Profi Treiber Handbuch

E-LAB AVRco

Pascal Multi-Tasking für Single Chips

Version für

AVR

Blaise Pascal Mathematiker 1623-1662

23 Oct. 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 Microcontroller 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 Graphic 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 sowas ähnliches wie Steuerung.

Viele Jahre, ja Jahrzehnte, 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 Graphic LCDs, Filesysteme 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 k.

2.2 Treiber und Handbuch Versionen

Dieses Handbuch bezieht sich auf Treiber die ausschliesslich in der AVRco 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 Profi-Version

3.1 I2Cexpand_5 Treiber für bis zu 40 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 PCA9698. Dieser enthält ein 5 PORT-Register, 5 PIN-Register und 5 DDR-Register und ist damit vollkommen gleichwertig zu fünf AVR Ports. Es können bis zu 8 solche Bausteine an den Bus angeschlossen werden. Damit können **40 Ports** bzw. **320 IO-Bits** realisiert werden.

Einführung I2Cexpander_5
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 der vorhandenen TWIs importiert werden: TWI_C, TWI_D, TWI_E oder TWI_F.

Als I2C Port-Expander muss pro **5 Ports** der Typ PCA9698 von Philips verwendet werden. Dieser Baustein kann bis zu 8 mal am Bus vorhanden sein. Der PCA9698 kann mit bis zu 1MBit/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 PCA9698 kann fast beliebig im I2C Adress Bereich von $10..$6F liegen. Da der Treiber jedoch bis zu 8 Chips adressieren kann, muss die notwendige Adress Einstellung jeweils mit Modulo 8 beginnen, $10, $18, $20 etc.

Die möglichen Ports haben die Namen ***PORT00…PORT39, PIN00…PIN39, DDR00…DDR39***.
Hierbei hat z.B. PORT00..PORT04 die I2C-Adresse $18, PORT05..PORT09 die I2C Adresse $19 etc.

Als Besonderheit haben diese I2C Chips die Möglichkeit Input Pins zu invertieren. Dazu werden die Spezial Ports INP_POL00..INP_POL39 exportiert. Eine log1 invertiert das zugehörige Input Bit.

Die PORT, PIN und DDR Register arbeiten exakt in der gleichen Weise wie ihre Gegenstücke im AVR. Deshalb ist es nicht notwendig irgendwas über die Internas eines PCA9698 zu wissen. Man betrachte diese einfach als weitere AVR ports.

3.1.1 Technische Daten

I2C Port	Software I2C importiert durch I2Cport
oder	CPU-TWI importiert durch TWImaster
oder	CPU-TWI importiert durch TWInet im Mastermode
XMegas	
	CPU-TWI importiert durch TWI_C, TWI_D, TWI_E oder TWI_F
Hardware	I2C I/O-Expander Chip PCA9698 von Philips, 1 Stück pro 5 Ports
I2C Adressen	Die PCA9698 liegen auf den Bus-Adressen $10..$17 oder $18, $1F oder …
	wobei PORT00..04 die Adresse $10 hat, PORT05..09 hat $11 etc.

Die PCA9698 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_5;
oder
Import  TWImaster, I2Cexpand_5;
oder
Import  TWInet, I2Cexpand_5;                // use Master mode
XMega
Import  TWI_C, I2Cexpand_5;                 // use TWI_C, TWI_D, TWI_E or TWI_F
```

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_5    = I2C_Soft, $10;    {use Software I2Cport}
        I2CexpPorts_5  = Port00, Port05;   {use Port00..04. and Port05..09  = 2x PCA9698}
```

Beispiel für *TWImaster*:

```
Define  ProcClock      = 8000000;          {8Mhz clock }
        TWIpresc       = TWI_BR100;        {100kBit/sec alt. TWI_BR400}
        I2Cexpand_5    = I2C_TWI, $18;     {use TWIport}
        I2CexpPorts_5  = Port00, Port05;   {use Port00..04. and Port05..09  = 2x PCA9698}
```

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_5    = I2C_TWI, $20;     {use TWIport}
        I2CexpPorts_5  = Port00, Port05;   {use Port00..04. and Port05..09  = 2x PCA9698}
```

Beispiel für *XMega*:

```
Import  TWI_C, I2Cexpand_5;                 // 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_5      = TWI_C, $18;      {use TWIport C}
        I2CexpPorts_5    = Port00, Port05;  {use Port00..04. and Port05..09  = 2x PCA9698}
```

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 die I2C-Basis Adresse: $10, $18, $20, $28 etc.

I2CexpPorts
Definiert welche und wieviel Ports unterstützt werden sollen. Zulässig sind die Angaben von Port00, Port05, Port10, Port15, Port20, Port25, Port30 und Port35 wobei jedes Define eine Gruppe von 5 Ports = 1x PCA9698 umfasst.

3.1.2 Typen und Funktionen

Der Import von I2Cexpand veröffentlicht eine spezial Type:

```
Type TI2CPORT_5 = internal;
```

Dieser Typ kann dazu benutzt werden, um den einzelnen Ports eine aussage kräftigeren Namen zu geben.

```
Var myName[@Port00] : TI2Cport_5;
```

Mit *myName* kann jetzt das Port00 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_5 (Port: TI2Cport_5) : boolean;
```

Diese Funktion gibt ein true zurück, wenn das Selektieren des PCA9698 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 möglichen Operationen mit diesen Ports sind aus dem Beispiel Programm zu ersehen.

3.1.3 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  for TWI_C…TWI_F
```

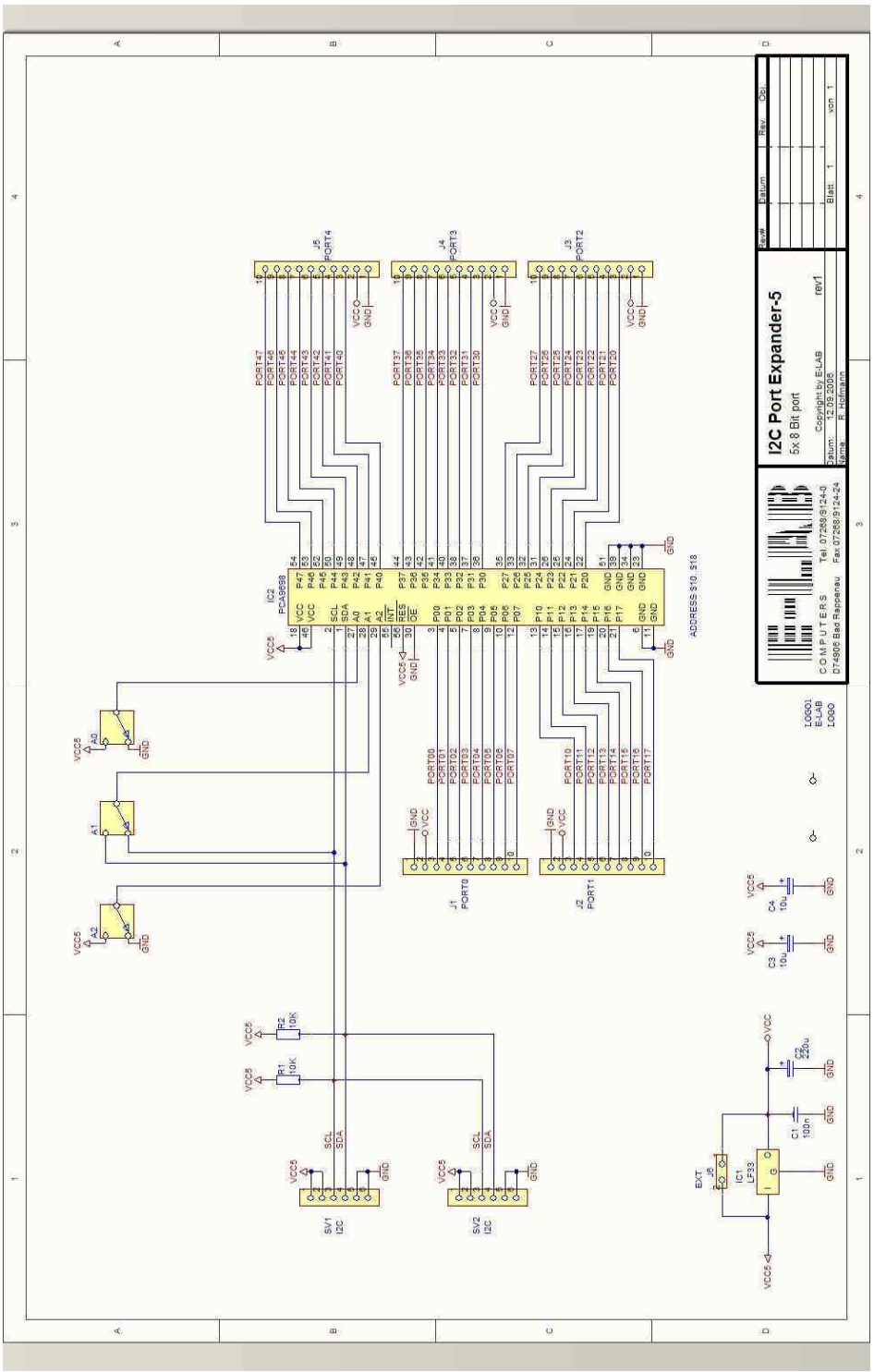
Der TWI Treiber beachtet und steuert diese Semaphore. Beim Eintritt in den Treiber wird geprüft ob der Treiber frei ist (Semaphore inaktiv). Ist das der Fall, dann wird die Semaphore aktiviert = Treiber gesperrt und der Auftrag wird ausgeführt. Nach dem Beenden des Auftrags wird die Semaphore wieder 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_5**
ein Xmega Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Xmega_ I2Cexpand5**



Schaltplan **I2Cexpand_5**

3.2 SpeechPort Sprach Ausgabe

In manchen Applikationen ist es sinnvoll oder notwendig bestimmte Informationen auch akustisch auszugeben.
Insbesondere, wenn eine Anzeige nicht einsehbar ist oder nicht immer im Blickfeld ist. Man kann hier mit einem Beeper zumindest die Aufmerksamkeit des Benutzers erreichen. Dann muss allerdings doch in den meisten Fällen das Bedien Tableau eingesehen werden.

Wesentlich eleganter ist es aber, Meldungen direkt in Sprache auszugeben. Dazu dient dieser Treiber.

Einführung SpeechPort

Dieser Treiber dient dazu, WAV Dateien abzuspielen. Diese Dateien müssen dazu folgendes Format haben:

8bit, Mono, 11kHz

Das zum AVRco System gehörige Konvertierungs Programm „***WAVconvert.exe***“ kann dazu benutzt werden, um fast beliebige WAV Files/Formate in obiges WAV Format zu konvertieren.

Um eine saubere Sprache zu erhalten, muss die Ausgabe über einen Timer Interrupt erfolgen. Dazu kann jeder AVR Timer herangezogen werden, sofern er über eine Capture/Compare Unit verfügt, was aber nicht immer der Fall ist. Weiterhin werden in schneller Folge Interrupts ausgeführt, so dass ein CPU Clock > 8MHz ratsam ist.

Da diese Interrupts im Abstand von ca. 90usec kommen, erzeugt natürlich jeder andere Interrupt, UART, Timer, MultiTasking Handling etc. gewisse Störungen, die sich durch eine mehr oder weniger rauhe Wiedergabe bemerkbar machen. Um dies zu verhindern, kann man temporär z.B. durch Sperrungen Abhilfe schaffen. In den meisten Fällen ist dies jedoch nicht notwendig.

WAV Files sind relativ gross. Man muss hier mit 11kByte pro Sekunde Spielzeit rechnen. Mehrere kurze Ansagen sind deshalb beim Mega128 noch problemlos in dessen Flash unterzubringen. Grössere Files müssen auf einem separaten Medium gespeichert werden. Das kann externes Flash sein, eine MMC Karte oder TCP/IP ist auch denkbar. Eine direkte Unterstützung dafür kann der Treiber aber nicht leisten.

Zum Lesen der Daten stehen zwei Funktionen zur Verfügung, eine für das Lesen aus dem internen Flash und eine für das Lesen aus dem internen RAM. Ein direktes Lesen von einem anderen Medium ist nicht möglich.

Bei Files aus einem externen Medium kann nur der RAM Mode verwendet werden. Da das interne RAM sehr beschränkt ist, muss ein RAM-Buffer angelegt werden, wo das File block-weise hinein kopiert wird um anschliessend vom Treiber ausgegeben zu werden. Hier bietet sich eine Doppel-Puffer Strategie an. Dabei wird im Wechsel ein Buffer beschrieben und der andere vom Treiber gelesen. Dazu bietet der Treiber ein READY Flag an, das anzeigt, wenn ein Buffer komplett ausgegeben wurde.

Der Treiber unterstützt drei unterschiedliche Ausgabe Möglichkeiten. Bei den zwei bit-seriellen Modi erfolgt die Ausgabe entweder über den AVR SPIport oder über einen beliebiges Portpin(s). Die 8bit parallel Ausgabe erfolgt über ein 8bit Port des AVR. Eine generelle Ausgabe kann auch über ein UserDevice erfolgen. Die beiden bitseriellen Modi erwarten einen seriellen 8bit DA-Wandler. Die parallel Port Ausgabe erwartet einen 8bit parallel DA-Wandler. Dem UserDevice werden die Daten Byte-weise übergeben.

Beim bit-seriellen Port Modus wird ein Software-SPI Port gebildet. Beim Define dieses Ports muss das erste benutzte Bit von diesem Port angegeben werden. Dieses wird zum Daten Bit. Das nächste Bit wird zum Clock Bit und dessen nächstes Bit wird zum Select bzw. Enable Bit. Dieses Bit ist low-active.

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

```
Import SpeechPort ;
```

Der Import von SpeechPort importiert auch automatisch diverse Bibliotheksfunktionen.
Wenn die Ausgabe über den SPI erfolgen soll muss auch der SPIdriver importiert werden.

```
Import  SpeechPort, SPIdriver;
XMega
Import SpeechPort;
```

Defines
Der Treiber benutzt einen internen 8 oder 16bit Timer, Timer0..Timer3, falls vorhanden.

```
Define
        ProcClock     = 8000000;      {8Mhz clock }
        SpeechPort    = SPI;          // SPIdriver must be imported and defined
        SPIorder      = LSB;          // SPI define only necessary if SpeechPort = SPI;
        SPIcpol       = 1;
        SPIcpha       = 1;
        SPIpresc      = 0;            // presc = 0..3 -> 4/16/64/128
        SPI_SS        = true;         // use SS pin as chipselect
oder
        SpeechPort    = UserPort;     // uses an user defined driver
oder
        SpeechPort    = PortG, 0;     // bit serial using PortG, bit0=DATA, 1=CLK, 2=SEL
oder
        SpeechPort    = PortC;        // 8bit parallel output on port C
        SpeechTimer   = Timer0;       // Timer0, Timer1, Timer2, Timer3
XMega
   SpeechTimer   = Timer_C0;
   SpeechPort    = SPI_C, SPImode3, SPImsb, PortF, 4;  // Mode0..3, MSB/LSB, SS-Port, SS-Pin
```

Wird das UserPort als Ausgabe gewählt, muss die Applikation auch den Treiber zur Verfügung stellen:

```
UserDevice SpeechIOS(b : byte);
begin
  ...
end;
```

3.2.1 XMega und XMega-DAC

Als Output Device kann beim XMega auch der interne DAC verwendet werden:

```
Import …, SpeechPort, DAC_B, …;
…
Define
  …
  DAC_B          = chan01, REFextB;   // DAC_B channel 0 + 1 defined
  SpeechTimer    = Timer_D1;          // use Timer_D1
  SpeechPort     = DAC_B1;            // use DAC_B1
```

Wird der interne DAC verwendet, dann wird auch diese Prozedur exportiert:

```
procedure SpeechSetGain(gain : byte);        // 0..4
```

hiermit kann zur Laufzeit die Verstärkung eingestellt werden.

Gain 0 = vOut x0 = off
Gain 1 = vOut x1
Gain 2 = vOut x2
Gain 3 = vOut x4
Gain 4 = vOut x8

3.2.2 Funktionen und Prozeduren

SpeechOutFlash
Mit dieser Prozedur wird ein Datenblock aus dem Flash ausgegeben. Die Definition ist:

Procedure *SpeechOutFlash(start : pointer; count : word);*

Der Parameter **start** ist ein Pointer auf den Datenblock, **count** ist die Anzahl der Bytes die ausgegeben werden sollen.

Achtung:
zeigt der Pointer auf den ersten Block eines WAV Files oder ist das komplette WAV File in diesem Block enthalten, so muss mit einem Offset von 36Bytes gearbeitet werden, da der WAV Header so lang ist. Der Byte Count muss dann um 36 erniedrigt werden.

SpeechOutRAM
Mit dieser Prozedur wird ein Datenblock aus dem RAM ausgegeben. Die Definition ist:

Procedure *SpeechOutRAM (start : pointer; count : word);*

Der Parameter **start** ist ein Pointer auf den Datenblock, **count** ist die Anzahl der Bytes die ausgegeben werden sollen.

Achtung:
zeigt der Pointer auf den ersten Block eines WAV Files oder ist das komplette WAV File in diesem Block enthalten, so muss mit einem Offset von 36Bytes gearbeitet werden, da der WAV Header so lang ist. Der Byte Count muss dann um 36 erniedrigt werden.

SpeechReady
Mit dieser Funktion kann abgefragt werden, ob die Ausgabe beendet ist. Wird bei Doppel Pufferung benötigt. Die Definition ist:

Function *SpeechReady : boolean;*

SpeechStop
Hiermit kann eine laufende Ausgabe abgebrochen werden. Die Definition ist:

Procedure *SpeechStop;*

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\Speech**
ein XMega Beispiel befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\XMega_Speech**
ein Xmega-DAC Beispiel befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\XMega_Speech**

3.3 IR RxPort InfraRed generic Treiber für RC5, NEC, Samsung etc

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.

Oft bleibt dann 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.

Der IR_RxPort Treiber stellt ein IR Empfänger dar, der diverse IR Protokolle einzeln oder auch gemeinsam empfangen und auswerten kann:

RxModes: IR_SONY, IR_SANYO, IR_NEC, IR_LG, IR_SAMSUNG, IR_RC5, IR_RC6

3.3.1 Receiver

Der Receiver kann an jeden beliebigen Input-Pin angeschlossen werden. Der Treiber arbeitet hier im Interrupt-Polling mit 16bit Timer. 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 RxPort Receiver

```
Define
  IR_RxPort       = PinE.0, Negative;    // or Positive

  // RxMode: IR_SONY, IR_SANYO, IR_NEC, IR_LG, IR_SAMSUNG, IR_RC5, IR_RC6
  IR_RxMode       = RC5, Samsung, NEC;

  IR_RXLED        = PortR.1, Negative;   // or Positive. Define of the LED is optional
  // XMega
  IR_RxTimer      = Timer_E0;            // Timer_C0..Timer_F1
  // AVR
  IR_RxTimer      = Timer1;              // Timer1 or Timer3

Uses uIR_Rx;
```

3.3.2 Exportierte Typen Receiver

```
TDecodeType   =       (UNUSED,
                       RC5,
                       RC6,
                       NEC,
                       NEC_EXT,
                       SONY,
                       SAMSUNG,
                       LG,
                       SANYO,
                       UNKNOWN = $FF);

TDecodeRes    =    record
                       DecodeType  : TDecodeType;// UNKNOWN, NEC, SONY, RC5, ...
                       Address     : word;       // 16bits Used by Panasonic & Sharp
                       Value       : word;       // Decoded value [max 16bits]
                       Overflow    : boolean;    // true if IR raw code too long
                   end;
```

3.3.3 Exportierte Funktionen Receiver

Procedure *IR_Start;*
Startet den Timer und das Empfangs System

Procedure *IR_stop;*
Stoppt den Timer und das Empfangs System.

Function *IR_isIdle : boolean;*
Gibt ein false zurück wenn gerade ein Paket empfangen wird

Function *IR_Decode(var Results : TDecodeRes) : boolean;*
Wenn der Receiver gestartet wurde und *IR_isIdle* mit true zurückkommt, dann kann mit IR_Decode das Paket abgeholt werden.

Function *IR_DecodeTypeStr(Index : byte) : string[12];*
Hiermit kann der Klarname des empfangenen Pakets als string angezeigt werden.

Procedure *IR_resume;*
Nach einem *IR_Decode* muss der Receiver wieder mit *IR_Resume* freigegeben werden.

Achtung;
Einige Protokolle bringen eine $00 im Address Feld und ein $FFFF im Value Feld wenn dasselbe Kommando wiederholt gesendet wird. Repeat mode.

Da hier ein 50usec Timer Interrupt läuft sollten andere Interrupts nicht zu lange aktiv sein (Interrupt Sperrung) um Störungen des Empfangs zu vermeiden.

Die Unit „uIR_Rx“ muss importiert werden.

Das Define *IR_RXLED* ist optional und importiert einen LED Treiber Portpin womit eine LED geflasht werden kann wenn ein Frame empfangen wird.

Beispiel Programm für AVR/Mega im Verzeichnis **..\E-LAB\AVRco\Demos\IR_RxPort\IR_RxPortA**
Beispiel Programm für XMega im Verzeichnis **..\E-LAB\AVRco\Demos\IR_RxPort\IR_RxPortX**

3.4 CAN Treiber

Allgemeines
Wird eine sehr schnelle, hoch sichere, zuverlässige aber preiswerte Datenverbindung benötigt, so ist der CAN BUS erste Wahl. Zwar bietet Ethernet technisch die gleichen Eigenschaften Eigenschaften und einen wesentlich höheren Datendurchsatz, ist aber system bedingt wesentlich teuerer und verbraucht mehr System Resourcen.

Der CAN arbeitet mit dem bewährten und extrem zuverlässigen und robusten (RS485) Differential Verfahren, das nur zwei Leitungen, möglichst verdrillt, und Masse benötigt. Hiermit sind Datenraten bis zu 1Mbit/sec erreichbar, abhängig von der Leitungslänge.

Der CAN BUS ist ein Meldungs (Message) orientiertes System, d.h. die im Telegramm enthaltene Kennung ist gleichzeitig auch eine Information und BUS Priorität in einem und wird ***Message Identifier*** genannt.

In dem durch die Firma Bosch definierten CAN 2.0A Format wird dieser als *Standard Identifier* (*SID*) bezeichnet und ist 11bit lang. Als „Nutzlast" können zusätzlich 0 bis 8 Datenbytes angefügt werden, deren Anzahl im *Data Length Code* (*DLC)* Feld angegeben wird. Dieses erscheint auf dem ersten Blick als recht wenig. Da aber die zugehörige ID auch Informationsträger ist ergeben sich vielfältige Möglichkeiten.

In der erweiterten CAN Norm 2.0B wird zusätzlich zu dem SID ein *Extended Identifier* (*EID*) mit 29 Bit Länge definiert, dessen oberen 11 Bit (MSB) mit dem SID überlappen.

Diese *Identifier* sollten nicht als Adresse verstanden werden. Besser als eine Absender, Objekt- oder Nachrichtenkennung. Bedingt durch das Protokoll können auch nicht alle möglichen Werte des *Identifiers* benutzt werden. So stehen von den theoretisch möglichen 2048 SID ($000-$7FF) Werten nur 1983 Werte ($000-$7BF) zur Verfügung.

Man kann wahlweise einen reinen SID oder EID CAN BUS betreiben oder im sog. Mixed Mode beide Format gleichzeitig einsetzen. Ein Identifier Extension Bitflag (IDE) kennzeichnet EID Nachrichten.

Wer weitere Details zur CAN BUS Norm erfahren möchte, dem seien die „Bosch CAN 2.0 Specs.pdf" zum Lesen wärmstens empfohlen. Auch ein Studium des Prozessordatenblatts zum AT90CANxx sollte nicht fehlen, da sich, dank der dem Compiler vollständig bekannten Register, die Möglichkeit besteht, zusätzlich zu den in dieser Unit vorbereiteten Funktionen weitere Besonderheiten des eingebauten CAN Controllers zu nutzen.

Es gibt weder einen Master noch einen Slave, Server oder Client. Alle Teilnehmer sind gleichberechtigt und können zu jeder Zeit senden oder empfangen. Das führt natürlich unweigerlich zu Kollisionen welche vom CAN hervorragend und automatisch aufgelöst werden. Der CAN BUS besitzt dafür ein spezielles „Schlichtungsverfahren" (***Arbitration***), welches im Gegensatz zu anderen Kollisionsverfahren, keinen Zeitverlust verursacht.

Dabei setzt sich normalerweise die Message (Packet) durch welche als Identifier die höchste Priorität hat. Gleichzeitig wird aber auch sichergestellt, dass niedriger eingestufte Packete trotzdem mit kurzer Verzögerung zum Zug kommen. Die Prorität wird, bedingt durch das Schlichtungsverfahren, durch den niedrigeren Binärwert bestimmt. Also $000 hat die höchste und $7FF (beim SID) die niedrigste Priorität. Per Definition haben SID Nachrichten, bei gleichem 11 Bit Wert mit den 11 MSB eines EID, den Vorrang vor EID Nachrichten.

Nachrichten können grundsätzlich gleichzeitig von allen Teilnehmern empfangen und ausgewertet werden, sofern deren Filter diese Packete akzeptieren. Damit nun nicht alle Pakete per AVRCo Programmcode ausgewertet werden müssen, besitzen CAN Controller, wie der im AT90CANxxx Baustein, eine Filterlogik, die die Anzahl per Programm auszuwertender Pakete drastisch reduzieren kann. Die Message ID ist ähnlich einer IP Adresse aber je nach Filter Einstellungen auch für mehr als ein Teilnehmer wirksam.

3.4.1 AT90CAN32/64/128

Diese AVR Bausteine bieten 15 sogenannte Message Objects (MOBs) die man auch als Briefkästen bezeichnen kann. Der vorliegende Treiber benutzt die BOX 0 zum Senden und die Boxen 1..14 sind frei zum Empfangen.

Zum eleganten Arbeiten mit dem CAN BUS arbeiten alle Empfangs Boxen auf eine PIPE (FIFO) . Für das Senden steht eine weitere PIPE zur Verfügung. Der Vorteil davon ist, dass der Treiber im Hintergrund die Rx PIPE füllen kann während die Applikation beschäftigt ist. Die Applikation wiederum stellt ihre Ausgangs Packete in die Tx PIPE wo sie der Treiber sequentiell ausliest und verschickt.

Der SysTick wird nur gebraucht wenn die Systemzeit benutzt werden soll. Der Treiber selbst arbeitet im Interrupt Betrieb.

Imports
Wie beim AVRco System üblich, muss der Treiber importiert und definiert werden. Der Import von CAN_AVR importiert auch automatisch diverse Bibliotheksfunktionen.

```
Import SysTick, CAN_AVR ;
```

Defines
Der Treiber benötigt als Info die Grösse der beiden PIPEs und die initiale Baudrate..

```
Define
        ProcClock       = 16000000;             // 16Mhz clock
        CAN_AVR         = 16, 16, iData;        // RxPipe, TxPipe, memory
        CAN_AVRbaud     = CAN_Baud125;
```

Als Option kann in jede Empfangs Message ein Zeitstempel (*Timestamp*) eingefügt werden. Dieser Zeitstempel kann wahlweise 16 oder 32 Bit umfassen und bei 16 Bit wahlweise durch die Systemzeit oder durch die im CAN Controller geführten Timestampzähler gebildet werden.

Für die Systemzeit ist der globale Import der Systemzeit notwendig als auch der Import in den CAN Treiber:

```
From SysTick import SystemTime16;               // or SystemTime32

Define CAN_AVR    = 16, 16, iData, CAN_SysTime;   // RxPipe, TxPipe, memory, Systemtime
```

Möchte man den internen 16 Bit Timestampzähler verwenden ist folgendes Define erforderlich:

```
Define CAN_AVR = 16, 16, iData, CAN_TimeStamp; // RxPipe, TxPipe, memory, Timestamp
```

3.4.1.1 Typen und Variable

Der Treiber exportiert diverse Typen die in der Kommunikation benützt werden müssen.

```
type

  tAVR_CAN_Flag = (CAN_AERR, CAN_FERR, CAN_CERR, CAN_SERR,CAN_BERR, CAN_RTR, CAN_IDE,
  CAN_DLCW);

  tAVR_CAN_Flags = BitSet of tAVR_CAN_Flag;

  tCANMessage = record
          MOBIdx      : byte;              // Message Object = Mailboxnumber, die die Nachricht annahm
          EID         : longword;          // Extended Identifier  ~ Accept mask 29 bits
          SID [@EID]  : word;              // Standard Identifier ~ Accept mask 11bits as Overlay
          TimeStamp   : word|longword;     // optional, only if CAN_TimeStamp or CAN_SysTime is defined
          Flags       : tAVR_CAN_Flags;    // Statusflags der Mailbox
          DLC         : byte;              // data length 0..8
          data        : array[0..7] of byte;
  end;

  tCAN_baud      = (CAN_Baud25, CAN_Baud50, CAN_Baud100, CAN_Baud125, CAN_Baud200,
                    CAN_Baud250, CAN_Baud500, CAN_Baud1000);

  tAVR_CAN_Stat = (CAN_ACKError, CAN_FrameError, CAN_CRCError, CAN_StuffError,
                   CAN_BitError, CAN_RxOK, CAN_TxOK, CAN_DLCwarn);
  tAVR_CAN_States = BitSet of tAVR_CAN_Stat;

var

  CAN_RxPipe: Pipe [defined] of tCANMessage;
  CAN_TxPipe: Pipe [defined] of tCANMessage;

  CAN_RejectFlags  : tAVR_CAN_States;
```

Diese Variable dient dazu bei bestimmten Fehlerbedingungen beim Empfang von vornherein die Nachricht zu ignorieren d.h. diese nicht in die RX Pipe einzustellen. Normalerweise werden alle empfangenen und mit den fRXOK Flag des Controllers gekennzeichneten Nachrichten in die RX Pipe eingestellt, um es der Anwendung zu überlassen, die Nachricht anhand der beigegebenen Statusflags selbst auszuwerten und gegebenenfalls zu ignorieren.

Beispiel:

```
CAN_RejectFlags := [CAN_Frameerror, CAN_CRCError];
```

Ignoriert Nachrichten mit einem Frame oder CRC Fehler.

Details zu den Fehlerarten finden Sie in der CAN 2.0 Spezifikation und im AVR Datenblatt.

3.4.1.2 Funktionen und Prozeduren

function *AVR_CAN_Init (RxMObCount : Byte) : boolean;*
Folgende Funktionen werden ausgeführt:
Die CAN Hardware wird durch einen Hardware - Reset komplett zurückgesetzt und neu initialisiert.
Alle Mailboxen werden entleert und neu initialisiert.
RxMObCount (1..14) bestimmt die Anzahl der für den Empfang zu initialisierenden Mailboxen.
Beide Pipes werden entleert
CAN_RejectFlags wird auf [] gesetzt und CANTCON auf 255 gesetzt, um die niedrigsten internen Timestamp-Frequenz zu erhalten.
Die *AVR_CAN_Enable* Funktion wird vor der Freigabe der CAN Interrupts aufgerufen.
Das Resultat ist in diesem Treiber immer true.

procedure *AVR_CAN_Disable;*
Sperrt den Treiber so dass die Applikation Änderungen an der Baudrate, den Filtern etc. vornehmen kann.

procedure *AVR_CAN_Enable;*
Gibt den Treiber nach einem Disable wieder frei.

function *AVR_CAN_BaudRate(br : tCAN_baud) : boolean;*
Stellt die Baudrate ein. Zuvor sollte ein Disable und danach ein Enable ausgeführt werden.

function *AVR_CAN_SetRxMask (yMBox: byte; wIDTag, wIDMask: word) : Boolean;*
Stellt den Standard ID-TAG und die ID-Mask für eine Rx-Mailbox (1..14) ein.
Beide Parameter zusammen bestimmen ob eine bestimmte Nachricht in dieser Box empfangen werden kann. Die Bedingung dafür ist:
(RxMsg.SID and wIDMask) = (wIDTag and wIDMask).

function *AVR_CAN_SetRxEMask (yMBox: byte; lwIDTag, lwIDMask: longword) : Boolean;*
Stellt den erweiterten ID-TAG und die ID-Mask für eine Rx-Mailbox (1..14) ein.
Beide Parameter zusammen bestimmen ob eine bestimmte Nachricht in dieser Box empfangen werden kann. Die Bedingung dafür ist:
(RxMsg.EID and lwIDMask) = (lwIDTag and lwIDMask).

function *AVR_Can_GetError(box : byte) : boolean;*
Prüft den aktuellen Status der Box auf aufgetretene Fehler. Beachten Sie, dass dieser Wert stets den letzten Zustand widerspiegelt und nicht unbedingt mit der aus der RX Pipe entnommene Nachricht übereinstimmen muss. Hier gibt das *Flags* Feld der Nachricht präzisiere Auskunft.

function *AVR_Can_GetStatus(box : byte) : tAVR_CAN_States;*
Gibt den aktuellen Status der Box zurück. Beachten Sie, dass dieser Wert stets den letzten Zustand widerspiegelt und nicht unbedingt mit der aus der RX Pipe entnommene Nachricht übereinstimmen muss. Hier gibt das *Flags* Feld der Nachricht präzisiere Auskunft.

function *AVR_CAN_RxErrCount : byte;*
Gibt Rx Error Count zurück. Jeder Fehler inkrementiert diesen Wert, jeder fehlerfreie Empfang dekrementiert ihn wieder.

function *AVR_CAN_TxErrCount : byte;*
Gibt Tx Error Count zurück. Jeder Fehler inkrementiert diesen Wert, jeder fehlerfreie Transmit dekrementiert ihn wieder.

procedure *AVR_CAN_StartMessage;*
Wurden ein oder mehrere Messages in die Tx-Pipe gestellt mit “PipeSend(CAN_TxPipe, Msg)“ muss diese Funktion aufgerufen werden um das Senden für eine Message/Nachricht freizugeben. Eine Pipe wird in einer loop komplett gesendet:

```
while PipeStat(CAN_TxPipe) <> 0 do
        AVR_CAN_StartMessage;
endwhile;
```

Beispiel Programme im Verzeichnis ..\E-LAB\AVRco\Demos\CAN_AVR: AVR CAN128M, AVR CAN128S

3.4.2 MCP2515

MCP2515
Diese preisgünstige CAN Controller Baustein wird über die SPI Schnittstelle angesteuert und bietet 6 sogenannte Message Objects (MOBs), die man auch als Briefkästen bezeichnen kann. Unabhängig von diesen Empfangsobjekten hat der Baustein 3 Sendepuffer, die vom Treiber automatisch ausgewählt werden. Als **wichtige Einschränkung** ist jedoch zu berücksichtigen, dass die 6 Empfangsobjekte, im Gegensatz zum AT90CANxxx Controller, voneinander nicht unabhängig sind, sondern sich in zwei Gruppen gliedern, die sich jeweils eine Annahmemaske (Acceptance mask) teilen. Die Mobs 1 und 2 bilden die eine und die Mobs 3 bis 6 die andere Gruppe. Dieses hat zur Folge, dass die Aufrufe der Filter und Maskenroutine (MCP_CAN_SetRxMask) sich gegenseitig beeinflussen. Sprich, der letzte Maskenwert eines Eingangs einer der Gruppen setzt den Maskenwert für alle anderen Eingänge.
Zum eleganten Arbeiten mit dem CAN BUS arbeiten alle Empfangs Boxen auf eine PIPE (FIFO). Für das Senden steht eine weitere PIPE zur Verfügung. Der Vorteil davon ist, dass der Treiber im Hintergrund die Rx PIPE füllen kann während die Applikation beschäftigt ist. Die Applikation wiederum stellt ihre Ausgangspakete in die Tx PIPE von wo sie der Treiber sequentiell ausliest und verschickt.

Der SysTick wird nur gebraucht, wenn die Systemzeit für den optionalen Zeitstempel benutzt werden soll. Der Treiber selbst arbeitet wahlweise im Interrupt oder im Polling Betrieb.

Imports
Wie beim AVRCo System üblich, muss der Treiber importiert und definiert werden. Der Import von CAN_2515 importiert auch automatisch diverse Bibliotheksfunktionen.

```
Import SysTick, CAN_2515 ;
```

Defines
Der Treiber benötigt als Info die Größe der beiden Pipes und die initiale Baudrate.

```
Define
ProcClock           = 16000000;          // 16 Mhz clock
CAN_2515            = 16, 16, iData; // RxPipe, TxPipe, memory
CAN_AVRbaud         = CAN_Baud125;
```

Als Option kann in jedes Empfangspaket ein Zeitstempel (Timestamp) eingefügt werden. Dieser Zeitstempel kann wahlweise 16 oder 32 Bit umfassen und wird durch die Systemzeit gebildet.

```
From SysTick import SystemTime16;          // or SystemTime32

Define CAN_2515 = 16, 16, iData, CAN_SysTime; // RxPipe, TxPipe, memory, Systemtime
```

3.3.1 Typen und Variable

Der Treiber exportiert diverse Typen die in der Kommunikation benützt werden müssen.

Type

```
tMCP_CAN_Flag = (CAN_RTR, CAN_IDE);

tMCP_CAN_Flags = BitSet of tMCP_CAN_Flag;

tCANMessage = record
MOBIdx          : byte;                 // Message Object = Mailboxnumber, die die Nachricht annahm
EID : longword;                         // Extended Identifier  ~ Accept mask 29 bits
SID [@EID] : word;              // Standard Identifier ~ Accept mask 11bits as Overlay
 TimeStamp : word|longword;     // optional, only if CAN_SysTime is defined
Flags : tMCP_CAN_Flags;         // Statusflags der Mailbox
DLC : byte;                     // data length 0..8
data : array[0..7] of byte;
end;
```

tCAN_baud = (CAN_Baud25, CAN_Baud50, CAN_Baud100, CAN_Baud125, CAN_Baud200, CAN_Baud250, CAN_Baud500, CAN_Baud1000);

tCAN_Mode = (CAN_RxInt, CAN_TxInt, CAN_Poll, CAN_RXonTx);
tMCP_CAN_Modes = BitSet of tCAN_Mode;

tMCP_CAN_State = (CAN_Rx0Full, CAN_Rx1Full,CAN_TX0Pending,CAN_TX0Empty, CAN_TX1Pending, CAN_TX1Empty,CAN_TX2Pending,CAN_TX2Empty);
tMCP_CAN_States = BitSet of tMCP_CAN_State;

tMCP_CAN_EFLG = (CAN_EWARN, CAN_RXWARN, CAN_TXWARN, CAN_RXEPASSIVE, CAN_TXEPASSIVE, CAN_TXBUSOFF,CAN_RX0OVERFLOW,CAN_RX1OVERFLOW);
tMCP_CAN_Errors = BitSet of tMCP_CAN_EFLG;

var

CAN_RxPipe: Pipe [defined] **of** tCANMessage;
CAN_TxPipe: Pipe [defined] **of** tCANMessage;

3.3.2 Funktionen und Prozeduren

function MCP_CAN_Init (rxMode: tMCP_CAN_Modes; yOPMode : byte) : boolean;
Folgende Funktionen werden ausgeführt:

- Die CAN Hardware wird durch einen Hardware - Reset komplett zurückgesetzt und neu initialisiert.
- Alle Mailboxen werden entleert und neu initialisiert.
- Gemäß **rxMode** wird der Empfangsbetrieb eingerichtet.
  - CAN_RxInt:
  - CAN_TxInt: Das Senden erzeugt einen TX Interrupt
  - CAN_RXonTx: Der Empfang erfolgt gepollt durch den regelmäßigen Aufruf von MCP_CAN_StartMessage
- Gemäß **yOPMode** wird die Betriebsart des Controller eingeschaltet (Details siehe Datenblatt)
  - OPMODE_NORMAL: Normalbetrieb
  - OPMODE_SLEEP: Schlaf/Energiesparbetrieb
  - OPMODE_LOOPBACK: Rückkopplungsschleife
  - OPMODE_LISTENONLY: passiver „Lauschbetrieb“
- Beide Pipes werden entleert
- Alle Filter und Masken werden gelöscht, so dass alle eingehenden Nachrichten angenommen werden.
- Die im EEProm gespeicherte Baudrate wird eingestellt.
- Die MCP_CAN_Enable Funktion wird vor der Freigabe der CAN Interrupts aufgerufen.

Das Resultat ist in diesem Treiber immer TRUE.

procedure MCP_CAN_Disable;
Sperrt den Treiber, so dass die Applikation Änderungen an der Baudrate, den Filtern etc. vornehmen kann.

procedure MCP_CAN_Enable;
Gibt den Treiber nach einem MCP_CAN_Disable wieder frei.

function MCP_CAN_SetBaudRate (BaudRate : tCAN_baud) : Boolean;
Stellt die Baudrate ein. Zuvor sollte ein MCP_CAN_Disable und danach ein MCP_CAN_Enable ausgeführt werden.
Die Unit speichert die gewählte Baudrate im EEProm und verwendet diesen Wert bei den MCP_CAN_Init Aufrufen.

function MCP_CAN_SetRxMask (yMBox: byte; wIDTag, wIDMask: word) : Boolean;
Stellt den Standard ID-TAG und die ID-Mask für eine Rx-Mailbox (1..6) ein.

Beide Parameter zusammen bestimmen ob eine bestimmte Nachricht in dieser Box empfangen werden kann.
Die Bedingung dafür ist:
(RxMsg.EID and wIDMask) = (wIDTag and wIDMask).

Beachten Sie obige Anmerkungen bezüglich der gemeinsam genutzten Maskenregister!

function MCP_CAN_SetRxEMask (yMBox: byte; lwIDTag, lwIDMask: longword) : Boolean;
Stellt den erweiterten ID-TAG und die ID-Mask für eine Rx-Mailbox (1..6) ein.
Beide Parameter zusammen bestimmen ob eine bestimmte Nachricht in dieser Box empfangen werden kann.

Die Bedingung dafür ist:
(RxMsg.EID and lwIDMask) = (lwIDTag and lwIDMask).

function MCP_CAN_SetRxFilter(yMBox : byte; blnWithData : boolean) : boolean;
Bestimmt die standard message receive masks
Diese Funktion stellt die Standard Filter und Mask für das vorgegebene rxfilter ein
Parameter yMBox =Filter Mailbox object (0..5) + Flags in high nibble

Beachten Sie obige Anmerkungen bezüglich der gemeinsam genutzten Maskenregister!

function MCP_CAN_GetError : tMCP_CAN_Errors;
Prüft den aktuellen Fehlerstatus des Controllers (EFLG Register) auf aufgetretene Fehler.

function MCP_CAN_GetStatus) : tMCP_CAN_States;
Liefert den aktuellen Status des Controllers (SPI_READ_STATUS Register).

function MCP_CAN_RxErrCount : byte;
Gibt den Inhalt des REC Registers zurück. Jeder Fehler inkrementiert diesen Wert, jeder fehlerfreie Empfang dekrementiert ihn wieder.

function MCP_CAN_TxErrCount : byte;
Gibt den Inhalt des TEC Registers zurück. Jeder Fehler inkrementiert diesen Wert, jeder fehlerfreie Transmit dekrementiert ihn wieder.

procedure MCP_CAN_StartMessage;
Wurden ein oder mehrere Nachrichten in die Tx-Pipe gestellt mit “PipeSend(CAN_TxPipe, Msg)“ muss diese Funktion aufgerufen werden, um das Senden freizugeben.

3.5 LCD Edit-Felder

by U.Purwin

Allgemeines
Ein sehr oft gestellte Aufgabe auch in uP Systemen sind die Erstellung, Verwaltung und Veränderung von Werten in einer Applikation. Der Anwender hat dazu meistens nur die Möglichkeit dies über kleine alphanumerische LCDs und ein paar wenige Tasten zu tun. Das Hauptproblem dabei ist ausreichende Edit Funktionen zur Verfügung zu stellen und Fehleingaben zu erkennen und zurückzuweisen. Weiterhin dürfen bei bestimmten Werten Ober und Untergrenzen nicht überschritten werden. Diese ganze Verarbeitung ist nicht ganz einfach.

Der hier vorgestellte Editor arbeitet mit fast beliebigen LCDs und Eingabe Möglichkeiten. Die Applikation übergibt dem Treiber einen Wert und dieser stellt ihn entsprechend dar. Wesentlich zur ganzen Funktion trägt dazu der ***KeyBoard Handler*** bei. Dies ist eine Callback Funktion innerhalb des Editors, der die eigentliche Keyboard Lese und Verwaltungs Routine in der Applikation aufruft. Vor dem ersten Aufruf eines Editors muss deshalb der Editor Unit die Adresse des Keyboard Handlers übergeben werden, so dass ein Editor diesen aufrufen kann.

Die Editor Unit "FEdit" enthält alle notwendige Edit Funktionen und reagiert auf Ereignisse die im Keyboard Handler auftreten. Erst wenn der Keyboard Handler den Key “EdKeyExit“ übergibt, wird die aktuelle Edit Funktion abgebrochen und kehrt zur Applikation zurück.

Diese Unit enthält mächtige und umfangreiche Funktionen zum Darstellen und Editieren von Strings, Boolean, Bytes, Words, Integer, Longword, LongInt, Listenfelder, Time, Date und IP-Adressen. Die Editoren sind voll parametrisierbar und gegen Fehleingaben geschützt.

Die hier beschriebenen LCD Edit-Felder funktionieren mit dem Standard LCD-Treiber genauso wie mit dem LCD-Multiport Treiber.

Imports
Die Unit "FEdit" muss importiert werden mit

```
Uses FEdit;
```

3.5.1 die PCU FEdit

Die Unit FEdit ist als vorcompilierte Unit (PCU) im System enthalten. Vom System bereitgestellte Units liegen im Suchpfad des Compilers in der Directory "System". Im Gegensatz zu reinen System Imports die mit der "Import" Klausel importiert werden, z.B. "LCDport", werden diese System Units ganz normal mit "Uses" importiert. Deshalb sind sie auch nur für die Profi Version nutzbar.

Diese Units enthalten zwar alle Funktionen, werden aber zur Compilezeit durch den AVRco Smartlinker bearbeitet. D.h. nur die von der Applikation tatsächlich benutzten Funktion erzeugen auch Maschinen Code. Damit ist sichergestellt, dass das Programm nicht unnötig aufgebläht wird.

Defines und Imports
Die Unit überprüft automatisch den Import der für Sie relevanten Imports und Defines.

Beispiel :
Ist LongInt oder LongWord nicht importiert, stehen die Edit-Funktionen EdLongInt und EdLongWord nicht zur Verfügung.

3.5.1.1 Konstanten

Absolute Konstante
Diese Konstanten sind fest vorgegeben und können nicht verändert werden.

EdEditLength : Byte = 40
Bestimmt die maximale Anzahl der Zeichen, die ein Edit-Feld haben kann.

EdLabelLength : Byte = 20
Bestimmt die maximale Anzahl der Zeichen, die der Label eines Edit-Feldes haben kann.

EdTimeDelim : Char = ':'
Zeichen das bei der Trennung von Zeit-Feldern benutzt wird

EdDateDelim : Char = '.'
Zeichen das bei der Trennung von Datum-Feldern benutzt wird

EdIPDelim : Char = '.'
Zeichen das bei der Trennung von IP-Feldern benutzt wird

Strukturierte Konstanten
Diese Konstanten sind vorgegeben, können aber jederzeit zur Laufzeit geändert/überschrieben werden.

EdPreClearLine : Boolean = True
Ist dieser Wert True, so wird vor der Darstellung des Edit-Feldes die ganze zu diesem Feld gehörende Zeile im LCD-Display gelöscht.

EdBooleanTrue : String = ' AN'
String der Anzeige für Boolean Felder (Wert True)

EdBooleanFalse : String = 'AUS'
String der Anzeige für Boolean Felder (Wert False)

KBRepeatTrigger : Byte = 100; // in SysTicks
Dieser Wert (Konstante x Systick) bestimmt die Zeit nach der eine gedrückte UP-oder DOWN Taste einen AutoRepeat des Wertes auslöst.

KBRepeatDelay : Word = 100; // in SysTicks
Dieser Wert (Konstante x Systick) bestimmt die Geschwindigkeit des AutoRepeat

3.5.1.2 Typen

Type *tEdArrayLocation = (edRam, edEEProm, edFlash);*
Speichertyp der Quelle des Arrays für EDArray

Type *tEdLCDType = (edLCDnone, EdLCDStandard, EdLCDMulti);*
Type des angesprochenen LCD-Displays. Default edLCDnone.

Type *tEdKeys = (EdKeyNone, EdKeyUP, EdKeyDown, EdKeyLeft, EdKeyRight, EdKeyExit);*
Tasten der Editfelder. Wird für die Callback-Funktion tEDKeyBoardHandler benötigt

Type *TEdActEditor = (edNoneEd, edTimeEd, edDateEd, edByteEd, edBooleanEd, edStringEd, edIntegerEd, edIPAddressEd, edLongIntEd, edWordEd, edLongWordEd, edListEd);*
Aufzählungstyp für die Installation des Keyboard und des Error Handlers. Die Handler übergeben hiermit den aktuellen Editor Typ an die Applikation.

Type *tEdErrorEvent = (edLeftLim, edRightLim, edUPLim, edDownLim, edNoKeyHandler, edNoLCDDefined);*
Mögliche Error oder Event Typen, die der optionalen ErrorHandler Callback Funktion übergeben werden.

Type *tEdKeyBoardHandler =* ***Function*** *(ActiveEditor : tEdActEditor; LookKey : tEdKeys) : tEdKeys;*
Template für die Callback-Funktion für das Handling der Keys

Type *tEdErrorEventHandler =* ***Procedure*** *(ActiveEditor : tEdActEditor; ErrorEvent : tEdErrorEvent);*
Template für die optionale Callback-Funktion für das Event und Errorhandling in der Applikation.

```
Type tEdIPAddress  = Record
                       IPOct1,
                       IPOct2,
                       IPOct3,
                       IPOct4 : Byte;
                     end;
```

Record Type zur Übergabe an, und als Ergebnis von der Funktion EdIPAddress. Beinhaltet die vier Adressen-Bytes der IP-Adresse.

3.5.1.3 Prozeduren und Funktionen

Allgemeine Support Funktionen
Diese Funktionen sind recht häufig gebrauchte Konvertierungs Routinen. Sie benutzen weder ein Display noch brauchen sie den Keyboard Handler.
Die Time Strings haben das Format **hh:mm:ss**
Die Date Strings haben das Format **dd.mm.yyyy**

Function *TimeBytesToTimeString (Hour, Minute, Second : Byte) : String[8];*
Konvertiert die Bytes einer RTC in einen formatierten Zeit-String.

Procedure *TimeStringToTimeBytes (TimeString : String[8];* ***var*** *Hour, Minute, Second : Byte);*
Konvertiert einen formatierten Zeit-String in Bytes.

Function *DateBytesToDateString (Day, Month, Year : Byte) : String[10];*
Konvertiert die Bytes einer RTC in einen formatierten Datum-String.

Procedure *DateStringToDateBytes (TimeString : String[10];* ***var*** *Hour, Minute, Second : Byte);*
Konvertiert einen formatierten Datum-String in Bytes.

Support Funktionen für die Editoren
Diese Funktionen bestimmen das Verhalten der Editoren. Der Display Typ und beim Multi-LCD die Display Nummer können zur Laufzeit beliebig geändert werden.

Procedure *EdSetLCDType (LCDType : tEdLCDType);*
Setzt den LCD-Typ den die Editoren benutzen müssen. edLCDStandard oder edLCDMulti.

Procedure *EdSetMultiLCDNum (Num : TLCD_num);*
Setzt die LCD_NUM bei MultiLCDs

Procedure *EdSetKeyBoardHandler (ActiveEditor : tEdActEditor; KeyHandler : tEdKeyBoardHandler);*
Bestimmt den Keyboard-Handler der Edit-Felder. Es können mehrere Keyboard Handler vorhanden sein und diese können zur Laufzeit auch beliebig ausgetauscht werden. Die Handler müssen eine vorgegebene Struktur haben.

Keyboard Handler:

```
Function MyKeyHandler (ActiveEditor : tEdActEditor; ReturnKey : tEDKeys) : tEDKeys;
begin
  case ActiveEditor of
  EdTimeEd : ...
        |
  EdDateEd : ...
        |
 endcase;
```

```
  iF INP_STABLE_G (LEFT) then
    Return(EdKeyLeft);
  endif;

  iF INP_STABLE_G (RIGHT) then
    return(EdKeyRIGHT);
  endif;
...
end;

// Hauptprogramm
begin
  ...
  EdSetKeyBoardHandler (@MyKeyHandler);
  ...
end.
```

Procedure *EdSetErrorEventHandler (ActiveEditor : tEdActEditor; ErrorEventHandler : tEdErrorEventHandler);*

Bestimmt den optionalen Error und Eventhandler. Events die auftreten können sind das überschreiten der Begrenzungen für die vier Keys UP, DOWN, LEFT und RIGHT. Fehler die auftreten können, sind: kein Keyboard Handler spezifiziert oder kein LCDtyp spezifiziert. Der Handler muss eine vorgegebene Struktur haben.

Error Handler:

```
Procedure MyErrorEventHandler (ActiveEditor : tEdActEditor; ErrorCode:  tEdErrorEvent);

begin
  case ActiveEditor of
   EdTimeEd : ...
          |
   EdDateEd : ...
          |
  endcase;

  case ErrorCode of
      EdLeftLim:              //Left Limit
            |
      EdRightLim:             //Right Limit
            |
      EdUPLim:                //Up Limit
            |
      EDDownLim:              //Down Limit
            |
      EDNoKeyHandler:         //No KeyHandler found
            |
      EDNoLCDDefined:         //No LCD-Display Defined
            |
  endcase;
end;

// Hauptprogramm
begin
  ...
  EdSetErrorEventHandler (@MyErrorEventHandler);
  ...
end.
```

3.5.1.4 Die Editoren

Fast alle Edit-Funktionen besitzen Standardparameter die nun beschrieben werden. Zusätzliche Parameter können der einzelnen Funktion entnommen werden. Weiter sind alle Funktionen in dem Demoprogramm **..\E-LAB\AVRco\Demos\LCD_Edit\LCDEdit** enthalten.

Parameter	Beschreibung
EdValue	Übergabe-Wert an die Funktion
LeadLabel	Optionales führendes Label des Edit-Feldes
Postlabel	Optionales Label nach dem Edit Feld z.B. *LeadLabel*-> Masse: 12.54 kg <- *PostLabel* ^- *EdValue*
X, Y	Koordinaten auf dem LCD-Display
BlinkCursor	Blinkender Block-Cursor auf dem LCD-Display an der Stelle Edit
VMin,VMax	Minimaler-Maximaler Edit-Wert
Repeater	AutoRepeater ein/aus. Für String und List editor
Decimal	Dezimalstellen des Wertes

Function *EdTime (EdValue : String[EdTimeLength]; LeadLabel : String[EdLabelLength];*
X, Y : Byte; BlinkCursor : Boolean;
EditSeconds : Boolean) : String[EdTimeLength];
EditSeconds bestimmt, ob im Edit-Feld auch die Sekunden editiert werden können

Function *EdDate (EdValue : String[EdDateLength]; LeadLabel : String[EdLabelLength];*
X, Y : Byte; BlinkCursor : Boolean) : String[EdDateLength];

Function *EdByte (EdValue : Byte; LeadLabel, PostLabel : String[EdLabelLength];*
X, Y : Byte; BlinkCursor : Boolean; VMin, VMax : Byte) : Byte;

Function *EdBoolean (EdValue : Boolean; LeadLabel, PostLabel : String[EdLabelLength]; X, Y : Byte;*
BlinkCursor : Boolean) : Boolean;

Function *EdString (EdValue : String[EdStringLength]; LeadLabel, PostLabel String[EdLabelLength];*
X, Y : Byte; BlinkCursor : Boolean; MaxLen : Byte; MinChar, MaxChar : Byte;
Repeater : Boolean) : String[EdStringLength];
MaxLen bestimmt die maximale Länge des zu editierenden Strings

Function *EdInteger (EdValue : Integer; LeadLabel, PostLabel : String[EdLabelLength];*
X, Y : Byte; BlinkCursor : Boolean; VMin, VMax : Integer; Decimal : Byte) : Integer;

Function *EdLongInt (EdValue : LongInt; LeadLabel, PostLabel :String[EdLabelLength]; X, Y : Byte;*
BlinkCursor : Boolean; VMin, VMax : LongInt; Decimal : Byte) : LongInt;

Function *EdWord (EdValue : Word; LeadLabel, PostLabel : String[EdLabelLength]; X, Y : Byte;*
BlinkCursor : Boolean; VMin, VMax : Word; Decimal : Byte) : Word;

Function *EdLongWord (EdValue : LongWord; LeadLabel, PostLabel : String[EdLabelLength];*
X, Y : Byte; BlinkCursor : Boolean; VMin, VMax : LongWord;
Decimal : Byte) : LongWord;

Function *EdList (EdValue : Pointer; Location : tEdArrayLocation;*
LeadLabel, PostLabel : String[EdLabelLength]; X, Y : Byte; BlinkCursor : Boolean;
StrLen, Count, Default : Byte) : Byte;

EdValue	ist hier ein Pointer auf ein beliebiges Array of string (siehe Demoprogramm)
Location	bestimmt den auf den Speichertyp (RAM, ROM, EEPROM) des Array's
StrLen	ist die maximale Länge eines Strings im Array (Achtung auf die Länge achten! Hier wird mit Pointern gearbeitet)
Count	ist die Anzahl der Einträge im Array (Achtung auf die Anzahl achten! Hier wird mit Pointern gearbeitet.)

Dieser Editor dient zur Anzeige von String Listen und gestattet dem User darin auf- und abwärts zu blättern und einen Eintrag auszuwählen. Bitte beachten, dass das Array mit dem Index 0 beginnen muss.

Function *EdIPAddress (EdValue : tEDIPAddress; LeadLabel, PostLabel : String[EdLabelLength];*
X, Y : Byte; BlinkCursor : Boolean;
IPMin, IPMax : tEdIpAddress) : tEDIPAddress;

Dient zum (nicht trivialen) Editieren von IP-Adressen im Ethernet Bereich.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\LCD_Edit**

Achtung bei *Demo* und *Standard* Version von AVRco

Demo Programm nicht neu compilieren, sondern einfach in den Simulator laden zum Testen.

3.6 LCD Graphics

Allgemeines
Die Kommunikation zwischen Mensch und Maschine wird immer komplexer aber auch komfortabler. War vor Jahren noch ein Taster, Schalter und ein paar LEDs ausreichend, so werden heute schon regelmässig mehrzeilige LCD Displays eingesetzt.

Durch die Leistungsfähigkeit moderner Prozessoren, ihren relativ grossen embedded Speicher und die Verfügbarkeit intelligenter und preiswerter Graphic LCDs ist der Trend zur graphischen Interaktion nicht mehr aufzuhalten. Handys, Palmtops etc. machen es vor.

Für den Endkunden, den Auftraggeber und auch für den Entwickler ist ein kleines „Windows“ eine verlockende Sache. Allerdings muss man im Auge behalten, dass hier kein Pentium Prozessor und keine Hochleistungs Graphic Karte zur Verfügung stehen.

Die vorliegende Implementation beschränkt sich deshalb auf LCDs mit einer Auflösung bis zu 1024x1024 Pixels. Das ist jedoch nur ein theoretischer Wert. In der Praxis muss irgendwann einmal, spätestens beim Bildschirm löschen, jeder Pixel beschrieben werden und das geht enorm in die Rechenzeit. Die benötigte Rechenzeit erhöht sich quadratisch mit der Auflösung. 128x128 = 16384 Pixels. 1024x1024 = 1048576 Pixels.
Gut handhabbare Display Grössen sind 128x64, 128x128, 240x128, 320x240 und evtl. noch 640x480.

Bei der Auswahl von Displays kommen 2 Typen in Frage: intelligente mit eingebautem Controller (z.B. Toshiba T6963C) und „dumme“ ohne Controller. Bei kleinen Stückzahlen sind Typen mit Controllern vorzuziehen, da der Aufwand eines eigenen Controllers zu hoch ist. Bei einer Auflage von 100Stk und mehr lohnt es sich evtl. den Controller (z.B. Seiko/Epson SED1xxxx) mit auf das Board zu bringen. Bei der Auswahl des Controllers ist darauf zu achten, dass je leistungsfähiger der Controller ist, desto aufwendiger ist seine Anpassung an die Software. PC Graphic Chips sind deshalb ungeeignet. Der Controller muss nur den Byte-weisen Zugriff auf das Display Ram unterstützen. Eine Controller-interne Umrechnung von x/y-Koordinaten nach linearer Adresse ist nicht notwendig.
Aber bei den Farb (TFT) LCDs ist der xy-address mode zwingend not wendig. Mit den Farb LCDs haben sich auch einige der Treiber Funktionen geändert bzw. wurden erweitert.

3.6.1 Eigenschaften des Graphic Systems

Da Sie ja offensichtlich mit Windows arbeiten, wissen Sie zumindest ungefähr was ein Fenster/Window ist ☺ Die AVRco Implementation verwendet auch Fenster. Da diese sich allerdings von der Komplexität und Leistungsfähigkeit her erheblich von den Fenstern von Windows® unterscheiden, wird hier nicht der Begriff Window verwendet, sondern **ViewPort**.

Ein ViewPort hat keinen sichtbaren Rahmen, lässt sich nicht ohne weiteres vergrössern, verkleinern oder verschieben. Es gibt keine Fenster Hirarchie, d.h. sich teilweise oder ganz überdeckende Fenster sind nicht gegen einander geschützt sondern sie überschreiben sich gegenseitig. Diese Nachteile sind gravierend, wenn man an Windows® denkt. Die Implementation dieser Funktionen würde jedoch einen mega103 um ein vielfaches sprengen. Eine moderne Windows® Implementation füllt eine CD-ROM☺ . Soviel zu den Nachteilen der hier implementierten ViewPorts.

Aber es gibt glücklicherweise auch ein paar Funktionalitäten.

ViewPort physisch
Ein ViewPort bezeichnet einen Teil eines Displays. Die Lage und Grösse dieses ViewPorts wird durch die Funktion *OpenView* bestimmt.

ViewPort logisch
Dem physischen ViewPort kann mit der Funktion *ScaleView* ein eigenes logisches Koordinaten System zugewiesen werden. Alle Zeichen und String Operationen benutzen dieses Koordinaten System für ihre Koordinaten bzw. Positionierungen. Die Dimensionierung erfolgt mit 16bit Integer.
Eine Skalierung des ViewPorts 0 ist nicht möglich.

ViewPort Attribute
Jedes ViewPort hat einen eigenen Satz Attribute für Strings und allgemeine Zeichen Operationen. Diese Attribute bestimmen wie z.B. eine Linie gezeichnet wird, mit XOR, OR oder NOR.

ViewPort Clipping
Sämtliche Schreib Operationen in ein ViewPort werden darauf geprüft, ob sie die Grenzen überschreiten. Ist das der Fall wird die Schreib Operation an dieser Stelle unterbrochen, bis evtl. wieder innerhalb des Ports weiter geschrieben werden kann (z.B. Kreise).

ViewPort Select
Sind mehrere ViewPorts definiert, kann zwischen den Ports mit *SwitchView* umgeschaltet werden. Alle nachfolgenden Operationen beziehen sich nun auf dieses Port mit dessen Attributen und Skalierungen.

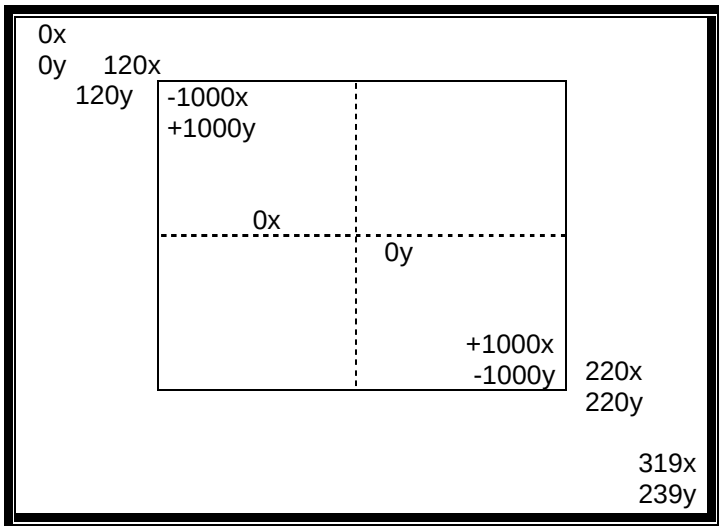
ViewPort Save/Restore
Will man es ganz komfortabel machen, und hat man ausreichend Speicher (und auch Rechenleistung) kann man ein ViewPort, bevor es von einem anderen, überlappenden, zerstört wird, in den Speicher retten. Bei Bedarf kann der gerettete Inhalt wieder mit *RestoreView* in das ViewPort zurückgeschrieben werden. Das ist allerdings nur bei kleinen Displays oder kleinen ViewPorts empfehlenswert (Speicherbedarf und Rechenzeit).

ViewPort Visuality
Will man die Umrisse eines ViewPorts auch optisch sichtbar machen, so kann man dazu die Funktion *FrameView* benutzen. Dieser Rahmen ist innerhalb des ViewPorts schreibgeschützt.

ViewPort Definition
Zur Verdeutlichung der Zusammenhänge von physischen und logischen Koordinaten eines ViewPorts hier eine kleine Graphik:

Beispiel LCD 320x240:
GOpenView (1, 120, 120, 220, 220)
eröffnet ein quadratisches ViewPort 1 mit der Grösse 100x100 Pixels. Der Ursprung des ViewPorts ist an der Stelle 120, 120 des Displays.

GScaleView (1, -1000, 1000, 1000, -1000)
skaliert das ViewPort 1 und bestimmt die interne Skalierung des gewählten ViewPorts.Dabei bezeichnen Xs, Ys die obere linke Ecke und Xe, Ye untere rechte Ecke des logischen ViewPorts.



3.6.2 Treiber Implementation

Das System AVRco stellt eine umfangreiche Sammlung von high-level Graphic Funktionen zur Verfügung. Diese Funktionen werden intern auf einfache Byte-read-write Funktionen reduziert. Diese Zugriffe auf das Display Refresh Ram muss der Anwender selbst programmieren. Dazu stellt das System eine sogenannte UserDevice Funktion mit dem Namen GraphIOS zur Verfügung. Diese Routine muss der Programmierer selbst implementieren und die hiermit übergebenen Adressen und Parameter in geeigneter Weise an den Display Controller weiterleiten.

3.6.2.1 Controller mit linearer Adressierung T6963

Das System übergibt 4 Parameter (5 wenn Farbe importiert wurde) an die Funktion GraphIOS und erwartet entweder ein Byte oder ein Boolean als Rückgabewert. Der erste Parameter (Byte) bezeichnet die auszuführende Aktion, der zweite (word) bezeichnet die Display Adresse eines Bytes, wobei das Display als ein linearer Adressraum bestehend aus Bytes aufgefasst wird. Die zwei letzten Parameter sind alternativ zu verwenden, wobei im allgemeinen der Parameter „Mask“ benutzt wird. Mit diesem wird ein einzelnes Bit (Pixel) innerhalb des adressierten Bytes manipuliert. Erlaubt und unterstützt der verwendete Controller selbst Bit- bzw. Pixel Manipulationen, kann dafür der Parameter „pixel“ benutzt werden.

Bemerkungen:

1. Durch die Verwendung eines words als Adressparameter wird die aktuelle Auflösung des Displays auf 65536 Bytes = 524288 Pixels beschränkt, was einer Dimension von max 724x724 Pixels entspricht.

2. Manche Controller bzw. Intelligente Displays lassen eine unterschiedlich Adressierung bzw. Byte Auflösung des Displays zu. Es wird hier unterschieden 5 und 8 bits pro Zugriff. Vermeiden Sie einen Modus der nicht volle 8Bits pro Zugriff/Byte adressiert.
   Die Rechenzeit zur Feststellung der Adresse eines gewünschten Bytes und der zugehörigen Maske wird ansonsten drastisch erhöht

Wenn Farbe importiert wurde, dann wird dem UserDevice ein weiteres Byte übergeben. Dieses Byte muss dann durch die Applikation interpretiert werden

