Compiler Handbuch

E-LAB AVRco

Pascal Multi-Tasking für Single Chips

Version für

AVR

Blaise Pascal Mathematiker 1623-1662

11-Mar-2019

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.

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 Jedem Toaster sein Prozessor!

So oder so ähnlich könnte man diverse Kommentare und Aussagen interpretieren. Und es ist nicht aus der Luft gegriffen. Mikroprozessoren finden immer mehr Verwendung in Anwendungen, wo man diese sich vor kurzem nicht vorstellen konnte, oder aber, sie machen bestimmte Lösungen überhaupt erst möglich.

Teilweise ist das auf stark gefallene Preise der Chips zurückzuführen, oft aber auch auf die Miniaturisierung der Chips. Immer mehr werden daher mechanische, elektromechanische und auch elektronische Lösungen bestimmter Aufgaben durch Prozessoren übernommen. Für den Entwickler stellt sich aber immer öfter das Problem, elegante Lösungen mit möglichst geringem Aufwand bzw. Kosten zu realisieren.

Der Kostenfaktor zwingt den Konstrukteur dazu, in jedem Einzelfall den richtigen Prozessor auszuwählen, um das beste Preis/Leistungsverhältnis zu erzielen. Es ist praktisch nicht mehr möglich, mit einem einzigen Typ (z.B. 80C535 oder 68332) alle Anwendungen zu erschlagen.

Die Halbleiter Industrie bietet seit einiger Zeit Controller an, die von 16polig/8bit (DM3.-) bis >84polig mit 32bit (DM80.-..) reichen. Für den Entwickler stellt sich spätestens jetzt aber das Problem des richtigen Entwicklungswerkzeuges. Je genauer der jeweilige Prozessor spezifiziert wird, desto mehr Entwicklungswerkzeuge werden im Lauf der Zeit benötigt. So kommen, bei einem üblichen Preis zwischen 4000.- und 10.000.- DM pro Tool, einige 10tausend DM zusammen. Das wiederum widerspricht aber genau dem Kostenproblem, dass alles, auch die Entwicklungen, immer billiger werden muss, ja manche Auftraggeber erwarten sogar, dass Entwicklungskosten gar nicht berechnet werden, oder als minimaler Anteil in die Serie eingerechnet werden.

Ab der Prozessorgrösse 8051, 68HC11, Z80 etc. sind viele, im Preis und in der Leistung unterschiedliche Tools zu finden. Leider, bis auf ein paar Ausnahmen , alle für “C”. Andere Hochsprachen sind fast komplett vom Markt verschwunden. Ein Entwickler, der mehrere Hochsprachen kennt, weiss in der Regel die Lesbarkeit, Selbstdokumentation und problemlose Pflege (auch nach Jahren) von Pascal und ähnlichen Sprachen (Modula-2, Oberon) zu schätzen.

Sicher, viele Argumente für oder gegen eine Sprache könnte man in den Bereich der Philosophie oder Religion verweisen. Tatsache ist jedoch, dass bei militärischen und Weltraumprojekten “C” verboten ist und nur der Pascal Abkömmling ADA verwendet werden darf. Sicherheits relevante Aufgaben, z.B. Zugsteuerung (Stellwerke) bei der DB, Flugzeugelektronik bei div. amerikanischen Herstellern werden in Pascal geschrieben und nicht in “C”. Und bestimmt nicht aus Nostalgie Gründen. Soviel zur unserer Entscheidung zugunsten von Pascal und gegen die kryptische Sprache “C”.

Um einen Ausweg aus der o.a. Misere (Kosten, nicht vorhandene bzw. schwache Tools, kein “C”) zu finden, hat E-LAB Computers einen Pascal Compiler für eine Reihe von Prozessor-Familien entwickelt. Ziel war es, ein Tool zu schaffen, das vom Leistungsumfang möglichst komplett ist, jedoch intern keine hohen Kosten verursacht. Deshalb und auch wegen mangelnder Resourcen in den anvisierten Prozessoren, wurde bewusst auf ein paar komplexe Funktionen und System Eigenschaften verzichtet. Z.B. gibt es in der Standard Version keinen Linker und damit kein modulares Programm/Units.

Das Tool ist aber trotzdem oder gerade deshalb problemlos auf andere (kleine) Prozessoren zu portieren. Eine Non-Multi-Task Version gibt es für MicroChip’s PIC. Eine Multi-Task Versionen ist für den AVR von Atmel erhältlich.

Das Tool besteht immer aus der IDE (Editor usw.), dem Compiler und dem Assembler. Zumindest die IDE steht derjenigen der um ein vielfaches teueren Konkurrenz in keinster Weise nach. Einen eigenen Simulator besitzt nur die AVR-Version. Dieser ist ein Bonus/Zugabe zum System.

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 Handbuch Versionen

Abschnitte die mit dem Attribut (*P*) gekennzeichnet sind, sind nur in der ***AVRco Profi Version*** enthalten.

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

2.4 Bekannte Einschränkungen

- Die Funktion **`IntegrateI`** hat noch einen Bug bei negativen Werten.

- Das Überschreiben von vordefinierten Typen, Variablen, Konstanten, Funktionen und Prozeduren ist zur Zeit noch nicht möglich.

- Die Wertigkeit der Operatoren (* AND SHR etc.) wird z.Zt. noch nicht überall eingehalten. Deshalb ist das **Klammern** von Ausdrücken unbedingt notwendig.

- Nicht implementiert sind "**With**" Konstrukte mit Records in Records

- Strings

  Das folgende **String Concat** bringt ein unerwünschtes Resultat:

  str:= str1 + str;

  Dieses Konstrukt ist wegen des limitierten RAM nicht realisierbar.

  Was hingegen funktioniert ist:

  str:= str + str1;

- Arrays

  Die Konstruktion von **Array of Arrays** wird nicht unterstützt:

3 Prinzipielle AVRco Sprach Elemente

3.1 Zeichensatz

Das Vokabular vom AVRco Pascal besteht aus Zeichen, eingeteilt in Buchstaben, Zahlen, und spezielle Symbole:

Buchstaben A bis Z, a bis z und _ (underscore). Um Konflikte mit internen Bezeichnern zu vermeiden, sollte das underscore nicht als erstes Zeichen eines Bezeichners (Symbol) verwendet werden..

Digits 0123456789

Special symbols ***/=<>()[]{}**.,

Es wird nicht zwischen Gross- und Kleinschreibung unterschieden. Bestimmte Operatoren und Delimiter bilden zusammen ein spezielles Symbols:

Assignment operator: **: =**
Vergleichs Operator: **<> <= >=**
Kommentare: **(*** und ***)** können statt **{** und **}** verwendet werden.
Auch C-style Kommentare können verwendet werden.: **//**

Zusätzlich verwendet AVRco Pascal verschiedene Konstrukte, die den Zugriff auf die CPU und ihrer Resourcen ermöglichen. Weiterhin gibt es eine ganze Anzahl von Spracherweiterungen um den Anforderungen eines embedded Systems und dessen Programmierung gerecht zu werden.

3.2 Reservierte Wörter

Reservierte Wörter sind ein integraler Bestandteil vom AVRco Pascal. Sie können nicht redefiniert werden und dürfen deshalb nicht als “user defined identifiers” benutzt werden.

Beispiele:
ABS, AND, ARRAY, ASM, BEGIN, BREAK, CASE, CONST, CONTINUE, DIV, DO, DOWNTO, ELSE, ELSIF, END, ENDFOR, ENDCASE, ENDWHILE, ENDIF, EXIT, FOR, FORWARD, FUNCTION, GOTO, IF, IN, LABEL, MOD, NOT, OF, OR, PROCEDURE, PROGRAM, RECORD, REPEAT, ROR, ROL, SHL, SHR, STRING, THEN, TO, TYPE, UNTIL, VAR, WHILE, WITH, XOR

3.3 Standard Bezeichner

AVRco Pascal definiert eine Anzahl von Standard Identifiers für predefined Types, Constants, Variables, Procedures und Functions. Diese dürfen ebenfalls nicht redefiniert (überschrieben) werden.

Beispiele:
FALSE, TRUE, NIL, CHAR, BOOLEAN, INTEGER, BYTE, INT8, LONGINT, WORD, LONGWORD, FLOAT, POINTER, SIZEOF, ADDR, @, INC, DEC, MOVE, LENGTH, COPY, INTTOSTR, BYTETOSTR, INTTOHEX, STRTOINT, LO, HI, LOWORD, HIWORD, INSERT, DELETE, UPCASE, POS

3.4 Trennzeichen

Sprachelemente müssen zumindest mit einem Delimiter (Trennzeichen) voneinander getrennt werden. Das sind: Leerzeichen, ein Zeilenende oder ein Kommentar.
Kommentare sind an jeder Stelle im Quellcode zulässig.

3.5 Programmzeilen

Die maximale Länge einer Programmzeile darf 250 Zeichen nicht überschreiten.

4 Sprachreferenz

4.1 Typen

4.1.1 Standard scalare Typen

Ein Daten Typ definiert einen Bereich von Werten den eine Variable annehmen kann. Jede Variable in einem Programm muss mit einem Daten Typ verbunden sein. Obwohl Daten Typen im AVRco sehr komplex sein können, sind sie doch alle aus einfachen Typen aufgebaut.

Die grundsätzlichen Typen in Pascal sind sog. scalare Typen. Diese bestehen aus einem linearen und endlichen Werte Bereich.

Ein einfacher Typ kann entweder vom Programmierer definiert werden (ein sog. *declared scalar* Typ, auch Aufzählungstyp oder Enumeration), oder er ist einer von den *standard scalar* Typen*:* integer, int8, word, longint, longword, float, boolean, char, byte, bit oder BitSet.

Neben den Standard Scalar Typen*,* unterstützt Pascal auch *user defined scalar* Typen, auch *declared scalar* Typen (enumeration) genannt. Die Definition einer scalaren Type spezifiziert in aufsteigender Reihenfolge alle ihrer möglichen Werte. Die Werte des neuen Typs werden durch Namen (Identifiers) dargestellt, welche den Konstanten Wert des neuen Typs repräsentieren:

```
type
    Operator     =   (Plus, Minus, Multi, Divide);
```

Im obigen Beispiel ist der Typ Operator eigentlich ein Byte. Die in Klammern nachfolgende Werte sind Aufzählungen beginnend mit dem Wert 0. Der Wert Plus ist damit eine Konstante mit dem Wert 0, Minus ist 1, Multi hat den Wert 2 etc.
Die Verwendung von “defined scalar types” bzw. Enumerationen bzw. Aufzählungstypen ist sehr empfehlenswert, da hiermit die Lesbarkeit eines Programms erheblich gesteigert wird.

```
type
    TDay   = (Mon,Tue,Wed,Thur,Fri,Sat,Sun);
    TMonth = (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
    TMArr  =  array[Jan..Dec] of TDay;

var
    MonArr   =  Tmarr;
    ...

  MonArr[Aug]:= Sun;
  If MonArr[Jan] = Fri then ...
```

4.1.2 Typ Konvertierung

Der Compiler kennt keine automatische Typ Konvertierung. Eine Zuweisung wie z.B. *byte:= word* führt zu einem **Type Missmatch**. Es ist jedoch möglich, praktisch jeden Typ in einen anderen umzusetzen, indem man das sog. Typecasting anwendet. In vielen Fällen wird dabei kein zusätzlicher Maschinen-Code erzeugt (*char:= char(byte)*).
AVRco Pascal kennt, wie die meisten Pascal Compiler, das “Type Casting”. Dazu wird einfach ein Typ-Identifier als "Funktions-Name" benutzt. Der Parameter dieser "Funktion" ist eine Variable vom Typ A die der Compiler als Variable vom Typ B sehen soll (als den Typ, der als "Funktionsaufruf" benutzt wird).

```
var  ch : char;
     b1 : byte;
     w1 : word;
     i1   : integer;
     p1 : pointer

ch:= char(b1);
b1:= byte(w1);
i1:= integer(w1);
p1:= pointer(w1);

type
   TRecordA   = record
                  a,b,c : word;
                end;
   TRecordB   = record
                  X : longint;
                  Y : word;
                end;

var
   A : TRecordA;
   B : TRecordB;
   C : char;
   D : byte;

begin
   A:= RecordA (B);
   RecordA (B):= A;
   D:= byte (C);
   C:= char (D);
end.
```

Tip:
In der Pascal Welt ist es üblich, dass Typ Namen mit einem grossen “T” beginnen. Das ist kein muss, erhöht allerdings die Lesbarkeit des Programms und hilft Fehler zu vermeiden.

4.1.3 Variable Overlay

Variable können direkt auf andere Variablen platziert werden (Overlay).
Dabei gibt es das Problem, dass die zweite Variable grösser sein kann als die referenzierte Variable.
Da aber nur die Referenz Variable Speicher allokiert, kann es zum Überschreiben von nach-platzierten Variablen zur Laufzeit kommen.

```
var abc       : byte;
    xyz       : byte;
    ovr1[@abc]: byte;      // it's ok
    ovr2[@abc]: word;      // problem
```

Im Beispiel würde ein Schreibzugriff auf die Variable "ovr2" nicht nur die Variable "abc" überschreiben, sondern auch (ungewünscht) die Variable "xyz".
Der Compiler prüft bei Overlays, ob die zu platzierende Variable in den vorgesehenen Speicherplatz der referenzierten Variablen passt.

Um aber untenstehende Konstruktion trotzdem zu ermöglichen, gibt es den Compiler Switch *{$NOOVRCHECK}*. Damit kann diese Prüfung gezielt für die nachfolgende Deklaration abgeschaltet werden.

```
var
   v1      : byte;
   v2      : byte;
   v3      : byte;
   v4      : byte;

{$NOOVRCHECK}
v5[@v1]: longword;
```

v5 würde durch den Check abgewiesen werden, da ein LongWord 4 Bytes benötigt aber die Variable v1 nur ein Byte bereitstellt. Durch die Abschaltung des Checks belegt v5 den Speicherplatz von v1..v4 und schliesst damit alle vier Bytes mit ein, was legal sein kann.

Borland Pascal hat dazu eine andere Implementation. Auch diese wird unterstützt:

```
var
 abc       : byte;
 xyz       : word;
 ovr1      : byte absolute abc;         // var abc overlayed
{$NOOVRCHECK}
 ovr2      : word absolute xyz;         // var xyz overlayed
```

"absolute" darf nur mit Variablen im RAM und EEprom Bereich verwendet werden. Das „absolute“ darf nicht auf lokale Variablen in Funktionen angewendet werden.

4.1.4 BOOLEAN

Benötigter Speicher: 8bit, Wert: true..false. Kann einen der logischen Wahrheitswerte *True* oder *False* annehmen. Diese sind derart definiert, dass *False = 0;* True > 0 (normalerweise $FF).
Eine boolsche Variable benötigt ein Byte im Speicher.

true = $FF false = $00

```
var    flag : boolean;
```

Bemerkung:
Ein Boolean benötigt als Speicherplatz 1 Byte. Um Speicherplatz zu sparen, ist es u.U. sinnvoll eine Byte-Variable zu verwenden und diese in einzelne Bits mittels der BIT-Definition aufzuteilen. Damit reduziert sich der Speicherbedarf für 8 Boolean auf 1 Byte.

```
var  flag             : byte;
     flag0[@flag, 0]  : bit;
     flag1[@flag, 1]  : bit;
            ...

if Bit (Flag0) then ...
Incl(Flag1);
SetBit(flag0, flag1);      {bit kopieren}
```

4.1.5 BIT

Benötigter Speicher: 1bit, true..false, 0..1

Um die bei embedded Anwendungen so wichtigen Bits verarbeiten zu können, wurde die ***Variable Bit*** eingeführt. Dieser Typ kann genau wie eine bool'sche Variable benutzt werden, jedoch zusätzlich mit *INCL(bit), EXCL(bit), TOGGLE(bit), SetBit* und "*if BIT(bit) then* .." . Die Deklaration einer Bit-Variablen besteht **immer** aus zwei Stufen: zuerst muss die Speicherstelle bzw. Variable, in der das Bit liegt, deklariert werden. Hier sind nur globale Variablen vom Typ Byte oder Word zulässig, in der ***REV4 auch LongWord***. Bits in Arrays oder Records sind nicht möglich. Zuerst muss die Variable, in der das/die Bits liegen sollen, definiert werden.

```
var  Leds[$05] : byte;   oder
     bits16    : word;
```

Der **Typ der Speicherstelle** (byte, word, longword) bestimmt dann auch, ob 8, 16 oder 32 Bits zur Verfügung stehen. Nach der Deklaration einer allgemeinen Variablen erfolgt dann die eigentliche BIT-Deklaration. Der erste Parameter bezeichnet dabei eine Speicherstelle, der zweite Parameter das entsprechende Bit dieser Speicherstelle. Es können auch symbolische Parameter eingesetzt werden.

```
const LedBit2                : byte  = 3;
var   port6                  : byte;
      Led2[@port6, Led2Bit] : bit;

  if BIT(Led2) then ...
    Toggle(Led2);

SetBit(Led2, not Led2);
```

Bits können auch innerhalb des Programms dynamisch generiert werden.

```
Toggle(Leds, 3);      {Bit3  innerhalb 8  Bits}
Incl(bits16, 12);     {Bit12 innerhalb 16 Bits}
```

4.1.6 BITSET

8bit, 16bit oder 32bit, abhängig von der zugrunde liegenden Enumeration. 32bits nur in REV4.

Grundsätzliches:
Bei einem Aufzählungstyp kann jedem der möglichen Werte von 0..255 ein Namen zugewiesen werden. Im Gegensatz dazu wird bei einem BitSet jedem Bit einen Namen zugewiesen. Damit kann ein Byte bis zu 8 "Namen" = Bits enthalten, ein Word bis zu 16 und ein longword bis zu 32. Hiermit kann jedes einzelne Bit direkt angesprochen und manipuliert werden. Auch der Zugriff auf Bit-Gruppen ist damit möglich.

Definition:
Bevor eine BitSet Type deklariert werden kann, muss ein Aufzählungs- bzw. Enumerations Typ gebildet werden, der die Bitnamen enthält. Die Anzahl der Bitnamen bestimmt, ob das BitSet in einem Byte, Word oder Longword Platz findet. Mehr als 32 Namen bzw. Bits sind nicht möglich.

Deklaration:

```
type
  TbitNames = (one, two, three, four, five, six);       // enum
  TBitSet   = BitSet of TbitNames;    // build a bitset type

var
 BitSet1  : TBitSet;         // build a bitset var
 BitSet2  : TBitSet;         // build a bitset var
 Bb       : byte;
```

Arbeiten mit BitSets:

```
BitSet1:= BitSet2;         // copy a set into another
bb:= byte (BitSet1);       // type convert
BitSet2:= TbitSet (bb);    // type convert
```

Um ein Bitset komplett zu füllen, müssen normalerweise alle Bits angegeben werden:

```
BitSet1:= [one, two, three, four, five, six];
```

Es ist jedoch auch möglich das mit dem Enum-Namen zu erledigen:

```
BitSet1:= [tEnum];
```

BitSets können mit **Operatoren** miteinander verrechnet werden. Diese sind **+ - * /**

Addition:

```
BitSet1:= [one, three];
BitSet2:= [two, four, six];
BitSet1:= BitSet1 + BitSet2;     // unification – logical or
//BitSet1 enthält jetzt one, two, three, four und six.
```

Subtraktion:

```
BitSet1:= [one, two, three];
BitSet2:= [two, four, three];
BitSet1:= BitSet2 - BitSet1;     // difference – logical and not
//BitSet1 enthält jetzt four.
```

Multiplikation:

```
BitSet1:= [one, two, three];
BitSet2:= [two, four, five];
BitSet1:= BitSet2 * BitSet1;     // logical and
//BitSet1 enthält jetzt two.
```

Division:

```
BitSet1:= [one, two, three];
BitSet2:= [two, four, five];
BitSet1:= BitSet2 / BitSet1;     // logical xor
//BitSet1 enthält jetzt one, three, four, five.
```

BitSets können durch **Vergleichen** miteinander verrechnet werden. **= <> <= >= IN**
Dabei gilt die binäre Entsprechung der Bit Muster.

```
if BitSet1 = [one, two, three] then ...
if BitSet2 <= [two, four, five] then ...
if BitSet2 in BitSet1 then ...
if [two, four, five] in BitSet1 then ...
```

4.1.7 BYTE

8bit, 0..255
Bytes sind ganze Zahlen. Ihr Bereich auf Werte zw. 0 bis 255 beschränkt.
Bytes benötigen ein Byte im Speicher.

Grundsätzlich können alle Variablen Deklarationen auf feste Adressen weisen.

Beispiel für feste Adresse:

```
var  b[$10]   : byte;         { memory adr. 10hex }
```

Beispiel für freie Adressvergabe durch den Compiler:

```
var   x   : byte;
```

Es ist auch möglich, eine zuvor deklarierte Variable als Referenz anzugeben. Damit ist es z.B. möglich, die zwei Bytes einer Word Variable direkt anzusprechen:

```
var  w[$12]      : word;
     b1[@w]      : byte;       {b1 liegt auf Adr. 12hex – low byte}
     b2[@w + 1]  : byte;       {b2 liegt auf Adr. 13hex – high byte}
```

4.1.8 CHAR

8bit, chr(0..255)

Ein Char Wert ist ein Zeichen des ASCII Zeichensatz.
Die Zeichen sind gemäß ihrem ASCII Wert geordnet. Z.B. ist 'A' < 'B'. Die ordinalen (ASCII) Werte der Zeichen überdecken den Bereich von 0 bis 255. Eine Char Variable benötigt ein Byte im Speicher.

Beispiel Variable vom Typ char:

```
var     c   : char;
```

Beispiel Konstante vom Typ char:

```
const cd     = 'D';
      Bell   = ^G;           {Control G}
      LF     = #10;          {Line Feed}
```

Zeichensatz: siehe weiter unten bei den Strings

4.1.9 STRING

0..255 bytes, Variable oder Konstante.

AVRco Pascal kennt String Typen um Zeichenketten zu verarbeiten, z.B. Folgen von Buchstaben. String Typen sind strukturierte Typen und in vielerlei Hinsicht den Array Typen ähnlich. In Pascal ist ein String absolut gleich einem *array of char* und ein solches Array kann als String behandelt werden. AVRco Pascal benutzt das ERSTE Zeichen eines Strings als Längenkennung, worauf dann der eigentliche String folgt. Dies ist kompatibel zu den meisten Pascal String Implementationen, unterscheidet sich aber von C, wo ein Null-Byte als String-Begrenzung benutzt wird.

Kurz gesagt resultiert daraus eine bessere Performance, da die Länge eines Strings schon bekannt ist ohne den gesamten String zu scannen. Ein Pascal String ist auch ein geeignetes Speichermedium für andere Daten als Buchstaben, da es keine Einschränkungen bezüglich den Zeichen gibt, die in den String platziert werden können. Andererseits können C Strings eine "unendliche" Länge haben, wohingegen Pascal Strings auf eine Länge von 255 Zeichen beschränkt sind.

```
type   st10  = string[10];         {stringlength = 10}

structconst  {constant in Rom, at startup copied into Ram}
  str   : st10    = 'abcde';

const        {constant in Rom}
  st    = '1234' + 'R' + #7 + ^L;

var
  st1   : string[8];

 st1:= st;
 ch:= st[2];
```

Bei Pascal Strings befindet sich an erster Stelle im String (str[0]) das sog. Längen Byte. Dieses gibt die tatsächlich belegte Länge des Strings an. Man kann z.B. die Länge dynamisch durch manipulieren der Stelle 0 in den Grenzen der Deklaration verändern (str[0]:= #5;). Der bessere Weg ist allerdings die System Funktion **SetLength**(st : string; len : byte); zu benutzen.

Dieses Längen byte (= char) kann auch zur Bestimmung der aktuellen Länge gelesen werden. Besser und schneller ist hier jedoch die Funktion **Length**(str). String-Manipulationen nur bei Var und StructConst!

Die Definition eines String Type muß die maximale Anzahl der Zeichen spezifizieren die enthalten sein können, d.h. die größte Länge die bei Strings dieses Types vorkommen können.
Die Definition besteht aus dem reservierten Wort **string**, gefolgt von der maximalen Länge in eckigen Klammern. Die Länge ist eine Byte Konstante im Bereich von 0 bis 255. Beachten Sie, daß Strings keine Default Länge haben. Die Länge muß immer spezifiziert werden.

Der Speicherbedarf von String Variablen ist immer die maximale Länge plus ein Byte welches die augenblickliche Länge enthält. Die einzelnen Zeichen eines Strings sind von 1 bis zur Länge des Strings indiziert.

Strings können durch *string expressions* manipuliert werden. String expressions bestehen aus Konstanten, String Variablen, Funktions Bezeichnern und Operatoren.
Das Plus-Zeichen wird benutzt um Strings aneinander anzuhängen. Die *Concat* Funktion, die in einigen Pascal Implementationen vorhanden ist bewirkt das Gleiche, ist jedoch im AVRco nicht implementiert, die der +-Operatator meist bequemer ist.

Das Ergebnis eines String Ausdruck kann nicht länger als 255 Zeichen sein. Ebenso wird Zuweisungen höchstens die maximale Länge des Zielstrings übertragen.

```
'E-LAB ' + 'Pascal ' + 'is '+ 'fun…
' '123' + ' . + '456'
 'A ' + 'B' + ' C ' + 'D '
```

Die relationalen Operatoren = und < > haben eine geringere Priorität als der Concat Operator.
Wenn diese auf einen String angewandt werden ist das Ergebnis ein Boolean (*True* oder *False).*
Beim Vergleich von Strings werden die einzelnen Zeichen von links nach rechts gemäß ihrem ASCII Wert verglichen.

Wenn die Strings verschiedene Länge haben, aber einschliesslich dem letzten Zeichen des kürzeren String gleich sind, dann wird der kürzere String als kleiner betrachtet.

Strings sind nur gleich wenn sowohl ihre Länge als auch ihr Inhalt identisch sind.

```
'A' = 'A'                              // ist true
'A' = 'a'                              // ist false
'2' <> ' 12'                           // ist true
'PASCAL' = 'PASCAL'                    // ist true
'PASCAL' = 'pascal'                    // ist false
'Pascal Compiler' <> 'Pascal compiler' // ist true
```

Der Assignment Operator wird benutzt um einer String Variable den Wert eines String Ausdrucks zuzuweisen.

```
Age:= 'twenty'
Line:= 'Many happy returns on your ' + Age + 'birthday’
```

Wenn die maximale Länge eines Strings überschritten wird (indem zu viele Zeichen der Variable zugewiesen werden) werden die überzähligen Zeichen abgeschnitten. Wurde beispielsweise die obige Variable *Age* als Type **string[5]** deklariert, enthält sie nach der Zuweisung nur die linken fünf Zeichen: 'twent'.

Einschränkung
Ist der Zielstring auch in der Quelle enthalten, muss dieser an erster Stelle stehen!

Das folgende String Concat bringt ein unerwünschtes Resultat:

```
str:= str1 + str;
```

Das funktioniert:

```
str:= str + str1;
```

Zeichensätze

AVRco benutzt für Chars und Strings den ANSII Zeichensatz. Oft wird aber von externen Geräten der OEM Zeichensatz erwartet. Die beiden Zeichensätze unterscheiden sich im wesentlichen nur in den onderzeichen und Umlauten.

Im System Import kann die Option "OEMcharSet" angegeben werden. Damit werden für Sonderzeichen in Chars und Strings der OEM Zeichensatz benutzt:

```
From System Import ..., OEMcharSet;
```

4.1.10 Property

Ein Property kann als eine „fast“ normale Variable betrachtet werden. Das Property kann gelesen und geschrieben werden. Der Unterschied zu einer „normalen“ Variablen besteht darin, dass ein Lesevorgang nicht auf eine Speicherstelle geht sondern zu einer Funktion (Getter) die den Wert zurückliefert. Beim Schreiben auf ein Property muss das eine Prozedur (Setter) übernehmen, die mit dem übergeben Datum entspr. Verfährt.

Ein Property hat also einen Namen, einen Typ zum Lesen/Schreiben und die Getter und Setter Funktionen.

```
Property
  MyWord    : word Read GetMyWord Write SetMyWord;      // read-write property
  MyWordWr  : word Write SetMyWordWr;                    // write only property
  MyByteRd  : byte Read GetMyByteRd;                     // read only property
```

Dazu müssen die Setter/Getter Funktionen bereitgestellt werden und zwar vor der ersten Benutzung:

```
procedure SetMyWord(w : word); // setter
begin
  ww:= w;
end;

function GetMyWord : word;     // getter
begin
  return($5678);
end;
```

Properties können auch im Definitions Teil einer Unit stehen und gelten damit auch global.
Die Setter/Getter Funktionen müssen vor der ersten Benutzung eines Properties implementiert sein!

Ein **Beispiel Programm** „Properties“ befindet sich im Demos Directory im Folder „XMega_Properties“.

4.1.11 ARRAY

Ein Array ist ein strukturierter Typ, der aus einer festen Anzahl von Komponenten besteht die alle vom gleichen Typ sind, dem sogenannten *Komponenten* oder *Basis Typ.* Auf jede Komponente kann einzeln durch Indices zugegriffen werden. Indices sind skalare Typen, in eckige Klammern eingeschlossen und ihnen geht der *Array Namen* voraus. Ihr Typ wird auch *Index Typ* genannt.

1..4 Dimensionen. Eine Dimension kann bis zu 61440 ($F000) Mitglieder haben. Totale Grösse beschränkt auf ca. 60kBytes. Typen: Bytes, Int8, Chars, Booleans, Words, Integers, LongWords, Floats, Fix64, Pointers, Procedures.

Die Definition eines Arrays besteht aus dem reservierten Wort **array**, gefolgt vom Index Typ in eckigen Klammern, dem reservierten Wort **of** und dem Komponenten Typ.

```
type Day = (Mon,Tue,Wed,Thu,Fri,Sat,Sun);

var  WorkHour   : array[1..8] of Integer;
     Week       : array [ 1. . 7] of Day;

type Players =  (Playerl, Player2, Player3, Player4);
     Hand     =  (One, Two, Pair, TwoPair, Three, Straight, Flush,
                  FullHouse,Four, 5traightFlush);
     Bid      =  array[Players] of byte;

var  Player   : array[Players] of Hand;
     Pot      : Bid;

Var  ArrTest : array[3] of byte; //identisch mit Array[0..3] of byte
```

Der Zugriff auf eine Array Komponente erfolgt durch den Array Namen, gefolgt durch einen in eckigen Klammern eingeschlossenen Index:

```
Player[Player3]:= FullHouse;
Pot[Player3]:= 100;
Player[Player4]:= Flush;
Pot[Player4]:= 50;
```

Zuweisungen sind zwischen Variablen identischen Typs erlaubt. Ganze Array können mit einer einzigen Anweisung kopiert werden.

Die Definition einer Array Konstante besteht aus einem Konstanten Bezeichner, einem Doppelpunkt, dem Typ Bezeichner eines zuvor definierten Array Types, einem Gleichheitszeichen und dem Wert. Der Wert wird als durch Kommata getrennte Konstante angegeben und in Klammern gesetzt.

```
Type  Status : array[0..2] of string[7] ;
const Stat   : Status = ('active', 'passive', 'waiting');
```

Im Beispiel wird die Array Konstante *Stat* definiert, die dazu benutzt werden kann die Werte des skalaren Typs *Status* in die entsprechenden Stringdarstellungen umzuwandeln.

Die folgenden Vergleiche würden true liefern:

```
Stat[0] = 'active'
Stat[1] = 'passive'
Stat[2] = 'waiting'
```

Mehrdimensionale Array Konstanten werden definiert indem die Konstanten jeder Dimension in ein zusätzliches Klammernpaar gesetzt und durch Kommata getrennt werden. Die innersten Konstanten entsprechen den Dimensionen am weitesten rechts.

```
type Cube = array[0..1,0..1,0..1] of integer;

const Maze : Cube = ( (  (0, 1) , (2, 3)  )  , ( (4, 5) , (6, 7)  ) ) ;

type Tars = array[0..1, 3..4] of byte;

structconst                    {constant in Rom, at startup copied into Ram}
  ars : Tars = ( (0, 1) , ($FE, $FF) ) ;

const                          {constant in Rom}
  arc : array[0..4] of word = (1, 200, 523, 1200, 9999);

var  ar1   : array[5..12] of char;
     ar2   : array [6 .. 9, 56 .. 67] of word;
     ar3   : Tars;

x:= ar[1];
ar1[8] := #7;
ars [1, 3] := #7;
ar2 [8, 60] := arc [3] ;
```

Array Manipulationen sind nur für Var und StructConst erlaubt!

Spezial Konstruktion
Array Konstante können auch aus Dateien eingelesen werden. Für den Inhalt der Datei ist der Programmierer selbst verantwortlich. Die Dateilänge spielt dabei keine Rolle. Ist die Datei zu kurz, wird mit Nullen aufgefüllt, ist sie zu lang, wird beim erreichen der Array-Grenze abgebrochen.

```
Const  Arr1 : array[0..31] of word = 'DateiName.ext';
```

4.1.12 TABLE

1 Dimension. Bis zu 255 Mitglieder.
Typen: Bytes, Int8, Chars, Booleans, Words, Integers, LongWords, Floats, Fix64, Pointers, Procedures.
TABLE ist ein spezialisiertes Array, in dem Look-Up Tabellen untergebracht werden. Die Table Länge ist auf 2er Potenzen beschränkt, um einen sehr schnellen Zugriff zu gewährleisten:
0..3, 0..7, 0..15, 0..31, ... 0..255.
Der Zugriff macht einen autom. Umbruch, wird z.B. auf ein Table von 0..7 mit dem Index 8 zugegriffen, erfolgt der Zugriff auf Index 0. Tables müssen, um den schnellen Zugriff sicherzustellen, global definiert werden. Normale Konstante im Rom sind beim AVR sind nicht zulässig . Strukturierte Konstante sind im Rom und EEprom möglich. Die Zugriffe müssen mit ***GetTable*** und ***SetTable*** durchgeführt werden.

```
structconst                    {constant in Rom, at startup copied into Ram}
  Table1 : Table[0..3]  = (0, $45, $A5, $FF);

var  tb1   : Table[0..15] of char;
     tb2   : Table[0..127] of word;

 x:= GetTable (tb1, 1);
 SetTable (tb1, x, $35);
 x:= GetTable (Table1, z);
```

4.1.13 RECORD

Ein Record ist eine Struktur, die eine bestimmte Anzahl von Komponente, sog. Felder, enthält.
Felder können aus unterschiedlichen Typen bestehen. Jedes Feld muss einen Namen und einen Typ besitzen. Der Namen dient zum zugreifen auf ein bestimmtes Feld in einem Record.

Record Definition

Die Definition einer Record Type besteht aus dem reservierten Wort ***record*** gefolgt von einer Liste mit Feldern. Den Abschluss bildet das reservierte Wort ***end***. Die Feld Liste besteht aus einer Reihe von Record Sectionen, abgeschlossen mit einem Strichpunkt. Jede Sektion besteht aus einem oder mehreren Namen, durch Kommas getrennt, einem folgendem Doppelpunkt und einer anschliessenden Typ Bezeichnung

```
type  tMonths = (Jan,Feb,Mar,Apr,May,Jun,July,Aug,Sep,Oct,Nov,Dec);
      Date  =  record
                  Day     :  byte;
                  Month   :  tMonths
                  Year    :  word;
               end;
var
      Birth       : Date;
      WorkDay     : array[1..5] of date;
```

Day, *Month* und *Year* sind field identifiers (Namen). Jeder Namen darf nur einmal in diesem Record vorkommen. Ein Feld eines Records wird durch den Variablen (Record) Namen und dem Feld Namen angesprochen. Beide Namen müssen durch einen Punkt getrennt werden.

```
Birth.Month := Jun;
Birth.Year := 1950;
WorkDay[Current] := WorkDay[Current-1];
```

Zuweisungen (komplettes Kopieren) wischen zwei Records, die vom gleichen Typ sind ist möglich. Da die Bestandteile innerhalb eines Records aus jedem möglichen Typ bestehen können, sind auch Konstruktionen wie die folgende möglich (record of records of records):

```
type  Tmonths = (Jan,Feb,Mar,Apr,May,Jun,July,Aug,Sep,Oct,Nov,Dec);
      Tname  =  record
                   FamilyName      : string[32],
                   ChristianNames : array[1..3] of string[16];
                end;

     TRate  =   record
                   NormalRate,
                   OverTime,
                   NightTime,
                   Weekend       : Integer;
                end;

     TDate  =   record
                   Day      : byte;
                   Month    : TMonths;
                   Year     : word;
                end;

     TPerson =  record
                    ID      : TName;
                   Time     : TDate;
                end;
```

```
Twages  =  record
               Individual   : TPerson;
               Cost         : TRate;
           end;

Var  Salary, Fee:  TWages;
```

Mit obigen Definitionen sind folgende Statements möglich:

```
Salary:= Fee;
Salary.Cost.Overtime := 950;
Salary.Individual.Time := Fee.Individual.Time;
Salary.Individual.ID.FamilyName :=  Smith;
```

Die Definition einer Record Konstanten besteht aus dem Identifier (Name) der Konstanten gefolgt von einem Doppelpunkt und dem Typ-Bezeichner. Dieser Bezeichner bezieht sich auf einen zuvor definierten Record Typ. Hierauf folgt ein Gleichheitszeichen und die Konstanten Liste in Klammern. Die einzelnen Konstanten müssen mit dem gleichen Namen versehen sein wie in der Typ-Deklaration.

```
type   Point = record
                  X,Y, Z : integer;
               end;

const  APoint : Point = (X : 0; Y : 2;  Z : 4);
```

Die Konstanten müssen in der gleichen Reihenfolge spezifiziert werden, wie in der Typ-Deklaration angegeben, auch die Namen müssen identisch sein.

Spezial Konstruktion
Record Konstante können auch aus Dateien eingelesen werden. Für den Inhalt der Datei ist der Programmierer selbst verantwortlich. Die Dateilänge spielt dabei keine Rolle. Ist die Datei zu kurz, wird mit Nullen aufgefüllt, ist sie zu lang, wird beim erreichen der Record-Grenze abgebrochen.

```
Const          Rec1 : TWages = 'FileName.ext';
```

4.1.13.1 WITH Statement zum Zugriff auf Records

Wenn Records wie z.B. oben beschrieben verwendet werden, kann es zu etwas langen Statements kommen. Man kann dies einschränken durch die Verwendung des **with** Statements. Dieses *bricht* den Record auf so dass man die Felder direkt ansprechen kann wie normale Variable.

Ein **with** Statement besteht aus dem reservierten Wort **with** gefolgt von einer Record Variablen und dem reservierten Wort **do** und zumindest einem Statement. Das ganze wird mit **EndWith** abgeschlossen.

Innerhalb des *with* Statements wird ein Feld nur noch mit seinem Namen angesprochen, der Record Namen kann entfallen.

```
with Salary do
  Individual := NewEmployee;
  Cost := StandardRates;
endwith;
```

4.1.14 PROCEDURE

16bit, Parameter, word, Adresse.
“Procedure” bezeichnet eine Variable, die die Adresse einer Prozedur aufnehmen kann.

```
var   proc  : procedure;

procedure indirtest;
begin
  ...
end;

begin                    {Main Program}
  ...
  Proc:= @indirtest;     {Variable besetzen mit Adresse von indirtest}
  Proc;                  {indirtest aufrufen}
  ...
end.
```

4.1.15 WORD

16bit, 2 Bytes, 0..65535

Words sind ganze Zahlen. Sie sind auf einen Bereich von 0 bis 65535 beschränkt.
Words benötigen zwei Byte Speicher: low Byte (niedrige Adresse), high Byte (hohe Adresse)

```
var   w       : word;
      w1[24]  : word;          {word an der Adresse 24}
```

4.1.16 INT8 oder ShortInt

8bit, 1 Byte, -128 .. + 127

Short Integer (Int8) sind ganze Zahlen. Sie sind auf einen Bereich von - 128 bis +127 beschränkt.
Short Integer benötigen ein Byte Speicher.

4.1.17 INTEGER

16bit, 2 Bytes, -32768..32767

Integer sind ganze Zahlen. Sie sind auf einen Bereich von - 32768 bis 32767 beschränkt.
Integer benötigen zwei Byte Speicher: low Byte (niedrige Adresse), high Byte (hohe Adresse)

```
var   i       : integer;
      w[@i] : word;            {word mit der gleichen Adresse wie i }
```

4.1.18 POINTER

16bit, 2 Bytes

Die bislang vorgestellten Variablen müssen *statisch* sein. D.h. ihre Form und Größe ist vorbestimmt und sie existieren im gesamten Block in dem sie deklariert wurden. Programme benötigen jedoch häufig Datenstrukturen die in Form und Größe während der Ausführung variieren. *Dynamische* Variablen erfüllen diesen Zweck, da sie bei Bedarf generiert und nach Benutzung wieder verworfen werden können.
Solche dynamischen Variablen werden nicht wie statische Variablen explizit deklariert und sie können auch nicht direkt durch Bezeichner referenziert werden. Vielmehr wird ein spezieller Variablentyp, der die Speicheradresse der eigentlichen Variablen enthält als Zeiger benutzt. Dieser spezielle Variablentyp heißt *Pointer Variable.*

Manchmal ist es sinnvoll dass man Variable nicht mit ihrem Namen ansprechen muss, sondern dass die Möglichkeit besteht, mit der Adresse einer Variablen zu arbeiten.

Ein Pointer ohne Qualifizierung (Typisierung) zeigt immer auf ein byte, ist also trotzdem "typisiert". Eine Ausnahme bildet die Typkonvertierung mittels p:= pointer(word1);. Dieser generierte Pointer ist immer untypisiert und wird erst durch eine Zuweisung in eine andere Pointer-Variable qualifiziert. Der bessere Weg ist jedoch grundsätzlich mit typisierten Pointern zu arbeiten. Dazu wird ein privater Typ generiert mit

```
type   tpb : pointer to Byte;

var    pb : tpb;

pb:= tpb (irgendeinPointer);
```

Einem Pointer muss mit dem Adress-Operator @ oder der Funktion ***Addr (x)*** ein Wert zugewiesen werden. Ein Pointer wird mit der Zuweisung ***p:= nil*** ungültig gemacht. Der Pointer selbst kann wie jede andere Variable manipuliert werden. Dabei ist allerdings darauf zu achten, dass ein manipulierter Pointer auch ins Nirwana zeigen kann!!

Eine Prüfung von Pointern auf Gültigkeit (Zieladresse) findet nicht statt! Der Programmierer sollte sehr sorgfältig mit Pointern umgehen. Hat ein Pointer seine Gültigkeit u.U. verloren, so sollte ihm NIL zugewiesen werden, um beim nächsten Gebrauch die Gültigkeit abprüfen zu können. Das muss aber alles per Software geschehen, der Compiler kann hier keine Unterstützung bieten. **'C' lässt grüssen!**

```
type  tpw : pointer to word;

var   p    : pointer to word;   {zeigt immer auf ein word}
      pb   : pointer;           {zeigt auf ein byte}
      pw   : tpw;               {zeigt auf ein word}
      b1   : byte;
      b2   : byte;
      w    : word;

Function DecWord (p : tpw) : word;
begin
  inc (p^);
  return(p^);
end;
```

```
Procedure IncByte (b : pointer to byte);
begin
  inc (b^);
  p:= pointer (b1);
end;

{Main}
...
pb:= @b1;
incByte (pb);
pb:= @b2;
incByte (pb);
p:= @w;
p^:= 1234;
p:= nil;
```

Manchmal ist es notwendig einen Pointer schon definiert zu haben, bevor das Objekt/Typ auf das der Pointer zeigt, schon existiert (z.B. Linked Lists). Hierzu wird eine Pointer Deklaration mit dem Attribut “Forward” versehen. Die eigentliche Deklaration erfolgt dann später, wenn der dazu benötigte Typ erstellt worden ist:

```
type   TPtr    = pointer; forward;  // vorläufige Deklaration
       TRec1   = record
                   ...
                   Ptr1 : TPtr;
                 end;
       TPtr    = pointer to Trec1;  // Haupt Deklaration
```

Normalerweise zeigt ein Pointer in den Adressraum der CPU. Beim AVR sind dies Register-Page, IO-Page, InternRam-Page und ExternRam-Page. Alle diese Bereiche liegen in einem linearen Adressraum von $0000 bis $FFFF. Solange der Programmierer auf seine Pointer Arithmetik achtet, kann hier auch nichts (oder besser: nicht viel) passieren ('C' lässt grüssen). Fast unlösbar wird es aber, wenn der Programmierer mit einem Pointer in das EEprom oder das Flash zugreifen will. Der Compiler kann nicht wissen, dass eine Pointer Manipulation als Ergebnis einen Zugriff ins Flash haben soll. Hier hilft nur, wenn der Compiler angewiesen wird, dass der Zugriff **nicht** in den normalen Adressraum gehen soll. Der Compiler bietet dazu vordefinierte Pointer Typen an, die zu einer quasi Typkonvertierung benutzt werden können:

```
EEpromPtr (pointer)
FlashPtr (pointer)
UsrDevPtr (pointer)
BankDevPtr (bank; pointer)
```

EEpromPtr
Ein Zugriff über *EEpromPtr(pointer)* erfolgt in das EEprom mit der Adresse, die in dem Pointer abgelegt ist. Enthält (zeigt) der Pointer z.B. $100, erfolgt ein Zugriff auf Adresse $100 im EEprom.

FlashPtr
Ein Zugriff über *FlashPtr(pointer)* erfolgt in das ROM mit der Adresse, die in dem Pointer abgelegt ist. Enthält (zeigt) der Pointer z.B. $100, erfolgt ein Zugriff auf Adresse $100 im Flash.

UsrDevPtr
Ein Zugriff über *UsrDevPtr(pointer)* erfolgt in das UserDevice mit der Adresse, die in dem Pointer abgelegt ist. Enthält (zeigt) der Pointer z.B. $100, erfolgt ein Zugriff auf Adresse $100 im UserDevice.

BankDevPtr
Ein Zugriff über *BankDevPtr(bnk; pointer)* erfolgt in das Banked Device mit der Bank, die *bnk* enthält und der Adresse, die in dem Pointer abgelegt ist. Enthält (zeigt) der Pointer z.B. $100 und *bnk* enthält 2, erfolgt ein Zugriff auf Adresse $100 in Bank 2 des Banked Device.

```
type    TRec1    = record
                     Rbb    : byte;
                     Rww    : word;
                   end;
        TPtr1    = pointer to TRec1;

const   FRec1    : TRec1 = (Rbb : $AA; Rww : $1234);

{$EEPROM}
var     ERec1  : TRec1;

{$UDATA}
var     URec1  : TRec1;

{$BDATA 2}
var     BRec1  : TRec1;

{$IDATA}
var     IRec1   : TRec1;
        Ptr1    : TPtr1;
        bb      : byte;
        ww      : word;
...
...

begin
  Ptr1:= @FRec1;
  bb:= FlashPtr (Ptr1)^.Rbb;
  ww:= FlashPtr (Ptr1)^.Rww;

  Ptr1:= @ERec1;
  EEpromPtr (Ptr1)^.Rbb:= $ff;

  Ptr1:= @URec1;
  UsrDevPtr (Ptr1)^.Rbb:= $ff;

  Ptr1:= @BRec1;
  BankDevPtr (2, Ptr1)^.Rbb:= $ff;

  Ptr1:= @IRec1;
  Ptr1^.Rbb:= $ff;
end.
```

4.1.18.1 Pointer AutoIncrement AutoDecrement

Einer der ganz wenigen Vorteile von C gegenüber Pascal ist das Autoincrement/Decrement von Pointern. Das ist auch im AVRco möglich mit *Pointer^++* und *Pointer^--*.
Einzige Bedingung dafür ist, dass der Pointer selbst nur entweder global oder lokal (auf dem Prozedur/ Funktions Frame) liegt. Nicht möglich sind Pointer im Flash (klar), im EEprom etc.
Der Pointer muss dereferenziert werden, d.h. ein Inkrement ist nur möglich wenn über den Pointer mit "^" lesend oder schreibend zugegriffen wird. Ausnahme *pointer++;* und *Pointer^--*;
Es dürfen nur typisierte Pointer verwendet werden. Beim **AutoDecrement** werden nur Pointer für **8bit Typen** unterstützt, und diese Operation benutzt die **pre-decrement** Eigenschaft des AVRs.
++ bedeutet **post**-increment und **--** bedeutet **pre**-decrement.

Das Ziel/Quelle des Pointers könne globale, lokale, Flash oder EEprom Variablen und Konstante sein. Zu beachten ist dabei dass der Pointer immer um die Grösse des bewegten Objekts inkrementiert wird, d.h. bei Bytes um 1, bei Words um 2, bei Arrays um sizeOf(Array) und bei Records um sizeOf(record).
Bei Strings wird nicht um length(string) sondern um sizeOf(string) inkrementiert.

Das alles ist der Unterschied zu inc(pointer) wo der Pointer immer nur um 1 inkrementiert wird.
Durch das Autoinkrement von Pointern lassen sich sehr schnelle und kurze Schleifen realisieren.

```
pointer^++:= variable;
pointer^++:= contant;
EEpromPtr(pointer)^++:= variable;
variable:= pointer^++;
variable:= FlashPtr(pointer)^++;
variable:= EEpromPtr(pointer)^++;
pointer1^++:= pointer2^++;

var arr : array[0..7] of byte
Pointer := @arr + sizeOf(arr) ;
pointer^--:= variable;
pointer^--:= variable;
pointer^--:= contant;
variable:= pointer^--;
pointer1^--:= pointer2^--;
```
etc.

4.1.19 LONGWORD

32bit, 4 Bytes, 0..4294967295
Longword sind ganze Zahlen. Sie sind auf einen Bereich von 0 bis 4294967295 beschränkt.
Longword benötigen vier Byte Speicher.

```
var   lw  : longword;
```

4.1.20 WORD64

64bit, 8 Bytes, 0..18446744073709551615
Word64 sind ganze Zahlen. Sie sind auf einen Bereich von 0 bis 18446744073709551615 beschränkt.
Word64 benötigen acht Byte Speicher.

```
var   w64  : word64;
```

Achtung:
64bit Typen wie Word64, Int64 und Fix64 stehen nicht standardmässig zur Verfügung. Der entsprechende Typ muss explizit importiert werden.

```
from System Import Word64, Fix64;
```

4.1.21 LONGINT

32bit, 4 Bytes, -2147483648..2147483647

LongInt sind ganze Zahlen. Sie sind auf einen Bereich von -2147483648 bis 2147483647 beschränkt. LongInt benötigen vier Byte Speicher.

```
var  li  : longint;
```

4.1.22 INT64

64bit, 8 Bytes, -9223372036854775808 ... 9223372036854775807

Int64 sind ganze Zahlen. Sie sind auf einen Bereich von -9223372036854775808 bis 9223372036854775807 beschränkt. Int64 benötigen acht Byte Speicher.

```
var  i64  : int64;
```

Achtung:
64bit Typen wie Word64 und Int64 stehen nicht standardmässig zur Verfügung. Der entsprechende Typ muss explizit importiert werden.

```
from System Import Int64;
```

4.1.23 FLOAT

32bit, 4 Bytes, 6..9 Digits, 10E-38..10E38

```
var  f  : float;
```

Achtung:
32bit Typen wie LongInt, LongWord und Float stehen nicht standardmässig zur Verfügung. Der entsprechende Typ muss explizit importiert werden.

```
from System Import LongInt, Float;
```

4.1.24 Fix64

Fixed point Typ. 64 bit, 8 Bytes. Begrenzt auf max 2147483647.999999999 bis min -2147483648.000000000 Besteht aus einem 32bit ganzahligen und aus einem 32bit gebrochenen (Nachkomma) Teil.

```
var  f64  : fix64;
```

Achtung:
64bit Typen wie Word64, Int64 und Fix64 stehen nicht standardmässig zur Verfügung. Der entsprechende Typ muss explizit importiert werden.

```
from System Import Int64, Fix64;
```

4.1.25 ENUM

8bit, 0..255 Enumeration (Aufzählungstyp)

```
type  eKey = (Key1, Key2, Key3);    {Typ Deklaration}
var   Keys : eKey;                  {Variable vom Typ eKey}
                                    {läuft von Key1..Key3}
```

if Key2 in Keys then ...

Enumerationen können auch mit *SUCC* und *PRED*, sowie *INC* und *DEC* verarbeitet werden.
Bei *INC* und *DEC* werden die oberen und unteren Grenzen zwar beachtet, aber im Gegensatz zu *SUCC* und *PRED* ist ein wrap (durchdrehen) zugelassen.

Aufzählungstypen können mit auch "Lücken dazwischen" definiert werden.

```
type
  tEnum = (aaa, bbb, ccc=100, ddd, eee=200, fff);
              0   1     100     101   200     201   <- numerischen Werte
```

4.1.26 SEMAPHORE

8bit, Byte wird durch Processes oder Tasks importiert

var sema : Semaphore;

Semaphoren sind spezialisierte Variablen und dienen zur Prozess-Synchronisation.
Ein Prozess oder Task kann eine Semaphore inkrementieren oder dekrementieren. Ein Wert > 0 bedeutet für einen Task/Prozess z.B. dass eine bestimmte Funktion/Prozedur gesperrt ist.

Ein Zugriff auf Semaphoren kann nur mit den zugehörigen speziellen Funktionen/Prozeduren erfolgen, da der Zugriff gegen Interrupts (Task Wechsel) geschützt werden muss.

Status einer Semaphore (byte)
Function SemaStat (sema : semaphore) : byte;

Inkrementieren einer Semaphore
Procedure IncSema (sema : semaphore);

Dekrementieren einer Semaphore
Function DecSema (sema : semaphore) : boolean;

Process wartet auf Semaphore > 0
Function WaitSema (sema : semaphore [; timeout: word]) : boolean;
Der TimeOut Parameter ist optional. Wird er weggelassen, muss der Prozess warten, bis die Semaphore > 0 ist. Das gleiche gilt auch, wenn TimeOut auf 0000 gesetzt wird. Mit einem Wert > 0 erfolgt nach (TimeOut * SysTicks) der Abbruch der Wait Funktion. Das Funktions Ergebnis ist true wenn kein Timeout aufgetreten ist. In Tasks ist die Angabe eines TimeOut nicht möglich bzw. wird ignoriert.

4.1.27 PIPE

generisch. Wird vorwiegend von Processes oder Tasks benutzt

Import:
from System import …, …, Pipes;

4.1.27.1 Pipe für ordinale Typen

Eine Pipe ist ein Speicher, der als ein sog. FIFO organisiert ist.
Die Länge der Pipe bzw. die mögliche Anzahl der Parameter/Variablen beträgt max. 255. Ein Zugriff auf Pipes kann nur mit den zugehörigen Funktionen/Prozeduren erfolgen, da der Zugriff gegen Interrupts (Task Wechsel) geschützt werden muss.

Deklaration:
var *Pipe1 : Pipe[16] of byte; // int8, boolean, word, integer etc.*

Status einer Pipe (byte) = Anzahl der vorhandenen Parameter
Function *PipeStat (pipe1 : pipe) : byte;*
Die Funktion PipeStat ist auch auf RxBuffer, RxBuffer1, -2, -3 und TxBuffer, TxBuffer1, -2, -3 der seriellen Schnittstellen anwendbar.

Status einer Pipe (boolean)
Function *PipeFull (pipe1 : pipe) : boolean;*

Parameter in Pipe anfügen
Function *PipeSend (pipe1 : pipe; parm : type) : boolean;*

Parameter aus Pipe abholen
Function *PipeRecv (pipe1 : pipe) : type;*
Wartet bis mindestens ein Parameter in der Pipe vorhanden ist und gibt dann den ältesten Eintrag zurück.

Function *PipeRecv_ND (pipe1 : pipe) : type;*
PipeRecv_ND gestattet das Auslesen einer Pipe, ohne den Inhalt selbst zu verändern "non-destructive-readout".

Pipe leer machen
Procedure *PipeFlush (pipe1 : pipe);*

Process oder Task wartet auf Pipe
Function *WaitPipe (pipe1 : pipe [; timeout: word]) : boolean;*
Der TimeOut Parameter ist optional. Wird er weggelassen, muss der Prozess warten, bis das ein Datum in der Pipe steht. Das gleiche gilt auch, wenn TimeOut auf 0000 gesetzt wird. Mit einem Wert > 0 erfolgt nach (TimeOut * SysTicks) der Abbruch der Wait Funktion. Das Funktions Ergebnis ist true wenn kein Timeout aufgetreten ist.

4.1.27.2 Pipe of Bit

Dieser Pipe Typ ist nützlich um Resourcen bzw. Speicher zu sparen.

BitPipe : pipe[xx] ***of*** *bit;*

Die Verarbeitung dieser Pipe ist identisch mit einer Pipe of boolean. Es wird allerdings pro Eintrag nur ein Bit verwendet. Dabei ergeben 8 Bits ein Byte. Die Zugriffe erfolgen nach dem FIFO Prinzip. Das zuerst geschriebene Bit wird auch als erstes ausgelesen.
Es sind bis zu 248 Bits möglich, was einem Speicherbedarf von 36 Bytes entspricht.

4.1.27.3 Pipe für komplexe Typen

Pipes können auch Strings, Arrays und Records enthalten.
Speziell zum Auslesen von diesen 3 komplexen Type gibt es die Funktion

Function *PipeRecv (Pp : Pipe;* ***var*** *Value : PipeType{record, array, string} [, doWait : boolean]) : boolean;*
Der Parameter Record, Array oder String muss ein vordefinierter Typ sein.
Der Funktion muss als Ziel Adresse der Namen der Ziel-Variablen übergeben werden. Über den optionalen Schalter "doWait" wird bestimmt ob die Funktion mit einen FALSE zurückkehrt, wenn nichts vorhanden ist, oder solange in einer Schleife wartet, bis ein Empfang vorliegt.
Ist der Schalter "doWait" nicht vorhanden, wartet die Funktion immer auf ein gültiges Datum und das Ergebnis ist somit immer true.

Function *PipeRecv_ND (P : Pipe;* ***var*** *Value : PipeType{record, array, string} [, doWait : boolean]) : boolean;*
Das selbe wie obige Funktion, nur dass die Pipe ausgelesen wird, ohne den Inhalt selbst zu verändern "non-destructive-readout".

4.1.28 SYSTIMER

16bit, Word wird durch SysTick importiert
Variable wird bei jedem System Tick dekrementiert, falls sie > 0 ist.

```
var  Timer1 : SysTimer[, UpCount];          {Variable vom Typ SysTimer}

SetSysTimer (Timer1, 50000);
repeat until GetSysTimer (Timer1) = 0;
```

(besser ist ->)

```
repeat until isSysTimerZero (Timer1);
```

Für einfache Messfunktionen ist ein UpCounter sinnvoll. Dieser wird mit *ResetSysTimer* gestartet. Das System inkrementiert bei jedem SysTick den Timer. Ein Überlauf wird verhindert, der maximale Wert ist deshalb $FFFF.
Die SysTimer Funktionen "*GetSysTimer, ResetSysTimer, SetSysTimer*" sind auch für die UpCounter anwendbar. Irrelevant ist für diese die Funktion "*isSysTimerZero*".

Es wird eine maximale Anzahl von 16 SysTimern unterstützt (SysTimer +SysTimer8+SysTimer32).

4.1.29 SYSTIMER8

8bit, Byte wird durch SysTick importiert
Variable wird bei jedem System Tick dekrementiert, falls sie > 0 ist.

```
var  Timer2 : SysTimer8[, UpCount];          {Variable vom Typ SysTimer8}

SetSysTimer (Timer2, 50);
repeat until isSysTimerZero (Timer2);
```

Während eines Zugriffs werden Interrupts nicht disabled. Deshalb ist ein Zugriff auf einen solchen Timer kürzer und schneller als der Zugriff auf einen „SysTimer“.
Für einfache Messfunktionen ist ein UpCounter sinnvoll. Dieser wird mit *ResetSysTimer* gestartet. Das System inkrementiert bei jedem SysTick den Timer. Ein Überlauf wird verhindert, der maximale Wert ist deshalb $FF.
Die SysTimer Funktionen "GetSysTimer, ResetSysTimer, SetSysTimer" sind auch für die UpCounter anwendbar. Irrelevant ist für diese die Funktion "*isSysTimerZero*".

Es wird eine maximale Anzahl von 16 SysTimern unterstützt (SysTimer +SysTimer8+SysTimer32).

4.1.30 SYSTIMER32

32bit, LongWord wird durch SysTick importiert
Variable wird bei jedem System Tick dekrementiert, falls sie > 0 ist.
Damit sind extrem lange Zeiten erreichbar. Dazu muss LongWords importiert werden.

```
from  System import longword;
var  Timer3 : SysTimer32[, UpCount];          {Variable vom Typ SysTimer32}

SetSysTimer (Timer3, 100000);
repeat until isSysTimerZero (Timer3);
```

Für einfache Messfunktionen ist ein UpCounter sinnvoll. Dieser wird mit ResetSysTimer gestartet. Das System inkrementiert bei jedem SysTick den Timer. Ein Überlauf wird verhindert, der maximale Wert ist deshalb $FFFFFFFF.
Die SysTimer Funktionen "GetSysTimer, ResetSysTimer, SetSysTimer" sind auch für die UpCounter anwendbar. Irrelevant ist für diese die Funktion "isSysTimerZero".

Es wird eine maximale Anzahl von 16 SysTimern unterstützt (SysTimer +SysTimer8+SysTimer32).

4.1.31 PIDCONTROL

Pseudo-Record
PID-Regler, wie er häufig in technischen Anwendungen eingesetzt wird,
z.B. Temperatur Regelung, Servos, Drehzahlregler etc.

PID-Regler haben zwei Eingangsparameter: der *Sollwert* = '*Nominal'* und der *Istwert* = '*Actual'*.
Vier Parameter, die normalerweise nur einmal eingestellt werden, sind *pFactor*, *iFactor*, *dFactor* und *sFactor*.
Die Stellgrösse, die an den Aktuator (Heizung, Motor etc.) geht, wird durch die Funktion '*Execute'* errechnet.

Den Reglertyp bestimmen die zwei Initialisierungs-Parameter '*iLimit'* und '*dIntVal'*.
Die internen Variablen *pValue, iValue, dValue* können gelesen werden, sollten aber nicht verändert werden.

iLimit
ist vom Typ LongWord (0..100000) und bestimmt die maximale Grösse des **I**-Anteils (clipping). Ist iLimit = 0 wird der **Integral**-Wert des Reglers nicht berechnet und entfällt dadurch (z.B. PD-Regler).

dIntVal
ist vom Typ Byte (0, 1, 2, 4, 8, 16, 32) und bestimmt die Schrittweite zur Berechnung des **D**-Anteils (Steigung).
Ist dIntVal = 0 wird der **Differential**-Wert des Reglers nicht berechnet und entfällt dadurch (z.B. PI-Regler).
Ist der Wert = 1, wird die Steigung vom letzten Fehlerwert zum aktuellen Fehlerwert gerechnet.
In den restlichen Fällen wird ein entsprechendes Array gebildet, das die Historie der letzten n Fehlerwerte aufnimmt. Dadurch kann die Steigung über eine grössere Anzahl von Sollwerten gebildet werden.

Der Regler rechnet selbst mit **LongInteger**. Overflow bei Execute sind deshalb nicht wahrscheinlich.

```
var  Pid1 : PIDcontrol[iLimit, dIntVal];

{Init}
Pid1.pFactor:= 1000;
Pid1.iFactor:= 2500;
Pid1.dFactor:= 678;
Pid1.sFactor:= 10000;

{Run}
Pid1.Actual:= 500;
Pid1.Nominal:= 550;
PWM1:= Pid1.Execute;
```

4.1.32 Alias Synonyme

Mit einer Alias Definition können Konstante, Variablen, Funktionen und Prozeduren einen weiteren Namen zugewiesen werden. Der Zweck einer solchen Zuweisung ist dass an beliebigen Stellen eines Projekts das Alias verwendet wird und bei einer Änderung (z.B. anderer UART) nur das Alias geändert werden muss ohne im ganzen Projekt die Begriffe austauschen zu müssen.

Aliase sollten immer global definiert werden und sollten auch nur auf globale Konstanten, Vars etc. angewendet werden. In Units im Interface Teil selbst müssen sowohl die Aliase als auch die referenzierten Typen im Interface Teil liegen.

```
var
  bo            : boolean;
  bb            : byte;
  ww            : word;

alias
  aBo           = bo;
  aBb           = bb;
  aWW           = ww;
  aSerOut       = SerOutC0;
  aSerInp       = SerInpC0;
  aSerStat      = SerStatC0;
  aSPI_IO       = SPIinOutD;
```

Manchmal ist es sinnvoll Aliase zuerst zu definieren bevor Units importiert werden. Damit haben alle Units gleich Zugriff auf die Aliase. Dazu muss dann im Main unmittelbar nach dem letzten Define Statement und bevor dem Unit Import mit "uses" eine oder mehrere Alias Definitionen im Main eingefügt werden:

```
Define
  RTCsource      = SysTick;

alias
  SerOutPC      = SerOutD1;
  SerStatPC     = SerStatD1;
  SerInpPC      = SerInpD1;

uses uuuuu;

Implementation
```

Ein Beispiel Programm befindet sich in der Demos Directory unter “XMEGA_Alias“.

4.1.33 Delphi – AVRco Typen Vergleich

AVRco	size	Delphi	Notes
Boolean	1 byte 8 bits	Boolean	Delphi 0 = false, <> 0 = true AVRco 0 = false, $FF = true
Byte	1 byte 8 bits	Byte	0..255
Int8	1 byte 8 bits	ShortInt	-128..+127
Char	1 byte 8 bits	AnsiChar	Delph9 and later use AnsiChar Older Delphi char can be used
Word	2 bytes 16 bits	Word	0..65536
Integer	2 bytes 16 bits	SmallInt	-32768..32767
LongWord	4 bytes 32 bits	LongWord	0..4294967295 0..2^32-1

LongInt	4 bytes 32 bits	Integer	-2^31..+2^31-1
Word64	8 bytes 64 bits	uInt64	0..2^64-1
Int64	8 bytes 64 bits	Int64	-2^63..+2^63-1
Float	4 bytes 32 bits	Single	1.5x10^-45..3.4x10^38
Fix64	8 bytes 64 bits	not applicable	-2147483648.000000000 .. 2147483647.999999999
String	1..256 bytes	ShortString	first byte in string contains length byte
Array		packed Array	
Record		packed Record	

4.2 Operatoren

4.2.1 NOT

*a:= **not** a; {invertiert var a }*

Als Operanten sind nur die Typen Byte, Int8, Boolean, Integer, Word, Longint, Longword, Word64, Int64 und Fix64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $FF, so hat sie nach der Operation den Wert $00. Beachten Sie auch die ***Negate*** Funktion.

4.2.2 DIV

*a:= a **div** b; {Division ganzer Zahlen}*

Als Operanten sind nur die Typen Byte, Int8, Integer, Word, Longint, Longword, Word64, Int64 und Fix64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $10 und 'b' den Wert $2, so hat 'a' nach der Operation den Wert $08.

4.2.3 MOD

a:= a **mod** *5; {Modulo ganzer Zahlen}*

Als Operanten sind nur die Typen Byte, Int8, Integer, Word, Longint, Longword, Int64 und Word64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $06 , so hat 'a' nach der Operation den Wert $01. Mit a = -6 ist das Ergebnis -1. Modulo hat als Ergebnis also immer den Rest von einer Division mit den gleichen Werten.

4.2.4 AND

*a:= a **and** $0f; {And Maske}*

Als Operanten sind nur die Typen Byte, Int8, Integer, Word, Longint, Longword, Int64 und Word64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $13 , so hat 'a' nach der Operation den Wert $03. 'And' lässt also im Ergebnis nur diejenigen Bits gesetzt, die sowohl im Operanten als auch in der Maske gesetzt waren. 'And' wird auch bei bool'schen Operationen verwendet.

***if** (a > b) **and** (a < c) **then** ... **endif**;*

4.2.5 OR

*a:= a **or** $30; {Or Maske}*

Als Operanten sind nur die Typen Byte, Int8, Integer, Word, Longint, Longword, Word64 und Int64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $09, so hat 'a' nach der Operation den Wert $39. 'Or' setzt im Ergebnis zusätzlich auch die Bits, die in der Maske gesetzt waren.

'Or' wird auch bei bool'schen Operationen verwendet.
***if** (a > b) **or** (a < c) **then** ... **endif**;*

4.2.6 XOR

*a:= a **xor** 1; {Xor Maske}*

Als Operanten sind nur die Typen Byte, Int8, Integer, Word, Longint, Longword, Word64 und Int64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $01, so hat 'a' nach der Operation den Wert $00. 'Xor' setzt im Ergebnis alle Bits auf 0, die sowohl in der Maske als auch im Operanten gesetzt waren.

4.2.7 SHL

*a:= a **shl** 5; {links schieben}*

Als Operanten sind nur die Typen Byte, Int8, Integer, Word, Longint, Longword, Word64 und Int64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $03, so hat 'a' nach der Operation den Wert $60. 'SHL' schiebt alle Bits im Operanten nach links und füllt dabei die freiwerdenden Bits von rechts her mit '0' auf.

4.2.8 SHLA

*a:= a **shla** 5; {arithmetisch links schieben}*

Als Operanten sind nur die Typen Int8, Integer, Longint und Int64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $03, so hat 'a' nach der Operation den Wert $60. 'SHLA' schiebt alle Bits im Operanten nach links und füllt dabei die freiwerdenden Bits von rechts her mit '0' auf. Im Gegensatz zu ‘SHL’ bleibt aber das höchstwertige Bit unverändert. Damit ist sichergestellt, dass eine negative Zahl auch immer negativ bleibt.

