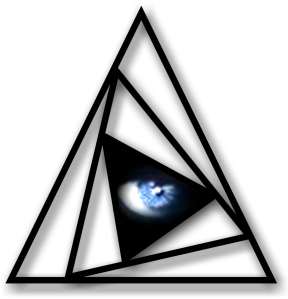
LunaAVR

Objektorientierte Programmiersprache
für AVR Mikrocontroller.

Version 2017.r1
http://avr.myluna.de



Sprachreferenz

Luna (AVR)

Dies ist die Dokumentation zur Programmiersprache **Luna** für "Atmel AVR" Mikrocontroller.

Luna ist eine objektorientierte, moderne Basic/Pascal-ähnliche Programmiersprache, deren Aufbau und Syntax sich an aktuellen Entwicklungswerkzeugen orientiert. Sie ist mit einer durchdachten und verständlichen Syntax ausgestattet, welche den Entwickler durch definierte symantische/sprachliche Rahmen bei der Vermeidung von Fehlern unterstützt (vergleichbar mit Modula gegenüber C). Sie bietet darüberhinaus komplexere Möglichkeiten auf technischer Augenhöhe mit Pascal und C/C++. Sie eignet sich damit für die effiziente und zeitsparende Entwicklung von kleinen bis großen, anspruchsvollen Softwareprojekten für AVR Mikrocontroller.

Luna besteht aus integrierter Entwicklungsumgebung (IDE), einem Präprozessor, Compiler und Assembler. Software kann wahlweise in der IDE oder in einem normalen Texteditor geschrieben werden. Die IDE bietet moderne Funktionen wie Syntaxfärbung, automatische Einrückung, Quelltext-Inspektor mit Auto-Vervollständigung, automatische Quelltextformatierung inklusive Ein- und Ausklappfunktion und vieles mehr. Zusätzlich sind ein Bibliothekseditor, Verzeichnis der Controller-Definitionen des Controller-Herstellers und zahlreiche weitere nützliche Werkzeuge vorhanden.

Der erzeugte ausführbare Binärcode ist in Ausführungsgeschwindigkeit und Größe vergleichbar mit leistungsfähigen existierenden Hochsprachen. Es gibt keine Beschränkung bei der Tiefe von Ausdrücken. Zudem bietet es eine hochoptimierte dynamische Speicherverwaltung inkl. dynamischer Strings, Speicherblöcke, Strukturen, Klassen, Datenobjekten, Datenstrukturen, sowie den Direktzugriff auf sämtliche Hardwareregister und -Funktionen des Controllers. Die AVR-spezifischen Code-Teile, inklusive der Bibliotheken sind im Quelltext zugänglich und vollständig in Assembler geschrieben.

Erste Schritte

Erste Schritte

Eine Zusammenstellung an Informationen für Neuanfänger mit Mikrocontrollern und/oder der Programmiersprache Luna.

Grundlegendes

- Was ist ein Mikrocontroller?

WAS WIRD BENÖTIGT?

- Ein ISP (In-System-Programmer) und
- Ein Programmier-Board mit einem Atmel-Mikrocontroller, oder
- Board & ISP als Kombination, z.B. **"Starterkit-Luna" von eHaJo**
  Siehe auch: Artikel ISP/Boards (mikrocontroller.net)

Tutorial

- Tutorial auf Basis des "Starterkit-Luna"

Start mit Luna

INSTALLATION

1. Aktuelles Archiv der Luna-AVR-Entwicklungsumgebung herunterladen.
2. Archiv entpacken und den Ordner mitsamt aller Dateien an einen beliebigen Ort kopieren. Die Luna-Entwicklungsumgebung kann auch von einem USB-Stick gestartet werden. Eine tiefgreifende Installation im System ist nicht notwendig, daher gibt es auch kein Installationsprogramm. Mit dem auspacken und kopieren an einen gewünschten Ort, ist die Installation abgeschlossen. Auf Wunsch kann man sich noch eine Verknüpfung auf den Desktop ziehen (bei Windows: das Programm "LunaAVR" mit der Maus festhalten und mit gleichzeitig gedrückt gehaltener "ALT"-Taste auf den Desktop fallen lassen).

ERSTER START & EINSTELLUNGEN

Nach dem Start des Programms "LunaAVR" zuerst den Programmer/Uploader einstellen.

Weitere Einstellungen sind zum grundlegenden Start nicht notwendig, man kann prinzipiell sofort losprogrammieren.

Beispiele (Allgemein)

Ein Luna-Programm benötigt minimal drei Angaben:

1. Definition des verwendeten Mikrocontrollers ("device")
2. Definition der Taktrate ("clock")
3. Definition der Stackgröße ("stack") - Was ist ein Stack?

Siehe hierzu: Basisklasse Avr

Ein leeres, korrektes Programm sähe demnach beispielsweise so aus:

```
avr.device = atmega168
avr.clock  = 8000000
avr.stack  = 32

halt()
```

Ein weiteres einfaches Programm, eine LED an PortB.0 (Pin 14) blinkt:

```
avr.device = atmega168
avr.clock  = 8000000
avr.stack  = 32

#define LED1 as portB.0    'LED1 wird als Bezeichner für portB.0 definiert
LED1.mode = output,low     'Port als Ausgang definieren und auf Low setzen

do
  LED1 = 1                 'LED einschalten
  waitms 500               '500ms warten
  LED1 = 0                 'LED auschalten
```

```
  waitms 500              '500ms warten
  'alternativ fuer die Befehle oben koennte man auch folgendes benutzen
  'LED1.toggle            'Aktuellen Status des PortPins umschalten (toggle)
  'wait 1                 '1 Sekunde warten
loop
```

Sprachreferenz

Kategorien

Grundlagen

- Grundlagen
- Bezeichner
- Kommentare
- Ausdrücke
- Syntax
- Operatoren
- Literale
- Datentypen
- Casting

Speicherung, Datenstrukturen

- Konstanten
- Variablen
- Pointer
- Strukturen
- Mengen
- Arrays
- Objekte

Programmablauf

- Bedingungen
- Schleifen
- Interrupts
- Events
- Exceptions

Programmstruktur

- Methoden
- Klassen
- Präprozessor

Sonstiges

- Interfaces
- Inline-Assembler

GRUNDLAGEN

Luna ist eine objektorientierte Programmiersprache.

Sie unterstützt mit wenigen Einschränkungen:

- Vererbung
- Datenkapselung
- Polymorphie
- Methoden-Überladen
- Operator-Überladen

OpenBook: Objektorientierte Programmierung von Bernhard Lahres, Gregor Rayman

BEGRIFFE

In der Programmierung gibt es verschiedene Begriffe, deren Bedeutung im Allgemeinen geläufig sind. Die objektorientierte Programmierung besitzt zudem weiterführende Begriffe und Bezeichnungen. Oft verwendete Begriffe sind:

- Variable
- Methode
- Objekt
- Klasse

PROGRAMMSTRUKTUR

- Controllerdefinition
- Definitionen, Deklarationen
- Konfiguration, Initialisierungen
- **Hauptprogramm**
- Unterprogramme/Interrupts
- Benutzerdefinierte Klassen
- Datenobjekte

WICHTIG!

- **Ausführbarer Code wird in der textuellen Reihenfolge kodiert, sodass Klassen und Unterprogramme im Quelltext *nach* dem Hauptprogramm folgen müssen!**
- Der Präprozessor ermittelt in einem **Vorlauf** automatisch die enthaltenen Unterprogramme, Klassen und Datenobjekte, sodass eine extra Deklaration von Klassen oder Methoden *vor dem Hauptprogramm* ***nicht*** *notwendig* ist.

BEZEICHNER

Bezeichner sind Namen für Variablen, Objekte, Label, Unterprogramme oder Konstanten und müssen folgenden Regeln genügen:

1. Am Anfang muss ein Buchstabe stehen
2. Mindestlänge für einen Bezeichner ist 1 Zeichen
3. Ein Bezeichner darf aus Buchstaben, Zahlen und dem Unterstrich bestehen.
4. Sonderzeichen und Umlaute sind bis auf den Unterstrich nicht gestattet

Beispiele für erlaubte Bezeichner:

- a
- Horst
- MeinUnterProgramm
- hallo123
- m1_var

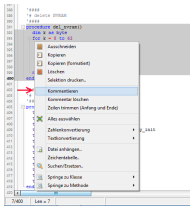
KOMMENTARE

Kommentare sind in Luna zeilenorientiert, beginnen mit dem Zeichen **'** oder durch **//** und enden am Ende der Zeile.

Beispiel für die Verwendung von Kommentaren

```
' Ich bin ein Kommentar
a = 100 ' ich bin auch ein Kommentar

// altbekannter Kommentar
a = 100 // Lange Rede gar kein Sinn!
```

Um mehrere Zeilen Code auf einmal als Kommentar zu makieren:



- die gewünschten Zeilen makieren
- Rechtsklick mit der Maus auf die makierten Zeilen
- aus dem erscheinenden Menü "Kommentieren" auswählen

Um den Kommentar von mehrern Zeilen wieder zu entfernen:

- die gewünschten Zeilen makieren
- Rechtsklick mit der Maus auf die makierten Zeilen
- aus dem erscheinenden Menü "Kommentar löschen" auswählen

Sollen umfangreichere Teile des Codes zeitweise vom Kompiliervorgang ausgenommen werden, kann dies mithilfe der Präprozessor-Befehle **#if..#endif** erfolgen.

Beispiel:

```
  #if 0
     ' Programmcode wird nicht übersetzt.
  #endif
```

AUSDRÜCKE

Ausdrücke sind arithmetische oder logische Konstrukte, welche gemäß der Semantik/Syntax in Bezug zu einem Kontext einen Wert liefern und ausgewertet werden. Einzelne arithmetische bzw. logische Ausdrücke oder auch Kombinationen daraus, sowie Ausdrücke mit Zeichenketten und deren Funktionen werden insgesamt unter dem Oberbegriff "Ausdruck" zusammengefasst.

LÄNGE VON AUSDRÜCKEN

Die Länge von Ausdrücken ist nicht beschränkt. Der Compiler optimiert komplexe Ausdrücke. Das Aufteilen auf einzelne Operationen (wie z.B. bei Bascom zwangsweise üblig) führt in den meisten Fällen zu einem höheren Speicherbedarf und langsamerer Ausführung.

SCHLÜSSELWÖRTER

true	Repräsentiert den Wert "wahr", arithmetisch <> *0* (ungleich Null). Verwendet in Bedingungen und in boolschen Ausdrücken.
false	Repräsentiert den Wert "falsch", arithmetisch = *0* (gleich Null). Verwendet in Bedingungen und in boolschen Ausdrücken.
nil	Repräsentiert den Wert "nicht existent" (= nicht erzeugt/belegt oder instanziert). Eine Zuweisung zu einem Objekt entfernt es aus dem Speicher (zerstört es).
new	Operator zur Erzeugung von Objekten.

BEISPIELE FÜR AUSDRÜCKE

- **Arithmetische Ausdrücke**
  - *5 * 7*
  - *(13 * (x - a) / b) << 5*
  - *x * 9 = 2 * y*
  - *z = c > 100*
  - *y = Berechnung(x, 23, 42)*
  - usw.

- **Logische Ausdrücke**
  - *a and b*
  - *b or a*
  - *(a or not b) and c*
  - usw.

- **Ausdrücke mit Zeichenketten**
  - *"Hallo"+**str**(22)*
  - *m.**StringValue**(2,3)+":"*
  - ***right**("00"+stunde,2)+":"+**right**("00"+minute,2)+":"+**right**("00"+sekunde,2)*
  - usw.

- **Ausdrücke mit Objekten**
  - *m = **new** MemoryBlock(123)*
  - *m = **nil***
  - *a = m.ByteValue(23) + 100 * 2*
  - usw.

DIE SYNTAX IN LUNAAVR

Die Syntax, also die in Luna definierte Semantik von Befehlen und/oder Ausdrücken ist an derzeitige objektorientierte Entwicklungswerkzeuge angelehnt. [1)]

VARIANTEN

Variablen

- *Variable = Ausdruck*
- *Variable = Methode(Ausdruck1, Ausdruck2)*
- *Variable = Objekt.Methode(Ausdruck)*
- *Variable = Objekt.Objekt.Eigenschaft*
- usw.

Methoden

- *Methode(Ausdruck)*
- *Call Methode(Ausdruck)*
- usw.

Bedingungen

- *If Ausdruck Then [..] ElseIf Ausdruck Then [..] Else [..] EndIf*
- *Select Variable, Case Konstante [..] CaseElse [..] EndSelect*
- *While Ausdruck [..] Wend*
- *Do [..] Loop Until Ausdruck*
- usw.

Objekte

- *Klasse.Objekt.Eigenschaft = Ausdruck*
- *Objekt.Eigenschaft = Ausdruck*
- *Objekt.Methode(Ausdruck)*
- *Objekt.Objekt.Eigenschaft = Ausdruck*
- usw.

Deklarationen/Dimensionierungen

- *Dim Bezeichner[, BezeichnerN] as Datatype*
- *Const Bezeichner = Konstante*
- usw.

Objektbasierte Strukturen

- *data Bezeichner [..] EndData*
- *eeprom Bezeichner [..] EndEeprom*
- usw.

1) Luna verzichtet als Strukturierungszeichen dabei absichtlich auf die geschweiften Klammern.

OPERATOREN

ALLGEMEIN

- Arithmetische Operatoren
- Logische Operatoren und Bitoperatoren
- Rangfolge der Operatoren

ZUWEISUNGS-OPERATOREN

- Eigen-Operation arithmetisch
- Eigen-Operation bitorientiert

LITERALE

Literale sind in Luna für Binärwerte und Hexadezimale Werte definiert. Für die Literaltypen verwendet man folgende Schreibweisen:

Binär: *0b* - wie in C & Assembler (Standard), auch möglich: *&b*

Das Beispiel zeigt einen 8 Bit Binärwert:

```
0b10111010
```

Hexadezimal: *0x* - wie in C & Assembler (Standard), auch möglich: *&h*

Das Beispiel zeigt einen 16 Bit Hexadezimalwert:

```
0x65b7
```

Hinweis: Binär- und Hexadezimaldarstellung, ob Ein- oder Ausgabe sind in *Big-Endian-Schreibweise* standardisiert.

DATENTYPEN

Luna kennt eine Auswahl an numerischen Standard-Datentypen sowie Sonderformen von Datentypen.

ÜBERSICHT DATENTYPEN

Name	kleinster Wert	größter Wert	Typ	Größe
Boolean	false[1)]	true[2)]	Variable	1 Byte
Byte	0	255	Variable	1 Byte
Int8	-128	127	Variable	1 Byte
UInt8	0	255	Variable	1 Byte
Integer	-32768	32767	Variable	2 Byte
Int16	-32768	32767	Variable	2 Byte
Word	0	65535	Variable	2 Byte
UInt16	0	65535	Variable	2 Byte
Int24	-8.388.608	8.388.607	Variable	3 Byte
UInt24	0	16.777.215	Variable	3 Byte
Long	0	4.294.967.295	Variable	4 Byte
UInt32	0	4.294.967.295	Variable	4 Byte
LongInt	-2.147.483.648	2.147.483.647	Variable	4 Byte
Int32	-2.147.483.648	2.147.483.647	Variable	4 Byte
Single	–3,402823E38	+3,402823E38	Variable	4 Byte
String	0 Zeichen	254 Zeichen	Objekt-Variable	2 Byte
MemoryBlock	nil	MemoryBlock	Objekt-Variable	2 Byte
sPtr	0	65.535	Pointer (Sram)	2 Byte
ePtr	0	65.535	Pointer (Eeprom)	2 Byte
dPtr	0	16.777.215	Pointer (Flash)	3 Byte
Benutzerdefiniert	?	?	Objekt-Variable	?