```
UserDevice GraphIOS (cmd : byte; adr : word; mask, pixel, color : byte) : byte;

UserDevice GraphIOS (cmd : byte; adr : word; mask, pixel : byte) : byte;
begin
// commands passed to user defined function "GraphIOS"
// 0 display init     adr = none       mask = none    pixel = none   [color = none]   result = none
// 1 display clear    adr = fillpatt   mask = none    pixel = none   [color = none]   result = bool
// 2 write byte       adr = byte adr   mask = byte    pixel = mode   [color = byte]   result = none
// 3 read byte        adr = byte adr   mask = none    pixel = none   [color = none]   result = byte
// 4 set pixel        adr = byte adr   mask = mask    pixel = pixel  [color = byte]   result = none
// 5 clear pixel      adr = byte adr   mask = mask    pixel = pixel  [color = byte]   result = none
// 6 xor pixel        adr = byte adr   mask = mask    pixel = pixel  [color = byte]   result = none

  case cmd of
    0  : cmd:= 0; |          // display init
    1  : cmd:= 1; |          // display clear
    2  : cmd:= 2; |          // write byte with attributes
    3  : cmd:= 3; |          // read byte
    4  : cmd:= 4; |          // set pixel
    5  : cmd:= 5; |          // clear pixel
    6  : cmd:= 6; |          // xor pixel
  endcase;
  return(cmd);
end;
```

CMD 0 Display Init
Dient zum initialisieren evtl. vorhandener Hardware, wie z.B. Ports und des Display Controllers.
Der Rückgabe Wert ist ohne Bedeutung.

CMD 1 Display Clear
Löscht das Display. Wenn der Controller eine eingebaute Löschfunktion hat, sollte diese aufgerufen werden. Der Rückgabe Wert ist in diesem Fall „true".
Hat der Controller diese Funktion nicht, wird über den Rückgabe Wert „false" eine interne Löschschleife aufgerufen, was natürlich wesentlich länger dauert. Man kann hier auch selbst eine evtl. schnellere eigene Lösch-Routine in Assembler schreiben und dem System mit „true" mitteilen, dass der Löschvorgang erfolgreich abgeschlossen wurde

CMD 2 Write Byte
Schreibt das in Parameter „Mask" übergebene Byte in das Display Ram an die Stelle „Adr". Im Parameter „Pixel" wird dabei das Schreib-Attribut übergeben. Eine „0" bedeutet dass das Byte „Mask" vor dem Schreiben invertiert wird. Bei einer „1" wird „Mask" ohne Veränderung in den Controller geschrieben. Bei einer „2" muss zuerst das Byte an der Stelle „Adr" aus dem Controller gelesen und dann mit „Mask" ein XOR ausgeführt werden. Das Ergebnis wird in den Controller an die Stelle „Adr" zurückgeschrieben.

CMD 3 Read Byte
Liest vom Display Ram das in Parameter „Adr" bezeichnete Byte und gibt es als Ergebnis zurück.

CMD 4 Set Pixel
Liest das vom Parameter „Adr" adressierte Byte vom Display Ram, setzt das durch „Mask" bezeichnete Bit und schreibt das so geänderte Byte zurück ins Display Ram. (Read-Modify-Write)
Unterstützt der Controller Pixel-Manipulation, so kann auf das Lesen verzichtet werden, indem dem Controller in geeigneter Weise die Byte-Adresse und die Pixel-Adresse (Parameter „Pixel") übergeben wird.

CMD 5 Clear Pixel
Liest das vom Parameter „Adr" adressierte Byte vom Display Ram, setzt das durch „Mask" bezeichnete Bit zurück und schreibt das so geänderte Byte zurück ins Display Ram. (Read-Modify-Write)
Unterstützt der Controller Pixel-Manipulation, so kann auf das Lesen verzichtet werden, indem dem Controller in geeigneter Weise die Byte-Adresse und die Pixel-Adresse (Parameter „Pixel") übergeben wird.

CMD 6 Xor Pixel
Liest das vom Parameter „Adr" adressierte Byte vom Display Ram, invertiert das durch „Mask" bezeichnete Bit zurück und schreibt das so geänderte Byte zurück ins Display Ram. (Read-Modify-Write)
Unterstützt der Controller Pixel-Manipulation, so kann auf das Lesen verzichtet werden, indem dem Controller in geeigneter Weise die Byte-Adresse und die Pixel-Adresse (Parameter „Pixel") übergeben wird.

Programm Beispiel

ein Beispiel befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\GraphLCD**

3.6.2.2 Controller mit Spalten Adressierung (HD61202, SED1531 etc)

Das System übergibt 2 Parameter an die Funktion GraphIOS. Ein Resultat wird nicht erwartet. Der erste Parameter (Byte) bezeichnet die auszuführende Aktion, der zweite (Byte) bezeichnet entweder eine Display Zeilen Adresse oder das zu schreibende Byte. Der Treiber schreibt grundsätzlich immer in sein internes RAM. Ein Display Update wird nur auf Befehl gemacht mit „gDispRefresh" wobei immer der komplette Display Inhalt als ganzes neu ausgegeben wird.

Bemerkung:
1. Durch die Verwendung eines schnellen Konvertierungs Algorithmus dauert ein Display update bei 16MHz ca. 25msec im seriellen Mode und 17msec im parallel Mode.
2. Dadurch wird die horizontale und vertikale Auflösung des Displays auf ein vielfaches von 8 beschränkt 32x64 oder 64x128. Zwischenwerte sind allerdings möglich.

Imports
Dieser Modus wird importiert mit

```
Define  LCDgraphMode = column, iData;   {column mode controller}

UserDevice GraphIOS (cmd : byte; data : byte);
begin
// commands passed to user defined function "GraphIOS"
// cmd 0  set row address   data = row
// cmd 1  write data byte   data = content
  if cmd = 0 then
    // set row addr = data
  else
    // write data byte
  endif;
end;
```

CMD 0 **Set row address**
Dem Controller wird die aktuelle Row Adresse übergeben. Der Controller inkrementiert diesen Wert nach jedem Data Schreibzugriff, so dass pro Zeile nur einmal das CMD 0 vorkommt.

CMD 1 **Write display data**
Das Userprogramm schreibt das übergebene Byte „data“ in das Datenregister des Controllers.

Programm Beispiele:

ein Beispiel für eine **SED1531** Implementation des UserDevice GraphIOS befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\LCD_1531**

ein Beispiel für eine **HD61202** Implementation des UserDevice GraphIOS befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\LCD_61202**

3.6.2.3 Controller mit Read-Only Linear Adressierung (PCF8548 etc)

Dieser Treiber entspricht weitgehend dem mit Spalten Adressierung, abgesehen davon dass hier eine Lineare Adressierung durch den Controller möglich ist. Diese Controller unterstützen dann jedoch auch nicht das Auslesen des Graphic Speichers, so dass immer nur der ganze Bild Speicher geschrieben werden kann.

Das System übergibt 2 Parameter an die Funktion GraphIOS. Ein Resultat wird nicht erwartet. Der erste Parameter (Byte) bezeichnet die auszuführende Aktion, der zweite (Byte) bezeichnet entweder eine Display Zeilen Adresse oder das zu schreibende Byte. Der Treiber schreibt grundsätzlich immer in sein internes RAM. Ein Display Update wird nur auf Befehl gemacht mit „gDispRefresh“ wobei immer der komplette Display Inhalt als ganzes neu ausgegeben wird.

Bemerkungen:
1. Durch die Verwendung eines schnellen Konvertierungs Algorithmus dauert ein Display update bei 16MHz ca. 25msec im seriellen Mode und 17msec im parallel Mode.
2. Dadurch wird die horizontale und vertikale Auflösung des Displays auf ein vielfaches von 8 beschränkt 32x64 oder 64x128. Zwischenwerte sind allerdings möglich.

Imports
Dieser Modus wird importiert mit

```
Define LCDgraphMode = readonly, iData;   {linear, readonly controller}

UserDevice GraphIOS (cmd : byte; data : byte);
begin
// commands passed to user defined function "GraphIOS"
// cmd 0  set row address   data = row
// cmd 1  write data byte   data = content
  if cmd = 0 then
    // set row addr = data
  else
    // write data byte
  endif;
end;
```

CMD 0 **Set row address**
Dem Controller wird die aktuelle Row Adresse übergeben. Der Controller inkrementiert diesen Wert nach jedem Data Schreibzugriff, so dass pro Zeile nur einmal das CMD 0 vorkommt.

CMD 1 **Write display data**
Das Userprogramm schreibt das übergebene Byte „data“ in das Datenregister des Controllers.

3.6.2.4 Farb/TFT Controller

Alle low-cost Farb TFT LCDs besitzen interne Controller die die xy-Adressierung unterstützen. Das bedeutet dass z.B. Pixel mit ihrer echten xy-Adresse angesprochen werden können. Ebenso horizontale und vertikale Linien. Auch ein Flächen füllen wird unterstützt. Das vereinfacht den eigentlichen Treiber sehr. Allerdings ist oft ein Rücklesen des Grafik Speichers nicht möglich und eine XOR Operation macht bei Farbe keinen Sinn.

Hier sind die meisten Funktionen des Treibers auf Farbe erweitert und die Funktionen laufen oft etwas anderst ab. Kleinere Displays werden sinnvollerweise mit einem SPI Interface betrieben wobei 16MHz SPI Clock kein Problem ist und die Operationen deshalb ausreichend schnell sind.

Dieser Modus wird importiert mit

```
From LCDGraphic import GraphColor;

Define LCDgraphMode = XYaddress, iData;   // colored xy-address mode
```

Dieser xy-address Mode exportiert dann diese Enumeration:

```
// gDrawType = (dtPixel, dtLineX, dtLineY, dtFillRect, dtClear);

UserDevice GraphIOS(xs, ys, xe, ye : word; pattern, colorFg, colorBg : byte;
draw : gDrawType) : byte;
begin
  case draw of
    dtClear    : // FillScreen
               |
    dtFillRect : // FillRect
               |
    dtLineX    : // draw line horizontal
               |
    dtLineY    : // draw vertical line
               |
    dtPixel    : // draw one pixel
               |
  endcase;
  return(0);
end;
```

3.6.3 Import des Graphic Systems

Wie beim AVRco System üblich, müssen Devices importiert und definiert werden.

```
Import SysTick, LCDGraphic, ...;

// nur bei Strings notwendig ->
From LCDGraphic Import CharSet;          {block CharSet, pixels}
From LCDGraphic Import GraphColor;       {simple color support, only in linear mode}
```

Defines
Durch die folgende Define Anweisungen wird das LCD-Graphic-Systems spezifiziert:

```
Define
     LCDGraphic      = 240, 128, 8;      {x-pix, y-pix, accesswidth}
     GViewports      = 4, iData;         {logical ViewPorts, scalings}
     LCDgraphMode    = linear, iData;    {optional, linear is default}
oder
     LCDgraphMode    = column, iData;    {column oriented controller}
oder
     LCDgraphMode    = readonly, iData;  {linear, readonly controller}
oder
     LCDgraphMode    = xyAddress, iData; {col or xy-address controller}
oder
     DefCharSet      = 'Graphchars.pchr'; {FileName, stored into Flash}
oder
     DefCharSet      = RAM;              {charset is stored into RAM}
     TGraphStr       = 20;               {Graphic Text String Length, max 24}
```

Diese 2 Defines sind „Muss“ Angaben.

LCDgraphic bestimmt die Display-Grösse (x, y) und die Zugriffsbreite (normalerweise 8).

GViewPorts bestimmt die Anzahl der logischen ViewPorts (Fenster) und deren Lage im Adressraum.

Wenn **CharSet** importiert wurde, müssen auch folgende beide Defines vorhanden sein:

DefCharSet
bestimmt den default Zeichensatz, der für String Ausgabe in allen ViewPorts benutzt wird.
Wird ein Filename in ’’ angegeben, werden alle Zeichensätze im ROM/Flash erwartet. Wird statt dessen der Parameter „RAM“ angegeben, werden alles Zeichensätze im RAM erwartet.

TgraphStr
bestimmt den Typ bzw. max. Länge des für Textausgabe zu verwendenden Standard Strings.

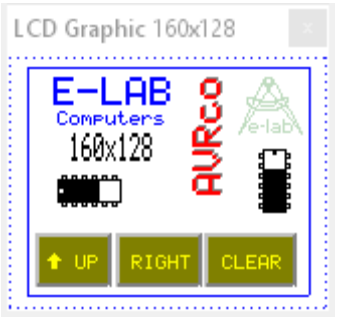
LCDgraphMode ist optional und bestimmt den Controller Typ. Wenn das Define fehlt oder auf linear gesetzt ist werden Controller vom Typ T6963, SED1520 oder ähnlich unterstützt.

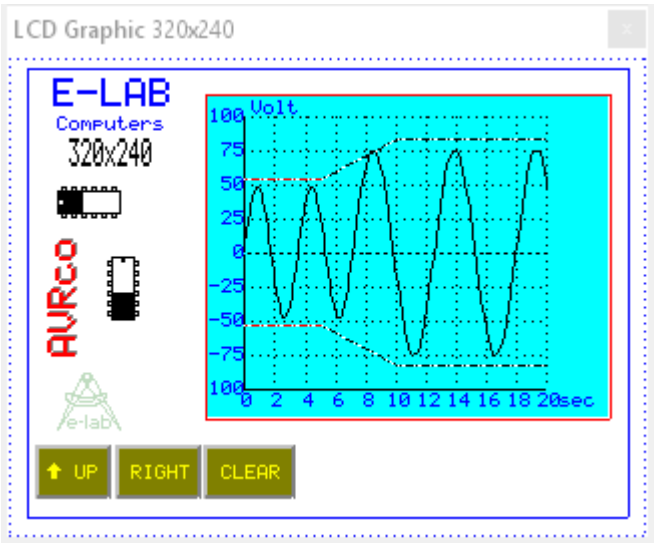
Ist column Mode gesetzt, werden Controller vom Typ SED1531, KS0107 oder kompatible unterstützt.
Ist der readonly Mode gesetzt, werden linear adressierende Controller wie der PCF8548 unterstützt.
Im Column und ReadOnly Mode wird der komplette Display Inhalt immer im RAM gehalten und manipuliert. Eine Ausgabe an das Display erfolgt nur mit der Treiber Funktion „gDispRefresh“. Der Speicherbedarf errechnet sich aus der Displaygrösse. 128x64pix benötigen 1kByte internes RAM. Mit „iData“ oder „xData“ wird die Lage des Refresh Speichers bestimmt.

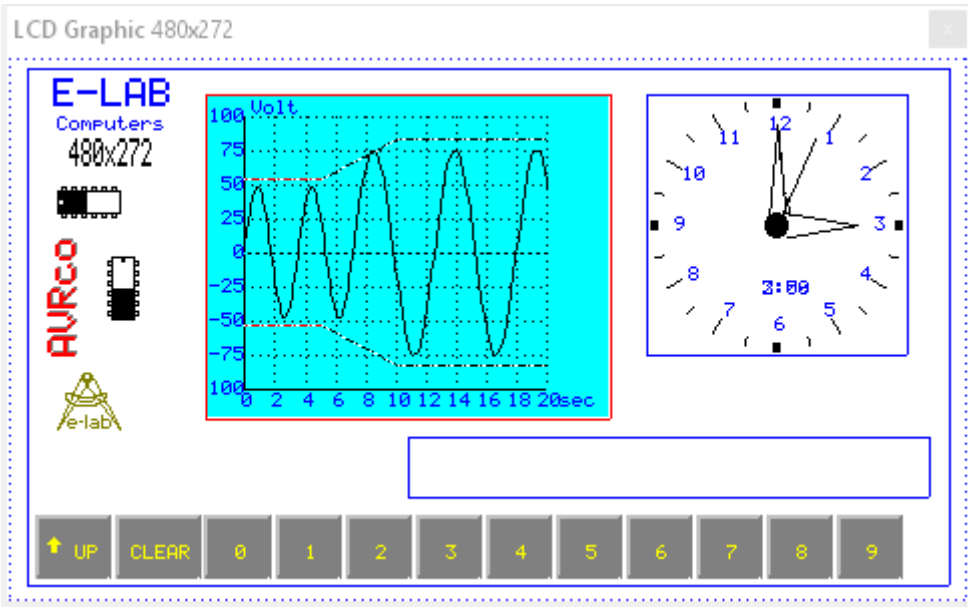
Ist der xyAddress Mode gesetzt, dann werden TFT Color Controller unterstützt.
Programm Beispiel für XY-Adress Mode:

ein Beispiel für eine TFT Color Implementation des UserDevice GraphIOS befindet sich im Verzeichnis
..\E-LAB\AVRco\Demos\XGraph_TFT160x128

und eines im Verzeichnis
..\E-LAB\AVRco\Demos\XGraph_TFT320x240

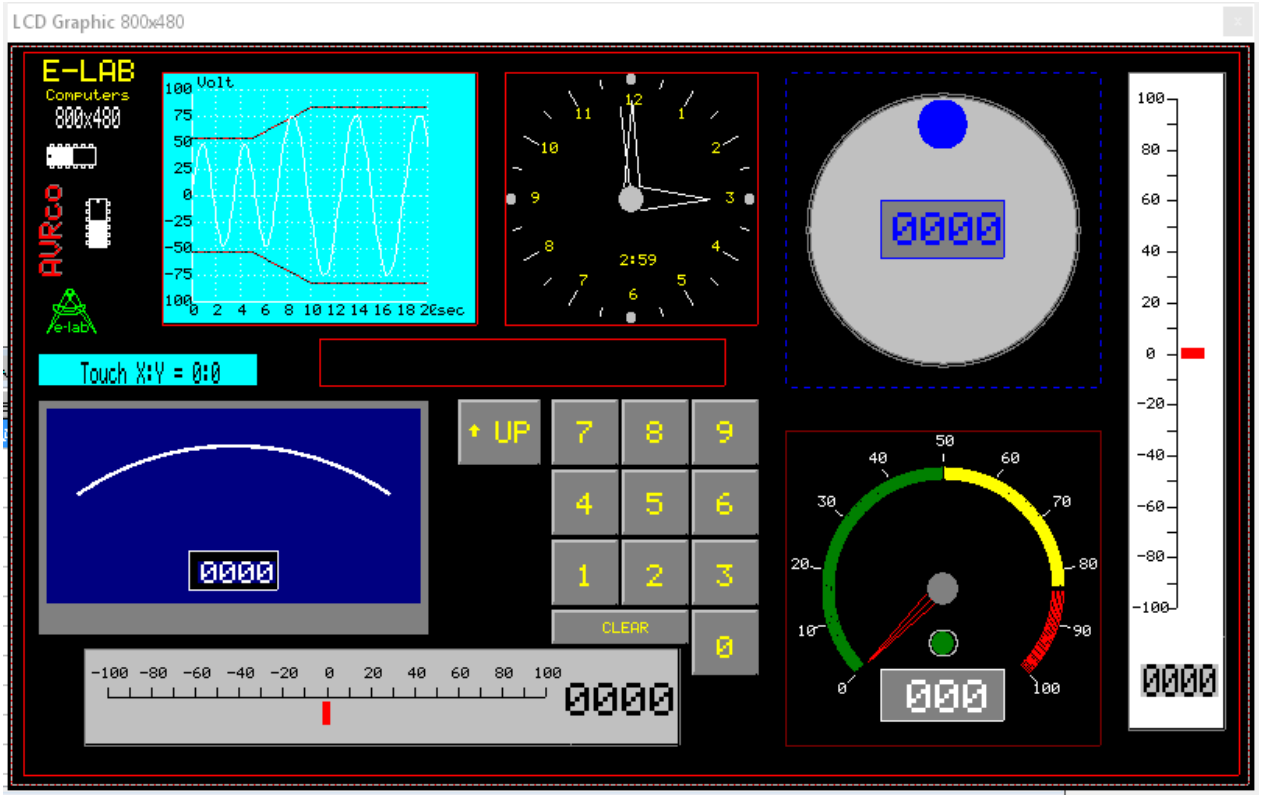






und eines im Verzeichnis
..\AVRco\Demos\XGraph_KeyBoard

und eines im Verzeichnis
..\E-LAB\AVRco\Demos\XGraph_TFT800x480



3.6.4 Typen, Funktionen und Prozeduren

Typen
Folgende Typen werden durch den Import des Graphic-Systems automatisch mit importiert:

Type *TGraphString = String[n]; {max. 24 Zeichen}*
Der Typ tGraphString wird bei den Prozeduren gDrawString und gDrawStringRel benötigt. Aus bestimmten Gründen (Geschwindigkeit, Speicherplatz etc.) muss mit diesem Stringtyp gearbeitet werden. Aus den gleichen Gründen darf die Definition 24 nicht überschreiten.

Type *TWriteMode = (wmClrPix, wmSetPix, wmXorPix);*
Ist ein Aufzählungstyp (Enumeration) für alle Zeichen Operationen incl. Strings. Wenn Texte, Linien etc. auch wieder gelöscht werden sollen, so ist immer mit dem Attribut **wmXorPix** zu arbeiten. Ein Text der mit diesem Attribut geschrieben wurde, kann mit exakt der gleichen Operation wieder gelöscht werden, ohne dass irgend etwas davon sichtbar bleibt. (nondestructive read/write)

Type *TtxtAlHor = (alHorLeft, alHorCenter, alHorRight);*
Aufzählungstyp (Enumeration). Text Ausrichtungs Attribut. AlHorRight z.B. lässt den Text an der vorgegebenen Koordinate enden.

Type *TTxtAlVert = (alVertBottom, alVertCenter, alVertTop);*
Aufzählungstyp (Enumeration). Text Ausrichtungs Attribut. AlVertTop z.B. setzt den Text unter die vorgegebene y-Koordinate.

Type *TTxtRotate = (TxtRot0, TxtRot90, TxtRot180, TxtRot270);*
Aufzählungstyp (Enumeration). Bestimmt die grundsätzliche Text Ausrichtung bzw. Rotation.

Type *TTextBkGnd = (bkNormal, bkTransp, bkInvers);*
Aufzählungstyp (Enumeration). Bestimmt den Hintergrund des Texts

Nur bei XY-address:
Type *gDrawType = (dtPixel, dtLineX, dtLineY, dtFillRect, dtClear);*
Aufzählungstyp (Enumeration). Bestimmt den Command Typ für das UserIOS im xy-address mode.

Allgemeine Funktionen

```
gSetCharSetRAM (RAM : boolean);
gSetBitMapRAM (RAM : boolean);
gClrScr (pattern : byte);   // standard mode
gClrScr (color : byte);     // xy-address mode
gScaleToPnt (ViewPort: byte; XL, YL : integer; VAR Xp, Yp : integer);
gPntToScale (ViewPort: byte; Xp, Yp : integer; VAR XL, YL : integer);
Function(gPointInRect(x, y, xs, ys, xe, ye : word) : boolean;
gGetFontWidthScaled : integer;
gGetFontHeightScaled : integer;
```

Wenn ***GraphColor*** importiert wurde:

```
gSetLineColor (c : byte);
gGetLineColor: byte;
gSetTextColor (c : byte);
gGetTextColor: byte;
```

gSetCharSetRam (RAM: boolean);
Schaltet zur Laufzeit zwischen Zeichensätze im ROM und im RAM um. Gleichzeitig muss aber auch mit „gSetCharSet“ ein entsprechender Zeichensatz aus dem ROM oder RAM ausgewählt werden. Der Default Wert nach einem Neustart ist Zeichensatz im ROM.

gSetBitMapRAM (RAM: boolean);
Schaltet zur Laufzeit zwischen BitMaps im ROM und im RAM um. Die Routine „gDrawBitMap“ wird zeichnet dann ein entsprechendes BitMap aus dem ROM oder RAM.

gClrScr (pattern : byte); // standard modes
Löscht das komplette Display mit dem Byte „pattern“. Dieses Attribute ist im Normalfall $00 oder $FF

gClrScr (color : byte); // xy-address mode
Löscht das komplette Display mit der Farbe „color“.

gScaleToPnt (ViewPort: byte; XL, YL : integer; VAR Xp, Yp : integer);
Wandelt eine ViewPort relative Koordinate in eine Display absolute Koordinate.

gPntToScale (ViewPort: byte; Xp, Yp : integer; VAR XL, YL : integer);
Wandelt eine Display absolute Koordinate in eine ViewPort relative Koordinate.

Function(gPointInRect(x, y, xs, ys, xe, ye : word) : boolean;
Prüft ob eine Koordinate (Punkt) innerhalb eines Rechtecks ist.

gGetFontWidthScaled : integer;
Gibt die Breite eines Characters in logischen Pixeln zurück. Das aktuelle ViewPort wird zur Berechnung herangezogen.

gGetFontHeightScaled : integer;
Gibt die Höhe eines Characters in logischen Pixeln zurück. Das aktuelle ViewPort wird zur Berechnung herangezogen.

Im **Linear Mode** kann auch eine einfache Farb Unterstützung für Texte und Linien importiert werden. Der Treiber GraphIOS gibt dazu dann die zugehörige Farb Einstellung abhängig vom ViewPort und der Zeichnungs Operation als Byte mit aus. Es ist dann aber Aufgabe der Applikation diese Info entsprechend umzusetzen.

gSetLineColor (c : byte);
Diese Prozedur setzt die Linien und Füllfarbe für das aktuelle Viewport. Die Werte 0 und 255 sind dabei Standard.

gGetLineColor: byte;
Mit dieser Funktion wird die für das aktuelle ViewPort eingestellte Linien/Füllfarbe zurückgegeben.

Im **Color Linear Mode** und **xy-Address Mode** muss die Text Farbe für nachfolgende Texte gesetzt werden
gSetTextColor (c : byte);
Diese Prozedur setzt die Textfarbe für das aktuelle Viewport. Die Werte 0 und 255 sind dabei Standard

gGetTextColor: byte;
Mit dieser Funktion wird die für das aktuelle ViewPort eingestellte Textfarbe zurückgegeben.

Im **xy-Address Mode** muss die Hintergrund Farbe für Pattern Operationen gesetzt werden:
gSetBkColor (c : byte);
Diese Prozedur setzt die Hintergrund/Pattern Farbe für das aktuelle Viewport. Die Werte 0 und 255 sind dabei Standard. gClrScr und gClearView setzen dieses Atrribut auch.

Im **Column** und **ReadOnly Mode** gibt es eine zusätzliche Treiber Funktion:
***Procedure** gDispRefresh;*
Schreibt den kompletten internen Refresh Buffer in das Display. Der RefreshBuffer und diese Funktion gibt es nur im Column und ReadOnly Mode.

ViewPort Funktionen

gOpenView (ViewPort : byte; Xs, Ys, Xe, Ye : integer) : boolean;
gScaleView (ViewPort : byte; Xs, Ys, Xe, Ye : integer) : boolean;
gSwitchView (ViewPort: byte) : boolean;
gGetCurView : byte;
gClearView (ClearMode : TWriteMode); // standard mode
gClearView(color : byte); // xy-Address mode
gFrameView (ViewPort: byte);

gOpenView (ViewPort : byte; Xs, Ys, Xe, Ye : integer) : boolean;
Bestimmt die Position und Grösse eines ViewPorts in physischen Pixeln. Dabei bezeichnen Xs, Ys die obere linke Ecke und Xe, Ye untere rechte Ecke. Die Koordinaten beziehen sich auf das physische Display.

Beispiel LCD 128x128:
gOpenView (1, 0, 0, 127, 127)
eröffnet das ViewPort 1 und belegt das komplette Display.
Das Ergebnis der Funktion gOpenView ist false wenn als ViewPort das Port 0 oder eine unzulässig grosse Zahl (> Define gViewPorts) angegeben wurde.

gScaleView (ViewPort : byte; Xs, Ys, Xe, Ye : integer) : boolean;
Bestimmt die interne Skalierung des gewählten ViewPorts. Dabei bezeichnen Xs, Ys die obere linke Ecke und Xe, Ye untere rechte Ecke des logischen ViewPorts. Die Koordinaten beziehen sich auf das physische ViewPort, das zuvor mit gOpenView definiert werden muss.

Beispiel LCD 128x128:
GScaleView (1, -1000, 1000, 1000, -1000)
skaliert das ViewPort 1. Um ein Pixel in der oberen linken Ecke des ViewPorts zu setzen muss die Prozedur

gSetPixel (-1000, 1000)
aufgerufen werden. Der Koordinaten Ursprung (0, 0) befindet exakt in der Mitte des ViewPorts.
Um eine Linie vom Mittelpunkt des ViewPorts in die untere rechte Ecke zu ziehen, ist die Prozedur

gDrawLine (0, 0, 1000, -1000, $ff)
aufzurufen.
Das Ergebnis der Funktion gScaleView ist false wenn als ViewPort das Port 0 oder eine unzulässig grosse Zahl (> Define gViewPorts) angegeben wurde.

gSwitchView (ViewPort: byte) : boolean;
Bestimmt das aktuelle ViewPort. Alle folgenden Operationen, die keinen ViewPort Parameter besitzen, beziehen sich auf das Koordinaten System dieses Viewports.
Das Ergebnis der Funktion gScaleView ist false wenn als ViewPort eine unzulässig grosse Zahl (> Define gViewPorts) angegeben wurde. ViewPort 0 ist ein zulässiges aber nicht skalierbares ViewPort.

gGetCurView : byte;
Liefert als Ergebnis das aktuelle ViewPort..

gClearView(ClearMode : TWriteMode); // standard mode
Löscht das aktuelle ViewPort. Der Parameter ClearMode bestimmt dabei die Art der Operation.
ClearMode = wmClrPix setzt alle Pixel zurück auf 0
ClearMode = wmSetPix setzt alle Pixel auf 1
ClearMode = wmXorPix invertiert den Inhalt des ViewPorts

gClearView(color : byte); // xy-Address mode
Löscht das aktuelle ViewPort mit der angegeben Farbe und setzt die Hintergrund Farbe für Linien.

gFrameView (ViewPort: byte);
Zeichnet einen Rahmen um das ViewPort. Dieser Rahmen ist schreibgeschützt gegen überschreiben innerhalb des ViewPorts. Der Rahmen ist in allen Richtungen ein Pixel grösser als das ViewPort selbst.

Text Funktionen

gSetCharSet (source : pointer);
gSetTextJustify (Horiz : TtxtAlHor; Vert : TTxtAlVert);
*gGetTextJustify (**var** Horiz : TtxtAlHor; **var** Vert : TTxtAlVert);*
gSetTextMode (TextWriteMode : TWriteMode);
gSetTextBkGnd (backgnd : TTextBkGnd);
gGetTextMode : TWriteMode;
gGetTextBkGnd : TTextBkGnd;
gDrawString (X,Y: integer; zx,zy: byte; rot: TTxtRotate; str: TGraphString);
gDrawStringRel (zx, zy : byte; rot : TTxtRotate; str : TGraphString);

gSetCharSet (source : pointer);
Bestimmt den aktuellen Zeichensatz (5x7) für Text Operationen in allen ViewPorts. Der mit

***define** DefCharSet = 'Graphchars.pchr'; { = FileName}*
importierte Zeichensatz ist grundsätzlich aktiv und braucht nicht eingeschaltet werden.
Wird mehr als ein Zeichensatz verwendet, so müssen die zusätzlichen im ROM abgelegt werden mit

***const** myCharSet : **array**[1..(128 * 7) + 2] **of** byte = 'myCharSet.pchr';*
Das Argument "myCharSet.pchr" ist eine Datei, die mit dem Zeichensatz Editor PixCharEd.exe erstellt werden muss. Dieses Programm ist Bestandteil des AVRco-Systems und wird innerhalb der IDE PED32 aufgerufen. Der Zeichensatz besteht aus 128 Zeichen x 7 Bytes + 2 Bytes Info.

gSetCharSet (Addr(myCharSet));
Diese Prozedur schaltet jetzt den Zeichensatz um und dieser wird sofort gültig. Das zurückschalten auf den Standard Zeichensatz erfolgt mit
gSetCharSet (Addr(Graphchars.pchr));

Variable Zeichensätze
Wurde der Zeichensatz mit

***Define** DefCharSet = RAM; { as a RAM variable}*

definiert, erwartet die Prozedur „gSetCharSet" dass der als Parameter übergebene Zeichensatz auch im RAM liegt. Zeichensätze die im ROM liegen, müssen ins RAM kopiert werden, so dass sie verwendet werden können.
Ausserdem ist hier zu beachten, dass in diesem Fall nach PowerUp das System noch keinen Zeichensatz besitzt und dieser erst bereitgestellt werden muss.

```
{-------------------------------------------------------------}
{ Const Declarations }
const
  // 128 chars = 128 * 7 bytes + 2 bytes size info
  myCharSet : array[1..(128 * 7) + 2] of byte = 'Graphchars.pchr';        // load from file

{-------------------------------------------------------------}
{ Var Declarations }
{$IDATA}
var
  CharSet  : array[1..(128 * 7) + 2] of byte;  // variable charset
  bb       : byte;

{-------------------------------------------------------------}
{ Main }
  CopyBlock (@myCharSet, @CharSet, sizeOf(myCharSet));
  gSetCharSet (Addr(CharSet));
```

gSetCharSetRAM (RAM : boolean);
Damit wird zur Laufzeit zwischen RAM und ROM basierenden Zeichensätzen umgeschaltet.

gSetTextJustify (Horiz : TtxtAlHor; Vert : TTxtAlVert);
Bestimmt die grundsätzliche Text Ausrichtung (Alignment) für das aktuelle ViewPort. Für den Parameter Horiz sind folgende Werte möglich: *alHorLeft, alHorCenter, alHorRight*
Für den Parameter Vert sind folgende Werte möglich: *alVertBottom, alVertCenter, alVertTop*

*gGetTextJustify (**var** Horiz : TtxtAlHor; **var** Vert : TTxtAlVert);*
Liefert die aktuelle Text Alignment Einstellung für das aktuelle ViewPort zurück.

gSetTextMode (TextWriteMode : TWriteMode);
Bestimmt den Text.Schreibmodus für das aktuelle ViewPort: *wmClrPix, wmSetPix, wmXorPix*
Der Parameter bestimmt, ob die relevanten Pixel eines Zeichens gesetzt, rückgesetzt oder xoderiert werden.

gGetTextMode : TWriteMode;
Liefert den aktuellen Text Schreibmodus für das aktuelle ViewPort zurück.

gSetTextBkGnd (backgnd : TTextBkGnd);
Bestimmt den Text Hintergrund für das aktuelle ViewPort.
Ist der Parameter backgnd = bkTransp, so werden beim überschreiben eines ViewPort Inhalts mit einem Text nur die Text-relevanten Pixel beschrieben. Diese Art der Textausgabe ist relativ schnell. Bei backgnd = bkNormal wird für jedes Zeichen eines Textes die komplette 5x7 Matrix beschrieben. Das kostet Zeit. Im Normalfall genügt es wenn bkTransp gesetzt wird. Das ergibt eine wesentlich schnellere Text Ausgabe. Mit bkInvers wird die ganze Matrix invertiert geschrieben.

gGetTextBkGnd : TTextBkGnd;
Liefert den aktuellen Text-Hintergrund Mode für das aktuelle ViewPort zurück.

gDrawString (X,Y: integer; zx,zy: byte; rot: TTxtRotate; str: TGraphString);
Zeichnet den String „str“ an die logische ViewPort Position „X,Y“ mit dem horizontalen Zoomfaktor „zx“ und dem vertikalen Zoom Faktor „zy“. Der Text wird mit dem Parameter „rot“ rotiert.
TxtRot0, TxtRot90, TxtRot180, TxtRot270

Zum zeichnen des Texts werden die Text Attribute des aktuellen ViewPorts verwendet. Verletzt der Text die aktuellen ViewPort Grenzen, so wird er dort abgeschnitten (Clipping). Die Prozedur speichert in den ViewPort- Daten des aktuellen ViewPorts die Position des letzten bzw. des nachfolgenden Zeichens ab (unsichtbarer Text Cursor). Die Prozedur gDrawStringRel bezieht sich auf diesen Cursor.

gDrawStringRel (zx, zy : byte; rot : TTxtRotate; str : TGraphString);
Diese Prozedur ist identisch mit obenstehender „gDrawString“ mit der Ausnahme, dass die Positionierung relativ zum (nicht sichtbaren) Text Cursor erfolgt.

Linien Funktionen

gSetLineMode (LineWriteMode : TWriteMode);
gGetLineMode : TWriteMode;
gDrawLine (Xs, Ys, Xe, Ye : integer; pattern : byte);
gDrawLine (Xs, Ys, Xe, Ye : integer; pattern, color : byte); // xy-address mode
gDrawLineTo (Xd, Yd : integer; pattern : byte);
gDrawLineTo (Xd, Yd : integer; pattern, color : byte); // xy-address mode
gDrawLineToRel (Xr, Yr : integer; pattern : byte);
gDrawLineToRel (Xr, Yr : integer; pattern, color : byte); // xy-address mode
gDrawRect (Xs, Ys, Xe, Ye : integer; pattern : byte);
gDrawRect (Xs, Ys, Xe, Ye : integer; pattern, color : byte); // xy-address mode
gFillRect (Xs, Ys, Xe, Ye : integer; pattern : byte);
gFillRect (Xs, Ys, Xe, Ye : integer; color : byte); // xy-address mode
gDrawCircle (Xc, Yc, R : integer; pattern : byte);
gDrawCircle (Xc, Yc, R : integer; pattern, color : byte); // xy-address mode
gFillCircle (Xc, Yc, R : integer; pattern : byte);
gFillCircle (Xc, Yc, R : integer; color : byte); // xy-address mode

gSetLineMode (LineWriteMode : TWriteMode);
Bestimmt den Linien Schreib-Modus für das aktuelle ViewPort. Dieses Attribut gilt für alle nicht-Text Operationen. Folgende Attribute sind möglich: *wmClrPix, wmSetPix, wmXorPix*
Eine Linie, die mit wmXorPix gezeichnet wurde, kann mit der gleichen Operation wieder gelöscht werden, ohne dass vom Schreiben bzw. Löschen etwas sichtbar bleibt.

gGetLineMode : TWriteMode;
Liefert den Schreib-Modus des aktuellen ViewPorts zurück.

gDrawLine (Xs, Ys, Xe, Ye : integer; pattern : byte);
gDrawLine (Xs, Ys, Xe, Ye : integer; pattern, color : byte); // xy-address mode
Zeichnet eine Linie in den aktuellen ViewPort unter Berücksichtigung des Schreib-Modus dieses ViewPorts. Die Koordinate „Xs,Ys“ bezeichnet den Startpunkt und die Koordinate „Xe, Ye“ den Endpunkt der Linie. Mit „pattern“ wird die Struktur der Linie bestimmt. Liegen Teile der Linie ausserhalb des ViewPorts werden diese nicht gezeichnet, d.h. sie werden geclippt. Der Endpunkt der Linie wird in die ViewPort-Daten des aktuellen ViewPorts als Linien Cursor abgelegt. Dieser Cursor wird dann als Startpunkt für die „Rel“ Linien Operationen herangezogen.

Beispiel:
Das ViewPort hat die logische Dimension –1000, 1000, 1000, -1000
gDrawLine (-1500, 1500, 1500, -1500, $72);
zeichnet eine Diagonale von links oben nach rechts unten und clippt dabei die überstehenden beiden Enden. Durch das Pattern „$72“ wird die Linie Strichpunktiert.

gDrawLineTo (Xd, Yd : integer; pattern : byte);
gDrawLineTo (Xd, Yd : integer; pattern, color : byte); // xy-address mode
Diese Prozedur ist identisch mit obenstehender „gDrawLine“ mit der Ausnahme, dass die Start-Koordinate die zuvor abgespeicherte Line-Cursor Position des aktuellen ViewPorts ist..

gDrawLineToRel (Xr, Yr : integer; pattern : byte);
gDrawLineToRel (Xr, Yr : integer; pattern, color : byte); // xy-address mode
Diese Prozedur ist identisch mit obenstehender „gDrawLineTo“ mit der Ausnahme, dass die Start-Koordinate die zuvor abgespeicherte Line-Cursor Position des aktuellen ViewPorts ist und die End-Koordinate relativ auf diese Start-Koordinate bezogen ist.

gDrawRect (Xs, Ys, Xe, Ye : integer; pattern : byte);
gDrawRect (Xs, Ys, Xe, Ye : integer; pattern, color : byte); // xy-address mode
Zeichnet ein Rechteck in den aktuellen ViewPort unter Berücksichtigung des Schreib-Modus dieses ViewPorts. Die Koordinate „Xs,Ys“ bezeichnet eine Ecke und die Koordinate „Xe, Ye“ die gegenüberliegende Ecke des Rechtecks. Mit „pattern“ wird die Struktur der Linien bestimmt. Liegen Teile des Rechtecks ausserhalb des ViewPorts werden diese nicht gezeichnet, d.h. sie werden geclippt.

gFillRect (Xs, Ys, Xe, Ye : integer; pattern : byte);
gFillRect (Xs, Ys, Xe, Ye : integer; color : byte); // xy-address mode
Füllt ein Rechteck in den aktuellen ViewPort unter Berücksichtigung des Schreib-Modus dieses ViewPorts. Die Koordinate „Xs,Ys“ bezeichnet eine Ecke und die Koordinate „Xe, Ye“ die gegenüberliegende Ecke des Rechtecks. Mit „pattern“ wird die Füll-Struktur bestimmt. Liegen Teile des Rechtecks ausserhalb des ViewPorts werden diese nicht gezeichnet, d.h. sie werden geclippt.

gDrawCircle (Xc, Yc, R : integer; pattern : byte);
gDrawCircle (Xc, Yc, R : integer; pattern, color : byte); // xy-address mode
Zeichnet einen Kreis in den aktuellen ViewPort unter Berücksichtigung des Schreib-Modus dieses ViewPorts. Die Koordinate „Xc,Yc“ bezeichnet den Mittelpunkt des Kreises, „R“ den Radius. Mit „pattern“ wird die Struktur der Kreis-Linie bestimmt. Liegen Teile des Kreises ausserhalb des ViewPorts werden diese nicht gezeichnet, d.h. sie werden geclippt.

gFillCircle (Xc, Yc, R : integer; pattern : byte);
gFillCircle (Xc, Yc, R : integer; color : byte); // xy-address mode
Füllt einen Kreis in den aktuellen ViewPort unter Berücksichtigung des Schreib-Modus dieses ViewPorts. Die Koordinate „Xc,Yc“ bezeichnet den Mittelpunkt des Kreises, „R“ den Radius. Mit „pattern“ wird die Füll-Struktur der Kreis-Linie bestimmt. Liegen Teile des Kreises ausserhalb des ViewPorts werden diese nicht gezeichnet, d.h. sie werden geclippt.

Pixel Funktionen

gSetPixel (Px, Py : integer);
gSetPixel (Px, Py : integer; color : byte); // xy-address mode
gClearPixel (Px, Py : integer); // not for xy-address mode
gXorPixel (Px, Py : integer); // not for xy-address mode

gSetPixel (Px, Py : integer);
gSetPixel (Px, Py : integer; color : byte); // xy-address mode
Setzt einen Pixel an der Stelle „Px,Py“ im aktuellen ViewPort. Liegt die Koordinate ausserhalb des ViewPorts wird der Pixel nicht gezeichnet.

gClearPixel (Px, Py : integer); // not for xy-address mode
Löscht einen Pixel an der Stelle „Px,Py“ im aktuellen ViewPort. Liegt die Koordinate ausserhalb des ViewPorts wird der Pixel nicht gelöscht.

gXorPixel (Px, Py : integer); // not for xy-address mode
Invertiert einen Pixel an der Stelle „Px,Py“ im aktuellen ViewPort. Liegt die Koordinate ausserhalb des ViewPorts wird der Pixel nicht verändert.

Line Cursor Funktionen

gMoveTo (Xd, Yd :integer);
gMoveToRel (Xr, Yr : integer);

gMoveTo (Xd, Yd : integer);
Setzt den virtuellen Zeichen Cursor auf die Koordinate „Xd, Yd“. Der Zeichen Cursor wird für die relativen Linien-Operationen benutzt.

gMoveToRel (Xr, Yr : integer);
Bewegt den virtuellen Zeichen Cursor relativ zu seiner alten Position. Der Zeichen Cursor wird für die relativen Linien-Operationen benutzt.

Kopier Funktionen

gDrawBitmap (Xs, Ys : integer; source : pointer; DrawMode : TWriteMode); // not for xy-address mode
gDrawBitmapN (Xs, Ys : integer; source : pointer; DrawMode : TWriteMode); // not for xy-address mode
gDrawBitmapXY(Xs, Ys : integer; source : pointer; DrawMode : TWriteMode; color : byte); // xyAddress mode

gSaveView (dest : pointer); // not for color modes
gRestoreView (source : pointer); // not for color modes

gDrawBitmap (Xs, Ys : integer; source : pointer; DrawMode : TWriteMode);
gDrawBitmapXY(Xs, Ys : integer; source : pointer; DrawMode : TWriteMode; color : byte); // xyAddress mode
Kopiert ein BitMap aus dem RAM/ROM in das aktuelle ViewPort. Der Kopiervorgang wird ***transparent*** ausgeführt, d.h. nur gesetzte Bits im Bitmap werden kopiert. Der Parameter „DrawMode“ bestimmt den Schreib Modus: wmClrPix, wmSetPix, wmXorPix

gDrawBitmapN (Xs, Ys : integer; source : pointer; DrawMode : TWriteMode); // not for xyAddress mode
Kopiert ein BitMap aus dem RAM/ROM in das aktuelle ViewPort. Der Kopiervorgang wird ***nicht-transparent*** ausgeführt, d.h. alle Bits im Bitmap werden kopiert. Der Parameter „DrawMode“ bestimmt den Schreib Modus: *wmClrPix, wmSetPix, wmXorPix*, wobei *wmClrPix* das Bitmap invertiert.

Die Koordinate „Xs, Ys“ bezeichnet den Zielpunkt im ViewPort. Der Parameter „source“ zeigt auf die entsprechende Speicherstelle (Quelle) im RAM/ROM.

```
const
    // 32*32 pixels = 128bytes + 2 bytes size info
    myBitMap : array[1..(32*32 div 8) + 2] of byte = 'E-LAB.pbmp';
```

Das Argument “E-LAB.pbmp“ ist eine Datei, die mit dem BitMap Editor BMPedit.exe erstellt werden muss. Dieses Programm ist Bestandteil des AVRco-Systems und wird innerhalb der IDE PED32 aufgerufen. Der Editor akzeptiert einfache schwarz/weiss Windows-BitMaps und konvertiert diese. Mit dem Editor können auch direkt BitMaps erstellt werden.

Ein BitMap besteht aus ((xPixels * yPixels) div 8) Bytes + 2 Bytes Info**.**

gDrawBitMap (207, 5, @myBitMap, wmXorPix);

Weiterhin ist es möglich zur Laufzeit zwischen RAM und ROM basierenden Bitmaps umzuschalten. Dies geschieht mit der Prozedur

gSetBitMapRAM (RAM : boolean);

gSaveView (dest : pointer);
Speichert den Inhalt des aktuellen ViewPorts in das RAM an die Stelle, auf die der Parameter „dest“ zeigt. Das Ziel sollte ein Array of Byte sein, das so gross sein muss, dass es auch alle Bytes des ViewPorts aufnehmen kann. Eine Überprüfung findet **nicht** statt.

```
var
  saveVP : array[0..4095] of byte;
  gSaveView (@saveVP);
```

gRestoreView (source : pointer);
Uberschreibt das aktuelle ViewPort mit dem Inhalt des Speichers, auf den der Parameter „source“ zeigt. ViewPort Attribute werden ignoriert.

gRestoreView (@saveVP);

Hilfs Funktionen

*RotatePntI (angle : integer; XPo, YPo : integer; **var** XPd, YPd : integer);*
SinInt (angle, v : integer) : integer;
CosInt (angle, v : integer) : integer;

*RotatePntI (angle : integer; XPo, YPo : integer; **var** XPd, YPd : integer);*
Der Punkt(XPo, YPo) wird mit dem Winkel „angle“ rotiert (Grad). Der Mittelpunkt der Rotation ist 0,0. Das Ergebnis steht in XPd, YPd.

SinInt (angle, v : integer) : integer;
Die Funktion liefert den Sinus des Winkels multipliziert mit dem Integer Argument. Sehr schnell und kurz!

CosInt (angle, v : integer) : integer;
Die Funktion liefert den Cosinus des Winkels multipliziert mit dem Integer Argument. Sehr schnell und kurz!

3.6.5 Text Display

Einige Graphic Controller können auch zum Text Display umgeschaltet werden. Davon wiederum sind auch welche in der Lage Text und Graphic gleichzeitig darstellen zu können. Ist das nicht möglich, so kann durch ein paar zusätzliche Funktionen dies auch via Software möglich gemacht werden.

Das folgende Beispiel zeigt anhand eines Column Mode 98x64 Displays wie das gemacht werden kann. Für lineare Displays sind nur ein paar Statements anzupassen.

```
Unit GraphicText;
// Graphics_LCD Text Unit
// This uses the Graphic Display as a Text Display

Interface

type
  tLCDstr = string[16];                  // depends on the x-pixel count

  Procedure LCDClr;                      { Clear entire screen }
  Procedure LCDclrEol;                   { Clear to End Of line  }
  Procedure LCDclrLine(line : byte);     { Clear current line    }
  Procedure LCDXY( X, Y : byte );
  Procedure LCDWrXY(X, Y : Byte; St : tLCDstr);
  Procedure LCDWrite(St : tLCDstr);
  Procedure LCDvalidate;                 { Coulumn mode final update }

{$IDATA}

Implementation

{$IDATA}
{-------------------------------------------------------------}
{ Type Declarations }
type

{-------------------------------------------------------------}
{ Const Declarations }
const
  // Size of Graphic Display 98x64
  // Size of Text Display 16*7
  Max_X    : byte  = 16;            // 16 x 6 -> 96
  Max_Y    : byte  = 7;             //  7 x 9 -> 63

structconst
  emptyStr      : tLCDstr   = '              ';
```

```
{---------------------------------------------------------------}
{ Var Declarations }
{$IDATA}
var
  { Current X,Y Positions on LCD }
  CurrentX    : byte;                 // char count
  CurXA       : byte;                 // pixel count
  CurrentY    : Byte;
  CurYA       : byte;

{ functions }
Procedure LCDvalidate;
begin
  gDispRefresh;
end;

Procedure LCDXY ( X1, Y1 : Byte );
begin
  { set the global X Y locations }
  CurrentX := x1;
  CurXA:= CurrentX * 6;
  CurrentY := y1;
  CurYA:= (CurrentY * 9)+9;
end;

Procedure LCDClr;
Begin
  gSetTextMode (wmSetPix);
  gSetTextBkGnd (bkNormal);
  gSetTextJustify (ALHORLEFT, ALVERTBOTTOM);
  LCDXY (0, 0);
  gClrScr (0);
End;

Procedure LCDClrEol;          {  Clear to End Of line  }
var
  temp_x        : Byte;
begin
  temp_X := CurrentX;
  emptyStr[0]:= char(Max_X - CurrentX);
  LCDWrite (emptyStr);
  LCDXY (temp_x, CurrentY);  { restore position }
end;

Procedure LCDclrLine (line : byte);
begin
  LCDXY (0, line);
  LCDClrEol;
end;

Procedure LCDWrXY (X, Y : Byte; St : tLCDstr);
begin
  LCDXY (X, Y);
  LCDwrite (st);
end;

Procedure LCDWrite (St : tLCDstr);
Begin
  GDrawString (Integer (CurXA), Integer(CurYA), 1{H}, 1{V}, TxtRot0, st);
  CurrentX := CurrentX + (length(st));
  CurXA:= CurrentX * 6;
End;
```

Initialization

Finalization

End GraphicText.

3.6.6 Support Programme

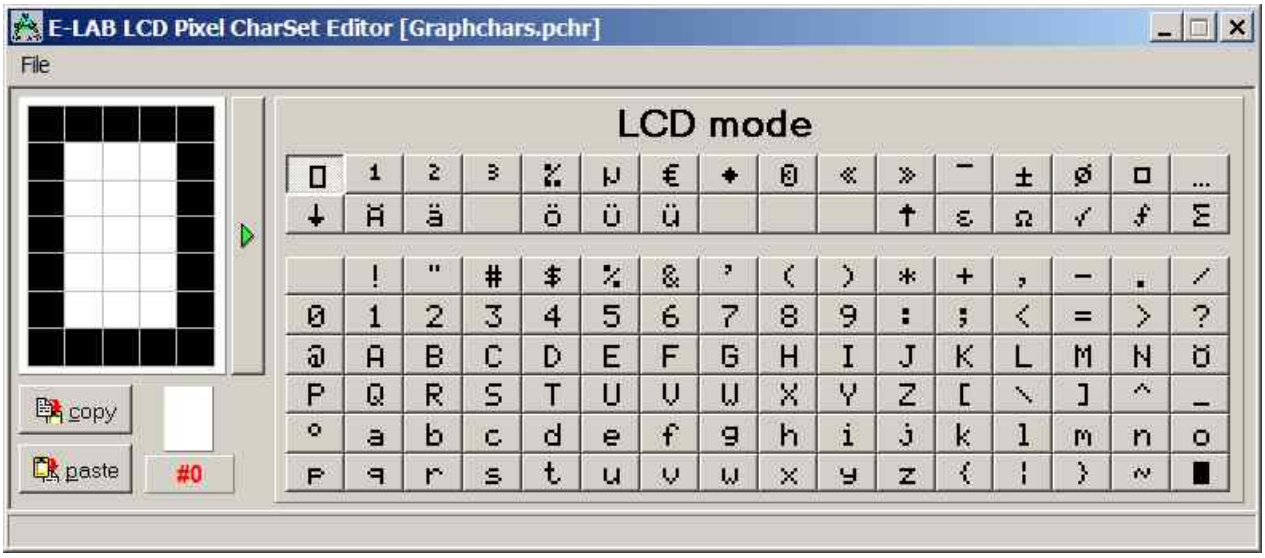
3.6.6.1 PixCharEd.exe

String Ausgaben auf das Display benötigen ein passenden Zeichensatz mit der Grösse 5x7 Pixel. Ein Zeichensatz besteht immer aus (128 Zeichen x 7 Bytes) + 2 Bytes Info. Im System ist ein solcher Zeichensatz enthalten in der Datei Graphchars.pchr. Aufruf des Programms im Editor mit dem Button 

Der User kann allerdings seinen eigenen Zeichensatz erstellen oder einen zusätzlichen, z.B. einen kyrillischen. Dazu dient dieser Zeichensatz Editor. Dieses Programm ist Bestandteil des AVRco-Systems und wird innerhalb der IDE PED32 aufgerufen.

Der Editor generiert eine binäre Datei, die als Konstante in die Source eingebunden wird:

```
const
  CharSet : array[1..(128 * 7) + 2] of byte = 'FileName.pchr';
```

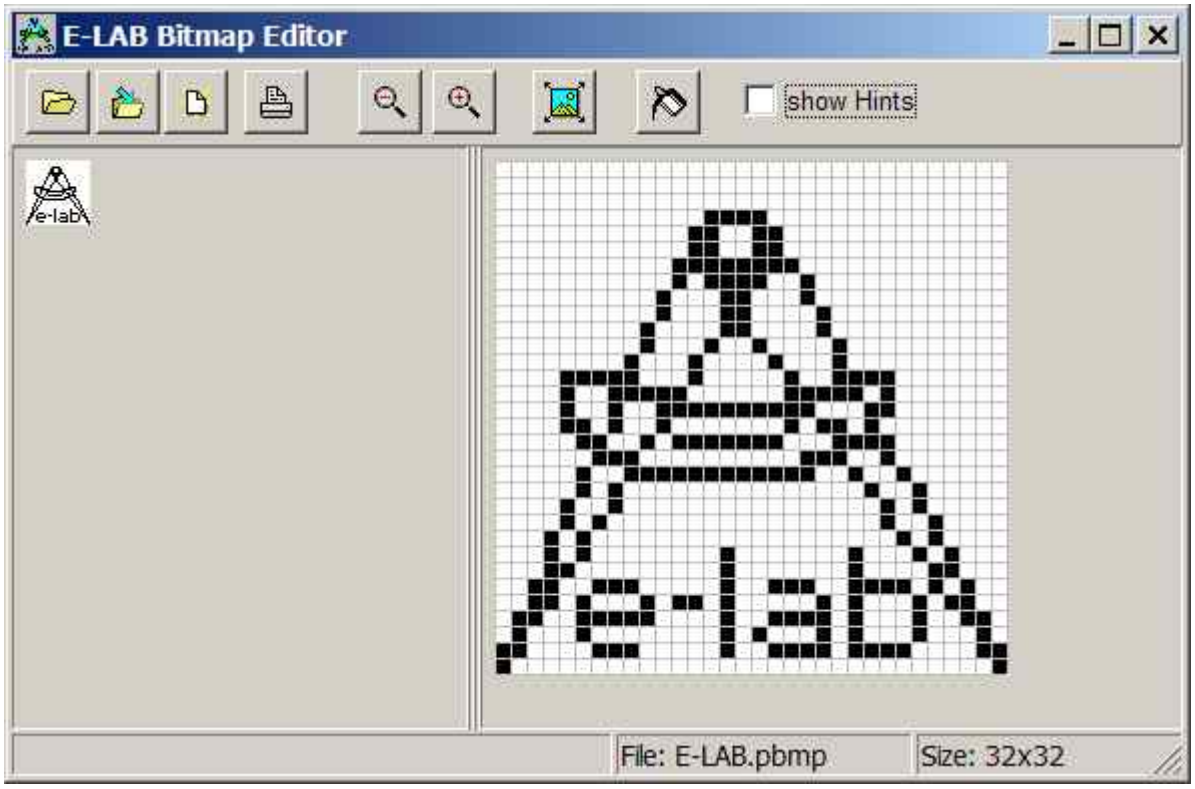


3.6.6.2 BMPedit.exe

Dieser Editor gestattet es einfache Windows schwarz/weiss BitMaps in AVRco BitMaps zu konvertieren. Das original Bitmap muss in x und y Richtung jeweils ein vielfaches von 8 gross sein. Weiterhin können hiermit BitMaps komplett neu erstellt werden. Dieses Programm ist Bestandteil des AVRco-Systems und wird innerhalb der IDE PED32 mit dem Button  aufgerufen.

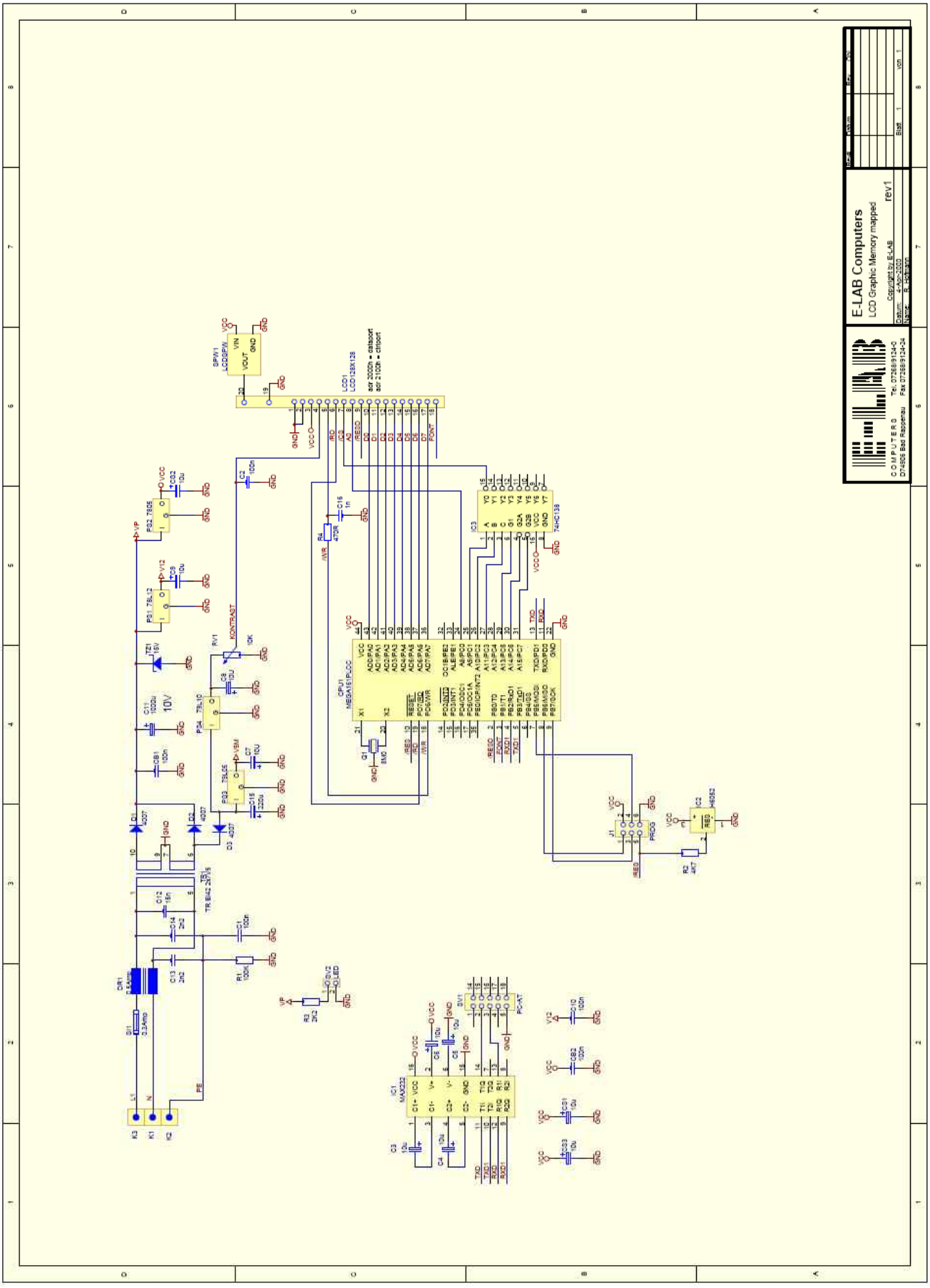
Der Editor generiert eine binäre Datei, die als Konstante in die Source eingebunden wird:

