Treibern im Boot Bereich zu haben.

Support Funktion für die Boot Applikation zum Einsprung in die Standard Applikation. Diese Prozedur sperrt auch einen evtl. aktiven WatchDog, sperrt die Interrupts und schaltet die Interrupts von Boot-Vektoren auf die Main-Vektoren ($0000) um.

```
Procedure Application_Startup;
```

Support Funktion um gezielt die Vektor Tabelle umzuschalten:

```
Procedure SetVectTabBoot(boot  : boolean);
```

Support Funktion um gezielt aus der MainApp die BootApp zu starten:

```
Procedure BootStart;
```

Die **MainApp**, die im Gegensatz zur BootApp von der Start Adresse $0000 beginnt und deren Vektor Tabelle auch auf $0000 beginnt, kann mit der BootApp über Traps kommunizieren. Dazu muss die **BootApp** Traps importieren und die entspr. Funktionen exportieren. Beispiel in der BootApp:

```
From System  Import Traps;
…
function BootFunction : boolean; Trap;
begin
  // do anything
  return(true);
end;

procedure BootProc(x : byte); Trap;
begin
  // do anything
end;
```

Wenn innerhalb einer **BootApp** solche Traps definiert werden, erzeugt der Compiler/Assembler ein File "*BootAppName*.traps". Um diese Traps/Einsprünge in der **MainApp** verwenden zu können muss dieses File in der MainApp importiert werden.

```
From System  Import Traps;

ExternalTraps 'BootAppName.traps';        // max 16 ext traps !

Define
  ...
```

Zur Info ein Beispiel Inhalt eines Trap Files:

```
BOOTSTART   $040000
1479   20241                          .TRAPTAB
1480   20241         SYSTEM.TrapTab:
1481   20241    0200FE          .ADDR     BootApp.BootFunction //function BootFunction : boolean; Trap;
1482   20242    020105          .ADDR     BootApp.BootProc //procedure BootProc(x : byte); Trap;
1483   20244    000000          .ADDR     0
1484   20245    000000          .ADDR     0
1485   20247    000000          .ADDR     0
1486   20248    000000          .ADDR     0
1487   2024A    000000          .ADDR     0
1488   2024B    000000          .ADDR     0
1489   2024D    000000          .ADDR     0
1490   2024E    000000          .ADDR     0
1491   20250    000000          .ADDR     0
1492   20251    000000          .ADDR     0
1493   20253    000000          .ADDR     0
1494   20254    000000          .ADDR     0
1495   20256    000000          .ADDR     0
1496   20257    000000          .ADDR     0
```

Der Compiler kennt nun diese Funktionen, ihre Parameter und die absoluten Einsprung Adressen im Boot. Nun können in der MainApp diese Trap Funktionen aufgerufen werden:

```
if BootFunction then
  // ...
endif;
BootProc($33);
```

Achtung:
In der BootApp laufen zu diesem Zeitpunkt keinerlei Interrupts mehr! Das bedeutet z.B. dass alle Funktionen die mit TimeOuts zu tun haben nicht mehr arbeiten können, da die Interrupts nur noch die MainApp bedienen. Daher ist in der MainApp sorgfältig zu prüfen ob es mit einem Trap nicht zu einem dead-lock kommen kann.

Aus Sicherheits Gründen werden die Interrupts vor dem Einsprung in eine Boot Funktion gesperrt. Nach der Rückkehr aus dem Boot Bereich werden die Interrupts wieder auf den vorhergehenden Status gesetzt.

Ausserdem ist zu beachten, dass ein re-compile der MainApp jederzeit möglich ist. Nach einem re-compile der BootApp muss aber in der Regel auch die MainApp neu compiliert werden.

Eine Beispiel Applikation ist in der Demos Directory unter "BootApp" zu finden.