4.2.9 SHR

*a:= a **shr** 4; {rechts schieben}*

Als Operanten sind nur die Typen Byte, Int8, Integer, Word, Longint, Longword, Word64 und Int64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $81, so hat 'a' nach der Operation den Wert $08. 'SHR' schiebt alle Bits im Operanten nach rechts und füllt dabei die freiwerdenden Bits von links her mit '0' auf.

4.2.10 SHRA

*a:= a **shra** 4; {arithmetisch rechts schieben}*

Als Operanten sind nur die Typen Int8, Integer, Longint und Int64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $71, so hat 'a' nach der Operation den Wert $07. 'SHRA' schiebt alle Bits im Operanten nach rechts und füllt dabei die freiwerdenden Bits von links her mit dem Wert des höchstwertigen Bits auf. Im Gegensatz zu ‘SHR’ bleibt das höchstwertige Bit unverändert. Damit ist sichergestellt, dass eine negative Zahl auch immer negativ bleibt.
Achtung: bei negativen Werten von ***a*** wird das Ergebnis Richtung positiv aufgerundet!

4.2.11 ROL

*a:= a **rol** 4; {links rotieren}*

Als Operanten sind nur die Typen Byte, Int8, Integer, Word, Longint, Longword, Word64 und Int64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $81, so hat 'a' nach der Operation den Wert $18. 'ROL' rotiert alle Bits im Operanten nach links. Alle Bits bleiben erhalten, tauschen aber ihre Positionen.

4.2.12 ROR

*a:= a **ror** x; {rechts rotieren}*

Als Operanten sind nur die Typen Byte, Int8, Integer, Word, Longint, Longword, Word64 und Int64 zulässig. Hat im obigen Beispiel die Variable 'a' den Wert $01 und 'x' den Wert $02, so hat 'a' nach der Operation den Wert $40. 'ROR' rotiert alle Bits im Operanten nach rechts. Alle Bits bleiben erhalten, tauschen aber ihre Positionen.

4.2.13 IN

```
if v2 in ['a'..'g'] then ...          {Enum auswerten}
if Key in [Key2..Key4] then ...       {Enum auswerten}
if x in[45..56] then ..
```

Als Argumente sind Enums und alle ordinalen Typen zulässig, Byte, Char, ... LongInt, sowie Float.

4.2.14 +

a:= a + 5; {addieren}

Als Operanten sind nur die Typen Byte, Int8, Integer, Word, Longint, Longword, Float und Fix64 zulässig. Ein Sonderfall sind Stringoperationen.

4.2.15 -

a:= a - b; {subtrahieren}

Als Operanten sind nur die Typen Byte, Int8, Integer, Word, Longint, Longword, Float und Fix64 zulässig. Wenn das Minuszeichen als Vorzeichen verwendet werden soll, **muss** dem Compiler dies durch Klammerung mitgeteilt werden.

*a:= a * (- b); {mit negativem Wert multiplizieren}*

4.2.16 /

a:= a / 5.5; {Float Division oder Fix64 Division}

4.2.17 *

*a:= a * %11000100; {Multiplikation, ordinale und Float, Fix64}*

*f:= f * 1.5;*

4.3 Unechte Operatoren

4.3.1 @

```
p:= @a;          {Adresse der Speicherstelle}
```

Als Operanten sind nur Variablen, Prozeduren und Funktionen zulässig, denn nur diese haben auch eine physikalische Adresse. Nach der Operation enthält ‘p’ die Adresse von ‘a’. Eigentlich ist '@' kein Operator sondern eine Systemfunktion. Das Ziel ist normalerweise ein Pointer.

4.3.2 ^

```
x:= p^;          {Variable als Pointer}
```

Als Operanten sind nur Variable vom Typ Pointer zulässig. Nach der Operation enthält ‘x’ den Wert, der sich in der Speicherstelle befindet, auf die ‘p’ zeigt. Eigentlich ist '^' kein Operator sondern eine Systemfunktion.

Weiterhin ist ein vorangestellter '^' vor einem Char die Anweisung an den Compiler, das nachfolgende Zeichen als Steuerzeichen zu interpretieren.

Ein **^G** z.B. erzeugt ein sog. Bell-Zeichen (hex 09). Der Compiler führt dazu folgende Operation aus: *result := 'G' - '@'.* ($49 - $40).

```
const Bell = ^G;     {Control G}
```

4.3.3

```
const LF  = #10;     {Line Feed}
```

Das Nummern-Zeichen ist neben dem ^ eine weitere Möglichkeit Steuerzeichen (nicht darstellbare Buchstaben) zu definieren. Das dem Zeichen nachfolgende Argument muss eine Dezimalzahl im Bereich 0..255 sein.

Auch '#' ist eigentlich kein Operator sondern eine Systemfunktion.

4.3.4 $

```
const  x1 = $10;    {Dezimal 16}
```

Das Dollar Zeichen erklärt die nachfolgende Konstante zum Hexadezimal Wert.

4.3.5 %

```
const  b1 = %10100101;    {Hex $A5, Dezimal 165}
```

Das Prozent Zeichen erklärt die nachfolgende Konstante zum Binär Wert.

4.4 Benutzer definierte Sprach Elemente

4.4.1 Identifier

Identifier (Namen) werden gebraucht um Labels, Konstante, Typen, Variable, Prozeduren, und Funktionen zu deklarieren. Ein Identifier besteht aus einem Buchstaben gefolgt von einer Kombination aus Buchstaben, Zahlen, oder Unterstreichungsstrichen. Ein Identifier ist auf 64 Zeichen beschränkt, wobei jedes Zeichen wichtig ist.

```
PASCAL
square
persons_counted
BirthDate
3rdRoot                 // illegal, fängt mit einer Zahl an
Two Words               // illegal, darf keine Leerzeichen enthalten
```

Da der AVRco nicht zwischen Gross- und Kleinschreibung unterscheidet, hat der Gebrauch von Gross- und Kleinbuchstaben, wie z.B. in *BirthDate* keine Bedeutung. Trotzdem wird empfohlen, davon Gebrauch zu machen, denn es führt zu besserer Lesbarkeit.
VeryLongIdentifier ist einfacher zu lesen als *VERYLONGIDENTIFIER.*

4.4.2 Zahlen

Zahlen sind Konstante des Typs Byte, Int8, Word, Integer, LongInt, LongWord, Int64, Word64, Float oder Fix64. Integer (Byte, Word etc) Konstante sind ganze Zahlen dargestellt entweder in dezimal, hexadezimal oder binärer Schreibweise. Hexadezimal Konstante erkennt man an dem vorangestellten Dollar Zeichen: $1234 ist eine hexadezimale Konstante. Binäre Konstante haben ein vorangestelltes Prozent Zeichen: %1011100 ist eine Binäre Konstante.

Der dezimale longint Bereich geht von -2147483648 bis +2147483647, der hexadezimale longword/longint Bereich geht von $0 bis $FFFFFFFF.
Der binäre longint/longword Bereich geht von %0 bis %11111111111111111111111111111111 (32 * 1).

```
12345
-1
$123
$ABC
$123G           // illegal, G ist keine legale hexadezimale Zahl
%1011
%1003           // illegal, 3 ist keine legale binäre Zahl
1.2345          // illegal für ein integer, enthält einen gebrochenen Anteil
```

4.4.3 Strings

Eine String Konstante ist eine Reihe von Zeichen in Hochkomma eingeschlossen:

```
'This is a string constant'
```

String, die nur ein Zeichen enthalten, können auch vom Typ Char sein. Der Kontext bestimmt hier den Typ.

```
'PASCAL'
'You''ll see'
''
```

Ein einzelnes Hochkomma in einem String wird als zwei aufeinander folgende Hochkommas geschrieben. Das letzte Beispiel – Hochkommas ohne ein eingeschlossenes Zeichen, *Leerstring* – ist ein String Typ und **nicht** mit dem Typ *Char* kompatibel.

4.4.4 Steuerzeichen

Der AVRco erlaubt auch Steuerzeichen, die in Strings eingeschlossen sind
Das # Symbol gefolgt von einer Byte Konstanten im Bereich 0..255 stellt ein ASCII Zeichen dar in der Auswahl der ASCII Tabelle 0..255

Der AVRco erlaubt auch den Gebrauch des ^ Zeichens, gefolgt von einem Buchstaben, um Steuerzeichen darstellen zu können.

```
#10          // ASCII 10 dezimal (Line Feed).
#$1B         // ASCII 1B hex (Escape).
^G           // ASCII 07 hex (Bel)
```

Sequenzen von Steuerzeichen können in Strings zusammengefasst werden, indem zwischen ihnen ein + eingesetzt wird:

```
#13 + #10
#27 + #20
```

Steuerzeichen können auch mit Text Strings gemischt werden:

```
'Waiting for input!'+#7+#7+#7+'Please wake up'
```

4.4.5 Kommentare

Ein Kommentar kann fast überall im Programm verwendet werden, wo ein Delimiter legal ist. Ein Kommentar wird entweder durch geschweifte Klammern { und } oder durch die Symbole (* und *) eingeschlossen.
Es ist auch die Symbole // zu verwenden, um den Rest der Zeile zum Kommentar zu erklären.

```
{Das ist ein Kommentar}
(* und das auch *)
```

Geschweifte Klammern können innerhalb geschweiften Klammern verschachtelt werden und (* *) können innerhalb (* *) verschachtelt werden.
Geschweifte Klammern können innerhalb (* *) verschachtelt werden und umgekehrt.
Somit ist es möglich ganze Code-Teile auszukommentieren, auch wenn diese Kommentare enthalten.

4.5 Ausdrücke (Expressions)

Expressions (Ausdrücke) sind algorithmische Konstruktionen die Vorgehensweise zum Errechnen eines Wertes festlegen. Sie bestehen aus Operanten, Variablen, Konstanten, und Funktions-Namen (designators) verbunden durch Operatoren.

4.5.1 Operatoren

Operatoren fallen in fünf Kategorien:

1) Unary minus (minus mit nur einem Operanten)
2) Not Operator.
3) Multiplying Operatoren: *, /, div, mod, shl, shr.
4) Adding Operatoren: +, -, or, and, xor.
5) Relationale Operatoren: =, < > , > , < , <= ,>= , in.

Sequenzen von Operatoren aus der gleichen Kategorie werden von links nach rechts abgearbeitet. Ausdrücke in Klammern werden zuerst berechnet, unabhängig von vorangestellten oder nachfolgenden Operatoren.
Die Operanten müssen vom gleichen Typ sein, ein automatisches Type casting findet nicht statt.

Achtung
Die Wertigkeit der Operatoren (* AND SHR etc.) wird z.Zt. noch nicht überall eingehalten.
Deshalb ist das Klammern von Ausdrücken unbedingt notwendig.

4.5.1.1 Unary Minus

Das Unary Minus ergibt eine Negation des Operands, der ein *Float, Longint, Int8,* oder *Integer* Typ sein kann.

4.5.1.2 Not Operator

Der Not Operator invertiert den logischen Wert des bool'schen Operanten:
***not** True = False*
***not** False = True*
Im AVRco kann der "not" Operator auch dazu verwendet werden, um einen Wert zu invertieren. Der Operand muss dazu vom Typ byte, integer, longint, word oder longword sein.

4.5.1.3 Multiplizierende Operatoren

Operator	Operation	Typen
*	Multiplikation	Float, Fix64, Longint, Longword, Word, Integer, Int8, Byte, Bitset
/	Division	Float, Fix64, Bitset
div	Division	Longint, Longword, Word, Integer, Int8, Byte
mod	Modulus	Longint, Longword, Word, Integer, Int8, Byte
and	Arithmetic And	Longint, Longword, Word, Integer, Int8, Byte
and	Logical And	Boolean
shl	Shift Left	Longint, Longword, Word, Integer, Int8, Byte
shr	Shift Right	Longint, Longword, Word, Integer, Int8, Byte
rol	Rotate Left	Longint, Longword, Word, Integer, Int8, Byte
ror	Rotate Right	Longint, Longword, Word, Integer, Int8, Byte

*12 * 34* *// = 408*
123 / 4 *// = 30.75*
*123 **div** 4* *// = 30*
*12 **mod** 5* *// = 2*
*True **and** False* *// = False*
*12 **and** 22* *// = 4*
*2 **shl** 7* *// = 256*
*256 **shr** 7* *// = 2*

4.5.1.4 Addierende Operatoren

Operator	Operation	Typen
+	Addition	Float, Fix64, Longint, Longword, Word, Integer, Int8, Byte, String, Char, Bitset
-	Subtraktion	Float, Fix64, Longint, Longword, Word, Integer, Int8, Byte, Bitset
or	Arithm. Or	Longint, Longword, Word, Integer, Int8, Byte, Bitset
or	Logical Or	Boolean
xor	Arithm. Xor	Longint, Longword, Word, Integer, Int8, Byte
xor	ogical xor	Boolean

```
123+456          // = 579
456-123          // = 333
True or False    // = True
12 or 22         // = 30
True xor False   // = True
12 xor 22        // = 26
```

4.5.1.5 Relationale Operatoren

Die relationalen Operatoren arbeiten mit den scalaren Typen: *Float, Fix64, Longword, Longint, Word, Integer, Boolean, Char, Int8,* and *Byte* als auch mit *Bitsets.*
Der Ergebnis Typ ist dabei immer ein boolean, d.h. *True* oder *False.*

=	gleich mit
<>	ungleich mit
>	grösser als
<	kleiner als
>=	grösser als oder gleich
<=	kleiner als oder gleich

```
a = b       // true wenn a ist gleich b
a <> b      // true wenn a ist ungleich b
a > b       // true wenn a ist grösser b
a < b       // true wenn a ist kleiner b
a >= b      // true wenn a ist grösser oder gleich b
a <= b      // true wenn a ist kleiner oder gleich b
```

Es ist auch ein Vergleich von Record und Array Typen möglich.
Bedingung dafür ist, dass die Variablen vom gleichen Typ sind, d.h. sie müssen als Typ-Deklaration implementiert sein:

```
Type TRec  = record
               ...
             end;

var Rec1, Rec2 : TRec;

if Rec1 <> Rec2 then ...
bool := Rec1 = Rec2;
```

4.5.2 Funktions Designatoren (Namen)

Ein Funktion Designator ist ein Funktions Namen, optional gefolgt von einer in Klammern gesetzte Parameter Liste, die aus Variablen oder Ausdrücken bestehen kann, unterteilt mit Kommas. Das Erscheinen eines Funktions Designators (Namen) führt zur Abarbeitung dieser Funktion mit Übergabe der Parameter-liste. Wenn die Funktion keine System-Funktion ist, muss sie vorher deklarariert werden, bevor man sie ansprechen kann.

```
Z:= func1 (x);      // func1 ist eine zuvor deklarierte Funktion
F:= sqr (a);        // sqr ist eine System Funktion für das Quadrat des Arguments
```

4.6 Schlüsselwörter

4.6.1 PROGRAM

Start des Programms. Aus formalen Gründen ist ein bestimmter Programmrahmen notwendig. Dieser beginnt mit *'Program name*' und endet mit *'end*.'

```
Program Test;
Device ...                          {Hardware Deklaration}
Import ...                          {System Funktionen}
from system Import ...              {Typen und Funktionen}
Define ...                          {Hardware Definition}
Implementation                      {Programm Start}
const  ...                          {Konstanten Deklaration}
var ..    .                         {Variablen Deklaration}

Procedure System_Init;              {private Initialisierung}
begin
  ...
end;

Interrupt Timer1;                   {Interrupt Deklaration}
begin
  ...
end;

Process Pxx (14,20 : iData);        {Prozess Deklaration}
begin
  ...
end;

Task Txx (iData);                   {Task Deklaration}
begin
  ...
end;

Procedure ABC (z : integer);        {Prozedur Kopf}
Var xy : byte;
begin
  ...
end;

Function CDE : boolean;             {Funktions Kopf}
Var a, b : byte;
begin
  ...
Return(a > b);                      {Ergebnis der Funktion}
end;

begin                               {Main = Haupt Programm}
  ABC (i);
  x:= CDE;
end.
```

Bemerkung:

Der Compiler besitzt in der ***Standard Version*** keinen Linker. Es ist daher nicht möglich, sogenannte Units zu bilden oder externen Code dazu zu linken. Das scheint auf den ersten Blick eine Einschränkung zu sein. In der Praxis ist dies jedoch kein besonderer Nachteil, da der generierte Code und damit die Programmgrösse durch die relativ kleinen Programmspeicher (typ. 8kB) der Ziel-CPU's beschränkt ist.
Immer wiederkehrende Programmteile können mittels der Compilerschalter “Include” *{$I ...}* und *{$J...}* eingebunden werden.

Weiterhin erlaubt der oft relativ kleine Arbeitsspeicher der CPU's keinen ausufernden Stack- bzw. Frame-Betrieb, was lokale Variable innerhalb Prozeduren und Funktionen einschränkt.

Im Gegensatz zu bekannten Programmier-Systemen für den PC (Turbo Pascal, Delphi etc.), wo der Compiler die Hardware ganz genau kennt, ist bei einem Compiler für Embedded Systeme (SingleChips) die Hardware, ja sogar die CPU nur in ungefähr dem Compiler bekannt.
Es ist daher zwingend notwendig, dass der Programmierer die Zielhardware/CPU genau spezifiziert. Wegen den oft sehr eingeschränkten System Resourcen (Ram/Rom) muss weiterhin sorgfältig überlegt werden, welche Funktionen des Compilers (System) wirklich benötigt werden.

Um dem Compiler die vorhandene Hardware und die gewünschten Funktionen exakt spezifizieren zu können, braucht man diverse Anweisungen (Device, Import und Define). Diese drei Definitionen sind zwingend vorgeschrieben und ***müssen spezifiziert*** werden in obenstehender Reihenfolge.

Weiterhin benötigt der Compiler zu genauen Lage des jeweiligen Rams und Roms sowie deren Grösse als auch der implementierten Hardware in der CPU (SCI, ADC etc) ein **Description-File**, das diese Daten enthält. Diese Daten sind z.B. in *P90S8515.dsc* zu finden.

4.6.2 DEVICE

Prozessor und Hardware Spezifikation.

Die Argumente für “Device” sind absolut Prozessor spezifisch. Der Prozessor muss an erster Stelle stehen. Je nach Prozessor Familie werden die Parameter im Compiler, im Assembler oder in beiden Programmteilen ausgewertet. Zum Teil fliessen diese Daten in das HexFile mitein, und werden auch vom Programmiergerät gebraucht. Die Device Parameter Namen entsprechen denen der jeweiligen Prozessor Datenbücher und sind auch dort spezifiziert.

Der verwendete CPU-Namen muss identisch sein mit dem Prozessor-Steuerfile. Dem Dateinamen wird immer ein ‘P’ vorangestellt.

Beispiel für eine Device Anweisung des AVR AT90S2313:

***device** = 90S2313;*

Das zugehörige Steuerfile muss dann heissen: P90S2313.dsc

4.6.3 IMPORT

Import von **Hardware abhängigen** Systemfunktionen

Wie schon weiter vorne erwähnt, zwingen die zum Teil recht bescheidenen Resourcen der mancher hier implementierten Prozessoren dazu, nur die notwendigsten Systemfunktionen automatisch einzubinden. Speicher- oder Programm intensive und auch etwas exotischere Funktionen müssen explizit importiert werden.
Bestimmte Imports wiederum benötigen eine zusätzliche Spezifikation, die mittels *Define* durchgeführt wird. Manche Funktionen importieren automatisch zusätzliche Funktionen, die dann u.U. mittels Define weiter spezifiziert werden müssen. Die Beschreibung der einzelnen Import Funktionen sind in den ***Driver Manuals*** bei den jeweiligen Treibern zu finden.

Viele Funktionen setzen eine entsprechende Hardware in der CPU voraus. SysTick benötigt z.B. einen Hardware Timer oder SwitchPort ein Port. Weiterhin benötigen manche Imports den SysTick, z.B. der ADCPort. Deshalb sollte SysTick immer zuerst importiert werden.

```
Import SYSTICK, SWITCHPORT1;
```

4.6.4 FROM

Mit FROM wird gezielt aus einer bestimmten Bibliothek ein Typ, Function oder Procedure importiert.

```
From System Import LongInt, LongWord, Float;
```

Mögliche Imports von **Typen** und Systemfunktionen:

Fix64	
Float	
Int64	
LongInt	
Word64	
LongWord	
Processes	Sleep, Suspend, Resume, Priority, WaitSema ...
Tasks	Sleep, Suspend, Resume, Priority, WaitSema ...
Pipes	PipeFlush, PipeSend, PipeRecv, PipeStat, PipeFull ...
Pids	pFactor, iFactor, dFactor, sFactor, Actual, Nominal, Execute
OEMcharSet	das System benutzt für chars und strings nun den OEM Zeichensatz

```
From System Import LongInt, LongWord, Float, Pipes;
```

4.6.5 DEFINE

Parameter für bestimmte Import Funktionen

Bestimmte Imports benötigen eine zusätzliche Spezifikation, die mittels *Define* durchgeführt wird. Hierbei wird z.B. dem SystemTick eine Zeit in msec zugewiesen, oder dem SwitchPort eine physikalische Port Adresse. Zumindest muss hier der Prozessortakt **ProcClock** spezifiziert werden.
Die Beschreibung der einzelnen Defines sind in den ***Driver Manuals*** bei den jeweiligen Treibern zu finden.

```
Define  ProcClock    = 4000000;          {4 Mhz}
        SYSTICK      = 10;               {alle 10msec}
        StackSize    = 82, iData;        {82 Bytes in iData}
        FrameSize    = 99, xData;        {99 Bytes in xData}
        SwitchPort1  = PortA;            {Port Adr}
        SerPort      = 9600, Stop2;      {9600Bd, 2 Stopbits}
        RxBuffer     = 8, iData;         {RxBuffer 8 Chars}
        TxBuffer     = 10, iData;        {TxBuffer 10 Chars}
        PWMpresc1    = 4;                {Vorteiler 4}
```

Tip:
Die meisten Defines, die aus einer numerischen Konstante bestehen, können innerhalb des Programms auch als Konstante verwendet werden.

```
Define  ProcClock  = 4000000;          {4 Mhz }
        SYSTICK  = 10;                 {alle 10msec }
        ...

const   myConst = ProcClock div SysTick;
          ...
if ProcClock > 4000000 then ...
```

Es ist es auch möglich, im Define Block ein Define mit "Forward" zu deklarieren. Dieses Forward kann dann später im Implementationsteil des Programms oder einer Unit nachgeholt werden.
Ein Beispiel ist im Verzeichnis **..\E-Lab\AVRco\Demos\Mega161** zu finden.

4.6.6 Hardware Imports innerhalb von Units

Wenn Units importiert sind, kann eine Definition auch innerhalb einer Unit erfolgen:

Hauptprogramm

```
Import SysTick, MatrixPort, SerPort;

From System Import longword, longint, float, pipes;

Define
    ProcClock    = 8000000;    {Hertz}
    SysTick      = 10;         {msec}
    StackSize    = $0020, iData;
    FrameSize    = $0040, iData;
    SerPort      = 9600;
    RxBuffer     = 16, iData;

DefineFrom unit1;    // Unit1 definiert das Matrixport
```

Unit

```
Unit Unit1;

Define
    MatrixRow    = PortD, 4;   {use PortD, start with bit4}
    MatrixCol    = PinD, 0;    {use PinD, start with bit0}
    MatrixType   = 3, 4;       {3 Rows at PortD, 4 Columns at PinD}

Interface
...
```

Mit dem reservierten Begriff "*DefineFrom*" wird im Hauptprogramm innerhalb des "Define" der Unitnamen angegeben, von der ab jetzt die Defines eingelesen werden. Ab "*Interface*" wird wieder ins Hauptprogramm zurückgeschaltet.

4.6.7 DEFINE_USR

Mit "Define_USR" können bei Bedarf Konstante definiert werden, die auch von jeder Unit aus sichtbar und zugreifbar sind. Es sind jedoch nur ordinale Typen möglich.
Dieses Define sollte nur in "Notfällen" benutzt werden, denn es ist kein sauberer Programmierstil!
Besser ist es solche Definitionen in eine Unit zu platzieren, die sich am untersten Ende der Unit-Kette befindet. Dann sind die Definitionen auch von allen anderen Teilen her sichtbar.

```
Define  SysTick = 10;
        ...
Define_USR  myConst = 1;
            myBool= true;
```

4.6.8 DEFINE_FUSES

Optional können die Fuses für die Programmer in der Source vorgegeben werden. Die Definition muss zwischen dem "Device" und dem "Import" Statement erfolgen.
Die Fuse Namen entsprechen den Namen im Datenblatt der CPU bzw. den Namen der Programmer-Software. Leerzeichen im Original Namen müssen durch "_" ersetzt werden.

Die vier möglichen Fuse Gruppen werden durch die Begriffe
"*LockBits0, FuseBits0, FuseBits1, FuseBits2*"
angegeben.

Alle Fuses sind **low-active**, d.h. wird ein Fuse Namen angegeben, z.B. "CKSEL1", so wird dieses Bit auf **NULL** (low = aktiv) programmiert.
Nicht aufgezählte Fuses werden deshalb immer auf "1" (high = inaktiv) programmiert!

Die Anweisung
```
LockBits0 = [];
```
programmiert alle Lockbits auf "1" = inaktiv.

Die Anweisung
```
FuseBits0 = [CKSEL1];
```
programmiert das Fusebit CKSEL1 auf "0", alle anderen Bits dieser Gruppe bekommen den Wert "1" = inaktiv.

Bei der Neu-Erstellung eines Projekts werden diese Fuses automatisch in das ISPE-File für den Programmer geschrieben. Wenn das ISPE File schon existiert, wird es nicht mehr verändert.
Mit der Angabe "*OverRide_Fuses*" wird ein update in das ISPE File erzwungen. *OverRide_Fuses* muss unmittelbar nach *Define_Fuses* stehen.

Der Programmer übernimmt die Einstellungen aus dem ISPE-File in die "write Felder" für die Fuses.

Die allgemeinen Programmer Optionen wie

program Fuses
program Lockbits
program User Row (XMega)

in *Programmer Options* werden dadurch jedoch **nicht** verändert, ausgenommen dies wird hier explizit angegeben. Hier legt der Anwender fest, ob die Fuses bei jedem Programmier-Vorgang geschrieben werden sollen.

Die Angabe eines *NoteBook* ist ausschliesslich für die professionellen Stand alone Programmer von Bedeutung.
Bei extrem vielen Projekten wird bei diesen Programmern die Auswahl sehr unübersichtlich.
Deshalb wurde bei diesen ein "Tabbed Notebook" implementiert, welches die Seiten A...N hat.
Das Define *NoteBook* bestimmt, in welche Notebook Seite die Applikation abgelegt werden soll.

Mit *ProgMode* kann eine der drei möglichen Programmier Arten eingestellt werden.

Mit *Supply* wird der Programmer angewiesen eine Spannung mit einem max. Strom (Limit) auszugeben.

Normalerweise wird zur SPI Programmierung der „ProcClock“ herangezogen. Soll z.B. mit einer niedrigeren Geschwindigkeit programmiert werden, kann mit „SPIclk“ dies erzwungen werden.

Mit *CalByte* wird der Programmer angewiesen, ein Calibration Byte aus der CPU auszulesen und in einer bestimmten Speicherstelle im Flash abzulegen. Bitte beachten, dass die CPU das erste CalByte immer automatisch lädt und dieses deshalb nicht vorgegeben werden muss. Number = 0..3 CPU-abhängig.
Define_Fuses CalByte = CalByteNumber, Address;

Mit *AddApp* wird der Programmer angewiesen weitere Hex Files eines anderen Projekts zu laden:
AddApp = 'pathname\projectname';
Damit kann eine BootApplikation mit der Main Applikation im Programmer zusammen gefügt und programmiert werden. Siehe auch Demos\BootApp

***Device** = mega16, VCC = 5;*

```
Define_Fuses
  Override_Fuses;                  // optional, always replaces fuses in ISPE
  COMport    = COM1;               // COM2..COM7, USB
  Supply      = 4.0, 200;          // programmer supplies target, 4.0Volt, 200mA
  SPIclk      = 1000000;           // optional SPI programming speed
  LockBits0   = [];
  FuseBits0   = [CKSEL1];
  FuseBits1   = [BOOTRST, BOOTSZ1, SPIEN, OCDEN];
  ProgMode    = SPI;               // SPI, JTAG or OWD
  ProgFuses = true;                // or false – program Fuse Bits
  ProgLock    = true;              // or false – program Lock Bits
  ProgFlash   = true;              // or false – program Flash
  ProgEEprom= true;                // or false – program EEprom
  ProgUsrRow= true;                // or false – program UserRow          (* XMega *)
  CalByte     = 2, $3FFF;          // read/write calibration byte
  AddApp      = 'pathname\projectname';
  AutoRelease = true;              // or false – Release Target
```

***Import** SysTick, …;*

***Define** SysTick …*

4.6.9 IMPLEMENTATION

Programm Start.
Der Compiler fügt hier den Reset-Code und die Initialisierung ein. Anweisung **muss** vorhanden sein und unmittelbar dem Define-Block folgen.

Auf Implementation folgen normalerweise die globale Type-, Const- und Var-Deklarationen.

4.6.10 TYPE

Beginn einer Typen Deklaration

In Pascal kann ein Datentyp entweder direkt im Variablen Deklarationsteil beschrieben oder durch einen Typ Bezeichner referenziert werden. Es sind verschiedene Standard Bezeichner wie *boolean, byte, word, integer* etc. verfügbar. Mittels der Typen Deklaration kann der Programmierer eigene Typen erstellen.
Dem reservierten Wort **type** folgen ein oder mehrere Type Vereinbarungen, die durch Strichpunkte getrennt sind. Jede Vereinbarung besteht aus einem Typ Bezeichner, gefolgt von einem Gleichheitszeichen und einem Typ.

Das Definieren eigener Typen ermöglicht einen eleganten gut lesbaren Programmierstiel.

```
type  tpb     = pointer to byte;
      tarr    = array[2..7] of byte;
      tpa     = pointer to tarr;
      tstr    = string[8];
      tps     = pointer to tstr;
      tKey    = (Forw, Stop, BackW);

var   pb      : tpb;
      pa      : tpa;
      ps      : tps;

      str     : tstr;
      ar1     : tarr;

      keys    : tKey;

      str:= '1234';
      ar1[3]:= 56;

      pb:= tpb (pa);
      pb^:= 0;

      keys:= Stop;
```

4.6.11 CONST

Beginn einer Konstanten Deklaration

Der Konstanten Definitionsteil erlaubt Identifiers (Namen) als Synonyme für konstante Werte. Das reservierte Wort **const** leitet diesen Definitionsteil ein und wird gefolgt von einer Liste von Konstanten Zuweisungen, die mit einem Strichpunkt abgeschlossen werden müssen. Jede Konstanten Zuweisung besteht aus einem Identifier (Namen) gefolgt von einem Gleichheitszeichen und dem Konstanten Wert. Konstante können aus skalaren Typen (byte, word etc.) als auch aus Strings, Records oder Arrays bestehen.
Dabei ist zu beachten, das die skalaren Konstanten **nicht** ins ROM/FLASH abgelegt werden, im Gegensatz zu den Arrays, Strings und Records.

4.6.11.1 vordefinierte Konstante

Die folgenden Konstanten sind vordefiniert und können angesprochen werden ohne eine weitere Definition:

Name:	**Type und Wert:**
False	Boolean (der bool'sche Wert false).
True	Boolean (der bool'sche Wert true).
Pi	Float pi
Nil	Zero, auch zero value pointer.

Die beiden folgenden Konstanten haben eine **Spezialfunktion**: es wird jeweils dynamisch ein String generiert, dem das aktuelle Computer Datum bzw. Zeit zugrunde liegt:

Date speichert das aktuelles (PC-) Datum als String Konstante in das ROM
Time speichert die aktuelle (PC-) Zeit als String Konstante in das ROM

```
Const
    Date  = 'dd.mm.yy';     // -> '26.12.99'        compile Datum
    Date  = 'dd.mm.yyyy';   // -> '26.12.1999'
    Date  = 'mm/yy';        // -> '12/99'
    Time  = 'hh:mm:ss';     // -> '22:02:06'        compile Zeit
    Time  = 'hh:mm';        // -> '22:02'
    Time  = 'hhmm';         // -> '2202'
```

Compiler Version

Die Compiler Version und andere können über vordefinierte Konstante abgefragt und im Programm verwendet werden.

```
CompilerRev      : word = rev;      // actual Compiler version
CompilerBuild_Y : byte  = yy;       // actual Compiler build, last 2 digits of current year 00..99
CompilerBuild_M : byte  = mm;       // actual Compiler build, current month     01..12
CompilerBuild_D : byte  = dd;       // actual Compiler build, last current day   01..31
CompileYear      : byte  = yy;      // last 2 digits of current year 00..99
CompileMonth     : byte  = mm;      // current month  01..12
CompileDay       : byte  = dd;      // current day      01..31
CompileHour      : byte  = hh;      // current hour    00..23
CompileMinute    : byte  = mi;      // current minute 00..59
PojectBuild      : word = pbuild;   // number, incremented with each successful project compile
OptimiserRev     : word  = rev;     // number, supplied by the optimiser
OptimiserBuild   : word  = build;   // number, supplied by the optimiser
```

4.6.11.2 Typ Angabe bei Konstanten Deklaration

In Borland Pascal werden Konstanten ohne Typ Angabe definiert:
const *abc = 1;*

Dieser Wert "1" kann jetzt innerhalb des Programms als byte, word, integer, float etc. eingesetzt werden. Im Avrco Pascal ist das ähnlich. Allerdings gibt es doch manchmal Missverständnisse. Daher wurde die Konstanten Deklaration optional erweitert und sollten so deklariert werden:

const *abc : byte = 1;*

Siehe dazu auch Compiler Schalter *{$TYPEDCONST ...}*
Damit steht eindeutig fest, dass dieser Wert als Byte behandelt werden soll.

Achtung:
bei Borland bedeutet obige Konstruktion eine strukturierte Konstante. Nicht verwechseln!

4.6.11.3 Konstanten aus Dateien

Array und Record Konstante können auch aus Dateien eingelesen werden. Für den Inhalt der Datei ist der Programmierer selbst verantwortlich. Die Dateilänge spielt dabei keine Rolle. Ist die Datei zu kurz, wird mit Nullen aufgefüllt, ist sie zu lang, wird beim erreichen der Array/Record-Grenze abgebrochen

```
const  name : array[a..z] of byte = 'FileName.ext';
```

Diese Operation gilt auch für Records, wobei der Record als Type schon definiert sein muss.
Solche Datei Konstante können auch im StructConst Bereich des EEProm abgelegt werden.

Alternativ kann auch die Dateigrösse die Grösse des Array bestimmen:

```
const name : array = 'FileName.ext';
```

Das Array ist nun definiert als ***Array**[0..filesize-1] **of** byte*.

```
const FileConst : array = 'E-LAB.pbmp';
```

Es kann bei einer Konstanten aus einer Datei auch ein optionaler Typ angegeben werden, der aber ein einfacher Typ sein muss (Byte...Float). Das Array wird implementiert als:

```
const array[0..(filesize div sizeOf (type))-1] of Type.
const FileConst : array of Char = 'E-LAB.txt';
```

Für Arrays und Records können auch mehrere Files angegeben werden, die dann sequentiell eingelesen werden.

```
const ArrXYZ  : tArrXYZ = 'FN1.ext', 'FN2.ext', 'FN3.ext';
```

Die Filenamen müssen durch Kommas getrennt werden. Weiterhin gilt dass solange aus den Files gelesen wird bis die Struktur gefüllt ist. Überzählige Bytes werden ignoriert. Ist die Summe der Bytes aus den Files kleiner als die Struktur Grösse, wird der Rest der Struktur mit Nullen aufgefüllt.

Beispiel:
Ein Beispiel dazu befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\LCD_PCF8548**

4.6.11.4 Konstanten im Flash

Byte und Word Konstante können auch direkt ins Flash an feste Adressen platziert werden:

```
{$PHASE $01EFF}
ASM;
  .WORD     0AA55h
ENDASM;
{$DEPHASE}
```

Dabei wird mit dem Schalter $PHASE die absolute Adresse angegeben.
Die Adresse ist eine WORD Adresse.
Obige Adresse $01EFF ergibt dabei die absolute Byte-Adresse $03DFE.

Es ist auch möglich ordinale Array und Record Konstante auf vorgegebene Adressen ins Flash zu platzieren. Dies geschieht ähnlich wie bei den Variablen.

```
const   wwC[$1000]   : byte = 123;
        fltC[$1010]    : float = 0.5;
```

Die Adresse wird nach dem Namen in [] angegeben. Eine Typangabe (byte, word etc) ist erforderlich.
Bei Arrays und Records muss ein schon definierter Typ verwendet werden. Die Adresse wird nur geprüft ob sie innerhalb des Flash liegt, aber nicht ob sie plausibel ist. Also grösste Vorsicht!

Weiterhin gibt es die Möglichkeit dass der Compiler wie auch schon bei den Variablen die Adress Vergabe automatisch vornimmt:

```
const
   Name[@FLASH] : type = value;
```

Der Bezeichner "FLASH" bestimmt hierbei die automatische Adressvergabe.

Alle Konstanten Definitionen in der Art

```
const
  name    = 'x';
  name1  = 'xxx';
```

werden als Strings ins ROM platziert.
Um dennoch Character Literale (immediate Konstante) zu haben, die nicht im ROM liegen, müssen diese qualifiziert werden:

```
const
  name   : char  = 'x';
  name1 : char  = #12;
```

Es können auch Konstante ins ROM/FLASH platziert werden, die Pointer enthalten, die wiederum ins ROM/FLASH zeigen. Ein Beispiel finden Sie im Verzeichnis **..\E-Lab\AVRco\Demos\Mega161**.
Es ist zu beachten, dass hier nur mit vordefinierten Arrays und Records gearbeitet werden kann, die im Typ-Deklarationsteil definiert wurden.

Für Konstanten im ROM gilt grundsätzlich immer:
Taucht der Namen dieser Konstanten nicht im Kontext (Statements) auf, entfernt sie der Optimierer. Wenn also nur mit einem Pointer dessen Wert zur Laufzeit gerechnet wird, darauf zugegriffen wird, ist es sehr wahrscheinlich, dass die Konstante nicht im ROM vorhanden ist. Es kommt dadurch zu einem Assembler Fehler. Abhilfe schafft hier ein Dummy Zugriff mit Namensangabe auf das jeweilige Konstrukt.

Typen, Konstante und Variable sollten vor der ersten Prozedur oder Funktion komplett deklariert sein. Einige Berechnungen und Operatoren sind auch für die Konstanten Deklaration zulässig.

Komplexe Konstanten (Arrays und Strings) werden im Rom abgelegt (Rom Konstanten).
Für den Programmierer ist das insofern von Bedeutung, dass entsprechender Rom-Platz belegt wird und diese Werte natürlich nicht veränderbar sind (read only).

```
const  x1   = Lo (1234);
       x2   = Hi (1234);
       x3   = 24 * 2;
       x4   = x2 div 2;
       x5   = 12 mod 10;
       x6   = 1 shl 4;
       x7   = 1200 or 34;
       x8   = $FFFF and $AAAA;
       str0 = '1234' + 'R' + #7 + ^L;
       str1 = '1234';
       ch   = '9';
       TAB  = ^I;
       TB   = TAB;
       bits = 3;
       st   = 'Hallo';
       x9   = %11001101;
       arr  : array[3..7] of byte = (0, 1, 7, 34, 128);
```

4.6.12 STRUCTCONST

strukturierte Konstanten

```
ars      : array[3..7] of byte = (0, 1, 7, 34, 128);
px       : pointer to word = $40;
sc[$15] : word = $1234;                {feste Adr $15}
```

Strukturierte Konstante verhalten sich zur Laufzeit wie Variable, d.h. sie haben eine Adresse und lassen sich lesen und schreiben. Der als “konstant” vorgegebene Inhalt bzw. Wert der Konstante und deren Adresse sind im Programm (ROM) oder EEprom abgelegt. Nach der Speicher-Initialisierung werden die vorgegebenen Werte der Konstanten aus dem Programmspeicher in den Arbeitsspeicher an die entspr. Adresse geladen und können ab jetzt wie Variable behandelt werden. Da eine strukturierte Konstante auch eine Adresse hat, kann auch optional eine feste Adresse mit x[adr] : byte = bb; vorgegeben werden. Einer StructConst Deklaration sollte immer der Zielspeicherbereich als Compilerschalter vorangestellt werden, z.B. {$IDATA}.

Eine Besonderheit bilden strukturierte Konstanten im EEprom, falls im Prozessor vorhanden. Die Deklaration einer solchen Konstanten sieht so aus:

```
{$EEPROM}
Structconst
     ee1  : word   = $1234;
     est  : string = ‘hallo’;
     eflt : float  = 1.23456;
{$IDATA}
```

Alle dem Compiler-Schalter **EEPROM** folgende Strukturierte Konstanten Deklarationen erfolgen jetzt in das EEprom der CPU (wieder zurückschalten z.B. mit {$DATA}). Diese Konstanten bleiben im EEprom und werden nicht in das Ram kopiert. Der Compiler/Assembler generiert ein spezielles Hex-File mit der Endung *xxx**.eep***, in dem diese Konstanten enthalten sind.

Die Programmiertools können in der Regel ein gesondertes Hex-File für das On-Chip EEprom lesen und die CPU entsprechend programmieren. Das EEprom wird mit den zugewiesenen Werten programmiert.

Die definierten Konstanten liegen im EEprom und können wie normale Variable gelesen und geschrieben werden indem die CPU-spezifischen Zugriffsalgorithmen vom Compiler automatisch angewendet werden.

Spezial Konstruktion
Array und Record Konstante können auch aus Dateien eingelesen werden. Für den Inhalt der Datei ist der Programmierer selbst verantwortlich. Die Dateilänge spielt dabei keine Rolle. Ist die Datei zu kurz, wird mit Nullen aufgefüllt, ist sie zu lang, wird beim erreichen der Array/Record-Grenze abgebrochen.

```
Structconst  Arr1     : array[0..31] of word = 'DateiName.ext';
```

4.6.13 VAR

Beginn der Variablen Deklaration
Jede Variable, die in einem Programm benutzt wird muss zuerst definiert werden, bevor auf sie zugegriffen werden kann. Die Deklaration muss vor dem ersten ansprechen erfolgen, so dass die Variable dem Compiler bekannt ist, wenn sie jetzt im Source Text auftaucht.

Eine Variablen Deklaration besteht aus dem reservierten Wort **var** gefolgt von einem oder mehreren Identifier (Namen), getrennt durch Kommas, abgeschlossen durch einen Doppelpunkt und einer Typbezeichnung. Das erzeugt eine neue Variable von dem spezifizierten Typ und verbindet sie mit dem Identifier (Namen).

Der 'scope' (Sichtbarkeit) dieser Variablen ist der Block (function, procedure, Unit Implementation) in der sie definiert wurde, und natürlich jeder Block innerhalb dieses Blocks. Eine Variable ist immer ***local*** zu dem Block in der sie deklariert wurde. Ausserhalb dieses Blocks ist sie unsichtbar. Globale Variable sind deshalb rein formal auch lokal, nämlich lokal zur Applikation, was aber hier keine weitere Bedeutung hat. In Pascal geht die Suche des Compilers von innen nach aussen. D.h. innerhalb einer Prozedur/Funktion/Unit wird zuerst lokal gesucht. Wird hier nichts gefunden, wird global gesucht, was auch den Definitions Teil von Units mit einschliesst.

Typen, Konstante und Variable **sollten vor** der ersten Prozedur oder Funktion komplett deklariert sein. Alle Variablen benötigen entsprechenden Speicherplatz. Der jeweilige Bedarf ist weiter oben unter ***typen*** nachzulesen. Vom Programm nicht benutzte Variablen verschwenden nur unnötig den ohnehin sehr beschränkten Speicher. Es ist daher ein sorgfältiger Umgang mit diesen Deklarationen notwendig. (siehe auch Compilerschalter **$W**)

Grundsätzlich müssen alle Variablen in eine bestimmte Speicherseite (Page) der CPU gelegt werden. Die möglichen Pages sind : ***Data, iData, iData1, pData, xData, xDATA1..xDATA4*** und ***EEprom, EEprom1***, abhängig vom verwendeten Prozessortyp (Datei xxx.dsc). Hierzu wird mit dem entsprechenden Compiler-Schalter die aktuelle Page vorgeben, z.B. **{$IDATA}**. Grosschreibung beachten! Diese Speicherseite bleibt bis zur nächsten Redefinition erhalten.

Hierbei ist zu beachten, dass die zuletzt eingestellte Page auch für das Hauptprogramm und damit für einige Systemvariablen gilt. Weiterhin gilt die aktuelle Page auch für Prozess- und Task Variable. Der **Defaultwert** für die Page nach dem Begriff ‘Implementation’ ist, falls frei ‘Data’, ansonsten ‘iData’.

XMega:
Vermeiden Sie Variable in den $DATA Bereich zu platzieren. Dabei kann es zu Konflikten mit mit dem IO/IDATA Bereich kommen, der hier nicht im linearen Adress Bereich liegt wie bei den Standard AVRs. Bei den XMegas ist der Register Bereich ein komplett getrennter Speicher verglichen mit den AVRs.

Eine **globale** Variable kann mit nachgestellter Adresse auf einen festen Speicherplatz gelegt werden, was z.B. für Memory-mapped Ports und Steuerregister notwendig ist. Hierbei ist zu beachten, dass die vorgegebene Adresse auch in der aktuellen Speicherseite (Data, iData, pData, xData, EEprom) liegt. Variable ohne Adresse werden vom Compiler vom Speicheranfang beginnend angelegt, abhängig von der aktuellen Speicherseite (Page).

```
{$DATA}
Var   TrisA[$85]        : byte;              {var auf feste Adr}
      TrisB[$86]        : byte;              {var auf feste Adr}
      Count             : byte;              {normale Deklaration}
      i                 : integer;
      ix[@i]            : byte;              {Lo-byte von i}
      iy[@i + 1]        : byte;              {Hi-byte von i}
      Bit7[@TrisB, 7] : bit;                 {bit7 im TrisB Reg}
      Ar1               : array[2..9] of byte; {array}
      St1               : string[6];         {string mit Länge 6}
```

4.6.14 LOCKED

Globale ordinale Variablen können mit dem Attribut **locked** versehen werden, um sie gegen konkurrierende Zugriffe von Interrupts, Prozesse oder Tasks zu schützen. Greifen z.B. mehrer Prozesse auf eine globale Bit-Variable zu, kann es vorkommen, dass ein Prozess mit **Incl(Bit)** gerade ein read-modify-write Zugriff beginnt und nach dem Bit-read vom anderen Prozess unterbrochen wird. Dieser verändert nun das Byte das der erste Zugriff gerade gelesen hat und verändert dieses. Nachdem der zweite Prozess die Kontrolle wieder an den ersten abgegeben hat, schreibt dieser die veränderte Bitvariable (Byte) wieder zurück und überschreibt damit die Änderungen des zweiten Prozesses. Locked verhindert dies indem ein Taskwechsel oder Interrupt während eines Zugriffs verhindert wird.
Read-Modify-Write Statements sind : INC, DEC, INCL, EXCL, SETBIT.

Bei 8-Bit Prozessoren sind ggf. auch Lese und Schreibzugriffe auf globale Variablen, die grösser als ein Byte sind, ebenfalls zu schützen. **Locked** sollte nur sehr sparsam angewendet werden, da für jeden Zugriff ein Overhead (zusätzlicher Code und Maschinenzyklen) anfällt und der Interrupt gesperrt und wieder freigegeben wird.

```
{$IDATA}
Var  i  : integer, locked;      {geschützt}
     Bit9[@i, 9] : bit;         {bit9 in i}
```

Die Integer Variable **i** ist locked und geschützt sowie auch die Bit-Variable **Bit9**, die von **i** abgeleitet wurde.

Achtung:
Ein kontinuierliches Pollen/Lesen einer “locked” Variablen z.B. innerhalb einer Schleife kann zu Störungen beim Task/Prozess-Wechsel führen, weil Interrupts fast kontinuierlich gesperrt sind.

4.6.15 Align2 Align4 Align8

Manchmal ist es notwendig gewisse Variablen auf geradzahlige Adressen zu legen. Dazu kann einer solchen Variablen ein entsprechendes Attribut angehängt werden:

```
VAR
  st1   : string[16], Align2;
  bb    : byte, Align4;
  Arr   : tArr, Align8;
```

4.6.16 Lokale Variablen

Lokale Variable innerhalb einer Prozedur sind im sog. Frame angesiedelt und nur in dieser Prozedur ansprechbar, da der Frame und damit sie selbst nur temporär vorhanden sind. Lokale Variable bedingen eine indizierte Adressberechnung und einen Lese/Schreibzugriff über Pointer, was zu einem Zuwachs an Code und Rechenzeit führt.
Lokale Variablen werden per Default nicht initialisiert. Siehe dazu Compiler Schalter *{$ZeroLocVars}.*

```
Procedure LokalVars;
Var  bb : Byte;                {lokale Variable}
begin
  bb:= not bb;
end;
```

Prozess und **Task** Variable werden definiert und programmiert wie normale lokale Variable, müssen jedoch immer vorhanden sein. Sie sind also statisch und werden wie globale Variablen direkt adressiert. Trotzdem sind sie nicht global, sondern gekapselt, d.h. von ausserhalb des Prozesses nicht direkt ansprechbar. Um sie trotzdem von aussen manipulieren zu können, muss der Prozessname der Variablen vorangestellt werden,

z.B. *procName.var1:= 0;*

Bei konkurrieren Zugriffen durch andere Prozesse oder Tasks müssen auch diese Variablen das Attribut "locked" erhalten (kein automatisches Lock).

```
{ process-vars and process-stack into xData}
Process DoTheJob (20, 10 : xData);
Var  pb : Byte;                {lokale Variable, aber statisch }
begin
  bb:= not bb;
end;

{$IDATA Main-stack into iData}
begin                          {Main}
  DoTheJob.pb:= $FF;
end;
```

4.7 Prozeduren und Funktionen

Ein Pascal Programm besteht aus einem oder mehreren ***Blocks***, wobei jeder wiederum aus Blöcken bestehen kann. Ein solcher Block ist eine *Procedure,* ein anderer ist eine *function* (im allgemeinen *Unterprogramme* genannt*). Daher ist eine* Prozedur ein separates Teil eines Programms, das von irgendwoher im Programm aufgerufen wird durch ein *Procedure statement*. Eine Funktion ist sehr ähnlich, errechnet und liefert einen Wert zurück, wenn ihr Namen, oder *designator,* im Source Kontext auftaucht.

C Programmierer kennen den Typ “function” sehr gut. Eine Prozedur ist eine Funktion die kein Ergebnis zurückliefert.

Parameter

Werte werden an Prozeduren und Funktionen durch *Parameter* übergeben. Parameter bieten einen Austausch Mechanismus der es der Logik des Unterprogramms erlaubt, dass sie unterschiedliche Werte enthalten können, die wiederum ein unterschiedliches Ergebnis erzeugen.

Das Prozedur Statement oder der Funktions Designator, das das Unterprogramm aufruft, kann eine Anzahl von Parametern enthalten, die sogenannten *actual parameters.* Diese werden als *formal parameters* übergeben, wie in der Deklaration der Prozedur/Funktion Kopfes angegeben. Die Reihenfolge und Typen der Übergabe der *actual Parameter* muss der Reihen und Typen der *forma Parameter* entsprechen. Pascal unterstützt zwei unterschiedliche Methoden der Parameter Übergabe. Übergabe durch den Wert (*value)* oder durch die Adresse (*reference).* Die gewählte Methode bestimmt ob und wie Änderungen des formalen Parameters innerhalb der Prozedur/Funktion Auswirkungen auf den Übergabe Parameter (actual parameter) haben.

Wenn Parameter durch den Wert (*by value)* übergeben werden, ist der “formal parameter” wie eine lokale Variable im Unterprogramm anzusehen, und Änderungen des formalen Parameters haben keinen Einfluss auf den “actual parameter”. Der actual parameter kann ein Ausdruck, eine Variable oder Konstante sein, die vom gleichen Typ sein muss wie der zugehörige “formal parameter”. Diese Parameter werden als “*value parameter”* bezeichnet. Die folgenden Beispiele zeigen Prozedur Deklarationen.

procedure *Example (Num1, Num2 : Number; Str1, Str2 : Txt);*

Number und *Txt* sind vorher definierte Typen (z.B. *Integer* und *string[255]*), und *Num1, Num2, Str1,* und *Str2* sind die “*formalen parameter”* denen die Werte der “*actual parameter”* übergeben werden. Die Typen der formalen und der actual parameter müssen übereinstimmen.
Zu beachten ist, dass hier keine neuen Typen gebildet werden dürfen, es müssen vordefinierte Typen sein. Deshalb ist diese Konstruktion nicht erlaubt:

procedure *selectIModel (**array**[1..5] **of** Integer);*

Der gewünschte Typ muss vorher im **type** Definition block deklariert werden, oder es wird ein System-interner Typ verwendet, wie z.B. Byte oder Word.

type
*Range: **array**[1..5] **of** integer;*

procedure *Select (Model: Range);*

Wenn ein Parameter (*by reference)* übergeben wird, stellt der “formal parameter” praktisch den “actual Parameter”, d.h. beide sind ein und derselbe Parameter. Jede Änderung des Parameters im Unterprogramm ändert sofort auch den Übergabe Parameter an der aufrufenden Stell. Deshalb **muss** der Parameter eine Variable sein. Parameter übergeben durch Referenz werden auch “*variable parameter”* genannt und werden folgendermassen deklariert:

```
procedure Example (Var Num1,Num2: Number)
```

Value Parameter und Variable Parameter können beliebig in der gleichen Prozedur/Funktion gemischt werden:

```
procedure Example (Var Num1, Num2 : Number; Str1, Str2 :  Txt);
```

Num1 and *Num2* sind Variable Parameter und *Str1* und *Str2* sind Value Parameter.

Alle Adress Berechnungen erfolgen beim Prozedur Aufruf. Wenn z.B. ein Parameter ein Teil eines Arrays ist, wird die Adresse (reference Parameter) bzw. der Wert (Value Parameter) beim Aufruf errechnet.

Wenn eine grosse Daten Struktur, z.B. ein Array, als Parameter übergeben wird, ist es u.U. sinnvoller ihn als variable Parameter zu übergeben, das spart Ausführungszeit und Speicher auf dem Stack, da ja nur die Adresse des Arrays (word) übergeben wird und keine Kopie des Arrays. Ein Value Parameter benötigt immer den gleichen Speicherplatz auf dem Frame, wie die gleiche Variable im normalen Arbeitsspeicher. Ein Variablen Parameter benötigt grundsätzlich immer ein Word (Pointer).

Tip:

Erleichtern Sie sich Ihre Arbeit und erhöhen Sie die Lesbarkeit Ihrer Programme indem Sie Funktionen eindeutig kennzeichnen:

```
Function ReturnAbyte : byte;
begin
  ...
end;
```

Diese Funktion wird normalerweise so verwendet:

```
var bb   : byte;
     ...
bb:= ReturnAbyte;
```

Aus dem Statement ist nun nicht zu ersehen, ob ReturnAbyte eine Variable oder eine Funktion ist.
Wenn Sie aber schreiben

```
bb:= ReturnAbyte();
```

ist es klar, es ist eine Funktion.

Bei folgenden Konstruktionen ist die leere Klammer immer ein Muss:

```
Function ReturnApointer : pointer;
begin
  …
end;
…
bb:= ReturnApointer()^;
ReturnApointer()^:= bb;
```

Hier wird das Funktions Ergebnis (pointer) direkt benutzt, um einen Move anzustossen.

4.7.1 PROCEDURE

Prozedur Deklaration
Eine Prozedur Deklaration dient zur Definition einer Prozedur innerhalb der momentanen Applikation oder Unit. Prozeduren in Prozeduren sind nicht erlaubt. Eine Prozedur wird durch ein Prozedur Statement aufgerufen und nachdem sie durchlaufen wurde, erfolgt die weitere Programm Ausführung mit dem Statement, das dem aufrufenden Statement unmittelbar folgt.

Eine Prozedur Deklaration besteht aus dem Prozedur Kopf gefolgt von einem Block, der wiederum aus einem Deklarationsteil (Variable, Typen, Konstante) und dem Statement Teil besteht.

Der Prozedur Kopf besteht aus dem reservierten Wort **procedure** gefolgt von einem Identifier, der den Namen der Prozedur darstellt. Optional kann jetzt eine Parameter Liste wie oben beschrieben folgen.

Alle Identifiers in der Parameter Liste und der Deklarations Teil sind lokal zu dieser Prozedur. Das nennt man den *scope* eines Objekts. Ausserhalb sind diese Deklarationen unbekannt. Eine Prozedur kann auf jede Konstante, Variable, Prozedur oder Funktion zugreifen, die innerhalb oder ausserhalb dieses Blocks liegen.

Der Statement Teil beschreibt die Aktionen die ausgeführt werden sollen, wenn die Prozedur aufgerufen wird. Wenn der Prozedur Identifier (Namen) selbst innerhalb des Statement Teils der Prozedur aufgerufen wird, wird die Prozedur rekursiv ausgeführt. Dies ist nur etwas für ganz gewiefte Programmierer und nicht ungefährlich.

Prozeduren sind Unterprogramme, die mit einem *Call* aufgerufen werden. Es wird zwischen parameterlosen Prozeduren und solchen mit Parametern unterschieden.

Beispiel für eine parameterlose Prozedur:

```
Procedure Test1;
begin
  Statement ...;
  Statement ...;
end;
```

Beispiel für eine Prozedur mit Parameter:

```
Procedure Test2 (par : byte);
begin
  if par > 0 then
    ...
  else
    ...
  endif;
end;
```

Der Übergabe Parameter (Argument) ist, falls kein **var** vorangestellt, grundsätzlich eine Konstante, d.h. mit dem Parameter kann innerhalb der Prozedur gerechnet werden, er kann verändert werden, das Original jedoch (in der aufrufenden Stelle) wird nicht verändert, da der Parameter nur eine Kopie vom Original ist.

Häufig sollen die Änderungen an den formalen Parametern auch die aktuellen Parameter betreffen.
Ist ein **var** vorangestellt, so wird mit der Original-Variablen gearbeitet und nicht mit der Kopie! Der Parameter wird wie ein Pointer behandelt.

```
Procedure Test2(var par : byte);
```

Übergabe-Parameter werden über den Frame-Stack übergeben und bleiben auf dem Frame.
Das bedeutet eine grösseren Code und eine langsamere Ausführungsgeschwindigkeit als mit globalen Variablen. Ist eine Prozedur/Funktion zeitkritisch, sollte deshalb ohne Übergabe-Parameter und lokalen Variablen gearbeitet werden.

Rekursionen (z.B. Aufruf von *Test1* innerhalb der Prozedur *Test1*) sind zwar nicht verboten, können jedoch schnell zu Stackproblemen führen.

Lokale Variable in Funktionen und Prozeduren sind unter dem gleichen Gesichtspunkt zu betrachten, wie Übergabe Parameter, nämlich grösserer und langsamerer Code.

```
Procedure Test3;
var loc : boolean;
begin
  if loc then
    ...
  endif;
end;
```

Jede Prozedur ***kann*** mittels dem *Return* Statement abgebrochen werden.

```
Procedure Test1;
begin
  Statement ...;
  if (a > b) then
    Return;
  endif;
  Statement ...;
end;
```

4.7.2 PROCEDURE SYSTEM_INIT

Spezial Prozedur Deklaration

System_Init wird unmittelbar nach der Stack-Initialisierung ausgeführt. Das Userprogramm kann hier bestimmte Hardware Initialisierungen ausführen, bevor das System seine eigene vornimmt. Im Anschluss an diese Prozedur erfolgt die systeminterne Speicher-, Hardware- und strukturierte Konstanten - Initialisierung etc.

```
Procedure System_Init;
begin
  Statement ...;
  Statement ...;
end;
```

Achtung:

System_Init darf nicht vom Programm selbst aufgerufen werden!!

4.7.3 PROCEDURE SYSTEM_MCUCR_INIT

Der Mega128 erlaubt dass externe Speicher unterschiedliche WAIT States haben können und dass die beiden Adressbus Ports zum Teil auch als normale IOs verwendet werden können. Dazu muss allerdings das MCUCR Register unmittelbar nach dem PowerOn entsprechend gesetzt werden.
Dazu gibt es eine CallBack Funktion die, wenn in der Applikation vorhanden, die System interne Initialisierung komplett der Anwendung überlässt. Dazu muss in der Applikation folgende Prozedur vorhanden sein:

```
Procedure SYSTEM_MCUCR_INIT;
begin
  MCUCR:= bb;
end;
```

Da zu diesem Zeitpunkt keine weiteren Teile des Systems initialisiert sind, sind lokale Variablen und Library Funktions Aufrufe hier verboten.

4.7.4 FUNCTION

Funktion Deklaration
Eine Prozedur Deklaration besteht aus dem Prozedur Kopf gefolgt von einem Block, der wiederum aus einem Deklarationsteil (Variable, Typen, Konstante) und dem Statement Teil besteht. Der Funktions Kopf ist wie der der Prozedur mit der Ausnahme, dass im Kopf der Typ angegeben sein muss den die Funktion als Ergebnis zurückliefert. Die Deklaration des Ergebnis Typs wird durch einen abschliessenden Doppelpunkt, gefolgt von einem Typnamen gebildet.

Der Deklarations- und Statement Teil ist identisch mit dem der Prozedur.

Eine Funktions Deklaration dient zum Definieren eines Unterprogramm Teils, das ein Ergebnis berechnet und dieses and den aufrufenden Teil zurückliefert. Eine Funktion wird aufgerufen wenn ihr designator (Name) als Teil eines Ausdrucks im Kontext auftritt. Es ist auch möglich, eine Funktion wie eine Prozedur aufzurufen. Das Ergebnis wird dann verworfen.

Funktionen sind Unterprogramme, die mit einem *Call* aufgerufen werden. Eine Funktion liefert grundsätzlich ein Ergebnis zurück. Das *Return*-Statement ist hierbei ein **muss** und muss mit dem Ergebnis der Funktion versehen werden. Grundsätzlich gilt für Funktionen das gleiche wie das unter *Procedure* ausgeführte. Es wird auch hier zwischen parameterlosen Funktionen und solchen mit Parametern unterschieden.

Beispiel für eine parameterlose Funktion:

```
Function Test1 : boolean;
begin
  Statement ...;
  Statement ...;
  Return(a > b);
end;
```

Beispiel für eine Funktion mit Parameter:

```
Function Test2 (var par : byte) : byte;
begin
  if par > 0 then
    Return(0);
  else
    Return(1);
  endif;
end;
```

Beispiel für eine Funktion mit lokaler Variable:

```
Function Test3 : boolean;
var loc : boolean;
begin
  if loc then
    Return(loc);
  endif;
end;
```

Mit dem Compiler Schalter *{$NORETURNCHECK}* kann bei Funktionen das Return Statement auch entfallen. Nur für spezielle Fälle gedacht.

Das Ergebnis einer Funktion kann auch ein Array oder Record sein.
Allerdings muss das Resultat einer solchen Funktion immer die Quelle sein für eine Variable.
Andere Zuweisungen bzw. Verwendung des Resultats in einem Ausdruck ist nicht zulässig.

```
type tRec = record
              abc  : byte;
              ...
            end;

var rec  : tRec;

function GetRec : tRec;
begin
  return(rec);
end;
...
rec:= GetRec();        // legal
```

Nicht zulässig ist dabei:

```
GetRec().abc:= $12;  // illegal
xy:= GetRec().abc;   // illegal
```

4.7.5 PROCESS

Prozess Deklaration

```
Process ProcessName (StackSize, FrameSize : word; DataPage);
```

Prozesse sind eigenständige Programme innerhalb einer Applikation, die absolut unabhängig von anderen Programmteilen (z.B. Main) laufen können, d.h. Prozesse kann man nicht aufrufen wie Prozeduren oder Funktionen. Sie werden stattdessen vom einem sog. Scheduler (Zeitscheibe) periodisch aufgerufen. Wurden Prozesse importiert, läuft das Hauptprogramm (Main) ebenfalls als Prozess und hat auch eine Priorität.

Sind mehrere Prozesse in einem Programm vorhanden, erfolgt die Verarbeitung der einzelnen Prozesse quasi-parallel, d.h. von aussen betrachtet scheinen alle Prozesse/Tasks gleichzeitig zu arbeiten = **Multi-Tasking**. Damit wird eine scheinbare **Parallelverarbeitung** z.B. von Ereignissen oder Daten erreicht, obwohl sie natürlich immer sequentiell, d.h. nacheinander verarbeitet werden.

Ein Prozess läuft praktisch unendlich lange, nur unterbrochen durch Interrupts und andere Prozesse und Tasks. Das ‘begin’ und ‘end’ grenzt einen Prozess bzw. dessen Statements ein. Da Prozesse nicht wie Funktionen aufgerufen werden können, besitzen sie auch keine Übergabeparameter und kein Ergebnisse.

Beim ersten Aufruf eines Prozesses durch den Scheduler wird mit dem Statement begonnen, das unmittelbar dem 'begin' folgt. Im folgenden werden alle Statements bis zu 'end' abgearbeitet, evtl. unterbrochen durch einen **Taskwechsel** durch den Scheduler (Umschalten auf einen anderen Prozess/Task). Wird das 'end' erreicht, wird automatisch mit dem ersten Statement nach 'begin' fortgefahren. Ein Prozess läuft also kontinuierlich im 'Kreis' bzw. hat kein Ende. Der Programmierer muss dazu jedoch keine Schleife (Loop) programmieren, denn der Rücksprung an den Anfang (begin) erfolgt automatisch. Hierin liegt der wesentliche Unterschied zu einem Task. Tasks brechen bei 'end' ab und übergeben die Kontrolle an den Scheduler bzw. nächsten Prozess.

Bei der Deklaration eines jeden Prozesses muss als Argument die von diesem Prozess benötigte **Stackgrösse** und **Framegrösse** angegeben werden (10..1000 Bytes) sowie die Data Area (iData, xData etc). Die Stackgrösse ist abhängig von der Tiefe der Unterprogramm Aufrufe und der damit verbundenen Adress Pushs sowie der durch diverse Statements veranlassten Parameter Pushs. Die Framegrösse nur von den Frames (lokale und Übergabe-Parameter) der aufgerufenen Unterprogramme ab. Die exakt benötigte Anzahl von Bytes lässt sich in der Praxis kaum feststellen, da hier mit einem Debugger alle Eventualitäten getraced werden müssen. Wie bei allen Stackdefinitionen kann eigentlich nur nach Gefühl und Erfahrung gearbeitet werden. Dabei ist nicht die Grösse (Statement Zahl) des Prozesses massgebend, sondern die Art der Statements (z.B. Floating Point oder mehrfache gekellerte Unterprogramm Aufrufe). Steht genügend Ram zur Verfügung geht man nach der Devise 'mehr ist besser' vor. Ein guter Wert für nicht allzu komplexe Prozesse ist 32 Bytes für den Stack und 16 Bytes für den Frame.

Jeder Prozess braucht ca. 20 Bytes Speicher, in denen während einer Unterbrechung durch den Scheduler die **Pseudo-Accus** (Arbeitsregister), Stackpointer und Flags des Prozesses abgelegt werden.

Lokale Variable innerhalb Prozesse und Tasks sind zwar gekapselt, d.h. von ausserhalb nicht direkt zugreifbar, werden aber im normalen Speicher gehalten wie globale statische Variable. Lokale Variable müssen jederzeit vorhanden sein und zugreifbar sein, egal ob der Prozess aktiv ist, schläft oder suspendiert ist. Sie belegen deshalb auch ganz normal Speicher im Ram und müssen bei der Speicherbedarfs Kalkulation miteinbezogen werden. Der Zugriff innerhalb des Prozesses oder Tasks erfolgt mit dem deklarierten Namen. Von ausserhalb (andere Prozesse, Main etc) kann ein Zugriff über '*ProzessNamen.VarNamen*' erfolgen.

Die o.a. Speicherbereiche (Register-Sicherung, Stack, Frame und Lokale Variable) liegen in der Speicherseite, die durch die Prozess-Deklaration ($IDATA, $XDATA etc) festgelegt wurde.

Da das **Hauptprogramm** (Main), falls Prozesse/Tasks importiert wurden, auch als Prozess behandelt wird, muss hier mit einem Compiler-Schalter die gewünschte Speicherseite unmittelbar vor dem **begin** von Main eingestellt werden.

```
Process DoTheJob (32, 16 : iData); {Stacksize = 32 bytes, Framesize = 16 bytes}
var px : integer;
begin
  Statement ...;
  Statement ...;
end;
```

Die Arbeitsweise eines Prozesses wird durch eine Vielzahl von zugehörigen Funktionen und Prozeduren gesteuert. Ein wesentlicher Parameter stellt dabei Priority dar.

Mit **Priority** (Wichtigkeit) wird einem Prozess einen bestimmten Anteil der zur Verfügung stehenden Rechenzeit zur Verfügung gestellt. Je höher der Wert von Priority ist, desto mehr Rechenzeit steht zur Verfügung. Gleichzeitig legt Priority die Anzahl der SystemTicks fest, die der Prozess 'am Stück' zu Verfügung hat. Die anteilige Rechenzeit an der Gesamtzeit in '%' errechnet sich aus:
Priority / Summe aller Prioritäten.

Angenommen, es gibt nur den Prozess 'DoTheJob' und dieser hat die Priorität 10 und Main_Priority ist 5 dann gilt: Rechenzeit = 10 / (5 + 10) = 66%. Die exakte Rechenzeit lässt sich jedoch nur festlegen, wenn kein Prozess suspendiert oder Locked wird und keine ProcessWaits etc. vorhanden sind. In der Praxis kann die anteilige Rechenzeit nur überschlagsmässig errechnet werden.