```
const
    // 32*32 pixels = 128bytes + 2 bytes size info
    myBitMap  : array[1..(32*32 div 8) + 2] of byte = 'FileName.pbmp';
```

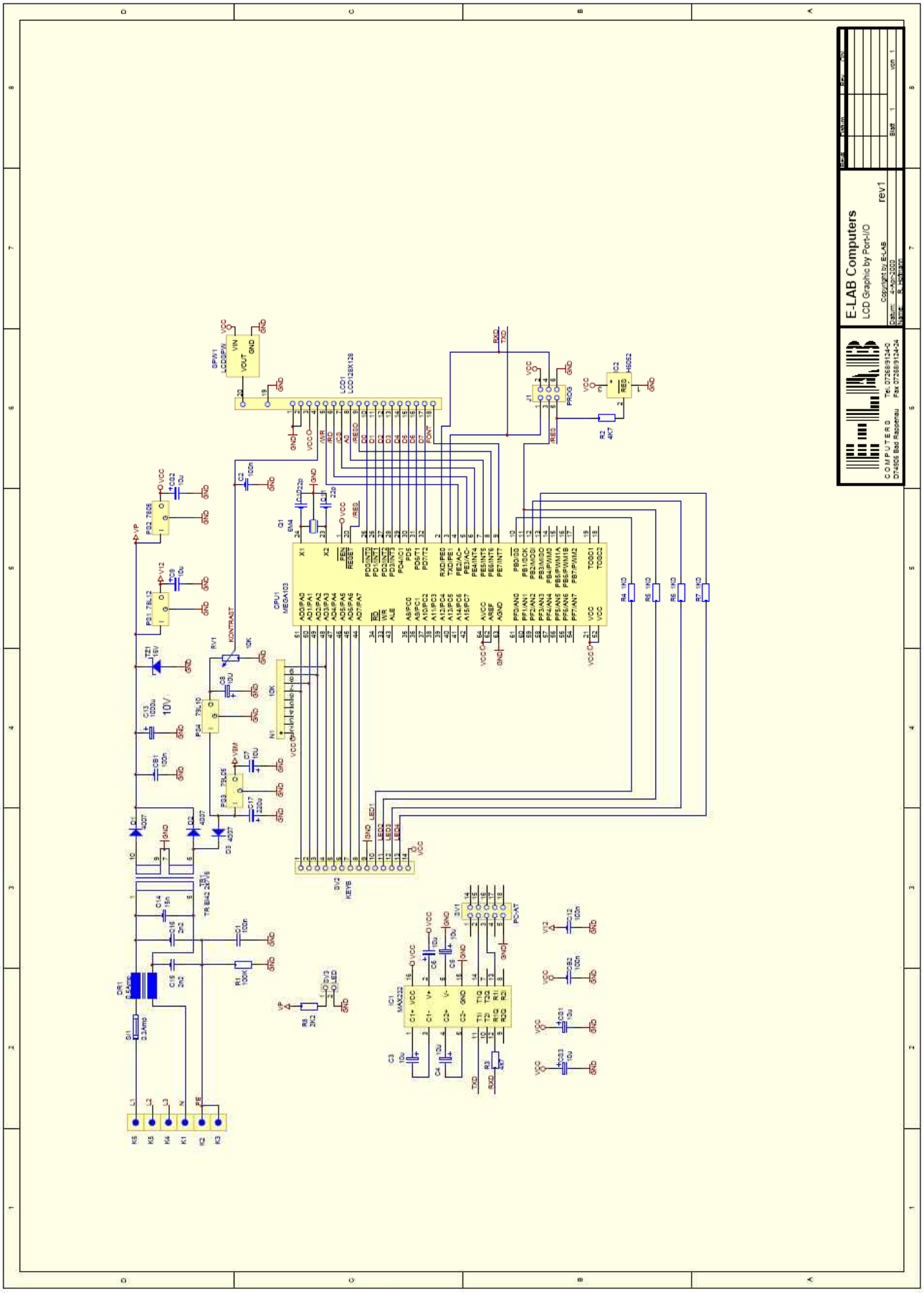


Programm Beispiele und Schaltpläne:

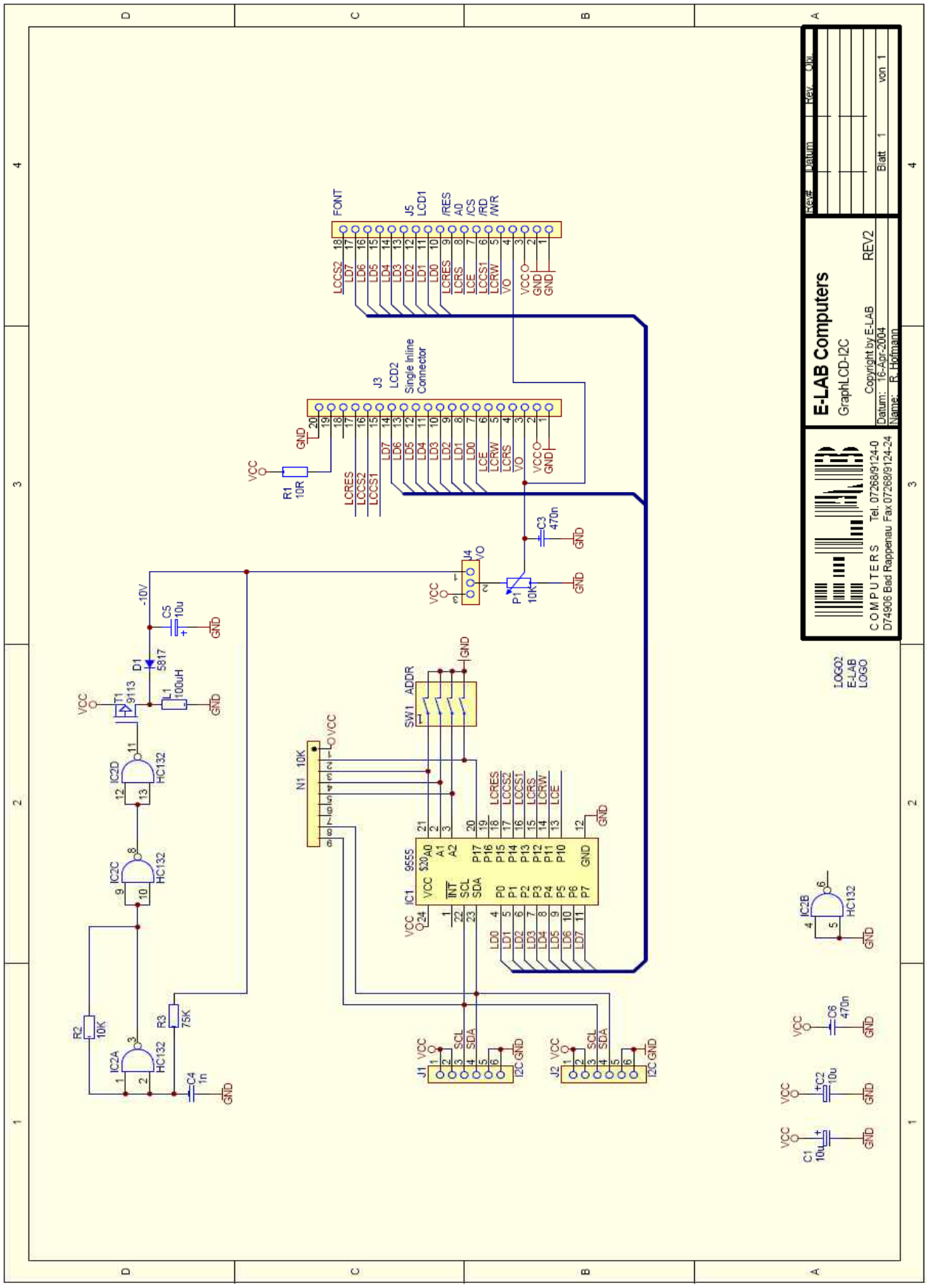
ein Beispiel befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\LCD_PCF8548**
ein Beispiel für I2C Anbindung befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\LCD_KS0108**
ein Beispiel für SPI connection ist in der Directory **..\E-Lab\AVRco\Demos\LCD_ST7565R (AVR Graph7565)**
ein weiteres in der Directory **..\E-Lab\AVRco\Demos\XMega_LCDgraph**



Schaltplan **Memory Mapped LCD Graphic**

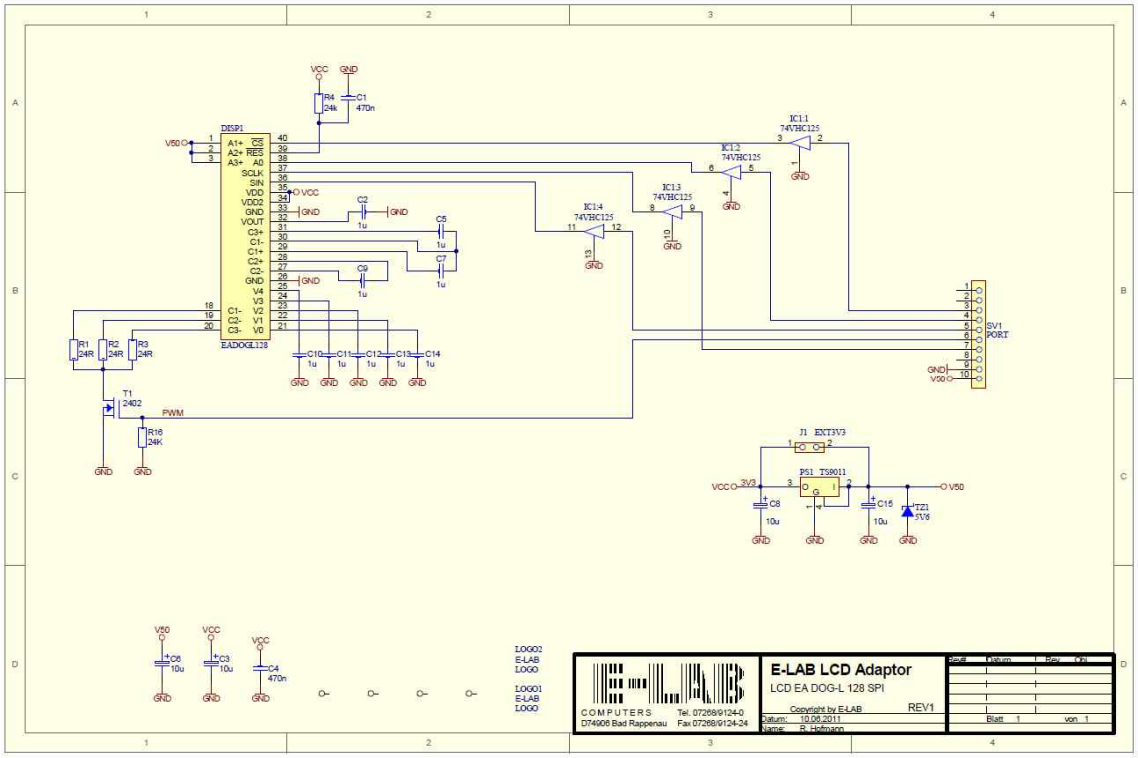


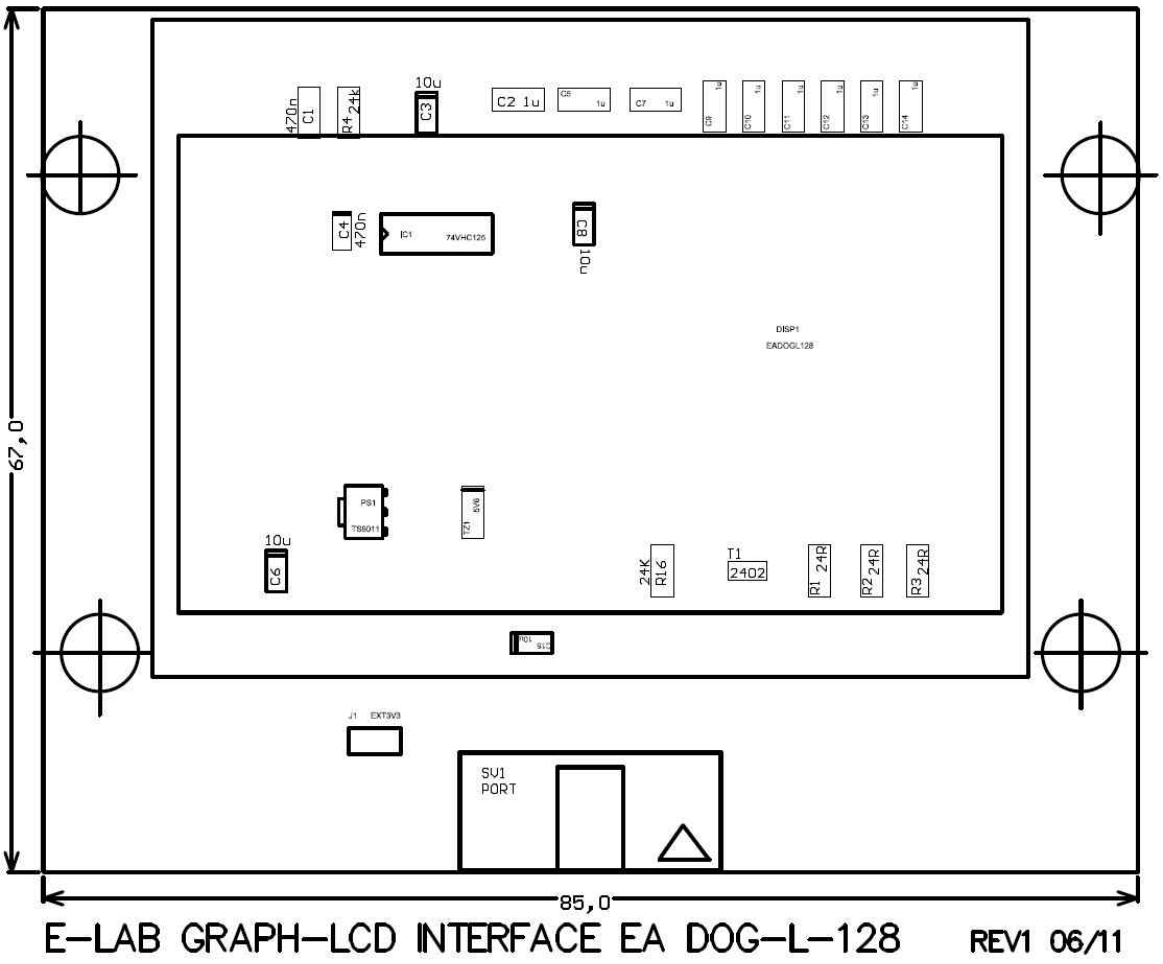
Schaltplan **Port Mapped LCD Graphic**



Schaltplan **TWI Interface LCD Graphics**

Schaltplan **LCD Graphic mit SPI Interface EA-DOG-128x64**





ein Beispiel für SPI ist in der Directory **..\E-Lab\AVRco\Demos\LCD_ST7565R (AVR Graph7565)**
ein weiteres in der Directory **..\E-Lab\AVRco\Demos\XMega_LCDgraph**

3.7 DDS10 Sinus-Dreieck Synthesizer

Die Erzeugung von sinusförmigen analogen Signalen mit einem Mikroprozessor ist relativ schwierig und aufwändig und sollte i.A. auch analoger Hardware vorbehalten bleiben. Allerdings macht es u.U. durchaus Sinn dies durch einen Prozessor zu erledigen, vor allen Dingen wenn die Frequenz auf Hertz oder sogar Bruchteile davon genau und stabil sein soll. Hier ist die digitale Lösung der analogen weit überlegen.

Man muss sich aber auch über die Nachteile der digitalen Lösung im klaren sein. Dies sind die immer vorhanden Treppen des DA-Wandlers und bei der Synthese das zusätzlich vorhandene Phasenrauschen. Je höher die Frequenz an die Grenzen der Implementierung kommt wird auch die Auflösung der Amplitude schlechter, d.h. an den Grenzen wird der Sinus u.U. nur noch aus 6..8 analogen Werten gebildet. Damit steigt der Klirrfaktor und evtl. die Amplituden Variation.

In diesen Grenzbereichen können alle diese Nachteile durch geschickte Filterung weitgehend wieder unterdrückt werden. Die Amplituden Auflösung beträgt 8 Bits. Die Frequenz Auflösung ist besser als 0.1Hertz.

Der Vorteil der Synthese ist wie gesagt die hervorragende Frequenz Stabilität und die präzisen Einstell-Möglichkeiten der Frequenz. Mit etwas Software Aufwand kann auch eine Amplituden oder Frequenz Modulation implementiert werden, soweit das mit 8 Bit DA-Wandlern möglich ist.

Einführung DDS10
Dieser Treiber dient dazu programmgesteuert eine sinusförmige Frequenz oder alternativ einen Sägezahn zu erzeugen. Die Synthese spielt sich innerhalb eines Timer Interrupts ab. Der Timer ist frei wählbar und kann ein 8 oder 16bit Timer sein. Da die komplette Berechnung in dem Timer Interrupt stattfindet und dieser mindestens 10mal öfters durchlaufen werden muss als die max. einstellbare Frequenz, zeigt sich hier ganz schnell die Obergrenze der erzielbaren Frequenz.

Eine typische Berechnung dauert inkl. Interrupt Handling ca. 50..60 Prozessor Zyklen. Bei 16MHz sind dies ca. 3..4usec. Will man als max. Ausgangsfrequenz 10kHz erreichen, muss der Timer Interrupt mit 100kHz arbeiten. Dies bedeutet, dass alle 10usec dieser Interrupt zuschlägt und dann 3..4usec Rechenzeit braucht. Bei der Zykluszeit von 10usec bleiben dann für den Rest der Applikation noch 6..7usec übrig. Oder anders ausgedrückt die Synthese verbrauch 30..40% der Rechenzeit des System, für den Rest und die Applikation bleiben dann noch 60..70% übrig.

Damit ist auch klar, dass andere Interrupts zumindest zu einem Jitter (Frequenzmodulation) des Sinus führen. Ist die Sperrzeit durch andere Interrupts grösser als 10usec gehen Timer Interrupts verloren und es kommt an dieser Stelle zu einer zumindest temporären Frequenz Erniedrigung. Damit muss auch klar sein, dass MultiTasking mit seiner u.U. 20..50usec langen Interrupt Sperrzeit hier zu starken Frequenz Fehlern führt. Das gleiche gilt in etwas abgeschwächter Form für den evtl. vorhandenen SysTick, wenn dieser viel zu tun hat, z.B. ADC lesen, SoftTimer etc.

Das Systemdesign bestimmt in diesem Fall die Frequenz Stabilität. Natürlich kann hier auch MultiTasking laufen. Mann muss dann entweder mit den Störungen durch den SysTick/Scheduler leben, oder man unterbindet in der Zeit wo der Synthesizer aktiv ist, den SysTick komplett.

Systembedingt ist also 10kHz die Obergrenze. Wollte man 100kHz synthetisieren, müsste der Timer mit 1MHz Interrupts laufen und die Rechenzeit innerhalb des Interrupts darf 300..400nSec nicht überschreiten. Dies würde dann allerdings eine CPU mit 100..200MHz bzw. mips bedeuten.

Um diese 10kHz mit einem AVR erreichen zu können, muss dieser mindestens mit 8..10MHz laufen. Dann bleibt allerdings für den Rest der Applikation fast nichts mehr übrig. Damit sind 16MHz ein absolutes Muss.

Läuft der Synthesizer nicht im Interrupt sondern als Hauptprogramm Schleife und es sind keine Interrupts vorhanden, dann lassen sich auch mit einem 16MHz AVR 100kHz problemlos herstellen. Dies ist jedoch nicht der Sinn und die Aufgabe dieses Treibers.
Der synthetische Sinus wird als Tabelle im ROM oder RAM gehalten und wird als 8 Bit Wert entweder auf einem Port parallel ausgegeben oder es kann der SPIport dafür benutzt werden. Der SPI kann dann allerdings für nichts anderes mehr verwendet werden. Ein UserPort ***DDS10IOS*** wird auch unterstützt.

Für Test Zwecke eignen sich die E-LAB Boards *SerDAC* und *SpeechDemo* die beide den Parallel- und den SPI Mode unterstützen.

3.7.1 Implementation

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

Der Import von DDS10 importiert auch automatisch diverse Bibliotheksfunktionen.

Imports

```
Import  SysTick, DDS10;          // and SPIdriver if necessary
or
Import  SysTick, DDS10_3P;       // 3 phases sine mode
```

Defines
Der Treiber benutzt einen internen Timer, entweder Timer1, Timer2 oder Timer3, falls vorhanden.

```
Define  ProcClock      = 16000000;   {16Mhz clock }
        DDS10Timer     = Timer1;     {use a 16bit Timer}
        DDS10port      = PortA;
        // DDS10port   = SPI;
        // DDS10port   = UserPort ;
        DDS10Tables    = 1;          {use 2 lookup tables}
```

DDS10Timer
Definiert den zu verwendenden 8/16bit Timer. Dieser Timer läuft intern im Interrupt und darf für nichts anderes mehr verwendet werden.

XMegas
Es können die Timer_C0 bis Timer_F1 verwendet werden, soweit vorhanden.

```
DDS10Timer = Timer_E1;
```

DDS10port
Definiert das DAC Port. Hierzu kann entweder ein normales CPU Port verwendet werden oder der CPU-interne SPI. Wird der SPI verwendet muss dieser Treiber importiert und definiert werden

```
SPIorder   = LSB;
SPIcpol    = 1;
SPIcpha    = 1;
SPIpresc   = 0;        // presc = 0..3 -> 4/16/64/128
SPI_SS     = true;     // use SS pin as chipselect
```

Hierbei sind die letzten zwei Defines ein Muss, d.h. der SPI muss mit maximaler Geschwindigkeit laufen und der SS-Pin muss freigegeben sein. Die anderen Defines hängen vom verwendeten SPI DAC Slave ab.

XMegas
Es können die SPIs SPI_C bis SPI_F verwendet werden, soweit vorhanden. Das SPI Port wird dazu nicht speziell importiert, sondern wird im Define ausführlich spezifiziert:

```
DDS10Timer   = Timer_C1;
DDS10port    = SPI_C, SPImode3, SPImsb, PortF, 4;  // Mode0..3, MSB/LSB, SS-Port, SS-Pin
```

Wird im Define das UserPort angegeben, so erfolgt die Ausgabe in die von der Applikation bereitzustellende IOS-Funktion:

```
UserDevice DDS10IOS(b : byte);
begin
  ...
end;
```

Im 3-Phasen Mode

```
UserDevice DDS10IOS(p0, p1, p2 : byte);
begin
  ...
end;
```

3.7.2 DDS10 Tables

Der Synthesizer arbeit mit einer oder mehreren Sinustabellen. Aus Geschwindigkeits und Genauigkeits Gründen kann eine Amplituden Änderung oder Modulation nicht direkt zur Laufzeit erfolgen, sondern muss als ganzes gerechnet und in eine der 256 Byte Tabellen im RAM abgespeichert werden.

Dieses Define steuert ganz prinzipiell das Verhalten und die Möglichkeiten des DDS Treibers. Wird hier “0“ angegeben, so befindet sich die 256 Byte grosse Sinus Tabelle im ROM und eine Amplituden Umschaltung ist nicht möglich, es wird aber praktisch auch kein weiteres RAM gebraucht.

Bei Werten 1..4 befinden sich entsprechend viele Tabellen im RAM und das ROM wird nicht benutzt. Jetzt können bis zu 5 Tabellen erzeugt werden, die alle unterschiedliche Amplituden Werte haben. Dadurch kann ohne Verzögerung zwischen den einzelnen Amplituden umgeschaltet werden. Die Umschaltung selbst findet allerdings im Null-Durchgang des Signals statt.

Während die Tabelle 0 (100% Amplitude) bei DDStables = 0 automatisch im ROM erstellt und zur Laufzeit nicht verändert werden kann, müssen bei DDStables > 0 die Tabellen 0..4 mindestens 1 mal durch die Applikation erstellt werden und können dann jederzeit neu gerechnet werden.

Da eine Tabelle 256 Byte benötigt, kann das bei kleinen CPUs ganz schnell den Speicher füllen, da diese Tabellen ja auch noch auf einer 256 Byte Grenze liegen müssen. Deshalb legt sie der Compiler direkt am RAM Ende ab.

Aus Präzisions Gründen werden einige Teile des Treibers in Floating Point gerechnet (Unit UDDS10), deshalb ist der Import von Float zwingend. Ebenfalls zwingend ist der Import der **Unit UDDS10**.

Beim 3-Phasen Mode kann nur mit der Table im Flash gearbeitet werden. *Define DDS10Tables = 0;*

3.7.3 Typen und Prozeduren

```
type
  tdsMode  = (dsSine, dsTriaLeft, dsTriaSym, dsTriaRight, dsSquare);
```

Diese ersten drei Prozeduren sind immer vorhanden und haben auch immer eine Auswirkung.

DDS10setFrequ
Hiermit wird die aktuelle Frequenz eingestellt. Der Wert kann im Bereich kann von 0.1Hz bis 10kHz liegen. Die Frequenz Änderung findet beim nächsten Null-Durchgang des Sinus statt. Nur im Stop Mode verwenden

```
Procedure DDS10setFrequ (freq : float);
```

DDS10start
Hiermit wird eine evtl. abgeschaltete Synthese neu initialisiert wieder gestartet.

```
Procedure DDS10start;
```

DDS10stop
Diese Prozedur schaltet die Frequenz Synthese aus. Der statische Ausgabe Werte wird dabei auf 127 gestellt was 50% oder Sinus 0 am DA-Wandler bedeuten.

Procedure *DDS10stop;*

DDS10run *nur XMega*
Hiermit wird eine Stop Operation wieder aufgehoben. Für schnellen Frequenz Wechsel.

Procedure *DDS10run;*

Die folgenden zwei Prozeduren sind nur vorhanden, wenn das Define **DDS10Tables > 0** ist,
d.h. mehr als eine Lookup Table definiert wurde.

DDS10buildTab
Wurden mehr als eine Lookup Table definiert (DDS10tables > 0) so sind die RAM Tabellen nach Programmstart nicht initialisiert. Um diese benutzen zu können, müssen sie initialisiert werden. Mit der Prozedur DDS10BuildTab wird eine Tabelle ausgewählt und ihr mit dem Parameter “amp“ die gewünschte Amplitude zwischen 0 und 100% übergeben.

Es bietet sich natürlich an, den Synthesizer nicht nur zum Erzeugen eines Sinus zu benutzen, sondern auch für Sägezähne. Der Parameter dsMode bestimmt dabei ob ein Sinus, ein steigender, abfallender oder symmetrischer Sägezahn, oder ein symmetrisches Rechteck erzeugt werden soll.

Procedure *DDS10buildTab(DDStab, amp : byte; dsMode : tdsMode);*

DDS10setTab
Wurden mehr als eine Lookup Table definiert (DDS10tables > 0) so kann zur Laufzeit zwischen den vorhandenen Tabellen umgeschaltet werden. Das Umschalten selbst findet im Nulldurchgang statt. Deshalb kann die nächste Umschaltung frühestens nach einem kompletten Zyklus stattfinden.

Procedure *DDS10setTab (DDStab : byte);*

3.7.4 XMega und XMega-DAC

Als Output Device kann beim XMega auch der interne DAC verwendet werden:

```
Import …, DDS10, DAC_B, …;
…
Define
  …
  DAC_B          = chan01, REFextB;    // DAC_B channel 0 + 1 used
  DDS10Timer     = Timer_D1;           // use Timer_D1
  DDS10port      = DAC_B1;             // use DAC_B1
```

Wird der interne DAC verwendet, dann wird auch diese Prozedur exportiert:

procedure *DDS10SetGain(gain : byte);*

hiermit kann zur Laufzeit die Verstärkung eingestellt werden. Allerdings sollte dann dazu in *DDS10buildTab* der Parameter amp auf 100 gestellt werden.
Gain 0 = vOut x0
Gain 1 = vOut x1
Gain 2 = vOut x2
Gain 3 = vOut x8
Gain 4 = vOut x16

Programm Beispiel:
ein Beispiel befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\DDS10**
ein XMega Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_DDS10**
ein XMega-DAC Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_DDS10**
ein XMega-3Phasen Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_DDS10_3P**

3.8 File System

Für grössere und/oder komplexere Filesysteme bitte FAT16 File System benutzen.

Erfassung, Verwaltung, Speicherung und Abruf von Daten sind Aufgaben, die oft auch von Mikroprozessor Systemen erledigt werden müssen. Zum Glück für die meisten Programmierer hält sich das Datenaufkommen hierbei in Grenzen. Meistens sind das wenige Kilobytes, die problemlos selbst in Arrays, Records etc. im gepufferten Speicher oder im EEprom gehalten und verwaltet werden können.

Werden die Datenmengen grösser und handelt es sich dabei noch um unterschiedliche Datenstrukturen wird die Handhabung wesentlich schwieriger oder gar unmöglich. Hier setzt ein Datei System an. Die Dateien (Datenblöcke) können fast beliebig gross werden und die Datenstruktur der Dateien ist beliebig. Man darf jetzt allerdings nicht dem Irrtum verfallen, alles und jegliches mit einem Filesystem verwirklichen zu müssen. Ein Datei System ist immer ein Verwaltungs Overhead und kostet nicht unerheblich Programmcode, Arbeitspeicher und Rechenzeit. Die Anforderungen an die System Resourcen sind nicht zu unterschätzen.

Das hier implementierte Datei System ist ganz stark an das von früher her bekannte File System CP/M Z80 angelehnt. Deshalb ist es sehr schnell und verbraucht relativ wenig System Resourcen. Ein minimal System ist schon in 6kB Code und ca. 500 Bytes RAM zu erstellen. Allerdings muss dabei eine fast fehlende Directory Struktur in Kauf genommen werden. Es gibt hier nur maximal 10 Directories, die immer feste Namen haben '0'..'9'. Es gibt nur eine Directory Ebene (flat, nonhierarchical). Die Dateinamen entsprechen der DOS Konvention mit max. 8 Zeichen Länge und max. 3 Zeichen Extension. Die Zahl der Laufwerke ist auf 4 beschränkt 'A'..'D'.

Eine weitere Einschränkung von CP/M wurde hierbei nicht übernommen. Die Granulation bei CP/M ist 128 Bytes bzw. 1 Sector. Das bedeutet beim Feststellen der Dateigrösse kann minimal auf 128 Bytes aufgelöst werden. Ein Einzelbyte schreiben oder lesen ist auch nur mit Tricks möglich und das Anhängen an eine Datei ist nur in Sektor Grösse möglich. Das bringt bei typisierten Dateien doch erhebliche Schwierigkeiten mit sich. Unsere Implementation umgeht dies, indem auch noch ein Byte Zähler für den letzten Sector einer Datei mit abgespeichert wird.

Es wurden alle geläufigen Filesystem, Drive und File Operationen implementiert. Das Filesystem arbeitet auf einen Hardwaretreiber **FileIOS,** dem bestimmte Kommandos und Parameter übergeben werden und der diese in Zusammenhang mit der Disk Hardware (externes SRAM, EEPROM, FLASH, Floppy, Harddisk etc) ausführen muss. Der Treiber muss vom Anwender in Abhängigkeit von seinem Speichermedium selbst erstellt werden.

Technische Daten

Laufwerke logisch/physisch	:	1..4	32kBytes ... 8Mbyte pro Laufwerk
DiskSize <= 256kB	:	BlockSize	1kB
		Min FileSize	1kB
		Max. Files	32
DiskSize <= 512kB	:	BlockSize	2kB
		Min FileSize	2kB
		Max. Files	64
DiskSize <= 1024kB	:	BlockSize	4kB
		Min FileSize	4kB
		Max. Files	128
DiskSize <= 2048kB	:	BlockSize	8kB
		Min FileSize	8kB
		Max. Files	256
DiskSize <= 8096kB	:	BlockSize	16kB
		Min FileSize	16kB
		Max. Files	512

3.8.1 Begriffs Erklärung

Sector
Ein Sector ist die kleinste physische Dateneinheit, die das System vom Datenträger Lesen oder Schreiben kann. Die Grösse eines Sector ist konstant und beträgt 128 Bytes. Die von der Anwendung bereit zustellende **FileIOS** Function muss damit immer einen 128Byte Block vom Speichermedium lesen oder schreiben.

Block
Ein Block ist die kleinste System-interne Dateneinheit, die das System verwalten kann. Die Grösse eines Blocks ist von der Disk Grösse abhängig und liegt zwischen 1kB und 16kB. Die Anwendung hat mit Blöcken direkt nichts zu tun. Es ist aber für den Programmierer wichtig zu wissen, dass eine Datei zumindest einen Block gross ist, wenn diese auch nur ein Byte enthält.

Track, Sector
Der Treiber **FileIOS** stellt zusätzlich zu der absoluten Byte-Adresse für das Lesen und Schreiben auch die Parameter Track und Sector zur Verfügung. Diese sind optimal für Block-orientierte Medien (Floppy, Flash, etc) geeignet.

Record
Ein Record bezeichnet die kleinste Lesbare Datenmenge, die die Applikation lesen oder schreiben kann. Die Grösse eines Records hängt vom Datei-Typ ab. Ein *File of Byte* hat die Recordgrösse 1Byte, ein *File of LongInt* hat die Recordgrösse 4Bytes.

Achtung
bitte File-Records nicht verwechseln mit dem allgemeinen Datentyp RECORD!!

Datei Typen
Das Datei System arbeitet grundsätzlich mit typisierten Dateien, d.h. eine Datei besteht immer aus einer bestimmten Anzahl von Records, wobei jeder Record ein Datentyp darstellt.
Ein *File of integer* besteht damit aus der Anzahl ***n*** Integer bzw. Integer Records. Der jeweilige Datei-Typ wird beim Erstellen einer Datei aber nicht in die Datei hineingeschrieben. Nur die Funktion **FileOpen** bestimmt den Dateityp und damit die Art bzw. Grösse des Records in Bytes. Sämtliche Datei Operationen, die eine Recordzahl erwarten oder als Resultat zurückgeben, rechnen mit diesen logischen Records und nicht mit Bytes. Auch ein FileOfByte besteht damit aus Records, allerdings mit der Grösse 1Byte. Es können praktisch alle Standard Datentypen als Record Grösse angegeben werden, incl. einem userdefined record oder array.

File of string
bildet eine Ausnahme. Da Strings hier eine beliebige Länge haben können und nur durch ein CRLF abgeschlossen werden, kann hier nicht mit Records gearbeitet werden. Die Dateigrösse wird immer in Bytes zurückgegeben und ein „normales“ lesen und schreiben erfolgt immer auf Byte Basis. Deshalb gibt es hierzu die Spezial Funktionen **Read, ReadLn, Write, WriteLn.**
Diese Lesen z.B. (ReadLn) solange ab der aktuellen Dateiposition in den Zielstring, bis ein CRLF erreicht wird oder der Ziel String voll ist.

FileNamen
Die Filenamen folgen den DOS Konventionen, also bis zu 8 Zeichen für den Namen und bis zu 3 Zeichen für die Extension. Es sind nur Alpha-Numerische Zeichen erlaubt, keine Sonderzeichen wie z.B. **$** oder **&**.

DriveNamen
Die Laufwerksnamen folgen ebenfalls den DOS Konventionen, es werden aber nur A..D akzeptiert, je nachdem wieviele Laufwerke definiert wurden.

Directories
Hier kommt ein wesentlicher Unterschied zu DOS zum tragen. Um Resourcen (Flash, RAM, Diskspace) zu sparen und das System auch für einen 8Bitter recht schnell zu machen, wurde nur eine recht einfache Directory Struktur implementiert, es gibt nur feste Directories mit dem Namen **0..9**. Da wir hier aber keine Gigabytes an Daten zu verwalten haben, ist das vollkommen ausreichend.

Wildcards/Joker
Wie beim DOS können Teile eines Filenamens durch Wildcards **?** oder Joker ***** ersetzt werden. Das ist jedoch (wie auch bei DOS) nur mit bestimmten Operationen möglich und sinnvoll.
Abc??.* ersetzt alle Dateinamen, die mit Abc anfangen und aus 5 Zeichen bestehen. Durch das ***** an der Stelle der Extension spielt diese daher keine Rolle und wird ignoriert bzw. jede Extension wird als ok betrachtet.

Ein ***.*** an der Stelle des Dateinamens bedeutet, dass jede gefundene Datei als passend gewertet wird.

Grundsätzlich spielt die Gross/Kleinschreibung bei Filenamen, Drivenamen und Directory Namen keine Rolle. Intern wird immer mit Grossbuchstaben gearbeitet.

Defaults
Hier bestehen ein paar weitere Unterschiede zu DOS. Ein mit **DiskSelect** gewähltes Drive gilt für alle Datei Operationen, die nicht ganz speziell eine Laufwerks Angabe in ihrem FileNamen haben. Das gleiche gilt für **ChangeDir**. Wurde mit ChangeDir z.B. die Directory 4 gewählt so ist ab jetzt bei allen Datei Operationen auf **allen Drives** die Directory 4 gültig. Ausgenommen natürlich, der Dateinamen einer Operation enthält auch eine Directory Vorgabe.
Dateinamen können/müssen so aussehen:

Abcd.xyz	**.xyz*	*Abcd.**	*Abcd.x??*
oder			
A:Abcd.xyz	*B:*.xyz*	*C:Abcd.**	*D:Abcd.x??*
oder			
0:Abcd.xyz	*1:*.xyz*	*2:Abcd.**	*3:Abcd.x??*
oder			
A4:Abcd.xyz	*B5:*.xyz*	*C6:Abcd.**	*D7:Abcd.x??*

Man sieht hier, dass der Directory Namen zum Laufwerk gezählt wird, also vor dem Doppelpunkt stehen muss! Fehlt die Directory, wird immer die Default Directory genommen, die, wie o.a., für alle Laufwerke identisch ist. Es gibt also keine Laufwerk spezifische Default Directory.

Fehlt die Laufwerksbezeichnung, wird das Default Laufwerk angenommen. Fehlen beide Angaben, wird das Default Laufwerk und die Default Directory genommen.

Drive und Directory Bezeichnungen können grundsätzlich weggelassen werden.

Das Default Laufwerk wird durch **DiskSelect** und die Default Directory durch **ChangeDir** eingestellt.

Nach PowerOn und **FileSysReset** ist Drive A: und Directory 0: eingestellt.

FileHandles
Wie beim DOS gibt das System nach dem Eröffnen einer Datei mit **FileOpen** ein Handle zurück, falls diese Datei existiert. Dieses Handle ist ein Pointer auf den sog. File-Control-Block, kurz FCB genannt. Alle weiteren Datei-Operationen die Lesen, Schreiben oder in der Datei Positionieren, benötigen dieses Handle. Mit **FileClose** wird das Handle wieder an das System zurückgegeben.

FileClose
ist eine extrem wichtige Funktion. Wurde eine Datei verändert, d.h. geschrieben, hat das System in aller Regel noch zumindest einen Teil der veränderten Directory Einträge lokal im Buffer, evtl. auch noch Daten. Diese werden erst komplett weggeschrieben mit FileClose. Kommt es vor einem Close zum System Absturz oder Stromausfall oder das FileClose wird einfach vergessen, dann ist die Datei zumindest unvollständig und die Directory des Drives enthält nicht den aktuellen Stand. Aber diese Probleme sind ja auch von Windows und Unix/Linux bekannt und gefürchtet.

Das System besitzt bis zu 4 FileHandles, womit mit bis zu 4 Dateien gleichzeitig gearbeitet werden kann. Aber auch hier ist zu beachten, dass hier kein Pentium die Operationen vornimmt, sondern ein 8Bitter.

Die durch FileOpen generierten FileHandles müssen bis zur Freigabe mit FileClose von der Applikation in einer Variablen vom Typ **File** zwischengespeichert werden. Die Definition der Variablen FILE bestimmt dann auch die Record-Grösse bei den File Operationen:

```
Var  ff : file of Byte      Record = 1byte
Var  ff: file of Word       Record = 2byte
Var  ff: file of Float      Record = 4byte
Var  ff: file of myRecord   Record = sizeOf(myRecord)
Var  ff: file of myArray    Record = sizeOf(myArray)
Var  ff: file of String     Record = 1byte
                            (ReadLn, WriteLn, Read oder Write benutzen)
```

3.8.2 Allgemeines zum Arbeiten mit Dateien und dem FileSystem

Das AVRco FileSystem dient zum Abspeichern und wieder zurücklesen von Daten auf einem externen Speicher Medium. Daten können unter einem festgelegten Namen abgelegt werden und stehen jederzeit wieder zur Verfügung. Bereits gespeicherte Daten können im gewissen Rahmen auch modifiziert werden.

Folgende grundsätzliche Operationen des FileSystems werden zur Verfügung gestellt:

Formatieren eines Drives
Laufwerk wählen
Laufwerk zurücksetzen
Directory wählen
Erstellen eines Files
Löschen eines Files
Umbenennen eines Files
File Attribute vergeben
Suchen eines Files
File Attribute lesen
Lesen einer Datei
Schreiben in eine Datei

Grundsätzlich ist folgendes zu beachten:

1. In eine Datei kann erst geöffnet werden, wenn diese durch FileCreate einmal erstellt wurde.
2. Auf eine Datei kann erst mit Lesen/Schreiben zugegriffen werden, wenn diese Datei geöffnet ist.
3 Alle durch Schreiben in eine Datei gemachten Änderungen werden erst gültig nachdem diese Datei mit FileClose geschlossen wurde.
4. Jede einzelne Datei ist einem bestimmten Laufwerk und Directory zugeordnet.
5. Funktionen zum Suchen von Dateinamen und zum Löschen von Dateien können mehrdeutige (ambiguous) FileNamen verarbeiten. D.h. diese Namen können Joker und Wildcards enthalten. Alle anderen Operationen benötigen genaue und eindeutige Angaben.
6. Drive- und Directory Namen dürfen, falls abgegeben, keine WildCards oder Joker enthalten.
7. Wenn eine Datei geöffnet ist, dürfen keine weiteren andere FileSys Funktionen gestartet werden, bis diese Datei wieder geschlossen ist

Drives/Medien

Ein Laufwerk bzw. Drive oder Medium ist immer in 2 Teile aufgeteilt.

1. Directory Area. Dieser Bereich liegt auf der logischen Adresse 0 des Laufwerks. Er ist exakt ein Block gross. Die Grösse dieses Blocks und damit die Anzahl der möglichen Dateien ist von der logischen Kapazität des Drives abhängig.
2. Datenbereich. Dieser schliesst direkt an die Directory Area an und belegt den Rest der logischen Kapazität.

Logische/physische Laufwerke
Ein physisches Laufwerk kann z.B. in zwei logische Laufwerke aufgeteilt werden. Angenommen es existiert ein externer Flash Baustein mit 128kBytes Flash. Dieses sogenannte physische Laufwerk kann jetzt in zwei logische Laufwerke aufgeteilt werden. Dies geschieht mit:

```
Define  Disk_A  = 64;          // 64kB
        Disk_B  = 64;          // 64kB
```

Das FileSystem kennt jetzt 2 Drives mit je 64kB Kapazität.
Der FileIOS Treiber muss jetzt mit diesen 2 logischen Drives arbeiten. Jede Lese- und Schreib Operation muss jetzt mit einem physischen Adress Offset arbeiten, d.h. wenn Drive# = 0 ist wird mit ARG2+0 gearbeitet, wenn Drive# = 1 wird mit ARG2+$10000 gearbeitet. Damit gehen alle Zugriffe auf das Drive B in den oberen 64k Bereich des Flash Bausteins.

MultiTasking
Wie bei allen sequenziellen Treibern (USART, SPI, LAN etc) üblich, darf eine laufende FileSys Aktion nicht durch eine weitere solche unterbrochen werden. In anderen Worten: sequentielle Treiber sind grundsätzlich nicht reentrant. File Operationen sollten deshalb nur aus einem einzigen Prozess heraus gemacht werden oder es muss zumindest mit **DeviceLock** gearbeitet werden.

Bemerkung
Das AVRco FileSystem ist proprietär, d.h. es ist nicht kompatibel mit anderen Systemen und kann deshalb auch nicht z.B. von DOS oder WIN gelesen oder geschrieben werden.

3.8.3 Exportierten Typen, Konstanten und Funktionen

```
Type File;              {File of Byte, Char, word, float, string, array ...}
Type TFileAttr          = (faReadOnly, faHidden, faUser);
Type TFileAttributes    = BitSet of TFileAttr;
Type TFName             = String[15];
```

Allgemeine Funktionen des FileSystems

```
Function DiskFormat (const drive : char) : boolean;   // 'A'..'D'
Procedure FileSysReset;
Function DiskReset (const drive : char) : boolean;    // 'A'..'D'
Function GetCurDisk : char;                           // 'A'..'D'
Function DiskSelect (const drive : char) : boolean;   // 'A'..'D'
Function ChangeDir (const dir : char) : boolean;      // '0'..'9'
Function GetCurDir : char;                            // '0'..'9'
Function DiskFree (const drive : char) : word;        // 'A'..'D'
```

Verwaltungs Funktionen für Dateien

```
Function FileCreate (const fn : tFName) : boolean;
Function FileSetAttr (const fn : tFName; attr : TFileAttributes) : boolean;
Function FileGetAttr (const fn : tFName) : TFileAttributes;
Function FileRename (const fn, fnNew : tFName) : boolean;
Function FileChangeDir (const fn : tFName; const dir : char) : boolean;
Function FileExists (const fn : tFName) : boolean;
Function FileDelete (const fn : tFName) : boolean;
Function FileSize (const fn : tFName [, f : fileType|type]) : longword;
Function FileFirst (var st : tFName; const fn : tFName) : boolean;
Function FileNext (var st : tFName) : boolean;
Function FileOpen (var f : file; const fn : tFName) : boolean;
```

Funktionen für offene Dateien

```
Function FileHandleCheck (const f : file) : boolean;
Function FileReset (const f : file) : boolean;
Function FileRewrite (const f : file) : boolean;
Function FileSeek (const f : file; const p : longword) : longword;
Function FilePos (const f : file) : longword;
Function FileRead (const f : file; var Buf [; const Count: word]) : word;
Function FileWrite (const f : file; const Buf [; const Count: word]) : word;
Function FileAppend (const f : file; const Buf[; const Count: word]) : word;
Function EndOfFile (const f : file) : boolean;
Function FileClose (var f : file) : boolean;
```

Funktionen für File Of String

```
Procedure Read (const f : file; var string|char);
Procedure ReadLn (const f : file; var string|char);
Procedure Write (const f : file; const string|char);
Procedure WriteLn (const f : file; const string|char);
Procedure WriteLn (const f : file);
```

3.8.4 Implementation

Imports
Wie beim AVRco System üblich, müssen Devices importiert und definiert werden.

```
Import SysTick, FileSystem, ...;
From System Import longWord, ...;
```

Defines
Durch die folgende Define Anweisungen wird das File-System spezifiziert:

```
Define
    FileBuffer    = iData;
    FileHandles   = 2, iData;
    Disk_A        = 1024, readOnly;   {kBytes}
    SecTrk_A      = 2;                {2sect/track = 512bytes/track}
    TRKOFFS_A     = 1;                {1 reserved system track}
    Disk_B        = 2048;             {kBytes}
    SecTrk_B      = 32;               {32sect/track = 4096bytes/track}
```

FileBuffer
bezeichnet den Memory Bereich der CPU, wo die internen Buffers und Parameters angelegt werden sollen.

FileHandles
bezeichnet die mögliche Anzahl der gleichzeitig geöffneten Dateien. Die Lage der zugehörigen File Control Blocks (FCB) wird durch iData/xData bestimmt.

Disk_A ... Disk_D
bestimmen die Zahl und Grösse der vorhandenen logischen/physischen Laufwerke. Die Laufwerks Grösse wird in kBytes angegeben, wobei der kleinstmögliche Wert 32kBytes und der grösste Wert 8092kBytes ist. Es ist auch möglich z.B. ein 16Mbyte grosses Flash in zwei 8Mbyte Drives zu unterteilen. Dazu muss dann bei der Drive Angabe im FileIOS mit einem entsprechenden Offset gearbeitet werden.

Das ReadOnly Attribute bei der Laufwerks Angabe verhindert grundsätzlich ein Schreiben auf das Drive, was eigentlich nur bei Wechsel Medien sinnvoll ist.

SecTrk_A ... SecTrk_D
bestimmen die Anzahl der Sectoren/Track. Für linear adressierbare Medien, z.B. SRAM eigentlich unwichtig. Daher ist dieses Define optional. Für Block orientierte Medien, wie z.B. Floppies oder gepagte Flash ist diese Option sehr hilfreich da das Define dann so gewählt werden kann, dass ein Track exakt einer Page oder einem phys. Track entspricht. Der Default Wert ist 32.

TrkOffs_A ... TrkOffs_D
legen die evtl. notwendigen für das System bzw. die Applikation reservierten Tracks fest. Dieser Teil des Drives wird vom FileSystem selbst nie benutzt, ausgenommen beim Formatieren des Drives, wo $E5 geschrieben wird. Die Applikation kann hier System spezifische Parameter ablegen und wieder lesen, allerdings nicht über das FileSystem, sondern direkt über die Routinen, die auch für das FileIOS bereitgestellt werden müssen. Die Angabe dieses Parameters ist optional, wenn er fehlt, werden keine Spuren reserviert.

3.8.5 Disk und File Funktionen

Das System AVRco stellt eine umfangreiche Sammlung von high-level Disk und File Funktionen zur Verfügung. Diese Funktionen werden intern auf einfache Sector-read-write Funktionen reduziert. Diese Zugriffe auf das Speicher Medium (RAM, Flash, phys. Drive) muss der Anwender selbst programmieren. Dazu stellt das System eine sogenannte UserDevice Funktion mit dem Namen FileIOS zur Verfügung. Diese Routine muss der Programmierer selbst implementieren und die hiermit übergebenen Adressen und Parameter in geeigneter Weise an das Speicher Medium weiterleiten.

Das System übergibt 4 Parameter an die Funktion FileIOS und erwartet ein Boolean als Rückgabewert. Der erste Parameter (Byte) bezeichnet die auszuführende Aktion, der zweite (Byte) bezeichnet das für diese Aktion ausgewählte Drive (0 = Drive_A, 1=Drive_B, 2=Drive_C, 3=Drive_D).

Die beiden letzten Parameter sind nur bei Read/Write Operationen von Bedeutung.
Der dritte Parameter (word) bezeichnet immer den aktuellen Read/Write Buffer des Systems (iData/xData) und ist beim Lesen vom Drive das Schreib-Ziel und beim Schreiben auf das Drive die Lese Quelle. Dieser Parameter kann als 16 Bit Pointer betrachtet werden.

Der vierte Parameter (LongWord) bezeichnet immer eine Adresse auf dem Drive/Medium und ist beim Lesen vom Drive die Quelle des Transfers und beim Schreiben auf das Drive das Ziel des Transfers. Dieser Parameter kann als 32 Bit Pointer betrachtet werden, der evtl. in die Block oder Sector Grösse des Drives umgerechnet werden muss.

Bei Block bzw. Track orientierten Medien (Floppy, Flash) kann dieser Parameter ignoriert werden und statt dessen wird zur Adressierung des Mediums die aktuelle Track- und Sector Nummer benutzt, die durch das CMD 2 und 3 bereitgestellt werden.

Alle Transfer Operationen übertragen **immer** einen 128 Byte Block. Der 32 Bit Pointer setzt immer auf einen 128 Byte Block auf.

```
UserDevice FileIOS (cmd, drive: byte; arg1: word; arg2: longword) : boolean;
var res : boolean;
begin
// commands passed to user defined function "FileIOS"
// 0 driver init    drive = none     arg1 = none    arg2 = none      res = none
// 1 set drive#     drive = drive#   arg1 = none    arg2 = none      res = bool
// 2 set track#     drive = drive#   arg1 = track   arg2 = none      res = bool
// 3 set sector#    drive = drive#   arg1 = sector  arg2 = none      res = bool
// 4 read sector    drive = drive#   arg1 = dest    arg2 = source    res = bool
// 5 write sector   drive = drive#   arg1 = source  arg2 = dest      res = bool
// 6 flush buffer   drive = drive#   arg1 = none    arg2 = none      res = none
// please note: the read/write operations transfer always an 128 byte block
```

```
begin
  res:= true;
  case cmd of
    0 : |         // init hardware
    1 : |         // select drive
    2 : |         // set track, only used for block devices
    3 : |         // set sector, only used for block devices
    4 : |         // read sector = 128 byte block
    5 : |         // write sector = 128 byte block
    6 : |         // flush buffer
  endcase;
  return(result);
end;
```

CMD 0 **Hardware Init**
Dient zum initialisieren der Hardware, wie z.B. Ports und eines evtl. vorhandenen Drive Controllers.
Der Rückgabe Wert ist ohne Bedeutung.

CMD 1 **Drive Select**
Stellt das aktuelle Drive ein, auf das sich die folgenden Operationen beziehen. Die Applikation muss mit einem TRUE antworten, wenn das Medium vorhanden und funktionsfähig ist, ansonsten mit einem FALSE.

CMD 2 **Set Track**
Diese Kommando übergibt den aktuellen Track an die Applikation, die diesen Parameter aber nur benötigt, wenn mit gepagten bzw. block orientierten Medien gearbeitet wird. Der Rückgabe Wert bestimmt, ob die Operation ausgeführt werden konnte.

CMD 3 **Set Sector**
Diese Kommando übergibt den aktuellen Sector an die Applikation, die diesen Parameter aber nur benötigt, wenn mit gepagten bzw. block orientierten Medien gearbeitet wird. Der Rückgabe Wert bestimmt, ob die Operation ausgeführt werden konnte.

CMD 4 **Read Sector**
Der Treiber muss vom Drive/Medium **DRIVE** ab der Byte Position **ARG2** 128 Bytes lesen und diesen Block in den RAM Speicher schreiben, beginnend mit der Memory Adresse **ARG1**. ARG2 setzt immer auf eine 128 Byte Grenze bzw. Sector auf. Bei gepagten bzw. Block-Devices kann der Parameter ARG2 ignoriert werden und statt dessen mit dem aktuellen Track und Sector gearbeitet werden, die durch das CMD2 bzw. CMD3 übergeben wurden.
Der Rückgabe Wert bestimmt, ob die Operation ausgeführt werden konnte.

CMD 5 **Write Sector**
Der Treiber muss aus dem Speicher/RAM ab der Byte Position **ARG1** 128 Bytes lesen und diesen Block in das Speichermedium **DRIVE** schreiben, beginnend mit der Drive/Medium Adresse **ARG2**. ARG2 setzt immer auf eine 128 Byte Grenze bzw. Sector auf. Bei gepagten bzw. Block-Devices kann der Parameter ARG2 ignoriert werden und statt dessen mit dem aktuellen Track und Sector gearbeitet werden, die durch das CMD2 bzw. CMD3 übergeben wurden.
Der Rückgabe Wert bestimmt, ob die Operation ausgeführt werden konnte.

CMD 6 **Flush Buffer**
Anweisung an den Treiber, evtl. zwischengespeicherte und noch nicht in das Drive geschriebene Daten Blöcke wegzuschreiben, da ein FileClose, DiskReset oder FileSysReset Kommando abgearbeitet wurde. Dieses Kommando muss nur beachtet werden, wenn ein Schreib-Befehl des Systems von dem FileIOS nicht direkt ausgeführt wird, sondern aus Blockgrössen Gründen (z.B. grossen Flash Chips) immer mehrere Sectoren zusammengefasst werden müssen, um einen Flash Block zu bilden. Der dazu verwendete temporäre Buffer muss dann in das Medium geschrieben werden, der temporäre Buffer sollte dabei aber nicht verändert werden.

3.8.6 Exporte des FileSystems

Das FileSystem exportiert einige Typen, Variablen und Konstante, die nur für Debug Zwecke da sind und im Simulator angezeigt werden können. Diese Werte sind für die Applikation irrelevant und werden hier nicht weiter beschrieben.

Exportierte Typen des FileSystems

```
Type File; {File of Byte, Char, word, float, string, array ...}
```

Das AVRco FileSystem kennt nur typisierte Dateien, das heisst jede Datei muss mit einem bestimmten Datei-Typ geöffnet werden. Fast alle Standard Typen des Systems und alle User-definierten Typen können verwendet werden. Ein File of word enthält dann nur Words und die FileSize, FilePos etc. wird in Word-Count angegeben. Jede Lese- und Schreib Operation erfolgt auf Word-Basis. Beispiel:

```
Type tFileW : file of word;
Var ff : tFileW;
FileOpen (ff, FileName);              // opens a file of word
FileRead (ff, buffer, 2);             // reads 2 words out of ff into buffer
FileWrite (ff, buffer, 3);            // writes 2 words out of buffer into ff
Lw:= FileSize (FileName);             // returns the size of FileName in bytes
Lw:= FileSize (FileName, float);      // returns the size in counts of floats
Lw:= FileSize (FileName, tFileW);     // returns the size in counts of words
```

Eine File Definition ohne Angabe eines Typs ist identisch mit File Of Byte

```
Var ff : file;                        // the same as file of byte
```

```
Type TFileAttr        = (faReadOnly, faHidden, faUser);
Type TFileAttributes  = BitSet of TFileAttr;
```

Existierende Dateien können durch die Funktion FileSetAttr mit Attributen versehen werden. Eine neu generierte Datei besitzt keine Attribute. Das Attribut faReadOnly schützt die Datei gegen weiteres Beschreiben und Löschen. Das Attribut faHidden verhindert, dass die Datei mit den List Funktionen angezeigt werden kann. Das Attribut faUser hat für das System keine weitere Funktion und ist für die Applikation reserviert.