Siehe auch: Arbeitsspeicher-Variable

BENUTZERDEFINIERTE DATENTYPEN

Es können Strukturen als benutzerdefinierte Datentypen deklariert werden.
Siehe hierzu: Struct-EndStruct

INTRINSISCHE DATENTYPEN

Intrinsisch bedeutet, dass der Wert den diese Variablen speichern als eine Adresse auf einen Speicherbereich interpretiert wird. Pointer, Stringvariablen und MemoryBlocks sind daher intrinsische Variablen.

STRING

Ein String ist eine Kette von beliebigen Zeichen. Jede Art von alphabetischer oder numerischer Informationen kann als Zeichenfolge gespeichert werden. "Heinz Ehrhardt", "13.11.1981", "23.42" sind Beispiele für Strings. In LunaAVR können Strings auch *binäre Daten* aufnehmen, z.Bsp. *Nullbytes*.

Im Sourcecode werden Zeichenketten in Anführungsstriche eingebettet. Die maximale Länge eines Strings in LunaAVR beträgt 254 Bytes. Der Standardwert eines Strings ist "" (Leerstring). LunaAVR legt Strings im Pascal-Format, eingebettet in ein MemoryBlock-Objekt ab.

Stringvariablen belegen im Arbeitsspeicher mindestens 2 Bytes (Zeiger auf MemoryBlock-Objekt). Eeprom-Strings jedoch die entsprechend statisch angegebene Anzahl Bytes im Eeprom. Ein als Arbeitsspeicher-Variable deklarierter String ist ein 16-Bit-Pointer auf einen MemoryBlock mit dynamischer Größe.

Stringkonstanten werden im Programmsegement angelegt und belegen dort die Anzahl Bytes der Zeichenkette zuzüglich eines Längenbytes (Pascal-String).

MEMORYBLOCK

Der Datentyp MemoryBlock ist eine *direkte* Objektreferenz auf ein MemoryBlock-Objekt und ermöglicht dadurch den direkten Zugriff über dessen Methoden und Eigenschaften.

Beispiel einer Variablendeklaration mit verschiedenen Datentypen im Arbeitsspeicher:

```
dim a as byte
dim b,e,f as integer
dim var as single
dim meinText as String
dim m as MemoryBlock

a=1
b=12345
var=0.44234
meinText="Dr. Who"
m = New MemoryBlock(100)
```

POINTER

Pointern kommt eine besondere Bedeutung zu: Sie können wie Word-Variablen mit Werten belegt werden und man kann mit ihnen Rechnungen durchführen. Weiterhin sind die Objektfunktionen des MemoryBlocks darauf anwendbar, also beispielsweise p.ByteValue(0) = 0x77 usw.

Da der Controller über drei unterschiedliche Speichersegmente verfügt, wurden auch drei verschiedene Pointer-Typen als neuer Datentyp implementiert. Dies sind:

- **sptr:** Pointer auf Adressen im Arbeitsspeicher (sram segment)
- **dptr:** Pointer auf Adressen im Flash (data segment)
- **eptr:** Pointer auf Adressen im Eeprom (eeprom segment)

Mit Pointern ist es also möglich, auf eine beliebige Adresse, z.Bsp. innerhalb eines MemoryBlocks oder in einer Flash-Tabelle mit den Objektfunktionen zuzugreifen.

1) false = *0*

2) true = *<>0*

TYPKONVERTIERUNG (CASTING)

Explizites konvertieren bzw. festlegen eines Wertes oder Ausdruckergebnisses in/auf einen bestimmten Datentyp. Dies ist manchmal sinnvoll, wenn eine bestimmte Funktion anhand des verwendeten Datentyps eine entsprechend angepasste Funktonalität aufweist. Einige Ausgabefunktionen passen z.T. die Art der Ausgabe an den übergebenen Wert an.

Beispielsweise erfolgen mathematische Berechnungen oder Bitmanipulationen im nächst größeren Datentyp, wenn der aktuelle Datentyp das Ergebnis möglicherweise nicht aufnehmen kann (Datentypen kleiner als Long).

Beispiel 1 zur Veranschaulichung:

```
dim a,b as byte
print hex(a+b)  ' Das Ergebnis ist vom Datentyp word,
                ' die Hex-Ausgabefunktion gibt daher einen word-Hexwert aus
```

In obigem Beispiel kann man nun mit einer expliziten Typkonvertierung festlegen von welchem Datentyp das Ergebnis sein soll.

FUNKTIONEN DER TYPKONVERTIERUNG

- ***byte****(Ausdruck)*
- ***int8****(Ausdruck)*
- ***uint8****(Ausdruck)*
- ***integer****(Ausdruck)*
- ***word****(Ausdruck)*
- ***int16****(Ausdruck)*
- ***uint16****(Ausdruck)*
- ***int24****(Ausdruck)*
- ***uint24****(Ausdruck)*
- ***long****(Ausdruck)*
- ***longint****(Ausdruck)*
- ***int32****(Ausdruck)*
- ***uint32****(Ausdruck)*
- ***single****(Ausdruck)*

Siehe auch:

- BcdEnc()
- BcdDec()
- CE16()
- CE32()

Beispiel 2:Beispiel 1 zur Veranschaulichung:

```
dim a,b as byte
print hex(byte(a+b))  ' Das Ergebnis ist vom Datentyp byte,
                      ' die Hex-Ausgabefunktion gibt daher einen byte-Hexwert aus
```

KONSTANTEN

Luna kennt normale numerische Konstanten, Zeichenketten und Konstanten-Objekte.

Normale Konstanten sind feste beliebige Zahlenwerte oder Zeichenketten die im Sourcecode Verwendung finden können. Konstanten können innerhalb einer Klasse nur einmal definiert werden und sind auch in Unterprogrammen sichtbar.

Eine Sonderform der Konstanten ist das Konstanten-Objekt (Datenstruktur).

Ausdrücke/Berechnungen mit ausschließlich Konstanten werden vom Präprozessor vorverarbeitet. Die Definition einer Konstanten durch einen mathematischen Ausdruck weist ihr also das *Ergebnis* der Berechnung zu. In mathematischen Funktionen innerhalb des Sourcecodes werden Konstanten auf ihren Wertebereich reduziert, d.h. die Konstante "100" wird wie eine Byte-Variable behandelt, "-100" dagegen wie ein Integer und "12.34" als ein Single.

HINWEIS

Um reine Konstantenberechnungen als Solche für den Präprozessor erkennbar zu machen, sollten sie in Klammern gesetzt werden wenn der Ausdruck auch Variablen o.Ä. enthält. So kann der Präprozessor sie besser differenzieren und unnötigen Binärcode vermeiden.

VORBELEGTE KONSTANTEN

Im Compiler sind folgende Konstanten mit einem festen Wert vorbelegt.

Name	Beschreibung	Typ
PI	Die Zahl PI (3.1415926)	Single
COMPILER_VERSION_STRING	Vollständige Zeichenkette der Compiler-Version.	string
COMPILER_MAJOR_NUMBER	Die Haupt-Version des Compilers.	integer
COMPILER_MINOR_NUMBER	Die Release-Version des Compilers.	integer
COMPILER_UPDATE_NUMBER	Die Release-Update-Version des Compilers.	integer
COMPILER_BUILD_NUMBER	Die Build-Nummer des Compilers.	integer

BEISPIEL

Definition und Verwendung von Konstanten:

```
Const zahl1 = 100
Const zahl2 = 33.891
Const zahl3 = (12 + 10) / 7       ' zahl3 = 3.14285
Const Text1 = "Ich bin ein Text"

dim x as single
x=zahl1+zahl2  ' Zuweisung des Wertes 133.891, Gleichbedeutend mit x = 133.891
print Text1    ' Ausgabe: "Ich bin ein Text"
```

VARIABLEN

Variablen sind im Sourcecode durch Bezeichner benannte Speicherzellen im Arbeitsspeicher (SRAM) oder Eeprom (ERAM). Ihnen können Werte zugewiesen werden, um sie an anderer Stelle auszulesen.

Ausführliche Beschreibung (Wikipedia)

DIMENSIONIERUNG

Variablen werden mit Dim (Arbeitsspeicher) und/oder eeDim (Eeprom) dimensioniert (erzeugt/angelegt).

METHODEN/EIGENSCHAFTEN

Variablen besitzen Objekteigenschaften:

Eigenschaften der *numerischen* Standard-Variablen (boolean, byte, word, integer, long, single, ..)			
Name	**Beschreibung**	**Typ**	**read/write**
.0-x[1)]	Bit n des Variablenwertes	boolean	read+write
.LowNibble	Low-Nibble des Variablenwertes	byte	read+write
.HighNibble	High-Nibble des Variablenwertes	byte	read+write
.Nibble***n***	Nibble ***n*** *(1-8)* des Variablenwertes	byte	read+write
.Byte***n***[2)]	Byte ***n*** *(1-4)* des Variablenwertes	byte	read+write
.LowByte[3)]	Low-Byte des Variablenwertes	byte	read+write
.HighByte[3)]	High-Byte des Variablenwertes	byte	read+write
.LowWord[4)]	Niederwertiges Word des Variablenwertes	word	read+write
.HighWord[4)]	Höherwertiges Word des Variablenwertes	word	read+write
.Reverse[5)]	Byte-Anordnung umdrehen, siehe auch CE16(), CE32()	word, long	read
.Addr[6)]	Adresse der Variable[7)]	word	read
.Ptr[8)]	Adresse des MemoryBlocks	word	read
.SizeOf	Vom Datentyp belegte Anzahl Bytes lesen.	byte	read

Methoden von Byte und Word-Arrays[9)]	
Name	**Beschreibung**
.Sort	Arraywerte aufsteigend sortieren (sehr schneller Qsort-Algorithmus)

BENUTZERDEFINIERTE DATENTYPEN

Benutzerdefinierte Datentypen mittels Struct erben die Methoden/Eigenschaften der jeweilig in der Struktur deklarierten Elemente.

STRING UND MEMORYBLOCK

Die Datentypen String und MemoryBlock besitzten weitere Methoden bzw. Eigenschaften.

VERWENDUNG

Variablen müssen dimensioniert werden, bevor man sie verwenden kann. Variablen speichern Werte entsprechend ihres Datentyps.

Siehe auch: Dim, Abschnitt "Sichtbarkeit von Variablen"

ZUGRIFF AUF EINZELNE BITS EINER VARIABLE

Wie in der obigen Tabelle ersichtlich, kann man auf einzelne Bits einer Variable zugreifen und diese lesen oder schreiben. Der Zugriff auf ein Bit ist auch mit einer weiteren Variable möglich, welche die Bit-Nummer enthält:

Beispiel:

- ***var*** = ***var1.var2***
- ***var1.var2*** = *Ausdruck*

Der Zugriff mit einem *konstanten Wert* ist in der Ausführungsgeschwindigkeit jedoch schneller:

Beispiel:

- ***var1*** = ***var.3***
- ***var.3*** = *Ausdruck*

BEISPIEL

```
dim a,b as byte
dim c as word
dim s as string

c=&ha1b2      ' Wert zuweisen, Hexadezimal (liegen rückwärts im Speicher da Hexwert immer Big-Endian)
a=c.LowByte   ' Low-Byte des Variablenwertes lesen, Ergebnis: &hb2
b=c.HighByte  ' High-Byte des Variablenwertes lesen, Ergebnis: &ha1
c.LowByte=b   ' vertauscht wieder hineinschreiben
c.HighByte=a

c.0=1         ' Bit 0 in c setzen
a=b.7         ' Bit 7 von b lesen
a=3
c=b.a         ' Bit 3 von b lesen, Bitnummer steht in a

Print "&h"+hex(c)   ' Ausgabe: &hb2a1
s = "Hallo"         ' Text zuweisen
a = s.ByteValue(1) ' Dezimalwert des 1. Zeichens vom im String gespeicherten Text lesen
```

Siehe auch

- Datentypen
- Klassen
- Dim
- eeDim
- String
- Swap
- MemoryBlock

1) Alle numerischen Datentypen

2) Nur long, single

3) Nur word, integer

4) Nur long, longint, single

5) Nur long, longint, integer, word

6) Alle numerischen Datentypen und Strukturen. Bei Arrays die Startadresse des Arrays

7) Adresse bezogen auf den Speicherbereich in dem die Variable liegt, Arbeitsspeicher oder Eeprom.

8) Die intrinsischen Datentypen string, memoryblock

9) Ab Version 2013.r1 build 4612

POINTER (SPTR, DPTR, EPTR)

Pointer sind spezielle intrinsische Variablen. Intrinsisch bedeutet, dass der Wert den sie speichern als eine Adresse auf einen Speicherbereich interpretiert wird. Stringvariablen und MemoryBlocks sind ebenfalls intrinsische Variablen.

Pointern kommt jedoch eine besondere Bedeutung zu: Sie können wie Word-Variablen mit Werten belegt werden und man kann mit ihnen Rechnungen durchführen. Weiterhin sind die Objektfunktionen des MemoryBlocks darauf anwendbar, also beispielsweise p.ByteValue(0) = 0x77 usw.

Da der Controller über drei unterschiedliche Speichersegmente verfügt, wurden auch drei verschiedene Pointer-Typen als neuer Datentyp implementiert. Dies sind:

- **sptr:** Pointer auf Adressen im Arbeitsspeicher (sram segment)
- **dptr:** Pointer auf Adressen im Flash (data segment)
- **eptr:** Pointer auf Adressen im Eeprom (eeprom segment)

Mit Pointern ist es also möglich, auf eine beliebige Adresse, z.Bsp. innerhalb eines MemoryBlocks oder in einer Flash-Tabelle mit den Objektfunktionen zuzugreifen:

Beispiel im Datensegment (Direktzugriff):

```
dim pd as dptr

pd = table.Addr
print "pd = 0x";hex(pd);" ";34;pd.StringValue(2,12);34

do
loop

data table
  .dw myfunc1
  .db "Menu Nr. 1  "
  .dw myfunc2
  .db "Menu Nr. 2  "
  .dw myfunc3
  .db "Menu Nr. 3  "
enddata
```

Das Ganze ist vergleichbar mit den Zugriffen über das Sram, Flash oder Eeprom-Interface, bei denen man im Unterschied zu den Pointern absolute Adressangaben vornehmen muss (also nicht zwangsläufig etwas super neues). Bei Pointern ist aber eine weitere Besonderheit vorhanden: **Die Möglichkeit zum SuperImposing mit Strukturen.**

SUPERIMPOSE

SuperImposing ist das "darüberlegen" einer Strukturdeklaration auf einen beliebigen Speicherbereich. D.h. wenn man bspw. eine Struktur in Form eines Menüelements deklariert hat, lässt sich mit Pointern diese Strukturdeklaration auf den Speicherbereich abbilden/maskieren und kann dann die Elemente der Struktur elegant lesen. Das SuperImposing/Maskieren von Strukturdeklarationen auf Speicherbereiche ist auf die Verwendung mit Pointern beschränkt.