Ein Prozess kann mit ***Lock*** die CPU voll an sich binden, so dass ausser ihm selbst nur noch Interrupts laufen. Mit einem ***Unlock*** wird dieser Zustand wieder aufgehoben.

Stellt ein Prozess fest, dass er z.Zt. nichts zu tun hat, sollte jetzt nicht unnötig Rechenzeit durch Warteschleifen oder Delays verbraucht werden. Es bestehen mehrere Möglichkeiten, die Kontrolle an andere Prozesse abzugeben:

Mit ***Schedule*** bricht der Prozess sofort ab, wird aber wieder in die Prozess-Warteschlange eingereiht.

Mit ***Sleep*** kann sich ein Prozess für eine bestimmte Anzahl von Systemticks abschalten.

Mit ***Suspend*** schaltet sich ein Prozess ab. Er kann sich selbst nie wieder aktiv schalten. Das muss von ausserhalb durch eine anderen Prozess/Task oder das Hauptprogramm mit **Resume** erfolgen.

Da die Kommunikation zwischen den Prozessen u.a. über Pipes oder Semaphoren erfolgen kann, besteht die Möglichkeit dass der Prozess sich abschaltet, indem er ***WaitSema*** oder ***WaitPipe*** aufruft. Der Prozess wird erst wieder aktiv, wenn in der spezifizierten Semaphore oder Pipe ein Datum vorhanden ist. Als Pipe kann hier auch **RxBuffer** spezifiziert sein.

Der Prozess wird unmittelbar im Anschluss an einer der o.a. Anweisungen abgebrochen.

4.7.5.1 Optionen bei Definition

Es ist es auch möglich schon bei der Definition von Prozessen als Option eine Priorität und/oder Suspend oder Resume vorzugeben:

Process *Name (StackSize, FrameSize : MemoryArea[; Priority, RunMode]);*

```
Process Proc1 (32, 32 : iData);                 // default prio=3, autostart
Process Proc1 (32, 32 : iData; 5);              // priority 5
Process Proc1 (32, 32 : iData; resumed);        // default prio=3, automatic start
Process Proc1 (32, 32 : iData; 5, suspended);   // prio= 5, no automatic start
```

4.7.6 TASK

Task Deklaration

Task *TaskName (DataPage);*

Tasks sind eigenständige Programme innerhalb einer Applikation, die absolut unabhängig von anderen Programmteilen (z.B. Main) laufen können, d.h. Tasks kann man nicht aufrufen wie Prozeduren oder Funktionen. Sie werden stattdessen vom einem sog. Scheduler (Zeitscheibe) zyklisch aufgerufen. Wurden Tasks importiert, läuft das Hauptprogramm (Main) als Prozess. Tasks sind **extrem spezialisierte Prozesse** und sollten nur für bestimmte Aufgaben verwendet werden, wie z.B. PID-Regler.

Sind mehrere Tasks/Prozesse in einem Programm vorhanden, erfolgt die Verarbeitung der einzelnen Tasks/Prozesse quasi-parallel, d.h. von aussen betrachtet scheinen alle Prozesse/Tasks gleichzeitig zu arbeiten = **Multi-Tasking**. Damit wird eine **Parallelverarbeitung** z.B. von Ereignissen oder Daten erreicht.

Ein Task läuft praktisch unendlich lange, nur unterbrochen durch Interrupts und andere Prozesse und Tasks. Das ‘begin’ und ‘end’ grenzt einen Task bzw. dessen Statements ein. Da Tasks nicht wie Funktionen aufgerufen werden können, besitzen sie auch keine Übergabeparameter und kein Ergebnisse.

Bei **jedem** Aufruf eines Tasks durch den Scheduler wird mit dem Statement begonnen, das unmittelbar dem 'begin' folgt. Im folgenden werden alle Statements bis zu 'end' abgearbeitet. Wird das 'end' nicht innerhalb eines SystemTicks erreicht, wird der **Task abgebrochen** durch einen **Taskwechsel** durch den Scheduler (Umschalten auf einen anderen Prozess/Task). Der Task erreicht also nie das 'end', wenn seine benötigte Rechenzeit von 'begin' bis 'end' grösser ist als ein SystemTick. Die Rechenzeit eines Tasks darf **nie** grösser sein als ein SystemTick. Ähnliche Bedingungen gelten übrigens auch für Interrupts. Eine Timer-Interrupt-Service Routine z.B. sollte auch nie mehr Zeit brauchen, als der Zeitraum zwischen zwei Interrupts.

Wird das 'end' erreicht, wird automatisch die Kontrolle an den Scheduler übergeben, der jetzt den nächsten Prozess oder Task aktiviert. Im Gegensatz zu einem Prozess läuft ein Task bei jedem Aufruf durch den Scheduler von 'begin' bis 'end' und bricht dann ab.

Wurden Tasks importiert, muss auch ein sog. ***TaskStack*** und ein ***TaskFrame*** definiert werden. Alle Tasks im System benutzen den gleichen Stack und Frame, d.h. dieser Stack bzw. Frame ist nur einmal vorhanden, da Tasks normalerweise nicht unterbrochen werden und an der gleichen Stelle (Statement) ihre Arbeit wieder aufnehmen müssen. Die benötigte **Stackgrösse** wird durch den Bedarf des grössten Tasks bestimmt (7..255 Bytes). Die Stackgrösse ist abhängig von der Tiefe der Unterprogramm Aufrufe und der damit verbundenen Adress Pushs sowie der durch diverse Statements veranlassten Parameter Pushs. Die exakt benötigte Anzahl von Bytes lässt sich in der Praxis kaum feststellen, da hier mit einem Debugger alle Eventualitäten getraced werden müssen.

Wie bei allen Stackdefinitionen kann eigentlich nur nach Gefühl und Erfahrung gearbeitet werden. Dabei ist nicht die Grösse (Statement Zahl) des Tasks massgebend, sondern die Art der Statements (z.B. Floating Point oder mehrfache gekellerte Unterprogramm Aufrufe). Steht genügend Ram zur Verfügung geht man nach der Devise 'mehr ist besser' vor. Ein guter Wert für nicht allzu komplexe Tasks ist 32 Bytes.

Jeder Task braucht ca. 20 Bytes Speicher, in denen während einer Unterbrechung durch den Scheduler die **Pseudo-Accus** (Arbeitsregister), Stackpointer und Flags des Prozesses abgelegt werden. Auch dieser Speicher ist für alle Tasks gemeinsam, also nur einmal vorhanden.

Lokale Variable innerhalb Prozesse und Tasks sind zwar gekapselt, d.h. von ausserhalb nicht direkt zugreifbar, werden aber im normalen Speicher gehalten wie globale statische Variable. Lokale Variable müssen jederzeit vorhanden sein und zugreifbar sein, egal ob der Task aktiv ist, schläft oder suspendiert ist. Sie belegen deshalb auch ganz normal Speicher im Ram und müssen bei der Speicherbedarfs Kalkulation miteinbezogen werden. Der Zugriff innerhalb des Prozesses oder Tasks erfolgt mit dem deklarierten Namen. Von ausserhalb (andere Prozesse, Main etc) kann ein Zugriff über 'Task*Name.VarName*' erfolgen.

Die o.a. Speicherbereiche (Register-Sicherung, Frame und Stack) liegen in der Speicherseite, die durch Define TaskStack = size, RAMpage (iData, xData etc) und Define TaskFrame = size festgelegt wurde.
Die 'lokalen' Variablen eines Tasks liegen in der Speicherseite, die durch die Definition des Tasks angegeben wurde.

```
Task RunPid (xData);
var tx : integer;
begin
  Statement ...;
  Statement ...;
end;
```

Die Arbeitsweise eines Tasks wird durch eine Vielzahl von zugehörigen Funktionen und Prozeduren gesteuert. Ein wesentlicher Parameter stellt dabei Priority dar.

Im Gegensatz zu einem Prozess wird mit **Priority** das Aufruf-Intervall des Tasks festgelegt. Je niedriger der Wert von Priority ist, desto häufiger der Aufruf des Tasks. Angenommen der Task 'RunPid' hat die Priorität 10 so wird er alle 10 SysTicks aufgerufen. Damit ist sichergestellt, dass der zeitliche Abstand zwischen zwei Aufrufen immer 10 Ticks beträgt.

Bemerkung:
Sind mehrere Tasks vorhanden und können davon mehr als einer aktiv sein, ist unbedingt darauf zu achten, dass die einzelnen Prioritäten **einen gemeinsamen Nenner** besitzen.
D.h. die Prioritäten müssen ein vielfaches von z.B. 2 sein. Wird diese Bedingung nicht erfüllt, so kommt es zu unregelmässigen Aufruf-Intervallen, d.h. der Abstand zwischen zwei Aufrufen ist nicht mehr konstant.
Weiterhin muss die kleinste Priorität grösser sein als die Summe aller Tasks.

Stellt ein Task fest, dass er z.Zt. nichts zu tun hat, sollte jetzt nicht unnötig Rechenzeit durch Warteschleifen oder Delays verbraucht werden. Es bestehen mehrere Möglichkeiten, die Kontrolle an andere Prozesse abzugeben:

Mit ***Schedule*** bricht der Task sofort ab, wird aber wieder in die Prozess-Warteschlange eingereiht.

Mit ***Sleep*** kann sich ein Task für eine bestimmte Anzahl von Systemticks abschalten.

Mit ***Suspend*** schaltet sich ein Task ab. Er kann sich selbst nie wieder aktiv schalten. Das muss von ausserhalb durch eine anderen Prozess/Task oder das Hauptprogramm mit ***Resume*** erfolgen.

Der Task wird unmittelbar im Anschluss an einer der o.a. Anweisungen abgebrochen.

Durch ***lock*** kann ein Task die CPU vollständig an sich binden, sodass ausser ihm nur noch Interrupts laufen. Dieser Zustand wird durch ***unlock*** aufgehoben.

Da die Kommunikation zwischen den Tasks/Prozessen u.a. über Pipes oder Semaphoren erfolgen kann, besteht die Möglichkeit dass der Task sich abschaltet, indem er ***WaitSema*** oder ***WaitPipe*** aufruft. Der Task wird erst wieder aktiv, wenn in der spezifizierten Semaphore oder Pipe ein Datum vorhanden ist. Als Pipe kann hier auch **RxBuffer** spezifiziert sein.

4.7.6.1 Optionen bei Definition

Es ist es auch möglich schon bei der Definition von Tasks als Option eine Priorität und/oder Suspend oder Resume vorzugeben:

Task *Name (MemoryArea[; Priority, RunMode]);*

```
Task Task1 (iData);                 // default prio=5, autostart
Task Task1 (iData, 8);              // priority 8
Task Task1 (iData, resumed);        // default prio=5, automatic start
Task Task1 (iData, 8, suspended);   // prio= 8, no automatic start
```

4.7.7 FORWARD

Vorwärtsreferenz von Pointern, Funktionen, Prozeduren, Tasks und Prozessen

Wie bei anderen Programmiersprachen und teilweise auch Assemblern ist eine sog, Vorwärtsreferenz in Pascal eigentlich nicht möglich. D.h. alle in einem Statement angesprochenen Elemente (Typen, Variablen, Konstanten, Prozeduren, Funktionen) müssen zuerst deklariert werden, bevor auf sie zugegriffen werden kann.

Manchmal ist es jedoch unumgänglich, dass eine Prozedur/Funktion/Prozess oder auch ein Pointer Typ auch schon angesprochen werden muss, bevor sie deklariert wurde. Dazu dient *Forward.* Der Prozedur/ Funktions/Task/Prozess Kopf wird nach den Variablen/Konstanten Deklarationen und vor den ersten Funktions/Prozedur Deklarationen noch einmal geschrieben mit dem Zusatz 'Forward'. Die eigentliche Deklaration kann jetzt zu einem beliebigen späteren Zeitpunkt erfolgen. Dabei darf dann jedoch *Forward* nicht mehr erscheinen!

```
Program Abc;
...
var   ...;
const ...;

Procedure Test1; Forward;
Process   Proc1(32); Forward;
Function  Test2(par : byte) : byte; Forward;
...
...
Process Proc1(32);
begin
  ...
end;

Procedure Test1;
begin
  Statement ...;
  if (a > b) then
    Return;
  endif;
  Statement ...;
end;

Function Test2 (par : byte) : byte;
begin
  if par > 0 then
    Return(0);
  else
    Return(1);
  endif;
end;
```

Forward kann auch zur vorwärts Deklaration von Typen, speziell Pointern eingesetzt werden.

```
Type  TRec1      = record; forward;     // preliminary declaration
      TPtr       = pointer to Rec1;
      TRec1      = record
                     Ptr1 : TPtr;
                   end;
```

4.7.8 BEGIN

Start Prozedur-, Funktions-, Process- oder Task Rumpf

Jede Prozedur, Funktion, Prozess, Task und auch das *HauptProgramm* **muss** mit einem ***begin*** beginnen und mit einem ***end*;** enden, wie in o.a. Beispielen ersichtlich. Pascal Programmierern ist sicherlich schon aufgefallen, dass nach *then* und *else* grundsätzlich **kein** *begin* und auch kein *end* folgt.

Der Konstrukteur von Pascal, N.Wirth, war hier nicht konsequent, und die Folge ist, dass begin/end je nachdem ein *muss* oder *kann* ist. Eindeutig ist die Inkonsequenz beim *Case..Else* Statement. Jeder Programmierer ist da häufig am Überlegen, *wie war das da nochmal?.* Im Nachfolger von Pascal, *Modula-2,* hat Wirth Klarheit geschaffen und das Begin/End in diesen Konstruktionen abgeschafft. Dies hat den Compilerbau an dieser Stelle auch erheblich einfacher und damit sicherer gemacht.

Diese Modula-2 Vereinbarung (und auch andere) wurden im vorliegend Compiler übernommen. Der Einwand: “Das ist aber nicht mehr Pascal-kompatibel” kann bei einem System, das so viele Hardware Abhängigkeiten kennt, wie ein “Embedded” Entwicklungssystem, ignoriert werden, denn, um komfortabel entwickeln zu können, müssen jede Menge Spracherweiterungen eingebaut werden.

Wer Borland-Pascal oder Delphi kennt, weiss, dass gut 50% der Konstruktionen hier auch nicht Standard Pascal kompatibel sind.

Anweisungen

Der Anweisungs-Teil ist der letzte Teil eines Blocks. Er spezifiziert die Aktionen, die vom Programm ausgeführt werden sollen. Der Anweisungs-Teil hat die Form einer Verbund Anweisung, von einem Punkt oder Semikolon gefolgt.
Eine Verbund Anweisung besteht aus dem reservierten Wort ***begin*,** gefolgt von durch von Semikolon getrennte Anweisungen und beendet durch das reservierte Wort ***end*.**

Dem abschließenden ***end*** eines Programms folgt ein “**.**” (Punkt) in traditionellem Pascal.

4.7.9 RETURN

Abbruch und Exit innerhalb Prozedur/Funktion

Das *Return* Statement dient zum Abbruch (Rückkehr) einer Prozedur oder Funktion an jeder beliebigen Stelle. Return ist ein **muss** für Funktionen. Mit dem Compiler Schalter *{$NORETURNCHECK}* kann bei Funktionen das Return Statement auch entfallen.

Jede Prozedur ***kann*** mittels dem *Return* Statement abgebrochen werden. Dabei ist das Return-Statement parameterlos.

```
Procedure Test1;
begin
  Statement ...;
  if (a > b) then
    Return;
  endif;
  Statement ...;
end;
```

Jede Funktion **muss** mittels dem *Return* Statement abgebrochen werden. Return **muss** ein Parameter haben. Der Typ dieses Parameters wird im Funktionskopf angegeben. Zulässige Type sind 8,16,32 und 64bit Werte.

```
Function Test2 : boolean;
```

```
begin
  Statement ...;
  Return(a > b);
  Statement ...;
end;
```

4.7.10 END

Ende Prozedur-, Funktions-, Task- oder Prozess-Rumpf

Jede Prozedur, Funktion, Task und Prozess **muss** mit einem ***begin*** beginnen und mit einem ***end;*** enden, wie in o.a. Beispielen ersichtlich. Das end-Statement muss mit einem Strichpunkt abgeschlossen werden. Siehe auch Beschreibung von *Begin.* Das *End*-Statement ist, im Gegensatz zu Standard-Pascal nur zum Abschluss von Funktionen, Prozeduren und dem Hauptprogramm zulässig. IF, WHILE, REPEAT etc. besitzen ihren eigenen qualifizierten Abschluss, z.B. ENDIF etc. Dies trägt zur besseren Lesbarkeit der Source bei.

Optional kann vor dem Strichpunkt noch der Name der Prozedur/Funktion zur besseren Lesbarkeit eingefügt werden (***end*** *Test2;*)

4.7.11 ASM:

Ein einzelnes Assembler Statement innerhalb der Pascal Source.

```
ASM:  PUSH _ACCA;
```

4.7.12 ASM;

Beginn eines Assembler Text Blocks

Ein Programm für Embedded Control kommt sehr oft ohne Assembler Code nicht aus, da entweder der vom Compiler generierte Code für manche Operationen zu langsam ist, oder bestimmte Assemblerbefehle ausgeführt werden müssen, die der Compiler nicht kennt oder benutzt.
Es besteht deshalb die Möglichkeit direkt Assembler Source an fast jeder beliebigen Stelle der Pascal source einzubinden. Diese source wird vom Compiler ungeprüft und unbearbeitet an den Assembler weitergegeben. Da der Compiler ebenfalls Assembler Code generiert, fügt sich der Assembler Text nahtlos ein.
Asm-Syntaxfehler werden deshalb nur vom Assembler erkannt und nicht vom Compiler.
Innerhalb des Assembler Textes kann auf alle deklarierten Variablen zugegriffen werden.

Achtung:
Labels innerhalb eines Assembler-Blocks müssen am Zeilenanfang beginnen und mit einem '**:**' abgeschlossen werden. Ausserdem darf diese Zeile keine weiteren Anweisungen wie z.B. Code enthalten. Die Analyse der vom Compiler generierten Assembler Dateien 'xxx.ASM' kann hier weiterhelfen:

```
ASM;
  LDI     _ACCA, 040h
  ANDI    _ACCA, myProg.a;          {a = Pascal var in myProg }
ENDASM;
```

4.7.13 ENDASM

Ende eines Assembler Textes

ASM und ENDASM sind Pascal Statements und müssen deshalb mit einem Strichpunkt abgeschlossen werden **;**

4.8 INTERRUPTs, TRAPs, EXCEPTIONs

4.8.1 Interrupt

Deklaration einer Interrupt Prozedur.

Mögliche Interrupt Quellen sind absolut Prozessor abhängig. Auch innerhalb einer CPU-Familie ergeben sich zudem erhebliche Unterschiede. Interruptquellen müssen deshalb im Prozessor-Beschreibungsfile (xxx.dsc) deklariert werden.
Diese Prozedur erstellt lediglich einen Eintrag in die Interrupt-Vektor-Tabelle und einen Programmrahmen für die gewünschte Register-Sicherung (s.u.).

Die jeweiligen CPU und I/O-Steuerregister für den spezifischen Interrupt müssen von der Applikation zusäzlich (wie im Controller Manual beschrieben) initialisiert werden.

Beispiel (AVR Mega103):

TIMER0	timer0 overflow int
TIMER0COMP	compare match int timer0
TIMER1	timer1 overflow int
TIMER1COMP	compare match int timer 1
TIMER1COMPA	compare match "a" int timer 1
TIMER1COMPB	compare match "b" int timer 1
TIMER1CAPT	capture event timer 1
TIMER2	timer2 overflow int
TIMER2COMP	compare match int timer2
EERDY	eeprom ready
ACOMP	analog comparator
SPIRDY	SPI serial transfer complete
ADCRDY	ADC conversion complete
INT0	External Interrupt 0
INT1	External Interrupt 1
INT2	External Interrupt 2
INT3	External Interrupt 3
INT4	External Interrupt 4
INT5	External Interrupt 5
INT6	External Interrupt 6
INT7	External Interrupt 7
RXRDY	uart1 rx complete
UDRE	uart1 data register empty
TXRDY	uart1 tx complete
RXRDY2	uart2 rx complete
UDRE2	uart2 data register empty
TXRDY2	uart2 tx complete

Die Deklaration eines Interrupts generiert automatisch entsprechende Einträge in die Interrupt-Vektor Tabelle sowie einen speziellen Code-Rahmen um die Interrupt Prozedur.
Hier werden per Default alle Prozessor Register gesichert -> **Komplettsicherung**.
Dieses Verhalten kann mit den Compiler Schaltern *{$NOSAVE}*, *{$NOREGSAVE}* und *{$NOSHADOW}* gesteuert werden.

Das von der CPU automatisch zurückgesetzte Global-Interrupt-Enable-Flag wird nicht verändert, d.h. die Interrupts bleiben während der Laufzeit der Interrupt-Prozedur gesperrt. Beim Erreichen des ‘end’-Statement

wird ein RETI (Return from Interrupt) ausgeführt, das den Interrupt wieder freigibt. Das bedeutet, dass eine Interrupt Prozedur **möglichst kurz** sein sollte, um andere Interrupts nicht allzu lange zu sperren.
Eine **Komplettsicherung** benötigt ca. 20 Bytes im RAM, und zwar statisch, nicht auf dem Stack oder Frame. Bei dieser Art der Sicherung ist ein verschachtelter Interrupt nicht möglich, d.h. innerhalb einer Interrupt Prozedur darf niemals der Interrupt freigegeben werden.
Dies führt die CPU selbst aus, wenn sie den *RETI* Befehl abarbeitet.

Da beim **XMega** der globale Interrupt beim Eintritt in eine Service Routine nicht gesperrt wird erfolgt das durch das System selbst, ebenso die Freigabe. Damit ist es möglich höher priorisierte Interrupts zuzulassen. Dazu darf dann allerdings der globale Interrupt nicht gesperrt sein. Dies kann durch das

```
Define Interruptible_Ints = true;
```

in der Define section veranlasst werden. Damit können höher priorisierte Interrupts innerhalb des aktuellen Interrupts zum Zuge kommen. System-interne Interrupts laufen grundsätzlich mit der Priorität von 2.

Ist eine Interrupt Verschachtelung unumgänglich, muss der Compiler Schalter *{$NOSAVE}* eingesetzt werden. Damit werden nur die Flags und die Haupt Arbeitsregister gesichert. Die Prozeduren *PushAllRegs* und *PopAllRegs* ermöglichen dann ggf. die Sicherung der restlichen Register über den Stack.

Die Verschachtelung von Interrupts ist höchst gefährlich und endet sehr leicht in einer Katastrophe. Derartige Konstruktionen sollten möglichst vermieden werden. Sind sie jedoch unabdingbar, ist eine extrem sorgfältige Planung notwendig.

Achtung:
die korrekte Initialisierung von Interrupts erfordert üblicherweise Einstellungen in verschiedenen Control- und Mask-Registern. Bei mitgelieferten Treibern, die im Interrupt laufen wird dies von den entsprechenden Treibern erledigt. Werden die Interrupts von der Applikation definiert, kann der Compiler dazu jedoch keine Unterstützung bieten.

Es ist dann also Aufgabe der Applikation für diese notwendigen Initialisierungen zu sorgen !!!

Ist die Systemantwort auf einen Interrupt nicht allzu zeitkritisch und muss eine grössere Anzahl von Statements dafür abgearbeitet werden, empfiehlt sich folgende Vorgehensweise:

der Interrupt inkrementiert eine Semaphore und verabschiedet sich. Ein Prozess wartet immer auf diese Semaphore und wird in kurzem die Kontrolle erhalten.

```
Interrupt Int0;
begin
  IncSema (sema0);
end;

Process ProcessInt0 (32, 16 : iData);
begin
  WaitSema (sema0);          {wartet auf sema0 > 0 }
  ...
  ...
end;
```

Interrupt Service Routinen können auch lokale Variablen besitzen. Da hier jedoch kein Frame eingesetzt werden kann, sind diese static/nonvolatil. D.h. sie werden im iData Bereich angelegt und sind damit immer zugreifbar (static). Auch verlieren sie ihren Inhalt nicht (nonvolatil).
Innerhalb der Interrupt Prozedur können sie wie jede andere Variable angesprochen und benutzt werden. Da sie statisch sind, können sie auch von anderen Programmteilen benutzt werden. Weil sie jedoch im "Besitz" einer Funktion sind, müssen sie allerdings auch entsprechend qualifiziert werden:

```
Interrupt Timer1;
var abc : byte;
begin
  abc:= 123;                 // no qualification
  ...
end;
...
```

```
Interrupt_Timer1.abc:= $67;  // qualify with "Interrupt_name"
```

Zu beachten ist hierbei, dass immer "Interrupt_" vor dem Qualifier stehen muss.

Standardmässig werden alle Interrupts, die nicht durch einen Handler bedient werden, in einer Default/ Dummy Interrupt Routine durch ein simples "RETI" abgefangen. Für Debug und Testzwecke kann dieser Interrupt-Error jetzt auch in die Applikation eingebaut werden. Im Normalfall sollte es nie vorkommen, dass diese Routine durch Interrupts angesprungen wird. Die Implementation erfolgt in der Applikation mit:

```
Interrupt IntErrorHandler;
begin
  ...
end;
```

4.8.1.1 Push, Pop

```
Procedure Push (regnum : byte  |  regname : internal);      //z.B. Push (24);
Procedure Pop (regnum : byte  |  regname : internal);       //z.B. Pop (_ACCFLO);
```

Zur besseren Lesbarkeit. Erzeugen den gleichen Code wie
ASM: PUSH ... ; bzw.
ASM: POP … ;

4.8.1.2 PushRegs, PopRegs

```
Procedure PushRegs;       // working registers to stack
Procedure PopRegs;        // working registers from stack
```

Diese Prozeduren sind eine vereinfachte Version von PushAllRegs und PopAllRegs und kann jederzeit paarweise in Interrupts eingesetzt werden wenn in Interrupts nur die 4 wesentlichen Register (ACCA, ACCB etc) gesichert wurden.

4.8.1.3 PushAllRegs, PopAllRegs

Manchmal, aber auch nur manchmal, ist es sinnvoll innerhalb eines Interrupts z.B. Timer Interrupt, den globalen Interrupt wieder freizugeben um eine zu lange Interrupt Sperrzeit zu vermeiden. Wenn hierbei alle Register gesichert werden müssen, dann kann dies nicht mit sonst wirksamen automatischen Sicherung erfolgen (Switch $NOSHADOW inaktiv). Diese Register Sicherung erlaubt keinerlei "nested interrupts".

Man kann eine spezielle Register Sicherung erzwingen, die es erlaubt innerhalb dieser Interrupt Prozedur den Interrupt wieder freizugeben:

```
{$NoSave}
Interrupt TIMER1COMPA; // TickTimer
begin
  PushAllRegs;
  EnableInts;
  ...
  PopAllRegs;
end;
```

Der Schalter {$NoSave} ist in diesem Fall zwingend vorgeschrieben.
Diese Methode sollte nur verwendet werden, wenn die allgemeinen Interrupts in diesem Interrupt unbedingt freigegeben werden müssen. Man sollte hier genau wissen was man tut :-)

4.8.2 Externe Interrupts

Die AVR Familie kennt prinzipiell zwei externe Interrupt Typen, die über Port Pins gesteuert werden.

Die erste Gruppe ist beinhaltet diese Pins/Interrupts, die jeder für sich einen eigenen Vektor Interrupt auslösen können. Dies sind die Interrupts INT0..INT7 und die zugehörigen Port Pins.

Die zweite Gruppe betrifft die sog. **P**in**C**hange**I**nterrupts PCINT0..PCINTxx. Hier werden immer bis zu 8 Pins eines Ports zusammengefasst und haben einen gemeinsamen Interrupt Vektor.

4.8.2.1 Interrupt Pins INT0..INTx

Diese Standard Externen Interrupts werden vom AVRco System insoweit unterstützt dass, wenn eine entspr. Interrupt Prozedur im Programm definiert ist, z.B. ***Interrupt INT0***, das System die Adresse dieser Prozedur in die Interrupt Vektor Tabelle einträgt und im Interrupt Fall die gewählte Register Sicherung vornimmt, bevor die Prozedur angesprungen wird. Die Initialisierung des Interrupts in den zugehörigen Enable und Mask Registern muss die Applikation jedoch selbst vornehmen.

4.8.2.2 PinChangeInterrupts PCINT0..PCINT3

Neuere AVR CPUs besitzen meistens diesen Interrupt Mechanismus. Das können bis zu 4 Ports (PCINT0..3) sein bzw. 32 Interrupt Pins. Das sieht sehr komfortabel aus, ist es aber nur mit Einschränkungen. Denn es gibt pro Port bzw. 8 Bits nur ein Vektor und jede Status Änderung an so einem Pin löst einen Interrupt aus, also die Low/High Flanke einen und die High/Low Flanke einen.

Für viele Anwendungen reicht das auch aus. Oft ist es jedoch erforderlich dass die hier benutzten Port Pins auch gezielt eine zugehörige Interrupt Prozedur anspringen so dass jedem PinChange ein spezieller Interrupt zugeordnet ist. Das AVRco System unterstützt dies mit dem Spezial Interrupt Handler ***PCintServer*** für die PCINT Interrupts.

Dazu wird der alte Port bzw. PIN Status mit dem neuen verglichen um zu sehen, welcher Pin diesen Interrupt ausgelöst hat. Ist dazu eine vom User spezifizierte Interrupt Service Routine vorhanden, so wird diese angesprungen und im Register R16 (_ACCB) ein true oder False ($00/$FF) übergeben, je nachdem welche Flanke diesen Interrupt ausgelöst hat. Die Interrupt Prozeduren müssen bestimmte Namen haben, PCINT00 bis PCINT31.
Der Interrupt Handler *PCintServer* für die PCINT Interrupts muss importiert werden.

```
From System Import PCINTserv0, PCINTserv1, …;
```

Um jetzt die einzelnen Pin Change Interrupts nutzen zu können, müssen auch noch die gewünschten Interrupt Prozeduren definiert werden.

```
Interrupt PCint00;   // PinB.0 mega168
begin
  if _ACCB <> 0 then
    …
  else
    …
  endif;
end;
```

PCINTxx wird auch vom Application Wizard und vom Simulator unterstützt.
Ein Beispiel Programm ist in der Demos Directory unter PCintServ zu finden.

4.8.3 Externe Interrupts XMega

Die XMega Familie bietet Interrupts für jeden einzelnen IO-Port Pin an. Da jedes Port aber nur zwei Interrupt Vektoren hat macht es Sinn hier zweigleisig zu fahren. Werden mehr als ein Interrupt pro Port gebraucht, dann muss ein Dispatcher implementiert werden, welcher der Applikation mitteilt welcher Port Pin der Auslöser ist. Wird nur ein Interrupt pro Port gebraucht, dann ist dieser Dispatcher unnötig.

Bei nur einem Interrupt pro Port wird ein Vektor belegt der dann einen bestimmten Interrupt auslöst, PortIntX. Vector1 wird hier benutzt.

Bei mehreren Interrupts pro Port wird der Dispatcher verwendet, der herausfinden muss welcher von den bis zu 8 Pins die Ursache ist, da nur zwei Vektoren pro Port existieren. Diese Methode nennt sich dann PinChangeInterrupt PCintX. Vector0 wird hier benutzt.

Beide Interrupt Typen können gleichzeitig für ein Port benutzt werden.

4.8.3.1 Interrupt Pins PortIntA .. PortIntR

Die PortInts müssen nicht importiert sondern nur definiert werden:

```
Define   PortIntD  = 0, PullUp, bothedges;    // pin0 used, Pullup on, both edge
```

PortInt**X** bestimmt das IO-Port wobei X für das Port steht, PortA…PortR
0..7 bestimmt den Port Pin
PullUp, PullDown, None bestimmt die interne Beschaltung, Widerstand ca. 20kOhm
LowLevel, BothEdges, Rising, Falling bestimmt den Port Status der den Interrupt auslöst. LowLevel sollte nur in Sonderfällen benutzt werden, da hier solange der Interrupt feuert solange der Pin aus 0 ist.

Jedem PortIntX muss eine entsprechende CallBack Prozedur zugeordnet werden, die durch den Interrupt aufgerufen wird. Für PortIntD :

```
Procedure PortIntD;          // Interrupt CallBack
begin
end;
```

Achtung: diese Proceduren sind Interrupt Prozeduren! Damit verbieten sich normalerweise grössere Operationen darin, die damit andere Interrupts nur unnötig lange sperren. Weiterhin ist zu beachten dass hier nur die Register R16/R17 (_ACCA/_ACCB) und R30/R31 (_ACCCLO/_ACCCHI) gesichert wurden. Werden innerhalb dieser Prozedur weitere Register verwendet, so sind diese mit Push und Pop zu sichern. Beispiel: Push(R18) und Pop(R18) oder PushAllRegs und PopAllRegs.

Ein Beispiel Programm ist in der Demos Directory unter XMega_PortInt zu finden.

4.8.3.2 PinChangeInterrupts PCintA .. PCINTR

PCints müssen importiert und definiert werden:
Für jedes Port das einen/mehrere PCints haben soll muss ein PCint importiert werden. PortA .. PortR

```
Import   …, PCintD, PCintE;
```

Jeder PCint muss definiert werden, welche PortPins und der Pin Status.

```
Define  PCintDmask    = $FF;          // all pins used
        PCintDedge    = $00;          // all falling edges
        PCintEmask    = %00100001;    // only Pin0 and Pin5 used
        PCintEedge    = $FF;          // all rising edges
```

PcintXmask **X** steht für das Port (A..R). Jedes auf 1 gesetzte bit innerhalb der Maske gibt den entsprechen Pin Interrupt dieses Ports frei.
PcintXedge **X** steht für das Port (A..R). Ein 0-Bit in Edge bestimmt die fallende, ein 1-Bit bestimmt die steigende Flanke als Interrupt Auslöser für das entsprechende Port-Bit.

Jedem freigegebenen PCintX-bit muss eine entsprechende CallBack Prozedur zugeordnet werden, die durch den Interrupt aufgerufen wird. Für PCintD :

```
Procedure PCintD0;          // Interrupt CallBack PinD.0
begin
end;

Procedure PcintD1;          // Interrupt CallBack PinD.1
```

Achtung: diese Proceduren sind Interrupt Prozeduren. Damit verbieten sich normalerweise grössere Operationen darin, die damit andere Interrupts nur unnötig lange sperren. Weiterhin ist zu beachten dass hier nur die Register R16/R17 (_ACCA/_ACCB) und R30/R31 (_ACCCLO/_ACCCHI) gesichert wurden. Werden innerhalb dieser Prozedur weitere Register verwendet, so sind diese mit Push und Pop zu sichern. Beispiel: Push(R18) und Pop(R18) oder PushAllRegs und PopAllRegs.

Ein Beispiel Programm ist in der Demos Directory unter XMega_PCint zu finden.

4.8.4 TRAPS und Software Interrupts (SWI)

Software-Interrupts (SWI) oder auch Traps genannt, sind bei grösseren Prozessoren die (fast) einzige Möglichkeit von der Anwendungs- Ebene in die System-Ebene zu kommen (application level -> system level). Viele (privilegierte) Operationen sind hierbei nur auf der System Ebene möglich, z.B. IO-Zugriffe, Speicher-Zugriffe in gesperrte Bereiche etc.

Ein weiterer Grund für SWI oder Traps ist die Kommunikation zwischen einem Programmteil und einem anderen, wo beide Teile die Struktur, Funktionen und Adressen des anderen nicht kennen, z.B. Debug-Monitore.

Privilegierte Ebenen gibt es beim AVR nicht. Deshalb gibt es hier auch keine SWI oder Traps, obwohl dies in manchen seltenen Fällen hilfreich wäre. Man kann jedoch, wie Atmel es vorschlägt, auch Hardware Interrupts dazu heranziehen. Die meisten Interrupts der AVRs scheiden hierbei jedoch aus, da immer eine interne Peripherie oder ein PIN Interrupt damit verschwendet wird. Bei den Mega CPUs gibt es jedoch den SPMRDY Interrupt, der praktisch nie gebraucht wird und der auch auf ganz einfache Weise zu starten ist.

Der AVRco unterstützt Traps durch den SPMRDY Interrupt. Nicht ganz so elegant wie echte Traps oder SWIs, aber doch in einer brauchbaren Weise. Eine Alternative zu Interrupts ist auch noch eine Jump Tabelle.

Wozu Traps beim AVR?
In den allermeisten AVR Applikation sind Traps/SWI absolut unnötig. Es gibt nur ein paar ganz wenige Spezial Fälle was diese Sinn machen. Immer dann wenn ein Programm gepatcht wird und Teile davon nichts von einander „wissen“ gibt es das Problem mit der Kommunikation zwischen diesen Teilen. Man kann keine Funktion des jeweils anderen Teil aufrufen. Man kann nur mit grössten Schwierigkeiten Daten austauschen etc.

Eine typische Anwendung ist z.B. ein Debug Monitor. Ohne Traps/SWI muss dieser seine Einsprungstelle auf einer vereinbarten absoluten Adresse haben, ansonsten kann die Applikation von sich aus keine Verbindung mit dem Monitor aufnehmen.

Wie funktionierts?
Das System stellt die Prozedur ***Trap (t : byte)*** zur Verfügung. Diese kann zu jeder Zeit aufgerufen werden. Der Parameter „t“ ist ein beliebiges Byte das dem Empfänger zur Verfügung gestellt wird. Vorausgesetzt dass der globale Interrupt freigegeben ist, löst die Prozedur „Trap“ einen SPMRDY Interrupt aus.

Dieser Interrupt wird im einfachsten Fall von einer Interrupt Prozedur bearbeitet ***Interrupt SPMRDY*** oder von einem TrapHandler, den die Applikation zur Verfügung stellen muss ***TrapHandler (t : byte)***.

Das ganze passiert bis jetzt aber alles komplett in der Applikation. Wie kommt nun der hier unbekannte externe Teil ins Spiel? Im ersten Fall mit „Interrupt SPMRDY“ kann bei der Definition die absolute Einsprung Adresse für die externe Interrupt Prozedur angegeben werden, dann kann diese Interrupt Prozedur in der Applikation selbst entfallen. Im Zweiten Fall gilt das gleiche für den TrapHandler.

In jedem Fall, wenn ein externes Programmteil ins Spiel kommt und der Trap in diesem bearbeitet werden soll, muss diese absolute Adresse angegeben werden. Das ist übrigens auch bei den „grossen Brüdern“ so ähnlich.
Diese Adresse ist eine **Byte**-Adresse und muss geradzahlig sein.

Der Parameter „t“ wird bei der Interrupt Prozedur „SPMRDY“ im _ACCA bzw. R17 übergeben.

Ein Spezial Fall sind Traps die in den Boot Bereich gehen. Hier sind Interrupts nicht der richtige Weg. Deshalb gibt es noch eine spezielle Trap Implementation die mit der hier beschriebenen nichts gemeinsam hat. Näheres weiter unten im Kapitel ***BootTraps***.

4.8.4.1 Implementation der Traps

Defines

Die Definition des TrapHandlers bestimmt die Art der Behandlung von Traps

```
Define  ProcClock    = 8000000;        {Hertz}
        SysTick      = 10;             {msec}
        StackSize    = $0030, iData;
        FrameSize    = $0030, iData;
        TrapHandler  = false, 0;       {Intproc only, no ext address}
```

Beim Define des TrapHandlers bestimmt der erste Parameter (true/false) ob eine Interrupt oder TrapHandler Prozedur erwartet wird.
Der zweite Parameter DestAddr (longword) bestimmt ob diese Prozedur Bestandteil dieser Applikation (0) oder extern (> 0) ist.

TrapHandler = false
Der *Trap*(*nn*) Aufruf löst einen SPMRDY Interrupt aus. Dieser Interrupt wird entweder

- Applikations-intern (DestAddr = 0) oder
- extern (DestAddr > 0)

ausgeführt.
Diese Interrupt Prozedur muss genauso definiert und behandelt werden wie alle anderen auch. Auch hier gilt möglichst nur ganz kurze Verweildauer, ggf. Register Rettung etc. Der globale Interrupt ist beim Aufruf noch **gesperrt**.

```
Interrupt SPMRDY;
begin
  // _ACCA/R17 contains the parameter "t"
end;
```

TrapHandler = true
Der *Trap(nn)* Aufruf löst einen SPMRDY Interrupt aus. Der Handler selbst wird entweder

- Applikations-intern (DestAddr = 0) oder
- extern (DestAddr > 0)

ausgeführt. Der selbst Interrupt wird intern verarbeitet und ruft die interne/externe Handler Prozedur auf, wobei der Interrupt schon wieder **freigegeben** ist. Damit ist der Handler eine ganz normale Prozedur mit absolut keinen Einschränkungen.

```
Procedure TrapHandler(t : byte);
begin
  ...
end;
```

DestAddr
Wie schon angeführt ist diese Adresse die Einsprung Adresse in die externe Interrupt oder Handler Prozedur und muss eine geradzahlige Byte Adresse sein. Damit hat z.B. das Flash-Ende des mega128 die Adresse $1FFFE.

Bemerkung:
Wird mit DestAddr = 0 gearbeitet, muss die entsprechende Prozedur (Interrupt oder TrapHandler) in der Anwendung definiert sein.
Wird mit DestAddr > 0 gearbeitet, muss die entsprechende Prozedur (Interrupt oder TrapHandler) extern vorhanden sein. Deklarationen in der Applikation selbst werden ignoriert.

4.8.5 EXCEPTIONS

Bei komplexen Programmen mit vielen Unter-Routinen und Treibern gibt es sehr häufig den Umstand, dass ganz tief unten in einer Basis Funktion (z.B. In/Out) ein Fehler oder Time-out auftritt. Diese Funktion kann jetzt z.B. ein false an die aufrufende Funktion zurückgeben, diese gibt ebenfalls wiederum ein false zurück etc. bis der Aufruf an der allerersten Stelle angelangt ist, wo die Aktion angestossen wurde. Hier kann bzw. muss jetzt das Resultat (false) ausgewertet werden, das „ganz tief unten“ seinen Ursprung hatte. Das funktioniert aber nur, wenn jede beteiligte Funktion ein boolean zurück gibt und die übergeordnete Funktion dies auch mit

```
if not funcx then
  Return(false);
else ...
```

entsprechend behandelt.

Das ganze ist sehr umständlich und fehlerträchtig. Borland hat vor Jahren für diesen Zweck die sehr eleganten und mächtigen ***Exceptions*** in Delphi eingeführt. Exceptions sind sogenannte „Ausnahmen“. Das bedeutet, tritt eine solche Ausnahme auf, dann kehrt das Programm direkt zu der Stelle zurück, an der die Exception implementiert wurde. Das umständliche „hochhangeln“ durch die einzelnen Funktionen entfällt.

Exceptions bestehen immer aus dem Implementation Teil, begrenzt durch ***Try*** und ***EndTry*** und einem oder mehreren ***RaiseException*** Statements. Zwischen *Try* und *EndTry* kann optional auch noch ein ***Except*** stehen.

```
Try
  StatementE..
  StatementE..
  StatementE..
EndTry;
```

oder

```
Try
  StatementE..
  StatementE..
  StatementE..
except
  StatementN..
  StatementN..
EndTry;
```

Trifft die Ausführung eines *Statement* auf die Anweisung

```
RaiseException (num);
```

so wird jede weitere Operation abgebrochen und die Applikation kehrt zu dem Try/EndTry Block zurück. Im ersten Beispiel wird mit dem Statement nach der Anweisung EndTry fortgefahren. Im zweiten Beispiel wird mit dem Statement nach der Anweisung except fortgefahren. Dann kann mit der Funktion

```
GetExceptResult
```

der Parameter „num” gelesen werden, der durch die Funktion *RaiseException* übergeben wurde.

Wenn keine Exception auftrat, werden im ersten Beispiel alle Statements abgearbeitet und im zweiten Beispiel nur die Statements zwischen *Try* und *Except*, dann wird die Ausführung nach *EndTry* fortgesetzt.

Der bessere Weg ist immer das Beispiel zwei, wo die Applikation über eine „Ausnahme“ informiert wird, indem der Block zwischen *Except* und *EndTry* abgearbeitet wird.

Achtung
Eine Exception wird nur ausgelöst (durch *RaiseException (num))* wenn diese in der Statement Folge zwischen *Try* und *EndTry* bzw. *Except* stattfindet. Ansonsten wird diese ignoriert. Ein *Try/EndTry* Block muss sich immer innerhalb eines übergeordneten Blocks befinden. Dies ist illegal:

```
if a > b then
  Try
else
 Endtry;
endif;
```

Einschränkung
Bei MultiTasking gelten zur Zeit noch erhebliche Einschränkungen. *Try/EndTry* dürfen nur im *MAIN* implementiert werden. Die *RaiseException* Funktion ist deshalb auch nur wirksam wenn in diesem Moment der *MAIN* Prozess die Kontrolle über das System hat, ansonsten wird sie ignoriert.

4.8.5.1 Implementation

Imports
Der Exception Support muss importiert werden.

```
Import SysTick, TickTimer, ..;
From System Import Exceptions, ...;
```

Defines
Die Exceptions können auch indirekt verschachtelt sein. Die Verschachtelungs-Tiefe muss definiert werden

```
Define Exceptions = 2[, Boot[, iData1]];       // 1..15 levels, Boot and iData1 is optional
```

Exceptions dürfen selbst nicht direkt verschachtelt werden. Es ist aber zulässig, dass während ein *Try/EndTry* aktiv ist, weitere in untergeordneten Statements platziert werden können. Wie viele das sein können, wird durch obiges Define festgelegt. Jeder Level braucht dazu einen Satz Parameter, der in einer Art Stack im RAM angelegt wird. Sind alle *Try/EndTry* Levels zur Laufzeit belegt, werden weitere Implementation ignoriert. Das ganze wird zur Laufzeit aufgebaut und nach dem *EndTry* wieder aufgelöst. Man kann hier von einem Exception Stack sprechen.

Ist die Option *Boot* gewählt, so wird der Exception Handler in den Boot Bereich gelegt. Da der Handler ein paar System Variablen erzeugt und benötigt, dürfen sich die Adressen dieser Vars auch nach einem Flash Download niemals ändern. Da dies von der Applikation nicht immer gewährleistet werden kann, macht es Sinn diese Vars mit der Option in den zu definierenden Idata1 Bereich zu legen, wo sie immer an erster Stelle stehen.

4.8.5.2 Funktionen

Mit dem Import von Exceptions werden auch zwei Support Funktion importiert:

```
Procedure RaiseException (num : byte);
```

Dies Prozedur löst die Exception (Ausnahme) aus. Das Programm kehrt sofort an die Stelle zurück, die im Exception Stack an erster Stelle (Top Level) steht. Der Parameter num wird abgespeichert und kann anschliessend durch untenstehende Funktion gelesen werden. Bitte beachten dass ab hier der globale Interrupt gesperrt wird.
Ist der Exception Stack leer, d.h. es findet sich momentan kein verantwortliches *Try / EndTry*, wird die Exception Prozedur ignoriert.

```
Function GetExceptResult : byte;
```

Der mit der Funktion *RaiseException* übergebene Parameter *num* kann durch diese Funktion gelesen werden. Der Parameter kann diverse Aufgaben erfüllen, z.B. anzeigen welche von mehreren Exceptions der Auslöser war.

Programm Beispiel:
Ein Beispiel Program befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Exceptions**

4.9 Statements

Der Statement Block eines Programms, einer Unit, Prozedur oder Funktion beschreibt die Operationen die ausgeführt werden sollen in einer Liste von *Statements.* Jedes Statement spezifiziert einen Teil der ganzen Aktion. Deshalb nennt man Pascal auch eine sequentielle Programmier Sprache: Statements werden zeitlich nacheinander ausgeführt und normalerweise nicht simultan. Der Statement Block ist durch die reservierten Wörter **begin** und **end** begrenzt. Innerhalb sind die Statements durch Strichpunkte von einander abgegrenzt. Statements können entweder *einfach* oder *strukturiert* sein.

4.9.1 Einfache Statements

Einfache Statements sind Statements die keine weiteren Statements enthalten. Diese sind das Assignment (Zuweisungs) Statement, das Prozedur Statement, das goto statement, und das leere Statement.

4.9.2 Assignment (Zuweisungs) Statement

Das wesentlichste aller Statements ist das Assignment (Zuweisungs) Statement. Mit ihm kann ein Wert einer Variablen zugewiesen werden. Ein Assignment besteht aus einem Variablen Identifier (Namen), gefolgt von dem Assignment Operator **:=** wiederum gefolgt von einem Ausdruck (Expression).

Eine Zuweisung zu Variablen jeglichen Typs ist möglich, so lange wie die Variable, Funktion oder Expression (Source, Quelle) vom gleichen Typ oder kompatibel ist zum Typ des Ziels. In anderen Worten: links und rechts vom Zuweisungs Operator müssen die gleichen Typen sein, oder zumindest kompatible.

```
Angle:= Angle * 3;
AccessOK:= False;
AccessOK:= Answer = PassWord;
Result:= (Entry * 13) shl 8;
```

4.9.3 Prozedur Statement

Ein Prozedur Statement dient zum Aufruf einer schon definierten Prozedur, sei sie nun User definiert oder eine vom System schon definierte Prozedur. Das Statement besteht aus einem Prozedur Identifier (Namen) evtl. mit Parameter Liste, welche eine geklammerte Liste von Variablen oder Ausdrücke ist, durch Kommas getrennt sind. Wenn das Prozedur Statement im Programm Ablauf erreicht wird, geht die Programm Kontrolle auf diese Prozedur über, und die evtl. vorhandene Parameter Liste wird an die Prozedur übergeben. Wenn die Prozedur fertig ist, fährt der Programm Ablauf mit dem nächsten Statement fort, das dem Procedure Statement/Aufruf folgt.

```
Find (Name,Address);
Sort (Address);
Uppercase (Text);
```

4.9.4 Leeres Statement

Ein 'leeres' Statement ist ein Statement das keine Symbole enthält, und das auch nichts bewirkt.
Es kann immer dann verwendet werden, wenn Pascal ein Statement erwartet, jedoch keine Aktion ausgeführt werden soll.

```
repeat until KeyPressed;     {wait for any key to be hit}
```

4.9.5 Strukturiertes Statement

Strukturierte Statements sind Konstrukte bestehend aus anderen Statements welche nacheinander abgearbeitet werden sollen (compound statements), oder abhängig von Bedingungen (conditional statements), oder wiederholt (repetitive statements) abgearbeitet werden.

4.9.6 Verbund Statement

Ein Verbund Statement wird dann verwendet, wenn mehr als ein Statement ausgeführt werden soll, wo eigentlich nur ein Statement möglich ist. Es besteht aus einer Anzahl von Statements, jeweils abgeschlossen mit einem Strichpunkt. Der ganze Block beginnt mit einem einleitenden Statement (z.B. *if .. then*) und endet mit einem zugehörigen reservierten Wort(z.B. *endif*).

```
if Small > Big then
  Tmp := Small;
  Small := Big;
  Big := Tmp;
endif;
```

4.9.7 NOP Statement

Das Pascal Statement "NOP" ist implementiert. Es erzeugt ein Assembler "NOP"

```
Incl (bit);
NOP;
Excl (bit);
```

4.9.8 Bedingte Statements

Ein bedingtes Statement führt eine einzelne Komponente seiner Statements aus.

Entscheidungen und Verzweigungen werden in der Regel mit "*if..then..else*" herbeigeführt. Das ist ein allgemeiner und jederzeit anwendbarer Weg, aber oft ineffizient. D.h. verschwendet Code und Laufzeit. Sind in einer Entscheidung Konstanten im Spiel, kann man abkürzen mit "***if** x **in**[a, b, m..p]* ".
Eine weitere Verkürzung bedeutet der Einsatz von "case" Konstrukten.

Da nicht in allen Fällen Konstante im Spiel sind fällt das *case* oftmals weg. Hier können System Funktionen wie z.B. *IncToLim, DecToLim, IncToLimWrap, DecToLimWrap, ValueTrimLimit, ValueInTolerance, Lower, Higher* etc. eine Code Reduktion in IF-Konstrukten und auch ganz allgemein bewirken.

4.9.8.1 IF Statement

IF
Das IF Statement benötigt zumindest THEN u. ENDIF

THEN

```
iF a > b then a:= b; endIf;
```

ELSE

```
iF a > b then a:= b; else b:= a; endIf;
```

ELSIF

```
iF a > b then ..; elsif b = a then ..; endIf;
```

ENDIF Ende eines IF Statements

IF ist das einleitende Statement für eine Verzweigung. Auf *IF* **muss** eine Operation mit einem bool'schen Ergebnis (*true*/*false*) folgen. Dies kann ein Vergleich sein (a > b) oder auch ein Boolean an sich (*bit* oder *var*).

Nach der Operation **muss** ein *then* folgen. Das Pascal-übliche *begin* ist hierbei jedoch nicht zulässig (siehe Beschreibung von *begin*). Nach *then* muss mindestens 1 ausführbares Statement kommen. Dann kann optional ein *else* oder *elsif* kommen, wobei auch hier kein *begin* zulässig ist und mindestens ein ausführbares Statement folgen muss.

Im Gegensatz zu Standard-Pascal muss *IF* immer mit *ENDIF* abgeschlossen sein.

Unter bestimmten Bedingungen ist es sinnvoll mit Goto statt mit IF zu arbeiten. Das sollte aber die Ausnahme sein.

Achtung:
Verschachtelte IF's sind zwar zulässig, müssen aber mit Vorsicht programmiert werden, da Zwischenergebnisse grundsätzlich auf dem Stack zwischengespeichert werden. Im Zweifelsfall (Stack-Overflow) ist der entsprechende Programmcode mittels einem Debugger/Simulator zu prüfen. Besser als eine Verschachtelung ist hier manchmal die Verwendung von **elsif.**

```
if a > b then
  ...
elsif a = b then
  ...
else          {a ist kleiner b}
  ...
endif;
```

4.9.8.2 GOTO Statement

GOTO ist eine absolute Sprunganweisung, die innerhalb eines **Blocks** (Blockdefinition siehe unten) an eine beliebige Stelle führen kann (vorwärts und rückwärts), und das ist auch die darin verborgene Gefahr. Das Sprungziel ist immer ein sog. Label, das zuvor definiert werden muss. Innerhalb eines Blocks ist immer nur **eine Labeldefinition** und damit auch nur **ein Sprungziel** zulässig. Es können aber **mehrere GOTOs** auf ein Ziel zeigen.

Eine GOTO Anweisung besteht aus dem reservierten Wort **goto**, gefolgt von einem Label Bezeichner. Sie dient dazu, die Programmausführung an denjenigem Punkt fortzusetzen, der mit dem Label gekennzeichnet ist.
Die folgenden Regeln müssen beachtet werden, wenn GOTO Anweisungen benutzt werden:

1) Labels müssen vor der Benutzung deklariert werden. Die Deklaration muß im Deklarationsteil desjenigen Blocks erfolgen, in dem das Label benutzt wird.
2) Der Geltungsbereich eines Labels ist der Block, in dem es deklariert wurde. Somit ist es nicht möglich in oder aus Funktionen und Prozeduren zu springen.

```
Procedure GotoSample;
begin
  Label: lab1;          {Definition des Labels}
  ...
  Goto Lab1;
   ...
  Lab1:                 {Sprungziel, kein Semikolon!!}
   ...
  IF a > b then
    Label: Lab2;
    Lab2:               {Sprungziel, kein Semikolon!!}
    ...
    goto Lab2;
    ...
  endif;
    ...
  goto Lab1;
end;
```

Block Definition
In Pascal sind Blöcke durch bestimmte Blockgrenzen bestimmt. Eine typ. Blockgrenze ist z.B. das BEGIN und END einer Prozedur oder Funktion. Schleifenblöcke werden durch LOOP..ENDLOOP, REPEAT..UNTIL, WHILE..ENDWHILE, FOR..ENDFOR eingegrenzt. Die Statements zwischen IF..ENDIF, IF.. ELSE, IF..ELSIF, ELSIF..ELSIF, ELSIF..ELSE, ELSE..END bilden ebenfalls **jeweils einen Block**.

Alle drei Statements (Label, Goto und das Ziel) für ein Goto müssen sich innerhalb eines solchen Blocks befinden. Dabei spielt es jedoch keine Rolle, wenn der Block durch Unterblöcke (s.o. Begin IF..ENDIF END) nocheinmal aufgeteilt wird. Im obigen Beispiel wird LAB2 nur innerhalb von IF..ENDIF anerkannt, während Lab1 in der gesamten Prozedur bekannt ist, jedoch nicht in dem Unterblock IF..ENDIF.

Anmerkung:
in C sind Blockgrenzen durch **{** und **}** bestimmt und damit sehr übersichtlich :-)

GOTO
Das Goto Statement benötigt eine Definition des zugehörigen Labels und den Einsatz dieses Labels innerhalb des gleichen Blocks.

LABEL
Das Label definiert die Ziel Adresse für das Goto Statement.
Der Programmierer sollte grundsätzlich auf lesbaren und damit pflegeleichten Code achten. Das hier beschriebene GOTO-Statement wiederspricht dieser Forderung generell. Manchmal, aber nur manchmal, bringt ein GOTO mehr Übersicht ins Programm. Aus diesem Grund sollte man GOTO nur in Notfällen einsetzen, da in den meisten Fällen ein IF- oder Loop-Statement wesentlich eleganter und sicherer ist.

4.9.8.3 CASE Statement

Die Case Anweisung besteht aus einem Ausdruck (dem Selektor) und einer Liste von Anweisungen denen jeweils ein Label vom Typ des Selektors vorangeht. Es wird diejenige Anweisung ausgeführt, deren Label gleich dem aktuellen Wert des Selektors ist.
Wenn kein Label gleich dem aktuellen Wert des Selektors ist, wird entweder keine Anweisung oder optional die Anweisungen nach dem reservierten Wort ***else*** ausgeführt. Die else Bedingung ist eine Erweiterung des Standard Pascal.

Ein Case Label besteht aus einer oder vielen Konstanten oder Unterbereichen, die durch Kommas getrennt sind und denen ein Doppelpunkt folgt. Ein Unterbereich wird durch zwei Konstanten ausgedrückt, die durch '..' getrennt sind. Der Typ der konstanten muß identisch dem typ des Selektors sein. Die dem Case Label folgende Anweisung wird ausgeführt wenn der Wert des Selektors gleich einer der Konstanten ist oder in einem der Unterbereiche liegt.

Das Case Label kann neben einem beliebigen 8 Bit Datentyp wie *Byte*, *Int8, Enum* oder *Char* auch *Word* und *Integer* (16 Bit) haben.

Case ist das einleitende Statement für einen Verzweigungs-Block. Auf Case **muss** eine 8bit Variable (byte, char oder enum) folgen und darauf das Wort “of” (case v1 of ..). Innerhalb des Blocks erfolgen Anweisungen, was zu tun ist, wenn die Variable identisch ist mit einer Konstanten oder einem Konstanten Bereich (7 : ... oder 8..12 : ...).
Diese Bereichs-Deklaration muss mit einem Doppelpunkt abgeschlossen werden. Der daran anschliessende Bereichsblock muss mindestens ein Statement enthalten. Jedes einzelne Statement **muss** mit einem ‘**;**’ abgeschlossen sein. Dieser Sub-Block ist mit einem Separator-Zeichen (**|**) abzuschliessen.

Nach einem oder mehreren Bereichs-Blöcken **kann** ein optionales **Else** folgen mit weiteren Statements. Der Abschluss bildet grundsätzlich das EndCase Statement. Case ist eigentlich ein spezialisiertes IF-Statement.

```
const   c1 = 83;
        c2 = 105;

Case x of
  0          : inc(a);
               |
  1, 7       : a:= 4;
               |
  2..6       : x:= x + a;
               dec(x);
               |
  8..9, 12 :   PWMport1 (45);
               |
  14..23,
  27..67     : a:= a * a;
               |
  c1..c2     : a:= c1;
               |
else
  x:= 0;
EndCase;
```

CASE
Beginn eines Case Blocks.

```
Case x of ...
```

ENDCASE
Ende eines Case Blocks.

4.9.8.4 FOR Statement

For ist das einleitende Statement für eine Programm-Schleife. Auf For muss eine 8 oder 16bit Laufzeit-Variable folgen, der mit **:=** ein Startwert zugewiesen werden muss. Dann folgt der Operator **TO** oder **DownTo**, welcher angibt, ob die Laufzeit-Variable bei jedem Durchgang inkrementiert oder dekrementiert werden soll. Das optionale Argument **BY** bestimmt den Inkrement- bzw. Dekrement-Wert (1..255). Darauf folgt der Abbruchwert und die Anweisung **DO**. Der Startwert und der Abbruchwert können 8 oder 16bit Variable oder Konstanten sein.

Nach der Kopf Deklaration (For x:= 0 to v1 do) folgt ein Block mit Anweisungen, die innerhalb eines Durchgangs ausgeführt werden müssen. Dieser Block kann allerdings auch leer sein.

Die Anzahl der Durchgänge ist abhängig von der Differenz von Abbruchwert und Startwert.

Bei ***for*** *i:= 2* ***to*** *1* wird die Schleife nicht durchlaufen und **i** hat anschliessend den Wert **2**
Bei ***for*** *i:= 2* ***to*** *2* wird die Schleife einmal durchlaufen und **i** hat anschliessend den Wert **3**
Bei ***for*** *i:= 2* ***to*** *7* wird die Schleife sechsmal durchlaufen und **i** hat anschliessend den Wert **8**

Im Gegensatz zu Standard-Pascal muss *FOR* immer mit *ENDFOR* abgeschlossen sein.

```
const   a  = 0;
var  v1  : byte;
     x   : byte;

for x:= a to v1 do              {ramp up}
  PWMport1:= x;
endfor;

for x:= x downto 0 do           {ramp down}
  PWMport1:= x;
endfor;
```

Eine FOR-Schleife kann mit einem Break-Statement abgebrochen werden:

```
for x:= 1 to 9 by 2 do
  ...
  if a:= 0 then Break;
  ...
endfor;
```

Siehe auch **Continue** Anweisung!

FOR
Beginn einer For Schleife

```
for a:= 0 to 45 do inc (x); endfor;
```

ENDFOR
Ende einer For Schleife

4.9.8.5 WHILE Statement

While ist das einleitende Statement für eine Programm-Schleife. Auf While **muss** eine Operation mit einem bool'schen Ergebnis (true/false) folgen. Dies kann ein Vergleich sein (a > b) oder auch ein Boolean an sich (Bit oder var). Die Schleife wird solange durchlaufen, wie die Operation *true* ergibt. Ist die Operation schon beim Eintritt *false,* wird die Schleife nie durchlaufen.

Nach der Operation **muss** ein ***do*** folgen. Das Pascal-übliche *begin* ist hierbei jedoch nicht zulässig (siehe Beschreibung von *begin*). Nach *do* muss mindestens 1 ausführbares Statement kommen. Ein verlassen der While-Schleife ist mit **BREAK** möglich. Siehe auch **Continue** Anweisung!

Im Gegensatz zu Standard-Pascal muss *WHILE* immer mit *ENDWHILE* abgeschlossen sein.

```
while x < 100 do
  inc (x);
endwhile;
```

WHILE
Beginn einer While Schleife

```
while a < b do inc (a); endwhile;
```

ENDWHILE
Ende einer While Schleife

4.9.8.6 REPEAT Statement

REPEAT ist das einleitende Statement für eine Programm-Schleife. Nach REPEAT **kann** ein oder mehrere ausführbares Statement kommen. Das Pascal-übliche *begin* ist hierbei jedoch nicht zulässig (siehe Beschreibung von *begin*).

Den Abschlu**s**s der Schleife bildet das Statement UNTIL. Danach **muss** eine Operation mit einem bool'schen Ergebnis (true/false) folgen. Dies kann ein Vergleich sein (a > b) oder auch ein Boolean an sich (Bit oder var). Die Schleife wird solange durchlaufen, wie die Operation false ergibt. Also, im Gegensatz zu *WHILE,* mindestens einmal. Ein verlassen der Repeat-Schleife ist mit **BREAK** möglich
Siehe auch **Continue** Anweisung!

```
x:= 0;
repeat
  inc(TCC);
until TCC > 20;
```

REPEAT
Beginn einer Repeat Schleife

UNTIL
Ende einer Repeat Schleife

4.9.8.7 CONTINUE

Das Statement Continue setzt die Programmausführung mit dem nächsten Durchlauf einer **for**-, **while**- oder **repeat**-Schleife fort. Die nachfolgenden Statements bis zu EndFor, EndWhile oder Until werden ignoriert.

4.9.8.8 LOOP Statement

In einer Controller-Anwendung wird das Programm normalerweise niemals beendet, d.h. das Programm läuft in einer Endlosschleife. Pascal bietet hier nur die Konstruktion an: *Repeat ... until false = true;*
Das ist formal zwar ok, aber eigentlich wiederum Nonsens.

Modula-2 kennt hier die Endlosschleife *LOOP* und diese ist hier auch implementiert worden.
Alle Statements innerhalb *Loop ... Endloop;* werden kontinuierlich abgearbeitet.

Loop .. Endloop; ist aber auch innerhalb einer Prozedur oder sogar innerhalb einer Loop zulässig.
Um daraus wieder herauszukommen, ist das Statement *ExitLoop* notwendig.
ExitLoop verlässt die Schleife und fährt mit dem nächsten Statement nach *EndLoop* fort.

```
begin
  loop
    ...
  endloop;
end.
```

LOOP
Beginn einer Endlos Schleife

ENDLOOP
Ende einer Endlos Schleife

EXITLOOP
Abbruch einer Endlos Schleife

```
loop
  ...
  ...
if a > b then
  exitloop;
endif;
  ...
endloop;
```

4.10 System Library - Standard

4.10.1 TRUE

Vordefinierte Konstante = $FF

4.10.2 FALSE

Vordefinierte Konstante = $00

4.10.3 PI

Vordefinierte Konstante = 3.141592654 (float)

4.10.4 NIL

Vordefinierte Konstante vom Typ Pointer = $0000

```
p:= nil;
```

4.10.5 Typ Konvertierung

4.10.5.1 BOOLEAN

Wandelt das Argument in ein Boolean

```
Function Boolean (a : type) : boolean;

bo:= boolean (x);
```

4.10.5.2 BYTE

Wandelt das Argument in ein Byte

```
Function Byte (a : type) : byte;

b:= byte (x);
```

4.10.5.3 INT8

Wandelt das Argument in ein Int8

```
Function Int8 (a : type) : int8;

i:= int8 (x);
```

4.10.5.4 CHAR

Wandelt das Argument in ein Char

```
Function Char (a : type) : char;

ch:= char (x);
```

4.10.5.5 WORD

Wandelt das Argument in ein Word

```
Function Word (a : type) : word;

w:= word (x);
```

4.10.5.6 INTEGER

Wandelt das Argument in ein Integer

```
Function Integer (a : type) : integer;

i:= Integer (x);
```

4.10.5.7 LONGWORD

Wandelt das Argument in ein LongWord

```
Function LongWord (a : type) : longword;

ww:=longword (x);
```

4.10.5.8 LONGINT

Wandelt das Argument in ein LongInt

```
Function LongInt (a : type) : longint;

Li:= LongInt (x);
```

4.10.5.9 FLOAT

Wandelt das Argument in ein Float.

```
Function Float (a : type) : float;

f:= Float (x);
```

4.10.5.10 FLOATASLONG

Wandelt das Argument in ein LongWord. Es findet allerdings keinerlei Konvertierung oder Verarbeitung statt.

```
Function FloatAsLong (f : float) : longword;

structconst
  lws : longword = FloatAsLong (1.234);

const
  lwc : longword = FloatAsLong (1.234);

// var
  lw:= FloatAsLong (fc);
  lw:= FloatAsLong (fs);
```

4.10.5.11 LONGASFLOAT

Wandelt das Argument in einen Float. Es findet allerdings keinerlei Konvertierung oder Verarbeitung statt.

```
Function LongAsFloat (L : LongWord) : float;

strutconst
  fs  : Float    = LongAsFloat ($12345678);

const
  fc  : Float    = LongAsFloat ($12345678);

// var
  f:= LongAsFloat (lws);
  f:= LongAsFloat (lwc);
```

4.10.5.12 POINTER

Wandelt das Argument in einen Pointer

```
Function Pointer (a : type) : pointer;

p:= Pointer (x);
```

4.10.6 Character und String Funktionen

4.10.6.1 ORD

Ordnungszahl eines Zeichens/Characters.

```
Function Ord (ch : char) : byte;

b:= ord ('a');              { $61 }
```

4.10.6.2 UPCASE

Wandelt char in Grossbuchstaben.

```
Function Upcase (ch : char) : char;

ch:= UpCase (ch);
```

4.10.6.3 LOWCASE

Wandelt char in Kleinbuchstaben.

```
Function LowCase (ch : char) : char;

ch:= LowCase (ch);
```

4.10.6.4 UPPERCASE

Wandelt String in Grossbuchstaben.

```
Function Uppercase (st : string) : string;

st:= UpperCase (st);
```

4.10.6.5 LOWERCASE

Wandelt String in Kleinbuchstaben.

Function *LowerCase (st : string) : string;*

st:= LowerCase (st);

4.10.6.6 COPY

Kopiert aus einem String einen Teilstring. Das Ergebnis ist ein String. Das Argument “pos” ist die Startposition in der Source und “count” ist die zu kopierende Länge. Kann in Verbindung mit "concatenate" benutzt werden.

Function *Copy (st : string; pos, count : byte) : string;*

st:= Copy (st, 2, 3);

4.10.6.7 STRREPLACE

Der String/char src überschreibt den Zielstring dest ab der Position pos.
Der Zielstring behält seine ursprüngliche Länge.