```
Type TFName = String[15];
```

Alle File Operationen, die einen Dateinamen brauchen, arbeiten mit dem String-Typ tFName. Das gilt auch für die DIR-List Funktionen, die einen Dateinamen zurückliefern.

Exportierte Konstanten des FileSystems

```
Const  DiskA_maxFiles  : word = nn;     {nn = total possible File count}
       DiskB_maxFiles  : word = nn;     {nn = total possible File count}
       DiskC_maxFiles  : word = nn;     {nn = total possible File count}
       DiskD_maxFiles  : word = nn;     {nn = total possible File count}

       DiskA_BlockSize : word = nn;     {nn = block size in bytes}
       DiskB_BlockSize : word = nn;     {nn = block size in bytes}
       DiskC_BlockSize : word = nn;     {nn = block size in bytes}
       DiskD_BlockSize : word = nn;     {nn = block size in bytes}
```

3.8.7 Funktionen des FileSystem

3.8.7.1 Allgemeine Funktionen des FileSystems

Grundsätzlich wird versucht, alle Fehlermöglichkeiten abzufangen. Daher werden alle Drive und Directory Angaben geprüft und ggf. mit einem FALSE zurückgewiesen. Das gleiche gilt normalerweise, wenn versucht wird mit WildCards oder Joker eine Funktion zu starten, wo dies verboten ist. Laufzeit Fehler wie z.B. FileReadOnly Write access etc. werden erkannt und mit einem FALSE beantwortet. Auch falsche oder geschlossene FileHandles werden normalerweise erkannt.

Die Laufzeit Fehler Möglichkeiten werden am Ende der Funktionsbeschreibung aufgelistet. Bei einer späteren Version dieses FileSystems kann nach einer fehlgeschlagenen Funktion auch ein Fehlercode abgefragt werden.

***Function** DiskFormat (**const** drive : char) : boolean; // 'A'..'D'*
Eine noch nie benutzte Disk/Drive muss wie bei anderen System auch üblich formatiert werden. Mit diesem Vorgang wird der Directory Teil des Drives initialisiert bzw. komplett gelöscht. DiskFormat kann bzw. muss auch benutzt werden nach einem totalen System Crash wo das Drive ganz oder teilweise überschrieben wurde. Fehlermöglichkeit 1, 5
Alle offenen Dateien müssen vorher geschlossen werden. Keine WildCards oder Joker erlaubt.

***Procedure** FileSysReset;*
Diese Prozedur schliesst alle internen Buffer, setzt alle internen Daten und Pointer zurück und baut anschliessend alle Control-Blocks wieder neu auf. Auch alle zur Verfügung stehenden Laufwerke werden zurückgesetzt. Beim nächsten Laufwerks Zugriff wird deshalb zuerst dessen Directory neu eingelesen (log-in).
Alle offenen Dateien müssen vorher geschlossen werden.

***Function** DiskReset (**const** drive : char) : boolean; // 'A'..'D'*
Identisch mit FileSysReset, ausser dass es sich nur auf das ausgewählte Laufwerk bezieht. Fehlermöglichkeit 1.
Alle offenen Dateien müssen vorher geschlossen werden. Keine WildCards oder Joker erlaubt.

***Function** GetCurDisk : char; // 'A'..'D'*
Gibt das aktuelle *DefaultDrive* zurück. Diese Funktion kann jederzeit ausgeführt werden.

***Function** DiskSelect (**const** drive : char) : boolean; // 'A'..'D'*
Stellt das *DefaultDrive* ein. Falls das Drive noch nicht eingeloggt war erfolgt jetzt ein Log-in. Fehlermöglichkeit 1.
Alle offenen Dateien müssen vorher geschlossen werden. Keine WildCards oder Joker erlaubt.

***Function** ChangeDir (**const** dir : char) : boolean; // '0'..'9'*
Stellt die *DefaultDirectory* für alle Drives ein. Fehlermöglichkeit 8.
Keine WildCards oder Joker erlaubt.

***Function** GetCurDir : char; // '0'..'9'*
Gibt das aktuelle *DefaultDirectory* zurück.

***Function** DiskFree (**const** dir : char) : word; // 'A'..'D'*
Gibt den noch freien Speicherplatz auf dem Drive zurück (in kBytes).
Ist ein Fehler aufgetreten, wird 0 zurückgegeben.

3.8.7.2 Verwaltungs Funktionen für Dateien

***Function** FileCreate (**const** fn : tFName) : boolean;*
Erstellt eine neue leere Datei. Der FileNamen kann Drive und Directory Angaben enthalten. Fehlt eine Angabe, so wird der jeweilige Default Wert genommen. Fehlermöglichkeit 1, 2, 4, 5, 8.
Alle offenen Dateien müssen vorher geschlossen werden. Keine WildCards oder Joker erlaubt.

***Function** FileSetAttr (**const** fn : tFName; attr : TFileAttributes) : boolean;*
Schreibt alle drei möglichen Attribute dieser Datei neu. Attr ist ein BitSet. Beschreibung der Attribute weiter oben unter Exportierte Typen. Fehlermöglichkeit 1, 2, 3, 5, 6, 8.
Alle offenen Dateien müssen vorher geschlossen werden. Keine WildCards oder Joker erlaubt.

***Function** FileGetAttr (**const** fn : tFName) : TFileAttributes;*
Liest alle Attribute dieser Datei. Attr ist ein BitSet. Beschreibung der Attribute weiter oben unter Exportierte Typen. Wenn die Datei nicht existiert, wird ein BitSet[] zurückgegeben.
Alle offenen Dateien müssen vorher geschlossen werden. Keine WildCards oder Joker erlaubt.

***Function** FileRename (**const** fn, fnNew : tFName) : boolean;*
Benennt eine Datei um. Die Datei *fn* muss existieren und erhält dann den neuen Namen *fnNew*. Eine Datei mit dem Namen fnNew darf aber noch nicht existieren. Der Parameter *fn* kann Drive und Directory Angaben enthalten, der Parameter *fnNew* darf das nicht.
Alle offenen Dateien müssen vorher geschlossen werden. Fehlermöglichkeit 1, 2, 3, 4, 5, 6, 8.
Keine WildCards oder Joker erlaubt.

***Function** FileChangeDir (**const** fn : tFName; **const** dir : char) : boolean;*
Die Datei erhält das neue Directory Attribut *dir* und wird damit in diese Directory „verschoben“.
Alle offenen Dateien müssen vorher geschlossen werden. Fehlermöglichkeit 1, 2, 3, 5, 6, 8.
Keine WildCards oder Joker erlaubt.

***Function** FileExists (**const** fn : tFName) : boolean;*
Stellt fest ob die Datei *fn* existiert. Der FileNamen darf wildCards und Joker enthalten. Evtl. vorhandene Drive oder Directory Angaben aber nicht. Wenn Drive und/oder Directory angegeben sind, wird da gesucht, ansonsten werden DefaultDrive und/oder DefaultDir dazu herangezogen. Da der Datei Namen WildCards oder Joker enthalten kann, können mehrere Dateien gefunden werden, auf die der File Namen passt. Wieviele das sind, kann nicht fest gestellt werden. Mit ***.*** z.B. gibt das Resultat ein true, wenn mindestens eine Datei in dem gewählten Drive/Directory existiert.
Alle offenen Dateien müssen vorher geschlossen werden. Fehlermöglichkeit 1, 2, 3.
WildCards oder Joker sind erlaubt.

***Function** FileDelete (**const** fn : tFName) : boolean;*
Löscht die Datei *fn*. Der FileNamen darf wildCards und Joker enthalten. Evtl. vorhandene Drive oder Directory Angaben aber nicht. Wenn Drive und/oder Directory angegeben sind, wird da gesucht und gelöscht, ansonsten werden DefaultDrive und/oder DefaultDir dazu herangezogen. Da der Datei Namen WildCards oder Koker enthalten kann, können mehrere Dateien gefunden und gelöscht werden, auf die der File Namen passt. Wieviele das sind, kann nicht fest gestellt werden. Mit ***.*** werden alle passende Dateien des Drives/Directory gelöscht.
Alle offenen Dateien müssen vorher geschlossen werden. Fehlermöglichkeit 1, 2, 3, 5, 6.
WildCards oder Joker sind erlaubt.

***Function** FileSize (**const** fn : tFName [, f : fileType|type]) : longword;*
Errechnet die Dateigrösse des Files *fn* in Records. Wird kein zweiter Parameter angegeben, ist das Ergebnis die Grösse in Bytes. Wird als zweiter Parameter ein File-Typ angegeben, stellt das Ergebnis die Anzahl der Records für diesen File-Typ dar. Das gleiche gilt auch, wenn der zweite Parameter ein Standard Typ ist, z.B. Word oder Integer. Im Fehlerfall wird die Size 0 zurückgegeben.
Alle offenen Dateien müssen vorher geschlossen werden. Fehlermöglichkeit 1, 2, 3, 8.
Keine WildCards oder Joker erlaubt.

Function *FileFirst (**var** st : tFName; **const** fn : tFName) : boolean;*
Ist die Eröffnungs Funktion um alle Dateienamen eines bestimmten Drives und Directories zu erhalten. Der FileNamen darf WildCards und Joker enthalten. Die optionale Drive und Directory Angabe allerdings nicht. Files mit dem faHidden Attribute werden nicht angezeigt. Wurde eine Datei gefunden, enthält der Parameter *st* den gefundenen Namen und das Ergebnis ist TRUE. Bei einem FALSE wird auch der String *st* zu einem Leerstring gemacht. Da diese Funktion immer nur die erste gefundene Datei anzeigt, müssen weitere Dateien mit der untenstehenden Funktion *FileNext* in einer Schleife angezeigt werden.
Alle offenen Dateien müssen vorher geschlossen werden. Fehlermöglichkeit 1, 2, 3.
WildCards oder Joker sind erlaubt.

Function *FileNext (**var** st : tFName) : boolean;*
Diese Funktion folgt einer *FileFirst* Funktion nach. Wurde eine Datei gefunden, enthält der Parameter *st* den gefundenen Namen und das Ergebnis ist TRUE. Bei einem FALSE wird auch der String *st* zu einem Leerstring gemacht.
Alle offenen Dateien müssen vorher geschlossen werden. Fehlermöglichkeit 1, 2, 3.

Beispiel für eine komplette Datei Anzeige eines Drives und einer Directory:

```
if FileFirst (st, 'B1:??.*.*') then
// display the first dir entry in st
  while FileNext (st) do
    // display the next dir entry in st
  endwhile;
endif;
```

Es wird hierbei die Directory **1** im Drive **B** untersucht und alle darin enthaltenen Dateien angezeigt.

Function *FileOpen (**var** f : file; **const** fn : tFName) : boolean;*
Öffnet eine vorhandene Datei zum Lesen oder Schreiben. Der Parameter *fn* muss eindeutig sein, d.h. er darf keine Joker oder WildCards enthalten. War die Funktion erfolgreich, enthält der Parameter *f* jetzt das generierte und gültige FileHandle mit dem aller weiteren Datei Operationen für dieses File gemacht werden müssen. Der Lese-Schreib Pointer zeigt auf die Position 0, d.h. auf den ersten Record.
Die Record Grösse in Bytes wird durch den Datei-Typ *f* bestimmt. Fehlermöglichkeit 1, 2, 3, 8.
WildCards oder Joker sind nicht erlaubt.

3.8.7.3 Funktionen für offene Dateien

Die Lese und Schreib Operationen dieses FileSystems sind in erster Linie ***sequentiell***. Das heisst, normalerweise wird fortlaufend gelesen oder geschrieben. Beginnend vom Date Anfang. Für das Lesen kann auch ein Random Read verwendet werden, indem auf eine bestimmten Record positioniert wird, dieser Record wird dann gelesen und eine erneute Positionierung mit Lesen oder ein sequentielles Lesen kann erfolgen.

Random write
ist nur rudimentär implementiert und auf Bytes Schreiben beschränkt (File of Byte). Um auch hier unliebsame Überraschungen zu vermeiden, muss vor jedem Byte Schreiben eine neue Positionierung erfolgen! Random write ist damit eigentlich nicht zu empfehlen.

Function *FileHandleCheck (**const** f : file) : boolean**;***
Prüft eine FileHandle Variable auf Gültigkeit, d.h. ob diese Datei schon geöffnet ist.
Fehlermöglichkeit 7.

Function *FileReset (**const** f : file) : boolean**;***
Setzt den ReadWrite Pointer für diese Datei an den Datei Anfang.
Fehlermöglichkeit 2, 7.

Function** FileRewrite (**const** f : file) : boolean**;
Schliesst die geöffnete Datei, löscht diese und eröffnet eine neue, nun leere Datei auf dem gleichen Drive und Directory mit dem gleichen Dateinamen wie die zuvor gelöschte Datei.
Fehlermöglichkeit 2, 5, 6, 7.

***Function** FileSeek (**const** f : file; **const** p : longword) : longword;*
Stellt den Lese/Schreibpointer für diese Datei an die Record Position *p*. Als Ergebnis kommt die aktuelle neue Record Position zurück. Zeigt die gewünschte Position jedoch über das Datei Ende hinaus, wird auf den letzten Record dieses Files positioniert und die Position auch als Resultat zurückgeliefert. Diese Funktion dient als Vorbereitung einer Random Read oder Write Operation.
Ist die Funktion fehlgeschlagen, wird das Ergebnis immer 0.
Fehlermöglichkeit 2, 7.

Function** FilePos (**const** f : file) : longword**;
Gibt die aktuelle Position des Schreib/Lesepointers der Datei zurück. Ist die Funktion fehlgeschlagen, wird das Ergebnis immer 0.
Fehlermöglichkeit 7.

***Function** FileRead (**const** f : file; **var** Buf [; **const** Count: word]) : word;*
Liest ab dem aktuellen ReadWrite Pointer die Anzahl *Count* Records in die Variable *Buf*
Der Lese/Schreib Pointer wird dabei jeweils um einen Record inkrementiert, so dass die nächste Leseoperation automatisch den nächsten Record zurück gibt. Das ist das sequentielle Read. Bei Random Read wird mit der Funktion *FileSeek* nach jedem Lesevorgang neu positioniert. Der Parameter Count ist optional. Wenn er fehlt, wird ein Record gelesen.
Fehlermöglichkeit 2, 7, 9.

***Function** FileWrite (**const** f : file; **const** Buf [; **const** Count: word]) : word;*
Schreibt ab dem aktuellen ReadWrite Pointer die Anzahl *Count* Records aus der Variablen *Buf* in das File. Der Lese/Schreib Pointer wird dabei jeweils um einen Record inkrementiert, so dass die nächste Schreib Operation automatisch den nächsten Record beschreibt. Das ist das sequentielle Write. Bei Random Write wird mit der Funktion *FileSeek* nach jedem Schreibvorgang neu positioniert. Random Write ist nur rudimentär implementiert und sollte nicht verwendet werden. Der Parameter Count ist optional. Wenn er fehlt, wird ein Record geschrieben.
Fehlermöglichkeit 2, 5, 6, 7.

***Function** FileAppend (**const** f : file; **const** Buf[; **const** Count: word]) : word;*
Diese Funktion gehört zum sequentiellen Schreiben. Hier wird der Filepointer ans Datei Ende Positioniert und die Anzahl *Count* Records aus der Variable *Buf* an die Datei angehängt. Jetzt kann mit weiteren *FileWrite* Operationen einfach sequentiell weiter geschrieben werden. Der Parameter Count ist optional. Wenn er fehlt, wird ein Record geschrieben.
Fehlermöglichkeit 2, 5, 6, 7.

***Function** EndOfFile (**const** f : file) : boolean;*
Stellt fest ob der Lese/Schreib Pointer am Dateiende angelangt ist oder ob noch weiter sequentiell gelesen werden kann.
Fehlermöglichkeit keine.

***Function** FileClose (**var** f : file) : boolean;*
Schliesst die geöffnete Datei. Alle Buffer werden geleert, die Directory des Drives wird upgedated und, falls noch nicht schon getan, der letzte geänderte Record bzw. Sector wird in das Drive geschrieben. Die Variable *f* wird ungültig gemacht, d.h. sie wird auf $0000 gesetzt.
Fehlermöglichkeit 7.

3.8.7.4 Funktionen für File of String

File of String bedürfen der besonderen Behandlung. Strings werden hier ohne Längenbyte gelesen und geschrieben. Damit das Ende eines Strings eindeutig indentifiziert werden kann, wird jeder String mit einem CRLF ($0D $0A) abgeschlossen. Da ein String beliebig lang sein kann, kann hier nicht mit Records gearbeitet werden und die File Positionierungen und FileSize Funktionen haben hier eigentlich keine grosse Aussage.
Auch Random Read und Write macht hier keinen Sinn. File of String sind reine sequentielle Dateien.

Um aber trotzdem eine relativ einfache Handhabung sicher zu stellen, wurden die schon bekannten System Funktionen Read, Write, ReadLn und WriteLn auf Files erweitert.
Da hierbei das Datei Ende nicht durch die Read bzw. ReadLn Funktion selbst festgestellt werden kann, muss hier immer im Wechsel mit der Funktion ***EndOfFile*** gearbeitet werden.

Procedure *Read (**const** f : file; **var** string|char);*
Liest ein Zeichen aus dem File in das Ziel, das eine Char oder String Variable sein kann. Ein String bekommt dann die Länge 1. Es werden auch die Limiter CR und LF gelesen und übertragen.

Procedure *ReadLn (**const** f : file; **var** string|char);*
Liest einen String aus dem File in das Ziel, das eine Char oder String Variable sein kann. Bei einem Char als Ziel bricht die Funktion natürlich sofort nach einem Zeichen ab und macht deshalb normalerweise wenig Sinn. Ein String wird gefüllt bis entweder der String voll ist oder ein CRLF erkannt wird. War der String schon vorher voll, bleibt der Lesepointer an dieser Stelle stehen. Das Längenbyte des Ziel Strings wird auf die richtige Länge gesetzt. Die Limiter CR und LF werden zwar gelesen aber niemals in den String übertragen.

Procedure *Write (**const** f : file; **const** string|char);*
Schreibt ein Zeichen aus der Quelle, das ein Char oder String sein kann in die Datei. Es wird kein Limiter CRLF angehängt.

Procedure *WriteLn (**const** f : file; **const** string|char);*
Schreibt einen String aus der Quelle, das ein Char oder String sein kann, in die Datei. Es wird der Limiter CRLF angehängt.

Procedure *WriteLn (**const** f : file);*
Schreibt einen Leerstring, nämlich nur ein CRLF in die Datei.

```
FileOpen (fs, 'Strings.Tst');     // open the existing file
WriteLn (fs, 'Monday');           // write first string at file start
WriteLn (fs, 'Tuesday');          // write next strings
WriteLn (fs, 'Wednesday');
WriteLn (fs, 'Thursday');
WriteLn (fs, 'Friday');
WriteLn (fs, 'Saturday');
Write (fs,   'Sunday ');
Write (fs,   'is weekend');
WriteLn (fs);                     // write an empty string
FileClose (fs);

FileOpen (fs, 'Strings.Tst');
FileRead (fs, buf, 1);            // read first char of first string
ReadLn (fs, st);                  // read first string
Read (fs, ch);                    // read first char of next string
Read (fs, st, 4);                 // read 4 chars into string
ReadLn (fs, st);                  // read rest of string into string
while not EndOfFile(fs) do        // read the entire file
  ReadLn (fs, st);                // read the next string
endwhile;                         // until end of file
FileClose (fs);
```

Fehler Möglichkeiten

Fehler 1: Drive doesn't exist.
Fehler 2: Drive not ready.
Fehler 3: File doesn't exist.
Fehler 4: File already exists.
Fehler 5: Drive readonly.
Fehler 6: File readonly.
Fehler 7: FileHandle error.
Fehler 8: Joker or WildCard found.
Fehler 9: End of File.

Simulator

Der AVRco Simulator SIM32 unterstützt das FileSystem komplett. Die Drives werden exakt im PC-Memory nachgebildet und gelesen und beschrieben. Damit ist ein hervorragende Testmöglichkeit gegeben, die Applikation betreffend dem FileSystem ausgiebig zu prüfen, ohne dass eine funktionierende Hardware zu Verfügung steht.

Programm Beispiele:

Im Verzeichnis **..\E-LAB\AVRco\Demos\FileSys** befindet sich allgemeines Testprogramm mit dem alle Funktionen des File Systems geprüft werden. Bitte beachten, dass hier auch illegale Operationen geprüft werden.

Im Verzeichnis **..\E-LAB\AVRco\Demos\FileFlash** befindet sich eine echte Anwendung,
die mit einem 16Mbit Atmel Dataflash AT45DB161 arbeitet.

3.9 FAT16 File System (FAT16_32)

Allgemeines

Erfassung, Verwaltung, Speicherung und Abruf von Daten sind Aufgaben, die oft auch von Mikroprozessor Systemen erledigt werden müssen. Zum Glück für die meisten Programmierer hält sich das Datenaufkommen hierbei in Grenzen. Meistens sind das wenige Kilobytes, die problemlos selbst in Arrays, Records etc. im gepufferten Speicher oder im EEprom gehalten und verwaltet werden können.

Werden die Datenmengen grösser und handelt es sich dabei noch um unterschiedliche Datenstrukturen wird die Handhabung wesentlich schwieriger oder gar unmöglich. Hier setzt ein Datei System an. Die Dateien (Datenblöcke) können fast beliebig gross werden und die Datenstruktur der Dateien ist beliebig. Man darf jetzt allerdings nicht dem Irrtum verfallen, alles und jegliches mit einem Filesystem verwirklichen zu müssen. Ein Datei System ist immer ein Verwaltungs Overhead und kostet nicht unerheblich Programmcode, Arbeitspeicher und Rechenzeit. Die Anforderungen an die System Resourcen sind nicht zu unterschätzen.

Im Gegensatz zum ***Standard*** AVRco FileSystem ist das AVRco FAT16 FileSystem voll PC kompatibel. Daher ist der Resourcen Verbrauch (RAM, ROM etc) grösser, aber mit dem Vorteil der Portabilität. Das heisst, dass die verwendeten Datenträger (FlashCards, MicroDrive oder IDE Drives) sowohl vom PC als auch vom AVRco System gelesen und geschrieben werden können. *Ein minimal System ist schon in 12kB Code und ca. 1kB RAM zu erstellen.* Da die Auflösung einer Directory Struktur den meisten Speicher benötigt, kann der Treiber per Define zwischen 1 und 10 Directory Levels eingestellt werden. Jeder Level benötigt hierbei zusätzlich ca. 30Bytes an RAM. Man sollte sich deshalb auf 1 Ebene beschränken. Die Pfad/Directory Namen müssen der DOS Konvention entsprechen und sind max. 8 Zeichen lang. Die Dateinamen entsprechen ebenfalls der DOS Konvention mit max. 8 Zeichen Länge und max. 3 Zeichen Extension. Aus Resourcen Gründen wird nur eine Disk bzw. Laufwerk unterstützt.

Das System unterstützt direkt die FlashCard Typen **MMC/SD/miniSD/microSD** über SPI Mode0.
Für spezielle Drives oder abweichende Hardware ist auch ein allgemeiner Treiber **FAT16_IOS** vorhanden.

Es wurden alle geläufigen Filesystem, Drive und File Operationen implementiert. Das Filesystem arbeitet auf einen der Spezial Treiber oder den Hardwaretreiber **FAT16_IOS**.

Wenn FAT16_IOS gewählt wurde so werden dem Treiber bestimmte Kommandos und Parameter übergeben der diese in Zusammenhang mit der Disk Hardware (externes SRAM, EEPROM, FLASH, Flashcard, Harddisk etc) ausführen muss. Der Treiber FAT16_IOS muss vom Anwender in Abhängigkeit von seinem Speichermedium und Hardware selbst erstellt werden.

Wurde einer der eingebauten Treiber ausgewählt, so wird der Treiber FAT16_IOS nicht benötigt, da diese Treiber vollkommen autark arbeiten.

Technische Daten

Laufwerke logisch/physisch	: 1	
DiskSize	: 1Mbyte ... 2Gbyte	(min. 32Mbyte bei portablen Medien)
Directory Levels	: 1 ... 10	definierbar
Max. Files	: min. 512	
Max. open Files	: 1 … 4	definierbar
FileSize	: 0 ... DiskSize	

Alternativ steht auch ein FAT32 System zur verfügung. Im Gegensatz zum reinen FAT16 System, das den Treiber **uFAT16** importiert, muss für FAT32 der Treiber **uFAT16_32** importiert werden. Hierbei andert sich:
DiskSize 1Mbyte..32GByte
FileSize max 4GByte

Diverse Funktionen ändern sich beim FAT32 minimal. Der Einfachheit wegen haben alle Funktionen in beiden Systemen den gleichen Namen. Es kommen auch zwei neue hinzu. Das FAT32 System unterstützt natürlich auch mit FAT16 formatierte Disks. Der Code, RAM und Rechenleistungs Verbrauch ist bei FAT32 natürlich grösser. Dafür laufen die meisten Funktionen mindestens **doppelt so schnell** wie beim FAT16.

Der Einsatz des **Optimiser** wird empfohlen wegen dem Speed und der Code Grösse.

Um hohe Geschwindigkeit der Operationen zu erreichen, die **clustersize** sollte beim formatieren unter WIN **32kB** sein, besonders mit Disksizes >= 4GB.

Bitte beachten: ein CheckDisk (ausgeführt beim startup oder Disk change) braucht nun wesentlich länger als zuvor, da jetzt alle durch WIN vorgegebenen Konventionen eingehalten und ausgeführt werden.

Einführung
Der enorme Vorteil der Kompatibilität und Portabilität muss durch einen wesentlich höheren Code und Speicherbedarf erkauft werden. Durch gewisse Beschränkungen hält sich der Aufwand jedoch im erträglichen Rahmen.

FAT16
Unter FAT16 versteht man das ursprünglich für DOS entwickelte Harddisk Filesystem, das dann später unter Windows 3.x auf lange Datei- und Pfadnamen erweitert wurde (**FAT32**). Daher können DOS Namen und Windows Namen gleichzeitig auf einer solchen Disk existieren. Die Datei und Pfad Einträge wurden so gestaltet, dass ein DOS Filesystem Dateien mit langen Namen lesen kann und umgekehrt ein Windows System kann auch reine DOS Dateien bearbeiten. Eine Datei mit langen Dateinamen hat auch immer parallel dazu einen Eintrag mit den DOS Konventionen, also 8 Zeichen für „Namen“ und 3 Zeichen für „Extension“.

Kurze Dateinamen
Schreibt Windows z.B. ein File mit dem Namen MyFile.txt auf die Disk, steht im Directory Eintrag dann (vereinfacht) MyFile.txt als Long_Entry und MYFILE.TXT als DOS_Entry. So können beide Systeme auf diese Datei zugreifen.

Lange Dateinamen
Schreibt Windows z.B. ein File mit dem Namen MyFileLong.txt auf die Disk, steht im Directory Eintrag dann (vereinfacht) MyFileLong.txt als Long_Entry und MYFILE~1.TXT als DOS_Entry. So können wiederum beide Systeme auf diese Datei zugreifen.

Das gleiche gilt auch für lange Extensions
Schreibt Windows z.B. ein File mit dem MyFile.text auf die Disk, steht im Directory Eintrag dann (vereinfacht) MyFile.text als Long_Entry und MYFILE~1.TEX als DOS_Entry. So können wiederum beide Systeme auf diese Datei zugreifen. Das gilt natürlich auch für die Pfadnamen.

Wenn nun ein System solche Dateien liest, genügt es, wenn auf den DOS Teil des Eintrags zugegriffen wird. Soll jedoch ein solcher Eintrag geändert werden, z.B. Namen ändern, Eintrag/File löschen, Daten an die Datei anhängen etc. so müssen beide Teile bearbeitet werden, der DOS-Eintrag **und** der Long-Eintrag eines FAT32 Mediums.

Das Lesen, Schreiben oder Löschen von Long-Entries bedeutet jedoch einen erheblichen Programm Aufwand ganz abgesehen von einem riesigen Speicherbedarf für die dafür notwendigen Strings. File- und Pfadnamen können theoretisch unendlich lang sein. Auch mit freiwilliger Längenbeschränkung desjenigen, der die Dateien und Pfade erstellt, sprengt die komplette Handhabung von Long-Namen sehr schnell die Möglichkeiten eines 8 Bit Controllers.

Das AVRco System arbeitet deshalb im DOS Mode mit der Erweiterung, dass auch Long-Entries gelesen werden können. DOS Mode bedeutet auch, dass Pfad- und Datei Namen nicht länger als 8 Zeichen sein dürfen und eine Extension nicht länger als 3 Zeichen. Wie weiter oben zu sehen ist, handhabt Windows dies automatisch, indem es immer auch einen DOS kompatiblen Eintrag erzeugt. Bei langen Namen werden diese immer gekürzt auf 8+3 und gekennzeichnet durch **~x** in den letzten beiden Zeichen des Namens. Das „x“ besteht aus einer Ziffer, die Windows vorgibt.

Die Beschränkung auf den DOS Mode hat in der Praxis keine gravierenden Auswirkungen. Entweder es werden durch Windows erstellte Dateien gelesen und verarbeitet oder das AVRco System erstellt Dateien und beschreibt diese. Durch das DOS System wird der Treiber relativ klein und schnell und verbraucht wesentlich weniger RAM.

Ein weiterer Speicherfresser stellen die möglichen Directory Ebenen und die Zahl der geöffneten Dateien dar. Der statische Speicherbedarf (Grösse bestimmter Variablen) wächst mit jeder möglichen Ebene und mit jedem möglichen File. Im gleichen Masse wächst auch die notwendige Frame Grösse. Deshalb müssen diese beiden Parameter durch Defines festgelegt werden.

Bitte beachten dass je mehr eine Disk gefüllt ist, desto länger dauern auch Lese- und Schreibzugriffe.

Die AVRco IDE **PED32** enthält eine **Disk Formatierungs Software**. Dieses Tool erlaubt es einen Datenträger neu zu formatieren. Damit ist es möglich das auf den Flashkarten meistens schon vorhandene FAT32 Datei System durch ein reines FAT16 zu ersetzen. Dadurch wird auch Windows gezwungen nur FAT16 Files/Directories anzulegen und zu benutzen. Damit ist es möglich auch von Windows erstellte Dateien zu verändern und zu löschen. Beim FAT32 Treiber spielt die Art der Formatierung keine Rolle.

3.9.1 Definitionen

Record
Ein Record bezeichnet die kleinste Lesbare Datenmenge, die die Applikation lesen oder schreiben kann. Die Grösse eines Records hängt vom Datei-Typ ab. Ein *File of Byte* hat die Recordgrösse 1Byte, ein *File of LongInt* hat die Recordgrösse 4Bytes.
Achtung:
bitte File-Records nicht verwechseln mit dem allgemeinen Datentyp RECORD!!

Datei Typen
Das Datei System arbeitet grundsätzlich mit typisierten Dateien, d.h. eine Datei besteht immer aus einer bestimmten Anzahl von Records, wobei jeder Record ein Datentyp darstellt.
Ein *File of integer* besteht damit aus der Anzahl ***n*** Integer bzw. Integer Records. Der jeweilige Datei-Typ wird beim Erstellen einer Datei aber nicht in die Datei hineingeschrieben. Nur die Funktion **FileAssign** bestimmt den Dateityp und damit die Art bzw. Grösse des Records in Bytes. Sämtliche Datei Operationen, die eine Recordzahl erwarten oder als Resultat zurückgeben, rechnen mit diesen logischen Records und nicht mit Bytes. Auch ein FileOfByte besteht damit aus Records, allerdings mit der Grösse 1Byte. Es können praktisch alle Standard Datentypen als Record Grösse angegeben werden, incl. einem userdefined record oder array.
Achtung: **FileType vars müssen global definiert werden**. Function-locale sind nicht zulässig.

File of Text
bildet eine Ausnahme. Da Strings hier eine beliebige Länge haben können und nur durch ein CRLF abgeschlossen werden, kann hier nicht mit Records gearbeitet werden. Die Dateigrösse wird immer in Bytes zurückgegeben und ein „normales“ lesen und schreiben erfolgt immer auf Byte Basis. Deshalb gibt es hierzu die Spezial Funktionen **Read, ReadLn, Write, WriteLn.**
Diese Lesen z.B. (ReadLn) solange ab der aktuellen Dateiposition in den Zielstring, bis ein CRLF erreicht wird oder der Ziel String voll ist. Textfiles können nur sequentiell gelesen und geschrieben werden.

FileNamen
Die Filenamen folgen den DOS Konventionen, also bis zu 8 Zeichen für den Namen und bis zu 3 Zeichen für die Extension. Es sind nur Alpha-Numerische Zeichen erlaubt, keine Sonderzeichen wie z.B. **$** oder **&**.

Directories
Die Anzahl der Directory Levels (Hierarchie) wird durch ein Define bestimmt. Die Directory Namen sind max. 8 Zeichen lang. Es sind nur Alpha-Numerische Zeichen erlaubt, keine Sonderzeichen wie z.B. **$** oder **&**. Directory Namen und auch komplette Pfad Namen dürfen keine Wildcards oder Joker enthalten.

Wildcards/Joker
Wie beim DOS können Teile eines Filenamens durch Wildcards **?** oder Joker ***** ersetzt werden. Das ist jedoch (wie auch bei DOS) nur mit bestimmten Operationen möglich und sinnvoll.
Abc??.* ersetzt alle Dateinamen, die mit Abc anfangen und aus 5 Zeichen bestehen. Durch das ***** an der Stelle der Extension spielt diese daher keine Rolle und wird ignoriert bzw. jede Extension wird als ok betrachtet.

Ein ***.**** an der Stelle des Dateinamens bedeutet, dass jede gefundene Datei als passend gewertet wird.

Grundsätzlich spielt die Gross/Kleinschreibung bei Filenamen und Pfad Namen keine Rolle.

Pfad und File Namen
Beim PC kann man bei programmierten Datei Zugriffen und ebenso bei allen Tools mit gemischten Pfad und Dateinamen arbeiten. Zum Beispiel: \ddd1\ddd2\ddd3\name.ext

System intern wird dann dieser String in Pfad, Filenamen und Extension zerlegt. Beim AVRco FAT16 Filesystem wäre das im Prinzip zwar auch möglich, aber das würde einen komplexen String Process bedingen, der alle Variationen beherrschen muss. Dies ist allerdings mit erheblichen Code und RAM Aufwand zu bezahlen. Daher arbeiten alle System Funktionen die solche Strings erwarten, mit getrennten Pfad und FileNamen.:

Function *F16_FileExist (Path : TPathStr; FName : TFileName; aAttr : tFAttr) : boolean;*

Das erspart enorm Rechenzeit und Systemresourcen und ist eigentlich keine sonderliche Einschränkung.

Defaults
Hier bestehen ein paar weitere Unterschiede zu DOS. Ein mit **F16_ChangeDir** gewählte Directory gilt für alle Datei Operationen, die nicht ganz speziell eine Pfad Angabe in ihrem PfadNamen haben. Wurde mit F16_ChangeDir z.B. die Directory *dddd* gewählt so ist ab jetzt bei allen Datei Operationen die Directory *dddd* gültig. Ausgenommen natürlich, der Pfadnamen einer Operation enthält auch eine Pfad Vorgabe.

Dateinamen können/müssen so aussehen:

Abcd.xyz **.xyz* *Abcd.** *Abcd.x??* *Ab??.x??*

Directory/Pfad Namen können/müssen so aussehen:

\ *\ppp* *..\ppp* *ppp* *ppp\ddd*

Fehlt der Pfad, wird immer der zuletzt eingestellte Pfad genommen. Mit ..\ wird ein relativer Pfad angegeben, relativ zum aktuellen Pfad.

Directory/Pfad Bezeichnungen können grundsätzlich weggelassen werden.

Die Default (current) Directory wird durch **F16_ChangeDir** eingestellt.

Nach PowerOn, **F16_DiskInit** oder **F16_DiskReset** ist Directory \ (root) eingestellt.

FileHandle
Das File System arbeitet mit FileHandles. Ein FileHandle ist eine Variable vom Typ **File**. Eine solche Variable wird zum FileHandle wenn sie von der Funktion FileAssign verarbeitet wurde.

Wie beim DOS gibt das System nach dem Eröffnen einer Datei mit **F16_FileAssign** ein Handle zurück. Dabei muss diese Datei nicht unbedingt vorhanden sein. Dieses Handle ist ein Pointer auf den sog. File-Control-Block, kurz FCB genannt. Alle weiteren Datei-Operationen die Lesen, Schreiben oder in der Datei Positionieren, benötigen dieses Handle. Mit **F16_FileClose** wird das Handle wieder an das System zurückgegeben und die Datei geschlossen.

F16_FileClose ist eine extrem wichtige Funktion. Wurde eine Datei verändert, d.h. geschrieben, hat das System in aller Regel noch zumindest einen Teil der veränderten Directory Einträge lokal im Buffer, evtl. auch noch Daten. Diese werden erst komplett weggeschrieben mit F16_FileClose.

Kommt es vor einem Close zum System Absturz oder Stromausfall oder das FileClose wird einfach vergessen, dann ist die Datei zumindest unvollständig und die Directory des Drives enthält nicht den aktuellen Stand. Aber diese Probleme sind ja auch von Windows und Unix/Linux bekannt und gefürchtet.

Procedure *F16_FlushBufSec;*
Stellt im Notfall (power down etc) sicher dass die Filebuffer noch weggeschrieben werden.

Das System besitzt bis zu 4 FileHandles, womit mit bis zu 4 Dateien gleichzeitig gearbeitet werden kann. Aber auch hier ist zu beachten, dass hier kein Pentium die Operationen vornimmt, sondern ein 8Bitter.

Die durch FileAssign generierten FileHandles müssen bis zur Freigabe mit FileClose von der Applikation in einer Variablen vom Typ **File** zwischengespeichert werden. Die Definition der Variablen FILE bestimmt dann auch die Record-Grösse bei den File Operationen:

```
Var ff : file of Byte;          Record = 1byte
Var ff : file of Word;          Record = 2byte
Var ff : file of Float;         Record = 4byte
Var ff : file of myRecord;      Record = sizeOf(myRecord)
Var ff : file of myArray;       Record = sizeOf(myArray)
Var ff : file of String[nn];    Record = nn+1byte
Var ff : file of Text;          (ReadLn, WriteLn, Read oder Write benutzen)
```

Funktionen die ein FileHandle und einen Laufzeit Fehler zurückgeben, geben im allgemeinen das benutzte Handle nicht an das System zurück, ausgenommen F16_FileClose. Deshalb muss im Fehlerfall immer ein F16_FileClose aufgerufen werden.

3.9.2 Allgemeines zum Arbeiten mit Dateien und dem FileSystem

Das AVRco FAT16 FileSystem dient zum Abspeichern und wieder zurücklesen von Daten auf einem externen Speicher Medium. Daten können unter einem festgelegten Namen abgelegt werden und stehen jederzeit wieder zur Verfügung. Bereits gespeicherte Daten können im gewissen Rahmen auch modifiziert werden.

Folgende grundsätzliche Operationen des FileSystems werden zur Verfügung gestellt:

Laufwerk zurücksetzen
Directory wählen
Erstellen eines Files
Erstellen einer Directory
Löschen eines Files
Löschen einer Directory
Umbenennen eines Files
File Attribute vergeben
Suchen eines Files
File Attribute lesen
Lesen einer Datei
Schreiben in eine Datei

Grundsätzlich ist folgendes zu beachten:

1. Auf eine Datei kann erst mit Lesen/Schreiben zugegriffen werden, wenn diese Datei geöffnet ist.
2. Alle durch Schreiben in eine Datei gemachten Änderungen werden erst gültig nachdem diese Datei mit FileClose geschlossen wurde.
3. Jede einzelne Datei ist einer bestimmten Directory zugeordnet.
4. Funktionen zum Suchen von Dateinamen und zum Löschen von Dateien können mehrdeutige (ambiguous) FileNamen verarbeiten. D.h. diese Namen können Joker und Wildcards enthalten. Alle anderen Operationen benötigen genaue und eindeutige Angaben.
5. Pfad/Directory Namen dürfen, falls abgegeben, keine WildCards oder Joker enthalten.
6. Wenn eine Datei geöffnet ist, dürfen keine weiteren andere FileSys Funktionen (Funktionen, die keinen File erfordern) gestartet werden, bis diese Datei wieder geschlossen ist.

MultiTasking

Wie bei allen sequenziellen Treibern (UART, SPI, LAN etcetc) üblich, darf eine laufende FileSys Aktion nicht durch eine weitere solche unterbrochen werden. In anderen Worten: sequentielle Treiber sind grundsätzlich nicht reentrant. File Operationen sollten deshalb nur aus einem einzigen Prozess heraus gemacht werden oder es muss zumindest mit **DeviceLock** gearbeitet werden.

Disk Format
Eine low level Disk Formatierungs Funktion ist nicht enthalten, da hier eine Vielzahl DOS und Windows Daten geschrieben werden müssen, die das System nicht wissen kann, da diese auch extrem vom Medium und der Disk Grösse abhängig sind.

Die implementierte Format Routine löscht nur alle File und Directory Einträge.

3.9.3 Exportierten Typen, Konstanten und Funktionen

```
Type File; {File of Byte, Char, word, float, string, text, array, record ...) = FileHandle
Type tFAttrEnum     = (faReadOnly, faHidden, faSysFile, faVolumeID, faDirectory, faArchive);
Type tFAttr         = BitSet of tFAttrEnum;
Type tDiskError     = (deNone, deMediaUnknown, deFATunknown, deInitFail, deHandleFail,
                       deReadFail, deWriteFail, deReadOnly, deFileExists, deNotFound);
Type TFileAccess    = (faNone, faAssign, faRead, faWrite, faAppend, faRandomWr);
Type TFileName      = string[12];
Type TpathStr       = string[nn];  // nn depends on max. DirLevels allowed
Type TF16TimeStr    = string[5];
Type TF16DateStr    = string[8];
Type TarrNameExt    = array[1..11] of Char;
Type  tFATtype      = (tFATnone, tFAT16, tFAT32);

Type TDir  = record
               NameExt     : TArrNameExt; // Filename or directory
               FAttr       : tFAttr;      // The file attributes
               NTRes       : Byte;        // NT reserved byte
               CrtTim10    : byte;        // Create Time 1/10secs
               CrtTime     : word;        // Create Time
               CrtDate     : word;        // Create Date
               LastAcc     : word;        // Last Access
               FAT32Res    : word;        // FAT32 reserved word
               WrAccTime   : Word;        // Time of last wr acccess
               WrAccDate   : Word;        // Date of last wr access
               FirstC      : Word;        // Number of the first cluster
               Size        : LongWord;    // File size
             end;


Type TSearchRec  = record                          // only the first 3 entries are relevant
                     Name    : TFileName;          // never change any content
                     Dir     : TDir;
                     Attr    : tFAttr;
                     DirMask : TArrNameExt;
                     Dirs    : LongWord;
                     OCDir   : Word;
                     CDir    : Word;
                     SDir    : LongWord;
                     IDir    : Byte;
                     SNDir   : Word;
                     FRoot   : Boolean;
                   end;
```

Wenn der FAT16_IOS importiert ist, dann exportiert das System diese Kommando Type:

```
Type tF16cmd = (cmdInitF16, cmdCheckF16, cmdReadF16, cmdWriteF16);
```

Variable

```
Var
  FATver       : tFATtype;
```

Allgemeine Funktionen des FileSystems

```
Function F16_DiskInit : boolean;
Function F16_DiskReset : boolean;
Function F16_CheckDisk : boolean;
Function F16_DiskFormat : boolean;
Function F16_GetDiskError : tDiskError;
Function F16_GetDiskSize : longword;
Function F16_GetDiskFree : longword;
Function F16_GetDiskUsed : longword;                    // FAT32 only
Function F16_GetUsedHandles : byte;
Function F16_TimeToStr (FileTime : word) : TF16TimeStr;
Function F16_DateToStr (FileDate : word) : TF16DateStr;
Function F16_StrToTime (strTime : TF16TimeStr) : word;
Function F16_StrToDate (strDate : TF16DateStr) : word;
```

Verwaltungs Funktionen für Pfade und Directories

```
Function F16_GetCurDir : TPathStr;
Function F16_ChangeDir (path : TPathStr) : boolean;
Function F16_CreateDir (path : TPathStr; DirName : TFileName; aTime, aDate : word): boolean;
Function F16_RemoveDir (path : TPathStr; DirName : TFileName) : boolean;
Function F16_DirGetDate(path : TPathStr; FDirName : TFileName; var aTime, aDate : word) : boolean;
Function F16_PathExist (path : TPathStr) : boolean;
Function F16_PathExpand (path : TPathStr; var ExpandedPath : TPathStr) : boolean;
```

Verwaltungs Funktionen für Dateien

```
Function F16_FileExist (path : TPathStr; fn : TFileName; attr : tFAttr) : boolean;
Function F16_FileSize (path : TPathStr; fn : TFileName; var size : longword) : boolean;
Function F16_FileSetAttr (path : TPathStr; fn : TFileName; attr : tFAttr) : boolean;
Function F16_FileGetAttr (path : TPathStr; fn : TfileName; var attr : tFAttr) : boolean;
Function F16_FileSetDate (path : TPathStr; fn : TFileName; aTime, aDate : word) : boolean;
Function F16_FileGetDate (path : TPathStr; fn : TfileName; var aTime, aDate : word) : boolean;
Function F16_FileRename (path : TPathStr; fn, fnNew : TFileName) : boolean;
Function F16_FileDelete (path : TPathStr; fn : TFileName) : boolean;
Function F16_FileCopy (srcPath : TPathStr; srcFn : TfileName;
                       dstPath : TPathStr; dstFn : TFileName) : boolean;
Function F16_FindFirst (path : TPathStr; fn : TFileName; attr : tFAttr;
                       var sr : TSearchRec) : boolean;
Function F16_FindNext (var sr : TSearchRec) : boolean;
```

Nur FAT16_32

```
Function F16_GetLFN_S(var sr : TSearchRec) : string;
```

Nach einem FindFirst oder FindNext kann damit auch ein evtl. vorhandener langer Filenamen erfahren werden.

```
Function F16_GetLFN_F(f : File) : string;
```

Bei geöffneten Files kann damit auch ein evtl. vorhandener langer Filenamen erfahren werden.

Funktionen für geöffnete Dateien

```
Function F16_FileAssign (var f : File; path : TPathStr; fn : TFileName) : boolean;
Function F16_FileReset (f : File) : boolean;
Function F16_FileRewrite (f : File; attr : tFAttr; aTime, aDate : word) : boolean;
Function F16_FileAppend (f : File) : boolean;
Function F16_RandomWrite (f: File) : boolean;
Function F16_FileSeek (f : File; p : longword) : longword;
Function F16_FilePos (f : File) : longword;
Function F16_BlockRead (f : File; pt : pointer; count : word; var res : word) : boolean;
Function F16_BlockWrite (f : File; pt: pointer; count : word; var res : word) : boolean;
Function F16_BlockRandomWrite (f: File; pt: Pointer; Count : Word; var res : Word):boolean;
Function F16_EndOfFile (f : File) : boolean;
Function F16_FileSizeH (f: File) : longword;
Function F16_FileClose (var f : File) : boolean;
Function F16_CheckHandle (f : File) : TFileAccess;
```

Spezial Funktionen

```
Function F16_FileCreate (Path: TPathStr; FName: TFileName; aAttr : tFAttr;
                         aTime, aDate: Word; Size : LongWord) : boolean;
```

Funktionen für File Of Text

```
Procedure Read (f : File; var string|char);
Procedure ReadLn (f : File; var string|char);
Procedure Write (f : File; string|char);
Procedure WriteLn (f : File; string|char);
Procedure WriteLn (f : File);
```

Spezial Funktionen

```
Function F16_ReadSector (SectNum : longword; pt : Pointer) : Boolean;
Function F16_WriteSector (SectNum : longword; pt : Pointer) : Boolean;
```

Imports

Wie beim AVRco System üblich, müssen Devices importiert und definiert werden.

```
Import SysTick, FAT16, ...;
```

oder

```
Import SysTick, FAT16_32, ...;
From System Import longword, ...;
```

Defines

Durch die folgende Define Anweisungen wird das File-System spezifiziert:

```
Define
  ProcClock        = 16000000;      {Hertz}
  SysTick          = 10;            {msec}
  StackSize        = $0040, iData;
  FrameSize        = $0100, iData;
  FAT16            = MMC_SPI, iData;
 F16_MMCspeed      = standard;      // standard, slow, fast, superfast -> XMega + FAT16_32 only
  F16_FileHandles  = 4;
  F16_DirLevels    = 2;
```

MMC-SPI
bezeichnet den importierten Hardware Treiber für das System. iData bestimmt den Speicherbereich für die internen Buffer.

XMega
Mit den XMegas werden bis zu vier SPI-ports unterstützt:
SPI_C, SPI_D, SPI_E, SPI_F

Define
FAT16 = SPI_C, PortA, 6, iData; // PortX bestimmt das SS-Port und n (6) den SS-Pin

MMC_Soft.
Alternativ kann auch ein reiner Software SPI Treiber importiert werden, der dann Port Pins benutzt. Diese sind dann mit dem Define F16_MMCport zu spezifizieren. Der Import des Software SPI erfolgt mit

Define *FAT16 = MMC_SOFT, IData;*
F16_MMCport = PortX, SS, SCK, MOSI, MISO;

Das Define F16_MMCport beschreibt die Bit-Funktionen des zu verwendenden IO-Ports.
Zuerst kommt der Port Name, dann die Position des SS-Pins (Chip select). Dann folgt die Position des Clock Pins, des MOSI Pins und des MISO Pins. Alle Pins bzw. Bits müssen in einem Port sein, splitting ist nicht möglich. Die Bits können jedoch beliebig über das Port verteilt sein.

XMega
Hier können die Bits beliegig über die Ports verteilt werden.
F16_MMCport = PortX.SS, PortY.SCK, PortZ.MOSI, PortW.MISO;

F16_MMCspeed
stellt die SPI Datenrate beim Zugriff auf die MMC oder SD Karte ein:

slow = OSC clock div 16
standard = OSC clock div 4
fast = OSC clock div 2
superfast = 16MHz SPI speed, XMega + FAT16_32 only

Zu beachten dabei ist dass manche Karten zwar im 4bit PC Mode sehr schnell, aber im hier verwendeten SPI Mode relativ langsam sind.

SDIO
SD Karten bieten neben dem SPI Interface auch das 4-bit SDIO Interface. Dieses ist sehr schnell und einem Software SPI hoch überlegen. Beim XMega sind die erzielbaren Datenraten ähnlich dem Hardware SPI.

Define *FAT16 = SDIO, PortE.4, PortB.5, PortB.6, iData; // 4xDATA, CMD, CLK*
Hierbei ist zu beachten dass die 4 Datenbits entweder auf PortX.0 oder PortX.4 beginnen und hintereinander liegen müssen, z.B. PortE.4. Dann folgt das CMD Port.bit und das CLK Port.bit die beliebig verteilt sind.

F16_FileHandles
bezeichnet die mögliche Anzahl der gleichzeitig geöffneten Dateien. Die Lage der zugehörigen File Control Blocks (FCB) wird durch iData/xData bestimmt. Der Speicherbedarf für jeden möglichen File ist relativ gross. Deshalb sollten hier nur das mindest notwendige angegeben werden. Wird immer nur mit einer geöffneten Datei gearbeitet, dann sollte hier auch nur „1“ angegeben werden.

F16_DirLevels
bestimmt die Directory Tiefe des Systems. In den meisten Fällen reicht hier 2 aus. Man sollte vermeiden vom PC aus zu viele Sub-Directories anzulegen. Das kostet alles nur viel Rechenaufwand und noch mehr Speicher. Auch werden die PathStrings entsprechend länger was den Frame Bedarf enorm in die Höhe treiben wird.

Wenn der MMC-SPI Treiber importiert ist, wird das CPU-interne SPI port für das Interface benutzt. Der SS-pin der CPU fungiert dann als die Chip Select Leitung für das Interface. Als Medium können sowohl MMC als auch SD Karten verwendet werden. Der FAT16_32 Treiber unterstützt auch SDHC Karten.

F16_Buffers (FAT16_32 only)
Bestimmt die Benutzung eines oder zwei Sector Buffer. Wenn “double” angegeben ist dann warden zwei separate Buffer benutzt, einer für das FAT und der andere für data read/write. Grösse ist jeweils 512bytes.

Abhängig vom Treiber import muss auch die passende Treiber Unit im Uses Import angegeben werden:
Uses *uFAT16, ...;*
oder
Uses *uFAT16_32, ...;*

3.9.4 Spezial Treiber Implementation

Das System AVRco stellt eine umfangreiche Sammlung von high-level Disk und File Funktionen zur Verfügung. Diese Funktionen werden intern auf einfache 512 Byte Sector-read-write Funktionen reduziert. Diese Zugriffe auf das Speicher Medium werden direkt auf dem physischen Drive (MMC etc) ausgeführt. Es sind keinerlei low-level Treiber Routinen zu erstellen, solange die internen Treiber benutzt werden.

Für allgemeine Hardware unabhängige Implementationen stellt das System eine sogenannte UserDevice Funktion mit dem Namen **FAT16_IOS** zur Verfügung. Diese Routine muss der Programmierer dann selbst implementieren und die hiermit übergebenen Adressen und Parameter in geeigneter Weise an das Speicher Medium weiterleiten.

```
UserDevice FAT16_IOS (F16cmd : tF16cmd; buffer : pointer; BlockAddr : longword) : boolean;
  begin
    case F16cmd of
      cmdInitF16   :       // initialize hardware
                           // buffer and blockaddr are don't care
                 |
      cmdCheckF16  :       // check for an existing file system
                           // buffer and blockaddr are don't care
                 |
      cmdReadF16   :       // read a 512byte block
                 |
      cmdWriteF16  :       // write a 512byte block
                 |
    endcase;
    return(true);
  end;
```

Der Parameter ***Buffer*** ist ein allgemeiner Pointer der zum Lesen und Schreiben verwendet werden muss. Soll vom Medium gelesen werden, so muss diese Funktion diesen Pointer als Schreib-Pointer verwenden. ***Buffer*** zeigt damit auf das Ziel der Lese-Operation. Beim Schreiben auf das Medium zeigt der Pointer auf die Quelle, von der gelesen werden muss und deren Inhalt auf das Medium geschrieben werden muss.

Der Parameter ***BlockAddr*** bezeichnet einen 512 Byte Block auf dem Medium, der das Ziel oder die Quelle einer Lese oder Schreib-Operation ist.

Beide Parameters haben nur Bedeutung für die Kommandos ***cmdReadF16*** und ***cmdWriteF16***.

3.9.5 Exporte des FileSystems

Das FileSystem exportiert einige Typen, Variablen und Konstante, die nur für Debug Zwecke da sind und im Simulator angezeigt werden können. Diese Werte sind für die Applikation irrelevant und werden hier nicht weiter beschrieben.

Exportierte Typen des FileSystems

Type *File; {File of Byte, Char, word, float, string, text, record, array ...)*
Das AVRco FileSystem kennt nur typisierte Dateien, das heisst jede Datei muss mit einem bestimmten Datei-Typ geöffnet werden. Fast alle Standard Typen des Systems und alle User-definierten Typen können

verwendet werden. Ein File of word enthält dann nur Words und die FileSize, FilePos etc. wird in Word-Count angegeben. Jede Lese- und Schreib Operation erfolgt damit auf Word-Basis.

```
Type tFileW : file of word;
Var ff : tFileW;
F16_FileAssign (ff, path, FileName);     // opens a file of word
F16_BlockRead (ff, buffer, 2, res);      // reads 2 words out of ff into buffer
F16_BlockWrite (ff, buffer, 3, res);     // writes 2 words out of buffer into ff
F16_FileSize (path, FileName, Lw);       // returns the size of "FileName" in bytes
F16_FileSizeH (ff);                      // returns the size in counts of words
```

Eine File Definition ohne Angabe eines Typs ist identisch mit File Of Byte

```
Var ff : file;                           // the same as file of byte
```

```
Type tFAttrEnum   = (faReadOnly, faHidden, faSysFile, faVolumeID, faDirectory, faArchive);
Type tFAttr       = BitSet of tFAttrEnum;
```

Existierende Dateien können durch die Funktion F16_FileSetAttr mit Attributen versehen werden. Das Attribut *faReadOnly* schützt die Datei gegen weiteres Beschreiben und Löschen.

```
Type tDiskError  = (deNone, deMediaUnknown, deFATunknown, deInitFail, deHandleFail,
                    deReadFail, deWriteFail, deReadOnly, deFileExists, deNotFound);
```

Dieser Fehlertyp wird von der Funktion F16_GetDiskError zurückgegeben.

```
Type TFileAccess = (faNone, faAssign, faRead, faWrite, faAppend, faRandomWr);
```

Die Funktion CheckHandle gibt einen Wert von diesem Typ zurück der den aktuellen Datei und FileHandle Status anzeigt.

```
Type TDir  = record
               NameExt     : TArrNameExt;     // Filename or directory
               FAttr       : tFAttr;          // The file attributes
               NTRes       : byte;            // NT reserved byte
               CrtTim10    : byte;            // Create Time 1/10secs
               CrtTime     : word;            // Create Time
               CrtDate     : word;            // Create Date
               LastAcc     : word;            // Last Access
               FAT32Res    : word;            // FAT32 reserved word
               WrAccTime   : word;            // Time of last wr acccess
               WrAccDate   : word;            // Date of last wr access
               FirstC      : word;            // Number of the first cluster
               Size        : longword;        // File size
             end;
```

Dieser Record ist Bestandteil des SearchRecords weiter unten. Er enthält 3 wichtige Parameter: NameExt ist der gefundene File/Directory Namen, Fattr sind dessen Attribute und Size ist dessen Datei Grösse in Bytes.

```
Type TArrNameExt = array[1..11] of Char;          // internal use

Type TSearchRec  = record                         // only the first 3 entries are relevant
                     Name      : TFileName;       // never change any content
                     Dir       : TDir;
                     Attr      : tFAttr;
                     DirMask   : TArrNameExt;
                     Dirs      : longword;
                     OCDir     : word;
                     CDir      : word;
                     SDir      : longword;
                     IDir      : byte;
                     SNDir     : word;
                     FRoot     : boolean;
                   end;
```

File- und Dir-List Funktionen arbeiten mit diesem Record. Die ersten drei Parameter sind für die Applikation wichtig. Der Rest ist nur für den internen Gebrauch. Für alle Parameter gilt, dass diese nicht verändert werden dürfen.

Type *TFileName = string[12];*
Alle File Operationen, die einen Dateinamen brauchen, arbeiten mit dem String-Typ tFName. Das gilt auch für die DIR-List Funktionen, die einen Dateinamen zurückliefern.

Type *TpathStr = string[nn]; // nn depends on max. DirLevels allowed*
Type *TF16TimeStr = string[5];*
Type *TF16DateStr = string[8];*

Type *tF16cmd = (cmdInitF16, cmdCheckF16, cmdReadF16, cmdWriteF16);*

Wenn der **FAT16_IOS** importiert ist, dann exportiert das System diese Kommando Type.

cmdInitF16 : Hardware Initialisierung
cmdCheckF16 : prüft ob das Drive ready ist und ob ein Medium vorhanden ist
cmdReadF16 : liest einen 512 Byte Block vom Laufwerk
cmdWriteF16 : schreibt einen 512 Byte Block in die Disk

Vordefinierte File Attribute

Const *faAnyFile : tFAttr = tFAttr($3F);*
Das Bitset ***faAnyFile*** umfasst alle Dateien und Directories

Const *faFilesOnly : tFAttr = [faReadOnly, faHidden, faSysFile, faArchive];*
Das Bitset ***faFilesOnly*** umfasst alle Dateien, schliesst aber Directories aus.

Achtung: diese beiden vordefinierten Bitsets dürfen nicht in **[fa…]** gesetzt werden.
F16_FileExist ('\', abc.txt, faAnyFile)
F16_FileExist ('\', abc.txt, [faReadOnly])

3.9.6 Disk und Drive Funktionen des FileSystems

Grundsätzlich wird versucht, alle Fehlermöglichkeiten abzufangen. Daher werden alle Drive und Directory Angaben geprüft und ggf. mit einem FALSE zurückgewiesen. Das gleiche gilt normalerweise, wenn versucht wird mit WildCards oder Joker eine Funktion zu starten, wo dies verboten ist. Laufzeit Fehler wie z.B. FileReadOnly Write Access etc. werden erkannt und mit einem FALSE beantwortet. Auch falsche oder geschlossene FileHandles werden normalerweise erkannt.

Die Laufzeit Fehler Möglichkeiten werden am Ende der Funktionsbeschreibung aufgelistet. Bei einer fehlgeschlagenen Funktion kann der Fehlercode mit der Funktion F16_GetDiskError abgefragt werden.

Function *F16_DiskInit : boolean;*
Diese Prozedur initialisiert die angeschlossene Hardware. Alle offenen Dateien müssen vorher geschlossen werden. Anschliessend muss die Funktion DiskReset aufgerufen werden.

Function *F16_CheckDisk : boolean;*
Prüft das Vorhandensein eines Mediums im Drive und ob dieses „ready" ist. Diese Funktion kann jederzeit ausgeführt werden.

Function *F16_DiskReset : boolean;*
Diese Prozedur initialisiert alle internen Buffer, setzt alle internen Daten und Pointer zurück und baut anschliessend alle Control-Blocks wieder neu auf. Alle offenen Dateien müssen vorher geschlossen werden. Diese Funktion muss auch immer nach einem Medium Wechsel aufgerufen werden

***Function** F16_DiskFormat : boolean;*
Eine noch nie benutzte Disk/Drive muss wie bei anderen System auch üblich formatiert werden. Dies kann allerdings nicht hiermit geschehen, da ein Bootblock und viele weitere Details geschrieben werden müssten, das den Rahmen des Systems sprengen würde. Daher muss dieser Vorgang grundsätzlich auf einem PC unter Windows etc. erfolgen.
Mit dieser Funktion wird der Directory Teil des Drives komplett gelöscht. Alle offenen Dateien müssen vorher geschlossen werden. Nach dem Aufruf muss zumindest die Funktion DiskReset durchgeführt werden.

***Function** F16_GetDiskSize : longword;*
FAT16 Treiber: gibt die Kapazität des Drives in bytes zurück.
FAT16_32 Treiber: mit einer FAT16 Disk wird ein Bytecount zurückgegeben, bei einer FAT32 Disk das Ergebnis ist in kBytes. Der FAT Typ kann in dem globalen Byte “FATver“ abgefragt werden.

***Function** F16_GetDiskFree : LongWord;*
Gibt den noch freien Speicherplatz in Bytes des aktuellen Mediums zurück. Beim FAT32 Treiber in kBytes. Diese Funktion erzeugt einen grossen Disk Verkehr braucht deshalb relativ viel Zeit.

***Function** F16_GetDiskUsed : LongWord;*
Gibt den noch freien Speicherplatz in kBytes des aktuellen Mediums zurück. **Nur beim FAT32 Treiber**. Diese Funktion erzeugt einen grossen Disk Verkehr braucht deshalb relativ viel Zeit.

***Function** F16_GetUsedHandles : byte;*
Gibt die Anzahl der belegten FileHandles zurück.

***Function** F16_GetDiskError : tDiskError;*
Der Status der letzten Operation kann jederzeit mit dieser Funktion abgefragt werden. Die Abfrage setzt den Status anschliessend immer auf deNone.

Der Aufzählungstyp tDiskError besteht aus:
deNone, deMediaUnknown, deFATunknown, deInitFail, deHandleFail, deReadFail, deWriteFail, deReadOnly, deFileExists, deNotFound

3.9.7 Support Funktionen des FileSystems

***Function** F16_TimeToStr (FileTime : word) : tF16TimeStr;*
Konvertiert eine DOS FileTime in einen String. Format = “hh:mm“

***Function** F16_DateToStr (FileDate : word) : tF16DateStr;*
Konvertiert ein DOS FileDate in einen String. Format = “dd.mm.yy“

***Function** F16_StrToTime (strTime : tF16TimeStr) : word;*
Konvertiert einen String in ein DOS FileTime word.

***Function** F16_StrToDate (strDate : tF16DateStr) : word;*
Konvertiert einen String in ein DOS FileDate word.

3.9.8 Directory und Pfad Funktionen des FileSystems

***Function** F16_GetCurDir : TPathStr;*
Gibt das aktuelle *DefaultDirectory/Path* zurück. Dieser Pfad/Directory wird immer dann bei File Operationen verwendet, wenn eine solche Funktion keine Pfad oder Directory Angabe enthält.

***Function** F16_ChangeDir (path : TPathStr) : boolean;*
Stellt die *DefaultDirectory/Path* für alle File Operationen ein, bei denen keine Pfade oder Directories angegeben wurden.

***Function** F16_CreateDir (path : TPathStr; DirName : TFileName; aTime, aDate : word): boolean;*
Diese Funktion erstellt eine neue Directory. ***Path*** gibt den dafür notwendigen Pfad an, unter dem der neue Eintrag erfolgen soll. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path*

herangezogen. In ***DirName*** wird der zu erstellende Eintrag vorgegeben. Die Parameter ***aTime*** und ***aDate*** sollten die aktuelle Uhrzeit und das Datum enthalten. Zum Erstellen dieser Parameter können die Funktionen *F16_StrToDate* und *F16_StrToTime* verwendet werden.

Function *F16_RemoveDir (path : TPathStr; DirName : TFileName) : boolean;*
Diese Funktion löscht eine vorhandene Directory. ***Path*** gibt den dafür notwendigen Pfad an, unter dem der Eintrag zu finden ist. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***DirName*** wird der zu löschende Eintrag vorgegeben.

Function *F16_PathExist (path : TPathStr) : boolean;*
Die Funktion prüft, ob der angegebene Pfad vorhanden ist.

Function *F16_PathExpand(path : TPathStr;* ***var*** *ExpandedPath : TPathStr) : boolean;*
Die Funktion expandiert einen relativen Pfad in einen absoluten Pfad.
Relative Pfade sind zum Beispiel

..\ und .\ oder auch ..\name

3.9.9 Funktionen für Dateien

3.9.9.1 Verwaltungs Funktionen für Dateien

Einige dieser Funktionen erwarten einen Parameter ***attr*** vom Typ ***tFAttr***. Dieser Parameter ist ein Bitset bestehend aus der Enumeration ***faReadOnly, faHidden, faSysFile, faVolumeID, faDirectory, faArchive***
Das Bitset ***faAnyFile*** umfasst alle Dateien und Directories
Das Bitset ***faFilesOnly*** umfasst alle Dateien, schliesst aber Directories aus.
Zu beachten ist dabei, dass ein File natürlich mehrere Attribute gleichzeitig besitzen kann:
[faArchive, faReadOnly].

Function *F16_FileExist (path : TPathStr; fn : TFileName; attr : tFAttr) : boolean;*
Diese Funktion prüft die Existenz einer Datei. ***Path*** gibt den dafür notwendigen Pfad an, unter dem das File zu finden ist. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***fn*** wird der Filenamen vorgegeben. Der Parameter ***attr*** schränkt die Suche auf Dateien mit einem bestimmten Attribut ein.

Function *F16_FileSize (path : TPathStr; fn : TFileName;* ***var*** *size : longword) : boolean;*
Diese Funktion errechnet die Datei Grösse (Bytes) einer vorhandenen Datei. ***Path*** gibt den dafür notwendigen Pfad an, unter dem das File zu finden ist. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***fn*** wird der Filenamen vorgegeben. Der Parameter ***size*** enthält die aktuelle Dateigrösse in Bytes.

Function *F16_FileSetAttr (path : TPathStr; fn : TFileName; attr : tFAttr) : boolean;*
Diese Funktion verändert die Attribute einer vorhandenen Datei. ***Path*** gibt den dafür notwendigen Pfad an, unter dem das File zu finden ist. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***fn*** wird der Filenamen vorgegeben. Der Parameter ***attr*** überschreibt die vorhandenen Datei Attribute.

Function *F16_FileGetAttr (path : TPathStr; fn : TfileName;* ***var*** *attr : tFAttr) : boolean;*
Diese Funktion liest die Attribute einer vorhandenen Datei. ***Path*** gibt den dafür notwendigen Pfad an, unter dem das File zu finden ist. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***fn*** wird der Filenamen vorgegeben. Im Parameter ***attr*** stehen dann die aktuellen Attribute der Datei.

Function *F16_FileSetDate (path : TPathStr; fn : TFileName; aTime, aDate : word) : boolean;*
Diese Funktion verändert das File Datum und Uhrzeit einer vorhandenen Datei. ***Path*** gibt den dafür notwendigen Pfad an, unter dem das File zu finden ist. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***fn*** wird der Filenamen vorgegeben. Die Parameter ***aTime*** und ***aDate*** enthalten die neue Uhrzeit und Datum. Zum Erstellen dieser Parameter können die Funktionen *F16_StrToDate* und *F16_StrToTime* verwendet werden.

Function *F16_FileGetDate (path : TPathStr; fn : TfileName;* ***var*** *aTime, aDate : word) : boolean;*
Diese Funktion liest das File Datum und Uhrzeit einer vorhandenen Datei. ***Path*** gibt den dafür notwendigen Pfad an, unter dem das File zu finden ist. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***fn*** wird der Filenamen vorgegeben. Die Parameter ***aTime*** und ***aDate*** enthalten die gelesene Uhrzeit und das Datum.

Function *F16_FileRename (path : TPathStr; fn, fnNew : TFileName) : boolean;*
Diese Funktion ändert den Filenamen einer vorhandenen Datei. ***Path*** gibt den dafür notwendigen Pfad an, unter dem das File zu finden ist. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***fn*** wird der alte Filenamen vorgegeben. In ***fnNew*** wird der neue Filenamen vorgegeben.

Function *F16_FileDelete (path : TPathStr; fn : TFileName) : boolean;*
Diese Funktion löscht eine vorhandene Datei. ***Path*** gibt den dafür notwendigen Pfad an, unter dem das File zu finden ist. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***fn*** wird der Filenamen vorgegeben.

Function *F16_FileCopy (srcPath : TPathStr; srcFn : TfileName;*
dstPath : TPathStr; dstFn : TFileName) : boolean;
Diese Funktion kopiert eine Datei in ein anderes Verzeichnis oder auch unter neuem Namen in das gleiche Verzeichnis. ***srcPath*** gibt den dafür notwendigen Quell-Pfad an, unter dem das File zu finden ist. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***srcFn*** wird der Quell Filenamen vorgegeben. In ***dstPath*** wird der Ziel-Pfad vorgegeben und in ***dstFn*** wird der neue Filenamen vorgegeben.

3.9.9.2 Allgemeines Suchen und Auflisten von Dateien

Eine oft gebrauchte Funktion in allen Datei Systemen ist das Suchen und Auflisten von Files. Hierbei werden durch Wildcards und Joker die Suche erweitert und eventuell durch die Vorgabe der File-Attribute die Suche wiederum auf bestimmte Datei Gruppen eingeschränkt.

Zur Suche muss die Applikation einen Speicherbereich vom Typ **tSearchRec** bereitstellen. Die Initial Such Funktion ***FindFirst*** füllt den Record mit den Such Parametern und sucht auch das erste passende File. Wurde eins gefunden, kann jetzt mit ***FindNext*** solange gesucht werden, bis kein File mehr gefunden wurde oder das gesuchte erreicht ist.

Jede Funktion stellt in dem Record in dem Parameter ***SR.Name*** den Dateinamen bereit. Die Parameter des Records dürfen nur gelesen aber niemals verändert werden.

Function *F16_FindFirst (path : TPathStr; fn : TFileName; attr : tFAttr;* ***var*** *sr : TSearchRec) : boolean;*
Diese Funktion eröffnet die Suche nach bestimmten Dateien, die z.B. gleiche Extensions haben, oder die alle das Attribut readonly haben. Es sind beliebige Kombinationen von Joker, Wildcards und Attributen möglich.
Path gibt den dafür notwendigen Pfad an, unter dem die Files zu finden sind. Hier sind keine Wildcards oder Joker erlaubt. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***fn*** wird der Filenamen (mit Joker und/oder Wildcards) vorgegeben. Im Parameter ***attr*** werden die gewünschten Attribute vorgegeben.

Function *F16_FindNext (****var*** *sr : TSearchRec) : boolean;*
Nach dem Eröffnen der Suche mit obiger Funktion FindFirst kann mit dieser Funktion solange gesucht werden, bis ein false zurückkommt.