SYNTAX

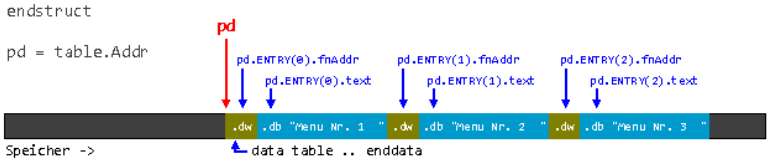
Die Syntax ist denkbar einfach, wobei jedoch bestimmte Vorgaben einzuhalten sind. Bei einer Strukturdeklaration handelt es sich um eine **virtuelle** Definition, welche dem Compiler mitteilt wie der Zugriff erfolgen soll:

mypointer.*[(Index)]*.

Ist das Strukturelement eine weitere Struktur, verschachtelt sich der Aufruf entsprechend tiefer.

Index ist ein optionaler **Konstantenwert** bzw. ab Version 2013.r5 **ein Ausdruck (z.B. Variable)**, der den Startoffset auf Basis der Gesamten Strukturgröße ergibt. D.h. ist die Struktur mitsamt seiner Elemente 4 Bytes groß, würde **Index = 2** zu einer Startadresse + 8 führen, ab der die Struktur über den Speicherbereich gelegt wird.

```
struct ENTRY
  word   fnAddr
  string text[12]
endstruct

pd = table.Addr
```



ANWENDUNG

Strukturdeklaration:

```
struct ENTRY
```

```
  word  fnAddr
  string text[12]
endstruct
```

Pointer für Flash-Segment dimensionieren:

```
dim pd as dptr
```

Pointer mit Adresse der Tabelle belegen:

```
pd = table.Addr
```

Indirekter Zugriff über Strukturdeklaration, die ab der im Pointerabgelegten Adresse über den Speicherbereich gelegt wird (superimpose).Bei Angabe eines Arrayindex bei einer Struktur, wird der Offset automatisch berechnet:

```
print "fnAddr = 0x";hex(pd.ENTRY(0).fnAddr);", menu text = ";34;pd.ENTRY(0).text;34
print "fnAddr = 0x";hex(pd.ENTRY(1).fnAddr);", menu text = ";34;pd.ENTRY(1).text;34
print "fnAddr = 0x";hex(pd.ENTRY(2).fnAddr);", menu text = ";34;pd.ENTRY(2).text;34
```

Die Tabelle im Flash:

```
data table
  .dw myfunc1
  .db "Menu Nr. 1  "
  .dw myfunc2
  .db "Menu Nr. 2  "
  .dw myfunc3
  .db "Menu Nr. 3  "
enddata
```

BEISPIELPROGRAMM

pointer.luna

```
const F_CPU=20000000
avr.device = atmega328p
avr.clock  = F_CPU
avr.stack = 64

uart.baud = 19200
uart.recv.enable
uart.send.enable

struct ENTRY
  word   fnAddr
  string text[12]
endstruct

dim a as byte
dim ps as sptr
dim pd as dptr
dim m as MemoryBlock

print 12;"pointer and superimpose example"
print

m.New(64)
m.WordValue(0) = 0xaabb
m.CString(2) = "Entry Nr. 1 "
m.WordValue(14) = 0xccdd
m.CString(16) = "Entry Nr. 2 "
m.WordValue(28) = 0xeeff
m.CString(30) = "Entry Nr. 3 "

ps = m.Ptr    'Speicheradresse der MemoryBlock-Daten
pd = table.Addr   'Speicheradresse der Tabelle im Flash

'direkt über Objektfunktionen
print "(sram)  ";34;ps.StringValue(2,12);34
print "(flash) ";34;pd.StringValue(2,12);34
print

'SuperImpose
'
'SuperImposing ist das "darüberlegen" einer Strukturdeklaration auf einen beliebigen Speicherbereich.
'D.h. wenn man bspw. eine Struktur in Form eines Menüelements deklariert hat, lässt sich mit Pointern
'diese Strukturdeklaration auf den Speicherbereich abbilden und kann dann die Elemente der
'Struktur elegant lesen. Bei Angabe eines Arrayindex (Konstante) bei einer Struktur, wird der
'Offset automatisch berechnet.

print "sram:"
```

```
print "fnAddr = 0x";hex(ps.ENTRY(0).fnAddr);", menu text = ";34;ps.ENTRY(0).text;34
print "fnAddr = 0x";hex(ps.ENTRY(1).fnAddr);", menu text = ";34;ps.ENTRY(1).text;34
print "fnAddr = 0x";hex(ps.ENTRY(2).fnAddr);", menu text = ";34;ps.ENTRY(2).text;34
print
print "flash:"
print "fnAddr = 0x";hex(pd.ENTRY(0).fnAddr);", menu text = ";34;pd.ENTRY(0).text;34
print "fnAddr = 0x";hex(pd.ENTRY(1).fnAddr);", menu text = ";34;pd.ENTRY(1).text;34
print "fnAddr = 0x";hex(pd.ENTRY(2).fnAddr);", menu text = ";34;pd.ENTRY(2).text;34
print

print "Aufruf per Icall aus der Tabelle:"
icall pd.ENTRY(0).fnAddr
icall pd.ENTRY(1).fnAddr
icall pd.ENTRY(2).fnAddr

print
print "ready"

do
loop

procedure myfunc1()
  print "myfunc 1"
endproc
procedure myfunc2()
  print "myfunc 2"
endproc
procedure myfunc3()
  print "myfunc 3"
endproc

'Wichtiger Hinweis:
'Adressen von Labels, Objekten oder Methoden liegen im Flash immer an einer Word-Grenze, sie werden
'im Flash-Objekt daher auch als Word-basierte Adresse abgelegt. Möchte man mit der Byte-Adresse arbeiten,
'muss man den Wert wie im Assembler mit 2 multiplizieren.
data table
  ENTRY { myfunc1, "Menu Nr. 1" }
  ENTRY { myfunc2, "Menu Nr. 2" }
  ENTRY { myfunc3, "Menu Nr. 3" }
enddata
```

STRUKTUREN

STRUCT-ENDSTRUCT

Mit Struct/EndStruct können benutzerdefinierte Strukturen angelegt werden.

Eine Struktur ist ein aus mehreren verschiedenen Datentypen bestehendes Speicher-Objekt. Es werden die in ihr definierten einzelnen Datentypen *strukturiert* unter einem einzelnen Namen zusammengefasst.

DEKLARATION

- ***Struct*** *Bezeichner*
  - *Datentyp Bezeichner*
  - *[..]*
- ***EndStruct***

SICHTBARKEIT

Deklarierte Strukturen sind innerhalb ihrer Klasse sichtbar. Dies bedeutet, eine im Hauptprogramm (Basisklasse "Avr") deklarierte Struktur ist innerhalb einer anderen Klasse unbekannt.

- **Bis Version 2014.r2.4:** Wenn Daten unter Nutzung einer Struktur *zwischen Klassen* getauscht werden sollen, muss die dafür verwendete Strukturdeklaration nochmalig in der jeweiligen Klasse vorgenommen werden. Die Deklarationen dürfen sich nicht unterscheiden, da es sonst zu Speicherverletzungen kommen kann.
- **Ab der Version 2015.r1** sind Struktur*deklarationen* auch über den Klassennamen verwendbar. D.h. ist in einer Klasse eine Struktur definiert, kann diese auch im Hauptprogramm zur Dimensionierung von Strukturen genutzt werden und umgekehrt. Die oben genannte Einschränkung bis Version 2014.r2.4 ist damit *aufgehoben*.

```
struct mystruct_t
  byte   value
  string text[20]
endstruct

dim var1 as mystruct_t          'the global structure type
dim var2 as myclass.mystruct_t  'the class-internal structure type

[...]

class myclass
  struct mystruct_t
    byte   value
    word   id
    string text[20]
  endstruct

  dim var1 as avr.mystruct_t  'the global structure type
  dim var2 as mystruct_t      'the class-internal structure type

  [...]
endclass
```

VERWENDUNG

Bevor man eine Struktur im Programm verwenden kann, muss man die Struktur deklarieren. Die Deklaration erfolgt mit Struct/EndStruct. Die *Deklaration* einer Struktur belegt *keinen* Speicherplatz. Der Speicherplatz wird erst alloziert, wenn man eine Variable als Struktur mit Dim in einem Speicherbereich (Arbeitsspeicher oder Eeprom) dimensioniert. Eine bereits deklarierte Struktur kann wiederum in einer weiteren Struktur implementiert werden.

Strukturen sind in in sich selbst *statisch*. Dies bedeutet, dass auch Strings immer eine feste Länge benötigen (wie bei der Dimensionierung im Eeprom). Strukturen belegen bei einer Dimensionierung dann so viele Bytes im Arbeitsspeicher oder Eeprom, wie alle in der Struktur vorhandenen Elemente an Speicherplatz zusammen benötigen.

SUPERIMPOSING

Die Strukturdeklarationen können auch für eine einfachere Ablage von Daten in Datenobjekten, oder als eine Art Maske für Speicherbereiche verwendet werden.
Siehe hierzu: Pointer, Absatz "SuperImpose"

STRINGS IN STRUKTUREN (DIMENSIONIERT)

Bei der Zuweisung von Strings/Zeichenketten an String-Elemente von *dimensionierten* Strukturen werden sie automatisch als Pascal-String gespeichert. D.h. das erste Zeichen im Speicher ist das Längenbyte. Dies bei der Größenangabe in Strukturelementen beachten, die im Arbeitsspeicher dimensioniert werden sollen.

STRUKTUR-ARRAYS

Strukturen können als *Array* dimensioniert werden, womit Felder mit mehreren Elementen je Array-Element möglich werden.

Beispiel1:

```
[..]
' Struktur deklarieren
struct point
  byte x
  byte y
endstruct

dim a(3) as point  ' Array mit 4 Elementen des Typs "point" dimensionieren

a(0).x = 1
a(0).y = 20
a(1).x = 23
a(1).y = 42
[..]
```

Beispiel2:

```
[..]
' Struktur deklarieren
struct buffer
  byte   var(7)
  string text[4](2)  ' String Array mit 3 Elementen
endstruct
' Struktur deklarieren
struct mydate
  byte    sec
  byte    min
  byte    hour
  byte    month
  word    year
  buffer  buf       ' verschachtelte Strukturen möglich
  string  text[12]  ' Stringspeicher mit max. 11 Zeichen (1 Länge + 11 Daten)
endstruct

dim d as mydate  ' Struktur dimensionieren

d.sec = 23
d.min = 42
d.hour = 12
d.mon = 2
d.year = 2012
d.text = "hello"
d.buf.var(4) = 123
d.buf.text(0)="ABC"

[..]
```

MENGEN

Mengen sind spezielle Ausdrücke, die eine Sammlung von konstanten Werten zusammenfassen und das Verwalten oder Zuweisen von Datensammlungen erleichtern (z.B. eine Liste von Zahlen und Texten).

Eine Menge steht im Quelltext innerhalb geschweifter Klammern ***{...}***. Wenn als Basis eine Strukturdeklaration zugrunde liegt, können innerhalb einer Menge weitere Untermengen eingebettet sein.

Eine Menge besteht also grundsätzlich aus einer sog. ***Basismenge*** und Ggf. weiteren ***Untermengen***. Als ***Basis**menge* wird das erste Klammerpaar bezeichnet, in welcher sich die Werte und Ggf. weitere Untermengen befinden. Weiterhin beginnt eine Menge mit einem numerischen Datentyp oder einem Strukturnamen.

Wird eine Menge mit einer Strukturdeklaration eingeleitet, werden die entsprechend in der Struktur angegebenen Datentypen erwartet. Strukturelemente erwarten wiederum eine Untermenge mit passendem Strukturnamen (verschachtelt). Siehe hierzu Beispiel 2.

- **Hinweis: Intrinsische Datentypen wie *string, MemoryBlock, Graphics, dptr, eptr, sptr* sind *nicht* erlaubt.**

BEISPIELE

Folgend eine Basismenge in welcher alle Werte als Typ "byte" betrachtet werden. Dies betrifft auch die Zeichenkette die in diesem Fall eine Sammlung von einzelnen Bytes - den Buchstaben - darstellt.

```
byte{ 1, 2, 3, "hallo" }
```

Folgend eine Basismenge in welcher alle Werte als Typ "uint24" betrachtet werden

```
uint24{ 1, 2, 3 }
```

Folgend eine Basismenge in welcher die Werte von **verschiedenem** Typ sind, der Aufbau "was ist was" wird durch eine Strukturdeklaration definiert.

```
struct mystruct
  byte myarray(2)
  word value1
  long value2
endstruct

mystruct{ { 1, 2 ,3 }, 4, 5 }
'              ^       ^  ^
'              |       |  |
'              |       |  +- mystruct.value2
'              |       +- mystruct.value1
'              +- mystruct.myarray
```

PRAKTISCHE ANWENDUNG

Anwendung finden Mengen z.B. beim definieren von Datensätzen im Programmspeicher (flash) oder der Zuweisung von Datensätzen auf einen Speicherbereich im Arbeitsspeicher (sram) oder Eeprom (eram).

Mithilfe von Mengen können Daten/Werte die sonst einzeln vorliegen für den Menschen leichter lesbar im Quelltext zusammengefasst werden. Weiterhin ermöglichen sie erst die Zuweisung von ganzen Datensätzen mit Elementen verschiedener Typen "in einem Rutsch" auf Speicherbereiche, z.B. einem Array.

DATENSÄTZE IM FLASH

BEISPIEL 1

```
struct mystruct
  byte   myarray(2)
  word   level
  string text[14]
endstruct

data table
  mystruct{ { 1, 2 ,3 }, 4, "hallo" } 'Text wird auf die in der Strukturdeklaration
  mystruct{ { 5, 6 ,7 }, 8, "ballo" } 'definierten Länge von "text" mit Leerzeichen aufgefüllt.
enddata
```

Obiges Beispiel ist dasselbe wie die Einzelangabe mit .db, .dw usw.:

```
data table
  .db 1, 2, 3
  .dw 4
  .db "hallo         "
  .db 5, 6, 7
```

```
  .dw 8
  .db "ballo         "
enddata
```

Basismengen erlauben auch "normale" Datentypen wie *byte, word* usw., womit man natürlich in Flash-Datenobjekten nicht viel gewinnt, aber es ist möglich:

```
data table
  byte{ 1, 2, 3, "hallo" }
  word{ 0x1122, 0xabcd9 }
enddata
```

Obiges Beispiel ist dasselbe wie die Einzelangabe mit .db, .dw usw.:

```
data table
  .db 1, 2, 3, "hallo"
  .dw 0x1122, 0xabcd9
enddata
```

BEISPIEL 2

Basismenge mit Struktur und Untermenge mit Struktur.

```
struct pair
  byte   value1
  byte   value2
endstruct
struct mystruct
  word   level
  string text[14]
  pair   bytepair
endstruct

data table
  mystruct{ 1, "hello", pair{ 23, 42 } }
  mystruct{ 2, "world", pair{ 10, 20 } }
  '         ^     ^      ^    ^   ^
  '         |     |      |    |   + mystruct.bytepair.value2
  '         |     |      |    + mystruct.bytepair.value1
  '         |     |      +- mystruct.bytepair
  '         |     +- mystruct.text
  '         +- mystruct.level
enddata
```

ZUWEISUNG IN DEN ARBEITSSPEICHER/EEPROM

Ein Array eines Datentyps ist nichts weiter als aneinandergereihte Speicherzellen die einzeln angesprochen werden können. Sie können also insgesamt auch als ein durchgängiger Speicherblock betrachtet werden.