Procedure *StrReplace (src : string[char];* ***var*** *dest : string; pos : byte);*

4.10.6.8 TRIM

Entfernt führende und nachfolgende Leerzeichen von dem String.

Function *Trim (****const*** *st : string) : string;*

4.10.6.9 TRIMLEFT

Entfernt führende Leerzeichen von dem String.

Function *TrimLeft (****const*** *st : string) : string;*

4.10.6.10 TRIMRIGHT

Entfernt nachfolgende Leerzeichen von dem String.

Function *TrimRight (****const*** *st : string) : string;*

4.10.6.11 PADLEFT

Fügt führende Leerzeichen (oder das optionale Pad Char) vor den String ein. Der String wird soweit verlängert dass seine neue Länge jetzt ***len*** ist. War der original String schon >= len dann wird nichts geändert.

Function *PadLeft (****const*** *st : string; len : byte [;pad : char]) : string;*

4.10.6.12 PADRIGHT

Hängt Leerzeichen (oder das optionale Pad Char) an den String an. Der String wird soweit verlängert dass seine neue Länge jetzt ***len*** ist. War der original String schon >= len dann wird nichts geändert.

Function *PadRight (****const*** *st : string; len : byte [;pad : char]) : string;*

4.10.6.13 LENGTH

Funktion gibt die aktuell belegte Länge eines Strings zurück.

```
Function Length (s : string) : byte;

x:= length (st1);
```

4.10.6.14 SETLENGTH

Die Prozedur verkürzt/verlängert den String auf die angegebene Länge, im Rahmen der max. Länge des zugrunde liegenden String Typs.

```
Procedure SetLength (st : string; len : byte);

SetLength (st, 6);
```

4.10.6.15 POS

Funktion gibt die Position eines Zeichens in einem String zurück.

```
Function Pos (a : char; s : string) : byte;

x:= Pos (ch1, st1);
```

4.10.6.16 POSN

Gibt die Position von Character a im String s zurück. Gesucht wird ab der Stelle start im String.

```
Function PosN (a : char; s : string; start : byte) : byte;
```

Diese Funktion ist ähnlich der Funktion ”POS”. POS sucht nach einem Zeichen vom Anfang des Strings an String[1]. Sie kann nur das erste vorkommen des gesuchten Zeichens finden. Mit PosN kann die Suche an jeder beliebigen Stelle innerhalb des Strings beginnen. Deshalb kann PosN dazu benutzt werden, um das mehrfache Vorkommen eines Zeichens in einem String festzustellen.

PosN gibt die Position von ”char” in ”string” zurück. Die erste mögliche Position in string ist ”1”. Das Ergebnis ist vom Typ Byte. Wenn ”char” nicht gefunden wird, ist das Ergebnis eine ”0”. ”start” kann eine beliebige Zahl zwischen 1 und 255 sein, abhängig von der aktuellen String Länge. Eine Suche über das String Ende hinaus ergibt als Ergebnis auch eine ”0”.
PosN (ch, st1, 1) ist das gleiche wie Pos(ch, st1).
Der String muss eine Stringvariable sein / darf kein String Ausdruck sein.

```
var
   S : string[30];
   N : byte;
begin
  S:= 'Hello World Hello World';
  N:= PosN ('W', s, 10);                //N enthält jetzt 19
end;
```

4.10.6.17 APPEND

Prozedur, hängt den String/Char "src" an den String "dst" an (concat).

Procedure *Append (src : string[char];* ***var*** *dst : string);*

Die maximale Länge von "dst" wird dabei beachtet.!

4.10.6.18 INSERT

Prozedur, fügt ein String/Char in einen String ein.

Procedure *Insert (src : string[char];* ***var*** *dst : string; p : byte);*

Insert ('abc', st1, 4);

4.10.6.19 DELETE

Prozedur, löscht eine Anzahl Zeichen in einem String.

Procedure *Delete (****var*** *s : string; pos, count : byte);*

Delete (st1, 3, 2);

4.10.6.20 STRCLEAN

Funktion, entfernt/ersetzt Steuerzeichen in einem String.

Function *StrClean (****const*** *st : string; gt127 : boolean; subst : char) : string;*

Der String "st" wird gescannt auf Zeichen < $20 (space) und wenn das Boolean "gt127" true ist auch auf Zeichen > $7F (> #127).
Wird ein solches Zeichen gefunden, wird es durch das Zeichen "subst" ersetzt, wenn subst > #0 ist.
Ist subst = #0 dann wird das gefundene Zeichen nur gelöscht und nicht ersetzt.

4.10.6.21 STRTOINT

Funktion, konvertiert einen Strings in ein Byte, Int8, Word, Integer, LongWord, LongInt, Word64 oder Int64, abhängig von der Ziel-Variable.
Die Funktion verarbeitet Variablen aus dem RAM oder EEProm. Strings aus dem Flash werden nicht akzeptiert.

Function *StrToInt (st : string) : byte; {integer ..}*

w:= StrToInt (st);

Stringformate:

dezimaler String:	"1234"
hexadezimaler String:	"$ABCD"

4.10.6.22 StrToFix64

Funktion, konvertiert einen Strings in ein Fix64.
Die Funktion verarbeitet Variablen aus dem RAM oder EEProm. Strings aus dem Flash werden nicht akzeptiert.

Function *StrToFix64 (st : string) : Fix64;*

f:= StrToFix64 (st);

4.10.6.23 HEXTOINT

Bei *StrToInt* und der hexadezimalen Repräsentation ist Bedingung, dass der String mit dem Dollarzeichen "$" beginnt. Da es auch vorkommen kann, dass fertige Strings, z.B. aus einer Kommunikation, ohne "$" ankommen, macht es Sinn, diese Strings auch ohne Manipulation der ersten Stelle weiterverarbeiten zu können. Die Funktion erwartet zwar einen Hex-String, der aber *kein* "$" enthalten darf.

Function *HexToInt (st : string) : integer [byte, int8, word, longint, longword];*

4.10.6.24 STRTOFLOAT

Konvertierung eines Strings in einen Float Wert. Der Dezimalpunkt wird durch die globale Konstante "*DecimalSep*" definiert. Default "**.**", kann umdefiniert werden.

Function *StrToFloat (st : string) : float;*
f:= StrToFloat (st);

4.10.6.25 STRTOARR

Zur Generierung von Null-terminierte Strings (C Strings).

Function *StrToArr (**var** st : string) : **array of** char*

Kopiert solange Zeichen aus dem String in das Array bis entweder der String zu Ende ist oder das Array voll ist. In beiden Fällen ist das letzte übertragene Zeichen immer eine Null. Die Funktion ist nur innerhalb einer Zuweisung verwendbar. Der String als Quelle darf nur im RAM, Flash, EEprom liegen. Funktionen als Quelle sind nicht zulässig.

array:= StrToArr (st);

4.10.6.26 ARRTOSTR

Zum Verarbeiten von Null-terminierte Strings (C Strings).

Function *ArrToStr (arr : **array of** char) : string;*

Kopiert solange Zeichen aus dem Array in den String bis entweder eine Null gefunden wird oder das Array zu Ende ist oder der String voll ist. Die Null wird nicht mitkopiert. Die Funktion ist nur innerhalb einer Zuweisung verwendbar. Das Array als Quelle darf nur im RAM liegen. Flash, EEprom oder Funktionen als Quelle sind nicht zulässig.

string:= ArrToStr (ar);

4.10.6.27 StrCompareN

Diese Funktion vergleicht zwei Strings aber ohne Gross/Kleinschreibung zu beachte. Non case sensitive. Beide strings müssen im RAM liegen.

Function *StrCompareN(var st1, st2 : string) : boolean;*

4.10.6.28 StrCompareW

Diese Funktion vergleicht zwei Strings wobei der zweite String “pattern“ wild cards “?“ und “* “ enthalten kann. Mit “caseSens“ wird vorgegeben ob Gross/Klein Schreibung beachtet werden soll. Beide Strings müssen im RAM liegen.

Function *StrCompareW(var inpStr, pattern : string; caseSens : boolean) : boolean;*

4.10.6.29 EXTRACTFILEPATH

liefert den Pfad-Teil innerhalb eines FileNamens zurück, falls vorhanden.
"A:FName.ext" -> "A:" "ppp\nnnn" -> "ppp\"

Function *ExtractFilePath (FName : string) : string;*

4.10.6.30 EXTRACTFILENAME

liefert den Namens-Teil innerhalb eines FileNamens zurück
"A:FName.ext" -> "FName" "ppp\nnnn" -> "nnnn"

Function *ExtractFileName (FName : string) : string;*

4.10.6.31 EXTRACTFILEEXT

liefert den Extension-Teil innerhalb eines FileNamens zurück, falls vorhanden.
"A:FName.ext" -> "ext" "nnnn.ee" -> "ee"

Function *ExtractFileExt (FName : string) : string;*

4.10.7 Zugriff auf Teilbereiche von Variablen / Konstanten

4.10.7.1 SWAP

Vertauscht das LoByte mit dem HiByte bei einem Word oder Integer bzw. das LoNibble mit dem HiNibble bei einem Byte bzw. Char. Bei 32bit Typen wird das low-Word mit dem high-Word vertauscht.

Function *Swap (x : type) : type;*

X:= Swap (i);

4.10.7.2 SWAPLONG

Manchmal ist es notwendig bei 32bit Werten eine Spiegelung vorzunehmen.
Dazu tauschen das erste und das letzte Byte ihre Position. Das gleiche gilt auch für die zwei mittleren Bytes.

Function *SwapLong (**const** x : LongWord|LongInt) : LongWord|LongInt;*

longVal:= SwapLong (longVal);

Hatte die variable "longVal" vor der Operation den Wert $12345678, so hat sie hinterher den Wert $78563412.

Achtung: nicht verwechseln mit *Swap (longVal)* !!

4.10.7.3 EXCHANGEV

Vertausch die zwei Argumente. Nur einfache Variablen erlaubt.

Procedure *ExchangeV(var v1, v2 : var);*

4.10.7.4 MIRROR8

Spiegelt das Argument. Tauscht Bit7 <-> Bit0, Bit6 <-> Bit1, ...

Function *Mirror8 (b : byteint8||char) : byte|int8|char;*

4.10.7.5 MIRROR16

Spiegelt das Argument. Tauscht Bit15 <-> Bit0, Bit14 <-> Bit1, ...

Function *Mirror16(w : word|integer) : word|integer;*

4.10.7.6 MIRROR32

Spiegelt das Argument. Tauscht Bit31 <-> Bit0, Bit30 <-> Bit1, ...

Function *Mirror32(Lw : longword|longint) : longword|longint;*

4.10.7.7 LONIBBLE

niederwertigen 4 Bit eins Bytes

Function *LoNibble (b : byte|int8) : byte|int8;*

4.10.7.8 LO (Funktion)

niederwertiges Byte eines 16bit Wertes

Function *Lo (w : word) : byte;*
Function *Lo (i : integer) : byte;*

a:= lo (i);

4.10.7.9 LO (Zuweisung)

Zuweisung zu niederwertigem Byte eines Word

LO (word):= byte;

4.10.7.10 LOWORD (Funktion)

niederwertiges Word eines 32bit Wertes (LongInt oder LongWord)

Function *LoWord (ww : Longword) : word [integer];*
Function *LoWord (ii : LongInt) : integer [word];*

a:= loWord (i);

4.10.7.11 LOWORD (Zuweisung)

Zuweisung zu niederwertigem Word eines LongWord

LOWORD (long):= word;

4.10.7.12 HINIBBLE

höherwertige 4 Bit eins Bytes

Function *HiNibble (b : byte|int8) : byte|int8;*

4.10.7.13 HI (Funktion)

höherwertiges Byte eines 16bit Wertes

Function *Hi (w : word) : byte;*
Function *Hi (i : integer) : byte;*

a:= hi (i);

4.10.7.14 HI (Zuweisung)

Zuweisung zu höherwertigem Byte eines Word

HI (word):= byte;

4.10.7.15 HIWORD (Funktion)

höherwertiges Word eines 32bit Wertes (LongInt oder LongWord)

Function *HiWord (ww : Longword) : word [integer];*
Function *HiWord (ii : LongInt) : integer [word];*

a:= hiWord (i);

4.10.7.16 HIWORD (Zuweisung)

Zuweisung zu höherwertigem Word eines LongWord

HIWORD (long):= word;

4.10.8 ABS

Absolutwert eines Int8, Integer, Longint, Int64, Fix64 oder Float Wertes

Function *Abs (i : integer) : integer;*
Function *Abs (f : float|fix64) : float|fix64;*

a:= abs (a);

4.10.1 Diff8, Diff16, Diff32, Diff64

Die absolute Differenz von ganzen positiven Zahlen wird errechnet.

Function *Diff8(b1, b2 : byte) : byte;*
Function *Diff16(w1, w2 : word) : word;*
Function *Diff32(lw1, lw2 : longword) : longword;*
Function *Diff64(lww1, lww2 : word64) : word64;*

4.10.2 Negate

Der negative Wert (Two's Complement) eines Byte, Int8..Longword, LongInt, Word54, Int64, Fix64 oder Float Wertes

Function *Negate (v : type) : type;*

a:= Negate (a);

4.10.3 INC

Variable inkrementieren. Nur für byte, Int8, word, integer, longword und longint. Es erfolgt ein "wrap" wenn die Grenzen des Typs überschritten werden.

Procedure *Inc (var v [, step] : type);*

inc (a);

4.10.4 INCTOLIM

Function *IncToLim (**var** v : ordinal [, limit : ordinal[; val : ordinal]]) : boolean;*

Das Argument "v" wird erhöht, vorrausgesetzt "v" hat noch nicht seine natürliche Grenze erreicht.
Wenn der optionale Parameter "limit" angegeben ist, dient dieser als Grenze.
Wurde die Funktion erfolgreich ausgeführt, d.h. es erfolgte ein increment, gibt die Funktion ein true zurück, ansonsten ein false.
Der optionale Parameter "*val*" gibt den Inkrement Wert an.
Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint
Die Funktion ist extrem schnell und kurz. Sie eignet sich sehr gut für Schleifen Implementationen.

```
var i : integer;

repeat
 ...
until not IncToLim (i, 1000);
```

Diese Schleife solange durchlaufen, bis "i" den Wert 1000 hat.

4.10.5 INCTOLIMWRAP

Function *IncToLimWrap (**var** value, lim, pres : type) : boolean;*

Inkrementiert die Variable "value" jeweils um 1. Wird dabei der Wert "lim" überschritten, wird die Variable auf den Wert von "pres" gesetzt und das Resultat wird true, ansonsten false.
Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint

4.10.6 DEC

Variable dekrementieren nur für byte, Int8, word, integer, longword und longint. Es erfolgt ein “wrap” wenn die Grenzen des Typs überschritten werden.

Procedure *Dec (**var** v [, step] : type);*

```
dec (b);
```

4.10.7 DECTOLIM

Function *DecToLim (**var** v : ordinal [, limit : ordinal[; val : ordinal]]) : boolean;*

Das Argument "v" wird erniedrigt, vorrausgesetzt "v" hat noch nicht seine natürliche Grenze erreicht.
Wenn der optionale Parameter "limit" angegeben ist, dient dieser als Grenze.
Wurde die Funktion erfolgreich ausgeführt, d.h. es erfolgte ein decrement, gibt die Funktion ein true zurück, ansonsten ein false.
Der optionale Parameter "*val*" gibt den Dekrement Wert an.
Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint
Die Funktion ist extrem schnell und kurz. Sie eignet sich sehr gut für Schleifen Implementationen.

```
var i : integer;

while DecToLim (i) do
 ...
endhile;
```

Diese Schleife solange durchlaufen, bis "i" den Wert -32768 hat.

4.10.8 DECTOLIMWRAP

Function *DecToLimWrap (**var** value, lim, pres : type) : boolean;*

Dekrementiert die Variable "value" jeweils um 1. Wird dabei der Wert "lim" unterschritten, wird die Variable auf den Wert von "pres" gesetzt und das Resultat wird true, ansonsten false.
Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint

4.10.9 VALUETRIMLIMIT

Function *ValueTrimLimit (value, vmin, vmax : type) : type;*

Vergleicht die Variable "value" mit den beiden Grenzen "vmin" und "vmax", wobei vmin immer kleiner als vmax sein muss. Überschreitet der Wert von value eine der Grenzen, so wird als Ergebnis die jeweilige Grenze zurück gegeben, ansonsten ist das Resultat der Wert von value.
Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint, Word64, Int64, Fix64, Float

4.10.10 VALUEINTOLERANCE

Function *ValueInTolerance (value, ref, tol : type) : boolean;*

Vergleicht die Variable "value" mit der Grenze vmin, die aus (ref - tol) gebildet wird und mit der Grenze vmax, die aus (ref + tol) gebildet wird. Überschreitet der Wert von value eine der Grenzen, so wird das Ergebnis false, ansonsten true.
Type = Byte, Int8, Word, Integer, Longword, Longint, Float

4.10.11 VALUEINTOLERANCEP

Function *ValueInToleranceP (value, ref : type; tol : byte) : boolean;*

Vergleicht die Variable "value" mit der Grenze vmin, die aus (ref - (ref div 100) * tol) gebildet wird und mit der Grenze vmax, die aus (ref + (ref div 100) * tol) gebildet wird. Überschreitet der Wert von value eine der Grenzen, so wird das Ergebnis false, ansonsten true. Der Wert "tol" muss im Bereich 0..100 liegen, da er als "Prozent" betrachtet wird.
Damit entspricht diese Funktion der Funktion "*ValueInTolerance*" mit der Ausnahme, dass die Toleranz hier nicht in absoluten Werten angegeben wird sondern in Prozent.
Type = Byte, Int8, Word, Integer, Longword, Longint, Float

4.10.12 VALUEINRANGE

Function *ValueInRange (value, vmin, vmax : type) : boolean;*

Vergleicht die Variable "value" mit den beiden Grenzen "vmin" und "vmax", wobei vmin immer kleiner als vmax sein muss. Überschreitet der Wert von value eine der Grenzen, so wird als Ergebnis false ansonsten true.
Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint, Word64, Int64, Fix64,Float

4.10.13 MULDIVBYTE

Beim Teilen von Bytes durch einen gegebenen Wert wird das Ergebnis oft sehr ungenau. Nur mit einem Trick lässt sich ein Byte mit einer gebrochenen Zahl multiplizieren. Zum Beispiel ist folgendes eigentlich nicht möglich:

```
b:= b * 0.2;
```

Man behilft sich jetzt mit:

```
b:= (b * 10) div 50;
```

Funktioniert in vielen Fällen, aber nicht wenn das Ergebnis der Multiplikation grösser als der Bereich eines Bytes ist (0..255).

```
b:= (100 * 100) div 250;
```

Die Multiplikation ergibt ein Byte-Overflow und das Gesamt Ergebnis wird absolut falsch. Um das zu umgehen, kann man die ganze Operation in Word ausführen. Das bedeutet jedoch, dass die beteiligten Byte-Variablen in Words konvertiert werden müssen, was zur Vergrösserung des Programms und der Ausführungszeiten führt.

```
Function MulDivByte (a1, a2, d : byte) : byte;
```

Die Funktion errechnet das 16bit Ergebnis der Multiplikation und teilt dies durch den 8bit Divisor. Damit wird ein Overflow Fehler ausgeschlossen. Das Ergebnis der Operation muss allerdings in ein Byte passen.

```
bb:= 100;                       // byte
bb:= MulDivByte (ww, 99, 100);  // -> bb:= bb * 0.99
```

4.10.14 MULDIVINT8

Ähnliche Funktion wie MulDivByte oder MulDivInt allerdings für ShortInt = Int8.

```
Function MulDivInt8 (a1, a2, d : Int8) : Int8;
```

4.10.15 MULDIVINT

Beim Teilen von Integern durch einen gegebenen Wert wird das Ergebnis oft sehr ungenau. Nur mit einem Trick lässt sich ein Integer bzw. Word mit einer gebrochenen Zahl multiplizieren.
Zum Beispiel ist folgendes eigentlich nicht möglich:

```
i:= i * 0.27;
```

Man behilft sich jetzt mit :

```
i:= (i * 100) div 370;
```

Funktioniert in vielen Fällen, aber nicht wenn das Ergebnis der Multiplikation grösser als der Bereich eines Integers ist (+/- 32767).

```
i:= (1000 * 100) div 370;
```

Die Multiplikation ergibt ein Integer-Overflow und das Gesamt Ergebnis wird absolut falsch. Um das zu umgehen, kann man die ganze Operation in LongInt ausführen. Das bedeutet jedoch, dass LongInts importiert werden müssen, was zur Vergrösserung des Programms und der Ausführungszeiten führt.

Function *MulDivInt (a1, a2, d : integer) : integer;*
Function *MulDivInt (a1, a2, d : word) : word;*

Die Funktion errechnet das 32bit Ergebnis der Multiplikation und teilt dies durch den 16bit Divisor. Damit wird ein Overflow Fehler ausgeschlossen. Das Ergebnis der Operation muss allerdings in ein Integer bzw. Word passen.
Der erste Parameter bestimmt die grundsätzliche Operation. Ist er ein Integer, wird mit Vorzeichen gerechnet, ist er ein Word wird ohne gerechnet.

```
ww:= 1000;                  // word
ww:= MulDivInt (ww, 2, 3);  // -> ww:= ww * 0.666

ii:= -1000;                 // integer
ii:= MulDivInt (ii, 100, 125);  // -> ii:= ii * 0.8
```

4.10.16 MulDivLong

Diese Funktion ist identisch mit obigenstehender MulDivInt, ausser dass hier mit LongInt bzw. LongWord gearbeitet wird.

function MulDivLong(a1, a2, d : longint|longword) : longint|longword;

4.10.17 SQUAREDIVBYTE

Function *SquareDivByte (val, divfact : byte) : byte;*

Bildet das Quadrat eines Wertes und teilt dies durch einen zweiten Wert. Der Vorteil hierbei ist, dass ein Überlauf des Quadrat Resultats mit in das Ergebnis einfliesst.

4.10.18 SQUAREDIVINT8

Function *SquareDivInt8 (val, divfact : int8) : int8;*

Bildet das Quadrat eines Wertes und teilt dies durch einen zweiten Wert. Der Vorteil hierbei ist, dass ein Überlauf des Quadrat Resultats mit in das Ergebnis einfliesst.

4.10.19 SQUAREDIVINT

Function *SquareDivInt (val, divfact : word|integer) : word|integer;*

Bildet das Quadrat eines Wertes und teilt dies durch einen zweiten Wert. Der Vorteil hierbei ist, dass ein Überlauf des Quadrat Resultats mit in das Ergebnis einfliesst.

4.10.20 INTEGRATEB

Function *IntegrateB (oldVal, newVal, fact : byte) : byte;*

Integriert einen neuen Wert zu einem schon vorhandenen.
Berechnung: *result:= ((oldVal * fact) + newVal)* ***div*** *(fact + 1);*
Ein Overflow in der Multiplikation kann nicht auftreten, da intern mit dem nächst grösseren Typ gearbeitet wird.

4.10.21 INTEGRATEI8

Function *IntegrateI8 (oldVal, newVal : int8; fact : byte) : int8;*

Integriert einen neuen Wert zu einem schon vorhandenen.
Berechnung: *result:= ((oldVal * fact) + newVal)* ***div*** *(fact + 1);*
Ein Overflow in der Multiplikation kann nicht auftreten, da intern mit dem nächst grösseren Typ gearbeitet wird.

4.10.22 INTEGRATEI

Function *IntegrateI (oldVal, newVal : integer; fact : byte) : integer;*

Integriert einen neuen Wert zu einem schon vorhandenen.
Berechnung: *result:= ((oldVal * fact) + newVal)* ***div*** *(fact + 1);*
Ein Overflow in der Multiplikation kann nicht auftreten, da intern mit dem nächst grösseren Typ gearbeitet wird.

4.10.23 INTEGRATEW

Function *IntegrateW (oldVal, newVal : word; fact : byte) : word;*

Integriert einen neuen Wert zu einem schon vorhandenen.
Berechnung: *result:= ((oldVal * fact) + newVal)* ***div*** *(fact + 1);*
Ein Overflow in der Multiplikation kann nicht auftreten, da intern mit dem nächst grösseren Typ gearbeitet wird.

4.10.24 Even

Wert auf geradzahlig testen. Funktion gibt true resp. False zurück.
Nur für byte, int8, word, integer, longword, longint, word64, int64

Function *Even (x : type) : boolean;*

If *Even(V1)* ***then*** *..*

4.10.25 ODD

Wert auf un-geradzahlig testen. Funktion gibt true resp. False zurück.
Nur für byte, int8, word, integer, longword, longint, word64, int64

Function *Odd (x : type) : boolean;*

If *Odd (V1)* ***then*** *..*

4.10.26 PARITY

ergibt die Parität eines Bytes oder Chars. odd = true

Function *Parity (****const*** *b : byte|char) : boolean;*

Bool:= Parity (bb);

4.10.27 ISPOWOFTWO

Funktion, prüft die Zahl "n" ob sie eine Zweier Potenz repräsentiert. Gültige Parameter sind Byte..LongInt.

Function *IsPowOfTwo (n : type) : boolean;*

4.10.28 SIGN

Funktion, liefert das Vorzeichen (sign) einer Zahl als Boolean:

Function *Sign (**const** num : integer[int8, longint*, *word64, int64, fix64, float]) : boolean;*

Ist das Argument positiv ergibt die Funktion ein true, ansonsten ein false.

4.10.29 SGN

Funktion, liefert das Vorzeichen (sign) einer Zahl. Das Ergebnis ist vom gleichen Typ wie die Zahl:

Function *Sgn (**const** num : integer, int8, longint*, *word64, int64, fix64, float) : type;*

Ist das Argument > 0 ergibt die Funktion ein '1', ist das Argument '0' so ist das Resultat '0', ansonsten '-1'.

4.10.30 PRED

Die Funktion gibt den nächst kleineren Wert einer Variablen zurück. Der Typen muss ordinal sein, z.B. byte, word, integer. Die Grenzen des Typs werden nicht überschritten, es erfolgt kein “wrap“. Der Vorgänger des Byte 0 ist daher immer 0.

Function *Pred (x : type) : type;*

x:= Pred (y);

4.10.31 SUCC

Die Funktion gibt den nächst grösseren Wert einer Variablen zurück. Der Typen muss ordinal sein, z.B. byte, word, integer. Die Grenzen des Typs werden nicht überschritten, es erfolgt kein “wrap“. Der Nachfolger des Byte 255 ist daher immer 255.

Function *Succ (x : type) : type;*

x:= Succ (y);

4.10.32 MIN

ergibt den kleinsten möglichen Wert des Typs.

Function *Min (x: type) : type;*

x:= min (a);

4.10.33 MAX

ergibt den grössten möglichen Wert des Typs.

Function *Max (x: type) : type;*

x:= max (a);

4.10.34 SIZEOF

Funktion gibt den Speicherbedarf eines Objekts in Bytes zurück.

```
Function SizeOf (x : type) : word;

n:= SizeOf (a);
n:= SizeOf (st1);       {gesamter Speicherplatz eines Strings}
```

4.10.35 BitCountOf

Funktion gibt die Anzahl der auf 1 gesetzten Bits einer Ordinalen zurück.

```
Function BitCountOf (x : ordinal) : byte;

n:= BitCountOf (a);
```

4.10.36 ADDR

```
p:= Addr (a);           {Adresse der Speicherstelle}
```

Als Operanten sind nur Variablen, Prozeduren und Funktionen zulässig, denn nur diese haben auch eine physikalische Adresse. Das Resultat der Funktion ist ein **typisierter** Pointer. Nach der Operation enthält ‘p’ die Adresse von ‘a’. Das Ziel ist normalerweise ein Pointer.

4.11 System Library - Fix64

Erstellt mit der hervorragenden Hilfe von User Avra.

Der Fix64 Typ muss imortiert werden:
from system **import** ..., Fix64;

Note:
Ausführungs Zeiten für die wesentlichen Operationen (@16MHz):

F1 + F2	speed : 7usec	
F1 - F2	speed : 8usec	
F1 * F2	speed : 250usec	
F1 / F2	speed : 500usec	
Sqr	speed : 250usec	
Sqrt	speed : 0.5msec	result : 5 frac digits
Fix64Sqrt	speed : 2.2msec	result : 9 frac digits
Delphi Sin(3.0)	result : 0.14112000806	
AVRco Sin(3.0)	result : 0.141120008	5..8msec

Achtung: die Funktion Fix64Sqrt und alle Trigonometrie Funktion brauchen bis 60 Bytes auf dem Frame!

4.11.1 FIX64 Unit

Compiler Schalter für Fix64 Library: ***{$Define FIX64_USE_PRECISE_SQRT}***.
Ist der Schalter aktiv, dann wird für alle Fix64 Funktionen die SQRT benutzen müssen (trigonometrische Funktionen) die hochpräzise aber langsame interne SQRT Funktion benutzt. Ohne diesen Schalter wird die zwar viel schnellere aber unpräzisere SQRT Funktion benutzt.
Precise SQRT 9 frac digits 81435 cycles
Standard SQRT 5 frac digits 9333 cycles

Nachfolgende komplexe Fix64 Funktionen sind in einer Unit enthalten, die bei Bedarf ebenso anzugeben ist:
uses uFix64;

Weniger komplexe Funktionen (wie die Standard mathematischen Funktionen) sind Teil des Compilers und somit auch in der *Standard Version* des AVRco verfügbar, siehe Kapitel “Operatoren” und “System Library – Standard”

Soll diese Unit nicht verwendet werden, dann ist hier eine Funktion um Fix64 in Float zu wandeln und umgekehrt:

```
type
  TFix64Overlay = record
                    fix        : fix64;
                    i  [@fix+4]: longint;
                    f  [@fix]  : longword;
                  end;

function FloatToFix64(a : float) : fix64;
var
  Tmp : TFix64Overlay;
begin
  Tmp.i:= Trunc(a);
  if (a <= -2.3283064365386962890625E-10) and (Frac(a) <> 0) then
    Dec(Tmp.i);
  endif;
  Tmp.f:= Trunc(Frac(a) * (float($FFFFFFFF) + 1));
  return(Tmp.fix);
end;
```

```
function Fix64ToFloat(a: fix64) : float;
var
  Tmp[@a] : TFix64Overlay;
begin
  return(float(Tmp.i) + float(Tmp.f) / (float($FFFFFFFF) + float(1)));
end;
```

4.11.2 FIX64 und Delphi

Die folgenden Funktionen sind für die Konvertierung von Fix64 Typen innerhalb von Delphi:

```
const
  FIX_ONE = $100000000; // 1.0 is neutral in multiplication and division of fixed
                        // point numbers
type
  fix64 = int64;

function ExtendedToFix64(const a: extended): fix64;
begin
  result := Round(a * FIX_ONE);
end;

function Fix64ToExtended(const a: fix64): extended;
begin
  result := a;
  result := result / FIX_ONE;
end;
```

4.11.3 mathematische Funktionen

Function *Fix64MulLong (const a: fix64; const b: longint): fix64;*
liefert das Produkt eines Fix64 und eines Long Integer.

Function *Fix64MulInt (const a: fix64; const b: integer): fix64;*
liefert das Produkt eines Fix64 und eines Integer.

Function *Fix64DivLong (const a: fix64; const b: longint): fix64;*
liefert den Quotient aus Fix64 dividiert durch Long Integer.

Function *Fix64DivInt (const a: fix64; const b: integer): fix64;*
liefert den Quotient aus Fix64 dividiert durch Integer.

Function *Fix64Mod (const a, modulus: fix64): fix64;*
liefert "a mod *modulus*". Beide Argumente sind vom Typ Fix64.

Function *Fix64ModInt (const a: fix64; modulus: integer): fix64;*
liefert "a mod *modulus*". *modulus* ist hier vom Typ Integer.

Function *Fix64Odd (const a: fix64): boolean;*
liefert *true* wenn das Argument ungerade ist.

Function *Fix64Even (const a: fix64): boolean;*
liefert *true* wenn das Argument gerade ist.

Function *Fix64Sqrt(const a: fix64): fix64;*
liefert die Quadratwurzel des Arguments. Sehr **schnell**, weniger präzis

Function *Fix64SqrtEx(const a: fix64): fix64;*
liefert die Quadratwurzel des Arguments. Langsam, sehr **präzis**

Function *Fix64Integrate (const aold, anew: fix64; const factor: byte): fix64;*
Integriert einen neuen Wert zu einem schon vorhandenen.

Berechnung: *result:= ((aold * factor) + anew)* ***div*** *(fact + 1);*
Ein Überflauf in der Multiplikation kann hier auftreten! Vorsicht mit grossen Zahlen !

4.11.4 Konvertierungsfunktionen

Function *Fix64ToLongInt (const a: fix64): longint;*
Type Cast: konvertiert einen Fix64 in einen Long Integer.

Function *IntToFix64 (const a: longint): fix64;*
Diese Funktion wandelt einen ordinal Wert (Byte…LongInt) in ein Fix64 um.

Function *Fix64ToIn t (const a: fix64): integer;*
Type Cast: konvertiert einen Fix64 in einen Integer.

Function *FloatToFix64 (const a: float): fix64;*
Type Cast: konvertiert einen Float in einen Fix64.

Function *Fix64ToFloat (const a: fix64): float;*
Type Cast: konvertiert einen Fix64 in einen Float.

4.11.5 Vergleichsfunktionen

Function *Fix64ValueInTolerance (const a, aref, atol: fix64): boolean;*
vergleicht den Inhalt von „a“ mit der Grenze “vmin“, die aus (aref - atol) gebildet wird und "vmax", die aus (aref + atol) gebildet wird.
Überschreitet der Wert von value eine der Grenzen, so wird das Ergebnis false, ansonsten true.

Function *Fix64ValueInToleranceP (const a, aref, atol: fix64): boolean;*
vergleicht den Inhalt von “a“ mit der Grenze „vmin“, die aus (aref - (aref div 100) * atol) und "vmax", die aus (aref + (aref div 100) * tol) gebildet wird.
Überschreitet der Wert von value eine der Grenzen, so wird das Ergebnis false, ansonsten true.
Der Wert von “atol” muss im Bereich 0..100 liegen, da es sich um einen Prozentsatz handelt.
Diese Funktion ist identisch zu "*Fix64ValueInTolerance*". Nur dass hier die Toleranz nicht absolut, sondern relativ in Prozent angegeben wird.

4.11.6 Logarithmen, etc.

Function *Fix64IsPowOfTwo (const a: fix64): boolean;*
liefert *true* wenn “a” eine 2’er Potenz ist.

Function *Fix64Exp (const a: fix64): fix64;*
liefert “e” hoch “a”. “e” ist die Basis des natürlichen Logarithmus (2.71828...).

Function *Fix64Ln (const a: fix64): fix64;*
Function *Fix64LogN (const a: fix64): fix64;*
beide Funktionen liefern den natürlichen Logarithmus von a zur Basis e.
“*e”* ist die Basis des natürlichen Logarithmus (2.71828...).

Function *Fix64Log10 (const a: fix64): fix64;*
liefert den Logarithmus von a zur Basis 10.

Function *Fix64Log (const a, base: fix64): fix64;*
liefert den Logarithmus von a zur Basis “*base*”.

Function *Fix64Power (const base, exponent: fix64): fix64;*
liefer *base ^ exponent*. “*base*” und “*exponent*” sind vom Typ Fix64.

Function *Fix64PowerInt (const base: fix64; const exponent: integer): fix64;*

liefer *base* ^ *exponent*. “*base*” ist vom vom Typ Fix64, “*base*” vom Typ Integer.

4.11.7 Trigonometrie

Function *Fix64Sin (const radians: fix64): fix64;*
liefert den Sinus des Arguments (Argument=Winkel im Bogenmaß).

Function *Fix64SinD (const degrees: fix64): fix64;*
liefert den Sinus des Arguments (Argument=Winkel in Grad).

Function *Fix64Cos (const radians: fix64): fix64;*
liefert den Cosinus des Arguments (Argument=Winkel im Bogenmaß).

Function *Fix64CosD (const degrees: fix64): fix64;*
liefert den Cosinus des Arguments (Argument=Winkel in Grad).

Function *Fix64Tan (const radians: fix64): fix64;*
liefert den Tangens des Arguments (Argument=Winkel im Bogenmaß).

Function *Fix64TanD (const degrees: fix64): fix64;*
liefert den Tangens des Arguments (Argument=Winkel in Grad).

Function *Fix64Cot (const radians: fix64): fix64;*
liefert den Cotangens des Arguments (Argument=Winkel im Bogenmaß).

Function *Fix64CotD (const degrees: fix64): fix64;*
liefert den Cotangens des Arguments (Argument=Winkel in Grad).

Function *Fix64Sec (const radians: fix64): fix64;*
liefert den Secans des Arguments (Argument=Winkel im Bogenmaß).

Function *Fix64SecD (const degrees: fix64): fix64;*
liefert den Secans des Arguments (Argument=Winkel in Grad).

Function *Fix64Csc (const radians: fix64): fix64;*
liefert den Cosecans des Arguments (Argument=Winkel im Bogenmaß).

Function *Fix64CscD (const degrees: fix64): fix64;*
liefert den Cosecans des Arguments (Argument=Winkel in Grad).

Function *Fix64Sinh (const a: fix64): fix64;*
liefert den Sinus Hyperbolicus von a.

Function *Fix64Cosh (const a: fix64): fix64;*
liefert den Cosinus Hyperbolicus von a.

Function *Fix64Tanh (const a: fix64): fix64;*
liefert den Tangens Hyperbolicus von a.

Function *Fix64ArcSin (const sine: fix64): fix64;*
liefert den Winkel im Bogenmaß für einen vorgegebenen Sinus diese Winkels.

Function *Fix64ArcSinD (const sine: fix64): fix64;*
liefert den Winkel in Grad für einen vorgegebenen Sinus diese Winkels.

Function *Fix64ArcCos (const cosine: fix64): fix64;*
liefert den Winkel im Bogenmaß für einen vorgegebenen Cosinus diese Winkels.

Function *Fix64ArcCosD (const cosine: fix64): fix64;*
liefert den Winkel in Grad für einen vorgegebenen Cosinus diese Winkels.

***Function** Fix64ArcTan (const tangent: fix64): fix64;*
liefert den Winkel im Bogenmaß für einen vorgegebenen Tangens diese Winkels.

***Function** Fix64ArcTanD (const tangent: fix64): fix64;*
liefert den Winkel in Grad für einen vorgegebenen Tangens diese Winkels.

***Function** Fix64ArcTan2 (const y, x: fix64): fix64;*
Abwandlung der Arctangens Funktion: für beliebige reelle Argumente x und y, die nicht beide Null sind, liefert arctan2(x,y) den Winkel im Bogenmaß zw. der positiven x-Achse der Ebene und dem durch die Koordinaten (x,y) gegebenen Punkt darauf.

***Function** Fix64ArcTan2D (const y, x: fix64): fix64;*
Abwandlung der Arctangens Funktion: für beliebige reelle Argumente x und y, die nicht beide Null sind, liefert arctan2(x,y) den Winkel in Grad zw. der positiven x-Achse der Ebene und dem durch die Koordinaten (x,y) gegebenen Punkt darauf.

***Function** Fix64ArcCot (const cotangent: fix64): fix64;*
liefert den Winkel im Bogenmaß für einen vorgegebenen Cotangens diese Winkels.

***Function** Fix64ArcCotD (const cotangent: fix64): fix64;*
liefert den Winkel in Grad für einen vorgegebenen CotTangens diese Winkels.

***Function** Fix64ArcSec (const secant: fix64): fix64;*
liefert den Winkel im Bogenmaß für einen vorgegebenen Secans diese Winkels.

***Function** Fix64ArcSecD (const secant: fix64): fix64;*
liefert den Winkel in Grad für einen vorgegebenen Secans diese Winkels.

***Function** Fix64ArcCsc (const cosecant: fix64): fix64;*
liefert den Winkel im Bogenmaß für einen vorgegebenen Cosecans diese Winkels.

***Function** Fix64ArcCscD (const cosecant: fix64): fix64;*
liefert den Winkel in Grad für einen vorgegebenen Cosecans diese Winkels.

***Function** Fix64ArcSinh (const a: fix64): fix64;*
liefert den Winkel im Bogenmaß für einen vorgegebenen hyperbolischen Sinus diese Winkels.

***Function** Fix64ArcCosh (const a: fix64): fix64;*
liefert den Winkel im Bogenmaß für einen vorgegebenen hyperbolischen Cosinus diese Winkels.

***Function** Fix64ArcTanh (const a: fix64): fix64;*
liefert den Winkel im Bogenmaß für einen vorgegebenen hyperbolischen Tangens diese Winkels.

***Function** Fix64RadToDeg (const radians: fix64): fix64;*
liefert das Gradmaß des Arguments (Argument im Bogenmaß).

***Function** Fix64DegToRad (const degrees: fix64): fix64;*
liefert das Bogenmaß Arguments (Argument in Grad).

***Function** Fix64Quadrant (const radians: fix64): byte;*
liefert den Quadrant (1..4) des Winkels (Winkel im Bogenmaß).

Note
Die Unit uFix64 enthält jede Menge mächtige Funktionen die auch erheblichen Code erzeugen. Es ist daher zu empfehlen, dass hierzu der **Merlin Optimiser** benutzt werden sollte.

Beispiele
Ein Beispiel Programm für die Math Funktionen ist im Verzeichnis **..\E-Lab\AVRco\Demos\Fix64**
Ein Beispiel Programm für die Trigonometrie Funktionen ist im Verzeichnis **..\E-Lab\AVRco\Demos\Fix64**

4.12 System Library - Bit Verarbeitung

Bitverarbeitung ist ein wesentlicher Teil von Controller Anwendungen. Pascal kennt hier eigentlich nur die BIT-Maske, z.B. ***if*** *(x* ***and*** *1) > 0* ***then*** *..*

Abhilfe schafft hierbei wiederum die von Modula-2 her bekannte Bitverarbeitung mit *incl (bit), excl (bit)* und *bit(bit).* Zur Vereinfachung wurde weiterhin die *TYPE BIT* eingeführt (siehe Beschreibung der *type bit* weiter oben). Dieser Typ kann für alle 5 Bit-Funktionen verwendet werden (Incl, Excl, Toggle, SetBit, Bit).

Die Deklaration einer Bit-Variablen besteht **immer** aus zwei Stufen: zuerst muss die Speicherstelle bzw. Variable, in der das Bit liegt, deklariert werden. Dies geschieht mit einem normalen VAR-Statement:

```
var   Leds[$05] : byte;        oder
      bits16    : word;
```

Der **Typ der Speicherstelle** (byte, word) bestimmt dann auch, ob 8 oder 16 Bits zur Verfügung stehen. Nach der Deklaration einer allgemeinen Variablen erfolgt dann die eigentliche BIT-Deklaration. Der erste Parameter bezeichnet dabei eine Speicherstelle, der zweite Parameter das entsprechende Bit dieser Speicherstelle. Es können auch symbolische Parameter eingesetzt werden.

```
Const  LedBit2 = 3;
var    port6[6] : byte;
       Led2[@port6, Led2Bit] : bit;

if BIT(Led2) then ...
Toggle(Led2);
```

Bits können auch innerhalb des Programms dynamisch generiert werden.

```
Toggle (Leds, 3);          {Bit3  innerhalb 8  Bits}
Incl (bits16, 12);         {Bit12 innerhalb 16 Bits}
```

Weiterhin können Bits wie bei 8051 Tools üblich mit "variable.bit" adressiert werden.

```
PortB.0:= true;
bool:= PortA.5;
Toggle(PortC.2);
```

"BIT" muss allerdings eine Konstante im Bereich 0..7 sein.
Funktioniert auch mit lokalen Variablen in Prozeduren/Funktionen.

Bei allen Bit-Schreib-Operationen ist hier auch eine "0" oder "1" statt "False" oder "True" möglich.

```
PortB.0:= 0;
BitX:= 1;
```

4.12.1 INCL

Bit setzen

```
Incl (port6, 3);           {Diese Anweisung ist identisch mit}
Incl (Led2);               {dieser Anweisung bei obiger Definition}
```

Unter bestimmten Bedingungen ist ein Einsatz der Prozedur **SetBit** vorzuziehen.

INCL bei Bitsets:

Procedure *Incl (SrcDest : BitSet; op : BitSet);*
"SrcDest" ist das zu verändernde BitSet ist und "op" enthält die zu verändernden Bits.
"op" kann dabei entweder eine BitSet Variable vom gleichen Typ sein oder eine BitSet Konstante "[aa, bb, cc, ...]" vom gleichen Typ.

XMega und atomic Port Manipulation
Bei den XMegas kann zum setzen einer Bit-Gruppe in einem Port das jeweilige OUTSET oder DIRSET Register verwendet werden. Dies muss dem Compiler aber eindeutig mitgeteilt werden mit “PortX.outset:= byte“ oder “PortX.dirset:= byte“:

```
PortA.outset:= $13;
PortB.dirset:= bb;
```

4.12.2 EXCL

Bit rücksetzen

```
Excl (Leds, a);          {Leds ist eine Byte-Variable}
Excl (b);                {Symbol b ist eine Bit Deklaration}
```

Unter bestimmten Bedingungen ist ein Einsatz der Prozedur **SetBit** vorzuziehen.

EXCL bei Bitsets:

Procedure *Excl (SrcDest : BitSet; op : BitSet);*
"SrcDest" ist das zu verändernde BitSet ist und "op" enthält die zu verändernden Bits. "op" kann dabei entweder eine BitSet Variable vom gleichen Typ sein oder eine BitSet Konstante "[aa, bb, cc, ...]" vom gleichen Typ.

XMega und atomic Port Manipulation
Bei den XMegas kann zum rücksetzen einer Bit-Gruppe in einem Port das jeweilige OUTCLR oder DIRCLR Register verwendet werden. Dies muss dem Compiler aber eindeutig mitgeteilt werden mit “PortX.outclr:= byte“ oder “PortX.dirclr:= byte“:

```
PortA.outclr:= $13;
PortB.dirclr:= bb;
```

4.12.3 TOGGLE

Invert Bit, als Bit, im BitSet, komplettes Byte oder Boolean

```
Toggle (Leds, 3);  //bit
Toggle (Led2);     // bit
Toggle(byte);
Toggle(Boolean);
```

TOGGLE bei Bitsets:

Procedure *Toggle (SrcDest : BitSet; op : BitSet);*
"SrcDest" ist das zu verändernde BitSet ist und "op" enthält die zu verändernden Bits. "op" kann dabei entweder eine BitSet Variable vom gleichen Typ sein oder eine BitSet Konstante "[aa, bb, cc, ...]" vom gleichen Typ.

XMega und atomic Port Manipulation
Bei den XMegas kann zum invertieren einer Bit-Gruppe in einem Port das jeweilige OUTTGL Register verwendet werden. Dies muss dem Compiler aber eindeutig mitgeteilt werden mit “PortX.toggle:= byte“.

```
PortA.toggle:= $13;
```

4.12.4 SETBIT

Bit setzen/rücksetzen
SetBit (BitType, boolean);

SetBit setzt, abhängig vom Parameter boolean, das im 1.Parameter beschriebene Bit auf 1 oder 0. Abhängig von der benutzten Variablen kann auf 8 oder 16 Bits zugegriffen werden. Die Definition der Type BIT ist unter dem Type BIT zu finden.

SetBit ist eine Kombination aus *Incl(bit)* und *Excl(bit)* und ist damit **langsamer** und von der Anzahl der Assembler-Befehle wesentlich **grösser** als z.B. ein einzelnes *Incl(bit)*. SetBit sollte, wenn Geschwindigkeit und/oder Programmgrösse wichtig sind, nur verwendet werden um die folgende Konstruktion zu ersetzen:

```
if boolean then
  Incl (bit);
else
  Excl (bit);
endif;
```

besser, schneller und kürzer ist hier:
SetBit (bit, boolean);

aber statt folgender Konstruktion:
SetBit (bit, true);

sollte besser diese Konstruktion verwendet werden:
Incl (bit);

```
const   a  = 7;
        c  = 5;

var     v1     = byte;
        Led1   = [@V1, a];

Function TestB : boolean;
begin
  ...
  return(x);
end;

SetBit (v1, c, TestB);
SetBit (Led1, TestB);
```

SETBIT bei Bitsets:

Procedure *SetBit (SrcDest : BitSet; op : BitSet|Enum; bool : boolean);*
"SrcDest" ist das zu verändernde BitSet ist und "op" enthält die zu verändernden Bits. "op" kann dabei entweder eine BitSet Variable vom gleichen Typ sein oder eine BitSet Konstante "[aa, bb, cc, ...]" vom gleichen Typ. Es kann hier aber auch die zugrunde liegende Enumeration verwendet werden.

```
SetBit (myBitSet, [a, b, c], true);
SetBit (myBitSet, myBitSet1, true);
SetBit (myBitSet, a, false);
```

4.12.5 BIT

Bit testen

```
if Bit (a, 0) then ... ; endif;
if Bit (b) then ... ; endif;
```

4.13 System Library - Diverse System Funktionen

4.13.1 SYSTEM_RESET

Setzt das ganze System zurück und startet es neu. Der Effekt ist fast derselbe wie bei einem Hardware Reset, ausser dass bei einem Hardware Reset manche Bits in Steuerregistern einen bestimmten Wert annehmen, was bei System_Reset nicht der Fall ist. Hier erfolgt eine Sperrung der Interrupts und ein Sprung an den Programmanfang.
Xmega
Es wird ein echter Hardware Reset erzeugt.

Wurde die Spezial Prozedur **System_Init** definiert, wird diese natürlich auch abgearbeitet.

```
System_Reset;
```

4.13.2 DELAY

4.13.2.1 mDelay

Software Delay in msec. Die Prozedur kehrt erst wieder nach Ablauf von x msec zurück. Der Übergabe-Parameter muss zwischen 1 und 65000 (word) liegen. Systembedingt liegt die Genauigkeit bei +/-20%. Für genaues Timing sollte der Typ ***SysTimer*** oder ***SysTimer8*** benutzt werden.
Bei MultiTasking sollte besser ***Sleep*** benutzt werde, um keine Rechenzeit zu verschenken.

```
Procedure mDelay (d : word);

mDelay (100);                {100 msec warten}
```

4.13.2.2 uDelay

Software Delay in usec x 10. Die Prozedur kehrt erst wieder nach Ablauf von n usec x 10 zurück. Der Übergabe-Parameter muss zwischen 1 und 255 (byte) liegen. Systembedingt liegt die Genauigkeit bei +/-20%. Bei CPU-Clocks unter 2MHz nimmt die Genauigkeit nocheinmal extrem ab.
Für genaues Timing sollte der Typ **SysTimer** oder **SysTimer8** benutzt werden.

```
Procedure uDelay(d : byte);

uDelay(10);                  {100 usec warten}
```

4.13.2.3 uDelay_1

Software Delay in 1 usec. "*uDelay_1*" kann natürlich nur einigermassen exakt funktionieren wenn der Prozessor Clock >= 8MHz ist. Dabei auftretende Interrupts verfälschen natürlich das Ergebnis auch erheblich.

```
Procedure uDelay_1 (d : byte);
```

4.13.2.4 sDelay

Software Delay in CPU Zyklen. Die Prozedur kehrt erst wieder nach Ablauf von n CPU Zyklen zurück. Der Übergabe-Parameter muss zwischen 1 und 255 (byte) liegen. Systembedingt kann keine Genauigkeit angegeben werden. Nur für extrem kurze Delays im Mikro-Sekunden Bereich geeignet.

```
Procedure sDelay (d : byte);

sDelay (10);                 {ca. 10 Zyklen warten}
```

4.13.3 SYSTIMER

4.13.3.1 SetSysTimer

Die Prozedur setzt einen ***SysTimer*** auf den Übergabe Wert. Dazu wird der Interrupt gesperrt und anschliessend wieder freigegeben. Ein kontinuierliches Beschreiben von SysTimern ist deshalb zu vermeiden.

```
var  Timer1 : SysTimer8;          {Variable vom Typ SysTimer 8bit}

SetSysTimer (Timer1, 50);
```

Auch für SysTimer im UpCount Mode.

4.13.3.2 SetSysTimerM

```
Procedure SetSysTimerM (tm : SysTimer; time : byte|word);
```

Damitkönnen Millisekunden Werte übergeben werden. Dabei bleibt es natürlich intern bei der SysTick Auflösung, d.h. mit SysTick 10msec ergibt der Parameter 27 dann einen Timer Wert von 30msec. (TimerValue:= time div SysTickTime).
Beim SysTimer8 werden auch gebrochene SysTick Werte richtig gerechnet, z.B. 2.5.
Das Ergebnis der Funktion (in SysTicks) darf hier aber 255 Ticks nicht überschreiten.
Beim SysTimer (16bit) werden gebrochene SysTick Zeiten gerundet.

4.13.3.3 GetSysTimer

Die Prozedur liest einen ***SysTimer*** und gibt den aktuellen Wert zurück. Dazu wird der Interrupt gesperrt und anschliessend wieder freigegeben. Ein kontinuierliches Lesen von SysTimern ist deshalb zu vermeiden.

```
var  Timer1 : SysTimer;           {Variable vom Typ SysTimer 16bit}

Ww:= GetSysTimer (Timer1);
```

Auch Für SysTimer im UpCount Mode.

4.13.3.4 ResetSysTimer

Die Prozedur setzt einen **SysTimer** auf den Wert 0. Dazu wird der Interrupt gesperrt und anschliessend wieder freigegeben. Ein kontinuierliches Rücksetzen von SysTimern ist deshalb zu vermeiden.

```
var  Timer1 : SysTimer8;          {Variable vom Typ SysTimer 8bit}

ResetSysTimer (Timer1);
```

Auch Für SysTimer im UpCount Mode.

4.13.3.5 IsSysTimerZero

Wurde der SystemTick importiert und sind ***SysTimer*** definiert, so könnte der Timer direkt auf Null gepollt werden. Allerdings wird dabei bei jedem Zugriff der Interrupt gesperrt und wieder freigegeben, was enorm Rechenzeit verschlingt und das Interrupt System langsam macht. Der bessere Weg ist die Abfrage des Timers mit der Function “isSysTimerZero”. Das ist schneller und Interrupts werden nicht gesperrt. Ist der Timer = 0 gibt die Function true zurück, ansonsten false.

```
var  Timer1 : SysTimer;              {Variable vom Typ SysTimer 16bit}

SetSysTimer (Timer1, 50000);

repeat until isSysTimerZero (Timer1);
```

Irrelevant für SysTimer im UpCount Modus.

4.13.4 LOWER

Die Funktion gibt den kleineren zweier Werte zurück. Die Typen der beiden Argumente müssen identisch sein, z.B. Byte, Int8, word, integer, Longint, Longword, Int64, Word64, Fix64, float.

```
Function Lower (x, y : type) : type;

x:= lower (y, z);
```

4.13.5 HIGHER

Die Funktion gibt den grösseren zweier Werte zurück. Die Typen der beiden Argumente müssen identisch sein, z.B. Byte, Int8, word, integer, Longint, Longword, Int64, Word64, Fix64, float.

```
Function Higher (x, y : type) : type;

x:= higher (y, z);
```

4.13.6 WITHIN

Funktion, prüft gegen Grenzwerte:
Die Funktion prüft eine Zahl gegen zwei Grenzwerte ab. Der erste Wert ist die untere Grenze, der zweite Wert der zu prüfende und der dritte ist die obere Grenze. Fällt der zu prüfende Wert innerhalb der Grenzen, so ist er das Ergebnis. Ist er zu klein, ist das Ergebnis der untere Grenzwert, ansonsten der obere Grenzwert. Alle Typen müssen identisch sein, z.B. Byte, Int8, word, integer, Longint, Longword, Int64, Word64, Fix64, float.

```
Function WithIn (lo, x, hi : type) : type;

x:= WithIn (low, a, high);
```

4.13.7 VAL

In Standard Pascal wandelt die Prozedur *Val* einen String-Wert in seine numerische Darstellung um.
Diese generelle Prozedur ist sehr komplex und würde sehr viel Code im Flash beanspruchen.
Aus diesem Grund ist *val* im AVRco Pascal **nicht** implementiert, sondern wurde durch mehrere kleinere Prozeduren ersetzt. Siehe Kapitel " System Library –Standard, Character und String Funktionen".

4.13.8 Block Funktionen

4.13.8.1 FILLBLOCK

Füllt bzw. löscht einen Speicherbereich mit einem Byte oder Char. Das Ziel kann eine beliebige Variable sein. Eine Bereichsüberprüfung findet nicht statt. ***Fill*** kann ein Byte oder ein Char sein.

Procedure *FillBlock (p : pointer; cnt : word; fill : byte);*

```
FillBlock (@Start, len, fill);
FillBlock (@array1, SizeOf (array1), 0);
FillBlock (@w1, 2, '1');
```

4.13.8.2 FILLRANDOM

Füllt einen Speicherbereich mit einem Zufallswert. Das Ziel kann eine beliebige Variable sein. Eine Bereichsüberprüfung findet nicht statt. Die Random Funktion muss vom System importiert sein.

Procedure *FillRandom (p : pointer; cnt : word);*

```
FillRandom (@Start, lenl);
FillRandom (@array1, SizeOf (array1));
FillRandom (@w1, 2);
```

4.13.8.3 COPYBLOCK

Kopiert einen Speicherbereich in einen anderen. Die Blöcke sollten nicht überlappen. Eine Bereichsprüfung findet nicht statt. Source und Dest können beliebige Variablen sein.

Procedure *CopyBlock (Source, Dest : pointer; len : word);*

```
CopyBlock(@array1, @array2, SizeOf (array1));
CopyBlock(@w1, @w2, 2);
```

4.13.8.4 COPYBLOCKREVERSE

Kopiert einen Speicherbereich in einen anderen. Die Blöcke sollten nicht überlappen. Eine Bereichsprüfung findet nicht statt. Source und Dest müssen ***iData*** Variablen sein. Dabei wird die Reihenfolge der Bytes vertauscht, das letzte wird das erste, das erste wird das letzte etc.

Procedure *CopyBlockReverse(Source, Dest : pointer; len : word);*

```
CopyBlockReverse(@array1, @array2, SizeOf (array1));
CopyBlockReverse(@w1, @w2, 2);
```

4.13.8.5 COMPAREBLOCK

Damit können zwei Speicherbereiche miteinander verglichen werden. Bei gleichem Inhalt wird ein true zurückgegeben, ansonsten ein false.

Function *CompareBlock (**const** addr1, addr2 : pointer; **const** len : word) : boolean;*

Mögliche Bereiche:

RAM	<-->	RAM
EEprom	<-->	EEprom
ROM	<-->	RAM
ROM	<-->	EEprom
RAM	<-->	EEprom

4.13.9 Pointer Zugriff ausserhalb des linearen Adressbereichs

Pointer sind immer 16 Bit breit und können keine weiteren Informationen über den referenzierten Speicherbereich enthalten. Somit impliziert ein Pointer immer den Zugriff in den linearen CPU Adressbereich von $0000..$FFFF.
Soll jedoch ein Zugriff über einen Pointer in das Flash, das EEProm oder in ein Banked Devive erfolgen, müssen die folgenden Funktionen verwendet werden:

4.13.9.1 FlashPtr

Weist den Compiler an, dass die Pointer Manipulation als Ergebnis einen Zugriff ins Flash haben soll.

```
Function FlashPtr (p:pointer): pointer;

Ptr1 := @FlashByte;
bb := FlashPtr (Ptr1)^;
```

4.13.9.2 EEPromPtr

Weist den Compiler an, dass die Pointer Manipulation als Ergebnis einen Zugriff ins EEProm haben soll.

```
Function EEPromPtr (p:pointer): pointer;

Ptr1 := @EEPromByte;
bb := EEPromPtr (Ptr1)^;
```

4.13.9.3 UsrDevPtr

Weist den Compiler an, dass die Pointer Manipulation als Ergebnis einen Zugriff ins User Device haben soll.

```
Function UsrDevPtr (p:pointer): pointer;

Ptr1 := @UsrDevByte;
bb := UsrDevPtr (Ptr1)^;
```

4.13.9.4 BankDevPtr

Weist den Compiler an, dass die Pointer Manipulation als Ergebnis einen Zugriff ins Banked Device haben soll.

```
Function BankDevPtr (b:byte; p:pointer): pointer;

Ptr1 := @BankDevByte;
bb := BankDevPtr (2, Ptr1)^;          //Bank #2
```

4.13.10 FLUSHBUFFER

Löscht den Inhalt eines Buffers der seriellen Schnittstellen. Als Argument ist *RxBuffer, TxBuffer, RxBuffer1, TxBuffer1, RxBuffer2, TxBuffer2, RxBuffer3, TxBuffer3* möglich.

```
Procedure FlushBuffer (Buffer : tBuffer);

FlushBuffer (RxBuffer);
```

4.13.11 CRC Checksumme

4.13.11.1 CRC CHECK

Ein Datenblock mit dem angehängten CRC-Word ergibt als Ergebnis null, wenn dieser Block durch die CRC-Funktion überprüft wird.

Function *CRCcheck (p : pointer; count : word) : word;*

Bemerkungen:
Entspricht dem CCITT Standard: CrcCCITT = x^16+x^12+x^5+1 mit einem Startwert von $0810.
Kann CRC vom RAM, EEPROM und FLASH erzeugen kann. Bedingung dafür ist, dass der bereitgestellte Pointer in den Speicher mit dem Adress-Operator gebildet wird:

xx:= CRCcheck (@EEprom, sizeOf (EEprom));

4.13.11.2 CRC STREAM

Anwendbar für Input/Output Streams von Byte und Char. Hiermit ist es möglich kontinuierlich Bytes in eine CRC Checksumme einfliessen zu lassen und am Ende die 16bit CRC Summe auszuwerten.
Für XMegas bitte auch das Standart Treiber Handbuch beachten.

Implementation
Wie üblich muss der Treiber importiert werden:

Import *SysTick, CRCstream, ...*

Prozeduren und Funktionen
Der Treiber stellt drei Funktionen zur Verfügung:

Procedure *CRCstreamInit (seed : word);*

Hiermit wird der CRC Start Wert auf 0 gesetzt und mit "seed" der Initialwert übergeben, normalerweise $0810. Nach dem Aufruf dieser Init Funktion können jetzt mit der

Function *CRCstreamAdd (value : byte) : word;*

kontinuierlich Bytes (value) der Checksumme hinzugefügt werden. Die Funktion gibt dann die jeweils aktuelle CRC Checksumme zurück.

Function *CRCstreamAddP (ptr : pointer; count : word) : word;*

Hiermit können der Checksumme ganze Speicherblöcke (RAM) hinzugefügt werden. Die Funktion gibt dann die jeweils aktuelle CRC Checksumme zurück.

4.13.11.3 FLASH CHECKSUM

Der Compiler kann eine 16bit Checksumme zur Compile-Zeit bilden und diese zur Laufzeit prüfen. Dazu muss die CheckSumme bzw. ihre Adresse definiert werden:

Define *FlashChkSum = $1ffe;* *// Byte Adresse*
oder
Define *FlashChkSum = ProgEnd;* *// Programm Ende*

Die Checksumme wird damit entweder auf die absolute Adresse oder an das Programm Ende abgelegt. Der Bereich der Prüfung erstreckt sich von Adresse $0000 bis zum Programm Ende oder bis zur Adressvorgabe. Das DEFINE bestimmt die obere Check Grenze. Dabei ist zu beachten, dass der Boot Bereich, wenn vorhanden, aus diversen Gründen nicht geprüft werden kann. Deshalb muss dann der Check unterhalb dieses Vereichs enden. Der Parameter zählt in Bytes.

Mit der Funktion

```
Function CalcFlashCheck : boolean;
```

wird zur Laufzeit eine Checksumme gebildet und mit der im Flash abgelegten verglichen. Das Ergebnis des Vergleichs wird als gut/true oder fehlerhaft/false zurückgegeben.

Bemerkung:
Diese Funktion kann nicht mit AVRs > 128kB Flash verwendet werden. Weiterhin sind alle Interrupts gesperrt während diese Funktion läuft.

Um auch CPUs mit mehr als 128kB zu unterstützen und um auch die Interrupt Sperrzeit möglichst kurz zu halten, sollte die untenstehende Funktion verwendet werden:

```
Function CalcFlashCheck_S (count : word) : byte;
```

Das ist ein sogenannter sequentieller oder partieller Flash Check. Er muss solange aufgerufen werden, bis die Funktion einen Wert <> 0 zurückgibt. Dadurch kann der Check in viele kleine Teile aufgeteilt werden und die Interrupts sind entsprechend nur ganz kurz gesperrt. Der Treiber muss importiert werden mit:

```
From System Import FlashCheck_S;
```

Der Übergabe Parameter der Funktion bestimmt die Anzahl der Bytes, die in diesem Durchgang geprüft werden sollen.

Die Funktion gibt eine 0 zurück wenn das Check Ende in diesem Durchgang noch nicht erreicht wurde. Ist das Ende erreicht, sow ird das Ergebnis ungleich Null:
1 = check finished and ok
2 = check finished but failed.

Wenn ein Ergebnis <> 0 zurückkommt, so wurden auch die internen Check Parameter auf die Default Werte gesetzt und der ganze Check kann neu gestartet werden.

```
repeat
  bb:= CalcFlashCheck_S ($1000);
until bb <> 0;
```

Für den evtl. vorhandenen **Bootbereich** kann auch eine Checksumme importiert werden:

```
From System Import FlashCheck_B;
```

Jetzt kann im Bootblock ein separater Flash Check über den ganzen Bootbereich aufgerufen werden:

```
Function CalcFlashCheck_B: boolean;
```

Es ist auch möglich aus dem Boot heraus die Checksumme der Applikation zu prüfen. Dazu müssen diese Imports und Defines vorhanden sein:

```
From System Import …, FlashCheck_A, FlashCheck_S, …;
```

FlashCheck_S importiert dabei die Checksummen Generierung für den Applikations Bereich.
FlashCheck_A importiert die Checksummen Test in dem Boot für den Applikations Bereich.

```
Define
    …
    FlashChkSum    = (2 * BOOTRST) -2;  // byte addr of checksum placement
                                        // at app area end, required for FlashCheck_A
```

Wurde der FlashCheck aus dem Boot über die Applikation mit FlashCheck_A importiert, so muss das System angewiesen werden, die gefundene Checksumme in die obersten/letzten 2 Bytes der Applikation abzulegen, optional auch andere Adressen. Dies kann mit Hilfe einer absoluten Byte Adresse geschehen

($xxxxx) oder symbolisch wie oben angegeben. Jetzt kann die Applikations Checksumme aus dem Bootbereich heraus geprüft werden:
In der BootApp muss diese Adresse auch angegeben werden mit:

```
Define FlashChkSum    = aaaa;     // Byte address

Function CalcFlashCheck_A (count : word) : byte;
```

Der Übergabe Parameter der Funktion bestimmt die Anzahl der Bytes, die in diesem Durchgang geprüft werden sollen.

Die Funktion gibt eine 0 zurück wenn das Check Ende in diesem Durchgang noch nicht erreicht wurde. Ist das Ende erreicht, sow ird das Ergebnis ungleich Null:
1 = check finished and ok
2 = check finished but failed.
Wenn ein Ergebnis <> 0 zurückkommt, so wurden auch die internen Check Parameter auf die Default Werte gesetzt und der ganze Check kann neu gestartet werden.

Beispiel:

```
repeat
  bb:= CalcFlashCheck_A ($1000);
until bb <> 0;
```

Ein Beispiel Programm ist in den Demos unter „BootAppChk“ zu finden.

4.13.11.4 EEPROM CHECKSUM

Bildet die Summe aller Bytes im angegebenen EEProm Bereich und gibt das negierte Resultat zurück.

```
Function calcCheckSum (const start, end : pointer) : word;

{$EEPROM}
structconst
  eInt      : word   = 1;
  eStr      : string  = 'eeprom';
  eWord     : word   = $1234;
  eByte     : byte    = $AA;
  ...
check:= CalcCheckSum (@eStr, @eByte);
```

Die Adresse von "eByte" wird als Ende-Pointer benutzt, der Wert von "eByte" selbst wird jedoch nicht mehr in die Kalkulation mit einbezogen.
Es ist auch möglich, eine EEprom Checksumme zur Compile-Zeit zu berechnen und im EEprom als strukturierte Konstante abzuspeichern:

```
{$EEPROM}
structconst
  eInt       : word   = 1;
  eStr       : string  = 'eeprom';
  eWord      : word   = $1234;
  eByte      : byte    = $AA;
  eCheck     : word   = CalcCheckSum (@eStr, @eByte);
```

Die Speicherstelle "eCheck" enthält jetzt die Checksumme aller EEprom Bytes von "eStr" (einschliesslich) bis "eByte" (ausschliesslich). Es ist auch möglich das Ziel des Funktions-Resultats (hier "eCheck") als Ende Pointer zu benutzen. Dies geschieht mit dem "$" Zeichen:

```
eCheck : word = CalcCheckSum (@eStr, $);
```

Diese Operationen sind nur im "StructConst" Bereich des EEproms zulässig.