3.52.1 XMega FlashLoader

Die XMegas unterscheiden sich im Bootloader extrem von den normalen Megas. Das betrifft vor allem die Flash Operationen im Boot aber auch die Tatsache dass es bei den XMegas ein spezielles Boot Flash gibt. Aus diesem Grund wird hier nicht die Phase/Dephase Anordnung unterstützt sondern nur die separate BootApp Applikation, definiert durch den Schalter ***{$BootApplication}***. Dieser Schalter muss unmittelbar nach dem Device Define angeordnet sein. Weitere Infos siehe im vorhergehenden Kapitel. Weiterhin sind mehrere Konstanten und Funktionen ein "Muss":

```
const
  // this constant must be the same as in the Main app
  DownLoaderID : word   = $1234;           // mandatory constant

procedure FlashLoaderInit;                    // mandatory
begin
end;

function FlashLoaderRecv : byte;              // mandatory
begin
  return(SerInpC0);
end;

procedure FlashLoaderTransm(arg : byte);      // mandatory
begin
  SerOutC0(arg);
end;
                                              // mandatory
procedure FlashLoaderExit;
begin
  if Application_Valid then     // Flag is cleared at Loader start and set by the
                                // Bootloader if successful
    EEprom[EEpromEnd]:= $00;    // validate application for Boot usage
  Endif;
  mDelay(100);
  HardwareReset;              // restart with a jump into the Boot
end;
```

Die Konstante ***DownLoaderID*** muss vorhanden sein und muss in der BootApplication und in der downloadbaren Main identisch sein.

Die Prozedur ***FlashLoaderInit*** wird immer beim Download Start aufgerufen und kann dazu benutzt werden um bestimmte Initialisierungen vorzunehmen.

Die Prozedur ***FlashLoaderRecv*** holt ein Byte von der Download Schnittstelle ab.

Die Prozedur ***FlashLoaderTransm*** übergibt ein Byte an die Download Schnittstelle.

Die Prozedur ***FlashLoaderExit*** muss einen erfolgreichen DownLoad kennzeichnen. Am besten geschieht dies durch ein Flag im EEprom auf das sowohl der Downloader als auch das Main zugreifen können. Im Main des Bootloaders wird bei jedem Neustart dieses Flag geprüft ob der letzte Download erfolgreich war. Wenn ja, dann springt der Loader gleich in die Main Applikation. Wenn nein, dann wartet er auf einen neuen Download. Durch den ***HardwareReset*** im LoaderExit wird ein kompletter Neustart erwungen der dann zur Prüfung dieses Flags führt.

Dazu muss aber immer die ***BootRst*** Fuse aktiviert sein:

```
Define_Fuses
  Override_Fuses;
  COMport   = USB;
  ProgMode  = PDI;
  LockBits0 = [];
  FuseBits0 = [];
  FuseBits1 = [];
  FuseBits2 = [BootRst];                        // mandatory !!
  // Brown-out is obligatory with USB !!!
  FuseBits5 = [BODACT0, BodLevel0, BodLevel1, BodLevel2];
```

Das Main im BootLoader muss diverse Prüfungen vornehmen:

```
{-------------------------------------------------------------}
{ Main Program }
{$IDATA}
begin
  // optional a port pin can be checked for a forced download
  // if Pin.x = false then ...
  //  ...
  if EEprom[EEpromEnd] = $00 then
    // if a Download failed or the app was never programmed then there is no $00
    // if the main app forces a download then the last byte in the EEprom must be $FF
    //
    Application_Startup;   // Jump into the Main Application
  endif;
  EnableInts($87);
  loop
    FlashDownLoader;
    // Downloader itself does an exit to the main app if
    // a valid main has been downloaded
  endloop;
end XMega_BootAppSer.
```

Die Main Applikation muss ebenfalls die gleiche DownloaderID wie das Boot enthalten:

```
const
  // this constant must be the same as in the Main app
  DownLoaderID : word   = $1234;           // mandatory constant
```

Wenn die **Main Applikation** läuft ist sichergestellt dass der letzte Download in Ordnung war. Es besteht hier aber die Möglichkeit einen neuen Download zu forcieren:

```
  EEprom[EEpromEnd]:= $FF;    // invalidate application for Boot usage
  mDelay(100);
  HardwareReset;              // restart with a jump into the Boot
```

Durch den ***HardwareReset*** wird ein kompletter System Restart durchgeführt der immer ins Boot führt. Im Boot wird dann, wie oben beschrieben, dieses Flag geprüft und da es <> $00 ist, wird sofort in den FlashDownLoader verzweigt. Voraussetzung dafür ist aber dass beim Programmieren der BootApp auch die Fuse ***BootRst*** aktiviert wurde.