ARRAY

Folgend ein Beispiel um ein Array in einem Rutsch mit festen Werten zu belegen:

```
dim a(7) as byte '8 Elemente
a() = byte{ 11, 22, 33, 44, 55, 66, 77, 88 }
'Das Array enthält nach dieser Zuweisung im Element a(0) den Wert 11, im
'Element a(1) den Wert 22 usw.
```

Hinweis: *Aufpassen muss man hier mit der Anzahl der Elemente in der Menge, denn wird die Anzahl des Speicherplatzes vom Array überschritten, werden Ggf. nachfolgende Daten überschrieben. Die Menge hat keinen direkten Bezug zum Array. Aus Sicht der Menge ist es nur ein Speicherplatz.*
Es kann auch nur ein Teil belegt werden:

```
dim a(7) as byte '8 Elemente
a(3) = byte{ 11, 22 }
'Das Array enthält nach dieser Zuweisung im Element a(3) den Wert 11 und im
'Element a(4) den Wert 22
```

POINTER/MEMORYBLOCK/STRING

Auch intrinsische Datentypen wie Pointer oder MemoryBlock, welche einen Zeiger auf einen Speicherbereich enthalten, können mit Daten beschrieben werden. Hierbei wird die Zieladresse aus dem Pointer/MemoryBlock/String ausgelesen und das als Speicherziel verwendet. D.h. z.B. bei einem MemoryBlock oder String ist das Ziel der Daten der Speicherblock welcher dem MemoryBlock oder dem String zugewiesen wurde.

Wichtig: *Es muss sichergestellt werden, dass dem Pointer, dem MemoryBlock oder String auch tatsächlich ein gültiger Speicherblock oder eine Zieladresse zugewiesen wurde!*

```
dim m as MemoryBlock
m = new MemoryBlock(16) 'Einen Speicherblock anlegen mit 16 Bytes Speicherplatz
if m <> nil then          'Prüfen ob der erzeugte Speicherblock zugewiesen wurde.
  m = byte{ 11, "Hallo" } 'Schreibt den Bytewert 11 und nachfolgend die bytes des Textes in den Speicherblock.
end if
dim p as sptr           'Einen Pointer anlegen
dim buffer(15) as byte 'Einen Speicherbereich mit 16 Bytes anlegen (Array)

p = buffer(4).Addr      'Startadresse ab Element buffer(4) zuweisen
p = byte{ 11, "Hallo" }
'Im Array buffer steht nun:
'  buffer(4) = 11
'  buffer(5) = 72  (ASCII-Zeichen "H")
'  buffer(6) = 97  (ASCII-Zeichen "a")
'  buffer(7) = 108 (ASCII-Zeichen "l")
'  buffer(8) = 108 (ASCII-Zeichen "l")
'  buffer(9) = 111 (ASCII-Zeichen "o")
```

ARRAYS

Arrays sind verkettete Datentypen gleichen Typs. In Luna können numerische Variablen, Stringvariablen und Strukturen als statisches, eindimensionales Array dimensioniert werden. Statisch bedeutet, dass die Anzahl der Elemente fest vorgegeben wird. Werden mehrere verschiedene Daten je Arrayelement benötigt, kann man Arrays mit Strukturen erstellen.

Um ein Array zu dimensionieren, fügt man bei der Dimensionierung von Variablen eine in Klammern eingefasste Elementanzahl ein.Die Elementanzahl und auch der Zugriff auf einzelne Elemente eines Arrays sind *nullbasiert*. Das erste Element eines Arrays ist also das Element "0".

ELEMENTANZAHL ALS EIGENSCHAFT

Bei Arrays kann mittels der Eigenschaft ***Ubound*** die dimensionierte Elementanzahl gelesen werden:

```
dim i,a(63) as byte

for i=0 to a().Ubound
  [..]
next
```

INITIALISIERUNG

Arrayelemente im Arbeitsspeicher werden nach der Deklaration und Dimensionierung (wie alle anderen Variablen im Arbeitsspeicher) mit dem Wert 0, nil oder leer initialisiert. Arrays/Variablen im Eeeprom werden *nicht* initialisiert.

Beispiel Dimensionierung Arbeitsspeicher

```
dim a(99) as byte    ' Byte-Array mit 100 Elementen dimensionieren
dim s(2) as string   ' String-Array mit 3 Elementen dimensionieren

a(22) = 42           ' Dem 23. Element den Wert 42 zuweisen
Print Str(a(22))     ' Wert des Elements ausgeben
s(0) = "Hallo Welt"  ' Dem 1. Element einen Text zuweisen
Print s(0)           ' Wert des Elements ausgeben
```

Beispiel Dimensionierung Eeprom

```
eedim a(99) as byte        ' Byte-Array mit 100 Elementen dimensionieren
eedim s[10](2) as string   ' String-Array mit 3 Elementen und je 10 Bytes Länge dimensionieren.
                           ' (Strings im Eeprom sind statisch und benötigen eine Längenangabe)
a(22) = 42                 ' Dem 23. Element den Wert 42 zuweisen
Print Str(a(22))           ' Wert des Elements ausgeben
s(0) = "Hallo Welt"        ' Dem 1. Element einen Text zuweisen
Print s(0)                 ' Wert des Elements ausgeben
```

EINGEBAUTE OBJEKTE

Die Objekte sind statische und/oder auch zur Laufzeit des Programms instanzierbare bzw. temporär erzeugbare Speicherobjekte für verschiedenste Anwendungsbereiche.

- MemoryBlock - Speicherblöcke dynamisch im Arbeitsspeicher
- String - Dynamische Variablen für Zeichenketten
- Struct-EndStruct - Speicherstrukturen Arbeitsspeicher
- Data-EndData - Speicherobjekte Flashspeicher
- IncludeData - Speicherobjekte Flashspeicher
- Eeprom-EndEeprom - Speicherobjekte Eeprom

BEDINGUNGEN

Mit Bedingungen können Fallunterscheidungen getroffen werden um die Eine oder Andere Aktion abhängig vom Ergebnis der Bedingung durchzuführen.

PRÄPROZESSOR

- #if #elseif #else #endif
- #select #case #default #endselect

PROGRAMMCODE

- if - elseif - else - endif
- select case - case - default - endselect
- when - do

SCHLEIFEN

Es sind die drei bekannten Schleifenkonstrukte implementiert:

- For-Next [1)]
- Do-Loop [2)]
- While-Wend [3)]

Zusätzliche Schleifenbefehle:

- exit/break Befehl zum vorzeitigen verlassen einer Schleife.
- continue Befehl zum Fortsetzen der Schleife mit der nächsten Iteration.

1) Start und Endwert, automatisch inkrementierender oder dekrementierender Zähler.

2) Optionale Exit-Bedingung am Schleifenende.

3) Exit-Bedingung am Schleifenanfang.

ISR-ENDISR

Mit Isr-EndIsr definiert man eine Interrupt-Serviceroutine. Der Name der Serviceroutine wird dann einem der Hardware-Interrupts des Controllers zugewiesen, z.B. einem Timer oder der seriellen Schnittstelle.

Variablen die in einer Interrupt-Serviceroutine dimensioniert werden sind statische Variablen (wie globale Variablen im Hauptprogramm) und *namentlich* der Interrupt-Serviceroutine vergleichbar einer Methode lokal zugeordnet.

Syntax:

- ***Isr*** *Bezeichner [Schalter (siehe Tabelle)]*
  - Programmcode der Serviceroutine
- ***EndIsr***

Schalter	Beschreibung
save	Alle Register werden Flash-speicherplatzsparend gesichert (Vorgabe).
nosave	Keine Register werden gesichert. Die enthaltene Routine muss dies selbst vornehmen.
fastall	Alle Register werden gesichert (Schnelle, längere Routine).
fastauto	Die zu sichernden Register werden automatisch vom Optimierer durch Verfolgung des Programmlaufs ermittelt (schnellste Variante).

ERLÄUTERUNG

save/nosave/fastall/fastauto: Optionale Parameter die das Sichern der Register steuern. Tritt ein Interrupt auf, wird die aktuelle Programmausführung unterbrochen und die Serviceroutine abgearbeitet. Dabei werden Register/Daten verändert, die vorher gesichert und nach der Abarbeitung wiederhergestellt werden müssen. Ist nicht genügend Arbeitsspeicher vorhanden, oder ist die Service-Routine sehr zeitkritisch, kann man bspw. durch ***nosave*** das sichern der Register abgestimmt auf die Aufgabe selbst vornehmen bzw. durch ***fastauto*** automatisch vornehmen lassen.

BESONDERHEIT FASTAUTO

Mit der Option ***fastauto*** werden die genutzten Register vom Optimierer durch Verfolgung des Programmlaufs automatisch ermittelt.

WICHTIGER HINWEIS

Interrupt-Serviceroutinen sollten kurz gehalten werden und keine Warteschleifen, unterbrechende Befehle oder Zugriffe auf Eeprom-Speicherbereiche enthalten. Der Programmcode in der Serviceroutine sollte immer so kurz gehalten, werden, dass die Ausführung beendet ist bevor ein nächster Interrupt eintrifft. Unterprogrammaufrufe sind problemlos möglich.

Vorsicht bei Datentypen die auf dynamischen Speicher verweisen wie *string* **oder** *MemoryBlock*. **Hier ist Sorge zu tragen, dass auf sie *nicht gleichzeitig in Hauptprogramm und der Serviceroutine* zugegriffen wird. Die Verarbeitung von Strings oder MemoryBlocks ist rechenintensiver. Was der Vorgabe ISR-Routinen kurz zu halten widerspricht. Es wird davon abgeraten!**

BEISPIEL

```
Timer0.isr   = meineServiceRoutine  ' Serviceroutine zuweisen
Timer0.clock = 1024                 ' Prescaler einstellen
Timer0.enable                       ' Timer einschalten

Avr.Interrupts.Enable               ' Globale Interrupts einschalten
do
loop

Isr meineServiceRoutine
  print "Timer-Interrupt ausgelöst"
EndIsr
```

EVENT-ENDEVENT

Dient zur Deklaration eines Events. Einige der Interfaces, wie z.Bsp. Twi/Ic2, Dcf77 u.A. rufen bei einem bestimmten Ereignis ein **namentlich benanntes** Event auf. Deklariert man ein Event mit dem angegebenen Namen, wird es vom Interface, Modul oder Objekt bei auftreten des Ereignisses aufgerufen.

Syntax:

- ***Event*** *eventname*
  - *[Programmcode]*
- ***EndEvent***

Beispiel:

```
[..]
Event TwiError
  Print "Status = ";str(twi.Status)
EndEvent
```

EXCEPTION-ENDEXCEPTION

Exceptions dienen der Fehlersuche direkt im Programm.Fehlerbehandlung bei Auftreten einer Exception.

Syntax:

- ***Exception*** *ExceptionName*
  - *Programmcode*
  - ***[Raise]***
- ***EndException***

ExceptionName	Beschreibung	Raise
DivisionByZero[1)]	Division durch Null in einer Kalkulation aufgetreten	ja
OutOfBoundsMemory	MemoryBlock Objektgrenze verletzt	ja
NilObjectException	Objekt auf das sich der Aufruf bezieht existiert nicht	ja
OutOfSpaceMemory	Arbeitsspeicher voll, Objekt kann nicht alloziert werden	nein
StackOverflow	Der Stack ist übergelaufen. Siehe auch..	nein

Wird ein Exceptionhandler im Programmcode verwendet, werden die jeweiligen Debugfunktionen aktiviert. Die Zugriffsgeschwindigkeiten werden z.T. reduziert und es wird zusätzlicher Speicherplatz im Flash benötigt.

Sofern möglich (siehe Tabelle), kann mit dem Befehl **Raise** die Programmausführung fortgesetzt werden. Andernfalls wird das Programm nach Auftreten einer Exception durch eine Endlosschleife angehalten (HALT).

BEISPIEL

```
' Initialisierung
[..]
' Hauptprogramm
dim a as byte
dim m,m1 as MemoryBlock
m.New(100)            'MemoryBlock allozieren mit 100 Bytes
a=m.ByteValue(100)    'Löst Exception aus, da außerhalb der Objektgrenzen
                      '(Zugriff hier nur 0-99 erlaubt, Offsets sind Nullbasiert)
m1.ByteValue(0)=1     'Zuweisung zu einem nicht existierendes Objekt
m1.New(3000)          'mehr Speicher allozieren als vorhanden ist (Programm wird angehalten)
do
Loop

' Exception-Handler definieren (Aktiviert die Überwachungsfunktionen)
Exception OutOfSpaceMemory
  ' Fehlermeldung ausgeben
  Print "*** Exception OutOfSpaceMemory ***"
EndException
Exception OutOfBoundsMemory
  ' Fehlermeldung ausgeben
  Print "*** Exception OutOfBoundsMemory ***"
  Raise  ' Programm fortführen
EndException
Exception NilObjectException
  ' Fehlermeldung ausgeben
  Print "*** Exception NilObjectException ***"
  Raise  ' Programm fortführen
EndException
Exception StackOverflow
  ' Fehlermeldung ausgeben
  Print "*** Exception StackOverflow ***"
EndException
```

1) Ab Version 2015.r1

METHODEN

Methoden sind Unterprogramme zu denen bei Aufruf gesprungen wird. Dort wird der vorhandene Programmcode des Unterprogramms ausgeführt. Im Anschluss wird zum Ausgangspunkt zurückgekehrt, um mit den dort nachfolgenden Befehlen fortzufahren.

In Luna gibt es zwei verschiedene Varianten von Methoden:

- Procedure-EndProc
  (Methode mit Lokalität und optionalen Parametern)
- Function-EndFunc
  (Methode mit Lokalität, optionalen Parametern und Rückgabewert)

EIGENSCHAFTEN

Methoden besitzen die Eigenschaft ***.Addr***, die man zur Ermittlung der Adresse der Methode im Programmspeicher (Flash) benötigt.**MeineMethode().Addr** gibt als Ergebnis die **Word-Adresse** zurück.