```
var
  SR   : TSearchRec;
  st   : TfileName;

// search for filename/directory entries, accept wildcards
if F16_FindFirst ('\', '*.*', faAnyFile, SR) then
  repeat
    // filename processing ...
```

```
    st:= SR.Name;
  until not F16_FindNext(SR);
endif;
```

3.9.9.3 Funktionen für offene Dateien

Alle Funktionen dieser Rubrik benötigen einen File. Dieser wird durch die Funktion FileAssign generiert und bleibt gültig bis er mit FileClose wieder an das System zurückgegeben wird. Das File bestimmt durch die Art seiner Definition, wie eine Datei verarbeitet wird. So wie eine Variable aus einem Typ besteht, müssen es die Dateien auch.

Das Datei System arbeitet grundsätzlich mit typisierten Dateien, d.h. eine Datei besteht immer aus einer bestimmten Anzahl von Records, wobei jeder Record ein Datentyp darstellt.

Ein ***File of integer*** besteht damit aus der Anzahl ***n*** Integer bzw. Integer Records. Der jeweilige Datei-Typ wird beim Erstellen einer Datei aber nicht in die Datei hineingeschrieben. Nur die Funktion **FileAssign** zusammen mit dem File bestimmt den Dateityp und damit die Art bzw. Grösse des Records in Bytes. Sämtliche Datei Operationen, die eine Recordzahl erwarten oder als Resultat zurückgeben, rechnen mit diesen logischen Records und nicht mit Bytes. Auch ein ***File of Byte*** besteht damit aus Records, allerdings mit der Grösse 1Byte. Es können praktisch alle Standard Datentypen als Record Grösse angegeben werden, incl. einem userdefined record oder array. Bei ***File of string*** muss der String eine Längenangabe enthalten (file of string[10]). Es werden damit immer diese Anzahl von Zeichen gelesen und geschrieben (ein record), dazu kommt noch das Längenbyte.

File of Text bildet eine Ausnahme. Da Strings hier eine beliebige Länge haben können und nur durch ein CRLF abgeschlossen werden, kann hier nicht mit Records gearbeitet werden. Die Dateigrösse wird immer in Bytes zurückgegeben und ein „normales“ lesen und schreiben erfolgt immer auf Byte Basis. Deshalb gibt es hierzu die Spezial Funktionen **Read, ReadLn, Write, WriteLn.** Diese Lesen z.B. (ReadLn) solange ab der aktuellen Dateiposition in den Zielstring, bis ein CRLF erreicht wird oder der Ziel String voll ist. Textfiles können nur sequentiell gelesen und geschrieben werden. Auch ein FileSeek macht hier keinen Sinn.

Mit dem generierten FileHandle können Dateien zum Lesen ***F16_FileReset*** oder Schreiben ***FileRewrite***, ***FileAppend, RandomWrite*** geöffnet werden. Wenn eine Datei geöffnet wurde, sollten keine der obenstehenden Verwaltungs Funktionen (FileDelete, FindFirst etc) aufgerufen werden. Das kann zu Fehlern führen.

Führt eine FileOpen Funktion auf gleichzeitig das Neu-Erstellen des Files durch, müssen dazu die Datei Attribute und das Erstellungsdatum/Uhrzeit mit angegeben werden.

Die Lese und Schreib Operationen dieses FileSystems sind in erster Linie ***sequentiell***. Das heisst, normalerweise wird fortlaufend gelesen oder geschrieben. Beginnend vom File Anfang. Für das Lesen kann auch ein ***Random Read*** verwendet werden, indem auf eine bestimmten Record positioniert wird (FileSeek), dieser Record wird dann gelesen und eine erneute Positionierung mit Lesen oder ein sequentielles Lesen kann erfolgen.

Random Write kann nur auf existierende Dateien angewandt werden. Dabei kann nur innerhalb dieser Datei geschrieben werden. Ein Schreiben über das Datei Ende hinaus ist nicht möglich.

Function *F16_FileAssign (**var** f : File; path : TPathStr; fn : TFileName) : boolean;*
Diese Funktion generiert das File (Handle), das für alle weiteren Operationen dieser Kategorie benötigt wird. Die Definition der Variablen ***f*** (File; File of Byte; etc) bestimmt dabei die Recordgrösse aller Funktionen. ***Path*** gibt den dafür notwendigen Pfad an, unter dem das File zu finden ist (keine WildCards oder Joker). Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***fn*** wird der Filenamen vorgegeben (keine WildCards oder Joker).

Function *F16_CheckHandle (f : File) : tFileAccess;*
Ein FileHandle kann zu jeder Zeit mit dieser Funktion auf Gültigkeit geprüft werden. Weiterhin gibt das Ergebnis den aktuellen Zustand bzw. Modus an.

faNone	das FileHandle ist ungültig
faAssign	das FileHandle ist durch FileAssign zwar gültig, die Datei ist aber noch nicht geöffnet
faRead	das FileHandle ist gültig, die Datei wurde durch FileReset zum Lesen eröffnet
faWrite	das FileHandle ist gültig, die Datei wurde durch FileRewrite zum Schreiben eröffnet
faAppend	das FileHandle ist gültig, die Datei wurde durch FileAppend zum Schreiben eröffnet
faRandomWr	das FileHandle ist gültig, die Datei wurde durch FileBlockWrite zum Schreiben eröffnet

Function *F16_FileReset (f : File) : boolean****;***
Dies ist eine der vier möglichen FileOpen Funktionen. Diese Funktion öffnet ein vorhandenes File zum Lesen. Die Datei muss existieren. Der Parameter ***f*** muss zuvor mit FileAssign erstellt werden.
Mögliche Operationen sind dann ***FileSeek***, ***FilePos***, ***EndOfFile***, ***FileSizeH***, ***BlockRead*** und ***FileClose***.

Function *F16_FileRewrite (f : File****;*** *attr : tFAttr; aTime, aDate : word) : boolean****;***
Dies ist eine der vier möglichen FileOpen Funktionen. Diese Funktion öffnet ein vorhandenes File zum Schreiben. Die Datei kann aber muss nicht existieren. Eine schon vorhandene Datei wird zuerst gelöscht. Alle Schreib Operationen hängen Daten an das File Ende an (sequentielles Schreiben).Der Parameter ***f*** muss zuvor mit FileAssign erstellt werden. Die Parameter ***aTime*** und ***aDate*** geben die Uhrzeit und Datum vor. Zum Erstellen dieser Parameter können die Funktionen *F16_StrToDate* und *F16_StrToTime* verwendet werden.
Mögliche Operationen nach dem Eröffnen sind dann ***FileSizeH*****,** ***BlockWrite*** und ***FileClose***.

Function *F16_FileAppend (f : File) : boolean****;***
Dies ist eine der vier möglichen FileOpen Funktionen. Diese Funktion öffnet ein vorhandenes File zum Schreiben. Die Datei muss existieren. Der interne Write Pointer wird an das Ende der Datei positioniert. Alle Schreib Operationen hängen Daten an das File Ende an (sequentielles Schreiben). Der Parameter ***f*** muss zuvor mit FileAssign erstellt werden.
Mögliche Operationen nach dem Eröffnen sind dann ***FileSizeH*****,** ***BlockWrite*** und ***FileClose***.

Function *F16_RandomWrite (f: File): boolean;*
Dies ist eine der vier möglichen FileOpen Funktionen. Diese Funktion öffnet ein vorhandenes File zum Schreiben. Die Datei muss existieren. Der interne Write Pointer wird an den Anfang der Datei positioniert. Mit ***FileSeek*** kann jetzt der Write Pointer an beliebige Stellen innerhalb der Datei positioniert werden. Alle Schreib Operationen mit ***BlockRandomWrite*** überschreiben die Daten an dieser Position (random write). Der Parameter ***f*** muss zuvor mit FileAssign erstellt werden.
Mögliche Operationen nach dem Eröffnen sind dann ***FileSizeH***, ***FileSeek***, ***BlockRandomWrite*** und ***FileClose***.

Function *F16_FileSeek (f : File; p : longword) : longword****;***
Diese Funktion positioniert den Lese/Schreib Pointer in einem mit ***FileReset*** oder ***RandomWrite*** geöffneten File an die Position ***p***. Alle Schreib Operationen mit ***BlockRandomWrite*** überschreiben die Daten an dieser Position. Eine Lese Operation BlockRead liest ab dieser Position. Der Parameter ***f*** muss zuvor mit FileAssign erstellt werden. Der Parameter ***p*** zählt in Records und definiert die gewünschte Position.

Function *F16_FilePos (f : File) : longword****;***
Diese Funktion gibt den aktuellen Lese/Schreib Pointer in einem mit ***FileReset*** oder ***RandomWrite*** geöffneten File zurück. Der Parameter ***f*** muss zuvor mit FileAssign erstellt werden. Das Ergebnis zählt in Records.

Function *F16_BlockRead (f : File; pt : pointer; count : word;* ***var*** *res : word) : boolean;*
Diese Funktion liest einen Datenblock aus einem mit ***FileReset*** geöffneten File an die Pointer Position ***pt***. Der Parameter ***f*** muss zuvor mit ***FileAssign*** erstellt werden und die Datei muss mit ***FileReset*** geöffnet worden sein. Der Parameter ***count*** zählt in Records und definiert den Record Count = Anzahl der zu lesenden Records. Die tatsächliche Anzahl der gelesenen Records wird in ***res*** zurückgegeben.

Function *F16_BlockWrite (f : File; pt : pointer; count: word;* ***var*** *res: word) : boolean;*
Diese Funktion schreibt einen Datenblock von der Pointer Position ***pt*** aus einem mit ***FileRewrite*** oder ***FileAppend*** geöffnetes File. Der Parameter ***f*** muss zuvor mit ***FileAssign*** erstellt werden und die Datei muss mit ***FileRewrite*** geöffnet worden sein. Der Parameter ***count*** zählt in Records und definiert den Record Count = Anzahl der zu schreibenden Records. Die tatsächliche Anzahl der geschriebenen Records wird in ***res*** zurückgegeben.

Function *F16_BlockRandomWrite (f: File; pt: pointer; Count : word;* ***var*** *res : Word*) : boolean;
Diese Funktion schreibt einen Datenblock von der Pointer Position ***pt*** aus einem mit ***FileRandomWrite*** geöffnetes File. Das Ziel im File kann mit der Funktion ***FileSeek*** bestimmt werden. Der Parameter ***f*** muss zuvor mit ***FileAssign*** erstellt werden und die Datei muss mit ***FileRandomwrite*** geöffnet worden sein. Der Parameter ***count*** zählt in Records und definiert den Record Count = Anzahl der zu schreibenden Records. Die tatsächliche Anzahl der geschriebenen Records wird in ***res*** zurückgegeben.

Der **Unterschied** zwischen BlockWrite und BlockRandomWrite besteht darin, dass mit BlockWrite immer nur sequentiell geschrieben werden kann, ein FileSeek ist hier nicht zulässig. Da jeder Schreibvorgang automatisch ans Ende der Datei angehängt wird vergrössert sich die Datei auch mit jedem BlockWrite Zugriff.

Mit BlockRandomWrite kann nur innerhalb einer existierenden Datei geschrieben (überschrieben) werden. Ein Schreiben über das Dateiende hinaus ist nicht zulässig. Deshalb wird hier die Dateigrösse auch nicht verändert.

Function *F16_EndOfFile (f : File) : boolean;*
Diese Funktion gibt ein true zurück, wenn bei einem Lesevorgang das Dateiende erreicht wurde. Der Parameter ***f*** muss zuvor mit ***FileAssign*** erstellt werden und die Datei muss mit ***FileReset*** geöffnet worden sein.

Function *F16_FileSizeH (f: File) : longword;*
Diese Funktion gibt die aktuelle Dateigrösse in Records zurück, unabhängig davon ob gelesen oder geschrieben wird. Der Parameter ***f*** muss zuvor mit ***FileAssign*** erstellt werden und die Datei muss mit ***FileReset***, ***FileRewrite*** oder ***FileRandomWrite*** geöffnet worden sein.

Function *F16_FileClose (****var*** *f : File) : boolean;*
Ein nicht mehr benötigtes FileHandle muss unbedingt mit dieser Funktion an das System zurückgegeben werden. Ausser der Freigabe des Handles erfolgt auch ein Datei Update im Directory und Datenteil der Datei. Bei einer nicht geschlossenen Datei besteht immer die Möglichkeit, dass durch einen Absturz oder PowerDown das Schliessen nicht mehr rechtzeitig stattfinden kann und das File zumindest korrumpiert ist. Im schlimmsten Fall ist das gesamte File System jetzt unbrauchbar.

3.9.9.4 Spezial Funktionen

Manchmal macht es Sinn eine neue Datei mit einer bestimmten Grösse zu erstellen. Trotz der Datei Grösse enthält diese eigentlich keine Daten. Die Datei kann aber mit BlockRandomWrite aufgefüllt werden.

Function *F16_FileCreate (Path: TPathStr; FName: TFileName; aAttr : tFAttr; aTime, aDate: Word; Size : LongWord) : boolean;*

Diese Funktion erstellt eine neue Datei und gibt gleichzeitig auch deren Dateigrösse vor. ***Path*** gibt den dafür notwendigen Pfad an, unter dem das File erstellt werden soll. Ist dieser Parameter ein Leerstring, so wird dazu das aktuelle *DefaultDirectory/Path* herangezogen. In ***fn*** wird der Filenamen vorgegeben. Die Parameter ***aTime*** und ***aDate*** geben die Uhrzeit und Datum vor. Zum Erstellen dieser Parameter können die Funktionen *F16_StrToDate* und *F16_StrToTime* verwendet werden. Der Parameter ***size*** gibt die Dateigrösse in Bytes vor. Eine evtl. schon vorhandene Datei wird gelöscht.

3.9.9.5 Funktionen für File Of Text

File of Text bedürfen der besonderen Behandlung. Strings werden hier ohne Längenbyte gelesen und geschrieben. Damit das Ende eines Strings eindeutig indentifiziert werden kann, wird jeder String mit einem CRLF ($0D $0A) abgeschlossen. Da ein String beliebig lang sein kann, kann hier nicht mit Records gearbeitet werden und die File Positionierungen und FileSize Funktionen haben hier eigentlich keine grosse Aussage.
Auch Random Read und Write macht hier keinen Sinn. File of Text sind reine sequentielle Dateien.

Um aber trotzdem eine relativ einfache Handhabung sicher zu stellen, wurden die schon bekannten System Funktionen Read, Write, ReadLn und WriteLn auf Files erweitert. Da hierbei das Datei Ende nicht durch die Read bzw. ReadLn Funktion selbst festgestellt werden kann, muss hier immer im Wechsel mit der Funktion *EndOfFile* gearbeitet werden.

Procedure *Read (f : file;* ***var*** *string|char);*
Liest ein Zeichen aus dem File in das Ziel, das eine Char oder String Variable sein kann. Ein String bekommt dann die Länge 1. Es werden auch die Limiter CR und LF gelesen und übertragen.

Procedure *ReadLn (****const*** *f : file;* ***var*** *string|char);*
Liest einen String aus dem File in das Ziel, das eine Char oder String Variable sein kann. Bei einem Char als Ziel bricht die Funktion natürlich sofort nach einem Zeichen ab und macht deshalb normalerweise wenig Sinn. Ein String wird gefüllt bis entweder der String voll ist oder ein CRLF erkannt wird. War der String schon vorher voll, bleibt der Lesepointer an dieser Stelle stehen. Das Längenbyte des Ziel Strings wird auf die richtige Länge gesetzt. Die Limiter CR und LF werden zwar gelesen aber niemals in den String übertragen.

Procedure *Write (****const*** *f : file; string|char);*
Schreibt ein Zeichen aus der Quelle, das ein Char oder String sein kann in die Datei. Es wird kein Limiter CRLF angehängt.

Procedure *WriteLn (****const*** *f : file; string|char);*
Schreibt einen String aus der Quelle, das ein Char oder String sein kann, in die Datei. Es wird der Limiter CRLF angehängt.

Procedure *WriteLn (****const*** *f : file);*
Schreibt einen Leerstring, nämlich nur ein CRLF in die Datei.

Keine dieser Write Funktionen überträgt das Längenbyte.

Das Schreiben von Strings wird enorm beschleunigt indem ein Zwischen Puffer (optional) importiert wird:

```
Define    F16_StrLen = 20;  // 4..254
```

Der Wert muss so gewählt sein, dass der längste zu schreibende String in diesen Buffer reinpasst. Überlängen werden abgeschnitten!

3.9.10 Konkurrierende SPI-Treiber

Normalerweise beansprucht der FAT16 MMC Treiber das SPI Port ganz für sich. Sollte ein weiterer SPI Slave an dieses Port angeschlossen werden müssen, so kann der Treiber den SS-PIN nicht mehr intern steuern.
Die Applikation muss dieses übernehmen und u.U. auch das SPI Protokoll auf den aktuell zu selektierenden Slave anpassen. Dazu dient die Call-Back Funktion *onFAT16_SS*

```
Procedure onFAT16_SS;
```

Findet das System diese Prozedur in der Applikation steuert der Treiber den zugehörigen SS-Pin der CPU nicht mehr sondern ruft diese Prozedur auf die dann einen beliebigen Port Pin als SPI Chipselect steuern muss.
Im Register _ACCA wird der Selekt Wert übergeben. _ACCA = 0 -> Chipselekt aktivieren.
_ACCA <> 0 -> Chipselekt deaktivieren. U.U. muss hierbei evtl. der SPI Mode für den aktuellen Slave berichtigt werden. Weiterhin ist zu beachten, dass solche Treiber wie SPI, UART, TWI etc. niemals re-entrant sein können.

Beispiele

```
Var  fs  : file of text;

F16_FileAssign (fs, 'Strings.Tst');      // create a file handle
F16_FileRewrite (fs, [], 0, 0);          // open the file for writing
WriteLn (fs, 'Monday');                  // write first string at file start
WriteLn (fs, 'Tuesday');                 // write next strings
WriteLn (fs, 'Wednesday');
WriteLn (fs, 'Thursday');
WriteLn (fs, 'Friday');
WriteLn (fs, 'Saturday');
Write (fs,   'Sunday ');
Write (fs,   'is weekend');
WriteLn (fs);                            // write an empty string
F16_FileClose (fs);

F16_FileAssign (fs, 'Strings.Tst');
F16_FileReset (fs);                      // open the file for reading
ReadLn (fs, st);                         // read first string
Read (fs, ch);                           // read first char of next string
Read (fs, st, 4);                        // read 4 chars into string
ReadLn (fs, st);                         // read rest of string into string
while not F16_EndOfFile (fs) do          // read the entire file
  ReadLn (fs, st);                       // read the next string
endwhile;                                // until end of file
F16_FileClose (fs);
```

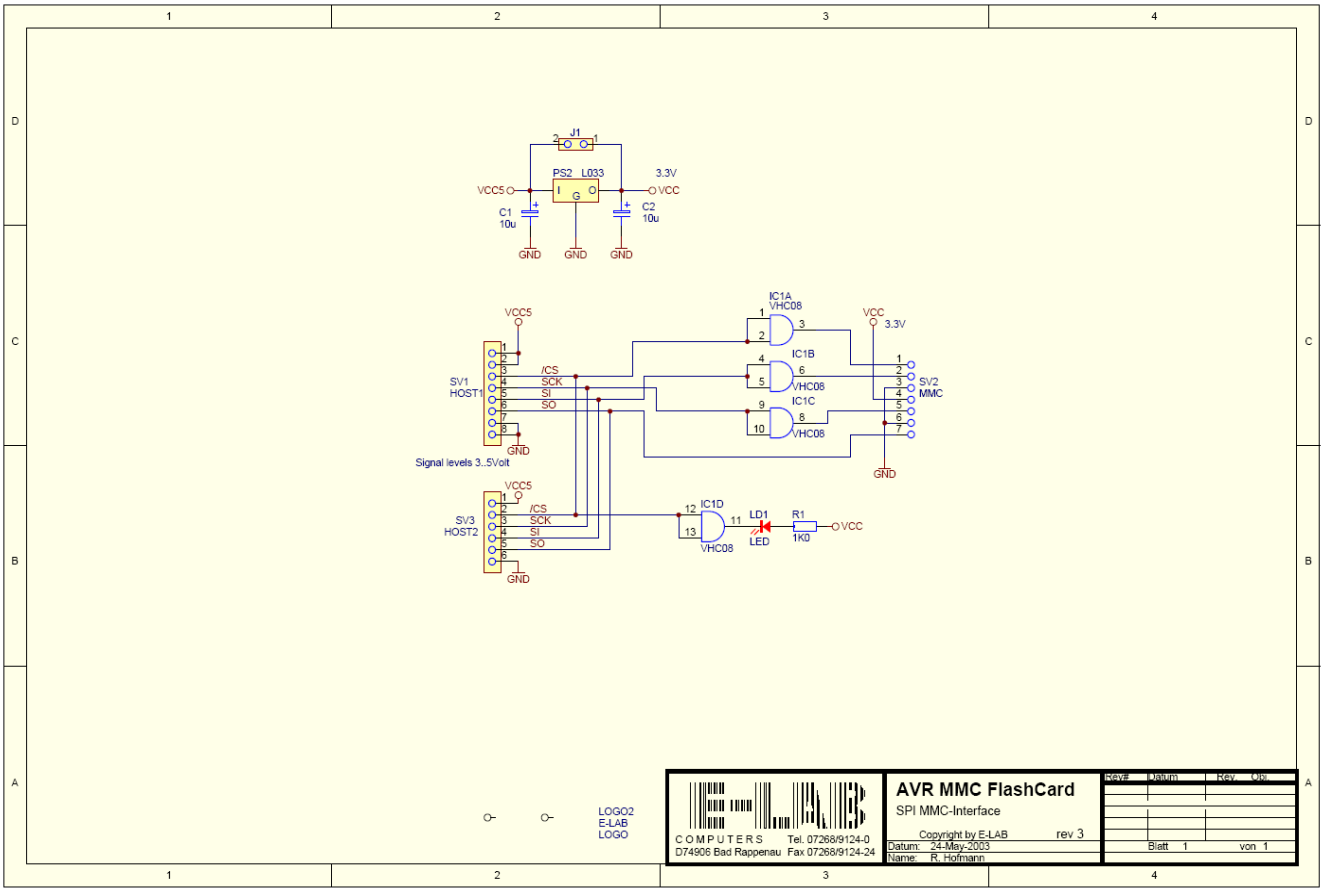
Simulator

Der AVRco Simulator SIM32 unterstützt das FileSystem komplett. Das Drive wird exakt im PC-Memory nachgebildet und gelesen und beschrieben. Damit ist ein hervorragende Testmöglichkeit gegeben, die Applikation betreffend dem FileSystem ausgiebig zu prüfen, ohne dass eine funktionierende Hardware zu Verfügung steht.

3.9.11 Programm Beispiele und Schaltpan

Im Verzeichnis **..\E-LAB\AVRco\Demos\FAT16test** befindet sich allgemeines Testprogramm mit dem alle Funktionen des FAT16 File Systems geprüft werden. Bitte beachten, dass hier auch illegale Operationen geprüft werden.

Im Verzeichnis **..\E-LAB\AVRco\Demos\FAT32test** befindet sich allgemeines Testprogramm mit dem alle Funktionen des FAT16_32 File Systems geprüft werden. Bitte beachten, dass hier auch illegale Operationen geprüft werden.



Schaltplan **MMC Interface**

3.10 wzNet EtherNet/InterNet Treiber AVRco NetStack

implementiert mit grossartigem Support von Udo Purwin

Es gibt eine Vielzahl Möglichkeiten, wie zwei Einheiten (Prozessoren, Steuerungen etc) miteinander kommunizieren können. Eine davon ist Ethernet. Der Begriff embedded Ethernet wird zur Zeit sehr viel benutzt, wenn es um schnelle Verbindungen zwischen mehreren Steuerungen geht oder um die PC<->Steuerung Anbindung. Hier wird sehr oft auch vom embedded TCP/IP gesprochen.

Mit der Lieferbarkeit von fast kompletten TCP/IP Stacks in einem IC stellt jetzt auch das TCP/IP Protokoll mit kleineren CPUs wie den AVR kein Problem mehr dar. Diese Implementation benutzt den W3100A Controller.

3.10.1 Architektur

Die zugrunde liegende Implementation dieses TCP/IP Stacks ist ein ***MultiTasking*** Kernel (wzKernel), der alle anfallenden Arbeiten auf der TCP/IP Ebene erledigt. Dieser Kernel läuft komplett im Hintergrund und wird zum grössten Teil von einem Spezial Task/Prozess (wzJobHandler) gebildet.

Der Vorteil ist eine absolute Transparenz des Kernels bei gleichzeitiger Unabhängigkeit der Applikation vom Kernel, zum Beispiel bleibt die Applikation nicht stehen, wenn der TCP/IP Treiber auf einen Timeout aufläuft, der ja bis zu 10sec dauern kann. Eine totale Blockade des Systems wie bei simplen Implementationen üblich, kann hier absolut nicht stattfinden.

Es können gleichzeitig bis zu 4 Sockets (Verbindungen) eröffnet werden, die alle unabhängig von einander arbeiten, und wo jeder einzelne ein beliebiges Protokoll fahren kann, sowohl im Server als auch im Client Mode.

Durch die Konstruktion des Kernels als separater Spezial Prozess/Task wirken sich auch vier aktive Sockets kaum auf die Gesamtleistung des Systems aus. Der maximale Durchsatz pro Socket ist auf ca. 20kByte/sec beschränkt. Dieser Durchsatz kann aber auch erreicht werden, wenn 4 Sockets gleichzeitig aktiv sind.

Die Zielsetzung für ein solches System beinhaltet immer zwei wesentliche Punkte, Geschwindigkeit und Sicherheit. In einer realen Applikation mit ihren Vorgaben (CPU, Speicher etc) muss hier immer ein Kompromiss eingegangen werden. Allein durch die limitierte Rechenpower des AVRs lässt sich nur ein mässiger Datendurchsatz erreichen, verglichen mit einem PC. Man muss also abwägen was wichtiger ist, high speed und damit nur noch geringe Rechenzeit für alle anderen Jobs der Applikation, oder mittlerer Durchsatz und damit mehr Rechenzeit für das restliche System. Wir haben uns für das zweite entschieden.

Wenn man das Ethernet als das betrachtet was es ist, nämlich nichts anderes als ein weiterer Kommunikations Kanal wie z.B. UART oder I2C, dann wird klar, eine High Speed Implementation wird hier zum Selbstzweck. Oder anders gesagt, ein MP3 Player beschäftigt den AVR zu 100% und es bleibt fast nichts mehr übrig für den Rest bzw. den eigentlichen Job.

Das gewählte Zeitscheiben Verfahren mit seiner sehr flexiblen Kopplung zwischen Applikation und Ethernet Prozess (wzJobHandler) beschränkt den Datendurchsatz ziemlich stark, hat aber den Vorteil, dass die vorhandene Rechenleistung optimal zwischen allen beteiligten Aufgaben verteilt werden kann. Eine Blockade kann nicht stattfinden, im Gegensatz zu hart gekoppelten Systemen, wo die Applikation direkt am Ethernet Treiber hängt.

Ein weiterer Punkt, der den Durchsatz in unserer Implementation zwar nicht wesentlich behindert, aber nach oben hin eine Grenze darstellt, ist dass der Standard Treiber über TWI/I2C arbeitet. Hier sind die 400kBit/sec eine echte Grenze. Eine mögliche Umstellung von TWI nach memory-mapped ist zwar vorgesehen und möglich, erhöht den Durchsatz aber nicht, da hier wiederum die Zeitscheibe bzw. der SysTick die Grenze bildet.

Trotzdem kann bei entsprechender Verteilung der Rechenzeit (Prioritäten des Haupt Prozess und des JobHandler) der Datendurchsatz in weiten Grenzen gesteuert werden.

Die erreichbaren Transferraten sind:

ca. 20 Packete/sec	Packetgrösse ca. 1.4kByte -> 28kByte/sec
CPU	16MHz
SysTick	5msec
JobHandler	Priorität high
Main/Main-Prozess	Priorität 5
Min 16kB Flash	min 1kB RAM

Imports

Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import SysTick, TWIMaster, wzNet4, ..;
```

Da der Treiber das TWI Interface benutzt, muss auch entweder der Treiber TWImaster oder TWInet importiert werden. Alternativ kann auch der Software I2C Treiber oder ein UserDevice benutzt werden.

Der im Hintergrund arbeitende Kernel benötigt auch den Import des MultiTasking Systems.

```
From System Import  Tasks, Processes;
```

Diese Imports erwarten nun diverse Defines.

Defines

```
Define
    ProcClock   = 16000000;          // Hertz
    SysTick     = 5;                 // msec
    StackSize   = $0040, iData;      // min size
    FrameSize   = $00C0, iData;      // min size
    Scheduler   = iData;
    TaskStack   = $0040, iData;      // min size
    TaskFrame   = $00C0;             // min size
    wzNet4      = I2C_TWI, iData;    // hardware I2C driver, var loc
    wzSocks     = 1;                 // socket count, 1..4
    TWIpresc    = TWI_BR400;         // max TWI speed
```

Software I2C Interface

```
Import  SysTick, I2Cport, wzNet4, ..;
From  System Import  Tasks, Processes;
Define
    ProcClock   = 16000000;          // Hertz
    SysTick     = 5;                 // msec
    StackSize   = $0040, iData;      // min size
    FrameSize   = $00C0, iData;      // min size
    Scheduler   = iData;
    TaskStack   = $0040, iData;      // min size
    TaskFrame   = $00C0;             // min size
    wzNet4      = I2C_Soft, iData;   // software I2C driver, var loc
    I2Cport     = PortA;
    I2Cclk      = 1;
    I2Cdat      = 2;
    wzSocks     = 1;                 // socket count, 1..4
```

UserDevice Interface

```
Import  SysTick, wzNet4, ..;
From  System Import  Tasks, Processes;
Define
    ProcClock   = 16000000;          // Hertz
    SysTick     = 5;                 // msec
    StackSize   = $0040, iData;      // min size
    FrameSize   = $00C0, iData;      // min size
    Scheduler   = iData;
    TaskStack   = $0040, iData;      // min size
    TaskFrame   = $00C0;             // min size
    wzNet4      = UserPort, iData;   // hardware driver, var loc
    wzSocks     = 1;                 // socket count, 1..4
```

Wird ein nicht-standard Treiber benutzt, also soll der W3100A Chip im Port oder Memory-mapped Modus betrieben werden, dann muss der User dies selbst tun, indem er untenstehende Funktion zur Verfügung stellt.
Das System ruft diese dann auf, um Daten und Steuerwerte vom/zum W3100A zu transportieren.

Der Parameter **doRead** bestimmt dabei ob gelesen oder geschrieben werden soll. Der Parameter **wzAddr** gibt eine relative Adresse innerhalb des W3100A an. **ptr** zeigt auf die Quelle bzw. das Ziel im RAM des AVR.
cnt gibt die Anzahl der zu transferierenden Bytes an. War die Operation erfolgreich muss die Funktion ein true zurückgeben, ansonsten ein false.

```
UserDevice wzNet_IOS (doRead : boolean; wzAddr : word; ptr : pointer; cnt : word) : boolean;
begin
  ...
  return(true);
end;
```

Beispiel memory mapped Treiber

```
Const wzBASE  : word = $8000;

UserDevice wzNet_IOS (doRead : boolean; wzAddr : word; ptr : pointer; cnt : word) : boolean;
begin
  if doRead then
    CopyBlock (pointer(wzAddr or wzBASE), ptr, cnt);
  else
    CopyBlock (ptr, Pointer(wzAddr or wzBASE), cnt);
  endif;
  return(true);
end;
```

Beispiel memory mapped Treiber im Indirekt Mode (Port)

```
const wzBASE  : word = $8000;

Procedure ind_mode_setup;
begin
  pbyte (BASE_WIZNET or $0C)^:= pbyte (BASE_WIZNET or $0C)^ or $82;
end;

UserDevice wzNet_IOS (doRead : boolean; wzAddr : word; ptr : pointer; cnt : word) : boolean;
var
   m : word;
begin
  if cnt > 1 then
    pbyte (BASE_WIZNET + $0C)^:= pbyte (BASE_WIZNET + $0C)^ or $01;
  endif;
  pword (BASE_WIZNET + $0D)^:= wzAddr;
  for m:=1 to cnt do
    if doRead then
      pbyte(ptr)^:= pbyte (BASE_WIZNET or $0F)^;
    else
      pbyte (BASE_WIZNET or $0F)^:= pbyte (ptr)^;
    endif;
    inc (ptr);
  endfor;

  if cnt > 1 then
    pbyte (BASE_WIZNET + $0C)^:= pbyte (BASE_WIZNET + $0C)^ and $FE;
  endif;
  return(true);
end;
```

Dieses Beispiel zeigt eine Implementation mit dem WizNet Chip im externen Speicher, wobei nur ein paar wenige Bytes benutzt werden. Eine Anschaltung nur durch die Ports des AVRs ist auch möglich. Dann muss der Treiber aber auch noch die Steuersignale RD/WR, CS und mehrere Adressen generieren.

3.10.1.1 Exportierte Typen und Konstanten

Der wzNet Treiber exportiert verschiedene Typ Deklarationen, die im Anwendungs Programm verwendet werden müssen:

```
Type  TMACaddr      = array[0..5] od byte;
Type  TIPaddr       = array[0..3] od byte;

Type  TwzStatus     = (wzsNoErrors, wzsInvalidHandle, wzsInitFailed, wzsNotInitialized, wzsSockClosed,
                       wzsBufferParam, wzsSendFailed, wzsTimeOutErr, wzsListenFailed,
                       wzsSockConnected, wzsSockListen, wzsSockCloseWait, wzsSockClosing,
                       wzsSockUDP, wzsSockRaw);

Type  TwzPriority   = (wzPrioLow, wzPrioMedium, wzPrioHigh, WzPrioVeryHigh, wzPrioAuto,
                       wzPrioSuspend, wzPrioResume);
```

```
Type TwzPacketReceive  =  Record
                            PeerIP       : tIPaddr;
                            PeerPort     : Word;
                            BufferPtr    : Pointer;
                            BufferLen    : Word;
                          end;

Type TwzSocketSWS     = (NoSillyWindow, SillyWindow);              // internal use

Type TwzSocketNDAck   = (NoDelayedAck, DelayedAck);                // internal use

Type  TwzNDTimeOut          = (NoDynamicTimeOut, DynamicTimeOut);      // internal use

Type  TwzBroadcast      = (NoBroadcast, Broadcast);                // internal use

Type TwzSocketProtocol = (CLOSED, protTCP, protUDP, protIPRAW, protMACRaw);

Type  TwzSocket  = Record
                     Protocol              : TwzSocketProtocol;
                     SWindow               : TwzSocketSWS;    // internal use
                     DelayAck              : TwzSocketNDAck;  // internal use
                     DynTimeOut            : TwzNDTimeOut;    // internal use
                     Broadcast             : TwzBroadcast;    // intern use
                     LocalPort             : Word;
                     RemoteHost            : tIPaddr;         // Client mode
                     RemotePort            : Word;            // Client mode
                     IPProtocol            : Byte;            // internal use
                     TypeOfService         : Byte;            // internal use
                     MaxSegSize            : Word;            // internal use
                     PeerTryToDisconnect   : Boolean;
                     SocketClosed          : Boolean;
                     PacketRecInfo         : TwzPacketReceive;
                     ErrorState            : TwzStatus;       // internal use
                     SocketState           : byte;            // semaphore
                   end;
Type tSocketHandle   = Pointer to twzSocket;
```

3.10.1.2 Exportierte Variablen

```
StructConst
  wzI2Cadd r      : Byte = $7F;

var
  TWI_DevLock : DEVICELOCK;
```

3.10.1.3 Exportierte Funktionen und Prozeduren

Setup

```
Procedure wzSetIPAddr (IPAddr, Mask : tIPaddr);
Procedure wzSetHWAddr (MacAddr : TMacAddr);
Procedure wzSetGatewayAddr (IPAddr : tIPaddr);
Procedure wzSetRetryCount (Retry : byte);
Procedure wzSetTimeOut (RetryTimeout : word);
Procedure wzReset;
Function  wzInit : boolean;
```

Operationen

```
Function wzCreateSocket : tSocketHandle;
Procedure wzFreeSocket (SocketPtr : tSocketHandle);
Function wzInitSocket (SocketPtr : tSocketHandle) : boolean;
Function wzReInitSocket (SocketPtr : tSocketHandle) : boolean;
Function wzConnect (SocketPtr : tSocketHandle) : Boolean;
Function wzDisConnect (SocketPtr : tSocketHandle) : boolean;
Function wzListen (SocketPtr : tSocketHandle) : boolean;
Function wzClientConnected (SocketPtr : tSocketHandle) : boolean;
Function wzAcceptConnection (SocketPtr : tSocketHandle; YesNo : boolean). boolean;
Function wzSendBuffer (SocketPtr : tSocketHandle;Buffer : pointer; Len : word): boolean;
Function wzReceiveBuffer(SocketPtr : tSocketHandle): word;
Function wzResumeReceive (SocketPtr : tSocketHandle) : boolean;
Function wzPacketReceived (SocketPtr : tSocketHandle) : boolean;
Function wzGetLastError (SocketPtr : tSocketHandle) : TwzStatus;
Function wzGetSocketState (SocketPtr : tSocketHandle) : TwzStatus;
```

Runtime Schalter

```
Procedure wzSetPriority (prio : TwzPriority);
```

Support Funktionen

```
Procedure STRtoIP (IPstr : String[15]; var Result : TIPAddress);

Function  IPtoSTR (IP : TIPAddress) : String[15];

Function  CompareNet (a1, a2, mask : TIPAddr ) : boolean;
```

3.10.2 Beschreibung der exportierten Typen, Konstanten und Funktionen

Ausführliche Beschreibung aller exportierten Typen, Konstante Variablen und Funktionen.

Typen
Der wzNet Treiber exportiert verschiedene Typ Deklarationen, die im Anwendungs Programm verwendet werden müssen:

```
Type TMACaddr  = array[0..5] of byte;
```
Das ist die sog. Hardware Adresse der Ethernet/Internet Einheit. Rein formal sollte diese weltweit nur einmal vorkommen. Da die meisten Systeme jedoch nur in Subnetzen arbeiten, muss i.A. darauf geachtet werden, dass diese Adresse zumindest in diesem Netzwerk nur einmal vorkommt.

```
Type TIPaddr  = array[0..3] of byte;
```
Das ist die logische Adresse des Netzwerk Knotens, auch lokale IP Adresse genannt. Wenn kein Server oder Gateway vorhanden ist, dann muss diese Adresse international sein. Das heisst eine Registrierungs Institution vergibt diese einmalige Adresse. In Subnetzen vergibt der Netzwerk Administrator diese IP Adresse.

```
Type TwzStatus =(wzsNoErrors, wzsInvalidHandle, wzsInitFailed, wzsNotInitialized, wzsSockClosed,
                 wzsBufferParam, wzsSendFailed, wzsTimeOutErr, wzsListenFailed,
                 wzsSockConnected, wzsSockListen, wzsSockCloseWait, wzsSockClosing,
                 wzsSockUDP, wzsSockRaw);
```

Die meisten Kernel Funktionen die fehlschlagen können, legen den resultierenden Status einer Operation in einem Status Byte vom Typ ***TwzStatus*** ab. Dieser Status kann zu jeder Zeit mit der Funktion ***wzGetLastError*** abgefragt werden. Der aktuelle Status eines Socket der neben den möglichen Fehlern auch den Status einer Verbindung enthält, kann mit der Funktion ***wzGetSocketState*** abgefragt werden***.***

```
Type TwzPriority = (wzPrioLow, wzPrioMedium, wzPrioHigh, wzPrioVeryHigh, wzPrioAuto,
                    wzPrioSuspend, wzPrioResume);
```
Je nach Aufgaben Stellung des Systems und dem aktuellen Zustand kann die Applikation die Rechenzeit Verteilung im System verändern. Die Rechenzeit des Kernels kann mit der Funktion ***wzSetPriority*** eingestellt werden. Damit kann der Datendurchsatz in weiten Grenzen gesteuert werden. Die Einstellung ***wzPrioVeryHigh*** sollte nur für extrem schnelle spezial Funktionen benutzt werden.

```
Type TwzPacketReceive  =   Record
                             PeerIP      : tIPaddr;
                             PeerPort    : Word;
                             BufferPtr   : Pointer;
                             BufferLen   : Word;
                           end;
```

Dies ist ein Record im Socket Record (SocketHandle^. PacketRecInfo). Die ersten zwei Parameter enthalten Daten, die im Server Mode einen connectierten Client indentifizieren. Der ***BufferPtr*** und ***BufferLen*** müssen von der Applikation gesetzt werden, so dass im Client oder im Server Mode Daten empfangen werden können.

Type *TwzSocketSWS = (NoSillyWindow, SillyWindow);*
Nur für internen Gebrauch.

Type *TwzSocketNDAck = (NoDelayedAck, DelayedAck);*
Nur für internen Gebrauch.

Type *TwzNDTimeOut = (NoDynamicTimeOut, DynamicTimeOut);*
Nur für internen Gebrauch.

Type *TwzBroadcast = (NoBroadcast, Broadcast);*
Nur für internen Gebrauch.

Type *TwzSocketProtocol = (CLOSED, protTCP, protUDP, protIPRAW, protMACRaw);*
Bestimmt die Art der Verbindung. Es wird nur TCP und UDP unterstützt

```
Type TwzSocket = Record
                   Protocol            : TwzSocketProtocol;
                   SWindow             : TwzSocketSWS;       // intern use
                   DelayAck            : TwzSocketNDAck;     // intern use
                   DynTimeOut          : TwzNDTimeOut;       // intern use
                   Broadcast           : TwzBroadcast;       // intern use
                   LocalPort           : word;
                   RemoteHost          : tIPaddr;            // Client mode
                   RemotePort          : word;               // Client mode
                   IPProtocol          : byte;               // intern use
                   TypeOfService       : byte;               // intern use
                   MaxSegSize          : word;               // intern use
                   PeerTryToDisconnect : boolean;
                   SocketClosed        : boolean;
                   PacketRecInfo       : TwzPacketReceive;
                   ErrorState          : TwzStatus;          // intern use
                   SocketState         : byte;               // semaphore
                 end;
Type tSocketHandle = Pointer to twzSocket;
```

Ein Socket enthält alle notwendigen Informationen die zum Aufbau einer Client oder Server Verbindung gebraucht werden. Die Applikation muss, nachdem sie mit **wzCreateSocket** einen Socket eröffnet hat, mit dem erhaltenen Handle ein paar wesentliche Parameter im Socket setzen:

```
SockHandle:= wzCreateSocket;                 // check if result = NIL !!
With SockHandle^ do
  Protocol:= protTCP;                        // desired protocol, protTCP or protUDP
  SWindow:= SillyWindow;                     // normal setting
  DelayAck:= NoDelayedAck;                   // normal setting
  DynTimeOut:= NoDynamicTimeOut;             // normal setting
  Broadcast:= NoBroadcast;                   // normal setting
  LocalPort:= LocPort;                       // current local port
  RemotePort:= RemPort;                      // Port#, only for client mode
  RemoteHost:= RemHost;                      // IP addr, only for client mode
  MaxSegSize:= 1460;                         // normal setting
  PacketRecInfo.BufferPtr:= @RxTxBuff;       // local RxBuffer
  PacketRecInfo.BufferLen:= SizeOf(RxTxBuff); // local RxBuffer size
endwith;
```

Das Protokoll bestimmt die Art der Verbindung, entweder TCP oder UDP

SWindow, ***DelayAck***, ***DynTimeOut***, ***Broadcast*** und ***MaxSegSize*** sind Parameter die immer so gesetzt werden müssen. Nur für extreme Spezialfälle wie MACraw und IPraw sind hier Ausnahmen zulässig.

Client Mode. ***LocalPort*** wird zwar nur für den Client Mode gebraucht, sollte aber auch im Server Mode immer mit einer zum Verbindungs Typ (HTTP, SMTP etc) passender Portnummer versehen sein.

Client Mode. ***RemotePort*** und ***RemoteHost*** sind die Server IP Adresse und das zugehörige Port für eine Client Verbindung zu einem Server.

Wenn Packete empfangen werden, speichert der Kernel mit jedem Aufruf von ***wzReceiveBuffer*** Daten in einen Buffer. Dieser Buffer muss für jeden aktiven Socket separat bereitgestellt werden. Die Applikation speichert die Adresse dieses Buffers im Parameter ***PacketRecInfo.BufferPtr*** ab. Die zugehörige Buffer Grösse wird im Parameter ***PacketRecInfo.BufferLen*** abgelegt.

Server Mode. ***SocketState*** sollte vom **Server** Prozess benutzt werden um auf einen Client zu warten, ohne kontinuierlich ***wzClientConnected*** aufrufen zu müssen.

procWaitFlag (SockHandle^.SocketState);

3.10.2.1 Exportierte Variablen

StructConst
wzI2CAddr : Byte = $7F;
Dieser Wert wird für alle Zugriffe auf die wzNet Hardware als I2C oder TWI Adresse benutzt. Diese Adresse kann von der Applikation beim Start-Up gegebenenfalls überschrieben werden.

var
TWI_DevLock : DEVICELOCK;
Im I2C oder TWI Modus kann die I2C/TWI Schnittstelle auch noch durch die Applikation selbst benutzt werden. Da es dabei mit absoluter Sicherheit zu konkurrierenden Zugriffen auf die Hardware kommt (User und Kernel), muss sicher gestellt werden, dass immer nur ein Programmteil zu einer Zeit einen Zugriff durchführt.

Dazu dient diese Semaphore vom Typ DeviceLock. Beim TWI Betrieb wird das Verriegeln des TWI Ports automatisch durch das System selbst durchgeführt. Die Applikation braucht sich darum nicht zu kümmern

3.10.2.2 Exportierte Funktionen und Prozeduren

Die meisten Kernel Funktionen die fehlschlagen können, legen den resultierenden Status einer Operation in einem Status Byte ab. Dieser Status kann zu jeder Zeit mit der Funktion ***wzGetLastError*** abgefragt werden.

Setup

Procedure *wzSetIPAddr (IPAddr, Mask : tIPaddr);*
Bestimmt die lokale IP-Adresse und die Subnet Mask dieses Stacks.

Procedure *wzSetHWAddr (MacAddr : TMacAddr);*
Bestimmt die MAC Adresse dieses Stacks.

Procedure *wzSetGatewayAddr (IPAddr : tIPaddr);*
Bestimmt die Gateway IP-Adresse für diesen Stack. Ein Gateway wird immer dann gebraucht, wenn das lokale Subnetz verlassen werden muss, z.B. um einen DNS oder SNTP Server zu kontaktieren. Im Normalfall ist der Router im eigenen Netzabschnitt das Gateway. Bei kleinen lokalen Netzen mit einem Server ist dieser lokale Server.

Procedure *wzSetRetryCount (Retry : byte);*
Bestimmt die Anzahl der Retries bei fehlgeschlagenen Sende Versuchen. Sollte auf 3 gesetzt werden.

Procedure *wzSetTimeOut(RetryTimeout : word);*
Bestimmt die Pause zwischen den Retries. Sollte auf 2000 gestellt werden.

Procedure *wzReset;*
Die Prozedur führt einen Software Reset auf dem W3100A Chip aus.

Function *wzInit : boolean;*
Führt einen Software Reset auf dem W3100A Chip aus und macht eine Grund-Initialisierung.

Operationen

Function *wzCreateSocket : tSocketHandle;*
Nach der Initialisierung des Systems erfolgen alle weiteren Operation mit Handles. Ohne ein gültiges Handle kann keine weitere Funktion erfolgreich durchgeführt werden. Je nach dem **Define *wzSocks*** können vom System bis zu 4 Handles vergeben und verwaltet werden.

Da jeder Socket/Handle einiges an Resourcen (RAM, CODE und CPU power) in Anspruch nimmt, sollte das Define entsprechend sparsam ausfallen. Wenn nur eine Verbindung zu einer Zeit geöffnet ist, wird auch nur ein Socket/Handle gebraucht.

Wenn diese Funktion fehlgeschlagen ist, weil z.B. alle Sockets/Handles schon vergeben sind, wird ein NIL-Pointer zurückgegeben.

Im Erfolgsfall zeigt das Handle (Pointer) auf eine Struktur vom Typ ***TwzSocket***. Diese Struktur ist fest an dieses Handle gebunden und wird dazu benutzt um Informationen und Status zwischen dem Kernel und der Applikation auszutauschen.

Procedure *wzFreeSocket (SocketPtr : tSocketHandle);*
Wenn eine Verbindung geschlossen wird und dieser Socket nicht mehr benötigt wird, sollte das Handle mit dieser Prozedur wieder freigegeben werden, so dass andere Teile der Applikation ebenfalls Verbindungen erstellen können.

Function *wzInitSocket (SocketPtr : tSocketHandle) : boolean;*
Nur für **UDP**. Grund Initialisierung des Sockets im UDP Mode. Wegen einem Bug im Wiznet Chip muss diese Funktion dann aufgerufen werden, wenn die Remote IP Adresse geändert/gewechselt wird.

Function *wzReInitSocket (SocketPtr : tSocketHandle) : boolean;*
Nur für **UDP**. Ähnlich wzInitSocket, aber nicht sehr nützlich wegen obigem Bug im Wiznet.

Function *wzConnect (SocketPtr : tSocketHandle) : boolean;*
Das ist die Funktion, mit der ein ***Client*** die Verbindung zu einem Server aufbaut. Bedingung für eine erfolgreiche Verbindung ist, dass diverse allgemeine globale Parameter (wzSetIPaddr etc) und Verbindungs spezifische Parameter (SocketHandle^.xxx) richtig initialisiert sind.

Function *wzDisConnect (SocketPtr : tSocketHandle) : boolean;*
Dient im **Server** und **Client** Mode dazu eine Verbindung abzubrechen.

Function *wzListen (SocketPtr : tSocketHandle) : boolean;*
Das ist die Funktion, mit der ein ***Server*** aktiviert wird, so dass sich ein Client verbinden kann. Bedingung für eine erfolgreiche Verbindung ist, dass diverse allgemeine globale Parameter (wzSetIPaddr etc) und Verbindungs spezifische Parameter (SocketHandle^.xxx) richtig initialisiert sind.

Function *wzClientConnected (SocketPtr : tSocketHandle) : boolean;*
Ein ***Server*** pollt solange diese Funktion bis ein true zurück kommt. In diesem Fall hat sich ein Client verbunden.

Die Peer-IP und das Peer-Port des Clienten kann mit SocketHandle^. PacketRecInfo.xxx festgestellt werden.

Function *wzAcceptConnection (SocketPtr : tSocketHandle; YesNo : boolean) : boolean;*
Hat sich im ***Server*** Mode ein Client connected, kann man diese Verbindung mit dieser Funktion akzeptieren oder abbrechen. Zum Beispiel kann mit

SocketHandle^. PacketRecInfo.PeerIP
und/oder
SocketHandle^. PacketRecInfo.PeerPort

der connectierte Client identifiziert werden.

Function *wzSendBuffer SocketPtr : tSocketHandle; Buffer : Pointer; Len : word): boolean;*
Das ist der Sende Auftrag an den Kernel. Wird vom ***Client*** als auch vom ***Server*** benutzt. Buffer ist ein Pointer, der auf die Quelle zeigt, Len ist die Anzahl der zu sendenden Bytes. Diese Funktion veranlasst den Kernel count Bytes aus Buffer^ zu lesen und in den internen txBuffer des W3100A zu schreiben.

Function *wzPacketReceived (SocketPtr : tSocketHandle) : boolean;*
Diese Funktion gibt ein true zurück wenn ein Packet empfangen wurde. Gilt sowohl im ***Server*** als auch im ***Client*** Modus.

Function *wzReceiveBuffer (SocketPtr : tSocketHandle): word;*
Erst wenn obige Funktion (wzPacketReceived) ein true zurückgegeben hat, dürfen Daten abgeholt werden. Das gilt für den ***Client*** und den ***Server*** Mode. Die Funktion veranlasst den Kernel Daten aus dem internen Buffer des W3100A auszulesen in den zugehörigen Applikations Buffer zu schreiben. Der Kernel transferiert bei jedem Aufruf Daten aus dem W3100A internen Buffer in den angegebenen Ziel-Buffer bis das W3100A interne Buffer leer ist.

Da ein empfangenes Packet bis zu 1460 Bytes gross sein kann, transferiert der Kernel immer nur soviel Bytes in den zuständigen Buffer

(SocketHandle^. PacketRecInfo. BufferPtr^)
wie in
(SocketHandle^. PacketRecInfo. BufferLen)

von der Applikation angegeben wurde.

Um jetzt das komplette Packet aus dem W3100A internen Buffer auszulesen, muss u.U. diese Funktion solange aufgerufen werden, bis das (evtl.) 1460 Byte grosse Packet komplett ausgelesen wurde und damit die Funktion 0 zurückgibt.

Die Applikation muss also grundsätzlich solange lesen (***wzReceiveBuffer***) bis die Funktion als Ergebnis eine 0 zurückgibt. Durch dieses Verhalten wird sicher gestellt dass auch ein grosses Packet komplett in kleine Buffer gelesen werden kann. Zwar in mehreren Teilen, aber doch komplett.

Function *wzResumeReceive (SocketPtr : tSocketHandle) : boolean;*
Um Overruns und ähnliche Probleme zu vermeiden, bleibt ein ***Server*** oder ***Client*** für weitere Empfangs Packete gesperrt, nachdem ein Packet hereingekommen ist. Die Applikation muss daher ihre Bereitschaft weitere Packete annehmen zu können durch den Aufruf dieser Funktion dem Kernel mitteilen.

Function *wzGetLastError (SocketPtr : tSocketHandle) : TwzStatus;*
Fast alle Kernel Funktionen setzen einen Errorstatus nach dem abarbeiten eines Auftrags. Funktionen die ein boolean zurückgeben haben in der Regel auch ein etwas ausführlicheren Fehlerstatus hinterlassen, der hiermit analysiert werden kann. Der Typ TwzStatus enthält weitere Zustände die nicht zu den Fehlern zählen und in dieser Funktion nicht berücksichtigt werden, z.B. wzsSockListen.

Function *wzGetSocketState (SocketPtr : tSocketHandle) : TwzStatus;*
Die weiter obenstehende sollte verwendet werden wenn ein Kernel Call mit einem false zurückkommt. Obige Funktion gibt immer ein ***wzsNoErrors*** zurück, wenn kein Fehler aufgetreten ist und gerade eine Operation läuft. Wenn der aktuelle Status eines Socket gebraucht wird, sollte die Funktion ***wzGetSocketState*** benutzt werden, da diese auch im nicht-fehler Fall einen Wert zurückgibt, der ausgewertet werden kann, zum Beispiel ***wzsSockListen***

Runtime Schalter

Procedure *wzSetPriority (prio : TwzPriority);*
Die Applikation kann jederzeit die Prioritäten ihrer eigenen Tasks und Prozesse bestimmen. In Verbindung mit dieser Funktion ist es möglich, das System zur Laufzeit sehr flexibel zu halten. Zum Beispiel macht es wenig Sinn, den Kernel mit wzPrioHigh laufen zu lassen, wenn absolut kein Socket geöffnet ist.

Bemerkung
Bei jeder TCP Verbindung hat man keine Gewähr, dass ein kleines Packet, das von einem Rechner geschickt wird, auch als kleines Packet ankommt. Wenn der Empfänger zu langsam reagiert oder die Verbindung selbst sehr langsam ist, packt der PC zum Beispiel solange kleine Packete in ein grosses, bis entweder der Empfänger wieder bereit ist, oder die physische Packet Grenze von 1460 Bytes erreicht sind.

Beispiel:
Eine PC Applikation sendet kontinuierlich mit grösserer Geschwindigkeit (Packet Rate) Daten an eine wzNet Hardware. Diese kann aus diversen Gründen aber nur die Hälfte der Packete pro Sekunde annehmen. Daher stauen sich die Packete im PC. Der bemerkt dieses und packt jetzt zwei Packete zusammen zu einem. Jetzt kann der Empfänger ein Packet annehmen und das Packet wird verschickt. Dann wiederholt sich das ganze. PC packt zwei Packete in eins und verschickt dies erc.

Wenn der Empfänger jetzt z.B. eine Datei erwartet (FTP) und diese speichern muss, ist das kein Problem. Das Packet wird solange gelesen und weggeschrieben, bis es leer ist. Dann kommt das nächste dran.

Erwartet der Empfänger immer ein Datenpacket mit einer festen Länge gibt es jetzt ein Problem. Auf der Empfängerseite muss jetzt sehr genau geprüft werden ob mehrere Packete in eins gepackt wurden. Dann muss das Ganze entsprechend aufgelöst werden.

Wenn es der gewünschte Datendurchsatz zulässt kann man das Packen auf der PC Seite verhindern, indem auf der wzNet Seite nach jedem Packet ein disconnect und connect gemacht wird. Auf der PC Seite muss die Software natürlich entsprechend agieren. Es kann auch ein Handshake nach jedem Packet zwischen wzNet und PC stattfinden.

Support Funktionen

Procedure *STRtoIP (IPstr : String[15];* ***var*** *Result : TIPAddress);*
Diese Funktion wandelt einen IP String, z.B. “192.168.1.16“ in eine IP Adresse um.

Function *IPtoSTR (IP : TIPAddress) : string[15];*
Das ist das Gegenstück zur obigen Funktion. Eine IP Adresse wird in einen String umgewandelt.

Function *CompareNet (a1, a2, mask : TIPAddr) : boolean;*
Diese Funktion vergleicht zwei IP Adressen unter Berücksichtigung einer Subnet Mask. Beim Compare werden nur die Bits bzw. Bytes betrachtet, die in der Maske 1 bzw. $FF sind. Bei Gleichheit wird ein true zurückgegeben.

3.10.3 Support Tools

Zur Zeit gibt es zwei Support Programme zum Testen und für den Betrieb des wzNet

3.10.3.1 TCPconf

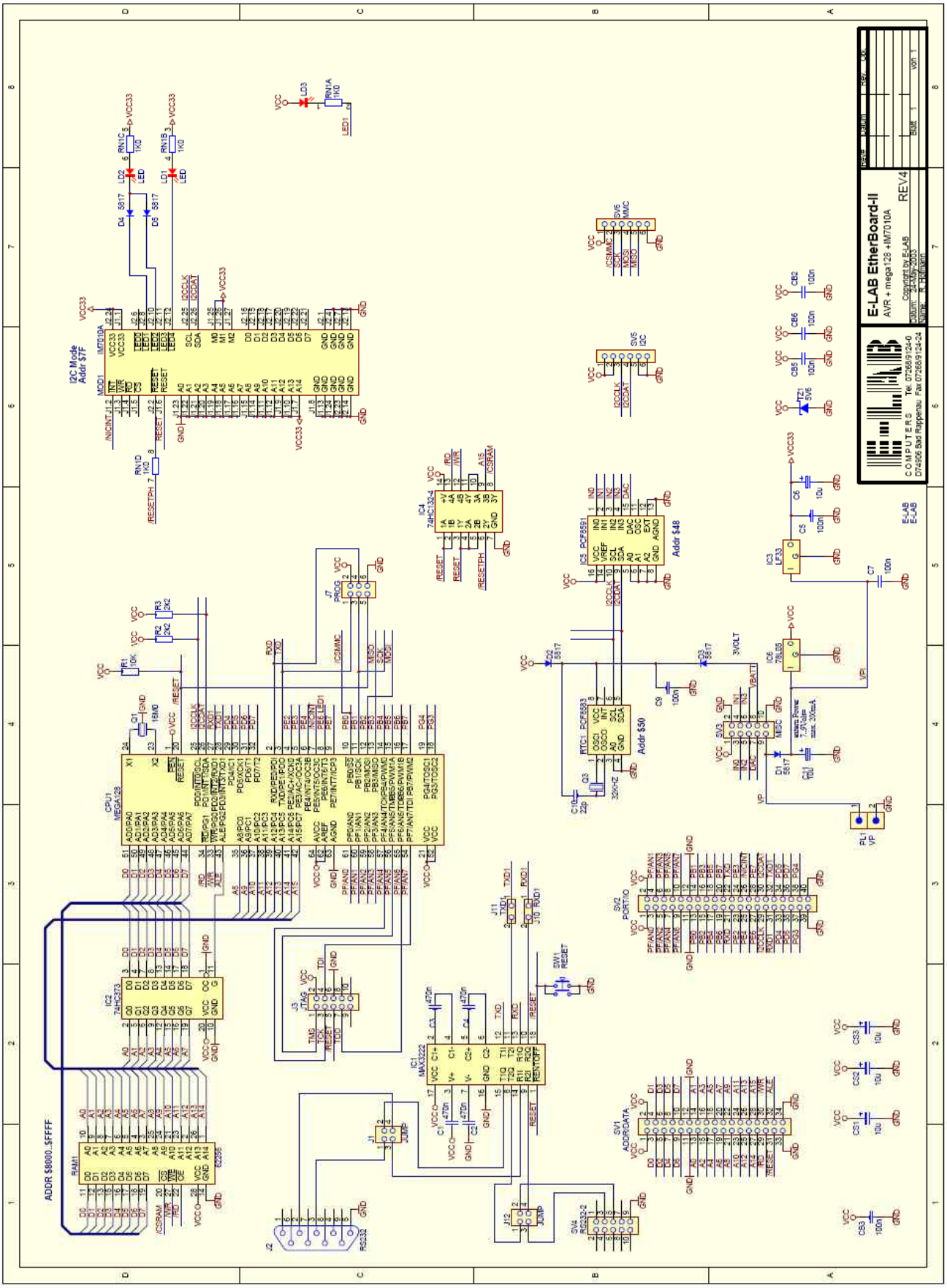
ist ein Programm, das aus der IDE PED32 heraus oder auch direkt aufgerufen werden kann. Es dient zur Inbetriebnahme einer W3100A Hardware und zur absolut notwendigen Konfiguration des Board Namens, der IP-Adresse und der MAC-Adresse

3.10.3.2 TCPcheck

ist ein umfangreiches Test Programm mit dem man eine wzNet Hardware ausgiebig und intensiv testen kann. Es kann aus der IDE PED32 heraus oder auch direkt aufgerufen werden.

Programm Beispiele und Schaltung:

Beispiel im Verzeichnis **..\E-LAB\AVRco\Demos\TCP_Serv**	ein kleines Server Projekt
Beispiel im Verzeichnis **..\E-LAB\AVRco\Demos\TCP_ServXXL**	ein Server Projekt mit grossem Buffer für Speed Tasks
Beispiel im Verzeichnis **..\E-LAB\AVRco\Demos\TCP_Client**	ein kleines Client Projekt



Schaltplan **EtherBoardII**

3.10.4 Telnet Server

Imports
Das Telnet Modul muss in die Applikation importiert werden:

```
Device = mega128, VCC=5;

Import SysTick, TWIMaster, wzNet4;

From wzNet4 import Telnet;

From System import  Tasks, Processes;

Define
    ProcClock     = 16000000;       {Hertz}
    SysTick       = 4;              {msec}
    StackSize     = $080, iData;
    FrameSize     = $080, iData;
    Scheduler     = 10,10,iData;    {use the IDLE process}
    TaskStack     = $80,iData;
    TaskFrame     = $80;

    wzNet4        = I2C_TWI, iData; {hardware I2C driver, var location}
    TWIpresc      = TWI_BR400;
    wzSocks       = 1;              {socket count, 1..4}
    TelnetStrLen  = 80;             // set work string length

Uses WzKernel, wzTelnet;
```

3.10.4.1 Exportierte Typen und Funktionen

```
type
  tTelnetStr = string[TelnetStrLen];

Procedure wzTelnetSetPort (port : word);
Function  wzTelnetCreate : boolean;
Function  wzTelnetListen : boolean;
Function  wzTelnetConnected : boolean;
Procedure wzTelnetClose;
Procedure wzTelnetFree;
Function  wzTelnetGetState : TwzStatus;
Procedure wzTelnetIdleTimeout (tnTimeout : byte);
Procedure wzTelnetEcho (EchoOn : boolean);

Function  wzTelnetWrite (Str : tTelnetStr) : boolean;
Function  wzTelnetWriteLn (Str : tTelnetStr) : boolean;
Function  wzTelnetRead (Prompt : tTelnetStr) : tTelnetStr;
Function  wzTelnetGetClient (var ClientIP : tIPaddr; var ClientPort : word) : boolean;
```

Procedure *wzTelnetSetPort (port : word);*
Diese optionale Funktion legt das Port für den Telnet Server fest. Default ist Port 23 eingestellt, was mit dieser Prozedur geändert werden kann. Eine Änderung muss allerdings vor dem Create stattfinden.