4.13.12 RANDOM

Funktion, liefert eine Zufallszahl vom Typ Word. Random muss dazu importiert werden.

```
From System import Random…

Function Random : word;

W:= Random;
```

4.13.13 RANDOMRANGE

Funktion, liefert eine Zufallszahl vom Typ Word. Der Wert wird von min und max begrenzt. Random muss dazu importiert werden.

```
From System import Random…

Function RandomRange(min, max : word) : word;

W:= RandomRange(100, 500);
```

4.13.14 RANDOMSEED

Gibt den Seed/Startwert vom Typ Word vor. Darf nicht 0 sein! Random muss dazu importiert werden.

```
Procedure RandomSeed(seed : word);

RandomSeed($1234);
```

4.13.15 SQR

Funktionen, liefern das Quadrat des Float/Fix64 Arguments

```
Function Sqr (f : float) : float;
Function Sqr (f : fix64) : fix64;

f:= Sqr (f);
```

4.13.16 SQRT

Funktionen, liefern die Quadratwurzel des Ordinal/Float/Fix64 Arguments

```
Function Sqrt (i : LongInt) : LongInt;
Function Sqrt (ii : Int64) : Int64;
Function Sqrt (f : float) : float;
Function Sqrt (f : fix64) : fix64;          // precision 5 frac digits, 600usec@16MHz

f:= Sqrt (f);
```

4.13.17 POW

Funktion, liefert das Ergebnis von x hoch y. Die Basis ***x*** muss immer positiv sein!

```
Function Pow (x, y : float) : float;

f:= Pow (x, y);
```

4.13.18 POW10

Funktion, liefert das Ergebnis von 10 hoch x

Function *Pow10 (f : float) : float;*

f:= Pow10 (x);

4.13.19 EXP

Die Funktion Exp gibt die Potenz von X (float) zurück.
Der Rückgabewert ist e hoch X, wobei e die Basis des natürlichen Logarithmus ist.

Function *Exp (f : float) : float;*

f:= Exp (x);

4.13.20 LogN

Die Funktion LogN liefert als Ergebnis den natürlichen Logarithmus von X zurück.
Der Logarithmus Naturalis basiert auf der Euler'schen Zahl e = 2.71…

Function *LogN(f : float) : float;*

f:= LogN(x);

4.13.21 Log10

Die Funktion Log10 liefert als Ergebnis den dekadischen Logarithmus von X zurück.

Function *Log10 (f : float) : float;*

f:= Log10 (x);

4.13.22 Trigonometrische Funktionen

4.13.22.1 TAN

Die Funktion liefert als Ergebnis den Tangens vom Winkel w(BogenMass) zurück.

```
Function Tan (w : float) : float;

t:= Tan (w);
```

4.13.22.2 TAND

Die Funktion liefert als Ergebnis den Tangens vom Winkel w(GradMass) zurück.

```
Function TanD (w : float) : float;

t:= TanD (w);
```

4.13.22.3 ARCTAN

Die Funktion liefert als Ergebnis den Arkustangens des Arguments.

```
Function ArcTan (w : float) : float;

a:= ArcTan (w);
```

4.13.22.4 SIN

Die Funktion liefert den Sinus des Arguments zurück.
w ist ein Ausdruck vom Typ Float. Liefern den Sinus des Winkels w im Bogenmaß zurück.

```
Function Sin (w : float) : float;

s:= Sin (w);
```

4.13.22.5 SININT

Die Funktion liefert den Sinus des Winkels multipliziert mit dem Integer Argument.
Sehr schnell und kurz!

```
Function SinInt (angle, v : integer) : integer;
```

4.13.22.6 SININT16

Diese Funktion errechnet den Sinus des Winkels, multiplizieren diesen mit 10000 und gibt das Resultat als Integer Wert zurück. Der Winkel wird in 10tel Grad angegeben (2.5grad -> 25).
Das Resultat für den Winkel 90grad (parameter angle = 900) ist damit 10000.

```
Function SinInt16 (angle : integer) : integer;          // angle in 0.1deg
// result:= round (Sin (angle / 10) * 10000);
```

Liefert wesentlich präzisere Resultate als *SinInt* ohne dabei wiederum wesentlich länger zu brauchen.
Nachteil dieser Version:
im ROM wird eine Sinus Tabelle mit ca. 2kByte Grösse abgelegt.

4.13.22.7 SIND

Die Funktion liefert den Sinus des Arguments zurück.
w ist ein Ausdruck vom Typ Float. Liefern den Sinus des Winkels w im Gradmaß zurück.

```
Function SinD (w : float) : float;

s:= SinD (w);
```

4.13.22.8 COS

Die Funktion liefert den Cosinus des Arguments zurück.
w ist ein Ausdruck vom Typ Float. Liefern den Cosinus des Winkels w im Bogenmaß zurück.

```
Function Cos (w : float) : float;

c:= Cos (w);
```

4.13.22.9 COSINT

Die Funktionen liefern den Cosinus des Winkels multipliziert mit dem Integer Argument.
Sehr schnell und kurz!

```
Function CosInt (angle, v : integer) : integer;
```

4.13.22.10 COSINT16

Diese Funktion errechnet den Cosinus des Winkels, multiplizieren diesen mit 10000 und gibt das Resultat als Integer Wert zurück. Der Winkel wird in 10tel Grad angegeben (2.5grad -> 25). Das Resultat für den Winkel 45grad (parameter angle = 450) ist damit 7071.

```
Function CosInt16 (angle : integer) : integer;          // angle in 0.1deg
// result:= round (Cos (angle / 10) * 10000);
```

Liefert wesentlich präzisere Resultate als *CosInt* ohne dabei wiederum wesentlich länger zu brauchen.
Nachteil dieser Version:
im ROM wird eine Sinus Tabelle mit ca. 2kByte Grösse abgelegt.

4.13.22.11 COSD

Die Funktion liefert den Cosinus des Arguments zurück.
w ist ein Ausdruck vom Typ Float. Liefern den Cosinus des Winkels w im Gradmaß zurück.

```
Function CosD (w : float) : float;

c:= CosD (w);
```

4.13.22.12 DEGTORAD

Die Funktion wandelt eine im Gradmaß vorliegende Winkelgröße in Bogenmaß um. Dabei entsprechen 360 Grad einem Vollwinkel, der wiederum als Mittelpunktwinkel eines Kreises ein Bogenmaß von 2 Pi (rad) hat.
{ rad := Grad * Pi / 180 }

```
Function DegToRad (w : float) : float;

r:= DegToRad (w);
```

4.13.22.13 RADTODEG

Die Funktion wandelt eine im Bogenmaß vorliegende Winkelgröße in Gradmaß um.
{ Grad := rad * 180 / Pi }

```
Function RadToDeg (w : float) : float;

w:= RadToDeg (r);
```

4.13.22.14 ROTATE PNTi

Der Punkt(XPo, YPo) wird mit dem Winkel angle rotiert (Grad).
Das Ergebnis steht in XPd, YPd.

```
Procedure RotatePnti (angle, XPo, YPo : integer; var XPd, YPd : integer);
```

4.13.23 TRUNC

Die Funktion beschneidet einen Wert vom Typ Float/Fix64 auf einen Integerwert.
f ist ein Ausdruck vom Typ Float/Fix64. Trunc gibt den Typ zurück der dem des Ziels entspricht.

```
Function Trunc (f : float|fix64) : integer; {LongInt}

i:= Trunc (f);
```

4.13.24 ROUND

Die Funktion rundet einen Wert vom Typ Float/Fix64 zu einem Wert vom Typ Integer (byte, word, longint, longword).
f ist ein Ausdruck vom Typ Float/Fix64. Liefern einen int-Wert zurück, der den Wert von *f* gerundet zur nächsten ganzen Zahl darstellt. Wenn *f* exakt die Mitte zwischen zwei ganzen Zahlen bildet, wird im Ergebnis die Zahl mit dem höchsten absoluten Wert zurückgegeben.

```
Function Round (f : float|fix64) : integer; {Byte,Word,LongInt,LongWord}

i:= Round (f);
```

4.13.25 FRAC

Die Funktion liefert den Bruchanteil des Arguments x zurück. X ist ein Ausdruck vom Typ Float/Fix64. Das Ergebnis ist der Bruchanteil von x; das heißt Frac(x) = x - Int(x).

```
Function Frac (f : float|fix64) : float|fix64;

f:= Frac (x);
```

4.13.26 INT

Die Funktion liefert den Integerteil des Arguments zurück. X ist ein Ausdruck vom Typ Float/Fix64. Das Ergebnis ist der Integerteil von X; das heißt, X zu Null gerundet.

```
Function Int (f : float|fix64) : float|fix64;

f:= Int (x);
```

4.13.27 IntToFix64

Diese Funktion wandelt einen ordinal Wert (Byte…LongInt) in ein Fix64 um.

```
Function IntToFix64(i : ordinal) : fix64;

F64:= IntToFix64 (x);
```

4.13.28 GETTABLE

Die Funktion GetTable liefert ein Mitglied einer LookUp-Table zurück.

```
Function GetTable (t : Table; index : byte) : type;

x:= GetTable (Table1, b);
```

4.13.29 SETTABLE

Die Prozedur SetTable ändert ein Mitglied einer LookUp-Table.

```
Procedure SetTable (t : Table; index : byte; new : type);

SetTable (Table1, b, x);
```

4.13.30 Konvertierung zu Strings

bei allen diesen Konvertierungen ist der Frame Bedarf nicht zu unterschätzen.
Dies gilt auch für Prozesse und Tasks wo diese Funktionen aufgerufen werden.

z.B.:
ByteToStr > 12 bytes
IntToStr > 17 bytes
LongToStr > 37 bytes !
Int64ToStr > 70 bytes !!!

4.13.30.1 BYTETOSTR

Konvertiert Numerischen 8bit Wert in einen String. Der Parameter kann eine ordinale numerische Konstante (Byte, Int8, Enum) oder eine Variable dieses Typs sein.

```
Function ByteToStr (b : byte[Int8, Enum]) : string;

const  st  = '1234' + 'R' + #7 + ^L;
var  st1   : string[9];
     bb    : byte;

write (LCDout, ByteToStr (100));
write (SERout, ByteToStr (100:6));   {-> '   100'}
bb:= 10;
st1:= ByteToStr (bb);                { -> '10'}
st1:= ByteToStr (bb:6);              { -> '    10'}
st1:= ByteToStr (bb:6:1);            { -> '   1.0'}
st1:= ByteToStr (bb:6:1:'_');        { -> '___1.0'}
st1:= ByteToStr (bb:6:'_');          { -> '____10'}
```

4.13.30.2 INTTOSTR

Konvertiert Numerischen 16bit Wert in einen String. Der Parameter kann eine ordinale numerische Konstante (integer, word) oder eine Variable dieses Typs sein.

```
Function IntToStr (i : word) : string;

write (LCDout, IntToStr (100));
write (SERout, IntToStr (i:6:2));    {-> '  1.00'}
st1:= IntToStr (123:4);              {-> ' 123'}
ii:= -1;
st1:= IntToStr (ii);                 {-> '-1'}
st1:= IntToStr (ii:10);              {-> '        -1'}
st1:= IntToStr (ii:10:2);            {-> '     -0.01'}
st1:= IntToStr (ii:10:2:'x');        {-> 'xxxxx-0.01'}
st1:= IntToStr (ii:10:'x');          {-> 'xxxxxxxx-1'}
```

4.13.30.3 LONGTOSTR

Konvertiert einen numerischen 32bit Wert in einen String. Der Parameter kann eine ordinale numerische Konstante (Longint, Longword) oder eine Variable dieses Typs sein.

```
Function LongToStr (ii : longword) : string;

write (LCDout, LongToStr (100000));
write (SERout, LongToStr (ii:6:2));  {-> '  1.00'}
st1:= LongToStr (123456:8);          {-> '  123456'}
Li:= 100;
st1:= IntToStr (Li);                 {-> '100'}
st1:= IntToStr (Li:10);              {-> '       100'}
st1:= IntToStr (Li:10:2);            {-> '      1.00'}
st1:= IntToStr (Li:10:2:'x');        {-> 'xxxxxx1.00'}
st1:= IntToStr (Li:10:'x');          {-> 'xxxxxxx100'}
```

4.13.30.4 FLOATTOSTR

Konvertiert Floating Point Wert in einen String. Der Parameter kann eine Float-Konstante oder eine Variable dieses Typs sein.

```
Function FloatToStr (f : float) : string;

write (LCDout, FloatToStr (1000.00));
write (SERout, FloatToStr (f:6:2));     {-> '  1.00'}
st1:= FloatToStr (123.456:8);           {-> '    123'}
f:= -100.1;
st:= FloatToStr (f);                    {-> '-100.1'}
st1:= FloatToStr (f:11);                {-> '      -100'}
st:= FloatToStr (f:11:0);               {-> '      -100'}
st:= FloatToStr (f:11:2);               {-> '   -100.10'}
st:= FloatToStr (f:11:2:'=');           {-> '====-100.10'}
st1:= FloatToStr (f:11:'=');            {-> '=======-100'}
st1:= FloatToStr (f:'E');               {-> '-1.001E2'}
st1:= FloatToStr (f:'E':11);            {-> '   -1.001E2'}
st1:= FloatToStr (f:'E':11:'=');        {-> '===-1.001E2'}
```

wird keine Formatierung angegeben "FloatToStr(f)" dann erfolgt die Ausgabe mit allen Vor- und Nachkomma Stellen.

wird nur ein Parameter angegeben "FloatToStr(f:n)" dann wird der String mindestens "n" Stellen lang, ggf. mit führenden Leerzeichen. Nachkomma Stellen werden unterdrückt. Ist also dasselbe wie "FloatToStr(f:n:0)"

werden zwei Parameter angegeben so kann der zweite entweder die Nachkomma Stellen bestimmen, wenn er numerisch ist. Wenn dieser ein Character ist bestimmt er den Füll Character und es werden keine Nachkomma Stellen angezeigt.
a. FloatToStr(f:n:3) ergeben einen String mit der totalen länge "n" mit 3 Nach Komma Stellen. Führende Stellen werden mit Spaces gefüllt.
b. FloatToStr(f:n:'x') ergibt einen String mit der Länge "n" wobei führende Stellen hier keine Spaces sind sondern "x". Keine Nachkomma Stellen.

werden 3 Parameter angegeben "FloatToStr(f:n:k:'x')" bekommt der String eine totale Länge von "n" mit "k" Nachkomma Stellen. Führende Leerzeichen werden durch 'x' ersetzt.

Achtung:
Diese System Funktion benötigt zusätzlich 20 Bytes auf dem Frame.

Bemerkung:
Bei den Funktionen *ByteToStr*, *IntToStr*, *LongToStr* und *FloatToStr* können als Längen und Dezimalstellen Parameter auch Byte Variable benutzt werden. Allerdings entfällt damit komplett die Plausibilitäts-Prüfung dieser Parameter und der Programmierer selbst für korrekte Werte verantwortlich. Bei illegalen Parametern kommt es mit Sicherheit zu unerwarteten Resultaten oder gar zum System Crash.

Beispiel für variable Parameter:

```
var  digs,
     dec : byte
     ii  : integer;
   ...
st:= IntToStr (ii:digs:dec);
```

4.13.30.5 LONG64TOSTR

Konvertiert einen numerischen 64bit Wert in einen String. Der Parameter kann eine ordinale numerische Konstante (Int64, Word64) sein.

Function *Long64ToStr (**const** ii : Int64|Word64[: **const** len : byte[: **const** space : char]]) : string;*

4.13.30.6 Fix64toStr

Konvertiert einen Fix64 Wert in einen String. Die optionalen Parameter int, frac und space formatieren den String.

Function *Fix64ToStr(f : Fix64[:int : byte[:frac : char] [:space : char]]) : string;*

```
f:= -100.1;
st:= Fix64ToStr (f);              {-> '-100.100000000'}
st:= Fix64ToStr (f:6);            {-> '      -100'}
st:= Fix64ToStr (f:6:0);          {-> '      -100'}
st:= Fix64ToStr (f:6:2);          {-> '    -100.10'}
st:= Fix64ToStr (f:6:2:'=');      {-> '====-100.10'}
st:= Fix64ToStr (f:6:'_');        {-> '_______-100'}
```

wird keine Formatierung angegeben "Fix64ToStr(f)" dann erfolgt die Ausgabe mit allen Vor- und Nachkomma Stellen. Nachkomma Stellen sind dann immer 9 Digits.

wird nur ein Parameter angegeben "Fix64ToStr(f:n)" dann wird der String mindestens "n" Stellen (int digits) lang, ggf. mit führenden Leerzeichen. Nachkomma Stellen werden unterdrückt. Ist also dasselbe wie "Fix64ToStr(f:n:0)"

werden zwei Parameter angegeben so kann der zweite entweder die Nachkomma Stellen bestimmen, wenn er numerisch ist. Wenn dieser ein Character ist bestimmt er den Füll Character und es werden keine Nachkomma Stellen angezeigt.
a. Fix64ToStr(f:n:3) ergeben einen String mit Int Digits "n" und 3 Nach Komma Stellen. Führende Stellen werden mit Spaces gefüllt.
b. Fix64ToStr(f:n:'x') ergibt einen String mit der Int digits Länge "n" wobei führende Stellen hier keine Spaces sind sondern "x". Keine Nachkomma Stellen.

werden 3 Parameter angegeben "Fix64ToStr(f:n:k:'x')" bekommt der String eine totale Int digit Länge von "n" mit "k" Nachkomma Stellen. Führende Leerzeichen werden durch 'x' ersetzt.

Achtung:
Diese System Funktion benötigt zusätzlich 20 Bytes auf dem Frame.

Bemerkung:
Bei dieser Funktion können als Längen und Dezimalstellen Parameter auch Byte Variable benutzt werden. Allerdings entfällt damit komplett die Plausibilitäts-Prüfung dieser Parameter und der Programmierer selbst ist für korrekte Werte verantwortlich. Bei illegalen Parametern kommt es mit Sicherheit zu unerwarteten Resultaten oder gar zum System Crash.

4.13.30.7 BOOLTOSTR

Konvertiert eine Boolean Variable in einen String.
Zwei Funktionen sind implementiert. Die erste setzt das Boolean Argument in den String *'true'* oder *'false'* um. Die zweite Funktion gibt als Ergebnis entweder den TrueStr oder den FalseStr zurück, abhängig vom Wert des boolean Arguments.

Function *BoolToStr (bool : boolean) : string;*
Function *BoolToStr (bool : boolean; TrueStr, FalseStr : string) : string;*

4.13.30.8 BYTETOHEX

Konvertiert Numerischen 8bit Wert in einen hex-String. Der Parameter kann eine ordinale numerische Konstante (byte, Enum) oder eine Variable dieses Typs sein.

Function *ByteToHex (b : byte[Int8]) : string;*

```
x:= 48;
st1:= ByteToHex (x) + 'h';          {st1 enthält '30h'}
```

4.13.30.9 INTTOHEX

Konvertiert Numerischen 16bit Wert in einen hex-String. Der Parameter kann eine ordinale numerische Konstante (integer, word) oder eine Variable dieses Typs sein.

Function *IntToHex (i : integer) : string;*

```
st1:= IntToHex (123);               {st1 enthält '7B'}
```

4.13.30.10 LONGTOHEX

Konvertiert einen numerischen 32bit Wert in einen hex-String. Der Parameter kann eine ordinale numerische Konstante (Longint, Longword) oder eine Variable dieses Typs sein.

Function *LongToHex (w : longword) : string;*

```
st1:= LongToHex (123456);           {st1 enthält '1E240'}
```

4.13.30.11 LONG64TOHEX

Konvertiert einen numerischen 64bit Wert in einen hex-String. Der Parameter kann eine ordinale numerische Konstante (Int64, Word64) sein.

Function *Long64ToHex (**const** ii : Int64|Word64) : string;*

4.13.30.12 Fix64TOHEX

Konvertiert einen Fix64 Wert in einen hex-String.

Function *Fix64ToHex (**const** f : Fix64) : string;*

4.13.30.13 BYTETOBIN

Function *ByteToBin (value : byte[Int8}) : string;*

Das Ergebnis ist die Repräsentation der einzelnen Bits des Arguments durch eine '0' oder '1' im String. Ein Byte mit dem Wert 5 ergibt daher '00000101'.

4.13.30.14 INTTOBIN

Function *IntToBin (value : word|integer) : string;*

Das Ergebnis ist die Repräsentation der einzelnen Bits des Arguments durch eine '0' oder '1' im String. Ein Word mit dem Wert 257 ergibt daher '0000000100000001'.

4.13.31 BYTETOBCD

Function *ByteToBCD (b : byte) : byte;*

Echtzeit Uhren Chips benötigen meistens zu Einstellung sog. *Packed BCD* Werte:
BCD Zahlen in den oberen 4bits die Zehner Stellen und in den unteren 4 Bits die Einerstellen.
Per Definition können beide Stellen nur von 0..9 laufen.
Diese Funktion konvertiert ein Byte in das BCD Format. Dabei ist zu beachten, dass der maximale Übergabe Wert nicht grösser als 99 sein darf. Der Wert $10 (16dez) wird dann als $16 zurückgegeben.

4.13.32 WORDTOBCD

Function *WordToBCD (w : word) : word;*

Konvertiert ein Word in das BCD Format. Dabei ist zu beachten, dass der maximale Übergabe Wert nicht grösser als 9999 sein darf. Der Wert $270F (9999dez) wird dann als $9999 zurückgegeben.
In den höchsten 4 Bits sind die Tausender Stellen, in den nächsten 4 Bits die Hunderter Stellen, dann die Zehner Stellen und in den untersten 4 Bits sind die Einer Stellen.
Per Definition können alle Stellen nur von 0..9 laufen.

4.13.33 BCDTOBYTE

Function *BCDtoByte (b : byte) : byte;*

Echtzeit Uhren Chips benötigen meistens zu Einstellung sog. Packed BCD Werte:
BCD Zahlen in den oberen 4bits die Zehner Stellen und in den unteren 4 Bits die Einerstellen. Per Definition können beide Stellen nur von 0..9 laufen. Beim Auslesen wird dann auch dieses Format zurückgegeben.
Diese Funktion erwartet einen packed BCD Wert und konvertiert diesen zu einem binären Byte. $16 -> $10.

4.13.34 PCU SI-Conversion (*P*)

by Tassilo Heinrich

4.13.34.1 Allgemeine Funktionen

Function *CountsToVolts (Counts:Word; VRef:Float; Res:word; Gain:Float):Float;*
Konvertiert AD-Counts zu Volt.
Volt := CountsToVolts (512, 5.0, 1024,1.0): Float; //= 2.5Volt

Function *MX_B (m:Float; x:Float; b:Float): Float;*
Y = mX+t

Function *ByteToBcd (byteVal : Byte) : Byte;*
Konvertiert Byte zu BCD.

Function *BcdToByte (bcdVal : Byte) : Byte;*
Konvertiert BCD zu Byte.

4.13.34.2 Temperatur

Function *F_CelsiusToKelvin (Cel:Float): Float;*
Konvertiert °C zu Kelvin.

Function *F_KelvinToCelsius (Kelvin:Float): Float;*
Konvertiert Kelvin zu °C.

Function *F_FahrenheitToCelsius (Fahrenheit:Float): Float;*
Konvertiert Fahrenheit zu °C.

Function *F_CelsiusToFahrenheit (Celsius:Float): Float;*
Konvertiert °C zu Fahrenheit.

4.13.34.3 Volumen

Function *F_LiterToGal (Liter:Float): Float;*
Konvertiert Liter zu US Galonen.

Function *F_GalToLiter (Gal:Float): Float;*
Konvertiert US Galonen zu Liter.

Function *F_LiterToCuFt (Liter:Float): Float;*
Konvertiert Liter zu Feet^3 .

Function *F_CuFtToLiter (CuFt:Float): Float;*
Konvertiert Feet^3 zu Liter.

Function *F_CuFtToCuIn (CuFt:Float): Float;*
Konvertiert Feet^3 zu Inches^3.

Function *F_CuInToCuFt (CuIn:Float): Float;*
Konvertiert Inches^3 zu Feet^3.

4.13.34.4 Druck

Function *F_PSITomBar (PSI:Float): Float;*
Konvertiert PSI zu Bar.

Function *F_mBarToPSI (mBar:Float): Float;*
Konvertiert Bar zu PSI.

Function *F_mmHgTomBar (mmHg:Float): Float;*
Konvertiert mmHg zu Bar.

Function *F_mBarTommHg (mBar:Float): Float;*
Konvertiert Bar zu mmHg.

Function *F_cmH2OtomBar (cmH2O:Float): Float;*
Konvertiert cmH_2O zu Bar.

Function *F_mBarTocmH2O (mBar:Float): Float;*
Konvertiert Bar zu cmH_2O.

4.13.34.5 Längen

Function *F_MeterToFeet (Meter:Float): Float;*
Konvertiert Meter zu Feet.

Function *F_FeetToMeter (Feet:Float): Float;*
Konvertiert Feet zu Meter

Function *F_InTocMeter (Inch:Float): Float;*
Konvertiert Inches zu Meter.

Function *F_cMeterToIn (cMeter:Float): Float;*
Konvertiert Meter zu Inches.

Function *F _ydToMeter (yd:Float): Float;*
Konvertiert Yards zu Meter.

Function *F_MeterToyd (Meter:Float): Float;*
Konvertiert Meter zu Yards.

Function *F_miTokMeter (mi:float): Float;*
Konvertiert Meilen zu Kilometer.

Function *F_kMeterTomi (kMeter:Float): Float;*
Konvertiert Kilometer zu Meilen.

Function *F_nmiTokMeter (nmi:float): Float;*
Konvertiert Nautische Meilen zu Kilometer.

Function *F_kMeterTonmi (kMeter:Float): Float;*
Konvertiert Kilometer zu Nautische Meilen.

4.13.34.6 Flächen

Function *F_SqrMeterToSqrFeet (Meter:Float): Float;*
Konvertiert Meter² zu Feet².

Function *F_SqrFeetToSqrMeter (Feet:Float): Float;*
Konvertiert Feet² zu Meter².

Function *F_SqrInToSqrcMeter (Inch:Float): Float;*
Konvertiert Inches² zu Zentimeter²

Function *F_SqrcMeterToSqrIn (cMeter:Float): Float;*
Konvertiert Zentimeter² zu Inches²

Function *F_SqrydToSqrMeter (yd:Float): Float;*
Konvertiert Yards² zu Meter²

Function *F_SqrMeterToSqryd (Meter:Float): Float;*
Konvertiert Meter² zu Yards²

Function *F_SqrmiToSqrkMeter (mi:float): Float;*
Konvertiert Meilen² zu Kilometer²

Function *F_SqrkMeterToSqrmi (kMeter:Float): Float;*
Konvertiert Kilometer² zu Meilen²

4.13.34.7 Gewichte

Function *F_KaratToGramm (Karat:Float): Float;*
Konvertiert Karat zu Gramm

Function *F_GrammToKarat (Gramm:Float): Float;*
Konvertiert Gramm zu Karat

Function *F_GrainsToGramm (Grains:Float): Float;*
Konvertiert Grains zu Gramm

Function *F_GrammToOunces (Gramm:Float): Float;*
Konvertiert Gramm zu Ounces.

Function *F_OuncesToGramm (Ounces:Float): Float;*
Konvertiert Ounces zu Gramm

Function *F_GrammToOuncesTroy (Gramm:Float): Float;*
Konvertiert Gramm zu Ounces troy.

Function *F_OuncesTroyToGramm (OuncesTroy:Float): Float;*
Konvertiert Ounces troy zu Gramm.

Function *F_kGrammToStones (kGramm:Float): Float;*
Konvertiert Kilogramm zu Stones.

Function *F_StonesTokGramm (Stones:Float): Float;*
Konvertiert Stones zu Kilogramm.

Function *F_kGrammToPounds (kGramm:Float): Float;*
Konvertiert Kilogramm zu Pound.

Function *F_PoundsTokGramm (Pounds:Float): Float;*
Konvertiert Pound zu Kilogramm.

4.13.34.8 Energie

Function *F_kWToPS (kW:Float): Float;*
Konvertiert Kilowatt zu PS.

Function *F_PSTokW (PS:Float): Float;*
Konvertiert PS zu Kilowatt.

Function *F_CalToJ (Cal:Float): Float;*
Konvertiert Kalorien zu Joule.

Function *F_JtoCal (J:Float): Float;*
Konvertiert Joule zu Kalorien.

4.13.34.9 Integer Funktionen

Function *I_CelsiusToKelvin (Cel:Integer): Integer;*

Function *I_KelvinToCelsius (Kel:Integer): Integer;*

Function *I_FahrenheitToCelsius (Fahrenheit:Integer): Integer;*

Function *I_CelsiusToFahrenheit (Celsius:Integer): Integer;*

4.13.34.10 Konstanten

ZeroPoint : Float = -273.16;

MSL_Pressure_PSI : float = 14.697;

MSL_Pressure_InHg : float = 29.92;

MSL_Pressure_mBar : float = 1013.25;

Euler : Float = 2.7182818284;

4.13.35 Interpolation

Viele Sensoren, aber auch andere Funktionen, arbeiten extrem nicht-linear.
Beispiele: PT100, PTC, NTC, Foto-Detektoren, aber auch z.B. Dioden.
Die Reihe lässt sich fast beliebig fortsetzen. Da dies alles analoge Werte sind, werden diese in der Regel mit einem AD-Wandler erfasst. Normalerweise steht das Messergebnis in einem bestimmten Verhältnis zu dem äusseren Ereignis, z.B. Temperatur.
Aber eben dieses Verhältnis ist sehr oft nicht linear. Ein PT100 z.B. hat bei 0° einen Widerstand von 100 Ohm bei 50° ca. 124 Ohm und bei 100° ca. 143 Ohm. Der Zusammenhang zwischen Temperatur und Widerstand ist also nicht-linear.

Um jetzt die aktuelle Temperatur zu erhalten, kann man zwei Wege beschreiten:
1. Mit einer passenden Formel, die meistens sehr komplex ist, kann die dem Widerstandswert zugeordnete Temperatur ermittelt werden.
2. Man erstellt sich eine sogenannte LookUp-Table in der in Stufen bestimmte Widerstandswerte und die zugehörige Temperatur abgelegt sind. Mit dem Widerstandswert (Messwert) als Index wird jetzt auf die Tabelle zugegriffen.
   Kann das Messergebnis von 100 bis 200 laufen, werden 100 Werte (Ergebnisse)in der Tabelle benötigt. Etwas schwieriger wird es, wenn der Analogteil und AD-Wandler so beschaltet ist, dass das Mess-ergebnis von 0 bis 1023 variiert. Dann braucht man 1024 Einträge in der Tabelle.

Die vorliegende Implementation basiert auf einer Tabelle, in der sowohl die Messergebnisse als auch die zugehörigen Resultate immer paarweise abgelegt sind. Der LookUp Algorithmus sucht mit dem Mess-ergebnis als Argument in der Tabelle, bis er auf einen gleichen Wert stösst, oder mit dem Argument zwischen 2 Werten liegt. Der Suchvorgang wird aus Geschwindigkeitsgründen mit einem binären Verfahren ausgeführt.

Ist ein passender Wert gefunden, wird das Ergebnis zurückgegeben. Liegt das Argument zwischen 2 Werten, wird linear interpoliert. Dieses Verfahren ermöglicht je nach Genauigkeits Forderungen relativ kurze Tabellen. Wenn die Koordinaten Zahl in der Tabelle relativ hoch ist, ergibt die lineare Interpolation eine ausreichende Genauigkeit, da man im allgemeinen davon ausgehen kann, dass kurze Abschnitte auch näherungsweise linear sind.

4.13.35.1 InterPolX, InterPolY

```
Function InterPolX (const LookUp : pointer; x: integer; var y: integer) : boolean;
Function InterPolX (const LookUp : pointer; x: longint; var y: longint) : boolean;
Function InterPolX (const LookUp : pointer; x : float; var y : float) : boolean;

Function InterPolY (const LookUp : pointer; y: integer; var x: integer) : boolean;
Function InterPolY (const LookUp : pointer; y: longint; var x: longint) : boolean;
Function InterPolY (const LookUp : pointer; y : float; var x : float) : boolean;
```

Der Pointer muss in die Tabelle im ***ROM*** oder ***EEprom*** zeigen. Das erste Argument ist der Suchwert. Das Ergebnis wird im zweiten Parameter abgelegt, wenn die Suche erfolgreich war.

Zur Erstellung der Tabelle enthält das System den Table Generator "CurveGen", mit dem man graphisch und interaktiv eine Kurve erstellen und als binär File zum Import in die Applikation ablegen kann.

Programm Beispiel:

Ein Beispiel Programm ist im Verzeichnis **..\E-Lab\AVRco\Demos\Interpol** zu finden.
Hierin wird ein optischer Entfernungsmesser (Sharp) linearisiert und die Entfernung in cm ausgegeben.

Ein Datenblatt dieses Sensor befindet sich in **..\E-Lab\DOCs\Sharp.pdf**
Eine ausführlichere Beschreibung des Hilfsprogramms und obiger Funktionen finden Sie im *Tools Manual*.
Eine Beispiel Schaltung finden Sie in **..\E-Lab\DOCs\NonLinSensSch.pdf**

4.13.36 Gleitendes Mittelwert Filter (Moving Average Filter)

Bildet den Mittelwert aus z.B. den letzten 16 Messungen.
Damit erhält man stabile Werte, die sich trotzdem relativ schnell an Veränderungen anpassen.
Das Filter wird praktisch aus einem Array of Byte/Word/Integer etc. gebildet. Wird ein neuer Wert hinzugefügt, ersetzt dieser den ältesten vorhandenen Wert. Durch aufaddieren aller enthaltenen Werte und anschliessender Division durch die Anzahl der Werte erhält man den Mittelwert.

Das Filter (Array) muss als Variable im normalen RAM deklariert werden. Banks, EEprom, prozedurlokal oder Records etc. sind nicht zulässig.
Es kann aus Bytes, Words, Integer, LongWords, LongInts oder Float bestehen.
Die Grösse läuft von 0 bis x, wobei x eine zweier Potenz im Bereich 4..256 sein muss. Bei der Deklaration ist hierbei eine 1 abzuziehen.

```
var Filter : AVfilter[0..15] of integer;
```

4.13.36.1 PresetAVfilter

```
Procedure PresetAVfilter (var Filter : AVfilter; val : type);
```
Besetzt das komplette Filter mit "val".

4.13.36.2 SetAVfilter

```
Function SetAVfilter (var Filter : AVfilter; val : type) : type;
```
Ersetzt den ältesten Eintrag durch den Wert "val" und liefert den neu gerechneten Mittelwert zurück.

4.13.36.3 AddAVfilter

```
Procedure AddAVfilter (var Filter : AVfilter; val : type);
```
Manchmal ist es nicht notwendig beim ersetzen des ältesten Wertes auch gleich eine Mittelwert Bildung vorzunehmen. Diese Funktion ist identisch mit "SetAVfilter" bis auf den Punkt, dass kein neuer Mittelwert gebildet wird.

4.13.36.4 GetAVfilter

```
Function GetAVfilter (var Filter : AVfilter) : type;
```
Ermittelt den aktuellen Mittelwert ohne die Inhalte zu verändern.

4.13.36.5 DeclAVfilter

```
Function DeclAVfilter (var Filter : AVfilter) : type;
```
Errechnet die Steigung zwischen dem ältesten und dem jüngsten Eintrag. Das Ergebnis ist immer mit Vorzeichen. D.h. Bei einem Byte, Integer oder Word Filter ist das Ergebnis ein Integer. Bei einem LongWord oder LongInt Filter ist das Ergebnis ein LongInt.

Programm Beispiel:

Ein Beispiel Programm ist im Verzeichnis **..\E-Lab\AVRco\Demos\AVfilter** zu finden.

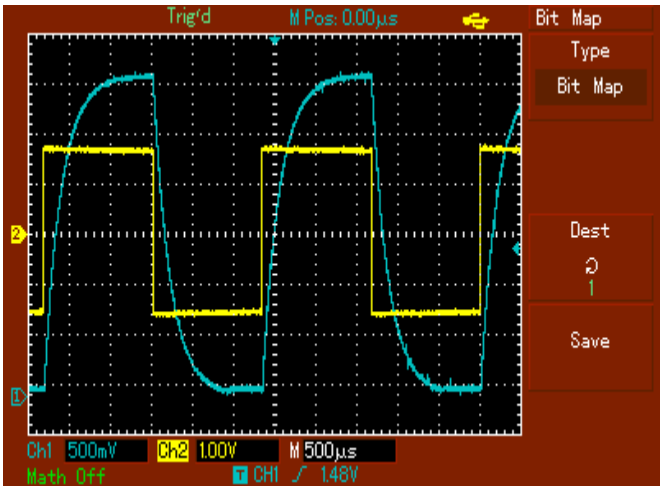
4.13.37 Filter Low und High Pass

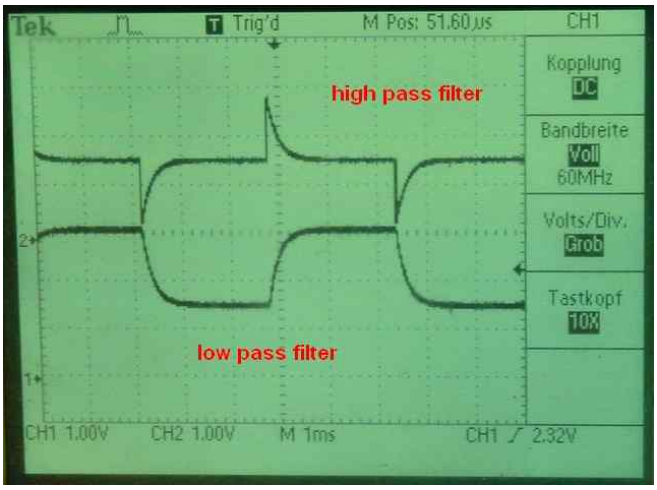
Filters muss importiert werden. Durch den Treiber exportierte Typen und Funktionen:

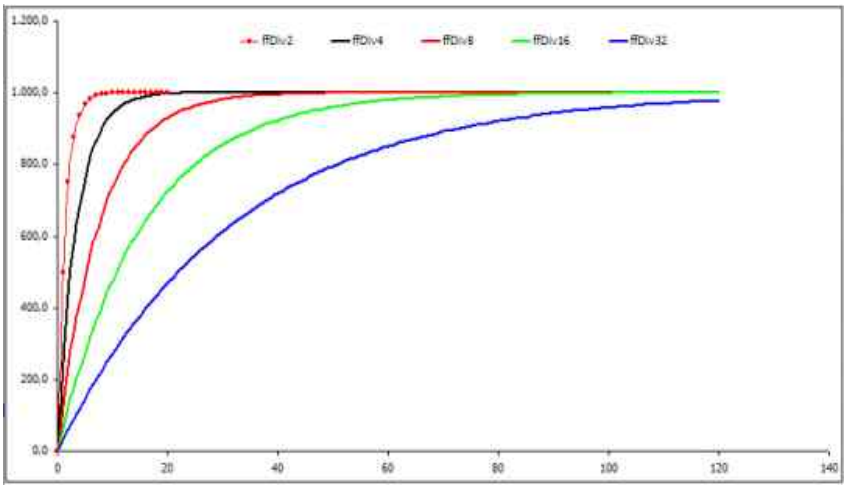
```
type
TFilterFreq = (ffdiv2, ffdiv4, ffdiv8, ffdiv16, ffdiv32, ffdiv64, ffdiv128, ffdiv256);

TFiltData = record
  value    : word;
  Err      : byte;
  HPbias : word;    // optional high pass filter bias
end;

function LowPassFW(var FiltData : TFiltData; NewVal : word; FilterFreq : TFilterFreq) : word;
function HighPassFW(var FiltData : TFiltData; NewVal : word; FilterFreq : TFilterFreq) : word;
```







Diese Filter bieten veränderbare Filter Werte durch Benutzung von resistance als TFilterFreq Parameter. Höhere Parameter Werte ergeben höhere resistance für die Veränderung der Filter Werte. Das Bild links zeigt diesen Effekt. Durch bit shifting in einem word zur schnellen Division werden nur zwischen 354 Zyklen (ffDiv2) und 423 Zyklen (ffDiv256) benötigt um ein sehr schnelles Resultat zu erreichen.

Wenn wir z.B. mit FiltData.Value = 0 anfangen und NewValue = 1000 vorgeben, mit ffDiv2, dann wird das Ergebnis nach dem ersten LowPassFW das Ergebnis (oder Result) die hälfte der Differenz zwischen FiltData.Value und NewValue sein, das ist (1000-0)/2=500. Beim nächsten Aufruf von LowPassFW wird die Differenz zwischen FiltData.Value und NewValue (1000-500)/2=250 sein und das Resultat wird 500+250=750 sein. Nach dem nächsten Aufruf wird das Resultat 750+125=875 sein, und beim nächsten 875+62=937, etc.etc. (bis das Resultat bei 1000 stabil bleibt). Also wird ffDiv2 immer die Hälfte der Differenz, ffDiv4 wird ein viertel der Differenz, ffDiv8 ein achtel der Differenz zurückgeben, etc. Deshalb erlaubt ffDiv2 eine wesentlich höhere Frequenz Änderung als z.B. ffDiv256, welches viel, viel langsamer ist. Deshalb erlaubt uns die Wahl eines höheren TFilterFreq Werts mehr Peaks in einem analogen Signal herauszufiltern und eine weichere Pegeländerung zu erhalten, ohne Sprünge darin.

Beispiel Programme sind im Verzeichnis **..\E-Lab\AVRco\Demos\XMega_Filters** zu finden.

4.13.38 Netzwerk-Funktionen

4.13.38.1 vordefinierte Typen

```
type TIPAddr = array[0..3] of Byte;
type TMACAddr = array[0..5] of Byte;
```

4.13.38.2 Funktionen zur Konvertierung

Procedure STRtoIP (IPstr : String[15]; var Result : TIPAddr);
Konvertiert einen IP-Adress String "aaa:bbb:ccc:ddd" in ein Byte Array

Procedure STRtoMAC (MACstr : String[17]; var Result : TMACAddr);
Konvertiert einen MAC-Adress String "aa:bb:cc:dd:ee:ff" in ein Byte Array

Function IPtoSTR (IP : TIPAddr) : String[15];
Konvertiert ein IP-Adress Array in einen String "aaa:bbb:ccc:ddd"

Function MACtoSTR (MAC : TMACAddr) : String[17];
Konvertiert ein MAC-Adress Array in einen String "aa:bb:cc:dd:ee:ff"

4.13.38.3 Funktionen zum Vergleichen

Function CompareNet (a1, a2, mask : TIPAddr) : boolean;
Vergleicht den Netzwerk-Teil zweier IP-Adress Arrays

Function CompareIP (ip1, ip2 : TIPAddr) : boolean;
Vergleicht zwei IP-Adress Arrays

Function CompareMAC (mac1, mac2 : TMACAddr) : boolean;
Vergleicht zwei MAC-Adress Arrays

4.13.38.4 Diverse Funktionen

Procedure SwapIPaddr (var ip : TIPAddr);
Spiegelt eine IP-Adresse. A3-A2-A1-A0 wird zu A0-A1-A2-A3 konvertiert.

Procedure SwapMACaddr (var mac : TMACAddr);
Spiegelt eine MAC-Adresse. A5-A4-A3-A2-A1-A0 wird zu A0-A1-A2-A3-A4-A5 konvertiert.

4.14 System Library - String Formatierung

Wie aus obenstehender Beschreibung von *ByteToStr* und *IntToStr* zu ersehen ist, kann das Ziel einer Stringkonvertierung eine Stringvariable oder die Prozedur WRITE sein.

Um z.B. ein LCD-Display sinnvoll ansteuern zu können, ist eine Konvertierung von Werten (Variablen) in Strings, sowie deren Formatierung und Ausgabe absolut notwendig.
Die Konvertierungsroutinen in Zusammenarbeit mit WRITE lösen diese Aufgabe.

Die **Formatierung** erfolgt weitgehend identisch mit Turbo Pascal. Der Parameter nach dem **ersten** Doppelpunkt gibt die gewünschte **Gesamtlänge** des Strings an. Dabei ist zu beachten, dass der String trotzdem länger werden kann, aber nicht kürzer, denn “100:0” ergibt als String “100”, hat also drei Stellen und nicht Null Stellen.

Nicht identisch mit Turbo Pascal ist der Parameter nach dem **zweiten** Doppelpunkt bei *ByteToStr* und *IntToStr*. Dieser gibt die **Anzahl** der Nachkommastellen an. Integer haben aber gar keine Kommastellen! Klar, aber oft wird mit Integern mit Festkomma gerechnet und dann erfolgt das Problem der korrekten Darstellung. Zu beachten ist dabei auch, dass der Dezimalpunkt mitgezählt wird! “100:6:2” ergibt dann “ 1.00” mit zwei führenden Leerzeichen, also insgesamt 6 Stellen.
Führende Leerzeichen können durch einen anderen Buchstaben ersetzt werden, der nach dem letzten Doppelpunkt angegeben werden muss.

4.14.1 Dezimal Separator

Diverse String Konvertierungen arbeiten mit einem Dezimal Punkt. Dieser Dezimal Punkt ist standardmässig das Zeichen ”**.**” Dieses Zeichen kann redefiniert werden, z.B. zum ”**,**” um speziellen Anforderungen gerecht zu werden. Hierzu muss im Define Block folgendes Statement eingefügt werden:

```
Define  DecimalSep = ‘.‘;
```

4.14.2 WRITE

String oder Zahlausgabe mit Konvertierung über eine Procedure. Der erste Parameter von Write muss eine Prozedur sein.

Ist der erste Parameter eine Prozedur (Device), wird diese solange aufgerufen, bis der zweite Parameter abgearbeitet ist. Die Prozedur **muss** folgende Form haben:

```
proc (b : byte);
```

Da die Prozedur wiederum beliebige Hardware ansprechen kann, ist damit eine komfortable Datenausgabe an externe Geräte möglich.

```
const  st  = '1234' + 'R' + #7 + ^L;
var  st1 : string[5];

Write (proc, 'x');                      {‘x’ wird ausgegeben }
Write (proc, st1);                      {st1 wird ausgegeben }
Write (LCDout, ByteToStr (100));
Write (LCDout, st[0]);                  {Längen byte}
Write (SerOut, st);                     {Ausgabe kompl. String}
Write (LCDout, st[1]);                  {1. Zeichen in st}
Write (LCDout, '1234');
Write (SERout, ByteToStr (100:6));      {-> '   100'}
Write (SERout, ByteToStr (100:6:2));    {-> '  1.00'}
Write (SERout, IntToStr (i:6:1));       {-> '  10.0'}
Write (DispOut, #13 + 'Hallo');
```

Wie schon o.a. kann Write auch selbstdefinierte Gerätetreiber ansprechen. Hierbei sind jedoch einige Besonderheiten zu beachten:

Diese Prozedur muss einen Übergabe-Parameter vom Typ Byte oder Char besitzen, sie darf keine lokalen Variablen besitzen und damit auch keinen Frame. Weiterhin dürfen innerhalb der Prozedur nur die Pseudo-Accus "_ACCA, _ACCB, _ACCCHI und _ACCCLO = Z" benutzt werden was einen Aufruf von anderen Prozeduren und Funktionen sowie von Systemfunktionen ausschliesst.

Sollten trotzdem zusätzliche ACCUs für Zwischenspeicher gebraucht werden, sind diese vorher mittels Push/Pop zu retten. Diese Einschränkungen erzwingen in der Praxis, dass diese Prozedur komplett in Assembler zu schreiben ist. Der Übergabe Parameter befindet sich in _ACCA. Wichtig ist der Compiler-Schalter *{$NOFRAME}* **oder** *{$DEVICE}.*

```
{$NOFRAME}
Procedure TestDriver (const b : byte);
begin
  ASM;
    OUT   SPDR, _ACCA;      { SPI Data reg }
  ENDASM;
end;
```

4.14.3 WRITELN

WriteLn fügt an das Ende der Ausgabe ein CarriageReturn/LineFeed $0D+$0A an.

```
Procedure WriteLn (DeviceFunc : function; var str : string);

WriteLn (SerOut, 'Test');
WriteLn (SerOut);               // schreibt eine Leerzeile
```

4.14.4 READ

Einlesen eines Zeichens oder Strings ähnlich Write. Der erste Parameter ist eine Funktion, die als Ergebnis einen Character liefert. Der zweite Parameter ist eine String-Variable und der optionale dritte die Anzahl der zu lesenden Bytes oder das Ende-Zeichen.

Die Funktion Read unterscheidet drei Arbeitsweisen:

1. Eine String komplett einlesen. Der String wird ganz gefüllt
2. Eine bestimmte Anzahl von Zeichen einlesen. Dazu muss der 3. Parameter vom Typ **byte** sein.
3. Eine unbestimmte Anzahl von Zeichen lesen, bis das Zeichen erscheint, das im 3. Parameter vorgegeben wird. Dazu muss der 3. Parameter vom Typ **char** sein. Das Limiter-Zeichen wird in den String eingefügt, die Stringlänge jedoch um eins erniedrigt.

Ist der String kürzer als die Byte-Zahl, so wird der String auf die volle Länge beschrieben. Die weiteren zu lesenden Bytes werden zwar gelesen, aber nicht weiter beachtet. Das Längenbyte des Strings wird zum Schluss auf die Anzahl der gelesenen Bytes berichtigt, jedoch maximal auf Length(str).

```
Function Read (p : Function; var st : string);
Function Read (p : Function; var st : string; count : byte);
Function Read (p : Function; var st : string; limiter : char);

Read (func(2), st1, 4);        {st1 wird gefüllt}
Read (func, st2, 20};          {st2 wird mit 20 Zeichen gefüllt}
Read (func, st2, #0};          {st2 wird gefüllt bis eine #0 erscheint}
```

Die aufzurufende Funktion **kann** einen Parameter besitzen. Ist das der Fall muss dieser eine Konstante sein.

4.14.5 READLN

ReadLn liest solange von dem Device bis entweder ein CarriageReturn/LineFeed $0D+$0A (#13+#10) erscheint oder der bereitgestellte String gefüllt ist.

```
Procedure ReadLn (DeviceFunc : function; var str : string);

ReadLn (SerInp, st);
```

4.15 Fehlerbehandlung

4.15.1 RUNERR

Funktion. Lesen setzt das Boolean auf false. Viele Operationen setzen, abhängig von ihrem Ergebnis die Variable RunErr. Ein RunErr muss nicht unbedingt schwerwiegend sein, so dass das Programm nicht mehr sicher weiterläuft. Ein Fehler kann z.B. ein Integer-Overflow nach einer Multiplikation sein. Alle Operationen, die einen RunTimeError auslösen können, setzen ggf. ebenfalls das Flag. Im Gegensatz zur Prozedur RunTimeError ist RunErr nicht abschaltbar und immer aktiv.

Folgende Operationen können das Flag RunErr setzen:

Division
bei Division durch Null. In diesem Fall wird der höchstmögliche Wert zurückgegeben.
bei Float Overflow wird der höchstmögliche Wert zurückgegeben.
bei Float Underflow wird null zurückgegeben.

Multiplikation
bei Overflow. Resultat z.B. bei Byte $64 x $64 = $10
bei Float Overflow wird der höchstmögliche Wert zurückgegeben.
bei Float Underflow wird null zurückgegeben.

Addition
bei Overflow. Resultat z.B. bei Byte $84 + $84 = $08
bei Float Overflow wird der höchstmögliche Wert zurückgegeben.

SQR
bei Float Overflow wird der höchstmögliche Wert zurückgegeben.

SQRT
bei Float Underflow wird null zurückgegeben. Bei negativem Argument.

Typ Konvertierung
bei Overflow. Float to Word, to Int, to LongWord, to LongInt

StrToInt
bei fehlerhaftem String. Bei Overflow.

StrToFloat
bei fehlerhaftem String.

Indizierte String und Array Manipulation
bei falschem Index

4.15.2 RUNTIMEERR

Deklaration der Laufzeitfehler Prozedur.

Ist der Schalter ***RangeCheck*** “ein” und die Systemprozedur “RunTimeErr” importiert, so wird bei jedem Index Zugriff (string/array) nach dessen Berechnung geprüft, ob der Index gültig ist. Bei einem ungültigen Index wird diese Prozedur aufgerufen und im Arbeitsregister (_ACCB) der Fehlercode 1 übergeben.
Der Fehlercode 2 gibt eine illegale Stringlängen Operation an.

Durch den überlegten Einsatz der Compilerschalter(+/-) kann die Vergrösserung des Codes und die Laufzeitverlängerung in Grenzen gehalten werden. Ist ein Schalter beim Erreichen des Programmendes (end.) noch aktiv, so wird er auch für die System-Bibliothek verwendet.

Wurde die Prozedur “RunTimeErr” nicht importiert, haben die beiden Compilerschalter auch keine Bedeutung bzw. werden ignoriert. Stringkopier Statements (z.B. str:= ‘abcd’;) werden jedoch immer ohne Überlaufen ausgeführt. Hier ist eine zusätzliche Überwachung nicht notwendig.

Bitte auch **StackSize** und **Compiler Switches** beachten.

```
Define  StackSize  = 32, iData;  {32 bytes in iData}

procedure RunTimeErr;
begin
  DisableInts;
  {WatchDog ??}
  {_ACCB contains runtime error number}
  {0 -> software stack or Frame overflow}
  {1 -> string or array index error}
  {2 -> string length error}
  {3 -> reserved}
  {4 -> convert error float -> ordinal}
  {5 -> float overflow}
  {6 -> float underflow}
  ASM;
    ; store error
    MOV    errnum, _ACCB;
    ; do not use high level instr if Stack overflow is possible
    TST    _ACCB;
    ; if _ACCB = zero then stack error
    BRNE   NOTSTACK;
    ; do something with stack error
    NOP
    NOP
    NOTSTACK:
  ENDASM;
end;
```

Achtung:
RunTimeErr darf nicht vom Programm selbst aufgerufen werden!!

4.15.3 CLEARRUNERR

Setzt einen aufgetretenen Laufzeitfehler zurück

```
Procedure ClearRunErr;
```

4.16 Multi-Task Funktionen

Der AVRco enthält ein Multitasking System, das durch eine Vielzahl von Funktionen und Prozeduren unterstützt wird. Es können bis zu 15 Prozesse und Tasks definiert werden. Diese werden durch den Scheduler periodisch aufgerufen, abhängig von ihrer Priorität und Status.

Aufgaben können im Hintergrund erledigt werden, ohne Beteiligung des Hauptprogramms. Die Prozesse und das Hauptprogramm können durch Pipes und Semaphoren miteinander kommunizieren.

Tasks sind spezialisierte Prozesse, die zyklisch vom Scheduler aufgerufen werden, wobei das Zeitintervall konstant ist. Dadurch können Tasks solche Jobs erledigen, die in einem festen zeitlichen Raster ablaufen müssen, wie z.B. PID-Regler.

Nahezu alle folgenden Funktionen die einen Prozess/Task Namen erwarten, akzeptieren ebenso den Bezeichner "SELF". Damit ist es möglich den aufrufenden Prozess/Task zu manipulieren.

```
Sleep (self, 10);        //aktueller Prozess wird 10 Ticks schlafen gelegt
```

Weiterhin ist es meist möglich statt des Prozess/Task Namen die Prozess/Task ID zu benutzen.

```
Sleep (const ProzessID, Ticks: word);
```

ProzessID muss eine numerische Konstante sein (siehe unten).

Achtung!
Alle Wait, Sleep und Suspend Funktionen dürfen nur benutzt werden, wenn der Idle-Prozess importiert wurde oder wenn die Applikation sicherstellt, dass niemals alle Prozesse inkl. des Main schlafen, gesperrt sind oder warten.

4.16.1 SLEEP

Prozess Sleep in Ticks. Der Prozess/Task bleibt n Ticks inaktiv und wird dann vom Scheduler wieder “aufgeweckt”.

```
Procedure Sleep (p : process; t : word);
```

```
Sleep (process1, 50);
```

4.16.2 SUSPEND

Der Prozess/Task wird angehalten und bleibt bis zu einem “Resume” inaktiv. Der Prozess kann von sich aus nie wieder die Kontrolle erlangen. Deshalb muss ein Resume von ausserhalb erfolgen.

```
Procedure Suspend (p : process);
```

```
Suspend (process1);
```

4.16.3 SUSPENDALL

```
SuspendAll (Processes, Tasks);      // disable processes + tasks
SuspendAll (Processes);             // disable processes only
SuspendAll (Tasks);                 // disable tasks only
```

Ist ein *Idle Prozess* definiert, erhält dieser die Kontrolle. Ansonsten der *Main Prozess*.

4.16.4 RESUME

Ein mit “Suspend” oder “Sleep” deaktivierter Prozess/Task wird aktiviert.

Procedure *Resume (p : process);*

Resume (process1);

4.16.5 RESUMEALL

```
ResumeAll (Processes, Tasks);      // resume processes + tasks
ResumeAll (Processes);             // resume processes only
ResumeAll (Tasks);                 // resume tasks only
```

4.16.6 PRIORITY

Bei Prozessen wird mit Priority die Wichtigkeit bzw. der Rechenzeit Anteil eines Prozesses bestimmt. Der Prozess arbeitet die durch Priority vorgegebene Anzahl von SystemTicks ohne Unterbrechung. Bei Tasks bestimmt Priority das Intervall der Taskaufrufe in SystemTicks. Ein Task läuft maximal einen SysTick ohne Unterbrechung.
Siehe auch Kapitel *Mutitasking Programming – Priority* !

Process min/max Priority = 1..15 (Default=3)
Task min/max Priority = 1..255 (Default=5)

Achtung:
Die Task Priorität von 1 ist nur für besondere Zwecke geeignet, da dieser Task dann bei jedem SysTick vom Scheduler aufgerufen wird. Dann darf allerdings kein weiterer Task mehr aktiv sein.

Procedure *Priority (p : process; prio : byte);*

```
Priority (process1, 12);
Priority (task1, 5);
```

4.16.6.1 GetPriority

liefert die aktuelle Priorität eines Prozess/Task

Function *GetPriority (prcs : process|task) : byte;*

4.16.7 MAIN_PROC

Werden Prozesse verwendet, so ist das Hauptprogramms (Main) ebenfalls ein Prozess. Dessen Name ist mit “Main_Proc” festgelegt. *Lock*, *Unlock* und *Priority* haben auch Zugriff auf den Haupt-Prozess.

```
Priority (Main_Proc, 5);        //Default = 5

Lock (Main_Proc);
```

4.16.8 IDLE PROCESS

Wenn alle Prozesse inkl. des MAIN_PROC durch SLEEP, SUSPEND oder WAIT stillgelegt sind gibt es keinen Prozess der die Rechenzeit verbraucht. Dazu ist der optionale IDLE-Prozess implementiert, womit dieses Problem nicht mehr existiert. Ohne Idle Prozess ist es zwingend notwendig, dass zumindest MAIN immer läuft und damit die Rolle des IDLE Prozesses übernimmt.
So können alle Prozesse stillgelegt werden ohne dass das System instabil wird. Da dieser IDLE-Prozess auch Interrupts abarbeiten muss, braucht er ein Environment, d.h. eine Laufzeit Umgebung mit Stack und Frame.
Weiterhin vergrössert sich der Code/Flash Verbrauch etwas und der Scheduler wird minimal grösser und ein paar usec langsamer. Der Import erfolgt durch eine Erweiterung des Scheduler Defines.

Der Standard Define (ohne IDLE Prozess) sieht so aus
```
Define Scheduler = DataArea;
```

Mit Idle Prozess muss noch die IDLE-Stack und Framegrösse angegeben werden:
```
Define Scheduler = IdleStack, IdleFrame, DataArea;          //optionalen Idle Prozess benutzen
```

Die minimale Grösse für den Stack und Frame ist 10Bytes.

4.16.8.1 On Idle Process

Callback Prozedur. Der Idle Prozess muss definiert sein (beim Scheduler define).

```
Procedure OnIdleProcess;
```

Wenn im Anwendungsprogramm diese Prozedur implementiert ist

```
Procedure OnIdleProcess;
begin
  ...
end;
```

dann wird "OnIdleProcess" bei jedem start des Idle Processes aufgerufen.

Achtung:
Der Idle Process lebt immer nur einen SysTick (wie ein Task). Deshalb darf diese Prozedur nie länger als ein SysTick sein, sonst wird sie vom Scheduler automatisch abgebrochen.

4.16.9 SCHEDULE

Der Prozess/Task wird an dieser Stelle abgebrochen und die Kontrolle an den Scheduler übergeben. Dieser sucht jetzt nach dem nächsten Prozess mit der höchsten Priorität und aktiviert diesen.
```
Procedure Schedule;
```

4.16.10 SCHEDULER ON/OFF

Der Prozess-Scheduler kann mit "SchedulerOff" angehalten und mit "SchedulerOn" wieder gestartet werden. Ist der Scheduler angehalten, läuft nur noch der aktuelle Prozess/Task, der Scheduler wird komplett übersprungen.
```
Procedure SchedulerOff;
Procedure SchedulerOn;
```

4.16.11 GetSchedulerState

Der Staus des Prozess-Schedulers kann mit "GetSchedulerState" abgefragt werden. Ist der Scheduler angehalten, dann gibt diese Funktion ein FALSE zurück, ansonsten ein TRUE.
```
Function GetSchedulerState : boolean;
```

4.16.12 LOCK

Die ganze Rechenzeit des Prozessors wird einem Prozess zur Verfügung gestellt. Interrupts werden jedoch weiter beantwortet. Gilt auch für Main_Proc.

Procedure *Lock (p : process);*

Lock (process1);
Lock (Main_Proc);

Ein *Lock(Prozess)* verhindert das Weiterschalten des Schedulers auf einen anderen Prozess. Tasks sind davon jedoch nicht betroffen, da diese ähnlich wie Interrupts behandelt werden.

4.16.13 UNLOCK

Ein gelockter Prozess gibt den Prozessor wieder frei.

Procedure *UnLock (p : process);*

UnLock (process1);
UnLock (Main_Proc);

4.16.14 RESET PROCESS

Diese Funktion initialisiert einen Prozess komplett neu und setzt ihn auf suspended.
Sie darf aber nicht auf den aktuell laufenden Prozess angewendet werden deshalb muss dieser Prozess zuerst supendiert werden.

Procedure *ResetProcess (P : Name | i : ID);*

4.16.15 SEMAPHORE

4.16.15.1 WAITSEMA

Ein Prozess/Task schaltet sich inaktiv, bis eine spezielle Semaphore > 0 ist. In diesem Fall schaltet der Scheduler den Task/Prozess wieder aktiv. Die Semaphore wird automatisch um 1 dekrementiert.

Function *WaitSema (s : semaphore [; timeout: word]) : boolean;*

WaitSema (sema1);

Nur innerhalb eines Prozesses oder Tasks
Der *TimeOut* Parameter ist optional. Wird er weggelassen, muss der Prozess warten, bis die Semaphore > 0 ist. Das gleiche gilt auch, wenn TimeOut auf 0000 gesetzt wird. Mit einem Wert > 0 erfolgt nach (TimeOut * SysTicks) der Abbruch der Wait Funktion. Das Funktions Ergebnis ist true wenn kein Timeout aufgetreten ist. In Tasks ist die Angabe eines TimeOut nicht möglich bzw. wird ignoriert.

WaitSema darf nur benutzt werden, wenn der Idle-Prozess importiert wurde oder wenn die Applikation sicherstellt, dass niemals alle Prozesse inkl. des Main schlafen, gesperrt sind oder warten.

4.16.15.2 ProcWaitFlag

Mit WaitSema kann bei MultiTasking enorm Rechenzeit gespart werden. Allerdings erwartet diese Funktion eine spezielle Variable vom Typ Semaphore, die dann automatisch um 1 dekrementiert wird, wenn sie grösser 0 ist.

Darf die Semaphore vom Scheduler aber nicht verändert werden, oder soll sogar eine x-beliebige Variable vom Scheduler auf <> 0 überwacht werden, muss eine wesentlich generellere Funktion dies tun:

Function *ProcWaitFlag (Flag : var[; timeout : word]) : boolean;*

Hier kann Flag eine beliebige Variable sein, die auch nicht verändert wird. Ansonsten gilt dasselbe wie für WaitSema.

4.16.15.3 SETSEMA

Setzen einer Semaphore

Procedure *SetSema (sema : semaphore; v : byte);*

4.16.15.4 INCSEMA

Eine Semaphore wird um eins erhöht.

Procedure *IncSema (s : semaphore);*

IncSema (sema1);

4.16.15.5 DECSEMA

Eine Semaphore wird um eins erniedrigt. Wenn erfolgreich (Sema war > 0) wird ein true zurückgegeben, andernfalls ein false.

Function *DecSema (s : semaphore) : boolean;*

DecSema (sema1);

4.16.15.6 SEMASTAT

Der Inhalt der Semaphore wird abgefragt.

Function *SemaStat (s : semaphore) : byte;*

b:= SemaStat (sema1);

4.16.16 PIPES

4.16.16.1 WaitPipe

Ein Prozess/Task schaltet sich inaktiv, bis eine spezielle Pipe Daten hat. In diesem Fall schaltet der Scheduler den Task/Prozess wieder aktiv.

Function *WaitPipe (p : pipe [; timeout: word]) : boolean; {auch RxBuffer und RxBuffer1, -2, -3}*

WaitPipe (pipe1);
WaitPipe (RxBuffer);

Nur innerhalb eines Prozesses oder Tasks
Der TimeOut Parameter ist optional. Wird er weggelassen, muss der Prozess warten, bis das ein Datum in der Pipe steht. Das gleiche gilt auch, wenn TimeOut auf 0000 gesetzt wird. Mit einem Wert > 0 erfolgt nach (TimeOut * SysTicks) der Abbruch der Wait Funktion. Das Funktions Ergebnis ist true wenn kein Timeout aufgetreten ist. In Tasks ist die Angabe eines TimeOut nicht möglich bzw. wird ignoriert.

WaitPipe darf nur benutzt werden, wenn der Idle-Prozess importiert wurde oder wenn die Applikation sicherstellt, dass niemals alle Prozesse inkl. des Main schlafen, gesperrt sind oder warten.

4.16.16.2 PipeFlush

Eine Pipe komplett leeren.

Procedure *PipeFlush (p : pipe); {auch RxBuffer und RxBuffer1, -2,- 3}*

PipeFlush (pipe1);
PipeFlush (RxBuffer);

4.16.16.3 PipeSend

Ein Argument in eine Pipe einfügen (anhängen). Das Ergebnis zeigt an, ob die Operation erfolgreich war. Ist die Pipe voll, wird ein false zurückgegeben.

Function *PipeSend (p : pipe; v : type) : boolean;*

bo:= PipeSend (pipe1, value);

4.16.16.4 PipeRecv

Ein Argument aus einer Pipe abholen (entfernen). Die Function kehrt erst zurück, wenn die Operation erfolgreich war. Prozesse oder Tasks sollten “PipeStat” oder noch besser “WaitPipe” verwenden, um keine Rechenzeit zu verschwenden.

Function *PipeRecv (p : pipe) : type;*

val:= PipeRecv (pipe1);

4.16.16.5 PipeStat

Der Inhalt bzw. Parameterzahl einer Pipe wird abgefragt.

Function *PipeStat (p : pipe) : byte;*

b:= PipeStat (pipe1);

Die Funktion PipeStat ist auch auf RxBuffer, RxBuffer1, -2, -3 und TxBuffer, TxBuffer1, -2, -3 der seriellen Schnittstellen anwendbar.

4.16.16.6 PipeFull

Der Voll-Status einer Pipe wird abgefragt.

Function *PipeFull (p : pipe) : boolean;*

bo:= PipeFull (pipe1);

4.16.17 PROCESS ID

Manche, von mehreren Prozessen oder Tasks benutzte Funktionen oder Prozeduren müssen sich, abhängig vom aufrufenden Prozess bzw. Task, unterschiedlich verhalten. Eine solche Prozedur muss also feststellen können, welcher Prozess/Task sie aufgerufen hat. Dazu dienen die drei folgenden Funktionen.

Das Hauptprogramm (Main) hat immer die Prozess-ID 0 und trägt den Prozessnamen 'Main_Proc'.

4.16.17.1 ISCURPROCESS

Funktion. Abfrage ob der aktuelle Prozess/Task die ID x oder den Namen ‚abc' hat.

Function *isCurProcess (ID : byte[; Name : ProcName]) : boolean;*

```
bo:= isCurProcess (0);                    { 0 = Main }
bo:= isCurProcess (Main_Proc);
```

4.16.17.2 GETCURPROCESS

Die Prozess ID des aktuellen Prozesses/Tasks wird abgefragt.

Function *GetCurProcess : byte;*

Id:= GetCurProcess;

4.16.17.3 GETPROCESSID

Die Prozess ID eines Prozesses/Task wird abgefragt.

Function *GetProcessID (ProcName) : byte;*

Id:= GetProcessID (Main_Proc);

4.16.18 PROZESS STATUS

Der Status eines Prozesses oder Tasks kann abgefragt werden.
Dazu exportiert das System den Aufzählungstyp:

tProcessState = (eProcStop, eProcRun, eProcIdle, eProcWait, eProcSleep, eProcLock);

und die Funktion:

Function *GetProcessState (name : process|Task) : tProcessState;*

```
if GetProcessState (ProcessA) = eProcSleep then
  ...
endif;

case GetProcessState (Main_Proc) of
  eProcStop :
          inc(bb); |
  eProcRun  :
          inc(bb); |
  eProcIdle :
          inc(bb); |
  eProcWait :
          inc(bb); |
  eProcSleep:
          inc(bb); |
  eProcLock :
          inc(bb); |
endcase;
```

4.16.19 DEVICE LOCK

Für Device Treiber innerhalb MultiTasking ist es manchmal notwendig diese Treiber gegen andere Prozesse/Tasks zu sperren. Dazu wurde der Typ "*DeviceLock*" implementiert (nur globale Var, nicht in Arrays oder Records).
Prozesse, die ein Device/Treiber benutzen wollen, sollten ein solches Datum benutzen, um anderen Prozessen zu signalisieren, dass der Treiber momentan aktiv ist.

4.16.19.1 SetDeviceLock

```
Function SetDeviceLock (d : DeviceLock) : boolean;
```

Kehrt mit einem true zurück, wenn das Device frei war und setzt dabei den Wert auf locked.
Ist das Device im Moment schon belegt, kehrt die Funktion mit einem false zurück.
Bei einem "true" kann der Prozess jetzt das Device benutzen. Anschliessend muss er mit der Funktion *ClearDeviceLocked* den Treiber wieder freigeben.