USB Download Mode

Hier ist zu beachten dass kein USBprodName angegeben werden kann. Dieses Define wird intern auf “BootLoader“ gesetzt.

```
Import USBboot, FlashLoader, ...;
Define
  ...
  USBmanufact    = 'E-Lab Computers';       // max 31 bytes
  USBpid         = 30;
  USBvid         = $2345;
  USBprodRel     = 201;
  USBcurrent     = 200;
  USBvBUS        = PortB.7;                 // port and pin
  // USBvBUS        = none;                 // not used
```

Erstellen einer kompletten Boot Applikation

1. Erstellen und compilieren der BootApp
2. Erstellen und compilieren der MainApp.
3. Daraus mit AVRprog ein PAC-File erstellen.
4. BootApp flashen. Beim USB Boot mit dem Tool USB Inf/Sys Builder den USB Treiber generieren
5. Das Zielsystem starten. Dieses erwartet jetzt den Firmware download der MainApp
6. Mit dem Tool FlashDownloader das erstellte PAC-File laden und in die Ziel Hardware laden.

Eine Beispiel Applikation für einen **COMport** Download ist in der Demos Directory unter“XMega_Boot“ zu finden. In “XMega_Boot“ ist auch die zugehörige MainApp zu finden.

Eine Beispiel Applikation für einen **USB** Download ist in der Demos Directory unter“XMega_BootUSB“ zu finden. In “XMega_BootUSB“ ist auch die zugehörige MainApp zu finden.

3.53 Device Treiber

Übersicht
Oftmals reicht der interne Variablen Speicher von SingleChips (RAM/EEPROM) nicht aus. Als Lösung des Problems bieten sich zwei Möglichkeiten an: entweder eine grössere CPU verwenden, die mehr Speicher hat oder einen externen Speicher anschliessen. Die erste Möglichkeit muss oft ausgeschlossen werden, denn welche CPU bietet zur Zeit 32kByte internes EEprom?

Externe Speichererweiterung mit kompletten Adressen und Daten scheidet aus mehreren Gründen (Platz, Ports) oftmals aus. Damit bleibt in der Regel nur noch ein Device mit 3-Draht Schnittstelle, z.B. I2C, SPI oder microWire. Da diese Bausteine alle unterschiedlich funktionieren, kann der Compiler eigentlich keine Variablen in diesen Bereich legen und ein Adressieren bzw. Lesen und Schreiben auf HLL Ebene ist normalerweise nicht möglich.

Bisher war es daher notwendig diese Zugriffe selbst zu organisieren wobei ein verhältnismässig grosser Verwaltungsaufwand betrieben werden musste. Der externe Speicher wurde dabei i.A. als array of byte adressiert. Damit bestand z.B. kein direkter Zusammenhang zwischen einer externen Adresse und dem Typ der im externen Speicher gespeichert war. Ein Byte dazwischen einfügen hatte oft verheerende Folgen an einer ganz anderen Stelle im Programm.

Was liegt also näher, auch über einen solchen “exotischen” Speicher eine Struktur zu legen, als wäre er ein ganz “normales” RAM? Bleibt nur noch den tatsächlichen physischen Zugriff zu gestalten. Dies ist die Aufgabe eines Device Treibers.

Diese Device Treiber arbeiten Variablen orientiert, d.h. es werden Variablen in einem beliebigen externen Speicher angesprochen, als wären diese im ganz normalen Zugriff (RAM, EEPROM) der CPU.