SIEHE AUCH

- Call
- Void
- Icall
- Variablen in Methoden

KLASSEN

In Luna existiert per Vorgabe die Klasse "Avr". Zur Erweiterung von Funktionen oder zur Modularisierung der Programme können benutzerdefinierte Klassen angelegt werden.

- Grundlagen
- Avr (Basisklasse)

BENUTZERDEFINIERTE KLASSEN

- Class-EndClass

PRÄPROZESSOR (LUNA-QUELLTEXT)

Der Präprozessor ist ein Teil des Compilers, der den Sourcecode für den Kompilier- und Assembliervorgang *vorverarbeitet*. Im Luna-Compiler existieren zwei Präprozessoren. Einmal für den Luna- und einmal für den Assembler-Quelltext.

Im Präprozessor werden folgende Teilbereiche bearbeitet:

- Auflösung von arithmetischen und logischen Ausdrücken mit Konstanten.
- Bedingte Einbindung von Code-Bereichen.
- Einbinden von externen Sourcedateien und -Daten
- Textersetzungen von Definitionen/Makros im Sourcecode.
- Auflösung von Inline-Funktionen.

PRÄPROZESSORFUNKTIONEN IM LUNA-CODE

ANWEISUNGEN

- **#define** - Defines/Aliase.
- **#undef** - Define entfernen.
- **#ide** - IDE-Steuerung im Source.
- **#pragma, #pragmaSave, #pragmaRestore, #pragmaDefault** - Compiler-Steuerung im Source.
- **#if #else #elseif #endif** - Bedingtes Kompilieren.
- **#select #case #default #endselect** - Bedingtes Kompilieren.
- **#macro** - Makros
- **#include** - Quelltextdateien einbinden.
- **#includeData** - Binärdaten in den Flash einbinden.
- **#library** - Externe Bibliothek einbinden.
- **#cdecl, #odecl, #idecl** - Parameterdefinition für indirekte Aufrufe.
- **#error** - Fehlermeldung ausgeben (Kompiliervorgang abbrechen).
- **#warning** - Warnmeldung ausgeben.
- **#message** - Nachricht ausgeben.

FUNKTIONEN

- **Defined()** - Prüfen ob eine Konstante oder ein Symbol definiert ist.
- **lo8()** bzw. **low()** - Low-Byte eines 16-Bit-Wertes
- **hi8()** bzw. **high()** - High-Byte eines 16-Bit-Wertes
- **byte1()** - 1. Byte eines 32-Bit-Wertes
- **byte2()** - 2. Byte eines 32-Bit-Wertes
- **byte3()** - 3. Byte eines 32-Bit-Wertes
- **byte4()** - 4. Byte eines 32-Bit-Wertes
- **byte()** - Typcasting auf 8-Bit-Integer.
- **int8()** - Typcasting auf 8-Bit-Integer mit Vorzeichen.
- **word()** - Typcasting auf 16-Bit-Integer.
- **integer()** - Typcasting auf 16-Bit-Integer mit Vorzeichen.
- **int24()** - Typcasting auf 24-Bit-Integer mit Vorzeichen.
- **uint24()** - Typcasting auf 24-Bit-Integer.
- **long()** - Typcasting auf 32-Bit-Integer.
- **longint()** - Typcasting auf 32-Bit-Integer mit Vorzeichen.
- **single()** - Typcasting auf 32-Bit-Float mit Vorzeichen.
- **float()** - Typcasting auf 32-Bit-Float mit Vorzeichen.

- **odd()** - Prüfen ob Wert ungerade.
- **even()** - Prüfen ob Wert gerade.
- **chr()** - Umwandlung nach Binärstring (byte).
- **mkb()** - Umwandlung nach Binärstring (byte).
- **mki()** - Umwandlung nach Binärstring (integer).
- **mkw()** - Umwandlung nach Binärstring (word).
- **mkl()** - Umwandlung nach Binärstring (long).
- **mks()** - Umwandlung nach Binärstring (single).
- **strfill()** - String auffüllen mit Zeichenkette.
- **hex()** - Konvertierung nach Hexadezimaldarstellung.
- **str()** - Konvertierung nach Dezimaldarstellung.

- **bin()** - Konvertierung nach Binärdarstellung.
- **asc()** - Umwandlung des ersten Zeichens einer Zeichenkette in sein numerisches Equivalent.
- **min()** - Arithmetische Funktion.
- **max()** - Arithmetische Funktion.
- **left()** - Linken Teil eines Textes lesen (String).
- **right()** - Rechten Teil eines Textes lesen (String).
- **mid()** - Teil eines Textes lesen (String).
- **format()** - Formatierte, dezimale Zahlendarstellung.
- **nthfield()** - Teil einer separierten Zeichenkette.
- **val()** - Zeichenkette mit Dezimalzahl zu Integerwert konvertieren.

EINGEBAUTE INTERFACES

In Luna sind die gebräuchlichsten Hardware-Controllerfunktionen bzw. -Schnittstellen als Compiler-interne oder als externe Interface- oder Modul-Bibliotheken verfügbar. Die Interfaces/Module unterstützen den Entwickler bei der Konfiguration, sowie bei Hardware- und Softwarezugriffen auf Schnittstellen oder Protokolle. Zusätzlich sind verschiedene Software-Implementationen von z.Bsp. Schnittstellen/Protokollen vorhanden.

ALLGEMEIN

Generell sind Controller-Funktionalitäten über den Direktzugriff auf die Konfigurations- und Datenports verwendbar. Die Konfiguration bzw. der Zugriff erfolgt hier anhand der Portnamen und Konfigurationsbits laut dem Datenblatt des Controllers.

Siehe auch: Externe Bibliotheken

Folgende Liste verzeichnet die direkt im Compiler eingebauten Module/Interfaces.

- **Avr - Basis (der Controller)**

- Eeprom - Eepromspeicher/-Objekte
- Sram - Arbeitsspeicher
- Uart - Uart-Interfaces (tiny,mega)
- SoftUart - SoftUart-Interface

ZUSÄTZLICH FÜR ATXMEGA

- **Universal-Interfaces** - Das universelle Hardwareinterface für die Atxmega-Controller.
- Usart - Usart-Interfaces (atxmega)

ASM-ENDASM

Einfügen von Assemblerquelltext an die aktuelle Position.**Syntax:**

- ***Asm***
  - *assembler-source code*
- ***endAsm***

INLINE-ASSEMBLER

Zeichenketten und Kommentare der Inline-Assembler-Quellcodes folgen der gleichen Syntax wie im Luna-Code. Zeichenketten sind daher immer mit Anführungszeichen versehen. Kommentare beginnen mit **'** oder ***//***, zusätzlich **;** am Zeilenanfang.

ALLGEMEIN

Die Zusatzfunktionen und Eigenschaften des Assemblers:

- Standard-Defines der Controller-Register im Luna-Assembler

- Assemblerbefehle
- Präprozessor
- Bedingungen
- Makros

- Liste der globalen Konstanten
- StdLib.interface (Standard-Bibliothek)

LABELS IM ASSEMBLER-SOURCE

Jedes Label steht in einer eigenen Zeile, nachfolgende Befehle sind nicht vorgesehen.

Inline-Assembler

Die Labels in Luna sind auf die aktuelle Klasse bezogen, z.B. das Label "mylabel" im Hauptprogramm wird vom Compiler zu "classAvrMylabel" erweitert. "mylabel" in einer Klasse "myclass" wird zu "classMyclassMylabel". Auch Labels die im Inline-Assembler erstellt wurden, werden diesbezüglich erweitert.

Die Namenserweiterung ist zwingend notwendig um die verschienenen Namensräume getrennt halten zu können und einen Zugriff aus dem Luna-Source heraus zu ermöglichen. Dies betrifft ausschließlich den Inline-Assembler im Luna-Source.

BEISPIEL 1

```
' somewhere in the main program
Asm
  classAvrMyLoop:
    nop
    rjmp classAvrMyLoop
endAsm
```

BEISPIEL 2

```
' somewhere in the main program
Asm
  MyLoop:             'will be expanded to  "classAvrMyLoop"
    nop
    rjmp classAvrMyLoop
endAsm
```

BEISPIEL 3

```
' call assembler routine
call example
' Somewhere in the program code
Asm
  classAvrexample:
    add  R16,R17
    ori  R16,&h33
```

```
    ldi  R16,(1<<3) or (1<<7)   ; Set bit 3 and 7
    ldi  ZL,lo8(classAvrexample)
    ldi  ZH,hi8(classAvrexample)
    inc  R17
    ret
endAsm
```

Präprozessor (Luna-Quelltext)

Der Präprozessor ist ein Teil des Compilers, der den Sourcecode für den Kompilier- und Assembliervorgang *vorverarbeitet*. Im Luna-Compiler existieren zwei Präprozessoren. Einmal für den Luna- und einmal für den Assembler-Quelltext.

Im Präprozessor werden folgende Teilbereiche bearbeitet:

- Auflösung von arithmetischen und logischen Ausdrücken mit Konstanten.
- Bedingte Einbindung von Code-Bereichen.
- Einbinden von externen Sourcedateien und -Daten
- Textersetzungen von Definitionen/Makros im Sourcecode.
- Auflösung von Inline-Funktionen.

PRÄPROZESSORFUNKTIONEN IM LUNA-CODE

ANWEISUNGEN

- **#define** - Defines/Aliase.
- **#undef** - Define entfernen.
- **#ide** - IDE-Steuerung im Source.
- **#pragma, #pragmaSave, #pragmaRestore, #pragmaDefault** - Compiler-Steuerung im Source.
- **#if #else #elseif #endif** - Bedingtes Kompilieren.
- **#select #case #default #endselect** - Bedingtes Kompilieren.
- **#macro** - Makros
- **#include** - Quelltextdateien einbinden.
- **#includeData** - Binärdaten in den Flash einbinden.
- **#library** - Externe Bibliothek einbinden.
- **#cdecl, #odecl, #idecl** - Parameterdefinition für indirekte Aufrufe.
- **#error** - Fehlermeldung ausgeben (Kompiliervorgang abbrechen).
- **#warning** - Warnmeldung ausgeben.
- **#message** - Nachricht ausgeben.

FUNKTIONEN

- **Defined()** - Prüfen ob eine Konstante oder ein Symbol definiert ist.
- **lo8()** bzw. **low()** - Low-Byte eines 16-Bit-Wertes
- **hi8()** bzw. **high()** - High-Byte eines 16-Bit-Wertes
- **byte1()** - 1. Byte eines 32-Bit-Wertes
- **byte2()** - 2. Byte eines 32-Bit-Wertes
- **byte3()** - 3. Byte eines 32-Bit-Wertes
- **byte4()** - 4. Byte eines 32-Bit-Wertes
- **byte()** - Typcasting auf 8-Bit-Integer.
- **int8()** - Typcasting auf 8-Bit-Integer mit Vorzeichen.
- **word()** - Typcasting auf 16-Bit-Integer.
- **integer()** - Typcasting auf 16-Bit-Integer mit Vorzeichen.
- **int24()** - Typcasting auf 24-Bit-Integer mit Vorzeichen.
- **uint24()** - Typcasting auf 24-Bit-Integer.
- **long()** - Typcasting auf 32-Bit-Integer.
- **longint()** - Typcasting auf 32-Bit-Integer mit Vorzeichen.
- **single()** - Typcasting auf 32-Bit-Float mit Vorzeichen.
- **float()** - Typcasting auf 32-Bit-Float mit Vorzeichen.

- **odd()** - Prüfen ob Wert ungerade.
- **even()** - Prüfen ob Wert gerade.
- **chr()** - Umwandlung nach Binärstring (byte).
- **mkb()** - Umwandlung nach Binärstring (byte).
- **mki()** - Umwandlung nach Binärstring (integer).
- **mkw()** - Umwandlung nach Binärstring (word).
- **mkl()** - Umwandlung nach Binärstring (long).
- **mks()** - Umwandlung nach Binärstring (single).
- **strfill()** - String auffüllen mit Zeichenkette.
- **hex()** - Konvertierung nach Hexadezimaldarstellung.
- **str()** - Konvertierung nach Dezimaldarstellung.
- **bin()** - Konvertierung nach Binärdarstellung.

- **asc()** - Umwandlung des ersten Zeichens einer Zeichenkette in sein numerisches Equivalent.
- **min()** - Arithmetische Funktion.
- **max()** - Arithmetische Funktion.
- **left()** - Linken Teil eines Textes lesen (String).
- **right()** - Rechten Teil eines Textes lesen (String).
- **mid()** - Teil eines Textes lesen (String).
- **format()** - Formatierte, dezimale Zahlendarstellung.
- **nthfield()** - Teil einer separierten Zeichenkette.
- **val()** - Zeichenkette mit Dezimalzahl zu Integerwert konvertieren.

Präprozessor - Assembler

Der im Luna-Assembler eingebaute Präprozessor ist ein sog. Makroprozessor.

Im Präprozessor werden folgende Teilbereiche bearbeitet:

- Auflösung von arithmetischen und logischen Ausdrücken mit Konstanten.
- Bedingte Einbindung von Code-Bereichen.
- Einbinden von externen Sourcedateien und -Daten
- Textersetzungen von Definitionen im Sourcecode.
- Auflösung von Inline-Funktionen.
- Auflösung von Makros.

PRÄPROZESSORFUNKTIONEN IM ASSEMBLER-CODE

ANWEISUNGEN

- **.if .else .elseif .endif** - Bedingtes Assemblieren.
- **.select .case .default .endselect** - Bedingtes Assemblieren.
- **.macro .endmacro** - Makros.

Command	Description	Example
.equ	create a constant	.equ var = 123.45
.set	create/assign a constant	.set var = 123.45
.def	Alias	.def temp = R16
.device	set the avr controller type	.device atmega32
.import	import a label	.import _myLabel
.importClass[1)]	import avr class. sets all defines and constants of the selected avr controller inclusive .device	.importClass atmega32
.importObject[1)]	import a library object	.importObject Macro_Delay
.importUsedObjects[1)]	auto-imports all in the source code used libraries objects	.importUsedObjects
.cseg	select flash segment	.cseg
.dseg	select sram segment	.dseg
.eseg	select eeprom segment	.eseg
.org	Sets the pointer of the actual active segment to a specific value	.org intVectorSize
.error	Display a error message. Break compile/assemble.	.error "message"
.warning	Display a warning message.	.warning "message"
.message, .print	Display a info message without line number and origin.	.message "message"
.regisr	register a label to a interrupt vector "on the fly" (used for Library Code)	.regisr vectorAddress,mylabel
.db	(byte) 1 byte each value and strings, data block. [2)]	.db "hello",0x3b
.dw	(word) 2 bytes each value, data block. [2)]	.dw 0x3bda,0xcf01
.dt	(triple) 3 bytes each value, data block. [2)]	.dt 0xf0d1a4
.dl	(long) 4 bytes each value, data block. [2)]	.dl 0xff01ddca
.bin	(file) data block from file byte-wise. [2)]	.bin "filepath"
.odb	(byte) 1 byte each value and strings, object data block	.odb "hello",0x3b
.odw	(word) 2 bytes each value, object data block. [3)]	.odw 0x3bda,0xcf01
.odt	(triple) 3 bytes each value, object data block. [3)]	.odt 0xf0d1a4
.odl	(long) 4 bytes each value, object data block. [3)]	.odl 0xff01ddca
.obin	(file) data block from file byte-wise. [3)]	.obin "filepath"
.dobjend	endmark for object data block with .odb, .odw, ... [4)]	.dobjend label

Beispiel für Objekt-Datenblock

```
classServiceudpdDatagram:
.odb  "this is a message from the luna udpd server",0x0D,0x0A
```

```
.odb  "build with lavrc version ","2013.r6.7",0x0D,0x0A
.odb  0x0D,0x0A
.dobjend classServiceudpdDatagram
```

FUNKTIONEN

Die Präprozessorfunktionen können nur mit Konstanten verwendet werden.