4.16.19.2 ClearDeviceLock

```
Function ClearDeviceLock (d : DeviceLock) : boolean;
```

Gibt den Treiber wieder frei.
War der Treiber schon frei, kommt ein "*false*" zurück, ansonsten ein "*true*".

4.16.19.3 TestDeviceLock

Die Funktion überprüft eine solche Semaphore, ohne diese zu verändern. Sie gibt ein "*true*" zurück, wenn das Device frei ist, bei gesperrtem Device ein "*false*".

```
Function TestDeviceLock (d : DeviceLock) : boolean;
```

4.16.19.4 WaitDeviceFree

Function *WaitDeviceFree (s : DeviceLock [; timeout: word]) : boolean;*

Damit kann sich ein Prozess/Task solange schlafen legen, bis das Device frei ist.

Der TimeOut Parameter ist optional. Wird er weggelassen, muss der Prozess warten, bis das Device frei ist.
Das gleiche gilt auch, wenn TimeOut auf 0000 gesetzt wird.
Mit einem Wert > 0 erfolgt nach (TimeOut * SysTicks) der Abbruch der Wait Funktion.
Das Funktions Ergebnis ist true wenn kein Timeout aufgetreten ist.
Wenn kein TimeOut aufgetreten ist, setzt sie automatisch das "DeviceLock" auf *locked* bzw. gesperrt.
In Tasks ist die Angabe eines TimeOut nicht möglich bzw. wird ignoriert.

WaitPipe darf nur benutzt werden, wenn der Idle-Prozess importiert wurde oder wenn die Applikation sicherstellt, dass niemals alle Prozesse inkl. des Main schlafen, gesperrt sind oder warten.

```
var DevSema : DeviceLock;

Procedure Init;
begin
  ClearDeviceLock(DevSema);              // erste Freigabe des Devices
  ...
end;

Process CheckDevice(32,64 : idata);
begin
  WaitDeviceFree(DevSema);
  // enter the device driver
  ....
  // free the device driver
  ClearDeviceLock(DevSema);
  ...
end;
```

4.16.20 Stack und Frame Verbrauch

4.16.20.1 GETSTACKFREE

Ermittelt den Stack Verbrauch von Prozessen zur Laufzeit.
Die Funktionen liefert ein Wort mit der Anzahl von Bytes, die im jeweiligen Stack noch unbenutzt sind.

```
Function GetStackFree (p : Process)        : word;

ww:= GetStackFree (Main_Proc);
ww:= GetStackFree (Proc1);
```

4.16.20.2 GETTASKSTACKFREE

Ermittelt den Stack Verbrauch der Tasks zur Laufzeit.
Die Funktion liefert ein Wort mit der Anzahl von Bytes, die im Task Stack noch unbenutzt sind.

```
Function GetTaskStackFree : word;

ww:= GetTaskStackFree;
```

4.16.20.3 GETFRAMEFREE

Ermittelt den Frame Verbrauch von Prozessen zur Laufzeit.
Die Funktionen liefert ein Wort mit der Anzahl von Bytes, die im jeweiligen Frame noch unbenutzt sind.

```
Function GetFrameFree (p : Process)        : word;

ww:= GetFrameFree (Main_Proc);
ww:= GetFrameFree (Proc1);
```

4.16.20.4 GETTASKFRAMEFREE

Ermittelt den Frame Verbrauch der Tasks zur Laufzeit.
Die Funktion liefert ein Wort mit der Anzahl von Bytes, die im Frame Stack noch unbenutzt sind.

```
Function GetTaskFrameFree : word;

ww:= GetTaskFrameFree;
```

4.16.20.5 CHECKSTACKVALID

Ermittelt ob ein Stack Überlauf stattgefunden hat. Dann ist das Ergebnis = $FFFF.
Ansonsten liefert die Funktion den noch freien Stack Bereich.

```
Function CheckStackValid (p : Process|Task) : integer;
```

4.16.20.6 CHECKFRAMEVALID

Ermittelt ob ein Frame Überlauf stattgefunden hat. Dann ist das Ergebnis = $FFFF.
Ansonsten liefert die Funktion den noch freien Frame Bereich.

```
Function CheckFrameValid (p : Process|Task) : integer;
```

Beide Funktionen benötigen den System Import *StackChecks*
Wenn kein MultiTasking implementiert ist, darf auch kein Argument (Process/Task) angegeben werden, ansonsten gilt für das Hauptprogramm das Argument *Main_Proc*

4.16.21 SCHEDULER CALL BACK

Für Debugzwecke ist es manchmal wichtig die Zeit zu wissen, die der Scheduler braucht um von einem Prozess/Task zum anderen zu schalten. Da diese Zeit extrem von dem Programm abhängt (Anzahl der Prozesse/Tasks, Priorities etc.) kann dies nur zur Laufzeit exakt festgestellt werden.
Dazu wurden folgende Prozeduren vordefiniert:

```
Procedure OnSchedulerEntry;
Procedure OnSchedulerExit;
```

Wenn im Anwendungsprogramm diese Prozeduren implementiert sind:

```
Procedure OnSchedulerEntry;
begin
  ...
end;

Procedure OnSchedulerExit;
begin
  ...
end;
```

dann wird "OnSchedulerEntry" bei jedem Eintritt in den Scheduler aufgerufen und "OnSchedulerExit" bei jedem verlassen des Schedulers.
Damit lässt sich z.B. im Simulator die im Scheduler verbrauchte Zeit feststellen, indem auf beide Prozeduren jeweils ein Breakpoint gesetzt wird. Die zum Teil extrem variierenden Ergebnisse lassen sich auf den gerade abgelaufen Job des Schedulers zurückführen, Task/Prozesswechsel ja/nein etc.
Die ermittelte Zeit bzw. Zyklen geben auch die totale Sperrzeit des globalen Interrupts während des Schedulings wieder.

Achtung:

1. Es dürfen keine lokalen Parameter definiert werden.
2. Wenn Pascal Statements oder Register benutzt werden, müssen diese Register vorher gerettet werden.
3. Operationen, die eine Veränderung der CPU Flags hervorrufen, dürfen nur benutzt werden, wenn das Status Register der CPU zuvor gesichert wurde.

4.17 PID-Regler

Pseudo-Record
PID-Regler, wie er häufig in techn. Anwendungen eingesetzt wird, z.B. Temperatur Regelung, Servos, Drehzahlregler etc.

PID-Regler haben zwei Eingangsparameter: der Sollwert = '*Nominal'* und der Istwert = '*Actual'*.
Vier Parameter, die normalerweise nur einmal eingestellt werden, sind *pFactor*, *iFactor*, *dFactor* und *sFactor*.
Die Stellgrösse, die an den Aktuator (Heizung, Motor etc.) geht, wird durch die Funktion '*Execute'* errechnet.

Den Reglertyp bestimmen die zwei Initialisierungsparameter '*iLimit'* und '*dIntVal'*.

iLimit
ist vom Typ LongWord (0..100000) und bestimmt die maximale Grösse des **I**-Anteils (clipping). Ist iLimit = 0 wird der **Integral**-Wert des Reglers nicht berechnet und entfällt dadurch (z.B. PD-Regler).

dIntVal
ist vom Typ Byte (0, 1, 2, 4, 8, 16, 32) und bestimmt die Schrittweite zur Berechnung des **D**-Anteils (Steigung). Ist dIntVal = 0 wird der **Differential**-Wert des Reglers nicht berechnet und entfällt dadurch (z.B. PI-Regler). Ist der Wert = 1, wird die Steigung vom letzten Sollwert zum aktuellen Sollwert gerechnet. In den restlichen Fällen wird ein entsprechendes Array gebildet, das die Historie der letzten n Sollwerte aufnimmt. Dadurch kann die Steigung über eine grössere Anzahl von Sollwerten gebildet werden.

Der Regler rechnet intern mit **LongInteger**. Overflows bei Execute sind deshalb nicht zu erwarten.

Imports

Der Treiber muss wie gewöhnlich beim AVRco importiert werden.

```
Import SysTick, Pids, ..;


Var  Pid1 : PIDcontrol[iLimit, dIntVal];

{Init}
Pid1.pFactor:= 1000;
Pid1.iFactor:= 2500;
Pid1.dFactor:= 678;
Pid1.sFactor:= 10000;

{Run}
Pid1.Actual:= 500;
Pid1.Nominal:= 550;
PWM1:= Pid1.Execute;
```

4.17.1 pFACTOR

```
PIDname.pFactor:= p;
```

Verrechnungs-Faktor für den P-Wert. Der pWert ist die Differenz zwischen Sollwert (*Nominal*) und Istwert (*Actual*), auch als Fehler bezeichnet. Das execute berechnet den internen Wert pValue. *PIDname.pValue*

4.17.2 iFACTOR

PIDname.iFactor:= i;

Verrechnungs-Faktor für den I-Wert. Der iWert ist die Summe der Differenzen zwischen Sollwert (*Nominal*) und Istwert (*Actual*). Allgemein: Integral-Wert der Fehler.
Das execute berechnet den internen Wert iValue. *PIDname.iValue*

4.17.3 dFACTOR

PIDname.dFactor:= d;

Verrechnungs-Faktor für den D-Wert. Der dWert ist die Steigung der Differenzen zwischen Sollwert (*Nominal*) und Istwert (*Actual*). Allgemein: Fehler-Steigung.
Das execute berechnet den internen Wert dValue. *PIDname.dValue*

4.17.4 sFACTOR

PIDname.sFactor:= s;

Verrechnungs-Faktor für den Stell-Wert (Ergebnis). Der Stell-Wert ist die Summe aus P, I und D. Allgemein: ((P x pFactor) + (I x iFactor) + (D x dFactor)) div sFactor.

4.17.5 NOMINAL

PIDname.Nominal:= i;

Soll-Wert. Ergibt zusammen mit "*Actual*" (Ist-Wert) den Regelfehler, der in die P, I und D Werte eingeht.

4.17.6 ACTUAL

PIDname.Actual:= i;

Ist-Wert. Ergibt zusammen mit "*Nominal*" (Soll-Wert) den Regelfehler, der in die P, I und D Werte eingeht.

4.17.7 EXECUTE

i:= PIDname.Execute;

Funktion, die aus den vorgegebenen Werten (Soll, Ist und Faktoren) den Stellwert errechnet. Das Ergebnis ist ein Integer-Wert.

4.18 HardWare abhängige Funktionen

4.18.1 PocClock

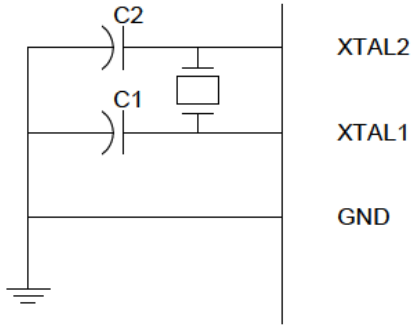
Prozessor Clock in Hertz. Wird für Software Delays, z.B. *Mdelay* und *uDelay,* gebraucht, ebenso zur Berechnung des SystemTicks. **Muss** definiert werden, nicht jedoch bei XMegas.

Define *ProcClock = 4000000;* *{4Mhz clock}*

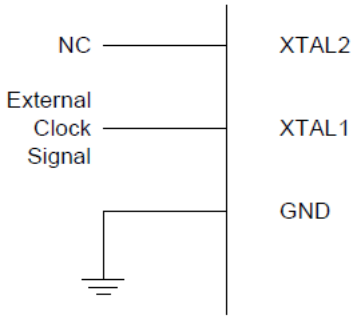
4.18.1.1 XMega ProcClock

Die XMegas haben keine Oscillator Fuses. Deshalb ist das Define des ProcClock hier nicht möglich und die Applikation muss die gewünschten Werte bzw. den Oszillator Typ bestimmen.
Mögliche OSC Typen: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz
Die intXXXhz wählen einen internen Oszillator aus, die extXXX einen externen.

extXTAL ext32kHz
min 440kHz max 16MHz



extClock
min 440kHz max 32MHz



Mögliche zusätzliche, meist optionale Parameter bestimmen dann den effektiven Prozessor und Peripheral Clock:
PLLmul, prescA, prescB, prescC, CLK256, CLK1K, CLK16K, LPM, FailDet, Lock, DFLLint, DFLLext.

Bei dem Clock Typ ***extXTAL*** kann die Start-Up Zeit angegeben werden:
CLK256, CLK1K oder CLK16K entspricht 256, 1024 oder 16000 Clock Zyklen.

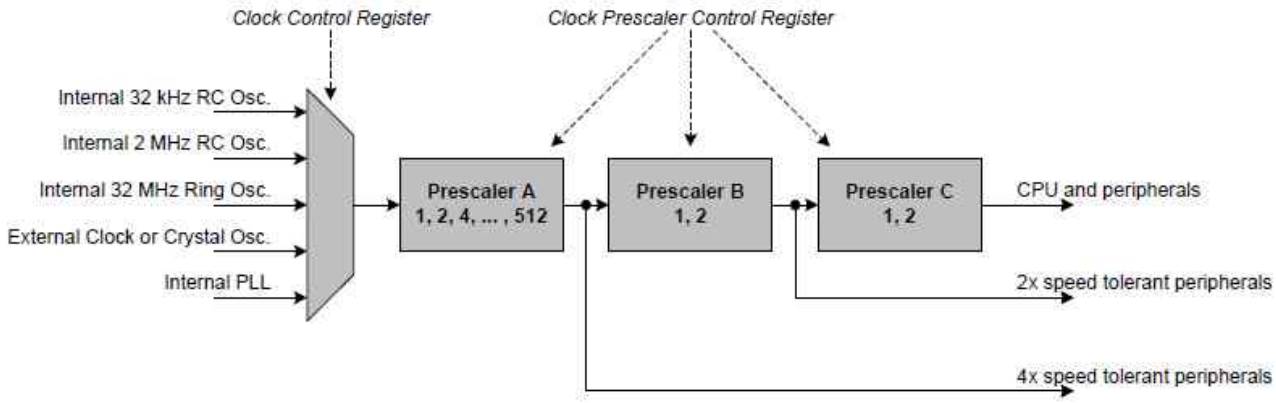
Wird mit einem externen Uhrenquarz (32kHz) gearbeitet so muss der Oszillator Typ ***ext32kHz*** gewählt werden. Dieser Oszillator Typ bietet auch die Option ***LPM*** an. Dann wird dieser Oszillator mit extrem niedrigen Strom Verbrauch betrieben. Voraussetzung ist ein Präzisions RTC Quarz.

Für extXTAL und extClock kann der optionale Parameter ***FailDet*** angegeben werden. Wenn dieser Clock ausfällt bzw. nicht vorhanden ist schaltet dann das Clock System automatisch auf int2MHz um. Wenn diese Option aktiv ist und der Oszillator ausfällt, dann wird auch immer der non-maskable Interrupt (NMI) *OSCF_INT* generiert!

Mit allen Oszillator Typen ist die Option ***Lock*** möglich. Diese bewirkt, dass nach einem *RESET* die Clock Einstellungen nicht zurückgesetzt werden (auf default Werte).

Die internen Oszillatoren können automatisch und kontinierlich kalibriert werden. Dazu sind zwei DFLLs vorhanden. Soll der interne 32kHz Oszillator dazu benutzt werden, so kann die Option ***DFLLint*** angegeben werden. Ist der Oszillator Typ ext32kHz im Betrieb, so kann alternativ die Option ***DFLLext*** benutzt werden.

Die Xmegas bieten eine interne PLL (phase locked loop) an. Durch die Angabe des optionalen Parameters ***PLLmul*** wird dem Prescaler A das Resultat der PLL zugeführt. Die PLL vervielfacht ihren Eingangs Clock im Bereich 1..31. Dabei ist zu beachten dass das Resultat der PLL 200MHz nicht überschreiten darf. Mögliche PLL Quellen sind: int2MHz, int32MHz, extClock > 440kHz, extXTAL > 440kHz.
Bei int32MHz wird dieser immer durch 4 geteilt, d.h. die PLL wird hier mit 8MHz gespeist!



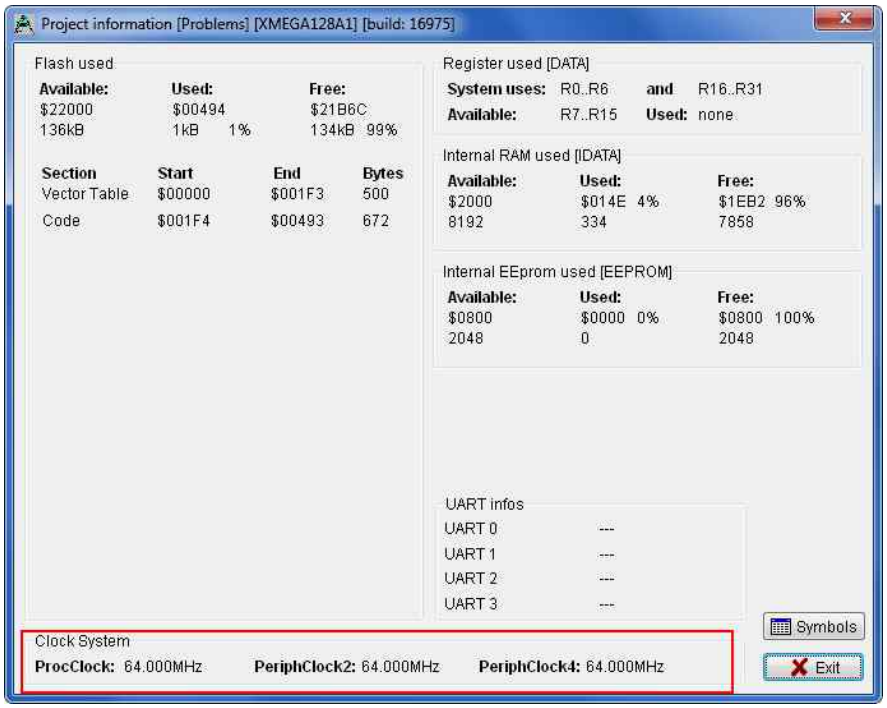
Alle Clock Typen werden immer dem Prescaler A zugeführt. Dieser teilt den Clock durch 1 bzw. in zweier Potenzen bis 512. Beispiel: prescA = 32. Dieser Clock wird immer dem HiRes module (extension of waveform generator) zugeführt = PeriphClock4. Der Ausgang des Prescalers B wird immer dem EBI (External bus interface) zugeführt) = PeriphClock4. Der Ausgang des Prescalers C steuert den Rest, CPU und Peripherie.

Durch eine entsprechende Kombination der Prescaler kann das System optimal eingestellt werden. Hat z.B. der PLL Ausgang 128MHz, und prescA=1, prescB=2 und prescC=2 dann läuft die CPU und Peripherie mit 32MHz, das EBI mit 64MHz und HiRes mit 128MHz. Wird die Angabe eines Prescalers beim Define weggelassen, erhält dieser automatisch den Wert 1. Beispiele von OSCtype Defines:

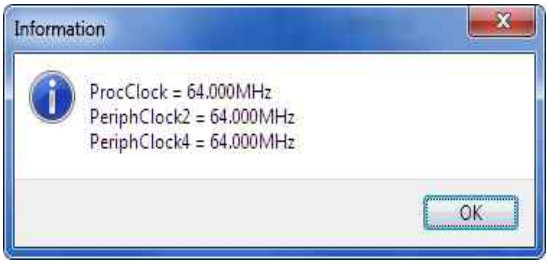
```
//>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
OSCtype      = int32MHz, PLLmul=4, prescB=1, prescC=1;
```

Ist identisch mit:

```
OSCtype      = int32MHz, PLLmul=4;
//>> CPU=32MHz, PeripherX4=128MHz, PeripherX2=64MHz
OSCtype      = int32MHz, PLLmul=16, prescB=2, prescC=2;
//>> CPU=16MHz, PeripherX4=16MHz, PeripherX2=16MHz
OSCtype      = int2MHz, PLLmul=8;
//>> CPU=10MHz, PeripherX4=20MHz, PeripherX2=20MHz
OSCtype      = extClock=5000000, PLLmul=8, prescA=2, prescB=1, prescC=2, FailDet;
```



Zur Kontrolle der Clock Resultate bietet die IDE PED32 nach erfolgreicher Compilation die Ergebnisse direkt an. Entweder im Dialog Project Informations oder via Maus Click. Dazu wird der Cursor auf das Wort *OSCtype* plaziert und die CTRL Taste gedrückt.



Wurden die Parameter so gewählt dass ein Clock > 32MHz erzielt wird, erfolgt eine Fehlermeldung. Wird ein "over clocking" gewünscht kann diese Fehler Meldung mit der Option "**overdrive**" unterdrückt werden.

```
OSCtype      = int32MHz, PLLmul=8, prescB=1, prescC=1, overdrive;   // 64MHz
```

4.18.2 STACKSIZE, RAMpage

Die gewünschte Stack Grösse (Software Stack) **muss** definiert werden. Wird zur Allocation und Überprüfung von Stack- und Programm Zwischen-Variablen gebraucht. Da das Stack während des Programmlaufs wächst und auch wieder schrumpft, und zwar von “oben” nach “unten”, die vom Programmierer verwendeten Variablen von “unten” nach “oben” aufgebaut werden, kann der Compiler eine Warnung ausgeben, wenn das Stack in die Variablen “hineinwachsen” könnte.

Um dem Compiler dazu eine Hilfe zu geben, denn die wirkliche Grösse des Stacks kann nur zur Laufzeit festgestellt werden, muss StackSize definiert werden. Bei einer Überschneidung von Programm-Variablen und Stack (+StackSize) gibt der Compiler eine Warnung aus.

Der minimale Wert von StackSize ist 16. Der gewünschte Ram-Bereich für das Stack muss ebenfalls angegeben werden (iData, xData).

```
Define  StackSize  = 32, iData;        {32 bytes in iData}
```

4.18.3 FRAMESIZE, RAMpage

Die gewünschte Frame Grösse (Übergabeparameter und lokale Variable) **muss** definiert werden. Wird zur Allocation und Überprüfung von Übergabeparametern (Prozeduren, Funktionen) und lokalen Variablen gebraucht. Da der Frame während des Programmlaufs wächst und auch wieder schrumpft, und zwar von “oben” nach “unten”, die vom Programmierer verwendeten Variablen von “unten” nach “oben” aufgebaut werden, kann der Compiler eine Warnung ausgeben, wenn dar Frame in die Variablen “hineinwachsen” könnte.

Um dem Compiler dazu eine Hilfe zu geben, denn die wirkliche Grösse des Frames kann nur zur Laufzeit festgestellt werden, muss FrameSize definiert werden. Bei einer Überschneidung von Programm-Variablen und Frame (+FrameSize) gibt der Compiler eine Warnung aus.

Der minimale Wert von FrameSize ist 8. Der gewünschte Ram-Bereich für den Frame muss ebenfalls angegeben werden (iData, xData).

```
Define  FrameSize = 32, iData;        {32 Bytes in iData}
```

4.18.4 TASKSTACK, RAMpage

Die gewünschte Stack Grösse (für alle Tasks gemeinsamer Stackbereich) **muss** definiert werden, wenn Tasks importiert wurden. Wird zur Allocation des Speichers benötigt.
Der minimale Wert von TaskStack ist 8. Der gewünschte Ram-Bereich für den Stack muss ebenfalls angegeben werden (iData, xData).

```
Define  TaskStack = 32, iData;      {32 bytes in iData}
```

4.18.5 TASKFRAME

Die gewünschte Frame Grösse (für alle Tasks gemeinsamer Framebereich) **muss** definiert werden, wenn Tasks importiert wurden. Wird zur Allocation des Speichers benötigt.
Der minimale Wert von TaskFrame ist 8. Der Ram-Bereich für den Frame ist der gleiche wie bei TaskStack (iData, xData).

```
Define  TaskFrame = 16;             {16 bytes}
```

4.18.6 SCHEDULER

Die gewünschte Speicherseite (iData, xData) für die Prozess- und Taskverwaltung **muss** definiert werden, wenn Prozesse oder Tasks importiert wurden.

Der Scheduler erledigt die Task/Prozess Verwaltung und schaltet diese um. Der Scheduler wird vom Timer0 bzw. SysTick aufgerufen. Da dies ein Interrupt ist, bleibt der globale Interrupt gesperrt, bis der Scheduler seine Arbeit erledigt hat. Diese Zeit kann, je nach Aufgaben des SysTicks, Anzahl der Prozesse etc. bis zu 500usec dauern. Wenn nun die Gefahr besteht, dass z.B. schnelle Interrupts nicht beantwortet werden können, da der Interrupt global noch gesperrt ist, kann durch die zusätzliche Anweisung “interruptible” der globale Interrupt innerhalb des Timer0 Ints sofort wieder freigegeben werden.

```
Define  Scheduler = iData;                {Scheduler in iData}
Define  Scheduler = iData, interruptible; {Scheduler in iData, nicht bei XMegas}
```

4.18.7 SYSTICK

Timergesteuerter Interrupt für Zeitfunktionen. Wurde SysTick über die *IMPORT* Klausel importiert, muss auch noch die gewünschte Tick-Zeit angegeben werden. Der Wert kann 0.1..100 (msec) betragen. Der Wert für das Define kann auch im Floating Point Format angegeben werden. Damit lässt sich der Tick relativ genau den Bedürfnissen anpassen. SysTick wird als Hardware-Interrupt eines Timers implementiert. Beim AVR ist dies normalerweise der Timer0 (8bit Timer). Besitzt die gewählte CPU den Timer2, ebenfalls ein 8bit Timer, kann auch dieser gewählt werden. Der verwendete Timer ist jetzt für das Programm selbst nicht benutzbar. Manipulationen an der Timerhardware oder Register können das Programm zum Absturz bringen.

Ein guter Wert für den Tick ist 10 (msec). Damit macht der Interrupt das System nicht *dicht* und die Aufgaben des SysTicks, z.B. Entprellen des *SwitchPort* lassen sich gut erledigen. Beim AVR ist *SysTick bzw. Timer0 (bzw. Timer2)* der einzige Interrupt, der vom System benutzt und auch direkt ausgewertet wird. Zusätzliche Hardware Interrupts sind vom Programmierer selbst zu programmieren und in der vordefinierten Prozedur *Interrupt xxx;* komplett abzuhandeln.

Der Timer wird aus der Variable "*SysTickTime*" geladen wird. Die Applikation hat damit die Möglichkeit durch Ändern dieser Variablen den SysTick in bestimmten Grenzen zur Laufzeit zu ändern.

SysTick ist für viele Treiber unbedingt notwendig.
Im Normalfall **muss** also **SysTick** immer **importiert** und **definiert** werden.

```
Import   SysTick;
Define   ProcClock = 4000000;     {4Mhz clock }
         SysTick   = 10;          {10msec Tick}
oder
         SysTick   = 5.5;         {5.5msec Tick}
oder
         SysTick   = 8.0, Timer2; {8msec Tick}
```

4.18.7.1 XMega SYSTICK

Bei den XMega Typen gibt es ein paar Unterschiede bei der Definition wegen der unterschiedlichen Timer. Der SysTick wird hier nicht durch den Timer0 oder Timer2 erzeugt sondern durch den internen 16bit 32kHz RTC Timer. Dessen Interrupt kann in Schritten von 1msec eingestellt werden, also 1, 2, 3, 4 …

Die interne Auflösung des RTC32K ist exakt 1.024msec und sehr stabil. Allerdings lassen sich damit keine genauen msec Schritte erzielen, was in den meisten Fällen auch nicht notwendig ist.

```
         SysTick   = 10;          {10msec Tick, 1..20msec}
```

Ein Fine-Tuning über die Variable *SysTickTime* ist hier nicht möglich. Ebenso wenig über das Attribut **ADJ**. Bedingt durch die interne Logik dieses Timers ist ein SysTick von 1msec fehlerhaft. Hier wird nur ca. 1.7msec erreicht. Der SysTick sollte deshalb nicht kleiner als **5msec** sein.

4.18.7.2 XMega SYSTICK ohne RTC Timer

Manche XMegas, z.B. XMega256A3B, haben keinen 16bit RTC Timer. Um auch hier einen SysTick benutzen zu können gibt es die Möglichkeit einen beliebigen 16bit Standard Timer zu benutzen. Dies muss beim SysTick Define festgelegt werden:

```
Define  SysTick      = 10, User;   // user = use any regular 16bit Timer
```

Der SysTick Job muss nun an einen Timer gekoppelt werden. Das kann durch die User Implementation eines Timer Interrupts geschehen, wobei die Initialisierung des Interrupts durch die Applikation durchgeführt werden muss. In der Interrupt Service Routine dieses Timer Interrupts muss dann der SysTick Job aufgerufen werden:

```
{$NoSave}              // standard register save is sufficient
Interrupt TCD0_INTOVF;
begin
  ASM: CALL System.$INTERRUPT_SYSTICK_USER;
end;
```

Eine elegantere Möglichkeit besteht darin einen TickTimer zu importieren und dessen Interrupt Callback zu benutzen:

```
Import SysTick, …, TickTimer;
…
Define  TickTimer    = Timer_C0;   // use Timer_C0 and no PortPin
…
Procedure onTickTimer;        // onTickTimer(SaveAllRegs); SaveAllRegs not necessary here
begin
  ASM: CALL System.$INTERRUPT_SYSTICK_USER;
end;
…
begin
  EnableInts($87);
  TickTimerTime(10000);    // 10000usec = 10msec
  TickTimerStart;
```

4.18.7.3 OnSysTick

Für Timing Aufgaben, die an den SysTick gekoppelt werden müssen, wurde die Prozedur "*OnSysTick*" implementiert. Da dies ein call-back aus dem Rimer Interrupt ist, ist innerhalb „onSysTick“ der globale

Interrupt gesperrt. Wie bei allen call-backs aus Interrupts muss man hier sehr sorgfältig sein und möglichst nur sehr kurzen und schnellen Code verwenden um eine allzu lange Interrupt Sperrung zu vermeiden. "OnSysTick" wird bei jedem SysTick (Timer Interrupt) aufgerufen.

```
Procedure OnSysTick;
Procedure OnSysTick (SaveAllRegs);
```

Implementation:

```
Procedure OnSysTick;
begin
  ...
end;
```
oder
```
procedure OnSysTick (SaveAllRegs);   // not possible if MultiTasking is used!
begin
  ...
end;
```

Achtung:
Bei ***MultiTasking*** Import darf "*SaveAllRegs*" nicht verwendet werden. Hier gibt es aus internen Gründen nur die Standard Sicherung von ACCA, ACCB, ACCLO, ACCHI. Wenn der globale switch *{$NOSHADOW}* aktiv ist, dann werden alle Register im non-multitasking Mode im SysTick gesichert.
Bei ***non-MultiTasking*** und dem Zusatz "*SaveAllRegs*" gibt es keine Limits, bedenken Sie aber dass alle Register retten und wieder restoren seine Zeit braucht.
Bei MultiTasking und nicht-MultiTasking ohne den Zusatz "*SaveAllRegs*" gilt:

1. Es dürfen keine lokalen Parameter definiert werden.
2. Die Register SREG, _ACCA/R17, _ACCB/R16, _ACCCLO/R30 und _ACCCHI/R31 werden automatisch gesichert. Wenn Pascal Statements oder andere Register benutzt werden, müssen diese Register vorher gerettet werden.

Kann *SaveAllRegs* nicht verwendet werden dann können alternativ innerhalb onSysTick und allen anderen Callbacks auch diese Funktionen eingesetzt werden:

```
Procedure PushRegs;   // push regs _ACCALO.._ACCFHI
Procedure PopRegs;    // pop  regs _ACCALO.._ACCFHI
```

4.18.7.4 SysTickStop

Für spezielle Zwecke zur SysTick Manipulation

```
Procedure SysTickStop;          // Disable Timer Interrupt, STOP Timer
```

Diese Funktionen muss immer paarweise mit *SysTickStart* benutzt werden.

4.18.7.5 SysTickStart

Für spezielle Zwecke zur SysTick Manipulation

```
Procedure SysTickStart;         // Start Timer Interrupt, preset TCNT, Start Timer
```

Diese Funktionen muss immer paarweise mit *SysTickStop* benutzt werden.

4.18.7.6 SysTickRestart

Initialisiert den Hardware Timer, der den SysTick triggert, komplett neu.

4.18.7.7 SysTickDisable

Für spezielle Zwecke zur SysTick Manipulation

```
Procedure SysTickDisable;       // nur Timer Interrupt disablen, Timer läuft noch.
```

Diese Funktionen muss immer paarweise mit *SysTickEnable* benutzt werden.

4.18.7.8 SysTickEnable

Für spezielle Zwecke zur SysTick Manipulation

```
Procedure SysTickEnable;          // nur Timer Enablen
```

Diese Funktionen muss immer paarweise mit *SysTickDisable* benutzt werden.

4.18.7.9 SystemTime

Für manche Zwecke ist eine Systemzeit sehr hilfreich. Das ist keine Uhrzeit sondern die Anzahl der SysTicks seit dem letzten Startup/Reset. Dazu wird der SysTick benötigt sowie ein Import der System Zeit:

```
From SysTick Import SystemTime16;   // 16bit word system time
```
oder
```
From SysTick Import SystemTime32;   // 32bit longword system time
```

Mit diesem Import wird eine Variable definiert, entweder als WORD oder LONGWORD. Diese wird mit jedem SysTick inkrementiert und kann von der Applikation gelesen und geschrieben werden. Durch das Attribut ***locked*** erfolgt bei jedem Zugriff der Applikation auch eine Interrupt Sperrung!

```
Var SystemTime16 : word; locked;        // 16bit word system time
```
oder
```
Var SystemTime32: longword; locked;   // 32bit longword system time
```

4.18.8 ENABLEINTS

Gibt den globalen Interrupt frei. Es ist oft notwendig, bevor der SystemTick anläuft, dass das Hauptprogramm noch diverse Initialisierungen (Ports etc) vornehmen kann. Deshalb wird der globale Interrupt im System selbst, auch wenn z.B. Interrupts importiert/definiert wurden (SysTick, Serial etc), nicht freigegeben. Das Anwenderprogramm kann und **muss** deshalb die System-Prozedur "*EnableInts*" einmal aufrufen, damit Interrupts möglich werden.

```
{Program Body = Main}
begin
  ...
  EnableInts;
  Loop
    ...
  EndLoop;
End.
```

Xmega

Bei den XMegas muss dieser Funktion auch noch der gewünschte Wert mitgegeben werden, der bestimmt welche Interrupt Levels freigegeben werden sollen. Dieser Wert wird in das PMIC Control Register geschrieben. Standard Wert ist $87 = alle Levels freigegeben.

```
  Procedure EnableInts(level : byte);
```

Sollen in der Folge wieder irgendwo nur die Interrupts freigegeben werden, genügt dort ein

```
  EnableIntsX;
```

um ein Ändern der Interrut Level dort zu verhindern.

4.18.9 START_PROCESSES

Startet Prozesse und Tasks. Setzt den Timer (SysTick) zurück und gibt den globalen Interrupt frei. Diese Prozedur **ersetzt** die Prozedur **EnableInts** wenn Prozesse oder Tasks importiert wurden.
Es ist oft notwendig, bevor der SystemTick/Prozesse/Tasks anlaufen, dass das Hauptprogramm noch diverse Initialisierungen (Ports etc) vornehmen kann. Deshalb wird der globale Interrupt im System selbst, auch wenn z.B. Prozesse importiert/definiert wurden (SysTick, Serial etc), nicht freigegeben. Das Anwenderprogramm **muss** deshalb die System-Prozedur "*Start_Processes*" einmal aufrufen, damit Interrupts möglich werden.

```
{Program Body = Main}
```

```
begin
  ...
  Start_Processes;
  Loop
    ...
  EndLoop;
End.
```

Xmega
Bei den XMegas muss dieser Funktion auch noch der gewünschte Wert mitgegeben werden, der bestimmt welche Interrupt Levels freigegeben werden sollen. Dieser Wert wird in das PMIC Control Register geschrieben. Standard Wert ist $87 = alle Levels freigegeben.

```
Procedure Start_Processes(level : byte);
```

Wenn in der Folge die Interrupt Prioritäten geändert warden sollen dann reicht ein *EnableInts(x)* aus.

Sollen in der Folge wieder irgendwo nur die Interrupts freigegeben werden, genügt dort ein

```
Procedure EnableIntsX;
```

um ein Ändern der Interrut Level dort zu verhindern.

4.18.10 DISABLEINTS

Sperrt den globalen Interrupt. Es werden jetzt keine Interrupts mehr akzeptiert. Interrupts sollten nicht allzu lange gesperrt bleiben. Freigabe der Interrupts mit EnableInts.

```
Procedure DisableInts;
```

4.18.11 NOINTS, RESTOREINTS

EnableInts und DisableInts haben noch eine weitere Funktion, sie setzen/rücksetzen ein System Flag, so dass System Funktionen immer wissen, welchen Interrupt Status die Applikation hat. Wenn das System den Interrupt sperren muss, darf es diesen nur wieder freigeben, wenn dieser vor der Sperrung auch schon frei war. Ähnliches gilt auch für Teile der Applikation wo der aktuelle Interrupt Status unbekannt ist.

Dieses Handling wird durch NoInts und RestoreInts unterstützt. NoInts sperrt den Interrupt ohne Rücksicht auf dessen aktuellen Status, verändert aber das System Flag nicht. RestoreInts gibt den Interrupt nur wieder frei, wenn das System Flag gesetzt ist, also der Interrupt vor NoInts auch freigegeben war.

Diese beiden Funktionen sind mit grosser Voisicht zu benutzen!

4.18.12 CPUSLEEP

Legt die CPU schlafen, d.h. das Programm wird gestoppt und interne Aktivitäten der CPU werden abgeschaltet. Die CPU wacht erst bei einem externen Interrupt oder Reset wieder auf.
Der Parameter der Prozedur wird in das CPU Steuerregister geschrieben (mcucr).

```
Procedure CpuSleep (sleepcmd : byte);

CPUsleep (MCUCR or $30);          // absolute powerdown
```

Beispiel Programm:

Ein Programmbeispiel finden Sie im Verzeichnis **..\E-Lab\AVRco\Demos\Sleep**
Es besteht aus den drei einzelnen separaten Programmen die die drei Sleep Modi des AVRs auf einfachste Weise darstellt.

XMega
Bei den XMegas wird ein speziell dafür vorgesehenes Steuerregister ***SLEEPCTRL*** verwendet. Hier kann eine grosse Zahl verschiedener Sleep bzw. PowerDown Modi eingestellt werden. Abschalten ganzer IO-Bereiche, Clocks, Interrupts etc. Der Parameter sleepcmd wird in dieses Register geschrieben.

4.18.13 HardwareReset XMega only

Die CPU führt einen echten Hardware Reset durch, das gleiche wie ein low am externen Reset Pin.

Procedure *HardwareReset;*

4.18.14 POWERSAVE

Legt die CPU n SysTicks schlafen. SysTick muss importiert sein. Der Parameter “mode” wird in das CPU Steuerregister geladen (siehe CPUsleep), der Parameter “ticks” bestimmt die zeitliche Länge des PowerDown, gezählt in SysTick Zyklen. Durch eine geeignete Wahl des Parameters “mode” muss sichergestellt werden, dass zumindest der für den SysTick verantwortliche Timer weiterarbeitet.
Der SysTick Timer weckt die CPU mit jedem Tick. Der Parameter “Ticks” wird um 1 erniedrigt und wenn er nicht null ist, geht die CPU wieder in den PowerDown Mode.

Procedure *PowerSave (**const** mode : byte; **const** ticks : word);*

XMega
Bei den XMegas wird ein speziell dafür vorgesehenes Steuerregister ***SLEEPCTRL*** verwendet. Hier kann eine grosse Zahl verschiedener Sleep bzw. PowerDown Modi eingestellt werden. Abschalten ganzer IO-Bereiche, Clocks, Interrupts etc. Der Parameter sleepcmd wird in dieses Register geschrieben. Es muss aber dafür gesorgt werden dass der SysTick weiterläuft.

4.18.15 WATCHDOG

Importiert bei “Import” den WatchDog. Bei “Define” wird der Vorteiler des Timers eingestellt, falls vorhanden.

Import *SysTick, WatchDog;*

```
Define  SysTick  = 4000000;        { 4MHz }
        WatchDog = 7;              { WatchDog Precaler}
```

Es ist auch eine besser lesbare Definition möglich:

```
Define WatchDog = msec16;        // msec32, msec64, msec125, msec250, msec1000, msec2000
```

Bei den **XMegas** ist nur dieses Define zulässig:

```
Define WatchDog = msec16;        // msec8, msec16, msec32, msec64, msec125, msec250,
                                 // msec1000, msec2000, msec4000, msec8000
```

4.18.16 WATCHDOGSTART

Initialisiert und startet den WatchDog, falls vorhanden.

Procedure *WatchDogStart;*

4.18.17 WATCHDOGSTOP

Stoppt den WatchDog, falls vorhanden.

Procedure *WatchDogStop;*

4.18.18 WATCHDOGTRIG

Triggert den Hardware WatchDog der CPU. Die generelle Freigabe des Watchdogs ist CPU-abhängig und muss ggf. über Device oder Import/Define eingestellt werden.

Procedure *WatchDogTrig;*

4.18.19 GETWATCHDOGFLAG

Funktion. Liefert ein Byte mit einer Kopie des "MCUSR" Register Inhaltes wie er unmittelbar nach dem PowerOn war. Damit kann die Applikation feststellen was der Grund für den letzten RESET war indem sie dieses Byte auswertet.
Die enthaltenen Bits haben aber nicht bei allen CPUs immer die gleiche Bedeutung

Function *GetWatchDogFlag : byte;*
Bemerkung: die alte Funktion "*WatchDogFlag*" wird durch die neue "*GetWatchDogFlag*" ersetzt.

4.18.20 {$NOWATCHDOGAUTO}

Compiler Schalter, wenn aktiv, dann erfolgen keine automatischen Watchdog Triggers in den Delays (mDelay etc) als auch in den systembedingten Warteschleifen. Nur für besondere Applikationen.

4.18.21 ENABLE_JTAGPORT

Will man das JTAG Port eines AVRs benutzen, ohne dass die JTAG Programmierung komplett abgeschaltet wird, muss die Applikation zur Laufzeit das JTAG Port einschalten können.

Ist JTAG (per Fuse Bit) enabled, kann die Applikation zur Laufzeit steuern, ob diese Pins tatsächlich für JTAG oder dennoch für "alternate functions" benutzt werden sollen.
Diese Funktion schaltet die entsprechenden Pins für **JTAG** frei.

Procedure *Enable_JTAGport;*

4.18.22 DISABLE_JTAGPORT

Will man das JTAG Port eines AVRs benutzen, ohne dass die JTAG Programmierung komplett abgeschaltet wird, muss die Applikation zur Laufzeit das JTAG Port ausschalten können.

Ist JTAG (per Fuse Bit) enabled, kann die Applikation zur Laufzeit steuern, ob diese Pins tatsächlich für JTAG oder dennoch für "alternate functions" benutzt werden sollen.
Diese Funktion schaltet die entsprechenden Pins für die **"alternate functions"** z.B. ADC frei.

Procedure *Disable_JTAGport;*
Achtung:
Sollten System Treiber (z.B. SysLedBlink) die auch vom JTAG benutzten Pins beim Startup schon initialisiert haben, dann geht diese hiermit verloren! Die Applikation muss dann nach Disable_JTAGport evtl. das DDR und das Port Register richtig setzen!

4.18.23 Debugger Breakpoints

AVR Typen mit JTAG und XMegas bieten mächtigen on-chip Debug Support. Da hier aber nur ein paar wenige breakpoints beim runtime Debugging möglich sind, ist es möglich eine unbegrenzte Anzahl von BreakPoint Opcodes zur compile Zeit in die Source setzen. Diese zwingen die CPU beim erreichen eines solchen Opcodes anzuhalten und auf den in-circuit Debugger zu warten. Aber bitte nicht vergessen diese Opcodes nach der Debug Session wieder aus der Source zu entfernen.

Ist eine CPU nicht im Debug Mode, dann wird ein solcher Breakpoint von der CPU als NOP behandelt.

Deshalb kann ein Breakpoit auch in der Source belassen werden. Um jetzt dann an bestimmten Stellen nicht mehr benötigte Breakpoints inaktiv zu schalten, können diese beide globalen Compiler Schalter benutzt werden:

```
{$Disable_DEBUG_BREAK}
  …
  …
{$Enable_DEBUG_BREAK}

Procedure Debug_Break;
```

4.19 XMega Support Funktionen

Die XMega Familie bieten diverse Erweiterungen an, wie z.B. Read Fuses, Read/Write LockBits, Read Serial Number, Read User Signature Row. Der AVRco unterstützt diese nützlichen Funktionen.

Achtung: Während der Ausführung dieser Funktionen kann der Interrupt kurzzeitig gesperrt sein. Weiterhin kann es ebenfalls kurzeitig zur Absenkung des System Clocks kommen, das die UARTs in diesem Moment beeinflusst wenn sie am Senden oder Empfangen sind!

Import
Import *SysTick, … XMegaSupport;*

Function *ReadDeviceID : longword;*
Gibt die DeviceID des Controllers zurück. XMega128A1 = $001E974C

Function *ReadDeviceRev : char;*
Gibt die Device Revision des Controllers zurück. Z.B. “G“

Function *ReadFuseByte0 : byte;*
Function *ReadFuseByte1 : byte;*
Function *ReadFuseByte2 : byte;*
Function *ReadFuseByte3 : byte;*
Function *ReadFuseByte4 : byte;*
Function *ReadFuseByte5 : byte;*
Gibt den Inhalt des Fuse Bytes zurück.

Function *ReadLockByte : byte;*
Gibt den Inhalt des Lock Bytes zurück.

Procedure *WriteLockByte(lb : byte);*
Verändert das Lock Byte. Es werden aber nur bits auf 0 gesetzt. 1-bits werden ignoriert!
Das LockByte kann nur mit einem externen ChipErase komplett zurückgesetzt werden ($FF).

Function *ReadDevSerNum: word64;*
Ermittelt eine 8 Byte Seriennummer anhand der internen Lotnummer, Wafernummer und Koordinaten.

Allgemeine Funktionen zum Auslesen der ***ProductionRow*** und der ***UserRow***:
Function *ReadProductionRow(loc : word) : byte;*
Function *ReadUserRow(loc : word) : byte;*
Procedure *WriteUserRow(loc : word; b : byte);*

Ein Beispiel Programm befindet sich in der Demos Directory unter “XMEGA_Support“.

4.20 XMega UserSignatureRow

Die XMega Typen haben neben Register, SRAM, IO-Bereich, Flash und EEprom einen weiteren Speichertyp erhalten, die UserSignatureRow. Dies ist ein separater Flash Speicher in der Grösse einer Flash Page.
Bei den kleineren Typen (<128kB) ist dieser bereich 256 Bytes gross, bei den grösseren (>=128kB) sind das 512 Bytes.

Dieser Bereich ist wie das Flash selbst für die Applikation nur lesbar und liegt ausserhalb des normalen Speicherbereichs, wie z.B. das EEprom. Ein Chip Erase durch den Programmer löscht diesen Speicher nicht, sondern er muss gezielt direkt durch den Programmer gelöscht werden. Das hat den Vorteil dass einmal darin gespeicherte Daten durch eine normale Neu-Programmierung nicht gelöscht werden, vorausgesetzt der Programmer wird nicht dazu angewiesen (checkbox program UserRow).

Diese Eigenschaft macht diesen Speicher optimal für Serien Nummern, Kalibrierungs Daten etc. Also Daten die normalerweise nie wieder geändert werden.

Da diese Daten aber dem Programmier Gerät zur Verfügung stehen müssen, bietet es sich an dass das AVRco System ein entsprechendes Hex-File erstellen kann. Dazu muss das System neben dem EEprom einen weiteren zusätzlichen Schalter im Konstanten Bereich anbieten *{$UserRow}*. Konstante, die in diesen Bereich hineingelegt werden, können dann ganz normal durch die Applikation gelesen werden.

Das AVRco System generiert daraus ein Hex-File ***name*.USR** welches von den Programmer verarbeitet werden kann. Beispiel:

```
{-------------------------------------------------------------}
{ Type Declarations }
type

{-------------------------------------------------------------}
{ Const Declarations }
const
{$UserRow}
  uSerNum[$1f8]  : word64    = $1234567890ABCDEF;
  urB            : byte      = $AA;
  urW            : word      = $1234;
  urI            : integer   = 1234;
  urLW           : longword  = $12345678;
  urLI           : longint   = -12345678;
  urW64          : word64    = $1234567890AABBCC;
  urI64          : int64     = -123456789054321;
  urSt           : string    = 'abcdefg';
{-------------------------------------------------------------}
{$IDATA}
...
  st:= urSt;
  bb:= urB;
  ww:= urW;
  ii:= urI;
  Lw:= urLW;
  Li:= urLi;
  i64:= urI64;
  w64:= urW64;
  w64:= uSerNum;
```

Achtung:
Es werden hier nur Zahlen und Strings unterstützt, keine Arrays oder Records.
Am Ende einer solchen Konstanten Liste muss unbedingt ein Speichertyp Wechsel erfolgen *{$IDATA}*, so dass evtl. weitere „normale“ Konstante nicht in diesem Bereich abgelegt werden.

Ein Beispiel Programm befindet sich in der Demos Directory unter “XMEGA_Support“.

4.21 EEPROM

Manche der älteren Singlechip Typen besitzen internen EEprom Speicher. Die meisten neueren Versionen sind grundsätzlich mit EEprom ausgerüstet.

Dabei stellt sich das Problem, dass der Zugriff darauf in der Regel über Index- bzw. Adressregister läuft, was der normalen Adressierung innerhalb eines Compilers widerspricht. Zumindest ist jedoch das Beschreiben des EEproms mit zusätzlichem Aufwand verbunden. Eine Zugriffsimplementierung direkt in Hochsprache durch das Anwendungsprogramm ist möglich, jedoch nicht sinnvoll, da der generierte Code dadurch ausufern würde. Eine Inline-Assembler Lösung ist zwar auch denkbar, ist jedoch nicht jedermanns Sache und dazu fehlerträchtig.

Besitzt der gewählte Prozessor ein EEprom und ist dieses im Description-File (xxx.dsc) eingetragen, dann bietet der AVRco für EEprom Zugriffe drei Möglichkeiten:

4.21.1 Strukturierte Konstante

Mit dem Compilerschalter {$EEPROM} und der Anweisung **StructConst** werden nachfolgende Konstante als im EEprom liegend definiert. Ein Zugriff auf diese Variablen erfolgt dann automatisch ins EEprom.

Eine indirekte Adressierung mittels Pointer ist nur über *EEpromPtr()* möglich.

Dieses Verfahren ist effizient und vor allen Dingen sehr übersichtlich und lesbar, da es sich problemlos in die Source einfügen lässt. Der Nachteil gegenüber Punkt 3. ist, dass jedes einzelne Byte, auf das zugegriffen werden soll, auch als Variable definiert werden muss.

Es wird ein separates Hex-File geniert, das die definierten Konstanten enthält und mit einem Programmierer in die CPU programmiert werden kann. Siehe Kapitel **StructConst.**

Achtung:
nach der Definition der EEprom StructConst mit {$DATA} oder {$IDATA} wieder auf normalen Speicher zurückschalten.

4.21.2 Variable

Mit dem Compilerschalter {$EEPROM} und der Anweisung **Var** werden nachfolgende Variable als im EEprom liegend definiert. Ein Zugriff auf diese Variablen erfolgt dann automatisch ins EEprom.

Eine indirekte Adressierung mittels Pointer ist nur über *EEpromPtr()* möglich. Typisierte (strukturierte) Konstanten sind möglich.

Dieses Verfahren ist effizient und vor allen Dingen sehr übersichtlich und lesbar, da es sich problemlos in die Source einfügen lässt. Der Nachteil gegenüber Punkt 3. ist, dass jedes einzelne Byte, auf das zugegriffen werden soll, auch als Variable definiert werden muss.

Achtung:
nach der Definition der EEprom Variablen mit {$DATA} wieder auf normalen Speicher zurückschalten.

4.21.3 Speicher Block

Das EEprom wird als eindimensionales Array of byte behandelt. Dazu ist im System schon ein entsprechendes Array mit dem Namen **EEprom** vordefiniert. Mit diesem Array kann jetzt über Indizes zugegriffen werden. Array kopieren ist jedoch nicht möglich.

Da der Index eine Konstante oder Variable ist, lässt sich damit komfortabel arbeiten. Die EEpromgrösse bzw. Arraygrösse kann während der Laufzeit mit SizeOF(EEprom) festgestellt werden.

Der Nachteil dieser Methode ist, dass ein Index nicht so aussagekräftig ist wie ein Namen und auch einer gewissen Verwaltung bedarf.

Grundsätzliche EEprom Einschränkung:
Bestimmte Read-Modify-Write Operationen sind nicht möglich.

4.21.4 EEprom Zugriff

Alle drei Methoden bilden für sich jeweils ausreichende Möglichkeiten, das EEprom zu benutzen. Eine Kombination der Verfahren bietet ein Optimum für die Nutzung. Beim StartUp z.B. kann die Applikation in einer Schleife mittels eines Pointers Daten aus dem Array in vorhandene RAM-Variablen gelesen und u.U. später wieder zurückschreiben. Im "Normalbetrieb" wird mit den EEprom-Variablen gearbeitet. Auf ungewollte Überlappungen achten!

Alle CPUs, die mit einem internen EEprom ausgestattet sind, erwarten, dass während einem Schreib-Zugriff auf das EEprom die Interrupts gesperrt sind. Der Compiler erfüllt diese Forderung.

Weiterhin darf, während die interne Byte-Programmierlogik aktiv ist (Schreibzugriff), kein weiterer Zugriff auf das EEprom erfolgen. Dies wird hier dadurch sichergestellt, dass vor jedem Zugriff auf das EEprom der Status abgefragt wird. Die jeweilige Schreib- bzw. Lese-Routine wartet zu Beginn ggf. auf das Freiwerden des EEproms.

```
{$DATA}
var
    b1   : byte;
    w1   : word;
    i1   : integer;

{$EEPROM}
var
    be            : byte;
    we            : word;
    ie            : integer;
    el[@we]       : byte;       // low byte
    eh[@we+1]     : byte;       // high byte

{$IDATA}
var
    b2   : byte;
    w2   : word;
    i2   : integer;

b2:= be;              {kopiere aus eeprom be nach b2}
W2:= we;              {kopiere aus eeprom we nach w2}
I2:= ie;              {kopiere aus eeprom ie nach i2}
```

4.22 HEAP (*P*)

In der Standard Version von AVRco und auch in vielen anderen Compilern für Embedded Anwendungen beschränkt sich die Speichernutzung/Verwaltung auf globale Variable, Stack und Frame. Das heisst, dass die Speicheraufteilung schon zur Compile-Zeit feststeht. Das ist in aller Regel kein Problem.

Im AVRco wird der Speicher von Adresse 0 (Register) über den I/O-Bereich bis in den iData (evtl. xData) Bereich aufsteigend belegt. Am Ende des benutzten Bereichs liegen die Stacks und Frames. Der nachfolgende Speicher, falls noch etwas frei ist, ist dem Programm normalerweise nicht zugänglich. Für zusätzlichen Speicherbedarf zur Laufzeit, z.B. für Linked Lists etc., gibt es keine Möglichkeit, diesen Speicher bereitzustellen.

In grösseren System, z.B. auf dem PC, kann der noch freie Speicher über die sogenannte „Heap-Verwaltung" dem Programm zur Verfügung gestellt werden (Heap = Haufen). In der Profi-Version des AVRco ist eine solche dynamische Heap-Verwaltung implementiert. Je nachdem, welcher Speicherbereich als Heap verwendet wird (iData, xData), wird der restliche noch freie Speicher in diesem Bereich komplett für den Heap benutzt.

Eine besondere Eigenschaft des Heaps ist, dass nur über Pointer gearbeitet werden kann. Dazu ist freier Speicher mit einer bestimmten Menge anzufordern. Die Adresse des zugewiesenen Blocks wird von der Verwaltung zurückgegeben. Nach Gebrauch durch das Programm sollte der Speicher wieder an die Verwaltung zurück gegeben werden. Ansonsten bleibt der Speicher belegt und kann von der Verwaltung nicht mehr anderweitig zugewiesen werden.

Sehr wichtig ist, dass die Adresse (Pointer) bei der Rückgabe die gleiche ist, wie bei der Anforderung. Ansonsten ist der Heap sehr schnell aufgebraucht und es steht keiner mehr zur Verfügung.

4.22.1 Implementation

Imports
Der Treiber muss wie gewöhnlich beim AVRco importiert werden.

```
Import SysTick, Heap, ..;
```

Defines
definiert den Speicherbereich, der für den Heap benutzt werden soll:

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

4.22.1.1 Funktionen

Speicher Allocation und Freigabe über die Heap Verwaltungs System Funktionen:

```
Function GetMem (var ptr : pointer [; const size : word]) : boolean;
```
Fordert Speicher vom Heap mit der optionalen Grösse „size“ an. Wird size nicht angegeben, wird soviel Speicher allokiert, wie die Struktur benötigt, auf die „ptr“ zeigt. Ist Ptr z.B. ein Pointer to string[10], so werden 11 Bytes allokiert. Die Variable „Ptr“ kann ein Pointer beliebigen Typs sein.

Die Funktion wird true, wenn genügend freier Speicher vorhanden war. In der Variablen ptr wird die Adresse dieses Speichers abgelegt. War die Funktion nicht erfolgreich, wird ein false zurückgegeben und die Variable ptr bekommt den Wert NIL.

Achtung:
Der optionale Parameter „size“ ist nur für spezielle Fälle gedacht und sollte normalerweise nicht verwendet werden.

***Function** FreeMem (**var** ptr : pointer) : boolean;*
Gibt Speicher an den Heap zurück. Es wird soviel Speicher wieder freigegeben, wie die Struktur benötigt, auf die „ptr“ zeigt. Ist Ptr z.B. ein Pointer to string[10], so werden 11 Bytes freigegeben. Die Variable Pointer muss normalerweise die gleiche sein, die bei der Anforderung mit GetMem übergeben wurde.

Konnte der Speicher freigegeben werden, d.h. obige Bedingungen wurden erfüllt, wird ein true zurückgegeben, andererseits ein false. Ein false sollte bei korrekter Programmierung **nie** auftreten. Die Variable ptr wird immer auf NIL gesetzt.

***Function** GetMemAvail : word;*
Gibt die Summe des freien Speichers zurück. Durch wiederholte Allokation und Freigabe unterschiedlicher Blöcke kann der Speicher zerstückelt sein. Der Rückgabe Wert sagt also nichts darüber aus, ob ein bestimmter zusammenhängender Speicher zur Verfügung steht.

***Function** GetLargestBlock: word;*
Gibt die Grösse des grössten, zusammenhängenden freien Speicherblocks zurück.

Bemerkungen:
Pointer im allgemeinen und im Heap im besonderen sind gefährlich (Hallo C). Wenn man beim Heap ein paar Regeln beachtet, eröffnen sich mit dieser Art der Speicherverwaltung einige elegante Möglichkeiten (z.B. "linked lists").

1. Benutzen Sie **immer** typisierte Pointer für GetMem und FreeMem.
2. Verwenden Sie die **gleiche** Pointer Variable für GetMem und das zugehörige FreeMem.
3. Vermeiden Sie die Angabe des Parameters „size“. Mit typisierten Pointern ist das nicht notwendig.
4. Auf keinen Fall sollte der Pointer selbst **manipuliert** werden.
5. Nicht mehr benötigter Speicher möglichst sofort nach Gebrauch freigeben, so dass der Speicher wieder zur Verfugung steht.
6. Beachten Sie unbedingt das bool’sche Ergebnis von GetMem. Bei **false** ist einfach nicht mehr genügend Speicher vorhanden.
7. Tritt bei FreeMem ein false auf, haben Sie einen dicken **Bug** programmiert.
8. Bedenken Sie bei der Speicher Allokation, dass für jeden Block den Sie requirieren, auch zusätzlich 4 Bytes an Verwaltung gebraucht wird. Also ein Speicher von der Grösse eines Byte (mit Pointer to Byte) benötigt insgesamt 5 Bytes an Speicher.
9. Benutzen Sie unbedingt den **Simulator** zum Austesten des Programms.
10. Wenn Sie unbedingt mit dem Parameter **size** arbeiten müssen, benutzen Sie zum bestimmen der Grösse einer Struktur die Funktion **sizeOf()**

Allgemein: seien Sie einfach vorsichtig mit Pointern ☺

4.22.1.2 Beispiel

```
Program AVR_Heap;

Device = 90S8515, VCC=5;

Import SysTick, Heap;

From System Import ;

Define
      ProcClock    = 6000000;      {Hertz}
      SysTick      = 10;           {msec}
      Heap         = iData;
      StackSize    = $0020, iData;
      FrameSize    = $0010, iData;

Implementation

{$IDATA}
{-----------------------------------------------------------------}
{ Type Declarations }
type
  tStr10    = string[10];
  tArr      = array[0..23] of byte;
  tRec      = record
                rb   : byte;
                rw   : word;
              end;

  tpStr     = pointer to tStr10;
  tpArr     = pointer to tArr;
  tpRec     = pointer to tRec;


{-----------------------------------------------------------------}
{ var Declarations }
{$IDATA}
var
   ww         : word;
   ptr, ptr1  : pointer;
   pStr       : tpStr;
   pArr       : tpArr;
   pRec       : tpRec;
   bool       : boolean;
```

```
{-------------------------------------------------------------}
{ Main Program }
{$IDATA}

begin
  ww:= GetMemAvail;
  ww:= GetLargestBlock;
  bool:= GetMem (ptr, 1);
  ww:= GetMemAvail;
  ww:= GetLargestBlock;
  bool:= GetMem (pStr);
  ww:= GetMemAvail;
  ww:= GetLargestBlock;
  bool:= GetMem (pArr);
  ww:= GetMemAvail;
  ww:= GetLargestBlock;
  bool:= GetMem (pRec);
  ww:= GetMemAvail;
  ww:= GetLargestBlock;

  bool:= FreeMem (ptr);
  ww:= GetMemAvail;
  ww:= GetLargestBlock;
  bool:= FreeMem (pStr);
  ww:= GetMemAvail;
  ww:= GetLargestBlock;
  bool:= FreeMem (pArr);
  ww:= GetMemAvail;
  ww:= GetLargestBlock;
  bool:= FreeMem (pRec);
  ww:= GetMemAvail;
  ww:= GetLargestBlock;

  EnableInts;
  loop
    Nop;
  endloop;
end AVR_Heap.
```

Programm Beispiel:

Ein komplettes Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\Heap**

4.23 BOOT VECTORS (BootVectors)

Fast alle Mega-AVR besitzen die Möglichkeit auch im Bootbereich eine Vektor Tabelle zu haben. Damit ist es möglich während eines Updates durch einen Flashloader auch mit Interrupts zu arbeiten, z.B. UART.

Das Problem beim Selbst-Update eines AVRs über den Boot Loader ist, dass dann normalerweise die original Interrupt Vektor Tabelle auch mit erneuert werden muss, weshalb diese zuerst gelöscht werden muss. Damit ist zumindest kurzfristig kein Interrupt mehr möglich. Das verbietet deshalb Interrupt getriebene Treiber im Boot Loader absolut.

Trotzdem kann es manchmal sinnvoll oder gar notwendig sein, im Loader mit Interrupts zu arbeiten. Die meisten Mega AVRs unterstützen dies, indem die Interrupt Vektoren auf den Anfang des Boot Bereich umgemappt werden. Dies geschieht durch die Application indem sie das IVSEL bit im MCUCR bzw. GICR Register manipuliert. Interrupts Vector Lesezugriffe gehen daher entweder in die Vektor Tabelle auf Adresse 0 (IVSEL =0) oder in die zweite Vektor Tabelle die am Anfang des Bootbereichs stehen muss (IVSEL = 1).

Das AVRco System unterstützt dies und bietet noch den weiteren Vorteil, dass Interrupt Service Routinen, die im Boot Bereich liegen, auch von der Applikation selbst benutzt werden können. Dazu wird der jeweilige Interrupt bzw. Vektor in beide Tabellen eingefügt. Die zugehörige Interrupt Service Routine muss deshalb im Boot Bereich implementiert werden.

Es gibt drei Arten von Interrupts:

1. Die Service Routine ist ausserhalb des Boot Bereichs implementiert. Das ist Standard und hat keinerlei Einfluss auf den Boot Bereich und dessen Vektor Tabelle.
2. Die Service Routine liegt im Boot Bereich und hat einen Standard Namen. Dadurch haben beide Vektor Tabellen einen Eintrag bzw. Zeiger auf diese Routine. Sowohl die Applikation als auch der Boot Loader können diesen Interrupt benutzen.
3. Service Routine liegt im Boot Bereich und hat eine Namenerweiterung ***_BOOT***, z.B. *INT0_BOOT*. Diese Routine ist nur für den Loader zuständig und auch nur in dessen Tabelle ist ein entspr. Eintrag zu finden. Die Applikation kann deshalb ihren eigen *INT0* Interrupt definieren.

Achtung
Der BootBlock muss immer mit folgender Anweisung beginnen:

{$PHASE BootBlock nnnnn}

Dann muss als erstes unbedingt eine Prozedur folgen, die als Einsprung Punkt (quasi Main) für den BootBlock dient. Dann folgen Interrupt Prozeduren oder weitere Funktionen oder Prozeduren. Den Abschluss bildet immer:

{$DEPHASE BootBlock}

Einschränkung
Wenn das System selbst einen Interrupt belegt, wie z.B. das SerPort mit RxBuffer, dann kann der RxRDY Interrupt nicht gemeinsam sein. Jede Table muss dann ihren eigen RxRDY Interrupt haben..

4.23.1 Implementation

Imports
Die Boot Interrupt Unterstützung muss importiert werden.

Import *SysTick, TickTimer, ..;*

From *System* ***Import*** *BootVectors, ...;*

4.23.2 Funktionen

Procedure *SetVectTabBoot (boot : boolean);*
Diese Prozedur übernimmt das zeitkritische Umschalten der Vektor Tabellen (IVSEL, MCUCR/GICR).

Procedure *Boot_Init;*
Stellt den Stack und Frame so bereit wie es im Define vorgegeben wurde. Ist externes RAM vorgesehen, erfolgt auch eine entspr. Initialisierung der CPU so dass auch im Boot auf das XRAM zugegriffen werden kann.
Wenn im Bootblock strukturierte Konstanten definiert wurden, werden diese hier initialisiert.
Ist *BootVectors* importiert, wird diese Funktion automatisch ausgeführt und ***darf nicht*** zusätzlich angegeben werden.

Procedure *BootRestart;*
Auch die Applikation kann einen Bootvorgang auslösen indem sie diese Prozedur aufruft. Dieser Einsprung ist aber nur bei importierten *BootVectors* vorhanden !

4.23.3 Konstante

Gerade im Zusammenhang mit komplexen Boot Treibern ist es manchmal unbedingt notwendig, dass „unverlierbare“ Konstante und Strukturierte Konstante im BootBlock sein müssen auf die die Boot Treiber zugreifen können.

Diese Konstante können/müssen deshalb im BootBlock definiert werden. Damit befinden sie sich nicht im allgemeinen Flash Konstanten Pool sondern im BootBlock. Wenn *BootVectors* importiert wurde, werden dann die strukturierten Konstanten automatisch auch an die richtige Stelle im RAM kopiert. Ohne BootVectors Import muss das dann die Funktion *Boot_Init* durchführen.

Alle Konstanten sind auch von der Applikation aus zugreifbar.
Es sind auch absolute Konstanten im BootBlock zulässig.