Eine andere Art des Zugriffs auf externe Speicher wird mit Blocktransfers erreicht. Hier geht jedoch der individuelle Speicherzugriff auf Typen etc. wiederum verloren. Für manche Anwendungen jedoch durchaus eine sinnvolle Implementation. Weiteres weiter unten unter **BlockDevice.**

3.53.1 Organisation

1. Dem Compiler muss mitgeteilt werden, dass ein solches Device existiert: **Import** UserPort;
2. Die Grösse in Bytes des externen Speichers wird bestimmt: **Define** UserPort = nnn;
3. Mit einem Compilerschalter müssen bestimmte Variablen in das Device plaziert werden. Alle nachfolgend definierten Variablen legt der Compiler (bis auf Widerruf) in diesen Bereich: **{$UDATA}**
4. Es muss eine Prozedur definiert werden, welche die Initialisierung des externen Devices vornimmt. Diese Prozedur wird zur Initialisierungs Zeit vom System aufgerufen: **UserDevice** UsrDevIni
5. Eine Prozedur, die Byte-weise ins Device schreibt, wird gebraucht: **UserDevice** UsrDevOut
6. Eine Funktion, die Byte-weise vom Device einliest, wird definiert: **UserDevice** UsrDevInp

Import *aa, bb, UserPort;*
Diese Import Direktive weist den Compiler an ein UserPort und die zugehörigen Mechanismen bereitzustellen.

Define *UserPort = nnn;*
Das Define bestimmt die Grösse in Bytes des UserDevices

{$UDATA}
Der Compilerschalter bestimmt, dass alle nachfolgenden Variablen im UserDevice liegen sollen.

3.53.2 Formale Deklarationen

UserDevice *UsrDevIni;*
Falls notwendig, müssen hier Port und Device Initialisierungen erfolgen.

UserDevice *UsrDevOut (**const** adr : word; **const** outp : byte);*
Das System übergibt an diese Routine eine 16Bit Adresse und ein Byte das darin abgespeichert werden soll.

UserDevice *UsrDevInp (**const** adr : word) : byte;*
Das System übergibt an diese Routine eine 16 Bit Adresse von der ein Byte gelesen werden soll. Das gelesene Byte wird als Ergebnis zurückgeliefert.