Function *wzTelnetCreate : boolean;*
Diese Funktion erstellt den Telnet Server. Sind alle verfügbaren Sockets belegt oder trat ein allgemeiner Fehler auf, wird ein false zurückgegeben. Der requirierte Socket muss ganz am Ende mit wzTelnetFree wieder freigegeben werden. Ein wzTelnetClose schliesst nur die Verbindung, der Socket bleibt jedoch belegt.

Function *wzTelnetListen : boolean;*
Nach dem Create wird normalerweise der Server in Listen Mode geschaltet.

Function *wzTelnetConnected : boolean;*
Wenn sich ein Client connected hat, geht der Server in den Connect Status. Nach dem Listen Auftrag sollte die Applikation kontinuierlich diese Funktion aufrufen, um zu sehen, ob ein Client da ist. Diese Funktion akzeptiert gleichzeitig ein Connect, ist also mehr als nur eine Status Abfrage.

Procedure *wzTelnetIdleTimeout (tnTimeout : byte);*
Ist ein Client connected, so kann die Verbindung nach einer gewissen Zeit der Inaktivität automatisch geschlossen werden. Das ist wichtig, da der Server bei einer bestehenden Verbindung auf neue Aufträge wartet und die Applikation sich normalerweise in der Funktion wzTelnetRead befindet, aus der sie nicht ohne Unterstützung herauskommt. Ein Client könnte also den Server für immer blockieren, solange er selbst keinen disconnect macht oder dem Server einen disconnect Auftrag erteilt. Der Parameter gibt den Idle-Timeout in Sekunden an. Der Wert 0 schaltet den Timeout komplett ab.

Procedure *wzTelnetClose;*
Die Applikation kann hiermit jederzeit den Server abschalten. Die erneute Bereitschaft muss dann wieder mit wzTelnetListen hergestellt werden. Diese Funktion gibt den Socket nicht frei.

Procedure *wzTelnetFree;*
Nach einem wzTelnetClose muss mit dieser Funktion der Socket freigegeben werden, wenn er nicht mehr benötigt wird. Ein erneutes Eröffnen muss dann mit wzTelnetCreate erfolgen.

Function *wzTelnetGetState : TwzStatus;*
Das ist eine reine Status Abfrage Funktion ohne Auswirkungen auf den Status des Servers. Der Typ TwzStatus wird weiter oben erklärt.

Procedure *wzTelnetEcho (EchoOn : boolean);*
Ein Telnet Server kann alle eingehenden Zeichen echoen. Das ist notwendig, wenn der Client sein eigenes Echo angeschaltet hat.

Function *wzTelnetWrite (Str : tTelnetStr) : boolean;*
Sende Auftrag an den Server. Der String wird an den Client geschickt.

Function *wzTelnetWriteLn (Str : tTelnetStr) : boolean;*
Identisch mit wzTelnetWrite, hängt aber zusätzlich ein CRLF an den String an.

Function *wzTelnetRead (Prompt : tTelnetStr) : tTelnetStr;*
Das ist die Haupt Funktion eines Telnet Servers, wenn sich ein Client connectiert hat. Das Prompt wird an den Client geschickt. Die Funktion wartet, bis der Client eine Antwort geschickt hat. Da das u.U. nie passieren wird, kann es hierbei zur Blockade des Servers kommen, da die Applikation keine weitere Kontrolle über den Server bekommt. Deshalb sollte immer der Telnet TimeOut Wert auf > 0 gesetzt werden.

Function *wzTelnetGetClient (**var** ClientIP : tIPaddr; **var** ClientPort : word) : boolean;*
Ist ein Client connectiert, kann zu jeder Zeit dessen IP-Adresse und Port abgefragt werden.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\TCP_Telnet**

3.10.5 DNS Client

Ein DNS Server setzt eine ihm namentlich übergebene Internet Adresse in eine IP Adresse um. Dazu wird ein DNS Client gebraucht, der diese Arbeit erledigt.

Imports

```
Device = mega128, VCC=5;

Import SysTick, TWIMaster, wzNet4;

From wzNet4 import DNS;

From System Import  Tasks, Processes;

Define
    ProcClock   = 16000000;         {Hertz}
    SysTick     = 4;                {msec}
    StackSize   = $080, iData;
    FrameSize   = $080, iData;
    Scheduler   = 10,10,iData;      {use the IDLE process}
    TaskStack   = $80,iData;
    TaskFrame   = $80;

    wzNet4      = I2C_TWI, iData;   {hardware I2C driver, var location}
    TWIpresc    = TWI_BR400;
    wzSocks     = 1;                {socket count, 1..4}

Uses WzKernel, wzDNS;
```

3.10.5.1 Exportierte Funktionen

```
Procedure wzSetDNSserver (IPAddr : tIPaddr);
```

Mit dieser Funktion wird einmalig die IP-Adresse eines DNS Servers gesetzt. Diese gilt bis zu einem System Neustart.

```
Function wzDNSQueryHost (Buffer : pointer; BuffLen : word; Hostname : pointer to string;
                         Var Result_IP : tIPaddr) : boolean;
```

Diese Funktion erstellt den DNS Clienten. Der Client kontaktiert den angegebenen DNS Server, übergibt diesem die Internet Adresse und erhält, falls vorhanden, von diesem die zugehörige IP-Adresse zurück. Der Client wird automatisch komplett gelöscht.

Buffer
ist ein Pointer der auf einen Speicherbereich mit mindestens 300 Bytes zeigen muss.

BuffLen
gibt die tatsächliche Grösse dieses Buffers an.

HostName
muss auf einen String im RAM zeigen, der die literale Internet Adresse enthält.

Result_IP
enthält, falls erfolgreich, die gewünschte IP Adresse

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\TCP_DNS**

3.10.6 SNTP Client

Ein SNTP Server gibt die aktuelle Zeit (GMT) und das Datum zurück. Dazu wird ein SNTP Client gebraucht, der diese Arbeit erledigt.

Imports

```
Device = mega128, VCC=5;

Import SysTick, TWIMaster, wzNet4;

From wzNet4 import SNTP;

From System import  Tasks, Processes;

Define
  …
  …

Uses WzKernel, wzSNTP;
```

3.10.6.1 Exportierte Typen und Funktionen

```
structconst
// value for Germany! Other time zones must overwrite this value
  TimeZoneBias   : integer = 2;     // hours
  TimeZoneBiasM  : byte = 0;     // minutes, optional minutes offset

type
  tDayOfWeek = (dwSunday, dwMonday, dwTuesday, dwWednesday, dwThursday, dwFriday,
                dwSaturday);

  tDateTime  = record
                 dayofWeek  : tDayOfWeek;
                 year       : word;
                 month      : byte;
                 day        : byte;
                 hour       : byte;
                 minute     : byte;
                 second     : byte;
               end;
```

Procedure wzSetSNTPserver (IPAddr : tIPaddr);
Mit dieser Funktion wird einmalig die IP-Adresse eines SNTP Servers gesetzt. Diese gilt bis zu einem System Neustart.

Function wzSNTPQueryDateTime (Buffer : pointer; BuffLen : word; var DateTime : tDateTime) : boolean;
Diese Funktion erstellt den SNTP Clienten. Der Client kontaktiert den angegebenen SNTP Server und erhält von diesem die Uhrzeit (GMT) und das Datum. Der Client wird automatisch komplett gelöscht.

Buffer
ist ein Pointer der auf einen Speicherbereich mit mindestens 60 Bytes zeigen muss.

BuffLen
gibt die tatsächliche Grösse dieses Buffers an.

DateTime
ist eine Variable, die von der Applikation bereitgestellt werden muss. Nach erfolgreicher Ausführung enthält diese die aktuelle Uhrzeit und das Datum. Der Wert der Konstante ***TimeZoneBias*** ist in der Berechnung schon enthalten.

TimeZoneBiasM ist eine optionale Konstante (initial Wert = 0) die in bestimmten Regionen gebraucht wird.

Programm Beispiel:

ein Beispiel befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\TCP_SNTP**

3.11 ModBus ASCII Serial Slave

implementiert mit grossartigem Support von Zeljko Avramovic ("AVRA")

Achtung:
ModBus Kommunikation ist ein komplexes Thema, das an dieser Stelle nur ganz kurz angesprochen werden kann. Für eine tiefere Einarbeitung ist das Studium der entsprechenden Literatur unumgänglich. Siehe dazu dazu auch die Links in den Fussnoten !

3.11.1 Einführung

In der heutig realen Welt arbeitet unser System nicht isoliert, sondern muss mit verschiedenen Geräten kommunizieren. Es ist zwar oft nur ein kleiner Baustein des Gesamtsystems, aber häufig ein besonders wichtiger. Deshalb ist es notwendig mit industriell eingesetzten Kommunikationsprotokollen kompatibel zu sein und Kompatibilität zu MODBUS [1] ist eine logische Folge:
er ist offen, weit verbreitet und gut dokumentiert.
Derzeit sind über 7 Millionen Modbus Geräte installiert (vor allem SPS [2] and Dummy „only read sensor“ Geräte).

Modbus ist ein "single master multi slaves request/reply" Protokol und nicht an einen physikalischen Layer gebunden. Größenteils werden RS232/485 (*ASCII* und *RTU* Varianten) und Ethernet (TCP/IP) Implementationen eingesetzt.

Da RTU einen Interrupt braucht und es das Ziel war, Ressourcen zu sparen ist ein MODBUS ASCII Slave als Hintergrund Prozess implementiert und digitale sowie analoge Ein-/Ausgänge benutzen den gleichen Adressraum (sind übereinander gemappt wie es in der Spezifikation empfohlen wird).

Vorteil von ASCII ist, daß man nur eine Terminal Software zum Testen braucht [3].

Alle wichtigen Modbus Funktionen sind verfügbar:

01 – Read Coils,
02 – Read Discrete Inputs,
03 – Read Holding Registers,
04 – Read Input Registers,
05 – Write Single Coil,
06 – Write Single Register,
15 – Write Multiple Coils, and
16 – Write Multiple Registers).

Wie man sieht, kennt Modbus nur Bit Übertragung (digitale Ein- und Ausgänge), sowie word Übertragung (analoge Ein- und Ausgänge). In der SPS Terminologie werden Bits als Coils und Words als Register bezeichnet.
Man kann andere Typen wie Integer, Float oder Strings jedoch in einem Server packen und diese dann im Client entpacken.
Lt. Spezifikation adressiert man digitale und analoge Ein- und Ausgänge. Aber üblicherweise ist dem nicht so. Es ist üblich, dass man ein intelligentes Modbus Slave Gerät hat, das mittels "tags" [4] den Zugriff auf einen gewissen Adressraum ermöglicht und diese dann nach jedem "scan" [5] auf die realen Ein- und Ausgänge gemappt werden.

[1] MODBUS – Offenes industrielles Kommunikationsprotokoll (www.modbus.org).
[2] SPS – Speicherprogrammierbare Steuerung = PLC - Programmable Logic Controller , ein Gerät, das digitale und analoge Eingänge lesen und dementsprechende Aktionen ausführen kann, z.B. digitale und analoge Ausgänge steuern. Siehe dazu www.plcs.net.
[3] Kommunikations Test Werkzeuge sind zu finden bei: www.ozm.cz/ivobauer/modlink, bray.velenje.cx/avr/terminal, www.wittecom.com, www.focus-sw.com/modpoll.html, www.jotmobile.com/plcpilot.html.
[4] TAG – Kleinster Datentyp eines Gerätes der gelesen oder geschrieben werden kann.
[5] SCAN – Die Software in SPS läuft in einer Endlosschleife. Ein Durchlauf wird "Scan" genannt.

Vor Jahren gab es noch ein grosses Durcheinander. Es gab Hunderte von verschiedenen Kommunikationsprotokollen und die Fähigkeiten einer HMI/SCADA [6] Software wurden daran gemessen, wieviele diese Protokolle sie unterstützte. Jeder SCADA Hersteller musste das Rad immer wieder neu erfinden und so haben verschiedene Teams an den selben Problemen gearbeitet und es wurde viel Energie vergeudet.
Es gab zwar Anstrengungen, die Treiber nur einmal zu schreiben und mit verschiedenen Applikationen zu benutzen, jedoch waren das keine sauberen Lösungen und deren Möglichkeiten waren beschränkt.
OLE [7] und DDE [8] sind schöne Beispiele dafür.
Das konnte nicht allzu lange gutgehen und die grossen Hersteller suchten eine Lösung.

Deshalb wurde OPC [9] erfunden. Es gab damit endlich eine universelle Art, mit SPS und anderen Geräten zu kommunizieren. Man braucht nur noch einen OPC Server, um Tags in ein Gerät zu schreiben aus von dem Gerät zu lesen. Ein kleines Software Modul reicht, um auf Alarme, Trends und historische Daten zuzugreifen. Ein weiterer Vorteil ist, dass man nicht mehr an einen bestimmten Hersteller gebunden ist.

Aber zurück zu unserem Treiber:
wenn wir nun einen OPC Server für den seriellen ASCII MODBUS haben (und da gibt es eine grosse Auswahl) sind wir mit unserem kleinen System im Geschäft.
Visualisierungen und Messungen werden zu einer Kleinigkeit.

Sie haben damit alles zusammen, um eine eigene SPS zu bauen. Statt eines isolierten Geräts steht Ihnen eine Verbindung zum Rest der Welt offen. Sie können mit anderen professionellen Geräten kommunizieren. Mit PCs zur Steuerung, Überwachung, Visualisierung und zur Datenerfassung. Im Raum nebenan oder auf der anderen Seite des Globus. Sie haben alle Werkzeuge [10] an der Hand und eine sofortige Kompatibilität zu MODBUS, OPC, SCADA, Delphi, Visual Basic, C++, C#, Java und dem Internet Explorer.

Mehr Informationen finden Sie in den Master Beispielen, die Sie von der E-LAB Seite downloaden können und unter den darin erwähnten Links

ModBus-ASCII-Masters.zip zu finden unter http://www.E-LAB.de/avrco/index.html

[6] HMI/SCADA – Human Machine Interface / Supervisory Control and Data Acquisition. Software zur Visualisierung, Überwachung, Datenerfassung und Steuerung industrieller Prozesse. Endbenutzer sind üblicherweise Bedienpersonal und Techniker in einem Produktionsumfeld. Weiterführende Links: www.citect.com, www.siemens.com/wincc, www.iconics.com, www.smartscada.com.

[7] OLE - Object Linking and Embedding (Microsoft).

[8] DDE - Dynamic Data Exchange (Microsoft).

[9] OPC - OLE for Process Control. Industriestandard der in Zusammenarbeit von Microsoft und einer Anzahl weltweit führender Hard- und Software Hersteller für Automatisierungssysteme entstand. Die Organisation, welche den Standard verwaltet, ist die OPC Foundation (www.opcfoundation.org). Basierend auf Microsoft's OLE (jetzt ActiveX), COM und DCOM Technologien besteht OPC aus einem Standard für Schnittstellen, Properties und Methoden die in der Prozess Steuerung und bei der Herstellung von Automatisierungsanwendungen benutzt werden.
Die ActiveX/COM Technologie beschreibt wie individuelle Software Komponenten interagieren und Daten austauschen. OPC stellt eine gemeinsame Schnittstelle zur Kommunikation mit diversen Prozess Steuerungs Geräten zur Verfügung unabhängig von der Steuerungssoftware und den Geräten im Prozess.
Obwohl es eine Reihe von OPC Spezifikationen gibt, sind die am häufigsten benutzten OPC DA (Data Access), OPC AE (Alarm and Events), und OPC HDA (Historical Data Access).

[10] Benutzer Werkzeuge: www.opcconnect.com, www.ozm.cz/ivobauer/modlink, www.pockethmi.com, www.signonline.de/dopc, www.abakus.vcl, www.iocomp.com, www.automatedsolutions.com, mbserver.w3.to www.smartscada.com, msol.io.com/ikysil/index-en.html, www.g7jjf.demon.co.uk/ppc.htm, www.prel.co.uk.

3.11.2 Implementation

Imports
Wie üblich muss der Treiber im AVRco importiert werden

```
Import SysTick, SerPort, ModBus;          //SerPort & SerPort2 are supported

{ $DEFINE MOD_RADIO}                      //important when RF is used

From System Import Processes;             //driver is implemented in process

From ModBus Import {MOD_Radio};           //if MODBUS is used on RF line

Define                                    //example for MEGA128
ProcClock     = 16000000;                 //Hertz
SysTick       = 10;                       //msec
StackSize     = $0020, iData;
FrameSize     = $0050, iData;
Scheduler     = iData;
SerPort       = 19200, Databit7, parEven, Stop1; //ASCII default
SerPortDTR    = PinB, 7, Positive;        //RF usually has busy signal
SerCtrl       = PortD, 2, Positive;       //control line if RS485 used
RxBuffer      = 255, iData;               //recommended, but may be lower
TxBuffer      = 100, iData;               //recommended, but may be lower
ModBus        = SerPort, 40;              //use port 1/2, capacity in words
ModBusMode = ASCII;                       //what modbus mode to use

Uses  ModBusServASCII;                    //modbus logic is in this unit

var
{$IDATA}

{$MODBUS filename}                        //build filename.pmbl to use with MODBUS Tester tool

  ModBuff[@ModDPR] :
              record                      //common memory for all modbus tags
                tag1 : mb_tag_type;
                tag2 : mb_tag_type;
                ...
                tagN : mb_tag_type;
              end;                        //do not exceed defined capacity in words!

// vordefinierte Tag Typen (mb_tag_type):
//
//   mb_InpB         = byte;              { !! always as couples }
//   mb_RdWrB        = byte;              { !! always as couples }
//   mb_InpW         = word;
//   mb_RdWrW        = word;              {prefix mb_Inp is used to identify}
//   mb_InpI         = Integer;           {that tag is read only, usually used}
//   mb_RdWrI        = Integer;           {for mapping input from some sensor,}
//   mb_InpW32       = longword;          {and prefix mb_RdWr is for read/write,}
//   mb_RdWrW32      = longword;          {usually used for physical outputs.}
//   mb_InpI32       = longInt;
//   mb_RdWrI32      = longInt;           {sufixes B, W, I, W32, I32 and F are}
//   mb_InpF         = Float;             {used to identify byte, word, int,}
//   mb_RdWrF        = Float;             {longword, longint, and float types}
//
//   zusätzliche Typen sind ein-dimensionale Arrays von Byte und BitSets.
//   BitSets werden als schreibbare "Coils" oder lesbare diskete Eingänge behandelt,
//   abhängig vom Namens Prefix "rw_" (alles mit diesem Prefix ist zu schreiben)
//   Bool'sche Variable werden indirekt in BitSets, Bytes, Coils oder diskreten Eingängen unterstützt.
```

Anschliessend kann man irgendwo im Programm (üblicherweise in der Main-Loop oder in einem Prozess Tag Werte ändern:

```
ModBuff.tagN:= NewValue;
```

Das ist schon alles. Sobald der ModBus Master diesen Wert abruft, wird er automatisch übertragen.

Jedes ModBus Gerät muss eine eindeutige Kennung (ID) haben. Standardmässig ist diese 1, kann aber leicht gelesen bzw. gesetzt werden, wie man in dem folgenden Beispiel sieht
(in einem ModBus Netzwerk mit vielen Geräten kann man zur Initialisierung zunächst mal DIP Schalter auswerten):

```
Procedure mb_SetModBusDevID(id: byte);        {set device ID}
Function mb_GetModBusDevID: byte;             {get device ID}
```

Alle ASCII ModBus Nachrichten enden mit <CR><LF>. Damit weiss jeder ModBus Teilnehmer wann eine Nachricht beender ist. Standardmässig wird aber nicht ewig auf das Ende einer Nachricht gewartet, sondern maximal 1000 msek bevor die gesamte Nachricht verworfen wird (lt. Spezifikation).
Glücklicherweise kann dieses Verhalten geändert werden, sodass man kein Weltmeister im Tippen am Terminal sein muss. Man kann eine beliebige Anzahl von msek einstellen oder eine Null um ewig zu warten.

Mit den folgenden Deklarationen sollte dies klar werden:

```
Procedure mb_SetModBusTimeout (time: word);   {set timeout in ms}
Function mb_GetModBusTimeout: word;           {get timeout in ms}
```

Die nächsten Deklarationen zeigen, dass man leicht Delphi ähnliche Events vor oder nach jedem Client Lesen/Schreiben des ModBus Speichers implementieren kann.

```
type
tBeforeRegisterRead   = Procedure (RegisterNumber: word);
tAfterRegisterRead    = Procedure (RegisterNumber: word);
tBeforeRegisterWrite  = Procedure (RegisterNumber: word; var NewValue: word);
tAfterRegisterWrite   = Procedure (RegisterNumber: word);
tBeforeCoilRead       = Procedure (CoilNumber: word);
tAfterCoilRead        = Procedure (CoilNumber: word);
tBeforeCoilWrite      = Procedure (CoilNumber: word; var NewValue: boolean);
tAfterCoilWrite       = Procedure (CoilNumber: word);
```

Prozeduren

```
procedure mb_setBeforeRegisterRead (proc: tBeforeRegisterRead);
procedure mb_setAfterRegisterRead (proc: tAfterRegisterRead);
procedure mb_setBeforeRegisterWrite (proc: tBeforeRegisterWrite);
procedure mb_setAfterRegisterWrite (proc: tAfterRegisterWrite);
procedure mb_setBeforeCoilRead (proc: tBeforeCoilRead);
procedure mb_setAfterCoilRead (proc: tAfterCoilRead);
procedure mb_setBeforeCoilWrite (proc: tBeforeCoilWrite);
procedure mb_setAfterCoilWrite (proc: tAfterCoilWrite);
```

Achtung:
Obenstehende Funktionen sind alles **Callbacks** die aus Interrupts heraus aufgerufen werden.
Bei Callbacks gilt grundsätzlich:
Keine weiteren Treiber oder Treiber Funktionen aufrufen. Keine zeitraubenden Operationen ausführen, vor allen Dingen keine Delays oder Waits auf weitere Erreignisse.
Weitere Interrupts würden zu lange gesperrt bleiben und z.B. UARTs würden wegen overrun Zeichen verlieren.

Somit können wir im Programm beispielsweise formulieren:

```
Procedure mb_BeforeRegisterRead (RegisterNumber: word);
begin
  // hier können Sie die Register bearbeiten bevor sie zum Client gesendet werden
  // oder auch nur einen Zähler inkrementieren
end;
```

und dann bei der Initialisierung den Pointer auf dieses Event setzen:

```
mb_SetBeforeRegisterRead (@mb_BeforeRegisterRead);      // enable event
```

dadurch wird unsere Prozedur aufgerufen bevor Tag gelesen wird durch den Client Request. Aber bitte beachten dass diese Callback Funktion sehr schnell sein muss ähnlich wie Antworten auf Interrupts.
Man sollte Flags, oder Pipes oder einen schnellen Memory Transfer in einem Callback benutzen.

Keinesfalls sollte zeit aufwendige Funktionen in Callbacks verwendet werden wie z.B. WriteLn oder andere langsame Operationen sonst riskiert man dass eine weitere ModBus Message hereinkommt bvor der aktuelle Callback erledigt ist. Und das kann fatale Konsequenzen für die Applikation haben.

Dennoch sollten Sie sich das folgende Beispiel ganz genau anschauen:

Program Beispiel:

ein Beispiel finden Sie im Verzeichnis **..\E-LAB\AVRco\Demos\ModBus_ASCII**

3.12 ModBus RTU Serial Slave

implementiert mit grossartigem Support von Zeljko Avramovic ("AVRA")

Achtung:
ModBus Kommunikation ist ein komplexes Thema, das an dieser Stelle nur ganz kurz angesprochen werden kann. Für eine tiefere Einarbeitung ist das Studium der entsprechenden Literatur unumgänglich. Siehe dazu dazu auch die Links in den Fussnoten und die Fussnoten im Kapitel MODBUS ASCII SERIAL SLAVE!

Um Wiederholungen zu vermeiden, wurde an dieser Stelle auf eine Einleitung verzichtet. Lesen Sie dazu „MODBUS ASCII SERIAL SLAVE“ und die offiziellen ModBus Dokumente[11].

Kurz gesagt überträgt dieser Treiber Datenblöcke an den "AVR Serial Slave" die aus word und boolean bestehen (andere Typen können allerdings in die words gepackt werden). Diese Daten werden in einer vom Master initiierten für uns transparenten Weise gelesen oder geschrieben.
Das einzige was zu tun ist, ist diese Datenblöcke zu lesen oder zu schreiben. Alle Änderungen sind automatisch für den Master und den Slave sichtbar.

In der ModBus Terminologie werden Slaves auch als Server bezeichnet (die auf die Nachrichten vom Master hören) und der Master als der Client (in einem ModBus Netzwerk kann es nur einen Master geben).

Um Speicher zu sparen folgten wir der Empfehlung der Spezifikation und überlagerten die zu lesenden und zu schreibenden Datenblöcke. Lt. ModBus sind zu lesende Datenblöcke immer digitale und analoge physikalische Eingänge. Dennoch kann man hier auch interne Variablen benutzen. Analog dazu sind zu schreibende Datenblöcke immer digitale und analoge physikalische Ausgänge.

Selbst digitale und analoge Signale sind überlagert. Beispielsweise wird für den digitalen Eingang 20 das gleiche Bit wie für den digitalen Ausgang 20 benutzt und diese sind dem zweiten analogen Eingang überlagert, der seinerseits das gleiche word für den zweiten analogen Ausgang benutzt.
Das kann manchmal ganz praktisch sein, aber im allgemeinen sollte es vermieden werden, digitale Signale mit analogen Signalen zu überlappen.

Da sich auch lesbare und zu schreibende Blöcke überlappen, sollte man auch sicherstellen, dass nicht sowohl Master als auch Slave den selben Speicherbereich eines Datenblocks schreiben. Die beste Art dies zu verhindern, ist read only Tags beginnend von 0 zu adressieren und für read/write Tags einen Offset zu benutzen, der größer als jedlicher read only Tag ist.

Der schnellste Weg die ModBus Kommunikation zu testen ist der *ModBus Tester*, der im Tools Menu der PED32 IDE zu finden ist.

Im Gegensatz zu ASCII ist RTU Ressourcen intensiv. Das exakte Timing benötigt einen Timer und RXRDY Interrupt. Der Vorteil ist, dass man in der gleichen Zeit doppelt so viele Daten übertragen kann.

[11] www.modbus-ida.org: Modbus_over_serial_line_V1.pdf, ModbusApplicationProtocol_v1_1.pdf

Da mehr Leistung benötiht wird, wurden im Vergleich zur ASCII Version mehr ModBus Funktionen implementiert.
Neben den auch im ASCII Slave implementierten Funktionen:

01 – Read Coils,
02 – Read Discrete Inputs,
03 – Read Holding Registers,
04 – Read Input Registers,
05 – Write Single Coil,
06 – Write Single Register,
15 – Write Multiple Coils
16 – Write Multiple Registers

gibt es auch die Funktionen

07 – Read Exception Status
(8 Bits die schön benutzt werden können um den Status des Geräts in standardisierter Form zu repräsentieren
08 – Diagnostics
(mit allen 15 Diagnose Funktionen. Für mehr Informationen siehe die offiziellen ModBus Dokumente).

Mit ModLink demo [12] ist es einfach möglich, die Diagnose Subfunktionen zu testen.

Natürlich sind auch OPC Server [13] für den serial RTU ModBus verfügbar.

Bei Benutzung von RS485 sollten Sie wissen, dass 2-Draht Verkabelung empfohlen wird
(4-Draht Verkabelung als Option) und RJ45 Verbinder (DB9 als Option).

Für zusätzliche Informationen sollten Sie das Dokument Modbus_over_serial_line_V1.pdf lesen.

[12] www.ozm.cz/ivobauer/modlink

[13] OPC - OLE for Process Control. Industriestandard der in Zusammenarbeit von Microsoft und einer Anzahl weltweit führender Hard- und Software Hersteller für Automatisierungssysteme entstand. Die Organisation, welche den Standard verwaltet, ist die OPC Foundation (www.opcfoundation.org). Basierend auf Microsoft's OLE (jetzt ActiveX), COM und DCOM Technologien besteht OPC aus einem Standard für Schnittstellen, Properties und Methoden die in der Prozess Steuerung und bei der Herstellung von Automatisierungsanwendungen benutzt werden. Die ActiveX/COM Technologie beschreibt wie individuelle Software Komponenten interagieren und Daten austauschen. OPC stellt eine gemeinsame Schnittstelle zur Kommunikation mit diversen Prozess Steuerungs Geräten zur Verfügung unabhängig von der Steuerungssoftware und den Geräten im Prozess.
Obwohl es eine Reihe von OPC Spezifikationen gibt, sind die am häufigsten benutzten OPC DA (Data Access), OPC AE (Alarm and Events), und OPC HDA (Historical Data Access).

3.12.1 Implementation

Imports
Wie üblich muss der Treiber im AVRco importiert werden

```
Import SerPort, ModBus;                          //SerPort & SerPort2 are supported

{ $DEFINE MOD_RADIO}                             //important only when RF is used

From System Import Processes;                    //driver needs processes
From ModBus import {MOD_Radio};                  //if MODBUS is used on RF line

Define                                           //example for MEGA128
ProcClock    = 16000000;
StackSize    = $0020, iData;
FrameSize    = $0050, iData;
Scheduler    = iData;
SerPort      = 19200, Databit8, parEven, Stop1; //RTU default
SerPortDTR = PinB, 7, Positive;                  //if RF has busy output signal
SerCtrl      = PortG, 3, Positive;               //control line if RS485 used
RxBuffer     = 255, iData;                       //recommended, but may be lower
TxBuffer     = 100, iData;                       //recommended, but may be lower
ModBus       = SerPort, 37, iData, 240;          //port, capacity of MODBUS data block in words,
                                                 //iData/xData, effective bytes (Framesize = effective bytes +
                                                 //function code + crc + EXTRA)
ModBusMode = RTU, Timer3;                        //MODBUS mode, timer 1..3

Uses ModBusServRTU;                              //MODBUS logic is in this unit

var
{$IDATA}

{$MODBUS filename}                               //build filename.pmbl to use with MODBUS Tester tool

  ModBuff[@ModDPR] :
              record                             //common memory for all modbus tags
                tag1 : mb_tag_type;
                tag2 : mb_tag_type;
                ...
                tagN : mb_tag_type;
              end;                               //do not exceed defined capacity in words!

// vordefinierte Tag Typen (mb_tag_type):
//
//  mb_InpB        = byte;                       { !! always as couples }
//  mb_RdWrB       = byte;                       { !! always as couples }
//  mb_InpW        = word;
//  mb_RdWrW       = word;                       {prefix mb_Inp is used to identify}
//  mb_InpI        = Integer;                    {that tag is read only, usually used}
//  mb_RdWrI       = Integer;                    {for mapping input from some sensor,}
//  mb_InpW32      = longword;                   {and prefix mb_RdWr is for read/write,}
//  mb_RdWrW32     = longword;                   {usually used for physical outputs.}
//  mb_InpI32      = longInt;
//  mb_RdWrI32     = longInt;                    {sufixes B, W, I, W32, I32 and F are}
//  mb_InpF        = Float;                      {used to identify byte, word, int,}
//  mb_RdWrF       = Float;                      {longword, longint, and float types}
```

```
//
//   zusätzliche Typen sind ein-dimensionale Arrays von Byte und BitSets.
//   BitSets werden als schreibbare "Coils" oder lesbare diskete Eingänge behandelt,
//   abhängig vom Namens Prefix "rw_" (alles mit diesem Prefix ist zu schreiben)
//   Bool'sche Variable werden indirekt in BitSets, Bytes, Coils oder diskreten Eingängen unterstützt.
```

Anschliessend kann man irgendwo im Programm (üblicherweise in der Main-Loop oder in einem Prozess Tag Werte ändern:

```
ModBuff.tagN:= NewValue;
```

Das ist schon alles. Sobald der ModBus Master diesen Wert abruft, wird er automatisch übertragen.

Jedes ModBus Gerät muss eine eindeutige Kennung (ID) haben. Standardmässig ist diese 1, kann aber leicht gelesen bzw. gesetzt werden, wie man in dem folgenden Beispiel sieht
(in einem ModBus Netzwerk mit vielen Geräten kann man zur Initialisierung zunächst mal DIP Schalter auswerten):

```
Procedure mb_SetModBusDevID (id: byte);          {set device ID}
Function mb_GetModBusDevID: byte;                {get device ID}
```

In der RTU Version vom ModBus ist Funktion 7 implementiert (Read Exception Status). Dies kann sehr praktisch sein, um den Status des Gerätes darzustellen.
Die Spezifikation legt keine Bits (sogenannte "exception status outputs") im Exception Status byte fest, sodaß Sie die frei benutzen können.
Den folgenden Beispielen können Sie entnehmen, wie man den Exception Status eines Geräts lesen/setzen lann:

```
Procedure mb_SetModBusExceptionStatus (status: byte);     {set status}
Function mb_GetModBusExceptionStatus: byte;               {get status}
```

Die nächsten Deklarationen zeigen, dass man leicht Delphi ähnliche Events vor oder nach jedem Client Lesen/Schreiben des ModBus Speichers implementieren kann.

```
type
  tBeforeRegisterRead  = Procedure (RegisterNumber: word);
  tAfterRegisterRead   = Procedure (RegisterNumber: word);
  tBeforeRegisterWrite = Procedure (RegisterNumber: word; var NewValue: word);
  tAfterRegisterWrite  = Procedure (RegisterNumber: word);
  tBeforeCoilRead      = Procedure (CoilNumber: word);
  tAfterCoilRead       = Procedure (CoilNumber: word);
  tBeforeCoilWrite     = Procedure (CoilNumber: word; var NewValue: boolean);
  tAfterCoilWrite      = Procedure (CoilNumber: word);
```

Prozeduren

```
  Procedure mb_setBeforeRegisterRead (proc: tBeforeRegisterRead);
  Procedure mb_setAfterRegisterRead (proc: tAfterRegisterRead);
  Procedure mb_setBeforeRegisterWrite (proc: tBeforeRegisterWrite);
  Procedure mb_setAfterRegisterWrite (proc: tAfterRegisterWrite);
  Procedure mb_setBeforeCoilRead (proc: tBeforeCoilRead);
  Procedure mb_setAfterCoilRead (proc: tAfterCoilRead);
  Procedure mb_setBeforeCoilWrite (proc: tBeforeCoilWrite);
  Procedure mb_setAfterCoilWrite (proc: tAfterCoilWrite);
```

Somit können wir im Programm beispielsweise formulieren:

```
Procedure mb_BeforeRegisterRead (RegisterNumber: word);
begin
  // hier können Sie die Register bearbeiten bevor sie zum Client gesendet werden
  // oder auch nur einen Zähler inkrementieren
end;
```

und dann bei der Initialisierung den Pointer auf dieses Event setzen:

```
mb_SetBeforeRegisterRead (@mb_BeforeRegisterRead);      // enable event
```

Damit wird unsere Prozedur automatisch vor jedem Lesen des Clients aufgerufen.

Dennoch sollten Sie sich das folgende Beispiel ganz genau anschauen:

Program Beispiel:

ein Beispiel finden Sie im Verzeichnis **..\E-LAB\AVRco\Demos\ModBus_RTU**

3.13 TINA EtherNet/InterNet Treiber AVRco NetStack

Implementiert mit grossartigem Support von Udo Purwin

Es gibt eine Vielzahl Möglichkeiten, wie zwei Einheiten (Prozessoren, Steuerungen etc) miteinander kommunizieren können. Eine davon ist Ethernet. Der Begriff embedded Ethernet wird zur Zeit sehr viel benutzt, wenn es um schnelle Verbindungen zwischen mehreren Steuerungen geht oder um die PC<->Steuerung Anbindung.

Ethernet spezifiziert die physikalische Verbindung zwischen zwei Teilnehmern.
Beschrieben werden Kabeltypen, Zugriffsmechnanismus/Kollisions-Auflösung, etc. sowie das low-level Telegramm Format (MAC-Addressen). Somit ist Ethernet auf einer Ebene mit z.B. dem CAN-Bus zu sehen.

Da Ethernet (und IP) schon vor der Einführung des OSI-Modell definiert wurde, basiert es auf dem (4-Schichten) Modell des DoD (Department of Defence). Im OSI-Modell ist Ethernet deshalb nicht eindeutig zu platzieren, sondern sowohl im "DataLink-Layer" (MAC-Adressen, CRC etc.), als auch im "Physical Layer" (Stichworte: UTP, RG58, CSMA/CD, etc.) anzusiedeln.

"Ethernet" wird heutzutage aber oft oft als Synonym für die gesamte IP-Protokoll Familie benutzt.
"FTP", "SNMP", "TCP", "UDP", "ICMP", "ARP","BOOTP" ... - alle sind häufig mit "Ethernet" gemeint.

Eine Applikation (eine eigene oder Applikationen wie z.B. FTP oder PING) kommuniziert jedoch nur mit dem OSI "Transport Layer", also TCP/IP und/oder UDP/IP (bzw. dem analogen DoD Layer: "Host-to-Host").
Die darunter liegenden low-level und physikalischen Schichten müssen durch den Treiber abgedeckt werden.
Das ist durch den Treiber

T**ransportation**I**ndependend**N**etwork**A**ccess* ***TINA

angegangen worden.
Ziel dieses Treibers ist es alle UDP Protokolle der Applikation zur Verfügung zu stellen, unabhängig von dem Leitungstreiber (MAC etc). Dazu kommen auf der Leitungs Seite die Anbindung des ENC28J60, des ENC424J600, eines UART (SLIP) und eines User Devices (generic).

Auf der Protokoll Seite wird nur UDP, xUDP und ICMP-PING unterstützt. Auf der Leitungsseite sind die MicroChip ENC28J60 und ENC424J600 implementiert.

3.13.1 Architektur

Die zugrunde liegende Implementation dieses Stacks ist ein ***MultiTasking*** Kernel (Unit Tina), der alle anfallenden Arbeiten auf der Protokoll Ebene erledigt. Dieser Kernel läuft komplett im Hintergrund und wird zum grössten Teil von einem Spezial Task (TINA_Job) gebildet.

Der Vorteil ist eine absolute Transparenz des Kernels bei gleichzeitiger Unabhängigkeit der Applikation vom Kernel, zum Beispiel bleibt die Applikation nicht stehen, wenn der STACK Treiber auf einen Timeout aufläuft, der ja bis zu 10sec dauern kann. Eine totale Blockade des Systems, wie bei simplen Implementationen üblich, kann hier absolut nicht stattfinden.

Es können gleichzeitig bis zu 8 Sockets (Verbindungen) eröffnet werden, die alle unabhängig voneinander arbeiten, und wo jeder einzelne ein beliebiges Protokoll fahren kann, sowohl im Server als auch im Client Mode.

Durch die Konstruktion des Kernels als separater Spezial Task wirken sich auch acht aktive Sockets wenig auf die Gesamtleistung des Systems aus. Der maximale Durchsatz pro Socket ist auf ca. 100kByte/sec beschränkt (ENCxxx), abhängig von der Leistung des Hardware Interfaces. Dieser Durchsatz kann aber auch erreicht werden, wenn mehrere Sockets gleichzeitig aktiv sind.

Die Zielsetzung für ein solches System beinhaltet immer zwei wesentliche Punkte: Geschwindigkeit und Sicherheit. In einer realen Applikation mit ihren Vorgaben (CPU, Speicher etc) muss hier immer ein Kompromiss eingegangen werden. Allein durch die limitierte Rechenpower des AVRs lässt sich nur ein mässiger Daten-durchsatz erreichen, verglichen mit einem PC. Man muss also abwägen was wichtiger ist, high speed und damit

nur noch geringe Rechenzeit für alle anderen Jobs der Applikation, oder mittlerer Durchsatz und damit mehr Rechenzeit für das restliche System. Wir haben uns für das zweite entschieden.

Wenn man das Ethernet als das betrachtet was es ist, nämlich nichts anderes als ein weiterer Kommunikations Kanal wie z.B. UART oder I2C, dann wird klar, eine High Speed Implementation wird hier zum Selbstzweck. Oder anders gesagt, ein MP3 Player beschäftigt den AVR zu 100% und es bleibt fast nichts mehr übrig für den Rest bzw. den eigentlichen Job.

Das gewählte Zeitscheiben Verfahren mit seiner sehr flexiblen Kopplung zwischen Applikation und Ethernet Prozess (TINA_Job) beschränkt den Datendurchsatz ziemlich stark, hat aber den Vorteil, dass die vorhandene Rechenleistung optimal zwischen allen beteiligten Aufgaben verteilt werden kann. Eine Blockade kann nicht stattfinden, im Gegensatz zu hart gekoppelten Systemen, wo die Applikation direkt am Ethernet Treiber hängt.

Ein weiterer Punkt, der den Durchsatz in unserer Implementation zwar nicht wesentlich behindert, aber nach oben hin eine Grenze darstellt, ist dass der Standard Treiber über SPI arbeitet. Hier sind die 8MBit/sec bei SPI/ENC eine echte Grenze. Eine mögliche Erhöhung dieser Bitrate ist zwar denkbar, erhöht den Durchsatz aber nicht wesentlich, da hier wiederum die Zeitscheibe bzw. der SysTick die Grenze bildet.

Trotzdem kann bei entsprechender Verteilung der Rechenzeit (Priorität des TINA JobHandlers) der Daten-durchsatz in weiten Grenzen gesteuert werden.

Die erreichbaren Transferraten sind:

CPU	16MHz		
JobHandler	Priorität high		
Min 16kB Flash	min 2kB RAM		
ENCxxxx with SPI			
Packetgrösse	20Byte	ca. 500 Packete/sec	-> 9kByte/sec ***xUDP***
Packetgrösse	1kByte	ca. 110 Packete/sec	-> 110kByte/sec ***xUDP***
W3100A with TWI			
Packetgrösse	1.4kByte	ca. 20 Packete/sec	-> 28kByte/sec ***TCP/IP***

3.13.1.1 Imports

Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import SysTick, TINAstack, ...;
```

Da der TINA-ENC Treiber ein SPI Interface benutzt, muss dieses angegeben werden, das SPI Port ist daher für andere Teile nur mit entspr. Vorkehrungen nutzbar. Bei den grossen Megas können auch die MSPI Ports benutzt werden. Bei den XMegas können die 4 möglichen SPIs benutzt werden: SPI_C, SPI_D etc.

Der im Hintergrund arbeitende TINA Kernel benötigt kein MultiTasking, aber es macht auch hier Sinn für jeden Socket einen Prozess zu definieren. Grundsätzlich benötigt der TINA Kernel eine Hardware Timer, da er absolut unabhängig vom SysTick arbeitet. Timer1, Timer2 oder Timer3 können benutzt werden, wenn vorhanden. Bei den XMegas muss einer der möglichen 8 Timer benutzt werden: Timer_C0, Timer_C1 etc.

Bemerkung:
Die zur Zeit erhältlichen ENC Chips brauchen eine SPI Daten Rate von mindestens 8Mbit/sec. Um das zu erreichen muss die CPU mindestens mit 16MHz laufen.

Diese Imports erwarten nun diverse Defines.

3.13.1.2 Defines

```
Define
    ProcClock     = 16000000;              // Hertz
    StackSize     = $0080, iData;          // min size
    FrameSize     = $00C0, iData;          // min size
    xData         = $8000, $87ff;          // 2kB optional, only necessary with the xData define below
    TINAdriver    = ENC28J60[, xData];     // TINA hardware,optional buffers in xData
    // TINAdriver = ENC424J600;            // TINA hardware, alternative chip, WizNet, WizNet5200
    TINAport      = SPI, PortB, 0;         // SPItyp, SS_Port, SS_Pin
    // TINAport   = MSPI0, PortA, 4;       // SPItyp, SS_Port, SS_Pin
    // TINAport   = SPI_Soft, PortD.7, PortD.6, PortD.5, PortD.4;  // SPItyp, SS, SCLK, MOSI, MISO
    TINAtimer     = Timer3;                // 1..3
    TINAsockets   = 4;                     // socket count, 1..8
```

XMega

Mit den XMegas werden bis zu vier SPI-ports unterstützt:

SPI_C, SPI_D, SPI_E or SPI_F

Deshalb muss hier der SPI Port definiert werden:

```
TINAport      = SPI_C, PortF, Pin3;
```

Die XMegas bieten bis zu 8 Timer:

Timer_C0, Timer_C1, Timer_D0, Timer_D1 etc.

Einer von diesen muss benutzt werden:

```
TINAtimer     = Timer_F1;
```

3.13.1.3 Exportierte Typen und Konstanten

Der TINA Treiber exportiert verschiedene Typ Deklarationen, die im Anwendungs Programm verwendet werden.

```
Type
  TMACaddr          = array[0..5] of byte;

Type
  TIPaddr           = array[0..3] of byte;

Type
  TTINAStatus       = (TinasNoErrors, TinasInvalidHandle, TinasInitFailed, TinasNotInitialized,
                       TinasSockClosed, TinasBufferParam, TinasSendFailed, TinasTimeOutErr,
                       TinasListenFailed, TinasSockConnected, TinasSockListen,
                       TinasSockCloseWait, TinasSockClosing, TinasSockUDP, TinasSockRaw);

Type
  TTinaPriority     = (TinaPrioMedium, TinaPrioLow, TinaPrioHigh, TinaPrioVeryHigh,
                       TinaPrioVeryLow, TinaPrioAuto, TinaPrioSuspend, TinaPrioResume);

Type
  TtinaPacketReceive = Record
                         PeerIP     : tIPaddr;
                         PeerPort   : Word;
                         RecLen     : Word;
                         BufferPtr  : Pointer;
                         BufferLen  : Word;
                       end;

Type
  TTinaSocketSWS    = (NoSillyWindow, SillyWindow);
  TTinaSocketNDAck  = (NoDelayedAck, DelayedAck);
  TTinaNDTimeOut    = (NoDynamicTimeOut, DynamicTimeOut);
  TTinaBroadcast    = (NoBroadcast, Broadcast);
  TTinaxUDPAKNPort  = (xAKNLocalPort, xAKNRemotePort, xAKNRemoteDynamicPort);

Type
  TProtocolType     = (pICMP, pUDP, pTCP, pxUDP);
```

```
Type
  TTinaSocket        = Record
                         Protocol            : TProtocolType;
                         Swindow             : TTinaSocketSWS;
                         DelayAck            : TTinaSocketNDAck;
                         DynTimeOut          : TTinaNDTimeOut;
                         Broadcast           : TTinaBroadcast;
                         LocalPort           : Word;
                         RemoteHost          : tIPaddr;
                         RemotePort          : Word;
                         TimeOut             : LongWord;
                         RetryCount          : Byte;
                         AKNPort             : TTinaxUDPAKNPort;
                         PeerTryToDisconnect : Boolean;
                         SocketClosed        : Boolean;
                         PacketRecInfo       : TTinaPacketReceive;
                         ErrorState          : TTINAStatus;
                         SocketState         : Byte;
                       end;

  tSocketHandle      = Pointer to TTinaSocket;

  TTinaCore          = Record
                         IP                  : tIPaddr;
                         Mask                : tIPaddr;
                         Gateway             : tIPaddr;
                         Mac                 : TMacAddr;
                         TimeOut             : LongWord;
                         Retry               : Byte;
                         Prio                : TTinaPriority;
                         ResponsePing        : Boolean;
                         RXCheckSumCheck     : Boolean;
                         SendICMPCtrlMessages : Boolean;
                       end;
```

3.13.1.4 Exportierte Variablen

```
var
  TinaCore           : TTinaCore;
```

3.13.1.5 Exportierte Funktionen und Prozeduren

Setup

```
Function   TINA_Init : Boolean;
Function   TinaLinkStat : boolean;
Procedure  TINA_Start;
Procedure  TINA_Stop;
```

Operationen

```
Function   GetDefaultMAC: tMACaddr;          // ENC424J600 only
Function   TinaInitSocket(SocketPtr : tSocketHandle) : Boolean;
Function   TinaCreateSocket : tSocketHandle;
Function   TinaRxStat(SocketPtr : tSocketHandle) : Boolean;
Function   TinaPacketReceived (SocketPtr : tSocketHandle) : Boolean;
Function   TinaResumeReceive (SocketPtr : tSocketHandle) : Boolean;
Function   TinaSendPacket (SocketPtr : tSocketHandle; Buffer : Pointer; Len : Word) : Boolean;
Procedure  TinaFreeSocket (SocketPtr : tSocketHandle);
```

Runtime Schalter
Procedure *TINASetPriority (prio : TTinaPriority);*

Support Funktionen
Function *TINA_Ping (PingAdr : TIPAddr; TimeOut : Word) : Word;*
Procedure *STRtoIP (IPstr : String[15];* ***var*** *Result : TIPAddress);*
Function *IPtoSTR (IP : TIPAddress) : String[15];*
Function *CompareNet (a1, a2, mask : TIPAddr) : boolean;*

3.13.2 Beschreibung der exportierten Typen, Konstanten und Funktionen

Typen
Der TINA Treiber exportiert verschiedene Typ Deklarationen, die im Anwendungs Programm verwendet werden müssen:

Type *TMACaddr =* ***array****[0..5]* ***of*** *byte;*
Das ist die sog. Hardware Adresse der Ethernet/Internet Einheit. Rein formal sollte diese weltweit nur einmal vorkommen. Da die meisten Systeme jedoch nur in Subnetzen arbeiten, muss i.A. darauf geachtet werden, dass diese Adresse zumindest in diesem Netzwerk nur einmal vorkommt.

Type *TIPaddr =* ***array****[0..3]* ***of*** *byte;*
Das ist die logische Adresse des Netzwerk Knotens, auch lokale IP Adresse genannt. Wenn kein Server oder Gateway vorhanden ist, dann muss diese Adresse international sein. Das heisst eine Registrierungs Institution vergibt diese einmalige Adresse. In Subnetzen vergibt der Netzwerk Administrator diese IP Adresse.

Function *GetDefaultMAC: tMACaddr; // ENC424J600 only*
Diese Funktion ist nur bei dem ENC424J600 vorhanden. Dieser hat eine feste und einzigartige MAC Adresse eingebaut. Trotzdem lässt sich diese überschreiben. Um das Init des TINA hier nicht grundlegend ändern zu müssen (TINAcore, TINA_Init), muss diese Adresse zum Start aus dem ENC mit *GetDefaultMAC* ausgelesen werden und im Record TINAcore wieder mitgegeben werden.

3.13.3 Das xUDP Protokoll

Das zur Zeit implementierte UDP Protokoll zählt zu den sogenannten unsicheren Protokollen.
Das heisst dass ein Packet abgeschickt wird, der Empfänger den Empfang aber niemals quittiert. Der Sender weiss somit auch nie, ob das Packet den Empfänger erreicht hat und ob es fehlerfrei ist.

Für viele Anwendungen ist das nicht weiters tragisch, man denke hier an Sprach Übertragung. Wenn hier z.B. ein Buchstabe bzw. Teil eines Wortes fehlt, macht sich dies bestenfalls durch ein Knacken bemerkbar.

Für manche Anwendungen ist das nicht tolerierbar und man muss deshalb auf das wesentlich langsamere und aufwändigere TCP/IP Protokoll umsteigen.

Wenn man jetzt aber bei UDP ein Handshake Verfahren mit TimeOut anwendet, so kann man auch hiermit eine sehr sichere und vor allen Dingen schnelle Datenübertragung erreichen.

Das xUDP Protokoll im TINA Kernel wartet nach einem Packet Transfer eine gewisse Zeit, bis der Empfänger ein $A55A zurückgeschickt hat. Wird diese Zeit (TimeOut) überschritten, so wiederholt sich dieses Aktion (retry mit timeout) bis das Packet entweder vom Empfänger quittiert wurde oder die Retries abgelaufen sind.
Das Protokoll, das TimeOut und die Retries werden als Parameter in den Socket Record eingefügt.
Das Daten Packet besteht aus den eigentlichen Daten plus dem angehängten Wort $A55A. Es gibt drei Arten des Handshakes:

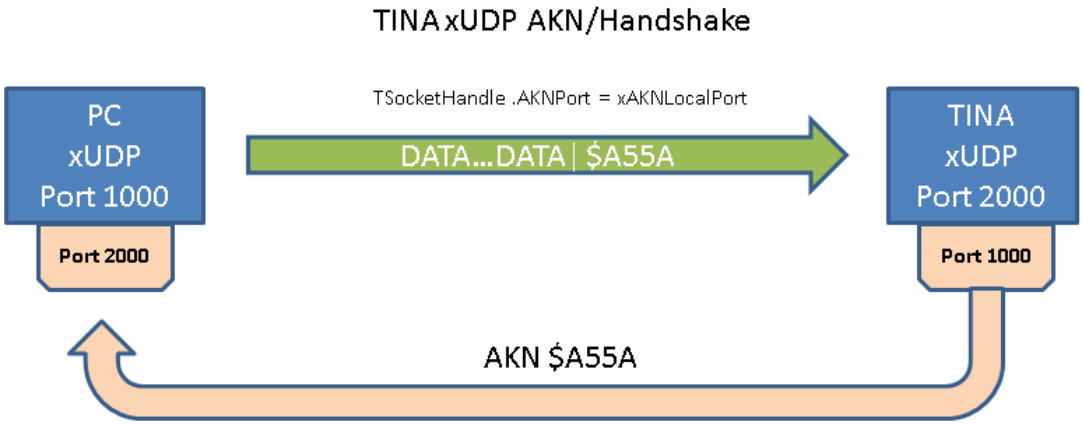
xAKNLocalPort. Hier schickt der Empfänger das Acknowledge Wort $A55A über das gleiche Port zurück wie der Sender das Packet gesendet hat. Das Ack Empfangsport des Senders hat die gleiche Nummer wie das Daten Port des Empfängers.

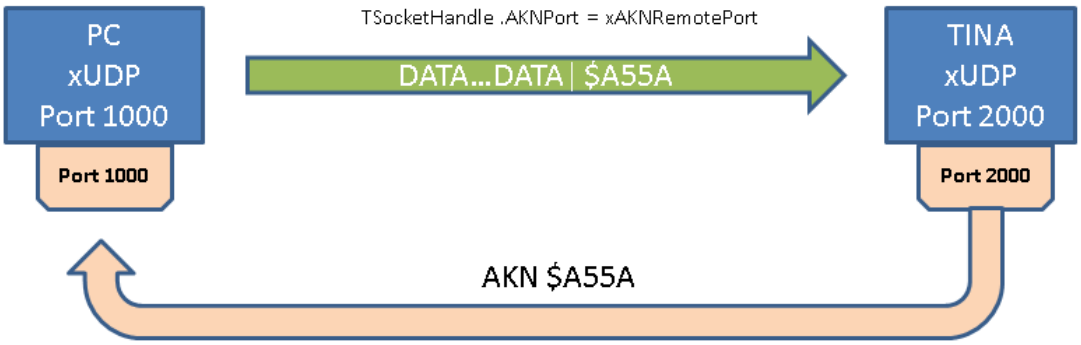
xAKNRemotePort. Hier schickt der Empfänger das Acknowledge Wort $A55A über das gleiche Port zurück wie das Daten Packet empfangen hat. Das Ack Empfangsport des Senders hat die gleiche Nummer sein Daten Sende Port.

xAKNRemoteDynamicPort. Die Arbeitsweise ist ähnlich der von xAKNRemotePort nur mit dem Unterschied dass das acknowledge word an das Port gesendet wird von wo die Daten herkanen und nicht an das Port in TinaSocket RemotePort, was der Fall wäre mit xAKNRemotePort. Reverse Communication ist die gleiche.

Welches der drei Modes benutzt werden muss hängt vom gewählten Socket Typ des PC ab.