Achtung
Die Verarbeitung dieser Konstanten im BootBlock bedingt aber grosser Vorsicht. Da ja das Flash gerade gelöscht oder neu beschrieben wird, sind meistens keinerlei System Funktionen mehr zugreifbar und damit sind
Auch viele Operationen mit den Konstanten eigentlich nicht möglich. Eine genaue Analyse der System Aufrufe (CALL oder RCALL) ist hierbei zwingend. Das gleiche gilt auch ohne Einschränkung für das Verarbeiten von Variablen.

```
type
  tTestrec  = record
                b1 : byte;
                b2 : byte;
              end;

{$PHASE BootBlock $0F000}
structconst
  abcde       : array[0..3] of byte = ($12, $34, $56, $78);
  TestrecS    : tTestrec = (b1 : $12; b2 : $34);
  vStr        : string = 'abcdef';

const
  xyz         : array[0..3] of byte = ($12, $34, $56, $78);
  cStr        : string = '12345';
  TestrecC : tTestrec = (b1 : $12; b2 : $34);

Procedure BootTest;
begin
  if abcde[0] <> xyz[0] then ...
  endif;
  if TestrecS.b1 <> TestRecC.b2 then ...
  endif;
  if cStr[1] <> 'a' then ...
  endif;
  if vStr[1] <> cStr[1] then ...
  endif;
```

4.23.4 Programm Beispiel

Im folgenden ist ein Beispiel Programm zu finden und die daraus resultierenden Vektor Tabellen.

```
program AVR_BootVectors;

{$BootRst $0F000}       {reset jumps to here}
{$NOSHADOW}

Device = mega128, VCC=5;

Import SysTick;

From System Import BootVectors;

Define
    ProcClock   = 8000000;      {Hertz}
    SysTick     = 10;           {msec}
    StackSize   = $0020, iData;
    FrameSize   = $0040, iData;
```

Implementation

```
{$IDATA}
{-------------------------------------------------------------}
{ Var Declarations }
var
  iii  : byte;

{-------------------------------------------------------------}
{ functions }

{$PHASE BootBlock $0F000}          // start of boot area
Procedure BootTest;                // this is the first code building
begin                              // part of the boot block
                                   // this is the absolute reset entry
  SetVectTabBoot(true);            // vector table in Boot area
  iii:= 0;
  EnableInts;
  repeat                           // wait for pin interrupts
  until iii > 3;                   // which increments iii
  DisableInts;                     // start over to the MAIN
                                   // enable vector relocation
  SetVectTabBoot(false);           // vector table on address 0
  ASM: JMP    SYSTEM.VectTab;      // absolute address $0000 = RESET vector
end;

Interrupt INT0_BOOT;               // only for the Boot vector table
begin                              // forced by the extension "_boot"
  inc(iii);
end;

Interrupt INT1;                    // for both vector tables
begin                              // because there is no extension
  inc(iii);                        // but it is still located in the boot area
end;
{$DEPHASE BootBlock}               // end of boot area

Interrupt INT0;                    // only for the basic vector table
begin                              // because it is outside of the boot block
  inc(iii);
end;

{-------------------------------------------------------------}
{ Main Program }
begin
  EnableInts;
  loop
    nop;
  endloop;
end AVR_BootVectors.
```

BOOT Bereich

```
0243  F000                                .PHASE     01E000h BOOT
0244  F046                                .ORG       01E08Ch, BOOT
0245  F046            SYSTEM.$BOOT_ENTRY:
0246  F046
0247  F046                                .FUNC      BootTest, 42, 00020h
0248  F046            AVR_BootVectors.BootTest:
0249  F046                                .BLOCK     45
0250  F046                                .LINE      45
0251  F046       EF1F                     LDI        _ACCA, 0FFh
0252  F047       2311                     TST        _ACCA
0253  F048       F009                     BREQ       AVR_BootVectors.L0000
0254  F049       E012                     LDI        _ACCA, 2
0255  F04A            AVR_BootVectors.L0000:
0256  F04A   91000055                     LDS        _ACCB, MCUCR
0257  F04C       7F0D                     CBR        _ACCB, 02h
0258  F04D       6001                     SBR        _ACCB, 01h
0259  F04E   93000055                     STS        MCUCR, _ACCB
0260  F050       7F0E                     CBR        _ACCB, 01h
0261  F051       2B01                     OR         _ACCB, _ACCA
0262  F052   93000055                     STS        MCUCR, _ACCB
0263  F054                                .LINE      47
0264  F054       E010                     LDI        _ACCA, 000h
0265  F055   93100101                     STS        AVR_BOOTVECTORS.III, _ACCA
0266  F057                                .LINE      48
0267  F057       E810                     LDI        _ACCA, 1 SHLB IntFlag
0268  F058       2A21                     OR         Flags, _ACCA
0269  F059       9478                     SEI
0270  F05A            AVR_BootVectors.L0001:
0271  F05A                                .BLOCK     50
0272  F05A                                .ENDBLOCK  50
0273  F05A                                .LINE      50
0274  F05A            AVR_BootVectors.L0002:
0275  F05A   91100101                     LDS        _ACCA, AVR_BootVectors.iii
0276  F05C       3013                     CPI        _ACCA, 003h
0277  F05D       E010                     LDI        _ACCA, 0h
0278  F05E       F011                     BREQ       AVR_BootVectors.L0004
0279  F05F       F008                     BRLO       AVR_BootVectors.L0004
0280  F060       EF1F                     SER        _ACCA
0281  F061            AVR_BootVectors.L0004:
0282  F061       2311                     TST        _ACCA
0283  F065                                .BRANCH    4,AVR_BootVectors.L0005
0284  F062       F411                     BRNE       AVR_BootVectors.L0005
0285  F05A                                .BRANCH    20,AVR_BootVectors.L0001
0286  F063   940CF05A                     JMP        AVR_BootVectors.L0001
0287  F065            AVR_BootVectors.L0005:
0288  F065            AVR_BootVectors.L0003:
0289  F065                                .LINE      52
0290  F065       94F8                     CLI
0291  F066       E71F                     LDI        _ACCA, 0FEH ROLB IntFlag
0292  F067       2221                     AND        Flags, _ACCA
0293  F068                                .LINE      54
0294  F068       E010                     LDI        _ACCA, 000h
0295  F069       2311                     TST        _ACCA
0296  F06A       F009                     BREQ       AVR_BootVectors.L0006
0297  F06B       E012                     LDI        _ACCA, 2
0298  F06C            AVR_BootVectors.L0006:
0299  F06C   91000055                     LDS        _ACCB, MCUCR
0300  F06E       7F0D                     CBR        _ACCB, 02h
0301  F06F       6001                     SBR        _ACCB, 01h
0302  F070   93000055                     STS        MCUCR, _ACCB
0303  F072       7F0E                     CBR        _ACCB, 01h
0304  F073       2B01                     OR         _ACCB, _ACCA
0305  F074   93000055                     STS        MCUCR, _ACCB
0306  F076                                .LINE      55
0307  F076   940C0000                     JMP    SYSTEM.VectTab;     // absolute address $0000
                                                                     // = RESET vector
0308  F078                                .ENDBLOCK 56
0309  F078            AVR_BootVectors.BootTest_X:
0310  F078                                .LINE      56
0311  0000                                .BRANCH    19
0312  F078       9508                     RET
0313  F079                                .ENDFUNC   56
0314  F079
```

```
0315  F079                                .FUNC     INTERRUPT_INT0_BOOT, 58, 00020h
0316  F079          AVR_BootVectors.INTERRUPT_INT0_BOOT:
0317  F079    94E8                        CLT
0318  F07A    F827                        BLD       Flags, IntFlag
0319  F07B                                .BLOCK    60
0320  F07B                                .LINE     60
0321  F07B  91100101                      LDS       _ACCA, AVR_BootVectors.iii
0322  F07D    9513                        INC       _ACCA
0323  F07E  93100101                      STS       AVR_BootVectors.iii, _ACCA
0324  F080                                .ENDBLOCK 61
0325  F080          AVR_BootVectors.INTERRUPT_INT0_BOOT_X:
0326  F080                                .LINE     61
0327  F080    9468                        SET
0328  F081    F827                        BLD       Flags, IntFlag
0329  0000                                .BRANCH   19
0330  F082    9508                        RET
0331  F083                                .ENDFUNC  61
0332  F083
0333  F083                                .FUNC     INTERRUPT_INT1, 63, 00020h
0334  F083          AVR_BootVectors.INTERRUPT_INT1:
0335  F083    94E8                        CLT
0336  F084    F827                        BLD       Flags, IntFlag
0337  F085                                .BLOCK    65
0338  F085                                .LINE     65
0339  F085  91100101                      LDS       _ACCA, AVR_BootVectors.iii
0340  F087    9513                        INC       _ACCA
0341  F088  93100101                      STS       AVR_BootVectors.iii, _ACCA
0342  F08A                                .ENDBLOCK 66
0343  F08A          AVR_BootVectors.INTERRUPT_INT1_X:
0344  F08A                                .LINE     66
0345  F08A    9468                        SET
0346  F08B    F827                        BLD       Flags, IntFlag
0347  0000                                .BRANCH   19
0348  F08C    9508                        RET
0349  F08D                                .ENDFUNC  66
0350  F08D
0351  F08D                                ; ============ String-constant tables ============
0352  F08D
0353  F08D          SYSTEM.BootIntErr:
0354  F08D    9518                        RETI
0355  F08E
0356  F08E          SYSTEM._Boot_Init:
0357  F08E    E011                        LDI       _ACCA, 01h
0358  F08F    BF1B                        OUT       RAMPZ, _ACCA
0359  F090
0360  F090    E110                        LDI       _ACCA, 010FFh SHRB 8
0361  F091    EF0F                        LDI       _ACCB, 010FFh AND 0FFh
0362  F092    BF1E                        OUT       sph, _ACCA
0363  F093    BF0D                        OUT       spl, _ACCB
0364  F094    E1D0                        LDI       _FPTRHI, 010BFh SHRB 8
0365  F095    EBCF                        LDI       _FRAMEPTR, 010BFh AND 0FFh
0366  F096
0367  F096          ; no Peripheral sram-waits
0368  F096    B715                        IN        _ACCA, mcucr
0369  F097    7B1F                        CBR       _ACCA, 040h
0370  F098    BF15                        OUT       mcucr, _ACCA
0371  F046
0372  F046                                .BRANCH   20,AVR_BootVectors.BootTest
0373  F099    CFAC                        RJMP      AVR_BootVectors.BootTest
0374  F09A
0375  F09A          SYSTEM.$INTERRUPT_INT0_BOOT:
0376  F09A    93EF                        PUSH      _ACCCLO
0377  F09B    93FF                        PUSH      _ACCCHI
0378  F09C    930F                        PUSH      _ACCB
0379  F09D    931F                        PUSH      _ACCA
0380  F09E    B71F                        IN        _ACCA, SREG
0381  F09F    931F                        PUSH      _ACCA
0382  F079                                .BRANCH   17,AVR_BootVectors.INTERRUPT_INT0_BOOT
0383  F0A0  940EF079                      CALL      AVR_BootVectors.INTERRUPT_INT0_BOOT
0384  F0A2    911F                        POP       _ACCA
0385  F0A3    BF1F                        OUT       SREG, _ACCA
0386  F0A4    911F                        POP       _ACCA
0387  F0A5    910F                        POP       _ACCB
0388  F0A6    91FF                        POP       _ACCCHI
0389  F0A7    91EF                        POP       _ACCCLO
0390  F0A8    9518                        RETI
0391  F0A9
```

```
0392  F0A9             SYSTEM.$INTERRUPT_INT1:
0393  F0A9       93EF                         PUSH      _ACCCLO
0394  F0AA       93FF                         PUSH      _ACCCHI
0395  F0AB       930F                         PUSH      _ACCB
0396  F0AC       931F                         PUSH      _ACCA
0397  F0AD       B71F                         IN        _ACCA, SREG
0398  F0AE       931F                         PUSH      _ACCA
0399  F083                                    .BRANCH   17,AVR_BootVectors.INTERRUPT_INT1
0400  F0AF   940EF083                         CALL      AVR_BootVectors.INTERRUPT_INT1
0401  F0B1       911F                         POP       _ACCA
0402  F0B2       BF1F                         OUT       SREG, _ACCA
0403  F0B3       911F                         POP       _ACCA
0404  F0B4       910F                         POP       _ACCB
0405  F0B5       91FF                         POP       _ACCCHI
0406  F0B6       91EF                         POP       _ACCCLO
0407  F0B7       9518                         RETI
0408  F000
0409  F000                                    .ORG       01E000h, VECTTABB
0410  F000                                    .VECTTAB_B
0411  F000             SYSTEM.VectTab_B:
0412  F000   940CF08E                         JMP       SYSTEM._Boot_Init
0413  F002   940CF09A                         JMP       SYSTEM.$INTERRUPT_INT0_BOOT
0414  F004   940CF0A9                         JMP       SYSTEM.$INTERRUPT_INT1
0415  F006   940CF08D                         JMP       SYSTEM.BootIntErr
0416  F008   940CF08D                         JMP       SYSTEM.BootIntErr
0417  F00A   940CF08D                         JMP       SYSTEM.BootIntErr
0418  F00C   940CF08D                         JMP       SYSTEM.BootIntErr
0419  F00E   940CF08D                         JMP       SYSTEM.BootIntErr
0420  F010   940CF08D                         JMP       SYSTEM.BootIntErr
0421  F012   940CF08D                         JMP       SYSTEM.BootIntErr
0422  F014   940CF08D                         JMP       SYSTEM.BootIntErr
0423  F016   940CF08D                         JMP       SYSTEM.BootIntErr
0424  F018   940CF08D                         JMP       SYSTEM.BootIntErr
0425  F01A   940CF08D                         JMP       SYSTEM.BootIntErr
0426  F01C   940CF08D                         JMP       SYSTEM.BootIntErr
0427  F01E   940CF08D                         JMP       SYSTEM.BootIntErr
0428  F020   940CF08D                         JMP       SYSTEM.BootIntErr
0429  F022   940CF08D                         JMP       SYSTEM.BootIntErr
0430  F024   940CF08D                         JMP       SYSTEM.BootIntErr
0431  F026   940CF08D                         JMP       SYSTEM.BootIntErr
0432  F028   940CF08D                         JMP       SYSTEM.BootIntErr
0433  F02A   940CF08D                         JMP       SYSTEM.BootIntErr
0434  F02C   940CF08D                         JMP       SYSTEM.BootIntErr
0435  F02E   940CF08D                         JMP       SYSTEM.BootIntErr
0436  F030   940CF08D                         JMP       SYSTEM.BootIntErr
0437  F032   940CF08D                         JMP       SYSTEM.BootIntErr
0438  F034   940CF08D                         JMP       SYSTEM.BootIntErr
0439  F036   940CF08D                         JMP       SYSTEM.BootIntErr
0440  F038   940CF08D                         JMP       SYSTEM.BootIntErr
0441  F03A   940CF08D                         JMP       SYSTEM.BootIntErr
0442  F03C   940CF08D                         JMP       SYSTEM.BootIntErr
0443  F03E   940CF08D                         JMP       SYSTEM.BootIntErr
0444  F040   940CF08D                         JMP       SYSTEM.BootIntErr
0445  F042   940CF08D                         JMP       SYSTEM.BootIntErr
0446  F044   940CF08D                         JMP       SYSTEM.BootIntErr
0447  F046
0448  F046                                    .VECTTABE_B
0449  0046                                    .DEPHASE
```

Applikation Bereich

```
0749  8004                                .ENDCODE
0750  0000                                .ORG       0, VECTTAB
0751  0000                                .VECTTAB
0752  0000              SYSTEM.VectTab:
0753  0000   940C0065                    JMP        SYSTEM.RESET
0754  0002   940C00A9                    JMP        SYSTEM.$INTERRUPT_INT0
0755  0004   940CF0A9                    JMP        SYSTEM.$INTERRUPT_INT1
0756  0006   940C00FB                    JMP        SYSTEM.DefIntErr
0757  0008   940C00FB                    JMP        SYSTEM.DefIntErr
0758  000A   940C00FB                    JMP        SYSTEM.DefIntErr
0759  000C   940C00FB                    JMP        SYSTEM.DefIntErr
0760  000E   940C00FB                    JMP        SYSTEM.DefIntErr
0761  0010   940C00FB                    JMP        SYSTEM.DefIntErr
0762  0012   940C00FB                    JMP        SYSTEM.DefIntErr
0763  0014   940C00FB                    JMP        SYSTEM.DefIntErr
0764  0016   940C00FB                    JMP        SYSTEM.DefIntErr
0765  0018   940C00FB                    JMP        SYSTEM.DefIntErr
0766  001A   940C00FB                    JMP        SYSTEM.DefIntErr
0767  001C   940C00FB                    JMP        SYSTEM.DefIntErr
0768  001E   940C00FB                    JMP        SYSTEM.DefIntErr
0769  0020   940C0093                    JMP        SYSTEM.$INTERRUPT_TIMER0
0770  0022   940C00FB                    JMP        SYSTEM.DefIntErr
0771  0024   940C00FB                    JMP        SYSTEM.DefIntErr
0772  0026   940C00FB                    JMP        SYSTEM.DefIntErr
0773  0028   940C00FB                    JMP        SYSTEM.DefIntErr
0774  002A   940C00FB                    JMP        SYSTEM.DefIntErr
0775  002C   940C00FB                    JMP        SYSTEM.DefIntErr
0776  002E   940C00FB                    JMP        SYSTEM.DefIntErr
0777  0030   940C00FB                    JMP        SYSTEM.DefIntErr
0778  0032   940C00FB                    JMP        SYSTEM.DefIntErr
0779  0034   940C00FB                    JMP        SYSTEM.DefIntErr
0780  0036   940C00FB                    JMP        SYSTEM.DefIntErr
0781  0038   940C00FB                    JMP        SYSTEM.DefIntErr
0782  003A   940C00FB                    JMP        SYSTEM.DefIntErr
0783  003C   940C00FB                    JMP        SYSTEM.DefIntErr
0784  003E   940C00FB                    JMP        SYSTEM.DefIntErr
0785  0040   940C00FB                    JMP        SYSTEM.DefIntErr
0786  0042   940C00FB                    JMP        SYSTEM.DefIntErr
0787  0044   940C00FB                    JMP        SYSTEM.DefIntErr
0788  0046
0789  0046                                .VECTTABE
0790  0046              SYSTEM.ENDPROG:
```

Programm Beispiel

Ein Beispiel befindet sich im Verzeichnis **..\E-Lab\AVRco\Demos\BootVectors**

4.24 BOOT TRAPS (BootTraps)

Werden wichtige Teile des Systems im Boot Bereich untergebracht und sollen diese auch aus dem Applikations-Bereich heraus erreichbar sein, dann stellt sich u.U. das Problem, dass bei einem Neu-Flashen (Boot Loader) der Applikation diese von anderen physischen Adressen ausgeht als das ursprünglich der Fall war.

Ein einfacher Aufruf von Funktionen/Prozeduren aus der Applikation heraus könnte also in falsche oder illegale Code Bereiche des Boot Blocks gehen mit katastrophalen Folgen. Um das zu verhindern hilft eigentlich nur eine Technik, die schon vor Jahren in CP/M und DOS für das BIOS angewendet wurde.

Dabei wird nicht direkt eine Funktion angesprungen, sondern eine Jump Table die am Anfang dieses Bereiches auf unveränderlichen Adressen liegt. Dadurch könnten sich theoretisch sogar beide Bereiche ändern ohne dass dies unabsehbare Folgen hat. Hier ist aber die Tabelle wie auch der Boot Bereich zur Laufzeit nicht änderbar.

Wie funktionierts?
Das System bildet hinter der Boot Vektor Tabelle, oder falls nicht vorhanden, auf dem Boot Einsprung Punkt des Boot Sektors eine Sprung Tabelle mit 16 möglichen JUMPs bzw. Vektoren. Jede Funktion oder Prozedur im Boot Bereich, die das Attribut "***TRAP***" besitzt wird in diese Tabelle eingetragen.

Ein Aufruf einer solchen Funktion oder Prozedur aus der Applikation heraus geht dann nicht wie üblich über ein "CALL Procedure", sondern die Adresse dieser Prozedur wird aus der Tabelle gelesen und für eine ICALL Operation verwendet. Für einen Aufruf einer solchen Trap Funktion direkt aus dem Bootbereich ändert sich nichts, hier bleibt es beim normalen CALL.

4.24.1 Implementation der Boot Traps

Imports
Der Boot Trap Handler muss importiert werden.
```
From System Import Traps {,BootVectors};
```

Defines
Es wird keine spezielle Boot Trap Handler Definition gebraucht. Es wird standardmässig Platz für eine Trap Table mit 16 Positionen reserviert. Soll Platz gespart werden, kann man dies hiermit einschränken:
```
Define  MaxTraps      = 4;   // 2..16

{$PHASE BootBlock $0F000}
...
Procedure TestTrapP(b : byte; w : word); Trap;
begin
  ww:= w;
end;

Procedure TestTrap; Trap;
begin
  TestTrapP (12, 1234);            // procedure is called as usual
end;

{$DEPHASE BootBlock}
// MAIN
begin                              // functions are called through the jump table
  TestTrap;
  bo:= TestTrapF (12, 1234);
  TestTrapP (12, 1234);
  ...
```

4.25 Vererbung (Inheritance)

Die Basis für Objekte ist die sog. Ableitung bzw. Vererbung (Inheritance).
Ein neues Objekt übernimmt alle Eigenschaften eines bestehenden und erweitert das neue Objekt um neue Eigenschaften. Objekte sind eigentlich nichts anderes als spezielle Records. Wenn man jetzt also einen bestehenden Record Typ in einen neuen Record Typ importieren kann, hat man schon einfache Objekte.

Der Import bzw. Vererbung eines bestehenden Records in einen neuen Record erfolgt mit

inherit *RecordName*;

```
type
     tFirstRec  =   record
                      bb    : byte;
                      ww    : word;
                      pp    : pointer;
                    end;

     tObjRec  =     record
                      inherit firstRec;  // import of firstRec
                      ii      : integer;
                      pro     : procedure;
                    end;
```

"ObjRec" erbt die Eigenschaft von "firstRec". Bitte beachten, dass die Inherit Anweisung immer das **erste** Mitglied des neuen Records sein muss.
Der neue Record sieht intern jetzt so aus:

```
     tObjRec  =     record
                      bb    : byte;       // geerbt
                      ww    : word;       // geerbt
                      pp    : pointer;    // geerbt
                      ii      : integer;
                      pro     : procedure;
                    end;
```

Es ist klar, dass kein Namen zweimal auftauchen darf. Eine einfache Anwendung eines solchen Konstrukts ist das Verbergen des geerbten Teils:

```
type
   tpFirstRec  = pointer to tFirstRec;
   tpObjRec    = pointer to tObjRec;

var
    ObjRec     : tObjRec;
    pFirstRec  : tpFirstRec;
    pObjRec    : tpObjRec;
    ...
pFirstRec:=  tpFirstRec (@ObjRec);  // nur FirstRec ist sichtbar
pObjRec:= @ObjRec;                  // alles ist sichtbar
```

5 Multi-Task Programmierung

5.1 Einleitung

Bei einer sog. Embedded Applikation (Single-Chip Anwendung) stellt sich sehr häufig das Problem, dass eigentlich mehrere Aufgaben gleichzeitig erfüllt werden sollten. Z.B. sollten die Zeichen einer seriellen Schnittstelle abgeholt, geprüft und evtl. von Hex zu einem Integer konvertiert werden. Gleichzeitig sind Ports mit Endschaltern zu Überwachen oder eine Leuchtdiode soll blinken. Zusätzlich ist ein Messwert von einem Poti zu erfassen und dieser Wert als Stellgrösse einem Regler zu übergeben. Der Regler wiederum soll in einem festen Zeitraster eine Ausgangsgrösse errechnen.

Alle diese Vorgaben stellen den Programmierer vor das Problem, möglichst alles gleichzeitig zu erledigen. Der Programmierer ist also in der schwierigen Lage, mehrere Vorgänge gleichzeitig überwachen zu müssen, wobei er damit rechnen muss, dass alle Funktionen **gleichzeitig und unabhängig** voneinander laufen müssen.

Mit einfachen Zeitschleifen etc. ist dieses Problem nicht mehr zu lösen oder nur noch mit Tricks, die das ganze Programm schwerfällig und unflexibel machen.

Man braucht also eine Lösung, die es ermöglicht, bestimmte Aufgaben so zu verteilen, dass sie möglichst oft zum Zug kommen, jedoch andere Aufgaben nicht blockieren. Ein solches System nennt man **Multi-Tasking**, wobei Task hier für Aufgabe/Job steht.

Die Begriffe **Tasks** oder **Prozesse** werden in der Literatur oft als identisch bezeichnet. Im AVRco wird hierbei jedoch stark unterschieden. Näheres weiter unten.

Im Zusammenhang von Multi-Tasking fällt auch immer der Begriff **Real-Time**. Streng genommen hat Real-Time (Echtzeit) mit Multi-Tasking nichts zu tun. Unter Echtzeit versteht man die möglichst kurze Antwort des Systems auf externe Ereignisse, im Allgemeinen sind das Interrupts. Die Reaktionszeit ist hierbei jedoch nirgends festgelegt. Je nach Anforderung kann das bedeuten, das System muss innerhalb von Mikrosekunden oder aber zig-Millisekunden reagieren. Die Applikation bestimmt im wesentlichen was Echtzeit ist und was zu langsam ist.

5.2 Funktionsweise

Multi-Tasking bzw. Multi-Processing wird definiert als die Möglichkeit, mehrere Aufgaben (Jobs, Tasks) quasi parallel, also scheinbar gleichzeitig abzuarbeiten. Da es kein Prozessor schafft, mehrere Maschinenbefehle gleichzeitig zu lesen und abzuarbeiten, muss dies sequentiell, d.h. zeitlich nacheinander geschehen.

Diese sequentielle Verarbeitung von Aufgaben (Aufruf eines Jobs, Zuteilung von Rechenzeit, weiterschalten zu dem nächsten Job) wird durch eine sog. Zeitscheibe (**Scheduler**) erledigt. Der Scheduler wird in einem Timer-Interrupt (SysTick) ausgeführt. Er überprüft, ob der momentan laufende Prozess/Task seine Rechenzeit aufgebraucht hat. Wenn ja, werden die Arbeitsregister, Stackpointer, Flags etc. in einem diesem Prozess zugeordneten Speicherbereich gesichert und der nächste Prozess bzw. Task wird aufgerufen.

Dieses Verfahren nennt man auch **Round-Robin**, da die Prozesse im wesentlichen immer im Kreis herum abgearbeitet werden. Andere mögliche Verfahren kommen für kleinere Kontroller nicht in Betracht, da der Verwaltungsaufwand zu gross ist.

Der Scheduler beachtet dabei den Status der einzelnen Tasks. Das sind die *Priorität*, die die Rechenzeit bestimmt, WarteFlags (Pipe, Semaphore, Sleep), die einen Prozess temporär ausser Betrieb setzen. *Suspend* schaltet einen Prozess ganz ab und *Lock* bindet die CPU ganz an diesen Prozess.

5.2.1 Prozesse und Tasks

Prozesse bzw. Tasks sind eigenständige Programme innerhalb einer Applikation, die absolut unabhängig von anderen Programmteilen (z.B. Main) laufen können, d.h. Prozesse kann man nicht aufrufen wie Prozeduren oder Funktionen. Sie werden stattdessen vom einem sog. Scheduler (Zeitscheibe) periodisch aufgerufen. Wurden Prozesse importiert, läuft das Hauptprogramm (Main) ebenfalls als Prozess und hat auch eine Priorität.

Sind mehrere Prozesse/Tasks in einem Programm vorhanden, erfolgt die Verarbeitung der einzelnen Prozesse quasi-parallel, d.h. von aussen betrachtet scheinen alle Prozesse/Tasks gleichzeitig zu arbeiten = **Multi-Tasking**. Damit wird eine scheinbare **Parallelverarbeitung** z.B. von Ereignissen oder Daten erreicht, obwohl sie natürlich immer sequentiell, d.h. nacheinander verarbeitet werden.

Ein Prozess läuft praktisch unendlich lange, nur unterbrochen durch Interrupts und andere Prozesse und Tasks. Das ‘begin’ und ‘end’ grenzt einen Prozess bzw. dessen Statements ein. Da Prozesse nicht wie Funktionen aufgerufen werden können, besitzen sie auch keine Übergabeparameter und kein Ergebnisse.

Beim ersten Aufruf eines Prozesses durch den Scheduler wird mit dem Statement begonnen, das unmittelbar dem ‘begin’ folgt. Im folgenden werden alle Statements bis zu ‘end’ abgearbeitet, evtl. unterbrochen durch einen **Taskwechsel** durch den Scheduler (Umschalten auf einen anderen Prozess/Task). Wird das ‘end’ erreicht, wird automatisch mit dem ersten Statement nach ‘begin’ fortgefahren. Ein Prozess läuft also kontinuierlich im ‘Kreis’ bzw. hat kein Ende. Der Programmierer muss dazu jedoch keine Schleife (Loop) programmieren, denn der Rücksprung an den Anfang (begin) erfolgt automatisch.

Hierin liegt der wesentliche Unterschied zu einem Task. Bei **jedem** Aufruf eines Tasks durch den Scheduler wird mit dem Statement begonnen, das unmittelbar dem ‘begin’ folgt. Im folgenden werden alle Statements bis zu ‘end’ abgearbeitet. Wird das ‘end’ nicht innerhalb eines SystemTicks erreicht, wird der **Task abgebrochen** durch einen **Taskwechsel** durch den Scheduler (Umschalten auf einen anderen Prozess/Task). Der Task erreicht also nie das ‘end’, wenn seine benötigte Rechenzeit von ‘begin’ bis ‘end’ grösser ist als ein SystemTick. Die Rechenzeit eines Tasks darf **nie** grösser sein als ein SystemTick. Ähnliche Bedingungen gelten übrigens auch für Interrupts. Eine Timer-Interrupt-Service Routine z.B. sollte auch nie mehr Zeit brauchen, als der Zeitraum zwischen zwei Interrupts.

Wird das ‘end’ erreicht, wird automatisch die Kontrolle an den Scheduler übergeben, der jetzt den nächsten Prozess oder Task aktiviert. Im Gegensatz zu einem Prozess läuft ein Task bei jedem Aufruf durch den Scheduler von ‘begin’ bis ‘end’ und bricht dann ab.

5.2.2 Priority

Die Arbeitsweise eines Processes/Tasks wird durch eine Vielzahl von zugehörigen Funktionen und Prozeduren gesteuert. Ein wesentlicher Parameter stellt dabei Priority dar.

Mit **Priority** (Wichtigkeit) wird einem **Prozess** einen bestimmten Anteil der zur Verfügung stehenden Rechenzeit zur Verfügung gestellt. Je höher der Wert von Priority ist, desto mehr Rechenzeit steht zur Verfügung. Gleichzeitig legt Priority die Anzahl der SystemTicks fest, die der Prozess ‘am Stück’ zu Verfügung hat. Die anteilige Rechenzeit an der Gesamtzeit in ‘%’ errechnet sich aus:
Priority / Summe aller Prioritäten.

Angenommen, es gibt nur den Prozess ‘DoTheJob’ und dieser hat die Priorität 10 und Main_Priority ist 5 dann gilt: Rechenzeit = 10 / (5 + 10) = 66%. Die exakte Rechenzeit lässt sich jedoch nur festlegen, wenn kein Prozess suspendiert oder Locked wird und keine ProcessWaits etc. vorhanden sind. In der Praxis kann die anteilige Rechenzeit nur überschlagsmässig errechnet werden.

Im Gegensatz zu einem Prozess wird mit **Priority** das Aufruf-Intervall des **Tasks** festgelegt. Je niedriger der Wert von Priority ist, desto häufiger der Aufruf des Tasks. Angenommen der Task ‘RunPid’ hat die Priorität 10 so wird er alle 10 SysTicks aufgerufen. Damit ist sichergestellt, dass der zeitliche Abstand zwischen zwei Aufrufen immer 10 Ticks beträgt.

Bemerkung:

Sind mehrere **Tasks** vorhanden und können davon mehr als einer aktiv sein, ist unbedingt darauf zu achten, dass die einzelnen Prioritäten **einen gemeinsamen Nenner** besitzen.
D.h. die Prioritäten müssen ein vielfaches von z.B. 2 sein. Wird diese Bedingung nicht erfüllt, so kommt es zu unregelmässigen Aufruf-Intervallen, d.h. der Abstand zwischen zwei Aufrufen ist nicht mehr konstant. Weiterhin sollte die kleinste Priorität grösser sein als die Summe (Anzahl) aller Tasks und Prozesse, jedoch zumindest grösser als die Anzahl der Tasks.

Ein Prozess kann mit **Lock** die CPU voll an sich binden, so dass ausser ihm selbst nur noch Interrupts laufen. Tasks sind davon jedoch nicht betroffen, da diese ähnlich wie Interrupts behandelt werden. Mit einem **Unlock** wird dieser Zustand wieder aufgehoben.

Stellt ein Prozess/Task fest, dass er z.Zt. nichts zu tun hat, sollte jetzt nicht unnötig Rechenzeit durch Warteschleifen oder Delays verbraucht werden. Es bestehen mehrere Möglichkeiten, die Kontrolle an andere Prozesse abzugeben:

Mit **Schedule** bricht der Prozess/Task sofort ab, wird aber wieder in die Prozess-Warteschlange eingereiht.

Mit **Sleep** kann sich ein Prozess/Task für eine bestimmte Anzahl von Systemticks abschalten.

Mit **Suspend** schaltet sich ein Prozess/Task ab. Er kann sich selbst nie wieder aktiv schalten. Das muss von ausserhalb durch eine anderen Prozess/Task oder das Hauptprogramm mit **Resume** erfolgen.

Da die Kommunikation zwischen den Tasks/Prozessen u.a. über Pipes oder Semaphoren erfolgen kann, besteht die Möglichkeit dass der Task sich abschaltet, indem er **WaitSema** oder **WaitPipe** aufruft. Der Prozess/Task wird erst wieder aktiv, wenn in der spezifizierten Semaphore oder Pipe ein Datum vorhanden ist. Als Pipe kann hier auch **RxBuffer** (RxBuffer1, -2, -3) spezifiziert sein.

Der Prozess/Task wird unmittelbar im Anschluss an einer der o.a. Anweisungen abgebrochen.

5.2.2.1 Default Priorities

ohne explizite Angabe von Prioritäten erhalten

Main:	Priority 5
Prozess:	Priority 3
Task:	Priority 5

5.3 Optimales Multi-Tasking

Beim Entwurf einer MultiTask Anwendung ist eine sorgfältige Planung sehr wichtig.

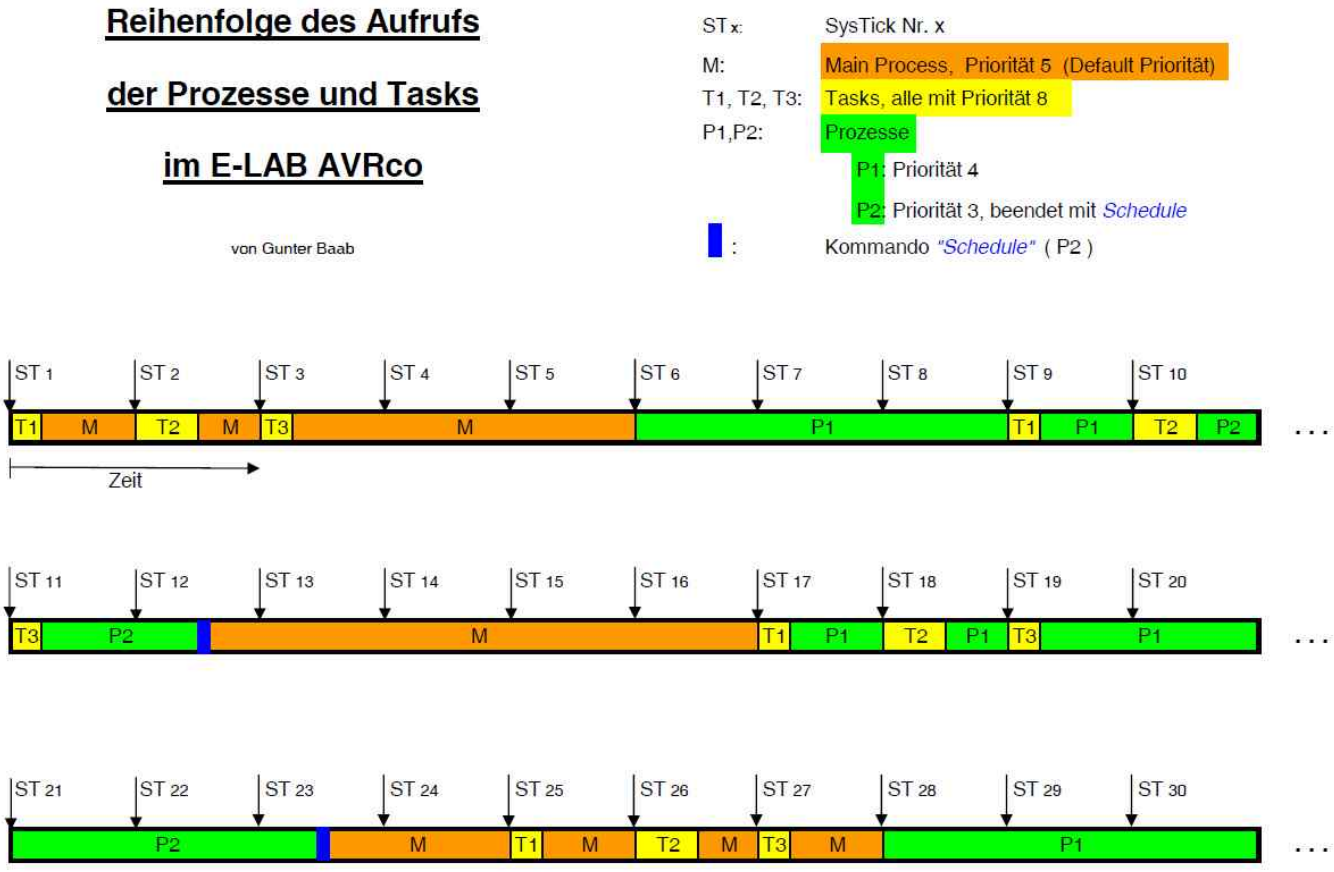
1. Mit der richtigen Strategie und einer guten Aufteilung der Aufgaben in einzelne Tasks und Prozesse wird eine sichere und schnelle Programmausführung sichergestellt.

2. Trotz des durch den Scheduler verursachten Overheads (ca. 330 Zyklen) läuft ein Multi-Task Programm wesentlich schneller als ein normal programmiertes, wenn von Schedule, Sleep, WaitSema, WaitPipe und Suspend überlegt Gebrauch gemacht wird.

3. Die richtige Verteilung der Prioritäten entscheidet über die Reaktionszeit der einzelnen Prozesse und damit der Echtzeitfähigkeit. Wenn die Prioritäten dynamisch, d.h. während des Programmlaufs angepasst werden, ist eine zusätzliche Verbesserung des Systems zu erreichen.

5.4 MultiTasking Diagramm

Das untenstehende Diagram veranschaulicht die Arbeitsweise des MultiTasking System und wie der interne Scheduler bei jedem SysTick abprüft ob ein neuer Task an der Reihe ist, oder von einem aktuell laufenden Prozess zu einem anderen umgeschaltet werden muss.



6 Optimierung

6.1 Bibliothek

Der Compiler optimiert die Bibliotheks Aufrufe komplett, das heisst, nur diejenigen Bibliotheks Funktionen, die auch benutzt werden, werden importiert und belegen Programmspeicher und evtl. auch Arbeitsspeicher (Variable). Damit ist sichergestellt dass in dieser Hinsicht keine Resourcen verschwendet werden.

6.1.1 Variable

Der Pascal Compiler besitzt zur Zeit noch keine Variablen-Optimierung. Das bedeutet, dass eine Variable, die deklariert worden ist, auch physisch Speicher belegt. Wird diese Variable im Programm nicht benutzt, so wird der jeweilige Speicher vergeudet. Es ist daran gedacht, zu einem späteren Zeitpunkt eine Variablen Optimierung einzuführen, die unbenötigte Variablen eliminiert.

Der Programmierer muss bis dahin, um Resourcen zu schonen, selbst dafür sorgen, dass nicht benutzte Variablen gelöscht werden. Eine Hilfe dafür bietet der Compilerschalter *{W+}*.

6.1.2 Konstante

Eine Konstanten Optimierung ist vorhanden. Eine Konstanten Deklaration wie z.B. "***Const*** *x = 123;*" belegt keine Resourcen in der CPU, ob diese Konstante nun benutzt wird oder nicht.

Ebenso werden Konstante Ausdrücke in aller Regel erkannt und vorberechnet.
Beispiel "*a:= (5 * 10) + (24* ***div*** *2);*".
Das Ergebnis dieser Operation ist "62". Der Compiler erkennt dies und setzt gleich den Wert 62 ein indem er z.B. *LDI _ACCA, 62* erzeugt.

6.1.3 Laufzeit

Das generierte Programm ist bei fast allen Compilern eine sog. Stackmaschine. Sämtliche Übergabe Parameter und Zwischenergebnisse bei Ausdrücken werden auf dem Stack bzw. Frame abgelegt. Damit sind aber oftmals unnötige PUSH und POP Operationen verbunden. Ebenso werden oft Variablen ins Arbeitsregister geladen, obwohl diese Variable sich schon in diesem Register befindet. Das rührt von dem in jedem Compiler befindlichen Formalismus her.

Eine Laufzeit Optimierung erkennt diese unnötigen Operationen und eliminiert sie.
Spätere Versionen vom E-LAB Pascal Compiler werden diese Optimierung ebenfalls besitzen.

6.2 Hochoptimierend?

Viele Compiler schmücken sich (manchmal sogar zu recht) mit dem Attribut "hochoptimierend". Eine sehr starke Optimierung ist aber oft ein zweischneidiges Schwert. Werden z.B. alle aktuellen Variablen in Registern gehalten und da bearbeitet, hat z.B. eine Interrupt-Routine oder Multi-Tasking keine Möglichkeit auf diese aktuellen Werte zuzugreifen! Weiterhin kann diese Routine nicht rekursiv (sich selbst aufrufend) sein oder sie muss zumindest die Register zu retten. Hochoptimierter Code ist in der Regel auch nicht reentrant (Abbruch einer Routine durch ein Interrupt oder Multi-Tasking und Aufruf derselben innerhalb des Interrupts bzw. Tasks).

Bei bestimmten Compilern ist es deshalb zwingend notwendig, dass entsprechende Prozeduren oder Funktionen mittels Compilerschalter entsprechend deklariert werden, um die Optimierung abzuschalten.

Anfänger aber auch Fortgeschrittene vergessen das häufig und ein sporadischer, nicht nachvollziehbarer Systemabsturz ist vorprogrammiert. Ein solches System ist daher nicht pflegeleicht und vor allem für Anfänger nur geeignet, wenn wesentliche Optimierungen abgeschaltet werden.

Der AVRco arbeitet grundsätzlich mit Stackframes, was zwar langsamer, aber extrem sicher ist. Alle System- und User Funktionen/Prozeduren sind grundsätzlich reentrant und auch rekursionsfest. Die Rekursionstiefe wird nur durch den Arbeitsspeicher und die Stackgrösse limitiert.

Beim AVRco ist die Floating Point Bibliothek reentrant, d.h. sie kann auch innerhalb Interrupts und Rekursionen benutzt werden, was nicht bei allen Compilern der Fall ist. (Hierbei Registersicherung beachten).

Obwohl Geschwindigkeit und kompakter Code ein wichtiges Ziel eines Compilers ist, wurde beim AVRco grössten Wert auf Betriebssicherheit gelegt und die Optimierung als zweitrangig betrachtet, was allerdings nicht heissen muss, dass eine gewisse Optimierung kein Thema wäre. Wie weiter oben schon erwähnt, wird es in Zukunft auch optimierende Versionen geben.

Der kompakteste und schnellste Code
wird in jedem System durch die Systembibliothek erzielt. Je mehr Funktionen im System vorhanden sind, desto weniger muss der Programmierer selbst in Hochsprache formulieren. Systemfunktionen sind komplett in Assembler geschrieben und damit hocheffizient, d.h. schnell und kurz. Der in manchen Compilern enthaltene “Librarian” generiert aus Hochsprache eine linkfähige Bibliothek. Der generierte Code entspricht jedoch exakt demjenigen, den der Compiler aus diesen Statements erzeugt, wenn diese direkt ins Programm eingebunden sind. In dieser Hinsicht also absolut kein Gewinn.

Einen I^2C-Bus muss ein Programmierer bei den meisten Systemen in Hochsprache formulieren. Dieser Code wird um ein vielfaches grösser und langsamer als die entsprechende Systemfunktion. Durch den Import von mehreren Systemfunktionen, von denen im AVRco viele vorhanden sind, wird die mässige Optimierung vom AVRco gegenüber manchen anderen Systemen mehr als wettgemacht.

Der beste Optimierer ist im übrigen der Programmierer selbst. Durch geschickten Einsatz von Multitasking und sparsamer Verwendung von lokalen Variablen kann ein optimaler Code erzeugt werden.

Der generierte Code vom AVRco (auch im Zusammenhang mit Multi-Tasking) braucht bei etwas komplexeren Applikation keinen Vergleich zu scheuen.

Ein “wasserdichtes” Programm ist das Ziel vom AVRco. Ein höchstoptimiertes Programm ist nur mit sehr sorgfältigem Einsatz von Compilerschaltern zu erreichen und damit auch sehr fehleranfällig!

Im übrigen bringt ein etwas grösserer Prozessor mit mehr Rechenleistung und grösserem Ram/Rom mehr Zuwachs als eine Optimierung und kostet oft auch nur wenig mehr als eine schwächere Version, ganz abgesehen vom Zuwachs an Leistungsfähiger Peripherie (Ports, Timer, Schnittstellen etc.).

6.3 Der "Merlin Optimiser"

Der Merlin Optimiser ist ein Plug-In für die Standard und Profi Version des AVRco Systems.
Damit kann die Code Grösse eines Programms um 5..30% reduziert werden.
Aufgerufen wird der Optimiser direkt aus der IDE mittels HotKey oder SpeedButton.
Somit erfordert der komplette Ablauf "Compilieren" – "Optimieren" – "Assemblieren" ebenfalls nur einen einzigen Mausklick.

{$OPTIMISE}
Der Schalter bestimmt die Compilierung unter Einbezug des Merlin Optimisers. Der Schalter **muss** in der ersten Zeile des Hauptprogramms stehen!

Seit der Version 4 (Standard und Profi) ist der Merlin Optimiser im AVRco enthalten.

6.3.1 Beste Ergebnisse mit dem Merlin Optimiser erzielen

6.3.1.1 Einleitung

Das Ziel eines jeden Optimierers ist es in erster Linie den Code zu verbessern damit dieser schneller und kürzer wird. Manchmal widersprechen sich diese beiden Anforderungen (Grösse und Geschwindigkeit) und viele Optimierer erlauben es, diese Anforderungen zu gewichten. Als Optimierer für Microcontroller optimiert der Merlin Optimiser ausschliesslich die Grösse, was aber üblicher Weise dazu führt, dass sich auch die Geschwindigkeit verbessert.
Deshalb gibt es auch nur ganz wenige Schalter um den Optimierer zu steuern. Diese sind am Ende aufgelistet. Genau wie Compilerschalter haben diese die Form eines Kommentars beginnend mit einem $ Zeichen.
Hier wird beschrieben wie einige der Optimierungen funktionieren und wie man seinen Code so formuliert, dass man möglichst gute Ergebnisse mit dem Optimierer erzielen kann

6.3.1.2 Wie viel Optimierung kann man erwarten?

Das hängt davon ab welche Art von Programm man schreibt. Wenn man viele eingebaute Treiber benutzt (von denen die meisten schon hoch optimiert sind) kann man nicht viel erwarten – vielleicht 5%. Bei großen Programmen mit viel eigenem Code kann es bis zu 30% sein. Dann kann man es sich ggf. sogar sparen auf einen größeren Prozessor zu wechseln.
Um die erreichte Optimierung zu überprüfen schauen Sie sich das Ende des generierten Assembler File (.asm) an. Dort wird das detailliert vermerkt.

6.3.1.3 Was wird optimiert ?

Optimiert wird nur der Code. Es wird niemals versucht die Daten in irgendeiner Form zu optimieren (obwohl die Loop Optimierung den Frame etwas entlasten kann).

6.3.1.4 Beta Code

Wie der Compiler wird auch der Optimierer ständig verbessert. Aufgrund der vielen Möglichkeiten zu Programmieren ist es unmöglich die neuen Erweiterungen 100% zu testen. Wenn Sie irgendwelche Fehler finden, kontaktieren Sie bitte Merlin im E-LAB Forum. Er wird sich bemühen das Problem so schnell wie möglich zu beheben.

Neue Optimierungen sind immer vom Schalter {$OPTI_BETA_OFF} abhängig.
Im Falle eines Bugs können Sie diesen Schalter benutzen und weiterarbeiten bis das Problem beseitigt ist. Wenn Sie diesen Schalter benutzen müssen teilen Sie es aber bitte unbedingt Merlin mit und stellen Sie ihm Programmbeispiele zur Verfügung (schicken Sie im E-LAB Forum die .asm und .dsm Datei)!

Ansonsten wird der Fehler nie beseitigt. Manche Bugs sind derartig obskur, dass sie nur in einem bestimmten Programm auftauchen.

Verlassen Sie sich nicht auf diesen Schalter! Eventuell wird die betroffene Optimierung irgendwann dadurch nicht mehr abgeschaltet und Sie wundern sich, warum Ihr Programm dann nicht mehr funktioniert.

6.3.1.5 Volatility

Beachten Sie den folgenden Code:

```
x:= 3;
x:= 4;
```

Ist das erste Statement redundant/unnötig? Das kommt darauf an. Wenn z.B. x ist ein Port dann ist x nicht redundant, es ist volatile. Andererseits wenn x Bestandteil eines normalen Memory ist, dann ist das erste Statement redundant (non-Volatile). Der Optimiser hat unterschiedliche defaults für die diversen Memory Typen.

$DATA, $IDATA und $EEPROM sind non-volatile (wenige Ausnahmen). $PDATA, $XDATA und $UDATA sind volatile. Mit volatile werden beide obigen Statements ausgeführt, mit non-volatile nur das zweite. Volatile Schalter sind normalerweise nur in XData notwendig.

Diese defaults können mit den {$OPTI VOLATILE_xxx} switches überschrieben werden. Diese begrenzen Speicherblöcke und müssen mit {$OPTI VOLATILE_DFT} abgeschlossen werden. Zum Beispiel:

```
{$XDATA}
var
{$OPTI VOLATILE_OFF}
lastChangedData : tDate;
{$OPTI VOLATILE_ON}
I2ClinesA : byte; // clock, data etc
I2ClinesB : byte; // clock, data etc
{$OPTI VOLATILE_DFT}
...
```

{$OPTI VOLATILE_ON} die folgenden Daten werden als volatile behandelt, keine Optimierung
{$OPTI VOLATILE_OFF} die folgenden Daten werden als non-volatile behandelt, Optimierung
{$OPTI VOLATILE_DFT} die folgenden Daten werden default behandelt, wie oben beschrieben

Achtung:
Der XDATA Bereich ist default volatile!

6.3.1.6 Schleifen Optimierung

Die AVRco Implementierung von *for* Schleifen ist sehr mächtig im Vergleich zu z.B. Delphi. Delphi erfordert, dass Schleifenvariablen lokal sind und führt alle Berechnungen am Anfang der Schleife durch.

Deshalb produziert in Delphi etwas wie

```
var
  i : byte;

begin
  for i := 0 to 255 do
```

eine Endlosschleife weil i niemals 255 überschreiten kann (i läuft über). Der AVRco fängt derartiges ab und arbeitet sauber.
Der AVRco lässt eine beliebige Variable als Schleifenzähler zu. Lokal, global oder sogar auch ein Record Element. Der Nachteil ist, dass die *for* Schleifen recht langsam sind und viel Code benötigen.

Schleifen Konstrukte wie die obige können viel einfacher und schneller sein und der Optimiser erledigt dies wo immer er kann. Dies ist jedoch nur für lokale Schleifenzähler möglich. Die Auswirkung dieser Optimierung können Sie im Beispiel ..\E-LAB\AVRco\Optimiser\Demos\ForLoopTest.pas sehen.

benutzen Sie lokale Schleifen Variablen für eine bestmögliche Optimierung

Schauen Sie sich nun folgendes an:

```
var
  i, iMax : byte;
begin
  iMax := 255;
  for i := 0 to iMax do
  ...
```

und dann das Äquivalent

```
const
  iMax : byte = 255;
var
  i : byte;
begin
  for i := 0 to iMax do
  ...
```

Eventuell gibt es Gründe, dass Sie es derart formulieren wollen. Vielleicht um iMax auch sonst zu benutzen und die Wartung zu vereinfachen. Der Optimiser produziert viel bessere Ergebnisse in Schleifen bei denen die Grenzen Konstante sind als bei Schleifen bei denen die Grenzen Variablen sind. Deshalb ist die zweite Form zu bevorzugen.

benutzen Sie möglichst Konstante als Schleifegrenzen für eine bestmögliche Optimierung

6.3.1.7 Gemeinsamer Ausgang von Teilmengen eines Ausdrucks

Das hört sich kompliziert an, ist aber in Wahrheit recht einfach. Am Besten kann das anhand eines Beispiels erläutert werden. Schauen Sie sich folgendes Code Fragment an

```
Procedure Setup (var a : byte; var b: byte; var c: byte; var Mode : byte);
begin
  Case Mode of
    0:    a := 1;   b := 5;   c := 1;   |
    1:    b := 1;   c := 5;   a := 1;   |
    2:    c := 1;   a := 5;   b := 1;   |
    3:    a := 1;   b := 1;   c := 1;   |
  End_case;
```

Der Optimierer verarbeitet das korrekt kann aber nicht besonders viel ausrichten. Wenn wir das etwas anders formulieren erzielt er eine dramatische Verbesserung,

```
Procedure Setup (var a : byte; var b: byte; var c: byte; var Mode : byte);
begin
  Case Mode of
    0:    a := 1;   b:= 5;    c := 1;   |
    1 :   a := 1;   b:= 1;    c := 5;   |
    2:    a := 5;   b :=1;    c := 1;   |
    3:    a := 1;   b :=1;    c := 1;   |
  End_case;
```

Jetzt erkennt der Optimierer, dass an 3 Stellen c eine 1 zugewiesen und dann an das Ende des Statements gesprungen wird. An 2 von diesen Stellen wird weiterhin a eine 1 zugewiesen.
Der Optimierer kann dies zusammenfassen (wenn Sie interessiert wie, schauen Sie sich das Beispiel CSE_Test.asm an nachdem Sie ..\E-LAB\AVRco\Optimiser\Demos\CSE_Test.pas compiliert und optimiert haben).

Jedoch ist das ein etwas zweischneidiges Schwert. Wenn Sie mit dem Simulator oder JTAG schrittweise den Code abarbeiten – welcher Zeile entspricht dann das jeweilige ‘c:=1’? Der Simulator kann das nicht wissen, da nach der Optimierung mehrere Code Zeilen zu einer einzigen zusammengefasst wurden. Deswegen müssen Sie zum Debuggen diese Optimierung unterbinden . Das geschieht mit dem Schalter {$OPTI NO_CSE_OPT}.
Damit wird diese Optimierung im gesamten Programm verhindert. Der Schalter wirkt global und kann an beliebiger Stelle im Programm stehen. Aber dies ist eine wichtige Optimierung, sodass Sie nicht vergessen sollten, diese nach dem Debuggen wieder zu aktivieren.

wenn Sie an mehreren Stellen, die einen gleichen Ausgang haben, den selben Wert der selben Variablen zuweisen, schreiben Sie die Zuweisungen in der gleichen Reihenfolge

6.3.1.8 Selbst geschriebener Assembler Code

Manchmal ist es notwendig eigenen Assembler Code einzubinden. Für die meisten Optimierungen ist das keine Thema – jedoch mit einer speziellen Ausnahme.
Der Compiler gibt dem Optimierer am Anfang der meisten Funktionen eine Information welche Register gerettet werden müssen (wen es interessiert: das ist die *.RETURNS* Directive).
Bei selbst geschriebenen ASM Code kann das der Compiler nicht erkennen und ggf. Probleme verursachen. Nachfolgend ein Beispiel das dies demonstriert:

```
UserDevice KeyBoard8IOS : byte;
begin
  ASM;
    PUSH   _ACCCLO
    LDI    _ACCCLO, x AND 0FFh
    LDI    _ACCCHI, x SHRB 8
    LD     _ACCA, Z
    COM    _ACCA
    POP    _ACCCLO
  ENDASM;
end;
```

Der Compiler teilt dem Optimierer mit, dass _ACCA vom aufrufenden Programm benötigt wird. Aber nicht, dass _ACCCLO ebenfalls gebraucht wird. Um dies zu lösen wurden zwei Schalter eingeführt um dem Optimierer mitzuteilen, dass alle Register benötigt werden:

```
{$OPTI   NO_CHECK_RETURN_REGS}
UserDevice KeyBoard8IOS : byte;
begin
  ASM;
    PUSH   _ACCCLO
    LDI    _ACCCLO, x AND 0FFh
    LDI    _ACCCHI, x SHRB 8
    LD     _ACCA, Z
    COM    _ACCA
    POP    _ACCCLO
  ENDASM;
end;
{$OPTI CHECK_RETURN_REGS}
```

Ein Beispiel finden Sie in ..\E-LAB\AVRco\Optimiser\Demos\No_Return_Check.pas

6.3.1.9 Setters und Getters

Oft wollen Programmierer (insbesondere die OO Programmierer), dass eine Variable nur von außerhalb einer Unit lesbar ist. Der übliche Weg ist eine sog. **Getter** Funktion wie folgt zu schreiben

```
Function GetVar_a : byte
begin
  return (Var_a);
end;
```

und die Funktion sichtbar zu machen (indem man diese in den Interface Teil nimmt).
Die Variable selber ist lokal (indem man sie in den Implementation Teil nimmt).

Diese Form (nur ein einfache Return Zeile) ist die “einfache” Form eines Getter.
Der Vorteil ist die Wartung. Damit wird ein unkontrollierter Zugriff verhindert, was später Probleme bereiten kann. Damit ist es auch leicht weitere Aktionen (Seiteneffekte) später hinzuzufügen, z.B.

```
Function GetVar_a : byte
begin
  inc (Var_a_AccessCount);          // weitere Aktion – Zugriffe zählen
  return (Var_a);
end;
```

Diese Form (z.B. mit Seiteneffekten) ist eine “komplexe” Form.

Das Gegenteil ist ein **Setter**:

```
Procedure SetVar_a (Value : byte);
begin
  Var_a := Value;
end;
```

Die Vorteile sind ähnlicher Natur. Bei späterer Fehlersuche oder weiteren Aktionen muß man nicht viele Units durchforsten um festzustellen wo überall diese Variable geändert wird.
Erneut ist die oben gezeigte Form (eine einfache Zuweisung) die “einfache” Form.
Ein Beispiel wie das folgende (mit Seiteneffekten) ist die “komplexe” Form.

```
Procedure SetVar_a (Value : byte);
begin
  Var_a := Value;
  if Var_a > VarAMax then          // weitere Aktion – was war der max Wert von Variable a?
    VarAMax := Value;
  end_if;
end;
```

Das Problem dieser Idee ist natürlich der Mehraufwand von Setter und Getter in ihrer einfachen Form. In der komplexen Form müssen diese ja sowieso Funktionen oder Prozeduren sein.
Der Optimierer sucht die einfachen Formen von Setter, Getter und ähnlichen (einfachen) Funktionen und ersetzt diese durch einfachen Zuweisungen. Dies wird auch als Erzeugung von ‘Inline’ Code bezeichnet, was heisst, dass die Source wie eine Funktion oder Prozedur aussieht, der erzeugte Assembler Code jedoch nicht.
Falls die einfache Form später zu einer komplexen Form wird, macht diese der Optimierer nicht mehr inline. Damit wird er jeder Situation optimal gerecht
(Bem.: der Optimierer macht jede Funktion, die nur eine Assembler Zeile lang ist ‘inline’. Das betrifft nicht nur Setter und Getter)
Wie schon bei anderen erwähnten Optimierungen kann das jedoch die Code -Verfolgung mit JTag oder dem Simulator erschweren wenn man nicht genau weiß was da passiert. Deshalb kann auch diese Optimierung mit
{$OPTI ALLOW_INLINE} und {$OPTI NO_ALLOW_INLINE} gesteuert werden, z.B.

```
{$OPTI   NO_ALLOW_INLINE}
```

```
x1 := x2Getter;
{$OPTI   ALLOW_INLINE}
```

Schreibt man {$OPTI NO_ALLOW_INLINE} an den Anfang der Main Unit wird die Optimierung im gesamten Programm ausgeschalter.
Beachten Sie, dass im oben stehenden Beispiel die Optimierung nur für diesen Aufruf unterlassen wird. Im Rest des Programms wird diese weiterhin ausgeführt, auch für die gleiche Funktion.
Einen Setter oder Getter mit diesen Schaltern zu kapseln hat keine Wirkung.

Also verhindert folgendes die Optimierung dieser Funktion nicht.

// FALSCH – das funktioniert NICHT:

```
{$OPTI   NO_ALLOW_INLINE}
Function MyGetter : byte;
begin
  Return (My);
end;
{$OPTI ALLOW_INLINE}
```

Beispiele dazu finden Sie in ..\E-LAB\AVRco\Optimiser\Demos\SetGet….pas

6.3.2 Zusammenfassung der Optimiser Schalter

{$OPTI_BETA_OFF}
Schaltet die in letzter Zeit eingeführten Optimierungen aus. Sollten Sie nur benutzen wenn Sie (wegen eines Bugs) dazu gezwungen sind. Dann teilen Sie dies bitte Merlin mittels privater Mail im E-LAB Forum mit. Im Verzeichnis mit Ihrer .pas Datei finden Sie eine gleichnamige .asm und .dsm Datei. Hängen Sie diese Dateien bitte an Ihre Nachricht an! Wenn Sie ein kurzes Beispiel schreiben könne, das das Problem zeigt, ist dies besser als eine ganze Applikation zu schicken.

{$OPTI NO_CSE_OPT}
Schaltet die Optimierung für gemeinsame Ausgangspunkte ab. Dies ist ein globaler Schalter der an beliebiger Stelle im Code stehen kann. Bitte höchstens für Debug Zwecke benutzen!

{$OPTI ALLOW_INLINE}** and **{$OPTI NO_ALLOW_INLINE}
Schaltet die Ersetzung durch Inline Code für einfache Prozeduren (wie einfache “Setter” und “Getter”) an oder aus. Sobald der Ablauf klar ist, sollte dieser Schalter nicht mal mehr für Debug Zwecke notwendig sein.

{$OPTI NO_CHECK_RETURN_REGS}** und **{$OPTI CHECK_RETURN_REGS}
Bestimmt bei eigenem Assembler Code ob die .RETURNS Anweisung bestimmt, welche Register für die aufrufenden Funktion gerettet werden müssen.
Müssen weitere Register für die aufrufende Funktion gerettet werden, muss der erste Schalter vor die Funktion/Prozedur, der zweite Schalter nach der jeweiligen ‘end’ Anweisung platziert werden.

{$OPTI_QUICK}
Der Optimierer arbeitet den Assembler Code mehrere Male ab. Jedoch wird der Großteil der Optimierungen im ersten Durchlauf erledigt. Die {$OPTI_QUICK} Anweisung teilt dem Optimierer mit, dass er nur einen Durchlauf machen soll. Diese Anweisung existiert hauptsächlich wegen eine Rückwärtskompatibilität. Version 2 des Optimierers war wesentlich langsamer als Version 3 und in Version 2 konnte dies ein wichtiger Schalter sein. Mit Version 3 wird dadurch jedoch nur sehr wenig Zeit eingespart.

{$OPTI SMARTLINK_ONLY}
Der Merlin Optimiser ersetzt nun komplett den AVRco SmartLinker. Werden nur die SmartLinker Funktionen benötigt, kann dies dem Merlin Optimiser mit diesem Schalter mitgeteilt werden.
Dieser entfernt dann nur unbenutzte Funktionen/Prozeduren (dead code).
Alle anderen Optimiser Funktionen sind dann abgeschaltet. Dieser Schalter arbeitet global, also auch für Units und Includes.

7 Compiler Schalter

Compilerschalter dienen dazu das Verhalten des Compilers zu steuern. Diese Schalter sind Bestandteil des Quelltextes/Source.

Ein Schalter wird durch eine geschweifte Klammer **{** und einem darauf ohne Leerzeichen folgenden **$** eingeleitet. Unmittelbar auf den $ muss der Schaltername folgen Die Angabe evtl. weiterer Parameter wie z.B. Filenamen erfolgen nach den üblichen Konventionen.
Die Schalter sollten in einer Zeile am Zeilen Anfang stehen. In der gleichen Zeile dürfen keine Statements stehen.

Syntax: **{$SWITCH [arg] }**

Jeder Import und jedes Device Define wird auch in die Compiler Schalter Liste eingefügt. Das bedeutet dass:

Import *LCDport, ...*

behandelt wird als wäre es: {$DEFINE LCDPORT} und

Define *ProcClock = 8000000;*

wird behandelt wie: {$DEFINE PROCCLOCK}

7.1 Speicher Verwaltung

{$DATA} {$IDATA} {$IDATA1} {$PDATA} {$XDATA} {$XDATA1}..{$XDATA4} {$EEPROM} {$UDATA}
Diese Schalter stellen die Speicher-Seite des Prozessors ein. Die meisten kleinen Prozessoren können nur einen kleinen Speicherbereich mittels direkter Adressierung erreichen. Evtl. vorhandener zusätzlicher Speicher muss besonders behandelt werden. Das gilt insbesondere für externen Speicher, falls möglich und vorhanden. Der Schalter weist den darauf folgenden Variablen den jeweiligen Bereich zu.

{$DATA}
$DATA weist allen nachfolgenden Variablen Deklarationen (dazu zählt auch **StructConst**) den im Prozessor Steuerfile (xxx.dsc) unter DATA ausgewiesenen Bereich zu. Beim AVR ist dieser Bereich von $04 bis $1F. Die definierten Variablen werden jetzt fortschreitend ab $04 platziert. Folgt ein anderer Schalter dieses Typs, wird mit den folgenden Variablen analog dazu verfahren. Variablen im Bereich $DATA sind immer mit sehr kurzen und schnellen Maschinenbefehlen zu erreichen.
XMega
Hier sollte die Verwendung von $DATA vermieden werden, da die Register ab addr 0 beginnen als auch die IO Bereich mit addr 0 beginnt. Das kann zu Problemen führen.

{$PDATA}
$PDATA weist allen nachfolgenden Variablen Deklarationen den im Prozessor Steuerfile (xxx.dsc) unter PDATA ausgewiesenen Bereich zu. Beim AVR 8515 ist dieser Bereich von $20 bis $5F. Die definierten Variablen werden jetzt fortschreitend ab $20 platziert. PDATA ist für einen IO-Bereich reserviert, falls vorhanden.
Variablen im Bereich $PDATA werden meistens mit speziellen Maschinenbefehlen erreicht. Bei der Definition von Variablen in diesem Bereich lässt man nicht den Compiler die Adressen vergeben, sondern der Programmierer muss/sollte zu jeder Variablen die gewünschte Adresse angeben:

Var *Port1[$35] : byte;*

{$IDATA}
$IDATA weist allen nachfolgenden Variablen Deklarationen (dazu zählt auch **StructConst**) den im Prozessor Steuerfile (xxx.dsc) unter IDATA ausgewiesenen Bereich zu. Beim AVR 8515 ist dieser Bereich von $60 bis $25F. Die definierten Variablen werden jetzt fortschreitend ab $60 platziert. Folgt ein anderer Schalter dieses Typs, wird mit den folgenden Variablen analog dazu verfahren. Variablen im Bereich $IDATA werden meistens mit etwas längeren und damit langsameren Maschinenbefehlen erreicht.

Der iData Bereich umfasst normalerweise den kompletten SRAM Bereich der CPU. Falls notwendig kann dieser Bereich aber nochmal unterteilt werden. Dies geschieht mit einem Define. Mit iDATA1 kann das interne RAM in zwei Teile aufgeteilt werden. Der notwendige Parameter bestimmt dabei das Ende des ersten Teils und gleichzeitig den Anfang des zweiten Teils. (* REV4 *)

```
Define iData1     = $800;
```

Variablen können jetzt entweder in iData oder iData1 plaziert werden.
Beispiel:

```
{$IDATA1}
Var
  Test1     : word;

{$IDATA}
Var
  Test     : word;
```

{$XDATA} {$XDATA1} {$XDATA2} {$XDATA3} {$XDATA4}
$XDATA weist allen nachfolgenden Variablen Deklarationen (dazu zählt auch **StructCons**) den mit **Define** XDATA ausgewiesenen Bereich zu. XDATA ist externer Speicher, der nur bei den grösseren Typen anzutreffen ist. Folgt ein anderer Schalter dieses Typs, wird mit den folgenden Variablen analog dazu verfahren. Variablen im Bereich $XDATA werden immer mit längeren und damit langsameren Maschinenbefehlen erreicht. Oft setzt die CPU noch zusätzliche Waitstates ein.

```
Define xDataWaits  = nn;      // 0..3
```
Erlaubt z.B. beim mega8515, mega162, mega64 und mega128 die Einstellung der xData Waits.

```
Define  XDATA1 = $8000, $80FF, NoInit;
```
Jeder XDATA Bereich kann mit dem Attribut **NoInit** versehen werden.

{$EEPROM}
$EEPROM weist allen nachfolgenden Variablen Deklarationen (dazu zählt auch **StructConst**) den im Prozessor Steuerfile (xxx.dsc) unter EEprom ausgewiesenen Bereich zu. EEprom kann nur ein Chip-interner Speicher sein.

Der EEprom Bereich umfasst normalerweise das komplette interne EEprom der CPU. Falls notwendig kann dieser Bereich aber nochmal unterteilt werden. Dies geschieht mit einem Define. Mit EEPROM1 kann das interne EEprom in zwei Teile aufgeteilt werden. Der notwendige Parameter bestimmt dabei das Ende des ersten Teils und gleichzeitig den Anfang des zweiten Teils. (* REV4 *)

```
Define EEprom1     = $200;
```

Variablen können jetzt entweder in EEPROM oder EEPROM1 plaziert werden.
Beispiel:

```
{$EEPROM1}
Var
  Test1     : word;

{$EEPROM}
Var
  Test     : word;
```

{$UDATA}
$UDATA weist allen nachfolgenden Variablen Deklarationen den im Definitionsteil deklarierten UserData (UserDevice) Bereich zu. Dieser Datenbereich liegt in einem externen Device, das nicht über die normale CPU-Adressierung angesprochen werden kann, z.B. serielles Eeprom. Der Programmierer muss dazu einen Device Treiber bereitstellen. Siehe auch Abschnitt **Device Treiber** im *Standard Driver Manual*.

Die Variablen bauen sich immer von den niederen Adressen zu den höheren auf. Wechseln die Speicherbereiche durch einen neuen Schalter, so wird mit dem aktuellen Speicher an der **zuletzt vergebenen Adresse** fortgefahren.