Funktionen aus dem Luna-Befehlssatz, abgebildet im Präprozessor.

- **lo8()** bzw. **low()** - Low-Byte eines 16-Bit-Wertes
- **hi8()** bzw. **high()** - High-Byte eines 16-Bit-Wertes
- **byte1()** - 1. Byte eines 32-Bit-Wertes
- **byte2()** - 2. Byte eines 32-Bit-Wertes
- **byte3()** - 3. Byte eines 32-Bit-Wertes
- **byte4()** - 4. Byte eines 32-Bit-Wertes
- **byte()** - Typcasting auf 8-Bit-Integer.
- **word()** - Typcasting auf 16-Bit-Integer.
- **integer()** - Typcasting auf 16-Bit-Integer mit Vorzeichen.
- **long()** - Typcasting auf 32-Bit-Integer.
- **longint()** - Typcasting auf 32-Bit-Integer mit Vorzeichen.
- **int8()** - Typcasting auf 8-Bit-Integer mit Vorzeichen.
- **int16()** - Typcasting auf 16-Bit-Integer mit Vorzeichen.
- **int24()** - Typcasting auf 24-Bit-Integer mit Vorzeichen.
- **int32()** - Typcasting auf 32-Bit-Integer mit Vorzeichen.
- **uint8()** - Typcasting auf 8-Bit-Integer.
- **uint16()** - Typcasting auf 16-Bit-Integer.
- **uint24()** - Typcasting auf 24-Bit-Integer.
- **uint32()** - Typcasting auf 32-Bit-Integer.
- **single()** - Typcasting auf 32-Bit-Float mit Vorzeichen.
- **float()** - Typcasting auf 32-Bit-Float mit Vorzeichen.
- **odd()** - Prüfen ob Wert ungerade.
- **even()** - Prüfen ob Wert gerade.
- **chr()** - Umwandlung nach Binärstring (byte).
- **mkb()** - Umwandlung nach Binärstring (byte).
- **mki()** - Umwandlung nach Binärstring (integer).
- **mkw()** - Umwandlung nach Binärstring (word).
- **mkl()** - Umwandlung nach Binärstring (long).
- **mks()** - Umwandlung nach Binärstring (single).
- **strfill()** - String auffüllen mit Zeichenkette.
- **hex()** - Konvertierung nach Hexadezimaldarstellung.
- **str()** - Konvertierung nach Dezimaldarstellung.
- **bin()** - Konvertierung nach Binärdarstellung.
- **asc()** - Umwandlung des ersten Zeichens einer Zeichenkette in sein numerisches Equivalent.
- **min()** - Arithmetische Funktion.
- **max()** - Arithmetische Funktion.
- **left()** - Linken Teil eines Textes lesen (String).
- **right()** - Rechten Teil eines Textes lesen (String).
- **mid()** - Teil eines Textes lesen (String).
- **format()** - Formatierte, dezimale Zahlendarstellung.
- **nthfield()** - Teil einer separierten Zeichenkette.
- **val()** - Zeichenkette mit Dezimalzahl zu Integerwert konvertieren.

Sonderfunktionen, nur im Präprozessor.

- **Descriptor()** - Aktuellen Deskriptor (Zeiger/Position) des Assemblers lesen (byte-Adresse).
- **MakeIdentifier()** - Ein Symbol aus einem String erstellen.
- **Defined()** - Prüfen ob eine Konstante oder ein Symbol definiert ist.
- **Replace()** - Zeichenkette in einem String suchen und erste Vorkommende ersetzen.
- **ReplaceAll()** - Zeichenkette in einem String suchen und alle Vorkommenden ersetzen.

1) only used by the compiler

2) Note: End of block auto-aligned/padding to even address!

3) Note: No auto-alignment/padding

4) Initiates the auto-alignment/padding to even address over the complete data set.

Bibliotheken (extern)

LunaAVR verwendet externe Bibliotheken um die Funktionalität zu erweitern. Externe Bibliotheken sind im Code einsehbar und können mit einem komfortablen, grafischen Editor bearbeitet oder neu erstellt werden. Es können neue Bibliotheken angelegt und/oder bestehende Bibliotheken bearbeitet werden.

- **Bibliotheks-Auswahl**
- **Bibliotheks-Dokumentationen**
- **Bibliotheken erstellen**

- Bibliotheken von Anwendern

BIBLIOTHEKEN (LISTE)

Bibliotheken im Ordner

- **"/LibraryStd"** - werden automatisch eingebunden.
  - **StdLib.interface - Standard-Bibliothek**
  - StdLib_Bool.module
  - StdLib_Dump.interface
  - StdLib_Flash.interface
  - StdLib_FloatMath.module
  - StdLib_Math.module
  - StdLib_Operator.interface
  - StdLib_PortX.interface
  - StdLib_String.module
  - StdLib_Timer0.interface
  - StdLib_Timer1345.interface
  - StdLib_Timer2.interface
  - StdLib_Uart.interface

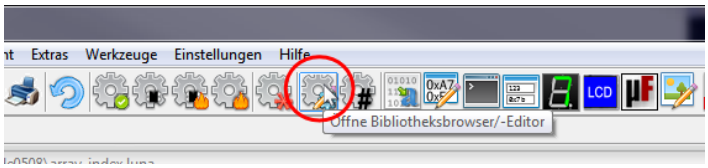
- **"/Library"** - werden mittels Befehl #library eingebunden.
  - **/Custom** - Von Nutzern erstellte Extrabibliotheken.
  - **/Example** - Beispielbibliotheken zum selbst Erstellen.
  - Adc.interface
  - ClkSys.interface
  - Crc16.interface
  - Crc8.interface
  - DCF77.interface
  - Encodings.module
  - FFT.interface
  - FT800.interface
  - Graphics.object
  - iGraphics.interface
  - INTn.interface
  - KeyMgr.interface
  - Lcd4.interface
  - Math64.module
  - NetworkFunctions.module
  - OneWire.interface
  - ow.interface
  - PCInt.interface
  - RotaryEncodier.interface
  - Sleep.interface
  - SoftSpi.interface
  - SoftTwi.interface
  - Sound.interface
  - Spi.interface
  - SpiC.interface
  - SpiD.interface
  - SpiE.interface
  - SpiF.interface
  - TaskKernel.interface
  - TaskMgr.interface
  - TaskTimer.interface

- Twi.interface
- Wdt.interface
- WS0010.interface

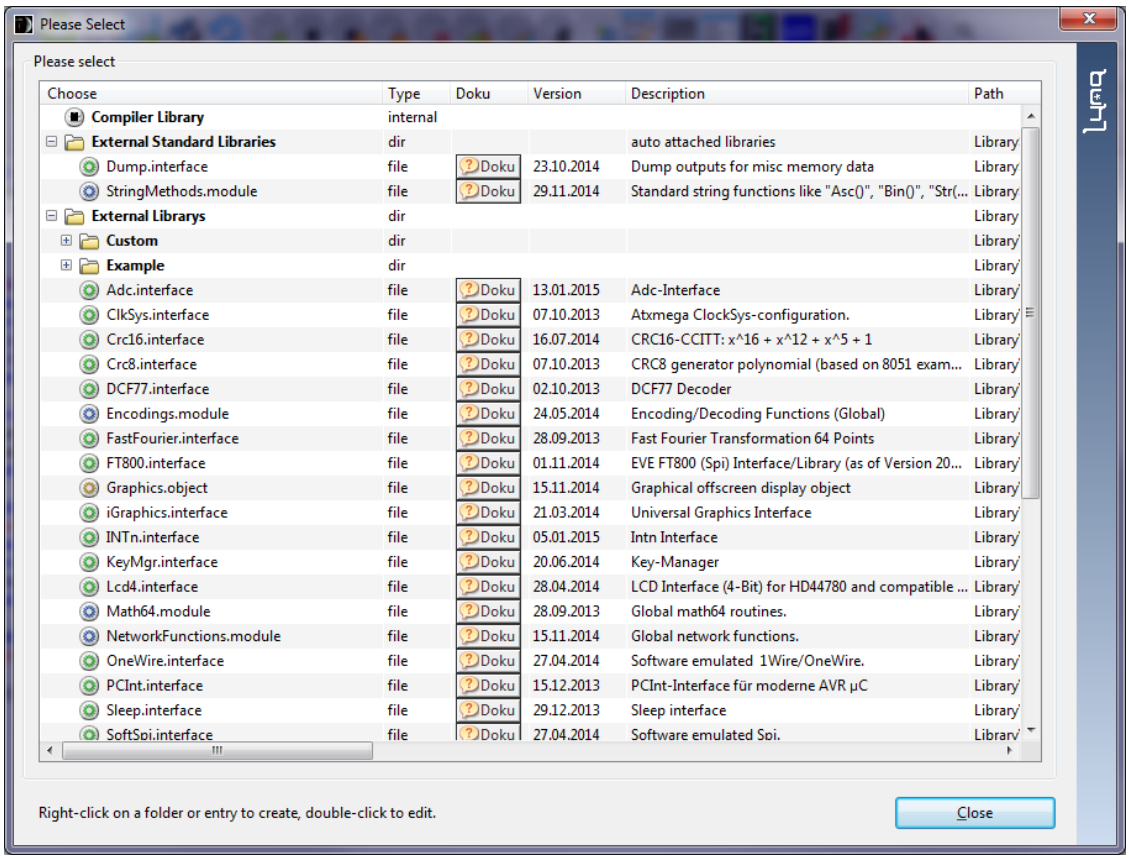
BIBLIOTHEKS-AUSWAHL

Siehe auch: **Externe Bibliotheken**

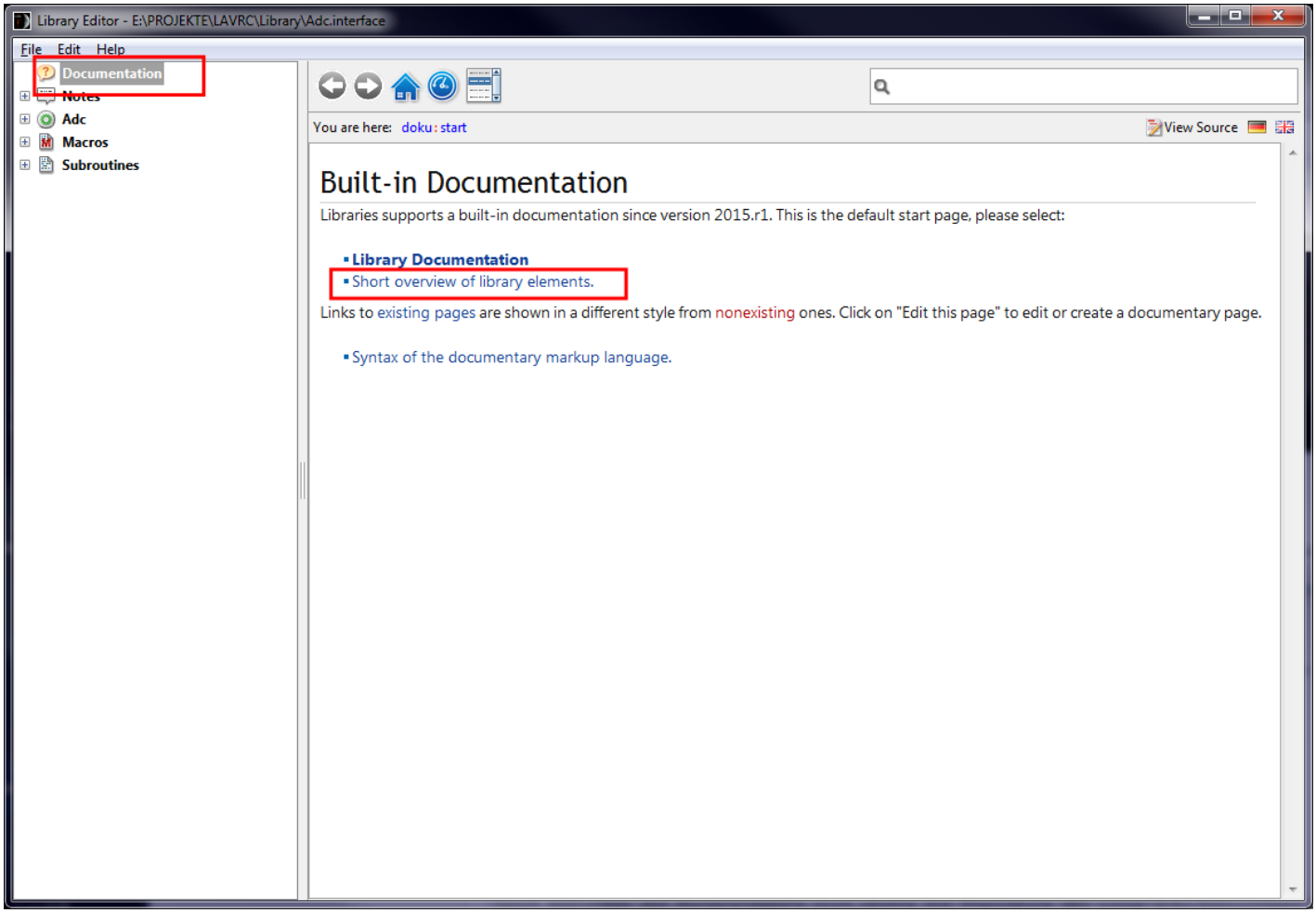
Eine Übersicht über die vorhandenen, externen Bibliotheken erhält man durch Klick auf den entsprechenden Button oder durch Anwahl des Menüpunktes *Bibliotheksbrowser/-Editor* im Menü *Werkzeuge*.



Anschließend öffnet sich die Bibliotheksauswahl:



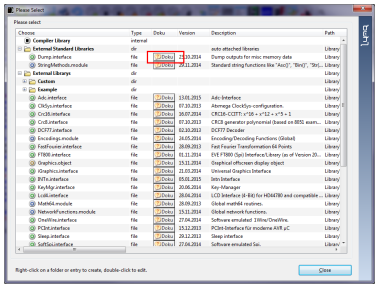
Ein Doppelklick auf den Namen öffnet den Bibliotheks-Editor.Durch Anklicken des Wurzelelements wird rechts die Startseite der Bibliotheks-Dokumentation angezeigt. Sie enthält einen Link auf die Seite der automatisch erzeugten Kurzübersicht.