Die Namensgebung für die obigen 3 Prozeduren ist **zwingend** vorgeschrieben.

```
Program AVR_UsrDataI2C;

Device = 90S8515;

Import SysTick, UserPort, I2Cport;

From System Import longword;

Define
     ProcClock   = 8000000;     {Hertz}
     SysTick     = 10;          {msec}
     StackSize   = $0020, iData;
     FrameSize   = $0010, iData;
     UserPort    = 2048;        // define Mem expansion size
     I2Cport     = PortB;       {24C16}
     I2Cdat      = 0;
     I2Cclk      = 1;

Implementation

{$IDATA}
{-----------------------------------------------------------------}
{ Type Declarations }

type
  ptB    = pointer to byte;
  ptW    = pointer to word;

{-----------------------------------------------------------------}
{ Const Declarations }
const
 I2Cadr    = $50;
```

```
{-------------------------------------------------------------}
{ Var Declarations }
var
   Timer1      : SysTimer8;
   bb          : byte;
   ww          : word;
   testB       : Byte;
   testCh      : char;
   testW       : word;
   testI       : integer;
   testL       : longword;
   testStr     : string[8];
   st          : string[5];
   wxyz        : Word;
   pp          : ptw;

{$UDATA}                          // place vars into User Device
var
  usrb         : byte;
  usri         : integer;
  usrl         : longword;
  usrArr       : array[0..10] of byte;
  uStr         : string[8];
  uRec         : record
                   ub : byte;
                   uw : word;
                   uSt: string[4];
                 end;
  pwDirT1      :  ptw;
  pbEepAdr1    :  ptb;
  prDir        : pointer to word;

{$IDATA}                          // reset to internal RAM
{-------------------------------------------------------------}
{ functions }

(* for a UserDevice the following 3 function must be implemented *)
UserDevice UsrDevIni;
begin
  (* is called at System Init Time *)
  (* initialize Device Hardware    *)
  (* for I2C nothing required      *)
end;

UserDevice UsrDevInp (adr : word) : byte;
var ok  : boolean;
   inp : byte;
begin
//  DisableInts;

  (* because of a possible previous write access   *)
  (* the I2Cout function can fail upto 10msec      *)
  (* and a timeout loop must be implemented        *)
  (* don't forget your watchdog                    *)
  SetSysTimer(Timer1, 2);
  repeat
    ok:= I2Cout (I2Cadr +hi (adr), lo (adr));      // set the read address
  until isSysTimerZero (Timer1) or ok;
```

```
  if ok then
    I2Cinp (I2Cadr, inp);                    // read the byte
    return (inp);
  else
//  error ...
  endif;

//  EnableInts;
  return($ff);
end;

UserDevice UsrDevOut (adr : word; outp : byte);
var ok  : boolean;
begin
//  DisableInts;
  (* because of a possible previous write access   *)
  (* the I2Cout function can fail upto 10msec      *)
  (* and a timeout loop must be implemented        *)
  (* don't forget your watchdog                    *)
  SetSysTimer (Timer1, 2);
  repeat
    ok:= I2Cout (I2Cadr +hi (adr), lo (adr), outp);
  until isSysTimerZero (Timer1) or ok;
//  EnableInts;
// if not ok then ....
end;

{-------------------------------------------------------------}
{ Main Program }

begin
  uStr[1]:= 'A';
//  uStr:= st;              // not implemented
//  uStr:= 'ABCD';          // not implemented
//  TestStr:= uStr;         // not implemented

  uRec.uw:= $1234;
  uRec.ub:= $AA;
  uRec.uSt[2]:= 'X';

  testB:= uRec.ub;
  testW:= uRec.uw;
  testCh:= uRec.uSt[2];

  usrb:= 155;
  usri:= -155;
  usrl:= 123456;
  usrArr[3]:= 44;

  testB:= usrb;
  testI:= usrI;
  testL:= usrL;
  testB:= usrArr[3];

  loop
    NOP;
  endloop;
end AVR_UsrDataI2C.
```

Das UserDevice/UDATA unterstützt auch Record, Array und String Kopier Funktionen.

```
UserRec:= Record1;
Record1:= UserRec;
UserArr:= Array1;
Array1:= UserArr;
UserStr:= String1;
String1:= UserStr;
```

Nicht unterstützt werden hier jedoch Pointer to Record, Array, String.
Auch ist bei den UserDev Strings ein String-Concat nicht möglich.

3.53.3 BlockDevice

Im Gegensatz zu einem sog. UserDevice arbeitet ein BlockDevice nicht Byte bzw. Typ orientiert, sondern Block-orientiert. Das heisst es werden immer ganze Speicherblöcke kopiert. Das Kopieren übernimmt hier ebenfalls eine vom Anwender bereitzustellende Routine. Die Quelle oder das Ziel einer solchen Kopier-Operation wird von der Konstruktion des Treibers bestimmt und kann deshalb beliebig sein, auch externe Devices sind möglich.

Für die System Funktion **ReadBlock** muss mindestens eine entsprechende Anwendungs Funktion zur Verfügung stehen, das gleiche gilt natürlich auch für **WriteBlock.**
Ein spezieller Import oder Define ist nicht notwendig.