Speicherinitialisierung
Die Speicherbereiche $DATA, $IDATA und die $XDATA Bereiche werden normalerweise auf $00 initialisiert. Diese Initialisierung kann man bei den $XDATA Bereichen schon bei der Definition mit dem Attribut **NOINIT** für den ganzen Bereich abschalten. Für die Bereiche $DATA und $IDATA ist folgender Compiler Schalter vorgesehen:

{$NOINIT}
Schalter für den $DATA und den $IDATA Bereich. Die nachfolgenden Variablen dieses Bereiches bis zu dessen Ende werden nicht initialisiert, d.h. nicht auf 0 gesetzt.
Man sollte jedoch dabei beachten, dass die Standard Initialisierung in einer sehr schnellen Schleife in einem Block erfolgt. Werden jetzt noInit Variable beliebig dazwischen gestreut, kann dieses Init u.U. wesentlich langsamer ablaufen und auch mehr Code verbrauchen. Man sollte also ggf. diese Var Definitionen zusammen fassen, so dass keine allzu grosse Stückelung erfolgt.

Da $NOINIT bis auf Widerruf gilt, muss ggf. das NoInit wieder zurückgesetzt werden mit
{$NOINIT OFF}

Variablen ausserhalb aller Bereiche
Soll aus bestimmten Gründen eine Variable eine Adresse erhalten, die nicht in einem schon deklarierten Bereich liegt, wird eine Fehlermeldung ausgegeben. Die Fehlermeldung kann mit nachfolgendem Compilerschalter unterdrückt werden:

{$NORAMCHECK}
Die nachfolgende Variable wird nicht auf einen gültigen Speicherbereich geprüft.

```
{$IDATA}
{$NORAMCHECK}
var
  Extreme[@$FFFF]  : byte;
```

{$PHASE} {$DEPHASE}
Schaltet auf feste **WORD**-Adresse im Flash um / zurück auf die Standard Code Page

{$PHASE $1E00};	legt den nachfolgenden Code ab Adr $1E00 ab.
{$DEPHASE} ;	schaltet wieder um auf die Standard Code page.

{$ALIGN2}, {$ALIGN4}, {$ALIGN8}
Plaziert die nachfolgende Variable auf eine Adresse die durch 2, 4, 8 ohne Rest teilbar ist.

7.1.1 Überlegungen zur Speicherbelegung

Wie schon ausgeführt, ist der offensichtliche Unterschied zwischen den einzelnen Speicherbereichen der Zeitbedarf und auch der Programmspeicher Verbrauch, bedingt durch die jeweiligen Adressierungsarten der CPU. Bei zeitkritischen Anwendungen sollte der Programmierer seine Variablen nach der Häufigkeit der Zugriffe entsprechen zuweisen.

Default: {$DATA}

```
var
      b1      : byte;          {$06}
      ch1     : char;          {$07}
      w1      : word;          {$08}
      bool1   : boolean;       {$0A}

{$IDATA}
var
      b2      : byte;          {$60}
      ch2     : char;          {$61}
      w2      : word;          {$62}
      bool2   : boolean;       {$64}

{$PDATA}
var
      Port1[$25]     : byte;   {$25}
      Timer4[$34]    : char;   {$34}
      Per2[$28]      : word;   {$28}

{$DATA}
var
      x       : byte;          {$0B}
      y       : char;          {$0C}

{$EEPROM}
var
      ex      : byte;          {$0}
      ey      : char;          {$1}
```

7.2 XData Externer Speicher

Ein evtl. vorhandener externer Speicher muss durch die Compilerschalter {$XDATA} {$XDATA1} {$XDATA2} {$XDATA3} und {$XDATA4} eingestellt werden.

Mit der Definition eines XDATA Bereichs wird ein entsprechender externer Speicherbereich freigegeben. Die Möglichkeit den externen Bereich in maximal 5 Abschnitte aufzuteilen unterstützt auch z.B. externe Peripherie Bausteine und Accu-gepuffertes RAM.

Im Normalfall initialisiert der Compiler allen vorhandenen Speicher mit “0”. Bei Peripherie Bausteinen oder Accupufferung ist das nicht erwünscht. Deshalb kann bei der Deklaration eines externen Bereichs das optionale Attribut **NoInit** angehängt werden. Damit wird eine Initialisierung dieses Bereichs unterbunden.

```
Define  Xdata =  StartAddr, EndAddr [, NoInit];
...
var
{$XDATA}
      abc             : integer;
      fix[$3000]      : byte;
{$XDATA1}
      port1[$8000]    : byte;
      port2[$8001]    : byte;
{$IDATA}
```

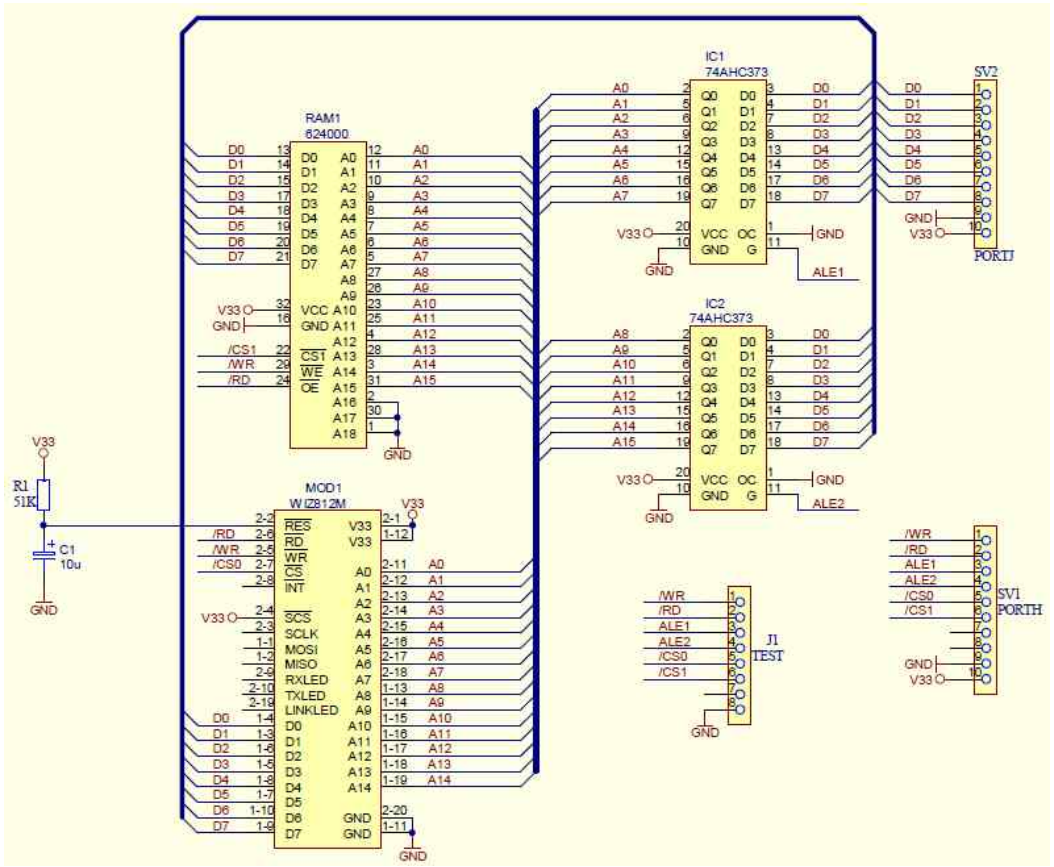
Wird externer Speicher verwendet und definiert, so muss zumindest und als erstes der Bereich **XDATA definiert** werden. Anschliessend können weitere Bereiche mit **XDATA1** etc. folgen. Die Definition beinhaltet die Start- und Endadresse des jeweiligen Speicherblocks, gefolgt von dem optionalen **NoInit.**

{$XIO +} {$XIO -}
Kann im XDATA Bereich verwendet werden um dem Optimierer z.B. Memory Mapped IO-Bereiche zu signalisieren. Der Bereich innerhalb dieser beiden Schalter wird getrennt behandelt bzw. redundante Zugriffe werden nicht entfernt.

XMega
Beim XMega kann statt $XDATA auch $XDATA0 verwendet werden. Weiterhin sind hier nur max. 4 Bereiche zulässig: XDATA0, XDATA1, XDATA2, XDATA3. Es wird nur 64kB unterstützt mit SRAM und Memory Mapped IO. Dazu wird das 2 Port Interface verwendet im LPC Mode.

Durch die interne Logik der XMegas bestehen noch weitere zu beachtende Details. Die minimale Blockgrösse für ein XData (= /CS) ist 4kByte. Die physische Blockgrösse wird durch die angegebene logische Blockgrösse bestimmt. Ist z.B. XDATA0 = $5000, $7FFF dann ist log Block = $3000 aber der phys Block = $4000. Damit muss dieser Block auf $4000 beginnen, denn ein Adressblock muss immer auf der Blockgrenze beginnen.
Beispiele:
Log Blockgrösse $1000 -> Blockgrösse = $1000 Blockstart = $0000, $1000, $2000, $4000 etc.
Wenn ein solcher log. 1kByte block auf Adresse $5000 beginnen und auf $5FFF enden soll, dann muss er physisch auf $4000 beginnen und auf $5FFF enden. Das sind jetzt $2000 phys. Grösse.
Der AVRco unterstützt die dazu notwendigen Kalkulationen und gibt im Fehlerfall eine Meldung aus.

Eine weitere Besonderheit beim XMega ist das sog. Overlapping. Damit kann ein höherwertiges /CS bzw. Block einen niederwertigen überschreiben. Die höchste Wertigkeit hat dabei iDATA gefolgt von XDATA0/CS0, XDATA1/CS1 etc. Um jetzt Zersplitterung einzelner Datenbereiche zu vermeiden muss der kleinste Block am obersten 64k Ende (XDATA0/CS0) gelegt werden. Der nächstgrössere liegt dann etwas darunter (XDATA1/CS1). Diese ganze Logik ist zuerst etwas undurchsichtig, da hilft ein Blick ins XMegaA1 Datenblatt weiter.



Ein Beispiel Programm befindet sich in der Demos Directory unter ...\Demos\XMega_XDATA

7.3 Include Dateien

{$I Filename.ext}
Liest eine Include Datei, wobei "Filename" auch ein Pfad enthalten kann. Hiermit lassen sich Sourcen (Konserven), die immer wiederverwendet werden, einbinden. Die Include Datei kann sowohl Assembler als auch Pascal-Source als auch beides enthalten. Die üblichen Konventionen des Compilers gelten hierbei natürlich weiterhin.

{$J Filename.ext}
Liest eine Include Datei, wobei "Filename" keinen Pfad enthalten darf. Als Pfad wird grundsätzlich die "Home-Directory" des Compilers vorangestellt. Das ist sehr vorteilhaft für immer wiederkehrende Prozeduren etc. Und ersetzt fast den Linker bzw. das Unit-Konzept.

7.3.1 Suchpfad für Include Dateien

Include Dateien werden in folgender Weise gesucht:
1. aktuelle Arbeits/Projekt Directory
2. unter Project Admin (IDE) angegebene Suchpfade
3. unter System Admin (IDE) angegebene Suchpfade
4. im AVRco Verzeichnis
5. im "System" Verzeichnis unterhalb des AVRco Verzeichnis

7.4 Runtime Checks

Solange genug Speicher vorhanden ist, kann “StackSize” vergrössert werden. Meldet jetzt der Compiler einen Stack-Überlauf, so hilft nur noch die Elimination von Variablen. Dies kann durch entfernen unbenutzter Variablen geschehen. Falls keine vorhanden, muss mit Doppelbelegung von Variablen gearbeitet werden. Hier muss der Programmierer extrem vorsichtig sein, um eine zeitliche Überschneidung der Belegung zu verhindern.

```
Var   ch1          : byte;
      ch2[@ch1]    : byte;     {ch1 und ch2 gleiche Adresse}
```

Bitte Kapitel **RunTimeErr** beachten.

{$ShowError err: string}
erzeugt eine Fehlermeldung
{$ShowError 'Fehler bei ...'} erzeugt die Fehlermeldung beim Assemblieren

{$ShowWarning err: string}
erzeugt eine Warnung
{$ShowWarning 'Achtung: ...'} erzeugt die Warnung beim Assemblieren

7.5 Variable, Konstante und Prozeduren Check

{$WG}
Schaltet global die Überwachung von Variablen, ROM-Konstanten und Prozeduren ein. Bei offensichtlich unbenutzten Variablen oder Prozeduren erzeugt der Compiler dabei eine Warnung, die durch die IDE ausgewertet wird.
Dieser Schalter ist default off und kann nur aktiviert werden.

{$W+} {$W-}
Schaltet die Überwachung von Variablen, ROM-Konstanten und Prozeduren für das aktuelle Modul (Main/Unit) ein bzw. aus. Bei offensichtlich unbenutzten Variablen oder Prozeduren erzeugt der Compiler dabei eine Warnung, die durch die IDE ausgewertet wird. (siehe auch Optimierung)
Default: {$W-}

{$NORETURNCHECK}
Gilt nur in Zusammenhang von Funktionen. Bei der nachfolgenden Funktion erfolgt keine Fehlermeldung, wenn das Return Statement fehlt.

{$NOOVRCHECK}
Schaltet die Prüfung für die nachfolgende Variablen Overlay Deklaration ab.

{$OverLay @VarName[, NoOvrCheck]} {$OverLay 0}
Um das übereinander Mappen von Variablen nicht für jede Variable einzeln mit: *yyy[@xxx] : byte;* durchführen zu müssen.

Damit können z.B. mehrere Variablen in ein vorhandenes Array reingelegt werden.
VarName bezeichnet eine beliebige, existierende Variable (@VarName) im RAM Bereich. Der optionale Parameter "*NoOvrCheck*" bestimmt, dass keine Bereichsprüfung stattfinden soll.

Wird der Switch mit dem Parameter "0" übergeben, dann schliesst dieser den Overlay Bereich ab. Alle Variablen, die zwischen den beiden Switches platziert werden, erhalten jetzt aufsteigende Adressen ab "VarName". D.h. Sie werden auf die ursprüngliche Variable platziert, die normalerweise in der Lage sein sollte alle neuen auch aufzunehmen.

Ein Overflow wird beim abschliessenden Compiler Switch angezeigt, bzw. ignoriert wenn die Option "*NoOvrCheck*" gesetzt wurde.

Statt einer Variablen als Basis kann auch eine absolute Adresse durch
{$OverLay $nnnn, NoOvrCheck}
angegeben werden. Die Angabe von "*NoOvrCheck*" ist hier dann zwingend.

{$Q-}
schaltet das Qualifizieren innerhalb des Assembler Codes ab.
Nur für schon vorhandene ältere Programme.

{$TYPEDCONST OFF} {$TYPEDCONST ON}
Für eine bessere Lesbarkeit des Programms und zur Vermeidung von Compiler Fehlern.
Mit "ON" erwartet der Compiler mit jeder Konstanten Deklaration auch die zugehörige Typ Deklaration.
Damit wird z.B. eine "0" auch eindeutig entweder ein Byte, Word, Integer oder Float. Die Option „OFF“ sollte nicht mehr verwendet werden!

```
const bb : byte = 0;
```

Der Schalter ist default "ON".
Wenn man vorhandene Programme nicht umschreiben will, so sollte gleich beim Programm Beginn der Schalter so verwendet werden:

{$VALIDATE name}
Konstante und Variable des Systems als auch der Applikation werden vom Compiler in der Regel "wegoptimiert", wenn diese nicht im Kontext angesprochen werden, d.h. wenn sie nicht irgendwo in einem Statement auftauchen.

Das kann zu Problemen beim Inline Assembler Code in der Pascal Source führen. Der Assembler gibt dann einen Fehler aus, da die angesprochene Prozedur etc. nicht vorhanden ist.
Mit diesem Schalter wird der Compiler gezwungen, auf jeden Fall das Konstrukt "name" zu importieren bzw. die Optimierung greift hier nicht. Der Schalter kann nur nach der Deklaration von “name“ verwendet werden.
ACHTUNG:
diese Option funktioniert (als einzige) nicht mit dem *"Merlin Optimiser"*

{$VALIDATE $}
Funktionen und Prozeduren können durch den vorangestellten Schalter von jedweder Optimierung ausgeschlossen werden.

{$VALIDATE_ALL}
Damit wird der Compiler veranlasst auch offensichtlich im Programm nicht benutzte Konstante vom Typ String, Array und Record im Programmcode abzulegen. Die Optimierung für diese Art der Konstanten ist damit komplett abgeschaltet.

{$VALIDATE_ON} {$VALIDATE_OFF}
damit werden ganze Blöcke von Variablen als "benutzt" gekennzeichnet, so dass die Meldung "possibly unused variable" für diesen Bereich nicht auftaucht.

{$ZeroLocVars +}
Wenn aktiv, werden alle lokalen Variablen in Funktionen und Prozeduren auf 0 gesetzt wenn eine Funktion oder Prozedur aufgerufen wird.

{$NOADDRCHECK}
Für Mega128..256: Das Platzieren von Konstanten erfolgt immer in die höchste Flash Page. Andere Pages sind eigentlich nicht dafür vorgesehen. Durch diesen Compiler Schalter wird die Bereichs Prüfung für die nachfolgend definierte Konstante abgeschaltet. Diese Konstante kann ein externes binäres File sein das hier abgelegt wird. Liegt die Vorgabe Adresse ausserhalb der höchsten Flashpage (standard constant page), so ist ein Zugriff auf diese Konstante durch die Applikation nur mit ganz speziellen selbst zu erstellenden Verfahren möglich.

```
const
{$NOADDRCHECK}
   LookUpTab[$20000] : array[1..256] of byte = 'Name.ext';  //mega2561
```

{$REUTILIZE }

XMega

dient dazu dass Timer, SPI und TWI für verschiedene Treiber doppelt benutzt werden können.
Wenn z.B. der Treiber A den SPI_C benutzt und der Treiber B soll auch SPI_C benutzen,
kommt es zu einer Fehlermeldung. Um dies trotzdem zuzulassen wird bei der zweiten Verwendung
des SPI_C im DEFINE dieser Schalter vorangestellt:

```
Define
  DriverA   = SPI_C;
{$REUTILIZE SPI_C}
  DriverB   = SPI_C;
```

Zulässige Argumente sind TIMER_C0..TIMER_F1, SPI_C..SPI_F, TWI_C..TWI_F

7.6 System Steuerung

{$NOSAVE}
Gilt nur in Zusammenhang von Applikations Interrupt Prozeduren. Bei der nachfolgenden Interrupt Prozedur sichert das System die Arbeitsregister nicht automatisch, ausgenommen des Status Registers und der 4 Haupt Arbeitsregister. Der Programmierer muss selbst dafür sorgen. Nur für schnelle, in Assembler geschriebene Service Routinen sinnvoll.

{$NOREGSAVE}
Gilt nur in Zusammenhang von Applikations Interrupt Prozeduren. Bei der nachfolgenden Interrupt Prozedur sichert das System absolut keine Arbeitsregister. Der Programmierer muss selbst dafür sorgen. Nur für schnelle, in Assembler geschriebene Service Routinen sinnvoll.

{$NOSHADOW}
Die Definition muss, falls benötigt, noch vor der Device Deklaration erfolgen. Bei non-Multitask Anwendungen werden bei allen Interrupts nur die durch die Interrupts benutzten Register gesichert. Dies spart wesentlich Ram, Rom und Rechenzeit. Dieser Schalter wird durch den Import von Prozesse und Tasks überschrieben.

{$NOFRAME}
Gilt nur in Zusammenhang mit Device Treiber Prozeduren, die durch die Prozedur **Write** aufgerufen werden. Bei der nachfolgenden Device Prozedur, die nur einen 8bit Übergabe Parameter besitzen darf, wird dieser Parameter in einem Register übergeben. Ein Parameterframe wird nicht gebildet. Damit sind auch keine lokale Variablen möglich. Nur für schnelle in Assembler geschriebene Treiber Routinen sinnvoll.

{$DEBDELAY}
Speziell für den Simulator. Verkürzt die mDelays innerhalb des Simulators um ca. 90%. Dieser Schalter hat keinerlei Einfluss auf das generierte Hexfile, d.h. er muss nicht entfernt werden.

{$D+} {$D-}
Debug Informationen ein bzw. aus. Wenn aus, werden die folgenden Statements nicht im Single Step Modus des Simulators abgearbeitet. Dieser Schalter hat keinerlei Einfluss auf das generierte Hexfile, d.h. er muss nicht entfernt werden.

{$X-} {$X+}
Schaltet das Abarbeiten eines Code Bereiches im Simulator aus bzw. ein. Nützlich wenn auf externe Hardware gewartet wird. Dieser Schalter hat keinerlei Einfluss auf das generierte Hexfile, d.h. er muss nicht entfernt werden.

{$DEVICE}
Gilt nur in Zusammenhang mit Device Treiber Prozeduren, die durch die Prozedur **Write** und **Read** aufgerufen werden. Bei der nachfolgenden Device Prozedur, die nur einen 8bit Übergabe Parameter besitzen darf, wird dieser Parameter in einem Register übergeben. Ein Parameterframe wird nicht gebildet.

Damit sind auch keine lokale Variablen möglich. Nur für schnelle in Assembler geschriebene Treiber Routinen sinnvoll. Identisch mit dem $NOFRAME Schalter.

{$LCDNOWAIT}
Schaltet das Busy-Polling des Display Treibers ab. Nur für Debug Zwecke!

{$LCDNOINIT}
Das LCD Display wird beim Reset nicht initialisiert. Das Anwendungs Programm macht dies selbst durch den Aufruf der System Prozedur "*LCDsetup*".

{$ENUMTOASM}
Enumerationen (Aufzählungstypen) werden normalerweise nicht in das Assemblerfile exportiert, um Rechenzeit im Compiler und Assembler zu sparen und die Dateien übersichtlich zu halten.
Wird im Assembler bzw. mit InLine Assembler Code die Werte der Enumeration gebraucht, so kann mit diesem Compilerschalter der Export der Enum-Werte in das Assembler File erzwungen werden.

{$SL+} {$SL-} (*P*) zur Zeit abgeschaltet ! Bitte stattdessen den Merlin Optimiser benutzen !
Schaltet den Smart-Linker ein oder aus. Damit wird das Code löschen freigegeben (+) oder gesperrt (-). Diese Schalter können beliebig gesetzt werden.
Zu beachten ist dass der Schalter am Anfang einer Unit immer auf "off" gestellt ist. Das gleiche gilt für das Hauptprogramm File.

{$SL ON} (*P*) *) zur Zeit abgeschaltet ! Bitte stattdessen den Merlin Optimiser benutzen !
Schaltet beim Programmstart das Default Verhalten des Linkers auf aktiv
Bei aktivem Schalter (on) kann eine Code Generierung für einzelne Funktionen oder Prozeduren auch erzwungen werden, indem der Schalter *{$VALIDATE ProzedurName}* eingesetzt wird.
Dieser Schalter wird aber erst wirksam, nachdem diese Funktion schon dem System bekannt, d.h. definiert ist. Als Alternative dazu ist vor einer Funktion/Prozedur Deklaration der Schalter *{$VALIDATE $}.*

Anstatt der obsoleten $SL Schalter kann der Merlin Optimser verwendet werden. Wenn nur die $SL Funktionen des Merlin gebraucht werden, muss dieser Optimiser Schalter verwendet werden:
{$OPTI SMARTLINK_ONLY}

{$OPTIMISE}
Der Schalter bestimmt die Compilierung unter Einbezug des Merlin Optimisers. Der Schalter **muss** in der ersten Zeile des Hauptprogramms stehen!

{$PCU} (*P*)
Der in der IDE angesiedelte global innerhalb des Projekts wirkende Schalter "Project/Project Options" wirkt sich auf alle Units des Projekts aus. Wenn aktiviert, werden alle Units des Projekts vorcompiliert und PCU Dateien werden erzeugt, abhängig von den weiteren Vorgaben ("copy" Schalter).
Durch Einsatz dieses Compiler Schalters im Source Bereich einer Unit wird die Generierung einer PCU für diese Unit erzwungen, unabhängig davon wie der globale Schalter in der IDE steht.
Die "copy" Schalter in der IDE sind auch hier wirksam.

{$VectTab $nnnn}
{$CodeStart $nnnn}
Bei normalen AVR Applikationen steht die Interrupt Vektor Tabelle immer ab Adresse $0000 im Code Bereich (Flash). Das gleiche gilt auch für den Codestart, d.h. auch der generierte Code beginnt offiziell auf Adresse $0000 bzw. exakt hinter der Vektor Tabelle. Mit diesen beiden Schaltern kann nun das System angewiesen werden, die Adress Generierung für diese Bereiche zu verschieben.
Die Adress Parameter für die Schalter sind dabei immer Word Adressen!

{$BootApplication $nnnn}
Dieser Schalter ist eine Kombination aus den obigen zwei. Zusätzlich ist es hier möglich den Flash Downloader einzubinden. Damit wird es möglich eine Applikation zu erstellen, die nur im Boot Bereich des Controllers läuft und trotzdem alle System Resourcen und Treiber zur Verfügung hat da diese auch alle in den Boot Bereich plaziert werden. Dieser Schalter muss nach der Device Deklaration platziert werden.
Der Adress Parameter für diesen Schalter ist immer eine Word Adresse!

Eine Beispiel Applikation ist in der Demos Directory unter "BootApp" zu finden.

7.7 Optimiser Schalter

siehe Kapitel “Der Merlin Optimierer“

7.8 Conditional Compile

Es ist manchmal notwendig, aus einem Programm unterschiedliche, z.B. hardware-abhängige, Versionen zu generieren. Das jeweilige Verhalten des Compilers kann dazu mit Hilfe der Compiler-Schalter für abhängige Compilation (Conditional Compile) gesteuert werden. Das dazu verwendete “***Label***” hat dabei nur symbolischen Charakter. Ist das Ergebnis eines Schalters “falsch”, so wird der ab hier beginnende Source-Code bis zum “wahr”-werden des Schalters als Kommentar behandelt bzw. existiert nicht. Alle Schalter dieser Gruppe können an beliebiger Stelle in der Source vorkommen. Eine “IFxx”-Anweisung muss immer mit einem “ENDIF” abgeschlossen sein. Dazwischen kann ein “ELSE” liegen. Verschachtelte Compilerschalter sind ebenfalls zulässig!!

LABELS können auch in der IDE PED32 unter ***Project/Project Options*** definiert werden. Mehrere Labels müssen durch Strichpunkte getrennt werden. Hierbei dürfen jedoch nur die “nackten” Labels stehen ohne $ und ohne DEFINE etc. PED32 informiert den Compiler über die DEFINES und dieser behandelt sie so, als wären sie direkt in der Source in der ersten Zeile.

{$DEFINE label}	Setzt “label” auf true
{$UNDEF label}	Setzt “label” auf false
{$IFDEF label}	Wenn “label” true ist, wird die nachfolgende Source compiliert bis zum “ELSE” bzw. “ENDIF”. Ist “label” false, wird umgekehrt verfahren.
{$ELSIFDEF label}	Wenn “label” true ist, wird die nachfolgende Source compiliert.
{$IFNDEF label}	Wenn “label” false ist, wird die nachfolgende Source compiliert bis zum “ELSE” bzw. “ENDIF”. Ist “label” true, wird umgekehrt verfahren.
{$ELSE }	Kehrt den momentanen Status um. Wurde z.B. die vorhergehende Source compiliert, so wird die nachfolgende Source bis zum “ENDIF” als Kommentar behandelt.
{$ENDIF}	Schliesst einen Conditional-Block ab.

Es können auch einfache bool'sche Ausdrücke angegeben werden. Als Operatoren sind nur "AND" und "OR" zulässig und als Argumente nur solche die mit {$DEFINE ..} deklariert wurden.

```
{$IFDEF ABC AND XYZ}
...
{$ELSIFDEF HIJK OR OPQ}
...
{$ENDIF}
```

Ebenso können Klammern verwendet werden:

```
{$IFNDEF ABC and (UVW or XYZ)}
...
{$ENDIF}
```

Weiterhin kann das Vorhandensein von bestimmten Units abgefragt werden.
Dazu werden die Unit Namen und Unitfile Namen werden automatisch in den Define Pool aufgenommen.
Wenn der Unit-FileName "ABC.pas" ist und der interne Unit Name "U245" ist dann geht folgendes:

{$IFDEF FILE_ABC}	ist true, da das Unit File "ABC" heisst.
{$IFDEF U245}	ist true, da die Unit "U245" heisst.

```
Program Test;
{$DEFINE pp}

{$IFDEF pp}
Procedure ABC;      {Prozedur Kopf}
begin               {wird compiliert und ausgeführt}
  ...
end;

{$ELSE}
Procedure ABC;      {Prozedur Kopf}
begin               {wird als Kommentar behandelt}
  ...
end;
{$ENDIF}
```

{$IF equation = true} {$ELSIF equation = true}
statt mit $DEFINE vorgegebene Werte können auch Konstante des Systems oder des Programms selbst benutzt werden:

```
const
    x   : 25;
    y   : 1;

{$IF x > y}
...
{$ELSIF x = y}
...
{$ENDIF}

{$IF PROCCLOCK = 8000000}
```

Beispiele für Systemkonstante die für Compiler Schalter (und auch in Statements) verwendet werden können:

```
_iDataStart
_iDataEnd
_EEpromStart
_EEpromEnd
_FlashStart
_FlashEnd
```

{$IFDEF CPUname}
Mit diesem Compiler Schalter kann CPU Typ abhängiger Code erzeugt werden.

{$HEXPATH 'pathname'}
Werden mit "conditional compile" aus einer Source unterschiedliche Firmware Versionen erstellt, ist es sinnvoll, die generierten Hex-Files auch in unterschiedliche Directories abzulegen. Dazu wurde dieser Compiler Schalter implementiert.
Alle beteiligten Tools, die aus der IDE heraus aufgerufen werden (Editor, Compiler, Assembler, Programmer) beachten diesen Schalter.
Das Argument 'pathname' muss in Hochkomma dargestellt sein. Falls der Pfad bzw. Directory nicht existiert, wird der Pfad und Directory neu angelegt.

{$HEXNAME 'filename'}
Werden mit "conditional compile" aus einer Source unterschiedliche Firmware Versionen erstellt, ist es sinnvoll, die generierten Hex-Files auch unterschiedlich zu benennen. Dazu wurde dieser Compiler Schalter implementiert.
Alle beteiligten Tools, die aus der IDE heraus aufgerufen werden (Editor, Compiler, Assembler, Programmer) beachten diesen Schalter.
Das Argument 'filename' muss in Hochkomma dargestellt sein. Beide Schalter können auch in Kombination verwendet werden.

Bemerkung:
Eine Kopie der Flash und EEprom Hexfiles wird auch weiterhin in der Projekt Directory unter dem jeweiligen Original Namen abgelegt. Der InCircuit Programmer kann dann direkt mit dem neuen bzw. geänderten Projekt aus der IDE heraus gestartet werden.
Im stand alone Betrieb des Programmers müssen dann allerdings die neuen Directories auch als neue Projekte angelegt werden.

8 Programm Aufbau

8.1 Programm Rahmen

Aus formalen Gründen ist ein bestimmter Programmrahmen notwendig. Dieser beginnt mit 'Program *name*' und endet mit 'End.' Wie an verschiedenen Stellen nachzulesen, müssen oder können diverse Devices, Systemfunktion etc. **importiert** bzw. **definiert** werden.

Der Programmierer sollte sich deshalb untenstehendes Beispiel als Vorbild für den eigenen Programm Aufbau nehmen. Beim Neuerstellen eines Projekts über die IDE PED32 wird automatisch eine Haupt-Datei mit den entsprechenden Einträgen erstellt. Voraussetzung dafür ist jedoch das Vorhandensein eines sog. Template-Files (*.tmpl) und der Eintrag dieses Files im entsprechenden “Control” von PED32.

In manchen Versionen ist ein sog. **Application Wizzard** enthalten, mit dessen Hilfe man interaktiv die Basis (source) einer neuen Applikation erstellen kann.

Selbstverständlich erzeugt der Compiler bei falschen oder fehlenden Imports oder Defines einen Syntaxfehler.

Unnötige Imports oder auch Variablen Deklarationen sind unter allen Umständen zu vermeiden. Denken Sie an die bescheidenen Resourcen (Ram) der meisten Prozessoren. Spätestens beim Ineinanderlaufen von Variablen und Stack meldet der Compiler eine Warnung.

Verschachtelte Prozeduren, lange arithmetische Statements mit vielen Klammern oder Prozeduraufrufe innerhalb von Interrupts führen zu Laufzeitproblemen mit dem **Stack-Überlauf** (Parameterstack).

8.1.1 Reihenfolge

Die **Reihenfolge** der Deklarationen von *Program* bis zu *Implementation* ist **zwingend** vorgeschrieben. *Const* und *Var* Deklarationen können anschliessend beliebig gemischt vorkommen. **Nach der ersten** *Prozedur* oder *Funktions* Deklaration sind **möglichst** keine globalen *Var* oder *Const* Deklarationen mehr zu verwenden.

Lokale Variablen oder Konstante innerhalb einer Prozedur oder Funktion bedeuten einen grösseren Code und längere Laufzeit. Bei Prozeduren oder Funktionen, wo eine möglichst kurze Ausführungszeit erwünscht ist, sollte nur mit globalen Variablen gearbeitet werden, das gilt auch für Übergabe-Parameter.

Mit **String** und **Array** Variablen sollte man sehr sparsam umgehen. Der Speicherverbrauch kann sehr schnell die Systemressourcen erschöpfen. Das gleiche gilt für die String-Konvertierungen.

8.2 Initialisierung

Nach einem Reset bzw. Programmstart wird der komplette Speicher (Variablen) auf null (var:= 0;) gesetzt. Damit brauchen globale Variable in der Regel nicht durch das Programm vorinitialisiert zu werden.
Die Initialisierung erfolgt unmittelbar nach der Anweisung *Implementation* bzw. nach dem Aufruf der Prozedur **System_Init**, falls vorhanden.

Unter Umständen kann es notwendig sein, bestimmte Speicherstellen nicht zu initialisieren bzw. zu löschen. Dies muss dem Compiler mit dem Compiler Schalter **{$NOINIT}** mitgeteilt werden. Hierbei ist jedoch zu beachten, dass ab diesem Schalter absolut keine Initialisierung mehr erfolgt. D.h. alle Speicherstellen, die vor dem Schalter liegen, werden gelöscht, und alle die nach diesem Schalter definiert werden, werden nicht gelöscht.

Der Compiler bzw. dessen System Bibliothek nimmt nur solche Hardware Initialisierungen vor, die für das Funktionieren der Importfunktionen notwendig sind. Wird z.B. **SwitchPort1** importiert, wird das dazu definierte Port auf Input geschaltet. Der Import **ADCport** initialisiert alle für den Betrieb des Wandlers notwendige Hardware. Einzelne Bits (z.B. bit7 von LCDport) werden nicht initialisiert und auch nicht verändert.

Es liegt also am Programmierer, die von ihm benötigte Hardware, die **nicht** von einem **Import** betroffen ist, **selbst zu initialisieren**. Die in der CPU vorhandenen IO-Ports werden im Prozessor-Beschreibungsfile *xxx.dsc* definiert und vom Compiler importiert. Die erneute Definition der Ports durch den Programmierer ist daher nicht notwendig. Beispiel für PortB Eingang/Ausgang gemischt:

```
var
   DDRB[@PortB -1] : byte;   { Data Direction reg PortB}
   Led1[@PortB, 0]   : bit;
   Rel1[@PortB, 1]   : bit;

begin                 {Main}
  DDRB:= $0F;         {obere 4 bits Input, untere 4 bits output}
  EnableInts;         {Interrupt Freigabe, falls notwendig}
  Incl (Led1);        {Led1 On}
  Loop
    Toggle (Rel1);    {Relais1 umschalten}
    mDelay(1000);     {1sec Verzögerung}
  endLoop;
end.
```

Beispiel für ein Programm Schema:

```
Program Test;

Device ...                      {Hardware Deklaration}

Import ...                      {System Funktionen/HardWare}

From System Import ..           {System Typen/Software}

Define ...                      {Hardware Definition}

Implementation ...              {Programm Start}

Type ...                        {Typen Deklaration}

Const  ...                      {Konstanten Deklaration}

Var    ...                      {Variablen Deklaration}


Procedure System_Init;          {optional}
begin
  ...
end;


Procedure ABC;                  {Prozedur Kopf}
begin
  ...
end;


Function CDE : boolean;         {Funktions Kopf}
begin
  ...
  Return(a > b);                {Ergebnis der Funktion}
end;


Process PPP (20, 10 : iData);   {Prozess Kopf}
begin
  ...
end;

Task TTTT (iData);              {Task Kopf}
begin
  ...
end;

begin                           {Haupt Programm Main}
  ...
  EnableInts;
  {Start_Processes;}            {wenn Prozesse importiert wurden}
  Loop
    ...
    ABC;
    x:= CDE;
  EndLoop;
end.
```

9 Compiler Errors

9.1 Fehler Datei

Stellt der Compiler irgendwelche Fehler fest, so generiert er eine Fehlerdatei mit der FileExtension ‘xxx.err’. Diese Datei benutzt die IDE PED32 zum lokalisieren der Fehler und deren Anzeige.

9.1.1 Type Missmatch

Anfänger im Programmieren werden relativ häufig über den Fehler **Type Missmatch** stolpern. Der erfahrene Programmierer wird sich genau über den gleichen Punkt wundern, da er u.U. die von Turbo Pascal, Delphi oder C her bekannte automatische Typ-Konvertierung gewöhnt ist.

Die im Compiler fast nicht vorhandene automatische Typ-Konvertierung bringt die o.a. Fehlermeldungen. Es gibt mehrere Gründe, dass es diese Art von Konvertierung im vorliegenden Compiler nicht gibt:

1. Es ist sehr schwierig und komplex eine einigermassen fehlerfrei funktionierende Automatik im Compiler zu realisieren.

2. Jeder Turbo-Programmierer kann ein Lied davon singen, wie nach den Ursachen von falschen Rechenergebnissen gesucht wurde, um schliesslich nach vielfachem Aufbrechen eines verschachtelten Statements/Ausdrucks festzustellen, dass die Typkonvertierung die Ursache dafür war. Zur Ehrenrettung von Turbo ist allerdings dann zu sagen, dass der Compiler formal immer richtig handelt, der Programmierer sich das aber ganz anders gedacht hatte. Oft hat nur eine zusätzliche Klammerung gefehlt.

3. Bei bestimmten Sicherheits relevanten Anwendungen ist eine Automatik in der Art nicht gern gesehen oder unerwünscht.

4. Es ist der bessere und sicherere Programmierstil, wenn der Programmierer gezwungen wird, dem Compiler genau zu ‘erklären’, was er eigentlich vorhat. Oder anders herum ausgedrückt: der Programmierer weiss am besten, was passieren soll und auf welche Art.

5. Type Casting (Typ Konvertierung) ist eindeutiger und lesbarer *word := word (byte);*

Ein kleines Problem soll an dieser Stelle jedoch nicht verschwiegen werden. Es kann bei Konstanten < 256 und bei bestimmten Operationen vorkommen, dass der Compiler nicht weiss ob er die Konstante als 8bit Wert oder 16bit Wert behandeln soll. Hierbei kann es dann u.U. zu einem Type Missmatch kommen.

```
var   b    : boolean;
      i    : integer;

      b:= 5 > i;
```

Kommt der Compiler an die Stelle ‘5’ weiss er nur, dass ein boolean erwartet wird und ‘5’ ein byte sein kann. Damit wird die ‘5’ als byte behandelt und löst durch den nachfolgenden integer ‘i’ ein Missmatch aus. Abhilfe:

```
      b:= integer(5) > i;
```

oder

```
      b:= i < 5;
```

10 Units (*P*)

10.1 Deklaration und Aufbau einer Unit

Eine Unit besteht aus Typen, Konstanten, Variablen und Routinen (Prozeduren und Funktionen). Jede Unit wird in einer separaten Unit-Datei (**.**PAS) definiert.
Eine Unit-Datei beginnt mit dem Unit-Kopf und enthält dann die Abschnitte **interface**, **implementation** (und optional **Initialization**). Die Struktur einer Unit-Datei sieht also folgendermaßen aus:

```
unit Unit1;

interface

uses  { Liste der verwendeten Units }

    { interface-Abschnitt }

implementation

    { implementation-Abschnitte }

initialization

    { initialization-Abschnitt }

finalization

    { finalization-Abschnitt }

end.
```

Eine Unit muß mit dem Wort **end** und einem Punkt abgeschlossen werden.

10.1.1 Unit-Kopf

Der Unit-Kopf gibt den Namen der Unit an. Er besteht aus dem reservierten Wort ***unit***, einem gültigen Bezeichner und einem abschließenden Semikolon. Der Bezeichner muss dem Namen der Unit-Datei entsprechen.

```
unit Hello;
```

Dieser Unit-Kopf kann in einer Quelltextdatei namens Hello.PAS verwendet werden. Die Datei mit der compilierten Unit trägt dann den Namen Hello.PCU.

Unit-Namen müssen in einem Projekt eindeutig sein. Auch wenn die Unit-Dateien in unterschiedlichen Verzeichnissen gespeichert werden, dürfen in einem Programm keine Units mit identischen Namen verwendet werden.

10.1.2 Interface-Abschnitt

Der Interface-Abschnitt einer Unit beginnt mit dem reservierten Wort ***interface***. Er endet mit dem Beginn des Implementation-Abschnitts. Der Interface-Abschnitt deklariert Konstanten, Typen, Variablen, Prozeduren und Funktionen, die für Clients verfügbar sind. Clients sind andere Units oder Programme, die diese Unit über die Uses-Klausel einbinden. Solche Bezeichner werden als öffentlich bezeichnet, da der Client auf sie wie auf Bezeichner zugreifen kann, die im Client selbst deklariert sind.

Die Interface-Deklaration einer Prozedur oder Funktion enthält nur den Kopf der Routine. Der Block der Prozedur bzw. Funktion wird dagegen im Implementation-Abschnitt definiert. Prozedur- und Funktions-Deklarationen im Interface-Abschnitt entsprechen also der forward-Deklarationen, obwohl die Direktive forward nicht verwendet wird.
Der Interface-Abschnitt kann eine eigene Uses-Klausel enthalten, die unmittelbar auf das Wort Interface folgen muß.

10.1.3 Implementation-Abschnitt

Der Implementation-Abschnitt einer Unit beginnt mit dem reservierten Wort ***implementation*** und endet mit dem Beginn des Initialization-Abschnitts oder - wenn kein Initialization-Abschnitt vorhanden ist - mit dem Finalization-Abschnitt bzw. dem Ende der Unit. Der Implementation-Abschnitt definiert Prozeduren und Funktionen, die im interface-Abschnitt deklariert wurden. Im Implementation-Abschnitt können diese Prozeduren und Funktionen in beliebiger Reihenfolge definiert und aufgerufen werden. Die Parameterlisten von Funktionen und Prozeduren müssen denjenigen der Deklaration im Interface-Abschnitt exakt entsprechen.

Außer den Definitionen der öffentlichen Prozeduren und Funktionen kann der Implementation Abschnitt Deklarationen von Konstanten, Typen, Variablen, Prozeduren und Funktionen enthalten, die für die Unit privat sind, auf die also Clients nicht zugreifen können.

Die Definition von absoluten globalen Konstanten ist mit *Define_USR* möglich.
Dieses Define sollte nur in "Notfällen" benutzt werden, denn es ist kein sauberer Programmierstil!
Besser ist es solche Definitionen in eine Unit zu platzieren, die sich am untersten Ende der Unit-Kette befindet. Dann sind die Definitionen auch von allen anderen Teilen her sichtbar.

10.1.4 Initialization-Abschnitt

Der Initialization-Abschnitt ist optional. Er beginnt mit dem reservierten Wort **initialization** und endet mit dem Finalization Abschnitt oder dem Ende der Unit. Der Initialization-Abschnitt enthält Anweisungen, die beim Programmstart in der angegebenen Reihenfolge ausgeführt werden. Arbeiten Sie beispielsweise mit definierten Datenstrukturen, können Sie diese im Initialization -Abschnitt initialisieren.

Die Initialization-Abschnitte von Units, die von Clients eingebunden werden, werden in hierarchischer Reihenfolge ausgeführt, das heisst, die Unit die sich ganz am Ende der Unit Kette befindet, wird als erstes initialisiert.

10.1.5 Finalization-Abschnitt

Der Finalization-Abschnitt ist optional. Er beginnt mit dem reservierten Wort **finalization** und endet mit dem Ende der Unit.
Die in diesem Block vorhandenen Statements werden abgearbeitet, wenn die Applikation die System Prozedur "*System_ShutDown*" aufruft.
Die Reihenfolge der Unit Aufrufe ist dabei genau umgekehrt wie die Unit-Initialization Aufrufe. Die Prozedur "*System_ShutDown*" ruft nur die einzelnen Finalization Statements auf, sonst nichts weiteres.
Das "Finalization" ist hilfreich beim gezielten Herunterfahren des Systems bevor einer Abschaltung oder einem Start des Sleep Modus.

10.1.6 Uses-Klausel

Die Uses-Klausel des Haupt-Programms gibt alle Units an, die in das Programm aufgenommen werden. Diese Units wiederum können eigene Uses-Klauseln enthalten. Eine Uses-Klausel in einem Programm oder einer Unit gibt die von diesem Modul verwendeten Units an. Eine Uses-Klausel kann an folgenden Stellen im Quelltext verwendet werden:

Mainfile eines Programms
Interface-Abschnitt einer Unit

Die Unit **System** wird automatisch von jeder Anwendung verwendet und darf nicht in der Uses-Klausel angegeben werden. (System implementiert Routinen für Hard- und Software Treiber, Stringverarbeitung, Floating Point, Speicherzuweisung usw.).

Eine Uses-Klausel besteht aus dem reservierten Wort ***uses***, einem oder mehreren durch Kommas voneinander getrennten Unit-Namen und einem abschließenden Semikolon.

```
uses Hello, MyMath;
```

In der Uses-Klausel eines Programms kann auf den Namen jeder Unit das reservierte Wort ***in*** mit dem Namen einer Quelltextdatei folgen. Der Name wird mit oder ohne Pfad in Hochkommas angegeben. Die Pfadangabe kann absolut oder relativ sein.

```
uses Hello, MyMath in 'C:\MyProg\MyMath', InitUnit;
```

Geben Sie **in** ... nach dem Unit-Namen ein, wenn Sie die Quelltextdatei einer Unit angeben müssen. Das reservierte Wort **in** wird nur benötigt, wenn die Position der Quelltextdatei nicht eindeutig ist weil die Unit-Quelltextdatei sich in einem anderen Verzeichnis befindet als die Projektdatei, und dieses Verzeichnis ist weder im Suchpfad des Compilers noch im Exe-Pfad des Compilers vorhanden ist.

10.1.6.1 Suchpfad für Units

Unit Dateien werden in folgender Weise gesucht:

1. aktuelle Arbeits/Projekt Directory
2. unter Project Admin (IDE) angegebene Suchpfade
3. unter System Admin (IDE) angegebene Suchpfade
4. im AVRco Verzeichnis
5. im "System" Verzeichnis unterhalb des AVRco Verzeichnis
   Mit dem AVRco evtl. mitgelieferte precompiled Units (PCUs) werden hier installiert

10.1.7 Info Teil einer Unit

Der Info Teil einer Unit besteht aus beliebig vielen Zeilen an beliebigen Stellen in dieser Unit.
Eine solche Info Zeile muss mit

```
||| hier der Info-Text
```

beginnen. Die drei Pipe Zeichen dürfen keine Leerzeichen enthalten. Eine Info Zeile wird vom System wie ein Kommentar behandelt.

In der IDE (Editor) ist ein Menu Punkt "Project/Unit infos". Alle Units die im SourceCodeControlSystem (SCCS) enthalten sind werden hier aufgelistet.
Ein Mausclick auf einen Unit Namen eröffnet ein Fenster welches den Info Teil dieser Unit anzeigt.

10.1.8 Hardware Imports innerhalb von Units

Wenn Units importiert sind, kann eine Definition auch innerhalb einer Unit erfolgen:

Hauptprogramm

```
Import SysTick, MatrixPort, SerPort;

From System Import longword, longint, float, pipes;

Define
    ProcClock    = 8000000;    {Hertz}
    SysTick      = 10;         {msec}
    StackSize    = $0020, iData;
    FrameSize    = $0040, iData;
    SerPort      = 9600;
    RxBuffer     = 16, iData;

DefineFrom unit1;    // Unit1 definiert das Matrixport
```

Unit

```
Unit Unit1;

Define
    MatrixRow    = PortD, 4;   {use PortD, start with bit4}
    MatrixCol    = PinD, 0;    {use PinD, start with bit0}
    MatrixType   = 3, 4;       {3 Rows at PortD, 4 Columns at PinD}

Interface
...
```

Mit dem reservierten Begriff "*DefineFrom*" wird im Hauptprogramm innerhalb des "Define" der Unitnamen angegeben, von der ab jetzt die Defines eingelesen werden. Ab "*Interface*" wird wieder ins Hauptprogramm zurückgeschaltet.

10.2 PreCompiled Units

Die Profiversion kann vorcompilierte (precompiled) Units und auch Include Files erstellen.
Die Funktionalität ist ähnlich die eines Linkers, aber nicht ganz identisch dazu.

Die Units werden precompiliert und sind daher für Dritte nicht mehr lesbar. In dem compilierten Projekt kann man allerdings den daraus generierten Assembler Quelltext dieser Units noch lesen. Normalerweise sollte das jedoch kein Problem sein, da der generierte Assembler Code immer im Assembler Fenster aller Simulatoren und Debugger sichtbar ist.

In der IDE PED32 in dem Menu "Project/Project options" gibt es eine Checkbox "Precompile Units". Wenn diese Box aktiv (checked) ist, werden alle Units und Includes des Projekts precompiliert und in Files mit der Extension "filename.PCU" abgelegt.

Diese Files können dann an Dritte ohne Sourcen weitergegeben werden.

11 Assembler

11.1 Allgemeines

Der zum System gehörige Assembler kann auch als separates Programm benutzt werden. Dabei ist jedoch zu beachten, dass zwar alle Prozessor Mnemonics beherrscht werden, jedoch keine Makros und ähnliche Funktionen komplexer Assembler.

In erster Linie dient der Assembler jedoch zum assemblieren des vom Compiler generierten Programm-Codes, als auch für in “ASM”-Anweisungen im Pascal-Source stehenden Assembler-Source Zeilen.

Der Aufbau und die Bedeutung der Mnemonics ist den jeweiligen Prozessor Handbüchern zu entnehmen. Bei manchen Prozessoren kann es allerdings vorkommen, dass bestimmte Memnonics, vor allem Registernamen in Konflikt mit evtl. Variablen-Namen des Programms kommen können.

ASM: *mnemonic* ;ein einzelnes Assembler Statement im HLL Text.

11.1.1 ASM;

Beginn eines Assembler Textes

Ein Programm für Embedded Control kommt sehr oft ohne Assembler Code nicht aus, da entweder der vom Compiler generierte Code für manche Operationen zu langsam ist, oder bestimmte Assembler Befehle ausgeführt werden müssen, die der Compiler nicht kennt oder benutzt.

Es besteht deshalb die Möglichkeit direkt Assemblersource an fast jeder beliebigen Stelle der Pascal source einzubinden. Diese source wird vom Compiler ungeprüft und unbearbeitet an den Assembler weitergegeben. Da der Compiler ebenfalls Assembler Code generiert, fügt sich der Assembler Text nahtlos ein.

Asm-Syntaxfehler werden deshalb nur vom Assembler erkannt und nicht vom Compiler.

Innerhalb des Assembler Textes kann auf **alle** deklarierten Variablen und Konstante zugegriffen werden.

Achtung:
Labels innerhalb eines Assembler-Blocks müssen am Zeilenanfang beginnen und mit einem ‘**:**’ abgeschlossen werden. Ausserdem darf diese Zeile keine weiteren Anweisungen wie z.B. Code enthalten. Die Analyse der vom Compiler generierten Assembler Dateien ‘xxx.ASM’ kann hier weiterhelfen.

```
ASM;
  LDI       _ACCA, 67;
  STS       a, _ACCA;           {a = Pascal var }
ENDASM;
```

11.1.2 ENDASM;

Ende eines Assembler Textes

ASM und ENDASM sind Pascal Statements und müssen deshalb mit einem Strichpunkt abgeschlossen werden.
Ausnahme: Einzel ASM Anweisung mit ***ASM:***

11.2 Assembler - Schlüsselwörter

11.2.1 Register

Beim AVR wurden die Registernamen ‘R0’.. ‘R31’ in den Memnonics _ACCA, _ACCB etc. genannt.
Es ist jedoch auch möglich innerhalb eines Assembler Statements die Bezeichner R0, R1 etc. zu benutzen.

Zuordnung von Registern zu den Pseudo ACCUs:

R0	=	_ACCGLO	Arithmetic reg and Flash access
R1	=	_ACCGHI	only with Imports of 32bit Typen and Floats
R2	=	_ACCHLO	only with Imports of 32bit Typen and Floats
R3	=	_ACCHHI	only with Imports of 32bit Typen and Floats
R4, R5	=	$_CURPROCESS	no User Access if defined by the system
R6, R7	=	$_CURTASK	no User Access if defined by the system
R8, R9	=	$_SAVERET	no User Access if defined by the system
R10	=	FLAGS	
R11	=	FLAGS2	
R12	=	_SYSTFLAGS	no User Access if defined by the system
R13	=		not used and not defined
R14	=		not used and not defined
R15	=		not used and not defined
R16	=	_ACCB	main working register low byte (16bit and 32 bit types)
R17	=	_ACCA	main working register 8bit types hi byte (16bit types) hi byte of low word (32bit and floats)
R18	=	_ACCALO	main working register Lo byte of hi word (32bit and float)
R19	=	_ACCAHI	main working register Hi byte of hi word (32bit and float)
R20	=	_ACCDLO	Arithmetik reg
R21	=	_ACCDHI	Arithmetik reg
R22	=	_ACCELO	Arithmetik reg
R23	=	_ACCEHI	Arithmetik reg
R24	=	_ACCFLO	Arithmetik reg
R25	=	_ACCFHI	Arithmetik reg
R26	=	_ACCBLO	Arithmetik reg
R27	=	_ACCBHI	Arithmetik reg
R26, R27	=	X-register	second pointer reg
R28, R29	=	_FRAMEPTR	
R30	=	_ACCCLO	Arithmetik reg
R31	=	_ACCCHI	Arithmetik reg
R30, R31	=	Z-register	main pointer reg

Alle oben aufgeführten Register _ACCxx, ausgenommen diejenigen, die nur bei 32bit typen Vorhanden sind, können innerhalb eines Assemblerteils immer benutzt werden. Die bei 32bit Imports definierten Register (_ACCGHI, _ACCHHI, _ACCHLO) können nur benutzt werden, wenn in der Import Anweisung auch 32bit Typen (LongInt, LongWord oder Float) importiert wurden.

Alle anderen Register, denen der Compiler einen Namen vergibt ($_CURPROCESS, FRAMEPTR etc) dürfen allerhöchstens gelesen werden, aber nie geschrieben. Ein Systemabsturz wäre sonst vorprogrammiert.

Eine Ausnahme bilden die drei Registerpaare X, Y und Z. Diese Mnemonics bzw. Register Namen müssen bei Pointer Operationen benutzt werden:

```
LDI    _ACCCLO, 00h;
LDI    _ACCCHI, 10h;        load _ACCCLO/HI = Z-reg with 1000h
LD     _ACCA, Z+;           load _ACCA with contents of RAM loc 1000h
```

Für eine bessere Lesbarkeit wurde den Registern R28..R30 einen weiteren Namen zugeordnet:

R26 -> XL *LDI XL, 45h*
R27 -> XH
R28 -> YL
R29 -> YH
R30 -> ZL
R31 -> ZH *LDI ZH, 01*

Labels
innerhalb eines Assembler Blocks müssen am **Zeilenanfang** beginnen und mit einem Doppelpunkt **:** abgeschlossen werden. In dieser Zeile dürfen **keine** weiteren Assembler Anweisungen oder Code stehen. Alle anderen Zeilen, ausgenommen Kommentare und Bezeichner, müssen mit mindestens einem Leerzeichen beginnen.

Namen
für Konstante oder ORG-Anweisungen müssen am **Zeilenanfang** beginnen und dürfen keinen Doppelpunkt haben. Die zugehörige Assembler Anweisung muss in der gleichen Zeile stehen.

11.2.2 Assembler Anweisungen

.ORG addr
setzt den internen Programm-Counter des Assembler auf die Adresse “addr‘. Darf vom Anwendungsprogramm eigentlich nie verwendet werden, ist für den Compiler reserviert

name .EQU nn
Konstanten Definition. Kann benutzt werden. Besser ist es, Konstanten im Pascal Code zu definieren. Dann sind sie sowohl im Assembler als auch in Pascal zugreifbar.

.BYTE .WORD .ASCII
Konstante im Flash. Darf im Assembler zumindest beim Mega nicht verwendet werden, da dessen Konstante grundsätzlich in der zweiten 64k-Page liegen müssen.

.END
Ende des Assembler Textes. Wird vom Compiler bestimmt.

ADDI ADCI
Der AVR kennt die Befehle "ADDI" und "ADCI" nicht. Diese können jedoch mit einem negierten Argument und dem SUBI/SBCI nachgebildet werden. Allerdings stimmt dann das Carry-Flag nicht mehr.
ADDI und ADCI wurden als Pseudo-Ops im Assembler integriert.

ADDI R16, nn
ADCI R16, nn

ADIW SBIW

Die Mnemonics "ADIW" und "SBIW" beziehen sich in der original Atmel Schreibweise auf R26/X R28/Y und R30/Z. Um eine bessere Lesbarkeit zu erzielen, wurde der Assembler um die entsprechenden Befehle erweitert:

```
ADIW  X, nn
ADIW  Y, nn
ADIW  Z, nn
SBIW  X, nn
SBIW  Y, nn
SBIW  Z, nn
```

SYSTEM.VectTab

Für JUMPs und CALLs an den Programm Anfang (ResetVector 0) können Assembler Statements wie z.B.

```
JMP  $0000
CALL $0000
```

verwendet werden.

Diese absolute Adressierung ist nur im Assembler zulässig und da auch nur beim mega16/32/, ... Kleinere CPUs unterstützen keine absoluten CALL/JMPs.
Um auch hier und in Pascal den Resetvector 0 referenzieren zu können, gibt es das Sprungziel "SYSTEM.VectTab".

```
RJMP  SYSTEM.VectTab
RCALL SYSTEM.VectTab
JMP   SYSTEM.VectTab
CALL  SYSTEM.VectTab
```

11.2.3 Operatoren für Konstanten Manipulation

NOT Invertiert das nachfolgende Argument.
LDI _ACCA, NOT 0FFh; lädt _ACCA mit 00h

AND logisches AND zwischen zwei Parametern.
LDI _ACCA, 0FFh AND 0F0h; lädt _ACCA mit 0F0h

SHRB Schiebe Operation mit einem Byte Ergebnis :
LDI _ACCA, 0FF00h SHRB 8; lädt _ACCA mit 0FFh

SHLB Schiebe Operation mit einem Byte Ergebnis.
LDI _ACCA, 0FFh SHLB 4; lädt _ACCA mit 0F0h

RORB Rotieren mit einem Byte Ergebnis.
LDI _ACCA, 0A5h RORB 4; lädt _ACCA mit 05Ah

ROLB Rotieren mit einem Byte Ergebnis.
LDI _ACCA, 081h ROLB 1; lädt _ACCA mit 03h

Der Assembler akzeptiert auch Character Konstante.

```
LDI   _ACCA, 'z';
```

11.2.4 Zugriff auf Pascal Konstante und Variablen

Wie schon o.a. kann innerhalb von Assembler Statements auf alle deklarierten Pascal Konstanten und Variablen Zugegriffen werden. Hierbei ist normalerweise immer der ***Modulname*** dem Variablen Namen voranzustellen. Bei Zugriffen auf Words und Longs ist zu beachten, dass der AVR nach dem “Little Endian” Prinzip arbeitet. D.h. das loByte eines Wortes liegt auf der niederwertigen Adresse, das hiByte auf der höherwertigen Adresse.

```
var
        bb   : byte;
        ww   : word;

begin
    ASM;
        LDS             _ACCA, module.bb      ; load byte bb to _ACCA
        ; load address of var ww to pointer reg Z
        LDI             _ACCCLO, module.ww AND 0FFh;
        LDI             _ACCCHI, module.ww SHRB 8;
        ;
        ; the Z-reg contains now the adr of var ww
        ; store a 0FFh to loByte of ww
        LDI             _ACCA, 0FFh;
        ST              Z, _ACCA;
        ;
        ; store a 00h to hiByte of ww
        LDI             _ACCA, 00h;
        STD             Z+1, _ACCA;
        ;
        ; move hiByte of ww to bb
        LDS             _ACCA, module.ww + 1;
        STS             module.bb, _ACCA;
    ENDASM;
End;
```

Der normalerweise immer notwendige ***Modulname*** kann auch durch **%** ersetzt werden.
Beispiel:

```
        Unit ABC;

        var xyz : byte;
        ...
        ASM
          LDS   _ACCA, ABC.xyz;
          ; or
          LDS   _ACCA, %.xyz
;
```

11.3 Einbindung von Assembler Routinen

11.3.1 Lokale Variable und Assembler Zugriffe

Lokale Variablen in Prozeduren und Funktionen, auch deren Übergabe Parameter, werden auf einem sog. Frame temporär gehalten. Zuständig für die Adressierung dieses Frames ist der FramePointer (Y-Register). Da nur über den FramePointer zugegriffen werden kann, ist ein Zugriff innerhalb eines Assembler Blocks nicht über die Namen der Parameter möglich, sondern nur FramePointer relativ.

Der FramePointer zeigt immer auf die zuletzt definierte Variable/Parameter. Den Offset der davor- bzw. darüberliegende Parameter muss der Assembler Programmierer selbst errechnen. Dabei muss er sowohl die Reihenfolge der Definition als auch den Speicherbedarf der einzelnen Typen beachten.

Eine falsche Offset Berechnung führt zumindest bei einem Schreibzugriff zu einem System Crash.

```
Procedure LocTest (var x : byte; bb : byte; ww : word; pt : pointer);
var
    Lbb    : byte;
    Lww    : word;
    Lpt    : pointer;

begin
    ASM;
        ; Access to local pointer Lpt
        ; copy Lpt to Z-reg
        LD      _ACCCLO, Y                  ; lo byte of Lpt
        LDD     _ACCCHI, Y+1                ; Y-reg offset 1
        ;
        ; Access to local word Lww
        ; copy ww to _ACCA, _ACCB
        LDD     _ACCB, Y+2                  ; lo byte of Lww
        LDD     _ACCA, Y+3                  ; hi byte of Lww
        ;
        ; Access to local byte Lbb
        ; copy lobyte of Lww to Lbb
        LDD     _ACCA, Y+2                  ; lo byte of Lww
        STD     Y+4, _ACCA                  ; byte Lbb
        ;
        ; Access to parameter pointer pt
        ; copy pt to Z-reg
        LDD     _ACCCLO, Y+5                        ; lo byte of pt
        LDD     _ACCCHI, Y+6                ; hi byte of pt
        ;
        ; Access to parameter word ww
        ; copy ww to _ACCA, _ACCB
        LDD     _ACCB, Y+7                  ; lo byte of ww
        LDD     _ACCA, Y+8                  ; hi byte of ww
        ;
        ; Access to parameter byte bb
        ; copy lobyte of ww to bb
        LDD     _ACCA, Y+7                  ; lo byte of ww
        STD     Y+9, _ACCA                  ; byte bb
        ;
        ; Access to parameter byte x
        ; mov a 00h to x
        ; remember that a var parameter is passed by it's address
        ; and not by it's value. So this param is always a pointer !!!
        LDD     _ACCCLO, Y+10               ; lo byte of adr of x
        LDD     _ACCCHI, Y+11               ; hi byte of adr of x
        LDI     _ACCA, 00h                  ; load a zero
        ST      Z, _ACCA                    ; store it with Z-Pointer reg to x
        ;
    ENDASM;
end;
```

Beachten Sie, dass eine Adressierung mit den Pointer Registern des AVR, in diesem Fall das Y-Register (FramePointer) nur im Bereich von 0..+63 möglich ist. Ist der Frame inkl. Parameter grösser als 63 Bytes, und ist der Offset eines Parameters/Var grösser als 63, so muss der FramePointer Y selbst in ein anderes Pointer Register geladen werden und der Offset hinzu addiert werden. Keinesfalls das Y-Register manipulieren !!

11.3.2 Prozedur Aufrufe und System Funktionen

Innerhalb eines Assembler Blocks können Pascal Prozeduren/Funktionen und auch System Funktionen aufgerufen werden. Hierbei ist zu beachten, dass die erforderlichen Sprung- und Call Labels qualifiziert werden müssen. Bei System Funktionen (ser. Schnittstelle etc.) muss die Qualifizierung “SYSTEM” vorangestellt werden. Bei User Funktionen im Hauptprogramm der Programm Name (nicht Dateiname) und bei Funktionen in Units der Unitname (nicht Dateiname).

```
Program CallTest;
...

Procedure isCalled;

begin
  ...
end;

ASM;
  LDI       _ACCA, 2Ah;          " * "
  RCALL     SYSTEM.SEROUT;
  RCALL     CallTest.isCalled;
ENDASM;
```

11.3.3 Funktions Ergebnisse und Assembler

Die Ergebnisse einer Funktion (Result) werden immer in Registern an die aufrufende Stelle zurück gegeben.

8bit Results (Byte, Char, Boolean etc)
Register _ACCA

16bit Results (Word, Integer, Pointer etc)
Register _ACCA (hiByte) _ACCB (loByte)

32bit Results (LongWord, LongInt, Float etc)
Register _ACCA (hiByte, loWord) _ACCB (loByte, loWord)
Register _ACCAHI (hiByte, hiWord) _ACCALO (loByte, hiWord)

11.3.4 Funktions/Prozedur Abschluss

Der Compiler generiert häufig, abhängig von der Konstruktion der Prozedur oder Funktion, einen bestimmten Code an der Stelle des Pascal End-Statements. Aus diesem Grund sollte keine *RET* oder *RETI* Anweisung programmiert werden. Es kann sonst zu einem Fehlverhalten der Prozedur kommen.
Besser ist es, ein *ASM-Label* vor das End-Statement zu setzen und dann mit einem *RJMP* dorthin zu springen.

11.3.5 Interrupt Prozeduren mit Assembler

Der Compiler sichert bei Interrupts grundsätzlich immer die Register *_ACCA*, *_ACCB*, *_ACCCLO* und *_ACCCHI*.

Sind Prozesse oder Tasks implementiert, werden per Default bei Interrupts vor dem eigentlichen Aufruf der Interrupt Prozedur alle Register gerettet. Man kann in diesem Fall durch den Compiler Schalter {$*NOSAVE*} unmittelbar vor der Interrupt Prozedur für diese die Komplettsicherung abschalten. Mit {$*NOREGSAVE*} wird überhaupt kein Register mehr gesichert, hier muss der Programmierer selbst für alles notwendige sorgen.

Sind keine Prozesse oder Tasks importiert, kann mit dem globalen Compilerschalter {$*NOSHADOW*} die Komplettsicherung abgeschaltet werden. Alternativ kann aber auch gezielt mit {$*NOSAVE*} die Sicherung auf die o.a. 4 Accus reduziert werden.

Bei Komplettsicherung der Register kann jeder ACCU auch im Assembler benutzt werden. Ansonsten nur die ACCUs *_ACCA*, *_ACCB*, *_ACCCLO* und *_ACCCHI*. Werden im letzteren Fall weitere ACCUs gebraucht, so müssen diese durch *PUSH* und *POP* gerettet und wieder hergestellt werden.

Eine Komplettsicherung benötigt ca. 20 Bytes im RAM, und zwar statisch, nicht auf dem Stack oder Frame. Dadurch ist ein verschachtelter Interrupt nicht möglich, d.h. innerhalb einer Interrupt Prozedur darf niemals der Interrupt freigegeben werden. Dies führt die CPU selbst aus, wenn sie den *RETI* Befehl abarbeitet.

Bei der Sicherung der 4 ACCUS (Stack) wäre eine Interrupt Freigabe denkbar, es wird aber heftigst davon abgeraten. Die Ergebnisse können, abhängig von den Operationen innerhalb der Prozedur, katastrophal sein.

11.3.6 Konstante und Optimierung

Konstante, Variable und Systemfunktionen die nur innerhalb eines Assembler Teils angesprochen werden, werden normalerweise vom Optimierer entfernt. Der Compiler analysiert den Assembler Code nicht und hat daher keine Kenntnisse von einem Zugriff. Deshalb kann es zu Fehlern bei der Assemblierung kommen, da die referenzierten Label nicht existieren.

Um eine Weg-Optimierung solcher Konstrukte (i.A. Konstante) zu vermeiden, ist es daher notwendig zumindest einen HLL (Pascal) Zugriff auf diese Konstante zu machen.

11.4 Assembler Schalter

Der Assembler kennt mehrere Kommando-Zeilen Schalter, die beim Aufruf im Batch-Mode durch die IDE PED32 nach dem Namen des Source-Files mit angegeben werden können. Wegen der notwendigen weiteren Infos ist allerdings ein stand-alone Betrieb des Assemblers nicht möglich.

-R Assembler läuft ohne sich zu zeigen im Hintergrund
-H Hexfile Ausgabe mit der FileExtension ‘xxx.hex’
-L Listfile Ausgabe ein. FileExtension ‘xxx.lst’

11.5 Assembler Errors

Stellt der Assembler irgendwelche Fehler fest, so generiert er eine Fehlerdatei mit der FileExtension ‘xxx.err’. Diese Datei benutzt die IDE PED32 zum lokalisieren der Fehler und deren Anzeige.

Assembler Fehler sollten grundsätzlich keine auftreten, wenn keine Assembler Statements benutzt werden.

Notizen

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