```
var UDP2        : TSocketHandle;
...
 UDP2 := TinaCreateSocket;
 with UDP2^ do
  Protocol:= pxUDP;
  ...
  TimeOut:= 500;        // msec
  RetryCount := 4;
  AKNPort:= xAKNLocalPort;    // xAKNRemotePort
  ...
 endwith;
```





3.13.4 Broadcasts

Für eigene low-level Dienste ist die Unterstützung von Broadcasts wesentlich. Der Treiber unterstützt deshalb sowohl den Empfang als auch das Senden von Broadcasts.

Ein kleines Broadcast Projekt befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\TINA_Broadcast**

3.13.5 DHCP

Dieser Treiber/Unit dient in Zusammenarbeit mit dem AVRCo TINA Ethernet Interface zum automatischen Erzeugen einer lokalen IP Adresse mit einem DHCP Server. Wenn möglich, lädt er auch einen DNS, den Gateway und DHCPLeaseTime Details.

In dem Beispiel werden die Ergebnisse in das RAM gespeichert. Die Applikation kann diese Parameter dann ins EEprom ablegen. Dadurch kann beim nächsten Power-UP:
1) die zuletzt zugeteilte IP Adresse vom DHCP Server validiert werden. (Die IP addr wird dem DHCP Server übermittelt und abgefragt)
2) Wenn der DHCP Server nicht erreichbar ist, bleibt diese IP bestehen und wird möglicherweise weiter verwendet. Dieses muss dann die Applikation entscheiden..

Dieser Treiber benötigt ca. 3750 Bytes.

Voraussetzung -
1. tnDHCP – diese Unit muss in das Uses Statement im Hauptprogramm eingefügt werden.
2. Das Hauptprogramm erstellt auch den TINA socket. Ein Sockethandle wird dem Treiber übergeben.

Benutzung -
Das Main ruft die
Function *DHCP_Request(Secs : Byte; DHCPSocke t :TSocketHandle) : Boolean;*
auf, wobei Secs das Timeout für die Antwort des DHCP Servers bestimmt, z.B. 3. Der Socket muss von der Application mit entsprechenden Werten aufgefüllt werden. Siehe auch das Projekt Beispiel.

Es gibt zwei Support Funktionen:
Function *DHCP_Renew(Secs : Byte; DHCPSocket : TSocketHandle) : Boolean ;*
Function *DHCP_Rebind(Secs : Byte; DHCPSocket : TSocketHandle) : Boolean ;*
Für eine erweiterte Lease Time wird DHCP_Renew oder DHCP_Rebind benutzt, abhängig von der Netzwerk Struktur (siehe DHCP protocol im Internet). Normalerweise ergeben beide Funktionen gute Resultate. Man kann die erreichte lease time in der Variablen DHCPLeaseTime erfahren. Aber das Dekrementieren dieser lease time muss ggf. die Applikation tun. Wenn eine Funktion ein false ergibt, kann man den Fehler durch die globale Variable DHCP_Error erfahren.

Das Resultat wird in diesen globalen Variable abgelegt (Unit tnDHCP):

```
DHCPLeaseTime     : Longword;
DHCP_Error        : tDHCP_Errors;    = (DHCPOK, DHCPNAK, FalseCookie, FalseXID, NoResponse)
MacAddr           : TMacAddr;
LocalIP           : tIPaddr;
IPMask            : tIPaddr;
DHCPServer        : tIPaddr;
IPGateway         : tIPaddr;
DNSServer1        : tIPaddr;
```

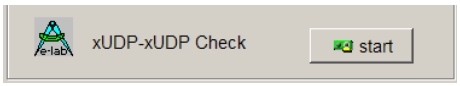
Beispiel -
DHCP_Request(3, TinaSocket1); // Call a DHCP server with a 3 second timeout and with a sockethandle.

Limits - 1) nur 1 DNS Server wird unterstützt.
2) die DHCP transaction ID benutzt vier Bytes der MAC Adresse als Identifier und keine random number. Dadurch wird Code gespart..

Ein komplettes DHCP Projekt befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\TINA_DHCP**
Für XMegas ist ein DHCP Project im Verzeichnis **..\E-LAB\AVRco\Demos\XMega_DHCP**

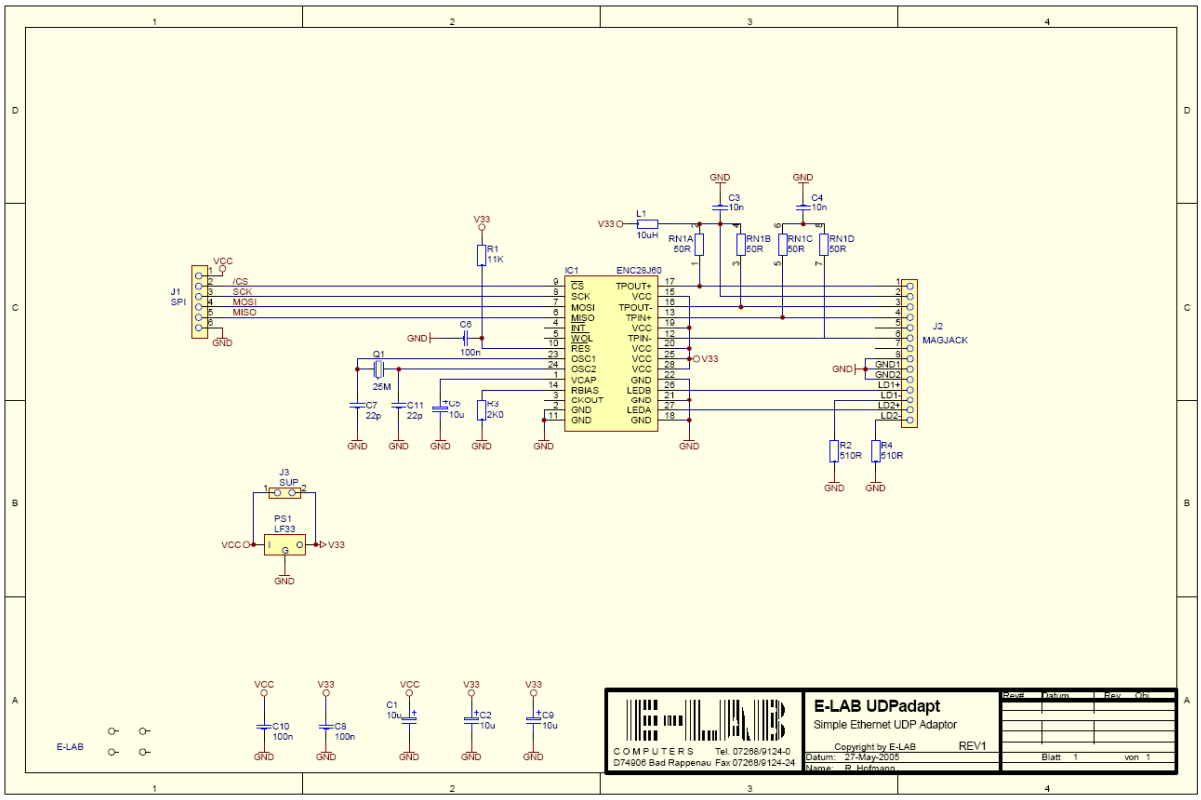
3.13.6 Support Tools

Zur Zeit gibt es ein vorläufiges Support Programm zum Testen des TINA xUDP Programms.
In der IDE kann es mit  aufgerufen werden.



3.13.7 Beispiel Programm und Schaltplan

Ein kleines xUDP Projekt befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\TINA_ENC_UDP**



3.14 USB Interface Einleitung

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 AVR Seite ist der FTDI wie ein MAX232 an den AVR anzuschliessen. Nachteile: Platz Bedarf, Kosten und der Konflikt mit anderen COMports auf der PC Seite.
2. HID und USB-AVRs. 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-AVRs. 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-AVRs. Der eleganteste Weg für einen USB Anschluss. Der AVR 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). Bis zu 4 Daten Pipes, 2xDownstream, 2xUpstream. Flexible Packet Grössen. Hohe Geschwindigkeit, 200kByte/sec mit übergeordnetem Handshake! 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 AVR-USB Treiber des AVRco System leistungsfähige aber trotzdem einfache Funktionen. Es werden alle AVRs mit interner USB Schnittstelle unterstützt.

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 AVR abgesetzt werden. Dort werden diese Daten über eine Callback Funktion an die Applikation weitergeleitet.

Simple Interface. Es werden zwei 8 Byte grosse Pipes, Endpoint3 und Endpoint4, zur Verfügung gestellt (1xUpstream, 1xDownstream). Diese dienen zur asynchronen Kommunikation ohne direkte Rückmeldung etc. Der AVR legt diese 8 Bytes in einen Buffer ab und informiert das Hauptprogramm darüber. Was dieses daraus macht ist sache des Main. Ein erneuter 8byte Block vom PC in diese Pipe überschreibt dann den Buffer ohne zu warten ob das Main die letzten Daten verarbeitet hat oder nicht. Ähnlich ist es hier mit dem Upstream. Der PC kann kontinuierlich den 8byte TxBuffer lesen ohne zu wissen ob dieser inzwischen von der Applikation upgedated wurde. Die Applikation wird nur informiert dass der Host gelesen hat.

Der eigentliche Datentransfer erfolgt über Endpoint1 und Endpoint2.

PacketDown: Der Host schickt hier bis zu 64Bytes oder ein vielfaches davon. Der Treiber setzt ein Flag dass Daten kommen und wieviele. Die Applikation muss dieses Flag beobachten und diese Daten jetzt über eine Funktion abholen. Diese Pipe bleibt gesperrt bis alle Daten abgeholt sind. Der Host kann also solange keine Daten mehr schicken. Im Host/PC gibt es die Möglichkeit Packete > 64bytes zu schicken. Der Treiber im Host teilt die totale Block Grösse dem AVR mit (Kommando über Control-Pipe/Endpoint0). Die von der Applikation im AVR aufgerufene Empfangs Routine verbleibt nun solange im Treiber bis alle Daten empfangen wurden. Natürlich muss der AVR in der Lage sein auch ein grösseres Daten Packet am Stück im Speicher abzulegen.

PacketUp: Der Host liest hier bis zu 64Bytes oder ein vielfaches davon. Der Treiber setzt ein Flag dass Daten geholt werden und wieviele. Die Applikation muss dieses Flag beobachten und diese Daten jetzt über eine Funktion dem Treiber übergeben. Diese Pipe bleibt gesperrt bis alle Daten gesendet sind. Der Host kann also solange keine Daten mehr anfordern. Im Host/PC gibt es die Möglichkeit Packete > 64bytes zu lesen. Der Treiber im Host teilt die totale Block Grösse dem AVR mit (Kommando über Control-Pipe/Endpoint0). Die von der Applikation im AVR aufgerufene Sende Routine verbleibt nun solange im Treiber bis alle Daten gesendet wurden. Natürlich muss der AVR in der Lage sein auch ein grösseres Daten Packet am Stück im zu senden.

Durch dieses Verfahren wird ein extrem sicherer Datentransfer installiert, was allerdings etwas auf Kosten der Geschwindigkeit geht. Es werden 200kByte/sec erreicht.

3.14.1 Import des USB Treibers

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

```
Import SysTick, USBport, WatchDog, ...;
```

Da der USB Treiber ein Time-Out braucht, muss auch der SysTick importiert werden. Der WatchDog wird für einen echten Hardware Reset gebraucht.

3.14.2 Definition des USB Treibers

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

```
Define
  ProcClock      = 16000000;          // Hertz
  SysTick        = 10;                // msec
  StackSize      = $0080, iData;      // min size
  FrameSize      = $00C0, iData;      // min size
  WatchDog       = 7;
  USBvid         = 9876;              // Vendor-ID
  USBpid         = 1234;              // Product-ID
  USBprodRel     = 4321;              // Product-Release
  USBmanufact    = 'E-LAB Computers';
  USBprodName    = 'EvaBoard USB128';
  USBcurrent     = 200;               // max current consumption
  USBsernum      = 2143;              // Product serial number
```

ProcClock hier sind nur 8 oder 16MHz zulässig.
SysTick ist notwendig für interne TimeOuts.
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.
USBsernum ist die serien Nummer des Geräts.

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.

3.14.3 Exportierte Typen

```
type
  tEp3_4Buf = array[0..7] of byte;
```

Dieser Array Typ wird zum Lesen und Schreiben der 8byte Daten des Simple Interface gebraucht.

3.14.4 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:

```
Procedure myUserProc(x : byte; v : word);
begin
  // ...      // this function returns with user defined commands in x and v send by the host
end;
…
  USBsetUserProc(@myUserProc);
```

Wird myUserProc nicht erstellt oder diese mit USBsetUserProc nicht an den Treiber übergeben, dann erfolgt auch kein Callback und die evtl. ankommenden Kommandos werden ignoriert.

Achtung: Callbacks erfolgen immer aus einem Interrupt heraus. Deshalb sind grosse Operationen (z.B. Float) hier nicht angebracht um die Interrupts nicht zu lange zu sperren. Treiberaufrufe (LCD, COMport etc.) sind hier absolut Tabu. Das führt in aller Regel zum Absturz oder zumindest zur Total Blockade.

3.14.5 Exportierte Funktionen und Prozeduren

3.14.5.1 Allgemeine Funktionen

Procedure *USBsetTimeOut(to : byte);*
In den Sende und Empfangs Funktionen kann es z.B. durch äussere Störungen zum Stillstand kommen. Damit dieser sich wieder auflöst wird durch den SysTick ein TimeOut Byte heruntergezählt. Mit dieser Funktion wird dieser Wert gesetzt, in SysTicks 0..255.

Function *USBgetTimeOut : boolean;*
Kehrt die Funktion *USBsetTxBuf* oder die Funktion *USBgetTxBuf* mit dem Resultat „0“ zurück, so kann ein Timeout Fehler vorliegen. Hiermit kann der Treiber auf einen TimeOut abgefragt werden. Diese Abfrage setzt auch das TimeOut Flag zurück. Es ist Sache der Applikation wie dann verfahren werden sollte.

Procedure *USBsetUserProc(@myUserProc);*
Hiermit wird dem Treiber die Adresse der User Funktion *myUserProc* im Main mitgeteilt. Schickt der Host über den Endpoint0 (Control) bestimmte User-definierte Kommandos wird die Prozedur *myUserProc* aufgerufen und in den Parametern x und v die empfangenen Kommandos übergeben.

Function *USBinit : boolean;*
Hiermit wird die USB Hardware initialisiert, ebenso diverse lokale Parameter. Danach ist die AVR Hardware und der Treiber bereit die Anmelde Abfrage des Hosts zu bearbeiten und zu beantworten. PC macht Ding-Dong.

Procedure *USBclose;*
Das USB Interface des AVRs wird geschlossen. Der PC/Host bemerkt dies und schliesst ebenfalls das zugehörige Interface (Ding-Dong). Soll das Interface wieder geöffnet werden, dann muss ein neues USBinit durchgeführt werden.

Function *USBconnected : boolean;*
Der Status des USB Interface wird abgefragt. Ein **false** wird zurückgegeben wenn

a. ein elektrisches Problem (z.B. Kabel) besteht.
b. ein USBclose durchgeführt wurde.
c. noch kein USBinit erfolgt ist.

3.14.5.2 Simple Interface

Function *USBgetRxState8 : byte;*
Die Funktion gibt als Ergebnis zurück wie oft der Host den internen RxBuf8 geschrieben hat. Zu beachten ist, dass der Host bis zu dieser Abfrage diesen Buffer mehrfach überschrieben haben kann. Ist das Ergebnis 0 dann ist nichts neues da. Bei > 0 ist der Buffer mindestens einmal geschrieben worden.

Procedure *USBgetRxBuf8(var buf : tEp3_4Buf);*
Kehrt die Funktion *USBgetRxState8* mit einem Resultat <> 0 zurück, enthält der RxBuffer im Treiber 8 neue Bytes. Dies sind immer die zuletzt empfangenen 8 Bytes. Die Prozedur schreibt nun diese 8 bytes in den durch var buf gekennzeichneten Bereich in der Applikation und setzt den zugehörigen Counter im Treiber auf 0.

Function *USBgetTxState8 : byte;*
Die Funktion gibt als Ergebnis zurück wie oft der Host den internen TxBuf8 gelesen hat. Zu beachten ist, dass der Host bis zu dieser Abfrage diesen Buffer mehrfach gelesen haben kann. Ist das Ergebnis 0 dann ist der Buffer zumindest 1 mal vom Hosdt gelesen worden. Bei > 0 ist der Buffer vom Host seit dem letzten schreiben der Applikation in diesen Buffer noch nicht gelesen worden.

Procedure *USBsetTxBuf8(buf : tEp3_4Buf);*
Kehrt die Funktion *USBgetTxState8* mit einem Resultat = 0 zurück, hat der Host diesen Buffer schon gelesen. Die Prozedur schreibt nun 8 bytes aus dem durch buf gekennzeichneten Bereich in der Applikation in den Treiber Buffer und inkrementiert den zugehörigen Counter im Treiber. Hat der Host nun diesen Buffer im Treiber gelesen, dann wird dieser Counter wieder auf 0 gesetzt.

Im **PacketMode** informiert der Host vor jedem Packet den Slave wieviele Bytes er insgesamt lesen oder schreiben will. Diese Werte werden im Treiber abgelegt und informieren die Applikation auf Nachfrage sowohl über die Transfer Richtung als auch über die Anzahl der zu schickenden bzw. empfangenden Bytes.

3.14.5.3 PacketDown

Function *USBgetRxState : word;*
Mit dieser Funktion muss die Applikation notfalls kontinuierlich den Rx-Status des PacketMode pollen. Wenn das Ergebnis <> 0 ist muss umgehend die untenstehende getRxBuf Funktion aufgerufen werden um ein Timeout auf der Host Seite zu vermeiden.

Function *USBgetRxBuf(p : pointer; len : word) : word;*
Hiermit wird das anstehende Packet vom Host direkt im Treiber abgeholt. Zu beachten ist, dass hier keine Zwischenpufferung im Treiber erfolgt, sondern direkt die Endpipe ausgelesen wird. Der Pointer Parameter bestimmt dabei die Ziel Adresse in der Applikation. Normalerweise werden soviele Bytes abgeholt wie getRxState angezeigt hat. Natürlich muss der Zielspeicher auf den der Pointer zeigt auch diese Anzahl Bytes aufnehmen können. Deshalb wird die Bytecount Grenze in „len“ übergeben. Wird bei der Übertragung „len“ erreicht, so wird diese im Treiber abgebrochen um ein Überschreiben von Variablen etc. zu vermeiden, was normalerweise eine Absturz zur Folge hätte. Als Resultat wird die tatsächlich empfangenen Anzahl der Bytes zurückgegeben.

3.14.5.4 PacketUp

Function *USBgetTxState : word;*
Mit dieser Funktion muss die Applikation notfalls kontinuierlich den Tx-Status des PacketMode pollen. Wenn das Ergebnis <> 0 ist muss umgehend die untenstehende setTxBuf Funktion aufgerufen werden um ein Timeout auf der Host Seite zu vermeiden.

Function *USBsetTxBuf(p : pointer; Len : Word) : word;*
Der Host holt ein Packet direkt im Treiber ab. Zu beachten ist, dass hier keine Zwischenpufferung im Treiber erfolgt, sondern direkt in die Endpipe geschrieben wird. Der Pointer Parameter bestimmt dabei die Source Adresse in der Applikation. Normalerweise werden soviele Bytes gesendet wie getTxState angezeigt hat. Möglicherweise hat die Applikation aber garnicht so viele Bytes zum Senden zur Verfügung. Deshalb muss der tatsächliche Bytecount in „len“ übergeben werden. Wird bei der Übertragung „len“ erreicht, so wird diese im Treiber abgebrochen, ansonsten mit Erreichen von „getTxState“. Als Resultat wird die tatsächlich

gesendete Anzahl der Bytes zurückgegeben. Falls keine Bytes zur Verfügung stehen kann daher in „len“ auch 0 übergeben werden.

3.14.5.5 StreamDown
T.B.D.

3.14.5.6 StreamUp
T.B.D.

3.14.6 AVR Implementation

```
program USB_Test;

{  $NOSHADOW}
{  $WG}                  {global Warnings off}

Device = 90USB128, VCC=4.5;
{   $BOOTRST $0F000}        {Reset Jump to $0F000}

Define_Fuses
//  Override_Fuses;
  NoteBook        = B;
  COMport         = USB;
  Supply          = 5.0, 100;
  LockBits0       = [];
  FuseBits0       = [];
  FuseBits1       = [SPIEN, JTAGEN, OCDEN];
  FuseBits2       = [];
  ProgMode        = JTAG;

Import SysTick, USBport, WatchDog, BeepPort, SerPort;

From System Import ;

Define
  ProcClock       = 16000000;      {Hertz}
  SysTick         = 10;          {msec}
  StackSize       = $064, iData;
  FrameSize       = $064, iData;
  WatchDog        = 7;

  SerPort         = 19200;
  BeepPort        = PortE, 5;

  USBvid          = 9876;             // Vendor-ID
  USBpid          = 1234;             // Product-ID
  USBprodRel      = 4321;             // Product-Release
  USBmanufact     = 'E-LAB Computers';
  USBprodName     = 'EvaBoard USB128';
  USBcurrent      = 200;              // max current consumption
  USBsernum       = 2143;             // Product serial number

Uses uAVR_USB;
```

Implementation

```
{$IDATA}
{------------------------------------------------------------------}
{ vars }
var
  USBrxBuf     : array[0..255] of byte;
  USBtxBuf     : array[0..255] of byte;
  USBrxBuf8    : tEp3_4Buf;
  USBtxBuf8    : tEp3_4Buf;

{------------------------------------------------------------------}
{ functions }
Procedure myUserProc(x : byte; v : word);
begin
  // ...
end;

{------------------------------------------------------------------}
{ Main Program }
{$IDATA}
begin
  USBsetUserProc(@myUserProc);

  USBsetTimeOut(100);
  USBinit;
  EnableInts;
  BeepOutHL;
  WriteLn(SerOut, 'E-LAB USB Test');
  While not USBconnected do
    BeepClick;
    SerOut('?');
    mDelay(300);
  endwhile;
  WriteLn(SerOut);
  WriteLn(SerOut, 'connected');
```

```
loop
   if USBgetRxState8 > 0 then
     USBgetRxBuf8(USBrxBuf8);
   endif;
   if USBgetTxState8 = 0 then
     USBsetTxBuf8(USBtxBuf8);
   endif;
   if USBgetRxState > 0 then
     USBgetRxBuf(@USBrxBuf, 256);
   endif;
   if USBgetTxState > 0 then
     USBsetTxBuf(@USBtxBuf, 256);
   endif;
   if SerStat then
     case SerInp of
       'D','d' : USBclose;     // detach, disconnect
                 |
       'C','c' : USBinit;      // re-init
                 |
       'S','s' : if USBconnected then
                   SerOut('c');
                 else
                   SerOut('x');
                 endif;      // status, connected
                 |
     else
       BeepOut(1000, 10);
       SerOut('?');
     endcase;
   endif;
 endloop;
end USB_Test.
```

Dieses Demo und Testprogramm befindet sich auch in der Installation unter

..\AVRco\Demos\EvaBoard128USB

3.14.7 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 USB 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:

3.14.7.1 Initialisierung etc.

```
Procedure usb_init;
Function usb_find_busses : longword;
Function usb_find_devices : longword;
Function  usb_get_busses : pusb_bus;
```

3.14.7.2 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;
```

3.14.7.3 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;
```

3.14.7.4 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 geschickt werden oder der Transfer Modus eingestellt werden

tmsg_type = (msgReset, msgTest, msgPacket, msgStream, msgDownLen, msgUpLen, msgUser = $10);

msgReset	löst einen Hardware Reset auf dem Slave (AVR) aus.
msgTest	versetzt Host und Slave in den Test Mode. Es werden beliebig viele 64byte Packete gesendet und empfangen werden. Auf der AVR Seite werden die Daten nicht weiter verarbeitet. Der Mode dient zum Testen der absoluten Transfer Rate.
msgPacket	ist der normale Arbeits Mode. Dazu sind Kommandos *msgDownLen* und *msgUpLen* zu vor jedem receive und transmit zu benutzen um dem Slave die Grösse des nächsten Packets anzukündigen.
msgStream	stellt den Stream Mode ein. Noch nicht implementiert!
msgDownLen	muss im Packet Mode jedem *usb_bulk_write* vorangestellt werden. Index enthält den Bytecount.
msgUpLen	muss im Packet Mode jedem *usb_bulk_read* vorangestellt werden. Index enthält den Bytecount.

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 oder endpipe3 geschickt.
Parameter:

ep = endpoint1 oder endpoint3. Endpoint3 kann nur 8bytes am Stück emfangen!
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.

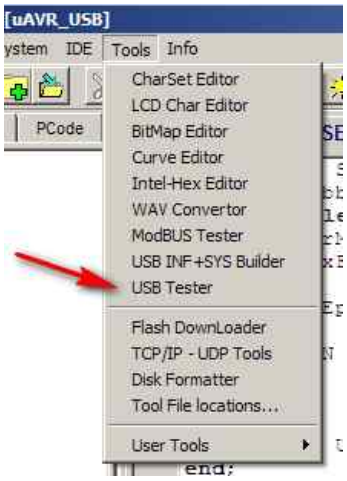
Beim Endpoint3 müssen immer 8 Bytes geschickt werden! Der aktuelle Mode spielt hierbei keine Rolle.

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 endpipe2 oder endpipe4 abgeholt.
Parameter:

ep = endpoint2 oder endpoint4. Endpoint4 kann nur 8bytes am Stück senden!
bytes = Buffer der die Empfangs Daten enthält
size = Anzahl der zu empfangenden Bytes. Bei endpoint2 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.

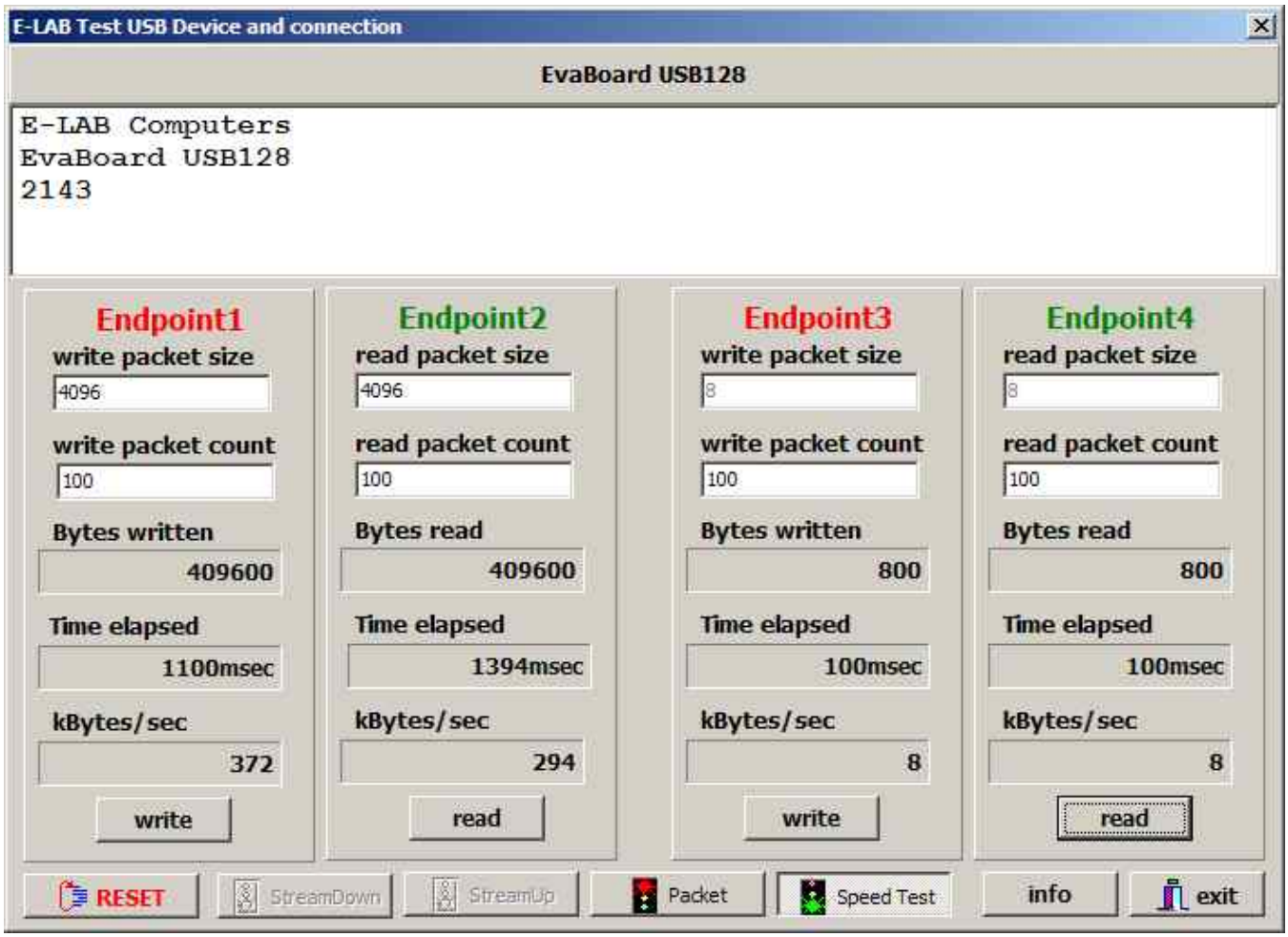
Beim Endpoint4 müssen immer 8 Bytes abgeholt werden! Der aktuelle Mode spielt hierbei keine Rolle.

3.14.8 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.



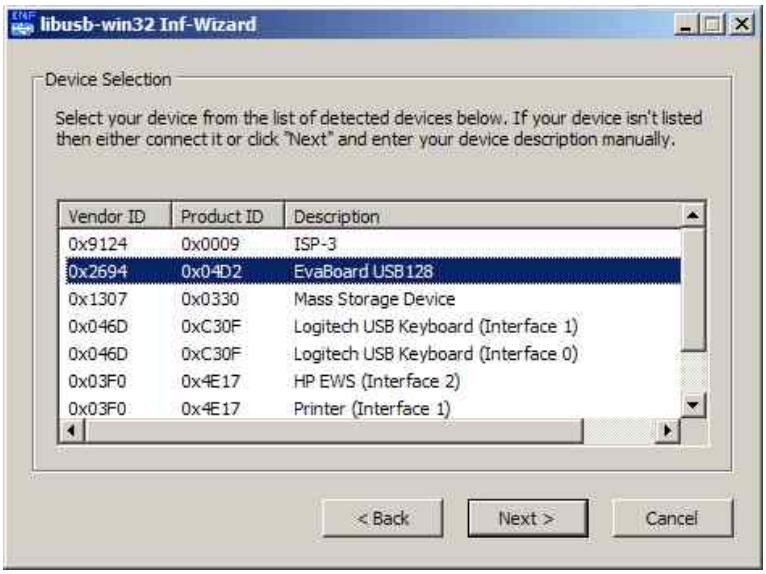
Hier können die diversen Modi eingestellt und Transfers gestartet werden.

Die Source dieses Delphi Programms befindet sich in der Installations Directory unter

..\AVRco\IDE\USBtester\

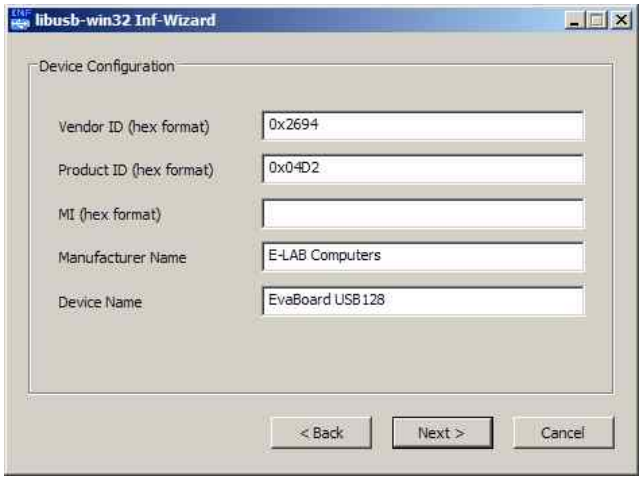
3.14.9 Support Tools

Inf-Wizard Start



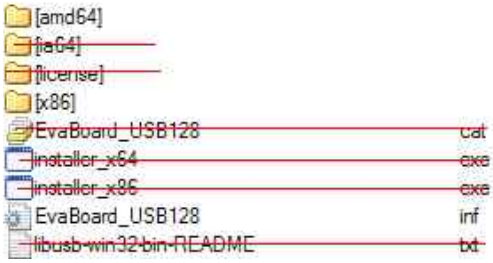
Die Erstellung der notwendigen SYS, INF und DLL Files für den Host/PC stellt eine enorme Aufgabe für den Entwickler dar. Für viele praktisch unlösbar ohne das richtige knowhow und Erfahrung. Das AVRco System enthält deshalb ein Tool das genau diese Files erstellt. Es wird mit obenstehenden Menu gestartet.

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.

3.15 USB-CDC Virtual Comport XMega

Die Implementation eines USB Treibers ist auf beiden Seiten, PC und AVR, recht kompliziert und aufwändig. Für viele Anwendungen würde auch eine ganz simple Schnittstelle wie z.B. ein Comport genügen.

Hier hilft dann ein virtueller Comport der auf beiden Seiten über das USB Interface gelegt wird. Auf dem PC ist das praktisch schon im System integriert. Man braucht hier nur noch ein passendes INF File.

Auf der AVR Seite ist da ein erheblich grösserer Aufwand zu treiben. Allerdings ist dieser Aufwand schon komplett im SerPortCDC enthalten. So wird der Programmieraufwand auf das Handling der entspr. Seriellen Schnittstelle im PC und im AVR beschränkt. Es sind auf beiden Seiten keinerlei USB Operationen notwendig. Allerdings muss das INF File für den PC selbst erstellt werden.

3.15.1 Import des CDC Treibers

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

```
Import SysTick, SerPortCDC, ...;
//From SerPort import SerPortSelect;   // optional
```

Da der CDC Treiber einige Time-Outs braucht, muss auch der SysTick importiert werden.

3.15.2 Definition des CDC Treibers

Der CDC Treiber benötigt ein paar Defines die für die USB Verbindung zum PC notwendig sind:

```
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

  // XMega USB must use the internal 32MHz OSC. So the system must use the 2MHz OSC
  OSCtype          = int2MHz,
    PLLmul         = 16,
    prescB         = 1,
    prescC         = 1;
  SysTick          = 10;          {msec}
  StackSize        = $0080, iData;
  FrameSize        = $00C0, iData;

  USBmanufact      = 'E-Lab Computers';     // max 31 bytes
  USBprodName      = 'Serial CDC-USB';      // "     "
  USBpid           = 77;                    // CDC
  USBvid           = $ABCD;
  USBprodRel       = 001;
  USBcurrent       = 500;
  //USBvBUS        = PortB.4;               // port and pin, optional

  //  SerPortCDC   = timeout;               // SysTick timouts, optional
  RxBufferCDC      = 64, iData;             // min 64, max 254
  TxBufferCDC      = 100, iData             // min 64, max 254;

uses uXMega_CDC;
```

Die USB Defines sind zwingend, allerdings muss der User für die korrekten Daten sorgen, z.B. USBvid.
Der Import der Unit uXMega_CDC ist ebenfalls notwendig. Diese Unit stellt alle notwendigen USB Funktionen zur Verfügung.

Die User Applikation benutzt das CDC Device wie eine ganz normale serielle Schnittstelle. Fast alle im AVRco enthaltenen SerPort Funktionen werden hier auch unterstützt.

3.15.3 Exportierte Funktionen

Procedure *CDCopenPort;*
Initialisiert und eröffnet das CDC Port.

Procedure *CDCclosePort;*
Schliesst das geöffnete CDC Port.

Function *CDCportValid : boolean;*
Gibt ein true zurück wenn das CDC Port gültig und betriebsbereit ist.

Procedure *SerOutCDC(ch : byte|char);*

Function *SerInpCDC : byte|char;*

Function *SerInp_TOCDC(var : byte|char; timeout : byte) : Boolean;*

Function *SerStatCDC : boolean;*

writeLn*(SerOutCDC, st);*

Procedure *SerOutBlock(UsartCDC; location: type);*

Procedure *SerOutBlock_P(UsartCDC; p : pointer; count : word);*

Procedure *SerInpBlock(UsartCDC; var location: type);*

Function *SerInpBlock_TO(UsartCDC; var location: type; timeout : byte) : Boolean;*

Procedure *SerInpBlock_P(UsartCDC; ptr : pointer; len : word);*

Function *SerInpBlockP_TO(UsartCDC; ptr : pointer; timeout : byte) : Boolean;*

Procedure *FlushBuffer(RxBufferCDC);*

Procedure *FlushBuffer(TxBufferCDC);*

Function *PipeStat(TxBufferCDC) : byte;*

Function *PipeFull(RxBufferCDC) : boolean;*

Weitere allgemeine Infos über die SerPort Funktionen im AVRco Standard Driver Manual

Achtung:
Ein CDC Port (Virtuelles ComPort) arbeitet im Prinzip genauso wie ein UART. D.h. man kann die Buffer überfahren. Deshalb sollte aus Sicherheits Gründen vom PC aus keine Blöcke > RxBufferSize Bytes geschickt werden. Kann der AVR nicht schnell genug den vollen RxBuffer leeren, werden überzählige Bytes ignoriert!

Beim Senden zum PC ist zu beachten, dass bei einer unterbrochenen Verbindung alle Zeichen verloren sind. Deshalb muss immer wieder mit „CDCportValid“ abgeprüft werden, ob der CDC noch richtig arbeitet.

Im übrigen macht es Sinn mit einer Art handshake zu arbeiten. Eine Seite schickt Daten und wartet bis die andere Seite ein „ok“ oder ähnliches zurückschickt bevor die nächsten Daten geschickt werden.

Das beiliegende INF File passt zum Demo. Falls USBvid oder USBpid in der Source geändert wird, muss auch das INF File angepasst werden!

Programm Beispiel:
ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_USB_CDC**
Ein passendes INF File für den PC ist dort ebenfalls vorhanden.

3.16 AES Encrypt/Decrypt XMega

Die Fehler- und Diebstahl Sicherheit in der Datenübertragung wird immer wichtiger. Einen grossen Schritt macht hierbei das Verschlüsseln von Daten jeglicher Art mit dem AES Verfahren. Dieses benutzt einen 128bit Schlüssel der praktisch nicht hack-bar ist.

Das AVRco System benutzt dafür die in den **meisten** XMegas vorhandene AES Hardware. Diese ist extrem schnell so dass der Zeitverlust für das entschlüsseln ohne grosse Bedeutung ist.
Beim AES gibt es zwei unterschiedliche Verfahren, das ECB und das CBC. Das ECB ist zwar einfacher aber auch weniger sicher als das CBC.

ECB
Hier wird immer ein 16byte Block verschlüsselt bzw. entschlüsselt. Das AES_Init darf nur einmal beim Start aufgerufen werden. Der Schlüssel wird nicht verändert. Damit wird klar dass gleiche Blöcke auch immer gleich verschlüsselt werden. Ist in einem Block ein bit falsch dann wird nur dieser Block falsch decrypted!

CBC
Auch hier werden immer 16byte Blöcke entschlüsselt. Das AES_Init darf aber nur einmal beim Start einer Serie von zusammenhängenden 16byte Blöcken ausgeführt werden. Hier wird intern der Init Schlüssel weiter verändert, abhängig vom vorrausgegangenen Block. Ist irgendwo ein Bit falsch so werden alle folgenden Blöcke falsch decrypted. Ein hacken ist damit ausgeschlossen.

3.16.1 Import des AES Treibers

Wie sonst auch im AVRco System muss die Unit des Treibers importiert werden:
Uses *uXMegaAES, ...;*

3.16.2 Exportierte Funktionen

Der Treiber exportiert mehrere Funktionen.

Function *AES_Init(Key, InitVector : Pointer) : boolean;*
Wird einmal vor dem Start einer AES Session aufgerufen. Beim ECB Mode muss der InitVector nil sein. Beim CBC muss InitVector einen Wert haben. Key beim ECB und Key und Vector beim CBC müssen eine 16byte Struktur sein.

Function *AES_DecryptECB(Data : Pointer; count : word) : boolean;*
Diese Funktion entschlüsselt einen Datenblock. Data muss auf einen Block zeigen der ein vielfaches von 16 gross ist. Auch Count muss ein vielfaches von 16 sein. Die Daten befinden sich anschliessend in Data block.

Function *AES_EncryptECB(Data : Pointer; count : word) : boolean;*
Diese Funktion verschlüsselt einen Datenblock. Data muss auf einen Block zeigen der ein vielfaches von 16 gross ist. Auch Count muss ein vielfaches von 16 sein. Die Daten befinden sich anschliessend in Data block.

Function *AES_DecryptCBC(CipherBlock, PlainBlock : Pointer; count : word) : boolean;*
Diese Funktion entschlüsselt einen Datenblock. CipherBlock (source) muss auf einen Block zeigen der ein vielfaches von 16 gross ist. PlainBlock (destination) muss auf einen Block zeigen der ein vielfaches von 16 gross ist. Der destination Block (PlainBlock) muss mindestens so gross wie der Source Block sein. Auch Count muss ein vielfaches von 16 sein.
Nach dem entschlüsseln befinden die entschlüsselten Daten im PlainBlock.

Function *AES_EncryptCBC(PlainBlock, CipherBlock : Pointer; count : word) : boolean;*
Diese Funktion verschlüsselt einen Datablock (PlainBlock -> CipherBlock). Ähnlich AES_DecryptCBC.

Achtung
Es muss klar sein, dass auf beiden Seiten einer AES Kommunikation identische Keys und InitVectors benutzt werden müssen.

Programm Beispiel:
ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_AES**

3.17 Wiegand Interface Einleitung

Das Wiegand Daten Interface ist eins der am meisten verwendeten Interfaces in Zugangs Kontroll Systemen der Industrie. Aber es ist im PC Bereich absolut unbekannt. Will man nun so ein System testen oder gar implementieren so muss man sich zumindest einen Wiegand Controller besorgen, meistens sogar ein komplettes Zugangs Konmtroll System.

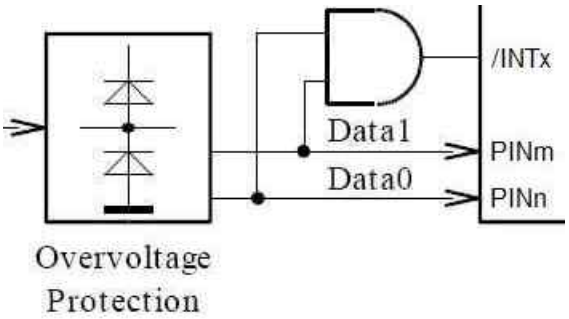
Das ACRco System unterstützt deshalb die 2-line Wiegand Sensoren, z.B. von HID. Die am meisten verwendete Versionen sind die Typen mit dem 26bit oder 37bit Protokoll welche auch hier implementiert sind. Der Treiber ist komplett Interrupt gesteuert.

3.17.1 Interface

Ein typischer Wiegand Sensor hat 2 Datenleitungen, DATA0 und DATA1. Beide sind im Ruhezustand log1 = high. Die Spannung einer “1” kann zwischen 5V und 24V liegen, abhängig von der Betriebsspannung des Geräts. Wenn diese Spannung höher ist als die Betriebs Spannung des uC, müssen die zwei Leitungen im high-Status auf die VCC der CPU geklemmt werden. Das geschieht meistens durch ein paar Schutzdioden.

Die Information die der Sensor sendet werden durch “low” Impulse auf zwei Leitungen dargestellt. Das Telegramm besteht aus low Pulse auf beiden Leitungen. Eine Leitung bringt die Nullen und die andere die Einsen. Ein impuls dauert ca. 50usec, die Pause zwischen zwei Pulsen ist ca. 2msec. Die Pause zwischen zwei aufeinander folgenden Telegrammen ist mindestens 200msec.

Durch diese relativ langen Zeiten macht es keinen Sinn den Wiegand Treiber in einem Polling. Verfahren zu betreiben. Ein Interrupt getriebener Treiber ist ein Muss.



Der beste Weg hierbei ist dass zwei Port Pins für die zwei Datenleitungen verwendet werden wobei jeder low Status einen Interrupt erzeugen muss. Mit einem AND Gatter erzeugen beide Leitungen einen low Impuls an dem INT Pin.

Dieses AND Gatter kann z.B. auch mit zwei Schalt Dioden vom Typ 1N4148 aufgebaut werden.

Achtung: die zwei low Impulse sind relativ kurz, ca. 50uSec. Wenn der globale Interrupt zu lange gesperrt ist, kann der Port Pin Interrupt zu sehr verzögert werden, so dass das Lesen der zwei DATA Pins erst erfolgt, wenn diese Leitungen im Idle Zustand sind. Das Ergebnis davon ist dann dass entweder ein Telegramm nicht erkannt wird oder dass Paritäts Fehler auftreten. Im schlimmsten Fall gibt es nie einen korrekten Datenblock.

Ganz im allgemeinen sollte man die Regel beachten, dass komplexe und/oder langwierige Operationen in Interrupt Services absolut vermieden werden müssen. Ansonsten erhält man ein sehr instabiles System.

3.17.2 Import des Wiegand Treibers

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

```
Import SysTick, WiegandPort, ...;
```

Da der Wiegand Treiber einige Time-Outs braucht, muss auch der SysTick importiert werden.

3.17.3 Definition des Wiegand Treibers

Der Wiegand Treiber benötigt ein paar Defines die bestimmen welcher Interrupt und welche Port Pins benutzt werden sollen.

```
Define
    ProcClock      = 16000000;          // Hertz
    SysTick        = 10;                // msec
    StackSize      = $0080, iData;      // min size
    FrameSize      = $00C0, iData;      // min size
    WiegandPort    = INT0, PINA, 2, 3;  // Wigand
```

Der erste Parameter des Wiegand Define ist der notwendige Externe Interrupt. Bitte beachten, dass dieser Interrupt den “negative edge triggered” Mode unterstützen muss. Nicht jeder ältere AVR kann das.

Der zweite Parameter bestimmt das IO-Port an welches die zwei Datenleitungen angeschlossen werden. Die zwei letzten Parameter bezeichnen die zwei benutzten Pins in diesem Port.

Die zwei Pins können miteinander vertauscht werden wenn das Ergebnis invertiert ist. $00 -> $FF

3.17.4 Exportierte Funktionen

Der Treiber exportiert zwei Funktionen die von der Applikation benutzt werden müssen.

WiegandGetState
Diese Funktion muss immer aufgerufen werden, bevor Daten ausgelesen werden können. Sie gibt ein true zurück wenn Daten zur Verfügung stehen. Aber sie macht keinen weiteren Gültigkeits Check.

```
Function WiegandGetState : boolean;
```

WiegandGetData
Beim Aufruf dieser Funktion werden die empfangenen Daten aus dem System Buffer in den User Buffer transferiert. Während des Transfers werden die beiden Paritäts Bits geprüft. Wenn das Telegramm valid ist (kein Paritätsfehler) gibt die Funktion den Frame Typ zurück, entweder eine 26 oder eine 37. Unabhängig vom Ergebnis macht die Funktion den internen Buffer ungültig so dass der nächste Aufruf von WiegandGetState ein false zurückgibt bis ein neuer Frame empfangen wurde.

```
Function WiegandGetData(p : pointer) : byte;
```

Weil der Datenempfang in Interrupts passiert macht es keinen Sinn die Paritätsbits hier zu berechnen und zu vergleichen. Das passiert deshalb in WiegandGetData. Deshalb gibt die Funktion WiegandGetState immer ein true zurück wenn ein kompletter Frame empfangen wurde, ob die Parität nun stimmt oder nicht.

Die Funktion WiegandGetData braucht einen Pointer als Übergabe Parameter. Dieser Pointer muss auf eine Struktur im RAM zeigen, die mindestens 5 Bytes gross ist. Ein Array[0..4] of byte z.B. wäre ok. Die Funktion transferiert dann den intern gespeicherten Frame in den Applikations Bereich auf den der Pointer zeigt.

3.17.5 Beispiel Programm und Schaltplan

Ein Beispiel Programm befindet sich im Verzeichnis **..\E-LAB\AVRco\Demos\Wiegand**

3.18 Inkremental Encoder Treiber IncrPort8

Das AVRco System bietet insgesamt 3 unterschiedliche Treiber für Inkremental Coder. Diese unterscheiden sich wesentlich in der Impuls Verarbeitung. Weil z.B. der *IncrPort4* Treiber zyklisch die Sensoren scannt und damit nur einen Timer und ein 8bit Port benötigt, ist dieser durch die Scanrate limitiert. Das Scannen bedeutet eine Dauerlast für das System und damit kann die Scanrate nicht sehr hoch gewählt werden. Das ergibt bei high-speed Sensoren Probleme.

Der vorliegende Treiber hat diese Einschränkungen nicht, da nur ein Phasen Wechsel der beiden Eingangssignale einen Interrupt erzeugen. Natürlich wird auch hier die mögliche Impuls Rate durch die Prozessor Geschwindigkeit und die Interrupt Verarbeitung limitiert. Wie bei allen high-speed Anwendungen mit Interrupts gilt auch hier dass wenn diese Interrupts durch andere verzögert werden, es zu Fehlern kommen kann.

Diese Implementation benutzt beliebige Input-Ports, an denen bis zu 8stk 2-Phasen Geber angeschlossen werden können. Bedingung ist dabei dass die zwei Signale eines Gebers an das gleiche Port angeschlossen werden. Die CPU muss die PIN-Change-Interrupts (PCint) unterstützen. Die verwendeten Portpins müssen auf Input stehen. Die Vervierfachung des Eingangssignals (Quadratur) ist immer aktiv.

Unbenutzte Pins können beliebig verwendet werden.

Der Treiber benutzt die PIN change Interrupts um die Kanäle zu scannen und auszuwerten. Deshalb kann jede längere Interrupt Sperrung zu fehlerhaften Resultaten führen. 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.

3.18.1 Imports

Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import  …, IncrPort8, ..;
```

Die Auflösung (16 oder 32bit), die gewünschten PCints und die Anzahl der Encoder werden mit Defines vorgegeben. Bitte beachten, dass die Port Pins durch die Vorgabe der PCint Nummer ***PCintxx*** festgelegt werden.

3.18.2 Defines

```
Define
    ProcClock    = 16000000;          {Hertz}
    StackSize    = $0020, iData;
    FrameSize    = $0040, iData;
   IncrPort8     = 32;                 // resolution 16 or 32 bits
   IncrChan0     = PCint00, PCint01;   // PCINT ports used = B.0 B.1
   IncrChan1     = PCint10, PCint11;   // PCINT ports used = C.2 C.3
  …
```

3.18.3 Funktionen

Es werden 8 verschiedene Funktionen zur Verfügung gestellt. Der Parameter *chan* zählt von 0 an (0..7).

Procedure *IncrCount8Start(chan : byte);*
Nach einem Reset oder PowerOn muss der Treiber bzw. jeder einzelne Kanal gestartet werden. Diese Prozedur verändert die Zählerstände nicht.

Procedure *IncrCount8StartAll;*
Alle Kanäle werden gestartet werden.

Procedure *IncrCount8stop(chan : byte);*
Jeder Kanal kann gezielt gestoppt werden. Die Zählerstände werden nicht verändert.

Procedure *IncrCount8stopAll;*
Alle Kanäle werden gestoppt. Die Zählerstände werden nicht verändert.

Function *GetIncr8Val (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. Das Auslesen bedingt einen Interrupt disable.

Procedure *ClearIncr8Val(chan : byte);*
Diese Prozedur setzt den internen Zähler von *chan* auf null.

Procedure *ClearIncr8All;*
Diese Prozedur setzt alle internen Zähler auf null.

Procedure *SetIncr8Val (chan : byte; val : integer [longint]);*
Diese Prozedur setzt den 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 *IncrPort8* 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\Increment8**

3.19 SLIP Treiber SLIPport1..4 / SLIPportC0..F1

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. Im Gegensatz zu der SLIP ***FUNKTION*** (Standard Treiber Manual), die nur eine Shell um den Rx- und TxBuffer herum bildet, arbeitet dieser Treiber vollkommen autark. Er besitzt keine Buffer sondern liest und schreibt direkt in/aus den von der Applikation zur Verfügung gestellten Buffern. Der Vorteil ist ein kompakterer Code und die Möglichkeit ein Handshake, Adressauswertung und Checksummen zu benutzen.

Es können bis zu 4 Ports (SLIPport1..4) gleichzeitig importiert und betrieben werden, abhängig von der Zahl der UARTs in dem Controller. XMega bis zu 8 SLIPports (SLIPportC0.. SLIPportF1).

RS485 Line Driver
Wurde für einen SLIPport der RS485 Modus definiert, so handhabt der Treiber dieses automatisch ohne weiteres zutun der Applikation.

Der Treiber bietet mehrere **Betriebsarten**:

1. ***Einfache*** Kommunikation. Es werden SLIP Packete geschickt und empfangen, ohne Handshake, Adresse, Checksumme etc.
2. Mit ***Checksumme***. Der Sender-Treiber fügt eine Checksumme als letztes Byte an. Der Empfänger-Treiber generiert aus den empfangenen Daten eine Checksumme und vergleicht diese mit der empfangenen. Das Ergebnis erfährt die Empfangs Applikation durch das Lesen des Empfangs Status.
3. Mit ***Handshake***. Der Sender-Treiber schickt ein Packet und wartet bis max. zum Timeout ob der Empfänger-Treiber das Packet quittiert hat (Acknowledge). Dadurch wird das Verfahren zu *half duplex*. Das Ergebnis des ACK oder Timeout erfährt die Sende Applikation durch das Lesen des Sende Status.
4. Mit ***Adresse***. Damit kann ein single-Master/multi-Slave Netzwerk z.B. auf RS485 Basis aufgebaut werden. Der Sender-Treiber schickt als erstes Byte im Packet die Adresse des gewünschten Slaves. Die Empfangs-Treiber aller Slaves lesen das Packet mit und wenn die Adresse mit ihrer internen Adresse übereinstimmt, wird das Packet ausgewertet, andernfalls wird es verworfen. Es sind 190 Adressen möglich. Die Adresse 0 sollte für den Master reserviert sein. Die Adressen 190..254 sind für interne Zwecke reserviert und dürfen nicht benutzt werden. Die Adresse 255 ist die Broadcast Adresse die von allen Slaves ausgewertet wird aber niemals mit einem Acknowledge beantwortet wird.

Alle Betriebsarten können miteinander kombiniert werden. So kann mit Handshake+Checksumme eine extrem sichere und stabile Kommunikation aufgebaut werden. Notfalls kann die Applikation im Fehlerfall hier sogar Retries vornehmen.

Mit der Kombination aller Optionen ist der Aufbau eines flexiblen RS485 Netzwerks möglich. Im Gegensatz zu dem LAN Netzwerk, wo das Multiprozessor Bit des AVRs zu Kennung von Frames benutzt wird, ist hiermit auch das Einbeziehen eines PCs oder anderer Steuerungen/Elektronik möglich, sofern dort ein entsprechender SLIP Treiber installiert ist.

3.19.1 Imports

Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import  SysTick…, SLIPport1, ..;        // SLIPport2, SLIPport3, SLIPport4
XMega
Import  SysTick…, SLIPportC0, ...;    // SLIPporC1, SLIPportD0, SLIPportD1, SLIPportE0, SLIPportE1...
```

Der SysTick wird für die Timeouts gebraucht. Je nach CPU Typ sind bis zu 4 UARTS vorhanden, die durch den jeweiligen Import gewählt werden. SLIPport1 => UART0 etc.

3.19.2 Defines

```
Define
      ProcClock    = 16000000;          {Hertz}
      SysTick      = 10;
      StackSize    = $0020, iData;
      FrameSize    = $0040, iData;
      SLIPport1    = 19200;                    // SLIPport2, SLIPport3, SLIPport4
      //  SLIPportCtrl1  = PortA, 0, positive;    // optional RS485 line driver control
    …
XMega
Define
      ProcClock    = 16000000;          {Hertz}
      SysTick      = 10;
      StackSize    = $0020, iData;
      FrameSize    = $0040, iData;
      SLIPportC0  = 19200;                    // SLIPportC1, SLIPportD0, SLIPportD1...
      //  SLIPportCtrlC0  = PortA, 0, positive;    // optional RS485 line driver control
    …
```

Wird mit einem RS485 Leitungs Treiber gearbeitet, dann muss die Richtung (tx/rx) des Treiber Chips durch den SLIP Treiber gesteuert werden. Um diese Eigenschaft freizugeben muss das steuernde Port, das Portbit und der aktive Pegel definiert werden.

3.19.3 Typen

Es werden 3 Typen exportiert:

```
tSLIPstate    = (SLIPidle, SLIPready, SLIPbusy, SLIPovr, SLIPtout, SLIPfrm, SLIPchkE);
```

Dieser Aufzählungstyp wird als Resultat einer Rx oder Tx Statusabfrage zurückgegeben.
SLIPidle = der Treiber ist im Grundzustand.
SLIPready = es wurden Daten empfangen oder versendet.
SLIPbusy = der Treiber sendet bzw. empfängt gerade.
SLIPovr = beim Empfang ist ein Überlauf aufgetreten, Packet zu gross für den Rx Buffer
SLIPtout = es ist ein Timeout beim Empfang oder Handshake aufgetreten.
SLIPfrm = es ist ein Protokoll Fehler beim Empfang aufgetreten.
SLIPchkE = es ist ein Checksummen Fehler beim Empfang aufgetreten.

```
tSLIPmodeEn   = (slpHsk, slpChkS, slpAddr);
tSLIPmode     = Bitset of tSLIPmodeEn;
```

Dieser Aufzählungstyp bzw. das daraus gebildete Bitset dient zum Einstellen der Betriebsart des Treibers. Alle receiver/transmitter, master/slaves, müssen im gleichen Mode arbeiten!

3.19.4 Vars

Zur Unterstützung von Multitasking wird eine jeweils eine Semaphore exportiert:
SlipRxSema1 : semaphore;
SlipRxSema2 : semaphore;
SlipRxSema3 : semaphore;
SlipRxSema4 : semaphore;
XMega
SlipRxSemaC0 : semaphore;
SlipRxSemaC1 : semaphore;
SlipRxSemaD0 : semaphore;
SlipRxSemaD1 : semaphore;
SlipRxSemaE0 : semaphore;
SlipRxSemaE1 : semaphore;
SlipRxSemaF0 : semaphore;
SlipRxSemaF1 : semaphore;
Wurde ein Frame empfangen, egal ob korrekt oder fehlerhaft, wechselt diese Semaphore ihren Wert von 0 auf 1. Damit kann ein Prozess schlafen/warten bis ein Frame eingetroffen ist. Beispiel:

```
WaitSema(SlipRxSema1);
```
XMega
```
WaitSema(SlipRxSemaC0);
```

3.19.5 Funktionen

Die folgenden Funktionen dienen in erster Linie zur ***Initialisierung*** des Treibers:

function *SLIPsetMode1(sMode : tSLIPmode) : boolean;*
function *SLIPsetMode2(sMode : tSLIPmode) : boolean;*
function *SLIPsetMode3(sMode : tSLIPmode) : boolean;*
function *SLIPsetMode4(sMode : tSLIPmode) : boolean;*
XMega
function *SLIPsetModeC0(sMode : tSLIPmode) : boolean;*
function *SLIPsetModeC1(sMode : tSLIPmode) : boolean;*
function *SLIPsetModeD0(sMode : tSLIPmode) : boolean;*
function *SLIPsetModeD1(sMode : tSLIPmode) : boolean;*
function *SLIPsetModeE0(sMode : tSLIPmode) : boolean;*
function *SLIPsetModeE1(sMode : tSLIPmode) : boolean;*
function *SLIPsetModeF0(sMode : tSLIPmode) : boolean;*
function *SLIPsetModeF1(sMode : tSLIPmode) : boolean;*
Diese Funktion dient zum Einstellen der Betriebsart des Treibers. Mit dem Bitset Parameter wird die Arbeitsweise des Treibers eingestellt. Es ist damit möglich die Konfiguration beliebig zu kombinieren aus Handshake, Checksumme und Adressprüfung. Die Funktion kehrt mit einem false zurück wenn Rx oder Tx noch busy ist.
Achtung: im Handshake Betrieb arbeitet der Treiber im Halb-Duplex Mode. D.h. solange das Packet inkl. Handshake nicht komplett abgearbeitet wurde, darf kein weiteres Packet gesendet werden, von keinem der Beteiligten.

procedure *SLIPsetRxAddr1(rxAddr : byte);*
procedure *SLIPsetRxAddr2(rxAddr : byte);*
procedure *SLIPsetRxAddr3(rxAddr : byte);*
procedure *SLIPsetRxAddr4(rxAddr : byte);*
XMega
procedure *SLIPsetRxAddrC0(rxAddr : byte);*
procedure *SLIPsetRxAddrC1(rxAddr : byte);*
procedure *SLIPsetRxAddrD0(rxAddr : byte);*
procedure *SLIPsetRxAddrD1(rxAddr : byte);*
procedure *SLIPsetRxAddrE0(rxAddr : byte);*
procedure *SLIPsetRxAddrE1(rxAddr : byte);*
procedure *SLIPsetRxAddrF0(rxAddr : byte);*
procedure *SLIPsetRxAddrF1(rxAddr : byte);*

Diese Prozedur gibt die interne Rx Adresse vor. Diese wird benötigt wenn im Mode der Wert *slpAddr* aktiv ist. Diese Adresse kann jederzeit geändert werden, ist aber bei anderen Modes ohne Bedeutung.

```
procedure SLIPsetTxAddr1(txAddr : byte);
procedure SLIPsetTxAddr2(txAddr : byte);
procedure SLIPsetTxAddr3(txAddr : byte);
procedure SLIPsetTxAddr4(txAddr : byte);
```
XMega
```
procedure SLIPsetTxAddrC0(txAddr : byte);
procedure SLIPsetTxAddrC1(txAddr : byte);
procedure SLIPsetTxAddrD0(txAddr : byte);
procedure SLIPsetTxAddrD1(txAddr : byte);
procedure SLIPsetTxAddrE0(txAddr : byte);
procedure SLIPsetTxAddrE1(txAddr : byte);
procedure SLIPsetTxAddrF0(txAddr : byte);
procedure SLIPsetTxAddrF1(txAddr : byte);
```
Diese Prozedur gibt die Ziel Tx Adresse vor. Diese wird benötigt wenn im Mode der Wert *slpAddr* aktiv ist. Diese Adresse kann jederzeit geändert werden, ist aber bei anderen Modes ohne Bedeutung.

```
function SLIPsetRxBuffer1(rxBuff : pointer; size : word) : boolean;
function SLIPsetRxBuffer2(rxBuff : pointer; size : word) : boolean;
function SLIPsetRxBuffer3(rxBuff : pointer; size : word) : boolean;
function SLIPsetRxBuffer4(rxBuff : pointer; size : word) : boolean;
```
XMega
```
function SLIPsetRxBufferC0(rxBuff : pointer; size : word) : boolean;
function SLIPsetRxBufferC1(rxBuff : pointer; size : word) : boolean;
function SLIPsetRxBufferD0(rxBuff : pointer; size : word) : boolean;
function SLIPsetRxBufferD1(rxBuff : pointer; size : word) : boolean;
function SLIPsetRxBufferE0(rxBuff : pointer; size : word) : boolean;
function SLIPsetRxBufferE1(rxBuff : pointer; size : word) : boolean;
function SLIPsetRxBufferF0(rxBuff : pointer; size : word) : boolean;
function SLIPsetRxBufferF1(rxBuff : pointer; size : word) : boolean;
```
Diese Funktion gibt den Empfangs Buffer und dessen Grösse vor. Die Grösse muss mindestens so gross sein wie das grösste zu erwartende Packet. Wenn ein reinkommendes Packet grösser ist wird es verworfen und ein Overrun Error gesetzt. Ist *slpChkS* aktiv so wird die Checksumme als letztes Byte in den Rx Buffer abgelegt. Damit muss der Buffer dann um mindestens ein Byte grösser sein als das grösste zu erwartende Packet. Ist der Rx Status des Treibers auf *SLIPbusy* dann wird die Funktion nicht ausgeführt und kehrt mit einem false zurück.

```
function SLIPsetTxBuffer1(txBuff : pointer; size : word) : boolean;
function SLIPsetTxBuffer2(txBuff : pointer; size : word) : boolean;
function SLIPsetTxBuffer3(txBuff : pointer; size : word) : boolean;
function SLIPsetTxBuffer4(txBuff : pointer; size : word) : boolean;
```
XMega
```
function SLIPsetTxBufferC0(txBuff : pointer; size : word) : boolean;
function SLIPsetTxBufferC1(txBuff : pointer; size : word) : boolean;
function SLIPsetTxBufferD0(txBuff : pointer; size : word) : boolean;
function SLIPsetTxBufferD1(txBuff : pointer; size : word) : boolean;
function SLIPsetTxBufferE0(txBuff : pointer; size : word) : boolean;
function SLIPsetTxBufferE1(txBuff : pointer; size : word) : boolean;
function SLIPsetTxBufferF0(txBuff : pointer; size : word) : boolean;
function SLIPsetTxBufferF1(txBuff : pointer; size : word) : boolean;
```
Diese Funktion gibt den Sende Buffer und dessen Grösse vor. “size“ bestimmt auch die Länge des zu sendenden Packets. Ist der Tx Status des Treibers auf *SLIPbusy* dann wird die Funktion nicht ausgeführt und kehrt mit einem false zurück.

procedure *SLIPsetTimeOut1(TimeOut : byte);*
procedure *SLIPsetTimeOut2(TimeOut : byte);*
procedure *SLIPsetTimeOut3(TimeOut : byte);*
procedure *SLIPsetTimeOut4(TimeOut : byte);*
XMega
procedure *SLIPsetTimeOutC0(TimeOut : byte);*
procedure *SLIPsetTimeOutC1(TimeOut : byte);*
procedure *SLIPsetTimeOutD0(TimeOut : byte);*
procedure *SLIPsetTimeOutD1(TimeOut : byte);*
procedure *SLIPsetTimeOutE0(TimeOut : byte);*
procedure *SLIPsetTimeOutE1(TimeOut : byte);*
procedure *SLIPsetTimeOutF0(TimeOut : byte);*
procedure *SLIPsetTimeOutF1(TimeOut : byte);*
Diese Prozedur gibt das Timeout für **Empfangs Funktionen** und das **Handshake** in SysTick vor. Bricht der Empfang eines Packets unerwartet ab, wartet der Treiber bis dieses Timeout abglaufen ist und setzt dann den Rx Status auf *SLIPtout*. Beim Handshake Betrieb wird beim Senden ebenfalls mit einem Timeout abgebrochen, wenn das Acknowledge des Empfängers ausbleibt.

Die folgenden Funktionen ***starten, stoppen*** und werten einen Transfer aus:

function *SLIPstartTx1 : boolean;*
function *SLIPstartTx2 : boolean;*
function *SLIPstartTx3 : boolean;*
function *SLIPstartTx4 : boolean;*
XMega
function *SLIPstartTxC0 : boolean;*
function *SLIPstartTxC1 : boolean;*
function *SLIPstartTxD0 : boolean;*
function *SLIPstartTxD1 : boolean;*
function *SLIPstartTxE0 : boolean;*
function *SLIPstartTxE1 : boolean;*
function *SLIPstartTxF0 : boolean;*
function *SLIPstartTxF1 : boolean;*
Diese Funktion startet einen Transfer. Die Daten die sich im *txBuff* befinden werden mit der Länge *size* (aus SLIPsetTxBuffer oder *count* aus SLIPstartTxC) gesendet. Ist der Tx Status des Treibers auf *SLIPbusy* dann wird die Funktion nicht ausgeführt und kehrt mit einem false zurück. War die Funktion erfolgreich, dann wird der Tx-Treiber Status auf *SLIPbusy* gesetzt.

function *SLIPstartTxC1(count : word) : boolean;*
function *SLIPstartTxC2(count : word) : boolean;*
function *SLIPstartTxC3(count : word) : boolean;*
function *SLIPstartTxC4(count : word) : boolean;*
XMega
function *SLIPstartTxC_C0(count : word) : boolean;*
function *SLIPstartTxC_C1(count : word) : boolean;*
function *SLIPstartTxC_D0(count : word) : boolean;*
function *SLIPstartTxC_D1(count : word) : boolean;*
function *SLIPstartTxC_E0(count : word) : boolean;*
function *SLIPstartTxC_E1(count : word) : boolean;*
function *SLIPstartTxC_F0(count : word) : boolean;*
function *SLIPstartTxC_F1(count : word) : boolean;*
Diese Funktion startet einen Transfer. Die Daten die sich im *txBuff* befinden werden mit der Länge *count* gesendet. Der Wert *count* gilt ab sofort, bis auf Widerruf mit SLIPsetTxBuffer oder SLIPstartTxC, für alle weiteren Sende Operation. Ist der Tx Status des Treibers auf *SLIPbusy* dann wird die Funktion nicht ausgeführt und kehrt mit einem false zurück. War die Funktion erfolgreich, dann wird der Tx-Treiber Status auf *SLIPbusy* gesetzt.

function *SLIPresumeRx1 : boolean;*
function *SLIPresumeRx2 : boolean;*
function *SLIPresumeRx3 : boolean;*
function *SLIPresumeRx4 : boolean;*
XMega
function *SLIPresumeRxC0 : boolean;*
function *SLIPresumeRxC1 : boolean;*
function *SLIPresumeRxD0 : boolean;*
function *SLIPresumeRxD1 : boolean;*
function *SLIPresumeRxE0 : boolean;*
function *SLIPresumeRxE1 : boolean;*
function *SLIPresumeRxF0 : boolean;*
function *SLIPresumeRxF1 : boolean;*
Nach dem Programm Start und nach jedem Packet Empfang, auch fehlerhaften, muss die Applikation mit dieser Funktion den Empfang wieder freigeben. Solange das nicht geschieht, werden alle einkommenden Packete ignoriert. Ist der Rx Status des Treibers auf *SLIPbusy* dann wird die Funktion nicht ausgeführt und kehrt mit einem false zurück. War die Funktion erfolgreich, dann wird der Rx-Treiber Status auf *SLIPbusy* gesetzt.

procedure *SLIPstopRx1;*
procedure *SLIPstopRx2;*
procedure *SLIPstopRx3;*
procedure *SLIPstopRx4;*
XMega
procedure *SLIPstopRxC0;*
procedure *SLIPstopRxC1;*
procedure *SLIPstopRxD0;*
procedure *SLIPstopRxD1;*
procedure *SLIPstopRxE0;*
procedure *SLIPstopRxE1;*
procedure *SLIPstopRxF0;*
procedure *SLIPstopRxF1;*
Wurde der Empfang mit SLIPresumeRx.. freigegeben, so kann er mit dieser Prozedur wieder gestoppt werden. Dieser Aufruf ist immer möglich.

function *SLIPrxReady1 : boolean;*
function *SLIPrxReady2 : boolean;*
function *SLIPrxReady3 : boolean;*
function *SLIPrxReady4 : boolean;*
XMega
function *SLIPrxReadyC0 : boolean;*
function *SLIPrxReadyC1 : boolean;*
function *SLIPrxReadyD0 : boolean;*
function *SLIPrxReadyD1 : boolean;*
function *SLIPrxReadyE0 : boolean;*
function *SLIPrxReadyE1 : boolean;*
function *SLIPrxReadyF0 : boolean;*
function *SLIPrxReadyF1 : boolean;*
Diese Funktion ergibt ein true wenn das zuletzt empfangene Packet zum Abholen bereit ist. Auch ein defektes Packet oder TimeOut ergibt ein true. Der Status des Packets kann mit SLIPgetRxStatex abgefragt werden.

function *SLIPgetRxState1 : tSLIPstate;*
function *SLIPgetRxState2 : tSLIPstate;*
function *SLIPgetRxState3 : tSLIPstate;*
function *SLIPgetRxState4 : tSLIPstate;*
XMega
function *SLIPgetRxStateC0 : tSLIPstate;*
function *SLIPgetRxStateC1 : tSLIPstate;*
function *SLIPgetRxStateD0 : tSLIPstate;*
function *SLIPgetRxStateD1 : tSLIPstate;*
function *SLIPgetRxStateE0 : tSLIPstate;*
function *SLIPgetRxStateE1 : tSLIPstate;*
function *SLIPgetRxStateF0 : tSLIPstate;*
function *SLIPgetRxStateF1 : tSLIPstate;*
Mit dieser Funktion kann zu jeder Zeit der Rx Status des Treibers abgefragt werden. Das mögliche Resultat kann dann sein: *SLIPready, SLIPbusy, SLIPovr, SLIPtout, SLIPfrm.* Während die letzteren 3 Fehlermeldungen des letzten Empfangs sind, sind die ersten drei die normalen Stati des Rx Treibers. Erst wenn das *SLIPready* oder ein Fehler angezeigt wird darf die Applikation entsprechend verfahren. *SLIPready* ist der Ruhezustand und zeigt an, dass gültige Daten empfangen wurden.

function *SLIPwasBC1 : boolean;*
function *SLIPwasBC2 : boolean;*
function *SLIPwasBC3 : boolean;*
function *SLIPwasBC3 : boolean;*
XMega
function *SLIPwasBC_C0 : boolean;*
function *SLIPwasBC_C1 : boolean;*
function *SLIPwasBC_D0 : boolean;*
function *SLIPwasBC_D1 : boolean;*
function *SLIPwasBC_E0 : boolean;*
function *SLIPwasBC_E1 : boolean;*
function *SLIPwasBC_F0 : boolean;*
function *SLIPwasBC_F1 : boolean;*
Diese Funktion ergibt ein true wenn das zuletzt empfangene Packet ein Broadcast Packet war.

function *SLIPgetRxCount1 : word;*
function *SLIPgetRxCount2 : word;*
function *SLIPgetRxCount3 : word;*
function *SLIPgetRxCount4 : word;*
XMega
function *SLIPgetRxCountC0 : word;*
function *SLIPgetRxCountC1 : word;*
function *SLIPgetRxCountD0 : word;*
function *SLIPgetRxCountD1 : word;*
function *SLIPgetRxCountE0 : word;*
function *SLIPgetRxCountE1 : word;*
function *SLIPgetRxCountF0 : word;*
function *SLIPgetRxCountF1 : word;*
Ist der Rx Status *SLIPready* so kann jederzeit mit dieser Funktion der Byte Count des empfangenen Packets abgefragt werden. Eine evtl. enthaltene Checksumme oder Adresse werden nicht mitgezählt.

XMega
Procedure *SLIPresetC0;*
Procedure *SLIPresetC1;*
Procedure *SLIPresetD0;*
Procedure *SLIPresetD1;*
Procedure *SLIPresetE0;*
Procedure *SLIPresetE1;*
Procedure *SLIPresetF0;*
Procedure *SLIPresetF1;*
Hiermit wird der SIP Treiber (Soft und Hardware) komplett zurückgesetzt und neu-initialisiert.