Library Editor - E:\PROJEKTE\LAVRC\Library\Adc.interface
File Edit Help
Documentation
Notes
Adc
Macros
Subroutines
You are here: doku : start
View Source
Built-in Documentation
Libraries supports a built-in documentation since version 2015.r1. This is the default start page, please select:
Library Documentation
Short overview of library elements.
Links to existing pages are shown in a different style from nonexisting ones. Click on "Edit this page" to edit or create a documentary page.
Syntax of the documentary markup language.

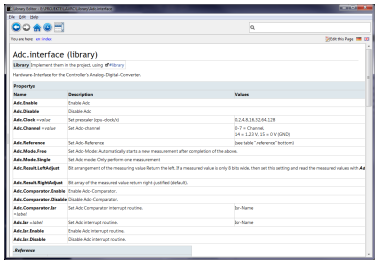
BIBLIOTHEKS-DOKUMENTATION

Die Bibliotheks-Dokumentationen befinden sich in jeder Bibliothek selbst.

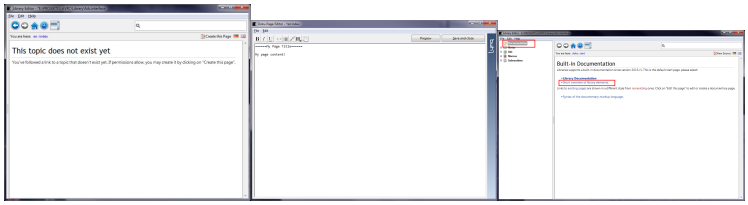
Zum Öffnen der Dokumentation einer Bibliothek klickt man aus der Bibliotheks-Auswahl auf den Knopf "doku". Der Bibliotheks-Editor öffnet sich dann im Dokumentationsmodus. Hierfür erscheint sofort die Dokumentations-Indexseite anstatt der Startseite.



Die Indexseite:



Mit den Landesflaggen rechts oben kann zwischen den verschiedenen Sprachen gewählt werden. Existiert eine Seite noch nicht, wird eine entsprechende Hinweisseite angezeigt. Durch klick auf "Quelltext anzeigen", "Diese Seite erstellen" bzw. "Diese Seite bearbeiten", kann eine Seite angezeigt, oder bearbeitet werden. **Eine Seite wird gelöscht, wenn sie leer gespeichert wird.** Die Beschreibung der Syntax findet man auf der Startseite. Die Startseite erreicht man durch einen Klick auf den "Home"-Knopf.



BIBLIOTHEKEN ERSTELLEN

GRUNDLAGEN

- **Grundlagen**
- **Namenskonventionen**
- **Abfolge des Einbindens von Bibliotheks-Code durch den Compiler/Assembler**
- **Nutzung von Funktionen aus der Standardbibliothek oder anderen Bibliotheken**

BESCHREIBUNG/TUTORIAL

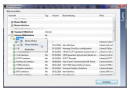
Es gibt folgende verschiedene Arten von Bibliotheken:

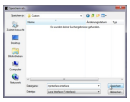
- **Modul** - Funktionssammlung
- **Interface** - Schnittstelle (statische "Klasse")
- **Object** - Dynamisches Speicherobjekt mit Funktionen.
- **Type** - Statisches Speicherobjekt (Struktur-Datentyp) mit Funktionen.

NEUE BIBLIOTHEK ANLEGEN (IDE)

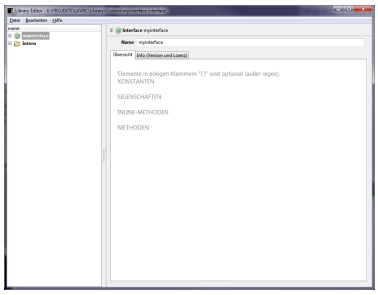
Zum Erstellen einer Bibliothek am Beispiel eines Interface, geht man wie folgt vor:

- 1. Library Browser öffnen.

- 2. Rechtsklick auf einen Eintrag und "Neues Interface" wählen.

- 3. In der erscheinenden Dateiauswahl den Ordner "Custom" öffnen (Ordner "/Library/Custom" im Luna-Verzeichnis).

- 4. Dem Interface einen eindeutigen Namen geben, z.B. "myinterface" und "speichern" wählen.


Das Interface wird nun angelegt und erscheint in einem neuen Bibliotheksedito-Fenster. In der Übersicht rechts lassen sich Name und Informationen des Interface editieren. Der Interface-Name ist der Basisname der im Luna-Code immer angegeben wird (vergleichbar mit ***Spi.[..]***). Durch einen Rechtsklick auf den Interfacenamen oder untergeordneten Einträgen links in der Liste, kann man nun Konstanten, Eigenschaften, Inlines, Methoden usw. anlegen.



BIBLIOTHEK ERSTELLEN - GRUNDLAGEN

Die externen Bibliotheken werden durch einen dafür extra vorhandenen, mächtigen Bibliotheks-Editor bearbeitet und/oder erstellt. Der Bibliotheks-Editor unterstützt den Autor einer Bibliothek grafisch durch Übersichten und leicht verständlichen Auswahlmöglichkeiten beim Erstellen von Methoden und Eigenschaften.

Die eigentlichen Bibliotheksfunktionen werden klassisch in **Assembler** programmiert.

In den Quelltexten der Interfaces und Module können Funktionen und Makros der Standardbibliothek verwendet werden. Mit ***.import*** kann ein externes Label importiert werden. Auch ist die Verwendung von Funktionen anderer Bibliotheken möglich. Durch die Verwendung der Präprozessorfunktion Defined() kann abgefragt werden, ob der Luna-Programmierer auch die notwendige Bibliothek eingebunden hat.

UM NAMENKOLLISIONEN ZU VERMEIDEN, MÜSSEN SÄMTLICHE BEZEICHNER VON DEFINES, KONSTANTEN, LABELS UND MAKROS INNERHALB EINER BIBLIOTHEK EINMALIG UND EINDEUTIG SEIN.

Es können sämtliche allgemeinen Präprozessorfunktionen des Luna-Assemblers genutzt werden wie in Inline-Assembler möglich, wie z.B. ***hi8()***, ***lo8()***, Schiebebefehle usw. Zusätzlich stehen spezielle Präprozessor-Befehle bereit, die nur bei der Erstellung von Bibliotheken Sinn machen (siehe unten).

Siehe auch:

- Präprozessor (Assembler)
- Standard-Defines (Namen) der Assemblerregister

CALLS

Aufrufe/Sprünge von/zu Bibliotheksinternen oder externen Methoden/Subroutinen sollte *immer* mit dem Befehl **call** bzw. **jmp** erfolgen, außer das Ziel befindet sich innerhalb der eigenen Methode/Subroutine. Der Optimierer des Assemblers optimiert hier automatisch zu den effizienteren indirekten Varianten **rcall** und **rjmp**. Auf Controllern die diese direkten Sprünge nicht unterstützen, werden **call** und **jmp** automatisch durch **rcall** und **rjmp** ersetzt.

SPEZIELLE PRÄPROZESSORFUNKTION(EN)

.REGISR

Diese besondere Direktive erlaubt das Registrieren eines Labels auf einen anzugebenden Interrupt-Vektor.

Syntax

- ***.regisr*** *vectorAddr, label*

```
;register the interrupt service routine to the timer-overflow-interrupt
;an error occours by the assembler, when the isr is already defined
.regisr  OVF0addr,__myIsr
```

MAKEIDENTIFIER(STRING)

Erstellt aus der übergebenen Zeichenkette einen Bezeichner (Symbol). Die Zeichenkette wird anschließend vom Assembler nicht mehr als Zeichenkette betrachtet, sondern als Konstantenname (Symbol). Wurde dieser Konstantenname bereits mit einem Wert belegt, kann er anschließend auch als Solcher im Assembler-Quelltext verwendet werden.

Beispiel

Zuweisung 0xf0 auf das bereits vorhande Register-Symbol *PORTB*. Die Konstante *myport* verweist dann auf die Konstante (Symbol) *PORTB*, sodass im Ergebnis hier der Wert von *PORTB* verwendet wird.

```
.set myport = MakeIdentifier("PORTB")
ldi  _HA0,0xf0
sts  myport,_HA0
```

Mit folgendem Code lässt sich beispielsweise in einem Interface ein bestimmter, im Luna-Code zugewiesener Port-Pin im Assemblerquelltext verwenden.

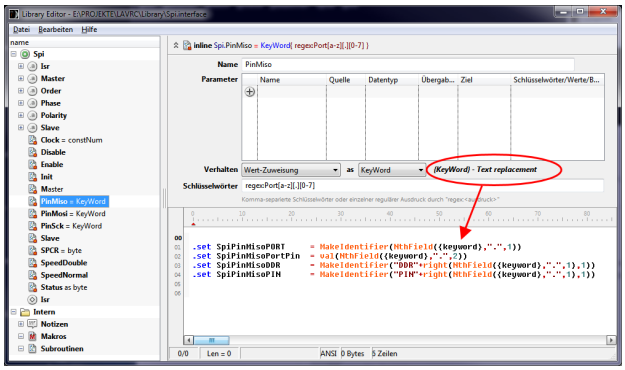
Luna-Code

```
Spi.PinMiso = PortB.2
```

Interface Inline-Methode "PinMiso"

```
.set SpiPinMisoPORT    = MakeIdentifier(NthField({keyword},".",1))
.set SpiPinMisoPortPin = val(NthField({keyword},".",2))
.set SpiPinMisoDDR   = MakeIdentifier("DDR"+right(NthField({keyword},".",1),1))
.set SpiPinMisoPIN   = MakeIdentifier("PIN"+right(NthField({keyword},".",1),1))
```

Im Library-Editor



Bei solchen Zuweisungen ist das ein wenig wie Textverarbeitung. Es wird die Eingabe des Programmierers so aufbereitet, dass man auf der Assemblerebene mit den Informationen sinnvoll arbeiten kann. Die Deklaration dieser Inline-Methode wiederum stellt sicher, dass der Programmierer ohne eine Fehlermeldung zu erhalten nur das eingeben kann, was er eingeben können soll. Dies wird hier durch den regulären Ausdruck sichergestellt.

In diesem Beispiel enthalten die vier Konstanten anschließend folgende Werte:

- **SpiPinMisoPORT =** PORTB
- **SpiPinMisoPortPin =** 2
- **SpiPinMisoDDR =** DDRB
- **SpiPinMisoPIN =** PINB

Assembler-Registerdefines

In Luna sind intern sowie für Inline-Assembler die Defines (Namen) für die Register 0-31 vorbelegt. Sie können in Inline-Assembler mit der Direktive ***.def*** nachträglich geändert werden.

Die Register 0-31 sind in Luna in 8 Register-Gruppen unterteilt. Luna nutzt keines der Register als ständig mitgeführten Status oder Pointer. Es können daher sämtliche Register zu jeder Zeit genutzt und verändert werden.

Die Registergruppen:

Gruppe	Register	Name
Low A	R0	_LA0
	R1	_LA1
	R2	_LA2
	R3	_LA3
Low B	R4	_LB0
	R5	_LB1
	R6	_LB2
	R7	_LB3
Temp 1	R8	_TMP0
	R9	_TMP1
	R10	_TMP2
	R11	_TMP3
Temp 2	R12	_TMPA
	R13	_TMPB
	R14	_TMPC
	R15	_TMPD
High A	R16	_HA0
	R17	_HA1
	R18	_HA2
	R19	_HA3
High B	R20	_HB0
	R21	_HB1
	R22	_HB2
	R23	_HB3
Low Pointer	R24	WL
	R25	WH
	R26	XL
	R27	XH
High Pointer	R28	YL
	R29	YH
	R30	ZL
	R31	ZH

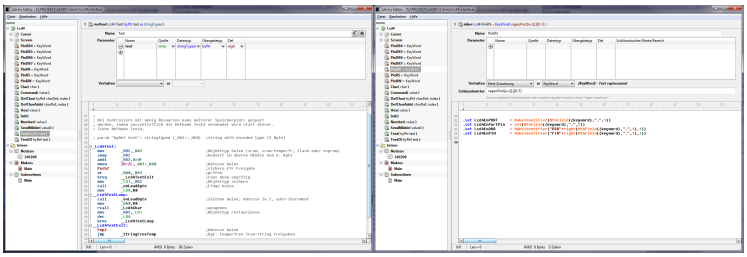
Nutzung der Gruppen

- **LA,LB,Temp1 und Temp2:** Werden für temporäre Daten innerhalb der internen Funktionen verwendet.
- **HA und HB:** Werden als allgemeine Arbeitsregister verwendet.
- **Low/High-Pointer:** Werden als TypeFlags, Array-Index, als Zwischenregister zum Verschieben auf den Stack, als klassische Pointer für den Zugriff auf Speicherbereiche, sowie beim Aufruf und innerhalb von internen Funktionen verwendet.

BIBLIOTHEK - NAMENSKONVENTIONEN

Die Namenskonventionen für Quelltext innerhalb einer Bibliothek sind:

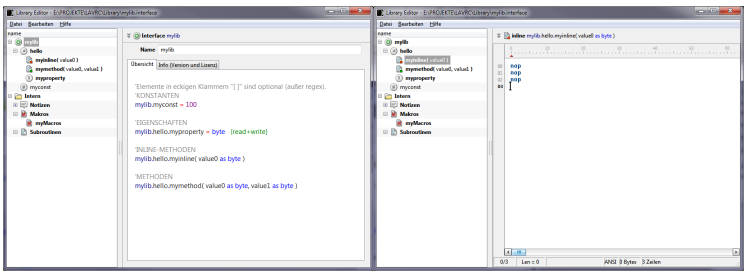
- **Konstanten/Defines (.def, .set, .equ):**
- **Haupt-Label: _[]**
- **Unter-Label: __[]**



ABFOLGE BIBLIOTHEKS-CODE

Der Compiler/Assembler bindet Inline-Code (den Source aus Inline-Methoden) direkt dort ein, wo er im Luna-Code textuell erscheint. Vor dem Einbinden werden Makros und Textersetzung bearbeitet.