Für **ReadBlock** muss eine Prozedur noch folgenden Schema vorhanden sein. Der Name ist beliebig. Wenn sichergestellt ist, dass alle Bytes übertragen werden, kann der Transfer mit einer Prozedur erfolgen:

```
Procedure DriverInpP (adr : pointer; size : word);
begin
  (* very simple implementation of a Block Device   *)
  (* because the calling library block function     *)
  (* doesn't receive a transfer count from here     *)
  (* this procedure !must! transfer "size" bytes    *)
  (* The destination address is "adr"               *)
  (* The source adr is the job of this service      *)
  CopyBlock(@source, adr, size);              // possible action
end;
```

Falls der Transfer nicht sicherstellen kann, dass auch die gewünschte Anzahl von Bytes transferiert werden, muss mit einer Funktion gearbeitet werden:

```
Function DriverInpF (adr : pointer; size : word) : word;
begin
  (* very simple implementation of a Block Device   *)
  (* requested transfer count in "size"             *)
  (* actually transferred bytes count returned      *)
  (* The destination address is "adr"               *)
  (* The source adr is the job of this service      *)

  CopyBlock (@source, adr, size);             // possible action
  Return(size);  // actual transferred byte count
end;
```

Die System Funktionen **ReadBlock** können jetzt mit einer der obigen Funktionen als Argument aufgerufen werden:

```
(* read-in the var arr2 with max count2 bytes      *)
(* count2 is limited to sizeOf(arr2)               *)
(* return the number of bytes read in result       *)

Count2:= 3;
result:= ReadBlock (DriverInpF, arr2, count2);
result:= ReadBlock (DriverInpP, arr2, count2);

(* read-in the var arr2 with the sizeOf(arr2)      *)
(* return the number of bytes read in result       *)

result:= ReadBlock (DriverInpF, arr2);
result:= ReadBlock (DriverInpP, arr2);
```

Für **WriteBlock** muss eine Prozedur noch folgenden Schema vorhanden sein. Der Name ist beliebig. Wenn sichergestellt ist, dass alle Bytes übertragen werden, kann der Transfer mit einer Prozedur erfolgen:

```
Procedure DriverOutP (adr : pointer; size : word);
begin
  (* very simple implementation of a Block Device   *)
  (* because the calling library block function     *)
  (* doesn't receive a transfer count from here     *)
  (* this procedure !must! transfer "size" bytes    *)
  (* The source address is "adr"                    *)
  (* The destination adr is the job of this service *)

  CopyBlock (adr, @dest, size);              // possible action
end;
```

Wenn es möglich ist, daß die vom Benutzer bereitgestellte Implementation nicht die gewünschte Anzahl von Bytes am Stück schreiben kann, wird eine Funktion benötigt, welche die genaue Anzahl der geschriebenen Bytes zurückliefert:

```
Function DriverOutF (adr : pointer; size : word) : word;
begin
  (* very simple implementation of a Block Device   *)
  (* requested transfer count in "size"             *)
  (* actually transferred bytes count returned      *)
  (* The source address is "adr"                    *)
  (* The destination adr is the job of this service *)

  CopyBlock (adr, @dest, size); // possible action
  return(size);                 // actual transferred byte count
end;

(* write-out the var arr1                          *)
(* pass the numbers of bytes to write in count1    *)
(* count1 is limited to sizeOf(arr1)               *)
(* return the number of bytes written in result    *)

count1:= 3;
result:= WriteBlock (DriverOutF, arr1, count1);
result:= WriteBlock (DriverOutP, arr1, count1);

(* write-out the var arr1 with the sizeOf(arr1) *)
(* return the number of bytes written in result *)
```

result:= WriteBlock (DriverOutF, arr1);
result:= WriteBlock (DriverOutP, arr1);

Wenn ein typisierter Pointer benutzt wird, kann auch damit gearbeitet werden, da der Compiler die Struktur und damit die Grösse der dahinterliegenden Variablen kennt.

result:= WriteBlock (DriverOutF, Ptr^, count1);
result:= WriteBlock (DriverOutP, Ptr^, count1);
result:= WriteBlock (DriverOutF, Ptr^);
result:= WriteBlock (DriverOutP, Ptr^);

Notizen

Grombacherstr. 27
D74906 Bad Rappenau

Tel. 07268/9124-0
Fax. 07268/9124-24

Internet: www.e-lab.de
e-mail: info@e-lab.de