```
function SLIPgetTxState1 : tSLIPstate;
function SLIPgetTxState2 : tSLIPstate;
function SLIPgetTxState3 : tSLIPstate;
function SLIPgetTxState4 : tSLIPstate;
XMega
function SLIPgetTxStateC0 : tSLIPstate;
function SLIPgetTxStateC1 : tSLIPstate;
function SLIPgetTxStateD0 : tSLIPstate;
function SLIPgetTxStateD1 : tSLIPstate;
function SLIPgetTxStateE0 : tSLIPstate;
function SLIPgetTxStateE1 : tSLIPstate;
function SLIPgetTxStateF0 : tSLIPstate;
function SLIPgetTxStateF1 : tSLIPstate;
```

Mit dieser Funktion kann zu jeder Zeit der Tx Status des Treibers abgefragt werden. Das mögliche Resultat kann dann sein: *SLIPready, SLIPbusy, SLIPtout.* Während *SLIPtout* nur auftritt wenn während eines Handshake ein Timeout aufgetreten ist, sind die ersten zwei die normalen Stati des Tx Treibers. Erst wenn das *SLIPready* oder ein Fehler angezeigt wird darf die Applikation entsprechend verfahren. *SLIPready* zeigt an, dass die letzte Sende Operation ohne Fehler durchgeführt wurde.

Der Treiber benötigt ca. 1.5kBytes an Code.

Ein kleines Demo und Testprogramm “AVR SLIPport” befindet sich in der Demos Directory in “SLIPport”.
Ein weiteres Demo und Testprogramm “AVR SLIP_PC” befindet sich in der Demos Directory in “SLIPport”.
XMega Ein kleines Demoprogramm "XMega_SLIP" ist in der Demos Directory unter "XMega_SLIPport"
Das letztere kommuniziert mit dem E-LAB Dual Terminal Programm, das um die SLIP Funktion erweitert ist.

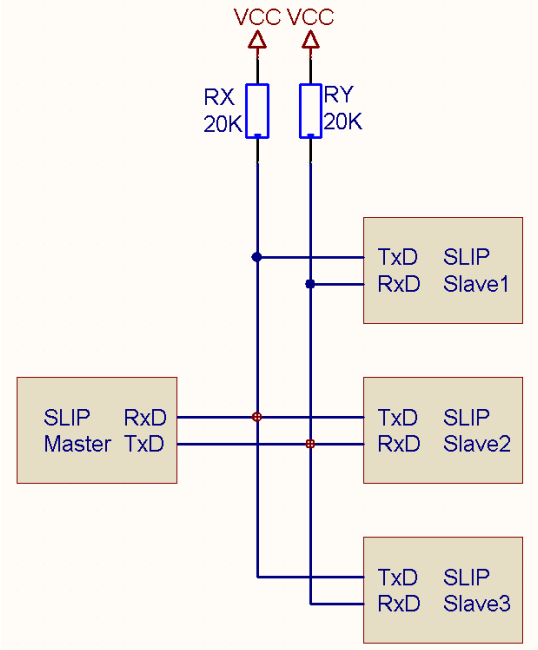
Eine rudimentäre Delphi SLIP Unit ist ebenfalls in der Demos Directory in “SLIPport” enthalten.

3.19.6 SLIP Local Area Network (XMega)

Der SLIP Treiber eignet sich auch hervorragend für schnelle on-board Netzwerke. Dazu werden keinerlei Leitungstreiber (RS232/RS485) benötigt! Es muss nur sichergestellt werden, dass immer nur ein TxD Pin eine Leitung treibt. Dazu wird das Define erweitert:

```
Define  SLIPportD0      = 19200, Master;       // onBoard local network
```
oder
```
Define  SLIPportD0      = 19200, Slave;        // onBoard local network
```

Damit wird bei der Initialisierung nur der TxD Pin auf output gestellt, der das Attribut “Master” hat. Jeder Teilnehmer an diesem Netzwerk muss entweder das Attribut Master oder Slave haben. Es ist allerdings nur ein Master möglich.



Das Netzwerk ist ein Master/Slave Netzwerk. Es gibt nur einen Master. Dieser sendet Aufträge an die Slaves und empfängt die evtl. Antworten.

Dazu schaltet der Master sein TxD Pin auf Output und sendet die Kommandos/Daten. Anschliessend schaltet er seinen TxD Pin auf Input und wartet auf die (mögliche) Antwort.

Der angesprochene Slave schaltet jetzt seinen TxD Pin auf Output und sendet die Antwort. Danach setzt er seinen TxD Pin auf Input und der Master, wenn er die Antwort erhalten hat, setzt seinen TxD Pin wieder auf Output.

Damit wird gewährleistet, dass es auf den Leitungen keine Kollission gibt.

Das *Define SLIPportCtrlXX* darf hier nicht definiert werden.

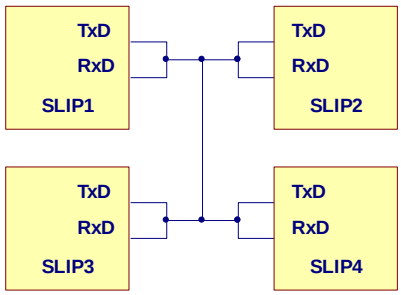
3.19.6.1 Single-Wire Half-Duplex

Eine weitere Möglichkeit eines on-board SLIP Netzwerkes besteht darin, dass man nur eine Leitung verwendet. Dies nennt man Single-Wire (Eindraht) Verbindung. Im Gegensatz zu obiger Implementation kann hier jeder Master oder Slave sein. Allerdings muss hier sehr genau geplant werden um Kollisionen zu vermeiden!

Die Implementation sieht hier so aus:

```
Define  SLIPportC0      = 19200;          // onBoard local network
        SLIPportCtrlC0  = SingleWire;     // defines a single-wire or half-duplex connection
```

Elektrische Implementation:



Achtung:
Das Handshake Verfahren ist in diesem Modus noch nicht implementiert!

3.20 MIRF Treiber

Beim Betreiben von Sensoren, Aktuatoren und allgemeinem Datenaustausch stellt sich immer wieder das Problem langer Verkabelung, galvanischer Trennung und Erdschleifen bzw. Masse Verkopplungen. Es bietet sich hierzu eine optisch getrennte Verbindung an (Optokoppler oder Glasfaser). Dadurch bleibt weiterhin das Problem der Verkabelung. Auch Ethernet ist wegen seiner mangelhaften Erdfreiheit und vor allen Dingen wegen der Kosten nicht geeignet.

3.20.1 Funkstrecke

Eine Mögliche Lösung dieser Probleme ist eine Datenfunkstrecke, wie z.B. WLAN, Bluetooth oder ZigBee. Diese benötigen jedoch einen unvertretbaren grossen Treiber Aufwand, ja sogar Overhead, auf der Controller Seite.

Da aber das 2.4GHz ISM Band lizenzfrei ist bietet sich dieses an. Leider wird dieser Bereich jedoch auch von WLANs, Bluetooth, ZigBee, DECT Telefone, Babyphones und auch Mikrowellen Öfen benutzt. ISM steht dabei für **I**ndustrial, **S**cientific, and **M**edical Band.

Wie aus der obigen Aufzählung der Dienste zu ersehen ist, kann es daher sehr eng zugehen.

3.20.2 Belegung des ISM Bands 2.4GHz

Hier

Channel	Frequency (GHz)	Permitted	Notes
1	2,412 WLAN	USA FCC, Europa ETSI, Japan	
2	2,417	USA FCC, Europa ETSI, Japan	
3	2,422	USA FCC, Europa ETSI, Japan	
4	2,427	USA FCC, Europa ETSI, Japan	
5	2,432	USA FCC, Europa ETSI, Japan	
6	2,437 WLAN	USA FCC, Europa ETSI, Japan	
7	2,442	USA FCC, Europa ETSI, Japan	
8	2,447	USA FCC, Europa ETSI, Japan	
9	2,452	USA FCC, Europa ETSI, Japan	Microwave Oven
10	2,457	USA FCC, Europa ETSI, Japan	Microwave Oven
11	2,462 WLAN	USA FCC, Europa ETSI, Japan	
12	2,467	Europa ETSI, Japan	
13	2,472	Europa ETSI, Japan	
14	2,484	Japan	

sieht man die offiziell freigegeben Kanäle bzw. Frequenzen. Dabei gibt es eine grundsätzliche Schwierigkeit: die 13 WLAN-Kanäle sind nicht sauber voneinander getrennt, sondern überlappen sich ihrerseits. Jeder WLAN-Kanal hat eine so genannte Bandbreite von drei Kanälen. Das heißt, dass Kanal Nummer 2 teilweise in die Frequenzen von den Kanälen Nummer 1 und 3 hineinfunkt. Vollkommen überlappungsfrei funken deshalb nur die WLAN-Kanäle 1, 6 und 11.

Besonders anfällig für die Störung durch Mikrowellenherde sind im übrigen die WLAN-Kanäle 9 und 10. Ihre Trägerfrequenzen (2,452 und 2,457 GHz) liegen sehr nah an der "Mikrowellen-Frequenz" 2,455 GHz.

Deshalb empfiehlt es sich dringend beim Einrichten solcher Verbindungen mit einem Scanner oder Spektrum Analysator das ISM Band zu beobachten um freie Kanäle zu finden. Die typischen PC Tools sind dazu absolut nicht geeignet, da diese nur reine WLANs erkennen, aber kein Bluetooth, ZigBee etc.

3.20.3 MIRF

Medical **I**ndustrial **R**adio **F**requency

Mit MIRF bezeichnet man die low-level Schicht von IEEE 802.15.xx, in dessen Untergruppierungen sind WLAN, Zigbee etc. angesiedelt

Der Standard **IEEE 802.15.4** beschreibt ein Übertragungsprotokoll für Wireless Personal Area Networks (WPAN). Er definiert die untersten beiden Schichten des OSI-Modells, den Bitübertragungs- und den MAC-Layer. Höhere Protokollebenen mit Funktionen zum Routing und einer Anwendungsschnittstelle obliegen anderen Standards für Funknetze wie ZigBee. Wesentliche Entwicklungsziele für das Protokoll sind geringe Leistungsaufnahme für einen langen Betrieb über Batterieversorgung, kostengünstige Hardware, sichere Übertragung, Nutzung der lizenzfreien ISM-Bänder und Parallelbetrieb mit anderen Sendern auf diesen Frequenzen, insbesondere WLAN und Bluetooth. Durch diese Eigenschaften eignet sich der Standard IEEE 802.15.4 vor allem für drahtlose Sensornetze (WSN).

Ende der 1990er Jahre wurde ein Bedarf für einen einfachen Standard zur drahtlosen Datenübertragung für Geräte mit geringer Leistungsaufnahme und niedriger Datenübertragungsrate gesehen. Die damals verfügbaren Standards IEEE 802.11 und Bluetooth waren zu komplex und besaßen einen zu großen Energiebedarf, um sie mit kostengünstigen Bauteilen implementieren zu können. Bei der Entwicklung von IEEE 802.15.4 besaß folglich nicht eine hohe Datenübertragungsrate, sondern das Energiemanagement und die Einfachheit des standardisierten Protokolls höchste Priorität.

Charakteristisch für die Knoten eines IEEE 802.15.4 Netzes sind die langen Ruhephasen, wodurch ein Knoten die meiste Zeit in einem energiesparenden Betriebszustand verweilen kann. Sobald er Daten senden oder empfangen möchte, kann er in lediglich 15 ms aufwachen, anschließend die Kommunikation abwickeln und sich wieder schlafen legen. Dadurch können batteriebetriebene Netzknoten typische Laufzeiten von sechs Monaten bis zu zwei Jahren erreichen.

3.20.3.1 Tx-Power

Die typische Sendeleistung eines Transceivers beträgt 0dBm (=1mW) und die Empfindlichkeit liegt unter -90 dBm. Auch wenn das vorgegebene Modell der Pfadverluste für Gebäude gilt, zeigt die Praxis, dass oft nur ein Drittel der angegebenen Reichweite möglich ist. IEEE 802.15.4 wurde für den Parallelbetrieb mit WLAN und Bluetooth ausgelegt. Praxis-Tests zeigten im 2,45 GHz-Band Probleme bei der Koexistenz von IEEE 802.15.4 mit WLAN und Bluetooth. Bei Bluetooth erweist sich das seit der Version 1.2 eingeführte adaptive Frequenzhopping, das WLAN ausweicht aber dafür die verbleibenden Frequenzen häufiger belegt, als Störer. WLAN macht durch den stark gewachsenen Datenverkehr Probleme.

3.20.3.2 Topologie

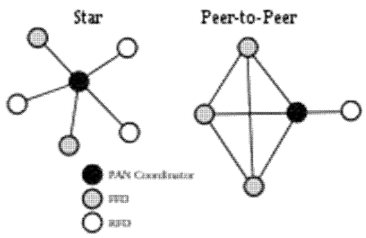
Der IEEE 802.15.4 sieht zwei Typen von Netzknoten mit unterschiedlichen Funktionsumfang vor, die Reduced Function Devices (RFD) und die Full Function Devices (FFD). Ein RFD besitzt nur eine Teilmenge des standardisierten Funktionsumfangs, wodurch ihm nur die Kommunikation mit FFDs möglich ist, er jedoch auch einfacher und kostengünstiger entwickelt werden kann. RFDs sind typisch Sensoren oder Aktoren im Netzwerk, die nur sehr selten Daten senden oder empfangen, keine Verwaltungsaufgaben übernehmen und so die meiste Zeit in einem stromsparenden Zustand verweilen. FFDs hingegen besitzen den vollen Funktionsumfang und können sowohl mit RFDs als auch mit anderen FFDs kommunizieren. Ein FFD pro Netz übernimmt die spezielle Funktion des PAN Koordinators. Er legt den PAN Identifier fest, der das Netzwerk von anderen IEEE 80.15.4 Netzen in Funkreichweite abgrenzt. Des weiteren übernimmt er im Slotted Mode die Synchronisation aller Netzknoten. Ein Netz kann bis zu 254 Knoten besitzen.

Der Standard definiert aufbauend auf die genannten Knotentypen drei verschiedene Netztopologien:

- **Stern**. In einem Stern kommunizieren alle Knoten direkt mit dem Koordinator. Der Koordinator ist in solch einer Konstellation gewöhnlich ein leistungsfähiges Gerät mit Anbindung ans Stromnetz, während die anderen Knoten batteriebetrieben sind.

- **Peer-to-Peer**. In diesem Netz gibt es zwar ebenfalls einen Koordinator, jedoch können die Knoten auch direkt untereinander kommunizieren, sofern sie sich in gegenseitiger Reichweite befinden.

- **Baumstruktur (Cluster Tree)**. Darauf wird hier nicht weiter eingegangen.



Da der Standard keine Vermittlungsschicht (Network Layer) definiert, müssen Funktionen wie Routing durch höhere Schichten anderer Protokolle realisiert werden, die auf IEEE 802.15.4 aufsetzen. Damit sind echte vermaschte Netze möglich, in denen die FFDs als Repeater dienen und jeder Knoten über Zwischenstationen mit einem anderen kommunizieren kann, sogar RFDs mit anderen RFDs.

3.21 MIRF24port

Ein MIRF Treiber ist natürlich an einen bestimmten Chip gebunden. Das ist hier der nrf24l01+ von Nordic Semiconductor. Dieser bietet ein paar hervorragende Eigenschaften, die der Treiber ausnutzt:

- Worldwide 2.4GHz ISM band operation
- 250kbps, 1Mbps and 2Mbps on air data rates
- Ultra low power operation
- 11.3mA TX at 0dBm output power
- 13.5mA RX at 2Mbps air data rate
- 900nA in power down
- 26µA in standby-I
- Automatic CRC generation/check
- Automatic handshake without CPU intervention
- Automatic multiple retries
- Automatic packet handling
- Auto packet transaction handling
- 6 data pipe MultiCeiver™
- GFSK modulation
- 250kbps, 1 and 2Mbps air data rate
- 1MHz non-overlapping channel spacing at 1Mbps
- 2MHz non-overlapping channel spacing at 2Mbps
- Programmable output power: 0, -6, -12 or -18dBm
- -82dBm sensitivity at 2Mbps
- -85dBm sensitivity at 1Mbps
- -94dBm sensitivity at 250kbps
- 1 to 32 bytes dynamic payload length
- Max 10Mbps SPI interface
- 3 separate 32 bytes TX and RX FIFOs
- 5V tolerant inputs
- 4x4mm

3.21.1 MIRF24 Treiber

Ein MIRF Treiber ist natürlich an einen bestimmten Chip gebunden. Das ist hier der nrf24L01+ von Nordic dessen Eigenschaften wesentlich zu einem sicheren und schnellen Betrieb beitragen.
Als Betriebsart ist sowohl Peer-to-Peer als auch Stern Netzwerk möglich.

3.21.1.1 Imports

Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import  SysTick…, MIRF24port, ..;        // driver import
```

Der SysTick wird für diesen Treiber nicht zwingend benötigt.

3.21.1.2 Defines

```
Define
    ProcClock      = 16000000;      {Hertz}
    SysTick        = 10;            {msec}
    StackSize      = $0040, iData;
    FrameSize      = $0040, iData;

    MIRF24port     = SPI_Soft, PortA, 2,   3,       4,     1,   0,    5;
                                  // SCK, MOSI, MISO, SS, CE, IRQ
  //  MIRF24port   = SPI, PortA,  0,   1,    2;                    // standard SPI port
                                // SS, CE, IRQ
  //  MIRF24port   = MSPI_2,  PortA,  0,  1,   2;                   // MSPI_0..MSPI_3
                                    // SS, CE, IRQ      …
```

Die MIRF Hardware wird über ein SPIport gesteuert. Es sind dabei drei Typen implementiert:
1. Software SPI über ein Standard IO-Port
2. Hardware SPI über den Standard SPI des Controllers. XMega SPI_C, SPI_D, SPI_E, SPI_F
3. Hardware SPI über SPI-schaltbare UARTs des Controllers, falls vorhanden.
Abhängig von SPI Typ müssen auch noch diverse Steuerleitungen definiert werden.

Die Unit uMIRF24 muss importiert werden.

```
uses uMIRF24;
```

3.21.1.3 Typen

Es werden diverse Typen von der Unit uMIRF24 exportiert:

```
tMRFchan      = (mrfChan1, mrfChan2, mrfChan3, mrfChan4, mrfChan5,
                 mrfChan6, mrfChan7, mrfChan8, mrfChan9, mrfChan10,
                mrfChan11, mrfChan12, mrfChan13, mrfChan14);

tMRFpwr       = (mrfdBm0, mrfdBm6, mrfdBm12, mrfdBm18);

enMRFstat     = (mrfTX_full, mrfRX_pn0, mrfRX_pn1, mrfRX_pn2,
                 mrfMAX_RT, mrfTX_DS, mrfRX_DR);
tMRFstat      = Bitset of enMRFstat;

tMRFpkt       = (mrfPKTnone, mrfPKTdata, mrfPKTbcast);
tMRFrfSpeed   = (mrfRF250, mrfRF1000, mrfRF2000);
```

3.21.1.4 Liste der Funktionen und Prozeduren

Die folgenden Funktionen dienen in erster Linie zur ***Initialisierung*** des Treibers:

```
procedure mrfSetChan(chan : tMRFchan; wr : boolean);
procedure mrfSetFreq(freq : word; wr : boolean);
procedure mrfSetPower(pwr : tMRFpwr);
procedure mrfSetLocalAddr(adr : byte);
procedure mrfSetRetryMax(rmax : byte);
procedure mrfSetRetryTimeOut(tmo : byte);
procedure mrfSetRFspeed(spd : tMRFrfSpeed);
function   mrfInit : boolean;
procedure mrfSetPWRdown;
```

Diese Funktionen liefern aktuelle **Zustände, Ereigniszähler** und **Errors** zurück:

```
function  mrfGetState : tMRFstat;
function  mrfGetLostPkts : byte;
function  mrfGetRetryCnt : byte;
function  mrfGetRxPower : byte;
```

Diese Funktionen sind die eigentlichen **Arbeits Funktionen**:

```
function  mrfGetRxType : tMRFpkt;
function  mrfTxPacket(adr : byte; srcPtr : pointer; cnt : byte; bc : boolean) : boolean;
function  mrfRxPacket(destPtr : pointer; TimeOut : byte; var recvd : byte) : tMRFpkt;
```

Zwei Demo und Test Programme “AVR MIRF24_M” und “AVR MIRF24_S” befinden sich in der Demos Directory in “MIRF24”.

3.21.1.5 Funktionen und Prozeduren

Die folgenden Funktionen dienen in erster Linie zur ***Initialisierung*** des Treibers. Vor dem Aufruf der Funktion mrfInit müssen alle RxTx-Parameter korrekt mit den zugehörigen Prozeduren gesetzt werden.

procedure *mrfSetChan(chan : tMRFchan; wr : boolean);*
Die Prozedur stellt den gewünschten RF Kanal im ISM Band ein. Der Kanal chan wird mit der Enumeration tMRFchan angegeben (mrfChan1.. mrfChan14). Das boolean wr bestimmt dabei, ob der Kanal sofort eingestellt werden soll. Vor dem ersten Init muss wr false sein. Nach einem Init kann der Kanal zur Laufzeit mit wr = true umgestellt werden, ohne ein Init durchzuführen. Alternativ kann auch die Prozedur mrfSetFreq benutzt werden.

procedure *mrfSetFreq(freq : word; wr : boolean);*
Die Prozedur stellt die gewünschte Frequenz im ISM Band ein. Die Frequenz freq wird in MHz angegeben. Gültige Werte sind dabei 2400.. 2484. Das boolean wr bestimmt dabei, ob die Frequenz sofort eingestellt werden soll. Vor dem ersten Init muss wr false sein. Nach einem Init kann die Frequenz zur Laufzeit mit wr = true umgestellt werden, ohne ein Init durchzuführen. Alternativ kann auch die Prozedur mrfSetChan benutzt werden.

procedure *mrfSetPower(pwr : tMRFpwr);*
Die Prozedur stellt die gewünschte Ausgangsleistung (RF power) ein. Der power wird mit der Enumeration tMRFpwr angegeben (mrfdBm0.. mrfdBm18). mrfdBm0 ist max. Power und mrfdBm18 ist minimal power.

procedure *mrfSetLocalAddr(adr : byte);*
Die Prozedur stellt die lokale Adresse (0..255) ein. Das ist die logische Adresse mit der andere Nodes diesen Node ansprechen müssen. Diese Adresse muss unique sein, d.h. sie darf nur einmal im Netzwerk vorkommen.

procedure *mrfSetRetryMax(rmax : byte);*
Die Prozedur stellt die max. Retry Anzahl ein (0..15). Tritt beim Datentransfer ein Fehler auf, z.B. das Acknowledge des Empfängers kommt nicht rechtzeitig oder gar nicht, dann wiederholt der Sender das Telegramm bis zu rmax mal. Fehlt das ACK dann immer noch, kommt die Sendefunktion mrfTxPacket mit einem false zurück.

procedure *mrfSetRetryTimeOut(tmo : byte);*
Die Prozedur stellt das Delay zwischen zwei Retries ein (1..15). Diese Zeit rechnet sich so:
250usec + (tmo * 250usec). Typische Werte sollten bei 1msec liegen (tmo = 3..4)

procedure *mrfSetRFspeed(spd : tMRFrfSpeed);*
Die Prozedur stellt die Datenrate “on air“ ein. Die Rate wird mit der Enumeration tMRFrfSpeed angegeben (mrfRF250, mrfRF1000, mrfRF2000).

function *mrfInit : boolean;*
Die Funktion initialisiert den 24L01 mit den oben vorgegebenen Werten und stellt den Empfang aktiv. Bei einem Fehler wird ein false zurückgegeben.

procedure *mrfSetPWRdown;*
Die Prozedur schaltet den Controller 24L01 in den Power-down Mode. Für einen Neustart genügt dann der Aufruf der Funktion mrfInit.

Diese Funktionen liefern aktuelle **Zustände, Ereigniszähler** und **Errors** zurück:

function *mrfGetState : tMRFstat;*
Diese Funktion gibt das Status Registers des 24L01 zurück. tMRFstat = BitSet of enMRFstat.
Wird normalerweise nicht gebraucht. Die Bedeutung der Bits ist dem 24L01 Datenblatt zu entnehmen.

function *mrfGetLostPkts : byte;*
Die Funktion gibt die Anzahl der trotz Retries nicht zustellbaren Pakete zurück. Gleichzeitig wird der interne Counter im 24L01 zurückgesetzt.

function *mrfGetRetryCnt : byte;*
Nach einer erfolgreichen Sende Operation kann damit die Anzahl der benötigten Retries ausgelesen werden. Je höher dieser Wert ist, desto schlechter war die Verbindung.

function *mrfGetRxPower : byte;*
Die Funktion liefert im Bit0 die Empfangsqualität zurück. 0 = schlechter Empfang, 1 = guter Empfang.

Diese Funktionen sind die eigentlichen **Arbeits Funktionen**:

function *mrfGetRxType : tMRFpkt;*
Die Funktion gibt den Empfangs Status zurück. tMRFpkt = (mrfPKTnone, mrfPKTdata, mrfPKTbcast) .
Sie dient zum Pollen des Empfangs.

function *mrfRxPacket(destPtr : pointer; TimeOut : byte; var recvd : byte) : tMRFpkt*
Die Funktion versucht ein Packet abzuholen. destPtr muss auf eine RAM Datenstruktur mit der Grösse 32bytes zeigen. TimeOut gibt die Zeit in msec an, die bis zum Erfolg bzw. Abbruch gewartet werden soll. In recvd steht die tatsächlich empfangene Anzahl der Bytes (max. 32).

Das Ergebnis ist tMRFpkt = (mrfPKTnone, mrfPKTdata, mrfPKTbcast) .
Bei einem Timeout erfolgt ein mrfPKTnone. Wurde ein Standard Datenpaket empfangen, so kommt ein mrfPKTdata zurück und bei einem Broadcast ein mrfPKTbcast.

function *mrfTxPacket(adr : byte; srcPtr : pointer; cnt : byte; bc : boolean) : boolean;*
Die Funktion versucht ein Daten Packet oder ein Broadcast Packet zu senden. Der Parameter adr bezeichnet die logische Adresse (0..255) des gewünschten Empfängers (Node). srcPtr muss auf die Quelle im RAM zeigen. Der Parameter cnt bestimmt die Anzahl der zu sendenden Bytes (max. 32). Das boolean bc bestimmt ob das Packet ein Broadcast oder ein Daten Packet ist. Bei einem Broadcast wird der Parameter adr ignoriert.

Das Ergebnis wird false wenn ein Hardware Fehler vorliegt (Data oder Broadcast).
Ebenfalls ein false kommt zurück (Data) wenn der Empfänger auch nach x Sender Retries kein ACK geschickt hat bzw. das ACK verloren ging.

Der MIRF24port Treiber bietet zwei Arten von Daten Packet Transfers an:

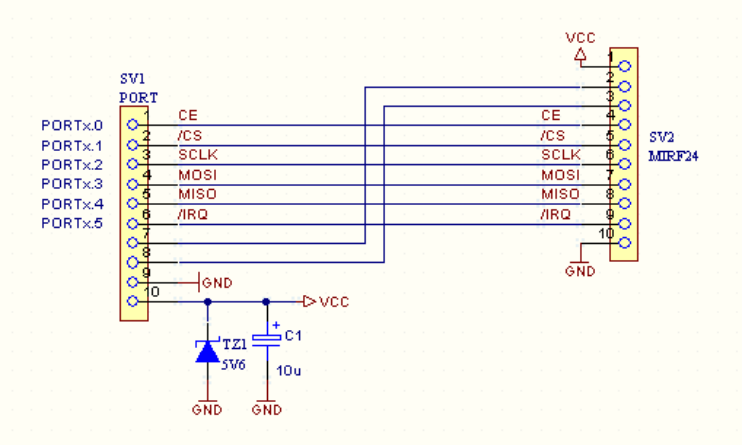
1. **Broadcast**. Dieses Packet hat eine bestimmte Kennung so dass alle erreichbaren MIRF24 Nodes dieses Packet empfangen und auswerten können. In diesem Fall ist das Hardware ACK abgeschaltet und auch der Empfänger (Applikation) sollte nicht darauf antworten.
2. **Data**. Dieses Packet geht nur an eine bestimmte Adresse (Node) und wird auch nur von diesem empfangen. Der Empfänger (Node) schickt ein automatisches Hardware ACK an den Sender zurück und quittiert damit den korrekten Empfang. Bei einem CRC Fehler etc. unterbleibt natürlich das ACK und der Sender muss ein Retry durchführen. Ein ACK vom Empfänger bedeutet zu diesem Zeitpunkt keinesfalls dass der Empfänger das Packet schon aus dem 24L01 ausgelesen hat. Solange dies nicht geschehen ist schickt der Empfänger keine Weiteren ACKs für einkommende Datenpakete so dass der Sender nach Ablauf der eingestellten Retries aus der Sende Funktion mit einem Fehler zurückkehrt.

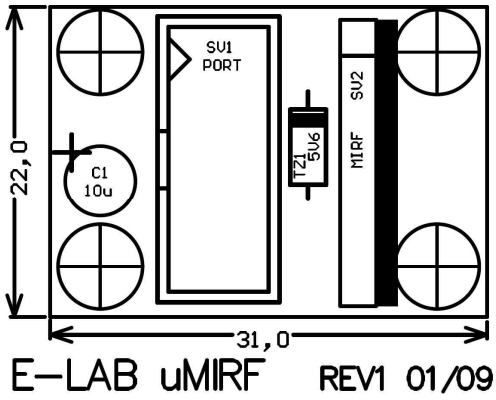
Achtung:
Vor dem Senden eines Broadcasts und danach sollte etwas Pause sein, so dass alle Empfänger bereit sind.

3.21.2 MIRF24 Hardware

3.21.2.1 MIRF24 Adapter für MIRF24 Modul und E-LAB EVA-Boards

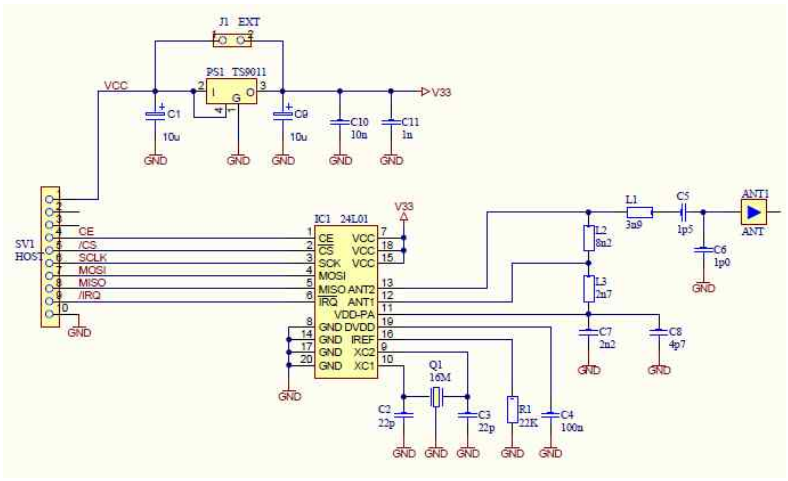
MIRFadaptor für MIRF24 und MIRF24P

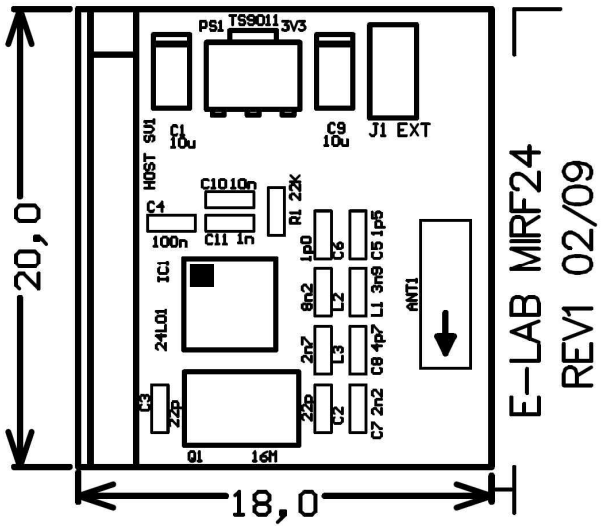




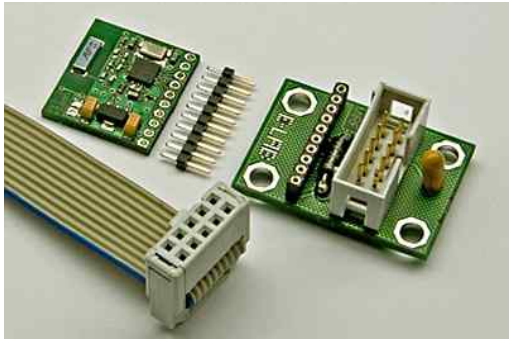
3.21.2.2 MIRF24 Transceiver Module

RF Power = 1mW = 0dBm ***Option*** MIRF24P = 100mW = 20dBm





MIRF24 Modul + Adapter



3.22 MIRF86

Ein MIRF Treiber ist natürlich an einen bestimmten Chip gebunden. Das ist hier der AT86RF231 von Atmel Semiconductor. Dieser bietet ein paar hervorragende Eigenschaften, die der Treiber ausnutzt:

• Worldwide 2.4GHz ISM band operation
• 250kbps, 1Mbps and 2Mbps on air data rates
• Ultra low power operation
• 14mA TX at +3dBm output power
• 12.3mA RX at 2Mbps air data rate
• 20nA in power down
• 400µA in standby
• Automatic CRC generation/check
• Automatic handshake without CPU intervention
• Automatic multiple retries
• Automatic packet handling
• Auto packet transaction handling
• Automatic AES encryption and decryption
• GFSK modulation
• 250kbps, 500kbs, 1 and 2Mbps air data rate
• 1MHz non-overlapping channel spacing at 1Mbps
• 2MHz non-overlapping channel spacing at 2Mbps
• Programmable output power: +3 to -17dBm (max 2mW)
• -89dBm sensitivity at 2Mbps
• -95dBm sensitivity at 1Mbps
• -101dBm sensitivity at 250kbps
• 1 to 112 bytes dynamic payload length
• Max 7Mbps SPI interface
• 3.3V operation with 3.3V inputs
• 4x4mm

Der AT86RF231 ist moderner und wesentlich besser als der Nordic Chip.
Doppelte Ausgangsleistung
Wesentlich höhere Eingangs Empfindlichkeit
PayLoad = Packet size bis 112bytes
AES Verschlüsselung

3.22.1 MIRF86 Treiber

Ein MIRF Treiber ist natürlich an einen bestimmten Chip gebunden. Das ist hier der AT86RF231 von Atmel dessen Eigenschaften wesentlich zu einem sicheren und schnellen Betrieb beitragen.
Als Betriebsart ist sowohl Peer-to-Peer als auch Stern Netzwerk möglich.

3.22.1.1 Imports

Der Treiber muss, wie beim AVRco üblich, importiert werden.

```
Import  SysTick…, MIRF86port, ..;        // driver import
```

Der SysTick wird für diesen Treiber nicht zwingend benötigt.

3.22.1.2 Defines

```
Define
    ProcClock      = 16000000;       {Hertz}
    SysTick        = 10;             {msec}
    StackSize      = $0040, iData;
    FrameSize      = $0060, iData;

    MIRF86port     = SPI_Soft, PortA.2, PortA.3, PortA.4, PortA.1, PortA.0, PortA.5;
    //                           SCK,     MOSI,   MISO,   SS,     CE,       IRQ
    // MIRF86port  = SPI, PortA.0,  PortA.1, PortA.2;            // standard SPI port
    //                        SS,       CE,      IRQ
    // MIRF86port  = MSPI_1, PortA.0, PortA.1, PortA.2;          // MSPI_0..MSPI_3
    //                           SS,      CE,      IRQ
```

Die MIRF Hardware wird über ein SPIport gesteuert. Es sind dabei drei Typen implementiert:
1. Software SPI über ein Standard IO-Port
2. Hardware SPI über den Standard SPI des Controllers. *XMega SPI_C, SPI_D, SPI_E, SPI_F*
3. Hardware SPI über SPI-schaltbare UARTs des Controllers, falls vorhanden.

Abhängig von SPI Typ müssen auch noch diverse Steuerleitungen definiert werden.

Die Unit uMIRF86 muss importiert werden.

```
uses uMIRF86;
```

3.22.1.3 Typen

Es werden diverse Typen von der Unit uMIRF86 exportiert:

```
Init
  tmrf86chan     = (mrf86Chan11, mrf86Chan12, mrf86Chan13, mrf86Chan14, mrf86Chan15,
                    mrf86Chan16, mrf86Chan17, mrf86Chan18, mrf86Chan19, mrf86Chan20,
                    mrf86Chan21, mrf86Chan22, mrf86Chan23, mrf86Chan24, mrf86Chan25,
                    mrf86Chan26);
```

Das sind 16 2.4GHz Kanäle 11..26 bzw. 2.405GHz..2.484GHz

```
  tmrf86pwr      = (mrf86dBm3P0, mrf86dBm2P8, mrf86dBm2P3, mrf86dBm1P8, mrf86dBm1P3,
                    mrf86dBm0P7, mrf86dBm0, mrf86dBm1, mrf86dBm2, mrf86dBm3, mrf86dBm4,
                    mrf86dBm5mrf86dBm5, mrf86dBm7, mrf86dBm9, mrf86dBm12, mrf86dBm17);
```

Bestimmt die TX-Ausgangsleistung. mrf86dBm3P0 = +3dBm, mrf86dBm0 = 0dBm, mrf86dBm17 = -17dBm

```
  tmrf86Speed    = (mrf86S250, mrf86S500, mrf86S1000, mrf86S2000);
```

Bestimmt die Daten Rate on-air. mrf86S250 = 250kbs, mrf86S2000 = 2Mbs

Transmit

tmrf86mode = (mrf86noBC, mrf86BC);

Bestimmt BroadCast oder noBroadCast

tmrf86AesMode = (mrf86AesNone, mrf86AesEncrypt, mrf86AesDecrypt);

Bestimmt das AES encryption. mrf86AesNone für Tx und Rx, mrf86AesEncrypt für Rx, mrf86AesDecrypt für Tx

tmrf86TxState = (mrf86TxSucces, mrf86TxInvalid, mrf86TxAccessFail, mrf86TxNoACK);

Ist das Ergebnis einer Sende Operation, res4 und res6 sind reserviert.

Receive

tmrf86pkt = (mrf86PKTBroadCast, mrf86PKTData, mrf86PKTnone);

Ergebnis einer Rx Operation, mrf86PKTBroadCast = Broadcast Packet, mrf86PKTData = normales Packet

```
tmrf86rxpacketInfo = record
                       PktTyp     : tmrf86pkt;
                       TxAddr     : Byte;
                       Crypted    : Boolean;
                       flen       : Byte;      // total byte count or length
                       Retries    : Byte;      // Retrys
                       rxdata     : Pointer;   // @ mrf86rxfrm
                       PayLoad    : Pointer;   // @ mrf86rxfrm + header= Payload start
                       PayCnt     : Byte;      // effective bytes received
                       ed         : Byte;      // field strength
                     end;
```

Dieser Record kann zur weiteren Auswertung des empfangenen Packets herangezogen werden.

3.22.1.1 Variable

Es wird eine Variable von der Unit uMIRF86 exportiert:

```
var
  mrf86rxpacketInfo    : tmrf86rxpacketInfo;
```

Dieser Record kann nach einem erfolgreichen Empfang eines Packets zur weiteren Auswertung des empfangenen Packets herangezogen werden.

3.22.1.1 Liste der Funktionen und Prozeduren

Die folgenden Funktionen dienen in erster Linie zur ***Initialisierung*** des Treibers:

```
function  mrf86Init : boolean;
procedure mrf86SetChan(chan : tmrf86chan; wr : boolean);
procedure mrf86SetFreq(freq : word; wr : boolean);
procedure mrf86SetPower(pwr : tmrf86pwr);
procedure mrf86SetPANAddr(adr : byte);       // main address
procedure mrf86SetLocalAddr(adr : byte);     // sub address
procedure mrf86SetRetryMax(rmax : byte);
procedure mrf86SetSpeed(speed : tmrf86Speed; wr : Boolean);
procedure mrf86SetPWRdown;
procedure mrf86SetKey(Const K1, K2, K3, K4, K5, K6, K7, K8, K9, K10, K11, K12, K13, K14, K15, K16 : Byte);
```

Diese Funktionen liefern aktuelle **Zustände, Ereigniszähler** und **Errors** zurück:

```
function  mrf86GetLostPkts : word;
function  mrf86GetRetryCnt : byte;
function  mrf86GetRandom : byte;
```

Diese Funktionen sind die eigentlichen **Arbeits Funktionen**:

```
function  mrf86GetRxType : tmrf86pkt;
function  mrf86TxPacket(adr, panID : byte; bc : tmrf86mode; srcPtr : pointer; Cnt : byte;
                        AESMode : tmrf86AesMode) : tmrf86TxState;
function  mrf86RxPacket(destPtr : pointer; var recvd : byte; AESMode : tmrf86AesMode) : tmrf86pkt;
```

Zwei Demo und Test Programme “AVR MIRF86_M” und “AVR MIRF86_S” befinden sich in der Demos Directory in “MIRF86”.

3.22.1.2 Funktionen und Prozeduren

Die folgenden Funktionen dienen in erster Linie zur ***Initialisierung*** des Treibers. Vor dem Aufruf der Funktion mrfInit müssen alle RxTx-Parameter korrekt mit den zugehörigen Prozeduren gesetzt werden.

procedure *mrf86SetChan(chan : tmrf86chan; wr : boolean);*
Die Prozedur stellt den gewünschten RF Kanal im ISM Band ein. Der Kanal chan wird mit der Enumeration tMRF86chan angegeben (mrf86Chan11.. mrf86Chan26). Das boolean wr bestimmt dabei, ob der Kanal sofort eingestellt werden soll. Vor dem ersten Init muss wr false sein. Nach einem Init kann der Kanal zur Laufzeit mit wr = true umgestellt werden, ohne ein Init durchzuführen. Alternativ kann auch die Prozedur mrf86SetFreq benutzt werden.

procedure *mrf86SetFreq(freq : word; wr : boolean);*
Die Prozedur stellt die gewünschte Frequenz im ISM Band ein. Die Frequenz freq wird in MHz angegeben. Gültige Werte sind dabei 2405.. 2484. Das boolean wr bestimmt dabei, ob die Frequenz sofort eingestellt werden soll. Vor dem ersten Init muss wr false sein. Nach einem Init kann die Frequenz zur Laufzeit mit wr = true umgestellt werden, ohne ein Init durchzuführen. Alternativ kann auch die Prozedur mrf86SetChan benutzt werden.

procedure *mrf86SetPower(pwr : tmrf86pwr);*
Die Prozedur stellt die gewünschte Ausgangsleistung (RF power) ein. Der power wird mit der Enumeration tMRF86pwr angegeben (mrf86dBm3P0.. mrf86dBm17). Mrf86dBm3P0 ist max. Power (+3dBm) und mrfdBm17 ist minimal power (-17dBm).

procedure *mrf86SetPANaddr(adr : byte); // main address*
Die Prozedur stellt die lokale PAN Adresse (0..254) ein. Das ist die logische Adresse mit der andere Nodes diesen PAN ansprechen müssen. Die wirksame Adresse setzt sich aus LocalAddr und PANaddr zusammen. Es ist möglich dass mehrere Nodes die gleiche PAN Adresse haben und sich nur durch die LocalAddr unterscheiden. Der Wert 255 darf nicht benutzt werden.

procedure *mrf86SetLocalAddr(adr : byte); // sub address*
Die Prozedur stellt die lokale Adresse (0..254) ein. Das ist die logische Adresse mit der andere Nodes diesen Node ansprechen müssen. Die wirksame Adresse setzt sich aus LocalAddr und PANaddr zusammen. Es ist möglich dass mehrere Nodes die gleiche PAN Adresse haben und sich nur durch die LocalAddr unterscheiden. Der Wert 255 darf nicht benutzt werden. Diese LocalAddr und PANaddr zusammen müssen unique sein, d.h. sie dürfen nur einmal im Netzwerk vorkommen.

procedure *mrf86SetRetryMax(rmax : byte);*
Die Prozedur stellt die max. Retry Anzahl ein (0..15). Tritt beim Datentransfer ein Fehler auf, z.B. das Acknowledge des Empfängers kommt nicht rechtzeitig oder gar nicht, dann wiederholt der Sender das Telegramm bis zu rmax mal. Fehlt das ACK dann immer noch, kommt die Sendefunktion mrf86TxPacket mit einem Fehler zurück.

procedure *mrf86SetSpeed(speed : tmrf86Speed; wr : Boolean);*
Die Prozedur stellt die Datenrate “on air“ ein. Die Rate wird mit der Enumeration tMRF86Speed angegeben (mrf86S250, mrf86S500, mrf86S1000, mrf86S2000). Vor dem ersten Init muss wr false sein. Nach einem Init kann der Speed zur Laufzeit mit wr = true umgestellt werden, ohne ein Init durchzuführen.

procedure *mrf86SetKey(Const K1, K2, K3, K4, K5, K6, K7, K8, K9, K10, K11, K12, K13, K14, K15, K16 : Byte);*
Hiermit wird das SEED für die encrypt und decrypt Funktionen des Chips vorgegeben Das ist ein 128bit Schlüssel. Soll der Key zur Laufzeit geändert werden muss nach dem neuen Keu ein Init durchgeführt werden.

function *mrf86Init : boolean;*
Die Funktion initialisiert den 86RF231 mit den oben vorgegebenen Werten und stellt den Empfang aktiv. Bei einem Fehler wird ein false zurückgegeben.

procedure *mrf86SetPWRdown;*
Die Prozedur schaltet den Controller AT86RF231 in den Power-down Mode. Für einen Neustart genügt dann der Aufruf der Funktion mrf86Init.

Diese Funktionen liefern aktuelle **Zustände, Ereigniszähler** und **Errors** zurück:

function *mrf86GetLostPkts : byte;*
Die Funktion gibt die Anzahl der trotz Retries nicht zustellbaren Pakete zurück. Diese Funktion setzt auch den Counter zurück.

function *mrf86GetRetryCnt : byte;*
Nach einer erfolgreichen Sende Operation kann damit die Anzahl der benötigten Retries für das zuletzt gesendete Packet ausgelesen werden. Je höher dieser Wert ist, desto schlechter war die Verbindung.

function *mrf86GetRandom : byte;*
Diese Funktion liefert ein echtes Random Byte zurück.

Diese Funktionen sind die eigentlichen **Arbeits Funktionen**:

function *mrf86GetRxType : tmrf86pkt;*
Die Funktion gibt den Empfangs Status zurück. tMRF86pkt = (mrf86PKTnone, mrf86PKTdata, mrf86PKTBroadCast) . Sie dient zum Pollen des Empfangs.

function *mrf86RxPacket(destPtr : pointer; var recvd : byte; AESMode : tmrf86AesMode) : tmrf86pkt;*
Die Funktion versucht ein Packet abzuholen. destPtr muss auf eine RAM Datenstruktur mit der Grösse 112bytes zeigen. In recvd steht die tatsächlich empfangene Anzahl der Bytes (max. 112). Mit AESmode wird das Encryption vorgeben. Das decrypten erfolgt automatisch wenn mrf86AesDecrypt eingestellt ist, beachtet aber ob das empfangene Packet tatsächlich encryptet ist oder nicht.

Das Ergebnis ist tMRF86pkt = (mrf86PKTnone, mrf86PKTdata, mrf86PKTBroadCast) .
Bei einem Timeout erfolgt ein mrf86PKTnone. Wurde ein Standard Datenpaket empfangen, so kommt ein mrf86PKTdata zurück und bei einem Broadcast ein mrf86PKTBroadCast.

function *mrf86TxPacket(adr, panID : byte; bc : tmrf86mode; srcPtr : pointer; Cnt : byte; AESMode : tmrf86AesMode) : tmrf86TxState;*
Die Funktion versucht ein Daten Packet oder ein Broadcast Packet zu senden. Der Parameter adr bezeichnet die logische Adresse (0..254) des gewünschten Empfängers (Node). panID ist die übergeordnete PAN Adresse. Beide zusammen ergeben das eigentliche Ziel. bc bestimmt ob Broadcast oder nicht. srcPtr muss auf die Quelle im RAM zeigen. Der Parameter cnt bestimmt die Anzahl der zu sendenden Bytes (max. 112). Das tmrf86mode bestimmt ob das Packet ein Broadcast oder ein Daten Packet ist. Bei einem Broadcast werden die Parameter adr und panID ignoriert. Sowohl Standard als auch Broadcast können encrypted werden.

Das Ergebnis kommt als enumeration tmrf86txState zurück:
mrf86TxSucces, mrf86TxInvalid, mrf86TxAccessFail, mrf86TxNoACK

Der MIRF24port Treiber bietet zwei Arten von Daten Packet Transfers an:

1. **Broadcast**. Dieses Packet hat eine bestimmte Kennung so dass alle erreichbaren MIRF86 Nodes dieses Packet empfangen und auswerten können. In diesem Fall ist das Hardware ACK abgeschaltet und auch der Empfänger (Applikation) sollte nicht darauf antworten. Auch Broadcasts können encryptet sein.

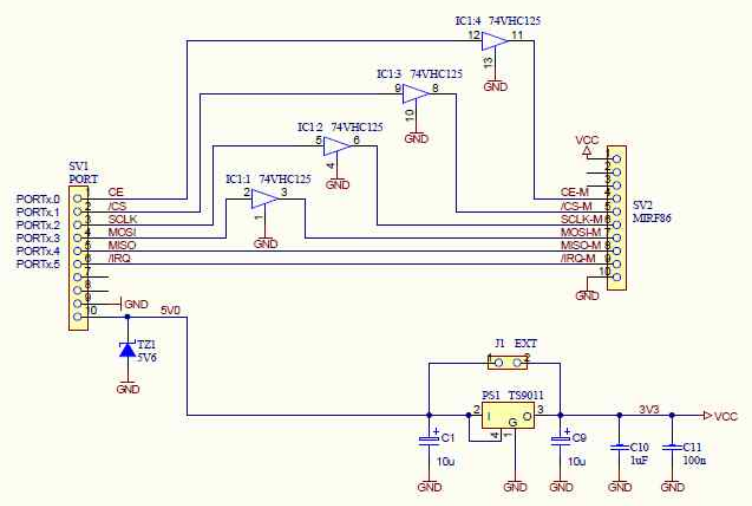
2. **Data**. Dieses Packet geht nur an eine bestimmte Adresse (Node) und wird auch nur von diesem empfangen. Der Empfänger (Node) schickt ein automatisches Hardware ACK an den Sender zurück und quittiert damit den korrekten Empfang. Bei einem CRC Fehler etc. unterbleibt natürlich das ACK und der Sender muss ein Retry durchführen. Ein ACK vom Empfänger bedeutet zu diesem Zeitpunkt keinesfalls dass der Empfänger das Packet schon aus dem 86RF231 ausgelesen hat. Solange dies nicht geschehen ist schickt der Empfänger keine Weiteren ACKs für einkommende Datenpakete so dass der Sender nach Ablauf der eingestellten Retries aus der Sende Funktion mit einem Fehler zurückkehrt.

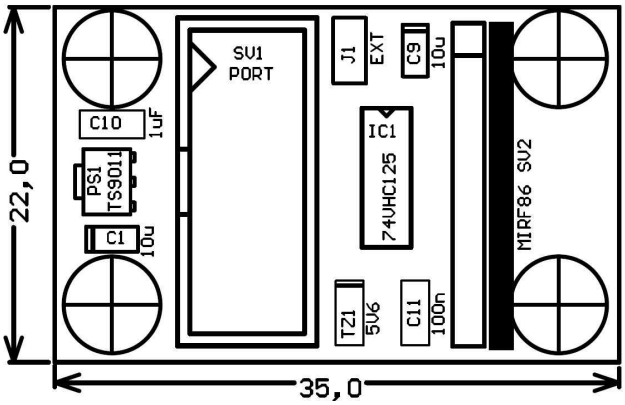
Achtung:
Vor dem Senden eines Broadcasts und danach sollte etwas Pause sein, so dass alle Empfänger bereit sind.

3.22.1 MIRF86 Hardware

3.22.1.1 MIRF86 Adapter für MIRF86 Modul und E-LAB EVA-Boards

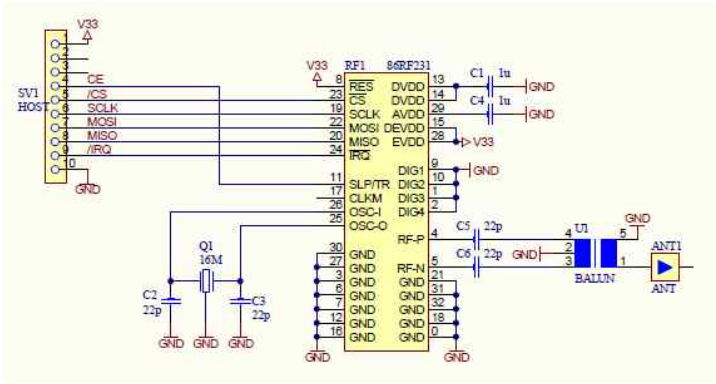
MIRFadaptor für MIRF86

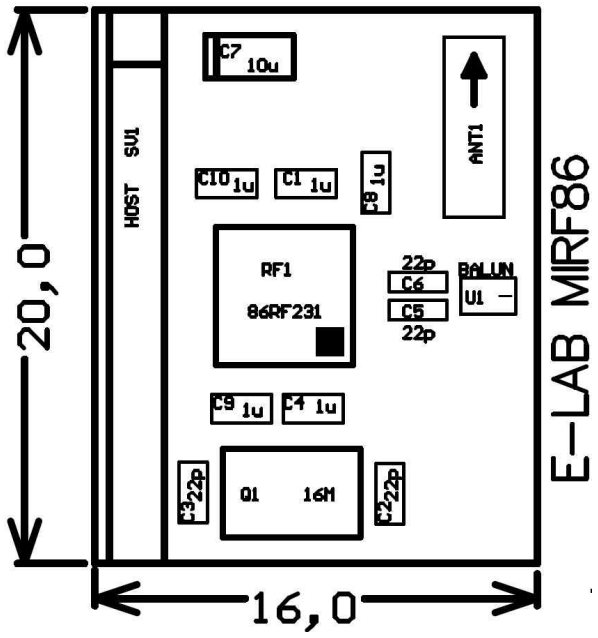




3.22.1.2 MIRF24 Transceiver Module

RF Power = 2mW = 3dBm





3.23 FAT Bootloader XMega

Immer wieder kommt es vor dass in einem laufenden Gerät die Firmware upgedatet werden muss. Dafür gibt es mehrere Möglichkeiten:

- Download der neuen Firmware über eine serielle Schnittstelle
- Download der neuen Firmware über die USB Schnittstelle
- Download der neuen Firmware über eine microSD Karte.
- Weniger geeignet sind SPI, I2C und andere Schnittstellen

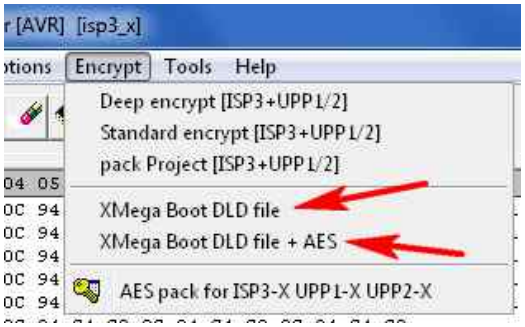
Allen diesen Verfahren gemeinsam ist, dass ein neu-programmieren der Applikation nur aus dem sogenannten Boot Bereich heraus erfolgen kann. Dieser Bereich liegt bei allen AVRs am Flash Ende und ist je nach Typ unterschiedlich gross. Dieser Bereich kann durch das Programm im Boot selbst nicht überschrieben werden und ist deshalb permanent. Dieser Bootloader muss einmal mittels Programmiergerät in den Bootbereich hinein programmiert werden.

Ein Kriterium beim „booten“ (neu-flashen) der Applikation ist dass dazu eine evtl. aktive Applikation in das Boot springen muss um dort diesen Download zu beginnen. Weiterhin muss natürlich sichergestellt werden dass ein Download der neuen Firmware auch erfolgreich war und nicht evtl. abgebrochen worden ist. Ansonsten besteht nie wieder die Möglichkeit aus der „defekten“ Applikation das Boot zu erreichen.

Das beste Verfahren dazu ist, dass ein RESET **immer** ins Boot erfolgt, FUSE **BOOTRST** muss beim Programmieren des Boots aktiv sein. Somit hat das Boot nach einem RESET etc immer zuerst die Kontrolle und kann feststellen ob eine gültige Applikation vorhanden ist. Im AVRco System dient dazu meistens die letzte Speicherstelle im EEprom. Ist diese $FF liegt keine gültige Applikation vor und der Bootloader wartet auf einen externen Download. War der Download erfolgreich so schreibt der Loader eine $00 in das EEprom und macht die Applikation gültig. Anschliessend schaltet der Loader die Vektor Tabelle auf 0000 um und springt nach $0000 in die Applikation.

Soll die Applikation selbst ein Download auslösen, so muss sie in diese EEprom Stelle ein $FF schreiben und einen **HardwareReset** aufrufen. Dieser führt einen internen echten RESET im AVR aus wobei der Einsprung in den Boot Bereich erfolgt. Hier wiederum wird das EEprom geprüft etcetc.

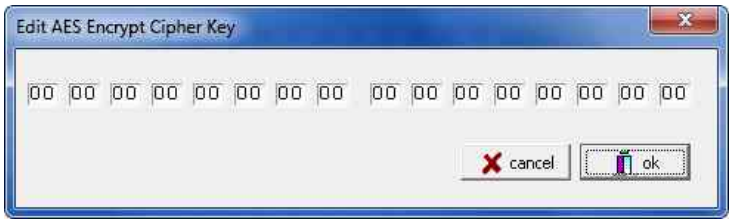
Voraussetzungen:
Der Bootloader muss mit dem Schalter ***{$BootApplication}*** kompiliert werden. Er muss mit einem externen Programmier Gerät in die CPU programmiert werden.
Die Applikation (für den Download) selbst muss keine Bedingungen erfüllen, ausser das die gleiche CPU wie im Boot angegeben werden muss und die Konstante DownLoaderID : word = nnnn; muss definiert werden. Diese ID wird vom Bootloader geprüft.
Die Applikation muss für den FAT Bootloader in eine geeignete Form gebracht werden. Das geschieht mit Hilfe von „AVRprog“. Die Applikation wird hier ganz normal geladen und in ein „DLD“ File gepackt.
Ist eine AES Verschlüsselung der Applikation vorgesehen, so muss der Schlüssel dazu sowohl im Bootloader als auch im AVRprog vorhanden und identisch sein.



Im AVRprog gibt es dazu zwei Menu Punkte:
Download File erstellen ohne AES Verschlüsselung
Download File erstellen mit AES Verschlüsselung.

Nach der Auswahl muss das Ziel bestimmt werden, File lokal oder z.B. auf SD-Karte speichern.

Achtung: es kann nur das Flash, EEprom und UserRow neu geflasht werden. Fuses und LockBits sind nicht möglich!



Wurde die AES Option gewählt, dann muss auch noch der AES Schlüssel angegeben werden.
Mit dem Key wird das File verschlüsselt. Dieser Key muss mit dem Key der im Bootloader angegeben ist, identisch sein!

3.23.1 Bootloader Programm

Das Boot Programm ist sehr kurz und für den User sehr einfach gehalten. Es besteht im wesentlichen aus Zwei Imports und dem Define der SPI Schnittstelle für das microSD Karten Port
dem Define des Boot Modus, plain oder AES
und dem Define des Boot File Namen xxxx.DLD, wobei das DLD weggelassen werden muss.
Ist der AES Modus angegeben muss auch noch der Schlüssel angeben werden, Beispiel:

```
Import FAT_BootX, FlashWrite;
From System Import LongWord;
...
Uses uFAT_BootX;
...
{$IFDEF FAT_bootModeAES}  // mandatory structconst for AES !!
structconst
  DecryptKey   : tAESarr  = ($00, $11, $22, $33, $44, $55, $66, $77,
                             $88, $99, $AA, $BB, $CC, $DD, $EE, $FF);
{$ENDIF}
...
```

Der Treiber exportiert zwei Typen, eine Variable und drei Funktionen:

```
…
type
  tBootErr  = (bootNoErr, bootInvCardType, bootInvBootType, bootReadFail,
               bootClusterNotfound,
               bootFalseID, bootFalseCPU, bootAESerr);
  tAESarr   = Array[0..15] of byte;

VAR BootErr   : tBootErr;

  // Initialize the MMC and check for a valid FAT16 file system
function FATInit : boolean;
{$IfDef FAT_bootModeAES}
procedure AES_Init(Key : Pointer); // Pointer TO flash Const!!!!
{$endif}

procedure FAT_Boot_SetUsrProc(p : procedure);  // Set the CallBack address
// This reads the UpdateFile and flashes it into the application area.
function UpdateFirmware(BootID : word) : boolean;
```

Ein neuer Applikation download aus dem Bootloader erfolgt mit FATInit, falls AES Mode mit AES_Init(@DecryptKey) und dann UpdateFirmware(LoaderID). Der Decrypt key muss mit dem identisch sein, der im AVRprog zum Verschlüsseln verwendet wurde! Die BootID muss identisch sein mit der in der Applikation selbst. FatInit und UpdateFirmware können mit einem false zurückkehren. Dann kann die Enumeration „BootErr“ ausgewertet werden.

Die Funktion UpdateFirmware kehrt niemals ins Boot zurück, wenn der Download erfolgreich war sondern startet automatisch die neue Applikation. Das kann unterbunden werden indem man das obige CallBack besetzt. Die Applikation selbst kann einen Bootloader Start erzwingen, wenn sie das letzte Byte im EEprom auf $FF setzt und dann einen Hardware Reset ausführt. Vorausgesetzt das Boot wurde als BootApplication generiert und die Fuse BootRst ist aktiv.

SD Karte
Es können praktisch beliebige Karten verwendet werden, 2GB..32GB, FAT16 oder FAT32, cluster grösse beliebig. Es können auch beliebige weitere andere Files vorhanden sein. Allerdings muss das DLD File(s) sich in den ersten 2GB befinden.

Resourcen
Der Bootloader passt auch mit AES in ein 4kB grossen Boot Bereich. Allerdings muss dazu der Merlin Optimiser benutzt werden.
Im Verzeichnis **..\E-LAB\AVRco\Demos\XMega_FATboot** befindet sich ein FAT Boot Beispiel und zwei DLD Files für XMega256A3U, einmal mit AES, einmal ohne.

Notizen

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