Die Bibliotheks-Sourcen (Methoden und Subroutinen) werden in der Reihenfolge wie im Bibliothekseditor dargestellt zusammengefasst eingebunden. Hierbei werden unbenutzte Sourcen ausgeblendet.



```
avr.device = atmega328p
avr.clock = 20000000
avr.stack = 64

uart0.baud = 19200            ' Baudrate
uart0.Recv.enable             ' Senden aktivieren
uart0.Send.enable             ' Empfangen aktivieren

#library "library/mylib.interface"

dim a as byte

a = mylib.myconst
a = mylib.Hello.Myproperty

mylib.Hello.Myproperty = 100

mylib.Hello.myinline(200)
mylib.Hello.mymethod(1,2)

halt()
```

Der resultierende Assemblercode:

```
;{12}{ a = mylib.myconst } ----------------------------------------------
ldi   _HA0,0x64 ; 0b01100100 0x64 100
sts  dVarClassAvrA,_HA0
;{13}{ a = mylib.Hello.Myproperty } ---------------------------------
lds  _HA0,_dVarMylibHellomyproperty
sts  dVarClassAvrA,_HA0
;{15}{ mylib.Hello.Myproperty = 100 } ---------------------------------
ldi  _HA0,0x64 ; 0b01100100 0x64 100
sts  _dVarMylibHellomyproperty,_HA0
;{17}{ mylib.Hello.myinline(200) } ------------------------------------
ldi  _HA0,0xC8 ; 0b11001000 0xC8 200
nop
nop
nop
;{18}{ mylib.Hello.mymethod(1,2) } ------------------------------------
ldi  _HA0,0x02 ; 0b00000010 0x02 2
push  _HA0
ldi  _HA0,0x01 ; 0b00000001 0x01 1
rcall  _MylibHelloMymethod
;{22}{ halt() } ---------------------------------------------------------
__halt0:
rjmp  __halt0

[...]

_MylibHelloMymethod:
pop  _LA0
pop  _LA1
pop  _HB0
push  _LA1
push  _LA0
ret

[...]
```

Im Quelltext innerhalb einer Bibliothek können externe Label und Makros verwendet werden. Ein extra Import von Makros und Labels aus der Standardbibliothek ist nicht notwendig, da der Linker entsprechende Abhängigkeiten erkennen kann. Wenn eine Abhängigkeit nicht erkannt wird, kann der Import eines Labels durch

```
.import _externalLabel
```

angewiesen werden.
Bei der Verwendung von Code aus anderen externen Bibliotheken ist es notwendig zu prüfen, ob die Bibliothek vom Programmierer eingebunden wurde. Hierfür wird die Präprozessorfunktion defined() verwendet.

Luna

```
#if defined(Graphics.interface)
  'Programmcode
#endif
```

Assembler

```
.if defined(Graphics.interface)
  ;Programmcode
.endif
```

BIBLIOTHEKEN - MODUL ERSTELLEN

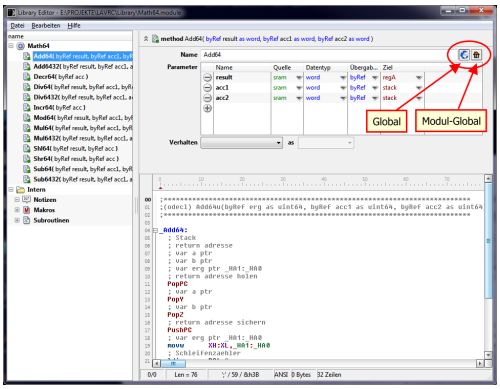
Eine Modul-Bibliothek ist vergleichbar mit einer *Funktionssammlung* (Sammlung von Funktionen). Bei einem Modul kann man je Funktion einzeln festlegen, wie im Luna-Code auf sie zugegriffen werden darf. Die beiden Varianten dafür sind

- **Global**
- **Modul-Global**

"Global" bedeutet, dass eine in einem Modul angelegte Funktion (Methode), direkt mit dem Funktionsnamen im Luna-Code verwendet werden kann. Die Verwendung wäre dann wie bei den fest eingebauten Luna-Funktionen. Module mit globalen Funktionen fügen also allgemeine Funktionen hinzu, die syntaktisch (schriftlich) von eingebauten Funktionen nicht unterschieden werden können.

"Modul-Global" bedeutet, dass die Methode nur in Verbindung mit dem Modulnamen verwendet werden kann. Beispielsweise ***Math64.Add()*** usw.

Die Verhaltensweise kann je Methode in der Deklaration separat eingestellt werden:




```
#library "Library/example.module" 'contains a global method 'mymethod(value as byte) as byte'
'[...]
a = mymethod(123)
```

MÖGLICHE ELEMENTE IN EINEM MODUL

- Inline
- Methode

- Notizen
- Makros
- Unterfunktionen

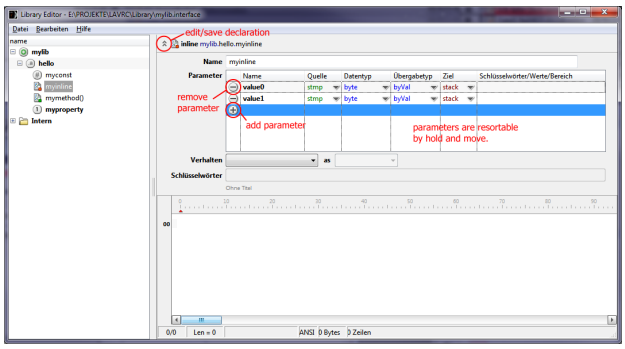
BEISPIEL-METHODEN

- Zwei numerische Parameter und Rückgabewert
- String-Parameter und Rückgabewert

Bibliothekselement - Inline

Der enthaltene Quelltext wird direkt an der Stelle der Verwendung eingesetzt. Es erfolgt *kein* Unterprogrammaufruf. Der erste Parameter bzw. eine echte Wertzuweisung erfolgt im Registerblock A, also Register _HA0 bis _HA3 (je nach Wertbreite). Alle Weiteren auf dem Stack. Alle weiteren Parameter werden in der Reihenfolge der Deklaration (von links nach rechts) vom Stack geholt.

Parameter



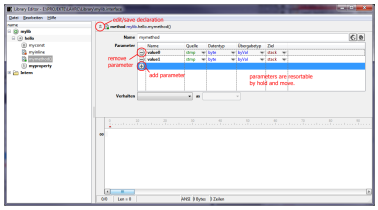
Hinweis

Wird eine echte Wertzuweisung (keine Textersetzung) in einer Methode/Inline-Methode verwendet, verdrängt Diese den Ggf. vorhanden ersten Parameter. D.h. der erste Parameter landet dann nicht mehr im Registerblock A sondern wie alle anderen Parameter auf dem Stack. Im Registerblock A steht dann die Wertzuweisung.

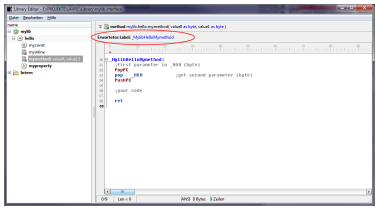
Der grafische Parametereditor ändert eine entsprechendes Ziel (Stack/RegA) automatisch.

Bibliothekselement - Methode

Der enthaltene Quelltext wird als *Unterprogramm* behandelt. Es erfolgt ein Aufruf des entsprechenden Labels als Unterprogrammaufruf. Der erste Parameter bzw. eine echte Wertzuweisung erfolgt im Registerblock A, also Register _HA0 bis _HA3 (je nach Wertbreite). Alle Weiteren auf dem Stack. Alle weiteren Parameter werden in der Reihenfolge der Deklaration (von links nach rechts) vom Stack geholt. Vor dem abholen der weiteren Parameter muss die Rücksprungadresse mit den fest eingebauten Assemblermakros "**PopPC**" und "**PushPC**" (siehe Standardbibliothek) vom und wieder auf den Stack zurückgelegt werden.



Bei einer Methode ist die Angabe des korrekten Labels notwendig. Mehr als ein Parameter (gezählt inkl. einer Ggf. definierten Zuweisung) wird vom Stack geholt:



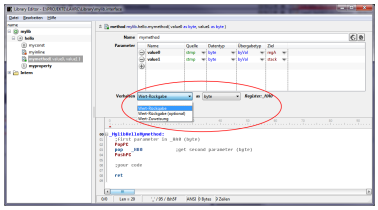
Hinweis

Wird eine echte Wertzuweisung (keine Textersetzung) in einer Methode/Inline-Methode verwendet, verdrängt Diese den Ggf. vorhanden ersten Parameter. D.h. der erste Parameter landet dann nicht mehr im Registerblock A sondern wie alle anderen Parameter auf dem Stack. Im Registerblock A steht dann die Wertzuweisung (siehe unten).

Der grafische Parametereditor ändert eine entsprechendes Ziel (Stack/RegA) automatisch.

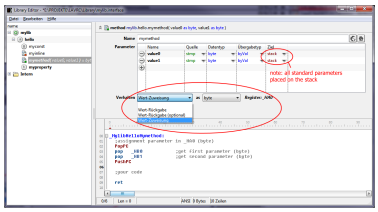
Methode mit Rückgabewert

Ein Rückgabewert wird im Registerblock A erwartet. Um eine Methode als Funktion mit Rückgabewert zu deklarieren, wählt man die gewünschte Möglichkeit und den Datentyp aus. Eine optionale Rückgabe ist ebenfalls möglich. Hier ignoriert der Compiler dann im Luna-Code den Rückgabewert, wenn nicht verwendet wird.



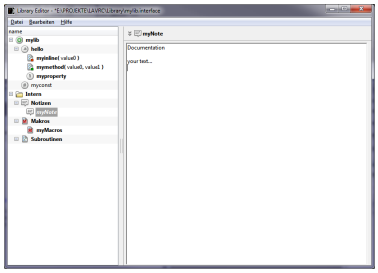
Methode mit Zuweisung

Alternativ kann eine Zuweisung konfiguriert werden. Hierbei ist zu beachten, dass dann die eigentlichen Parameter (sofern vorhanden) alle auf dem Stack abgelegt werden. Der Wert der Zuweisung wird im Registerblock A abgelegt.



Bibliothekselement - Notiz

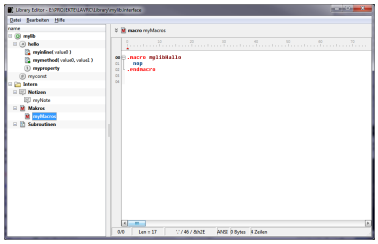
Notizen dienen dem Programmierer zur Dokumentation wichtiger Informationen.



Bibliothekselement - Makros

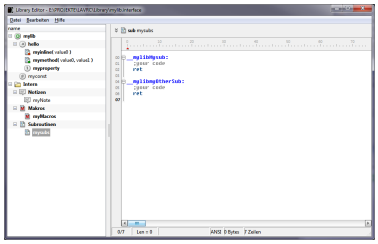
Makros sind Bibliotheksweit verfügbar und können zur besseren Übersicht im dafür vorgesehenen Ordner "Makros" gespeichert werden.

Wichtig: Einen eindeutigen Namen für Makros wählen, z.B. "mylib_myname"!



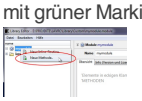
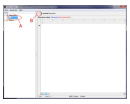
Bibliothekselement - Unterfunktionen

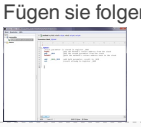
Unterfunktionen sind für den Luna-Programmierer nicht sichtbare, interne Bibliotheks-Methoden/Routinen, welche Bibliotheksweit verwendet werden können. Sie werden automatisch bei Nutzung einer verwendeten (Inline-)Methode eingebunden.



Modul Beispiel-Methode 1

Zwei Parameter mit Rückgabewert.

1. Öffnen sie das zuvor erstelle Modul "mymodule.module" im Bibliothekseditor.
2. Rechtsklicken sie auf "mymodul" in der Liste links und erstellen sie eine neue **Methode**. Es erscheint eine neue Methode "Namenlos" (Dokumentsymbol mit grüner Markierung).

3. Durch das Anklicken der Methode **(A)** erscheint rechts der Methodeneditor.

4. Klicken sie auf den Pfeile **(B)** links vom Symbol/Namen um den Parametereditor zu öffnen.
5. Ändern sie Name auf **myAdd**, fügen sie zwei Parameter hinzu, Ändern sie das Ziel des Ersten Parameter auf **regA** und wählen sie unter **Verhalten** den Eintrag **Wert-Rückgabe**.
6. Klappen sie den Parameter-Editor wieder zu mit Pfeilen oben links **(B)**.
7. Speichern sie die Änderungen mit Strg-S oder durch den entsprechenden Menüpunkt im Menü "Datei".

8. Unter "Erwartetes Label" Zeigt der Methodeneditor das vom Compiler erwartete Assembler-Label der Methode an.
9. Fügen sie folgenden Code in das Editierfeld ein und speichern sie die Änderungen


```
_MyAdd:
  ;first parameter is stored in register _HA0 by the compiler at call
  PopPC                 ;pop the method's return address from the stack
  pop    _HB0            ;pop the second parameter from the stack
  PushPC                ;push the methotd's return address back to the stack

  add    _HA0,_HB0       ;add both parameter, result in _HA0
  ret                   ;result already in register _HA0
```

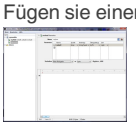
Ab sofort können sie nun das Modul und die Methode ***myAdd()*** in ihrem Luna-Code nutzen. Nach einem Neustart der IDE wird diese auch im Luna-Quelltext farblich hervorgehoben. Im Parameter-Editor der Methode haben sie die Möglichkeit zu wählen, ob man im Luna-Code den Modulnamen mit angeben muss oder nicht (siehe hier, "global" und "Modul-Global")

Testen sie die Methode. Sie addiert zwei Byte-Werte zu einem Byte-Wert.

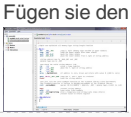
```
avr.device = atmega328p
avr.clock = 20000000
avr.stack = 64

uart0.baud = 19200  'baud rate
uart0.Recv.enable  'enable receive
uart0.Send.enable  'enable send

#library "library/custom/mymodule.module"


print "myAdd() = ";str(myAdd(23,42))


halt()
```

Modul Beispiel-Methode 2

1. Erstellen sie eine Methode wie im Ersten Beispiel mit dem Namen **mylen**.
2. Fügen sie einen String-Parameter und einen Rückgabewert wie im Bild hinzu.



3. Fügen sie den folgenden Code in das Editfeld ein und speichern sie die Änderungen.



```
;simple non-optimized all-memory-type string-length function

_MyLen:
  mov _HB2,_HA2  ;copy 3. byte (memory type encoded in upper nibble)
  swap _HB2   ;swap the upper nibble with lower nibble
  andi _HB2,0x0f  ;mask the lower nibble
  andi _HA2,0x0f  ;mask the lower nibble from 3. byte of string address

  ;string address now in _HA0,_HA1 and _HA2
  ;memory type now in _HB2

  ;now check for valid string address (>0)
  clr _LA0   ;clear a temp register
  mov _LA0,_HA0  ;move the 1. byte of string address
  or _LA0,_HA1
  or _LA0,_HA2
  breq __MyLenReturn ;if address is zero, break and return with value 0 (_HA0 is zero)

  movw ZH:ZL,_HA1:_HA0 ;copy the 1. and 2. byte to Z-Pointer
  PushZ     ;save the string address

  ;now just call the auto-Loadbyte-Function of the standard library (auto-increment!)
  ;the LoadByte-Function is located in MainObj/ooLoadByte of the standard library
  call _ooLoadByte  ;param: ZH:ZL[:RAMPZ] = Address, _HB2 = memory type, result in _LA0

  PopZ     ;restore string address
  push _LA0   ;save value from LoadByte
  call _StringFreeTemp ;free a temporary sram string (auto skips others)
  pop _HA0   ;restore value to return register

__MyLenReturn:
  ret
```

Testen sie die Methode. Diese Methode gibt die Länge eines Strings zurück, egal ob er im Arbeitsspeicher, Flash oder Eeprom liegt. Es sind also alle Varianten eines Strings als Parameter erlaubt, auch Ergebnisse eines String-Ausdrucks. Ein Ggf. erzeugter temporärer String im Arbeitsspeicher (z.B. aus einem Ausdruck) wird automatisch freigegeben.

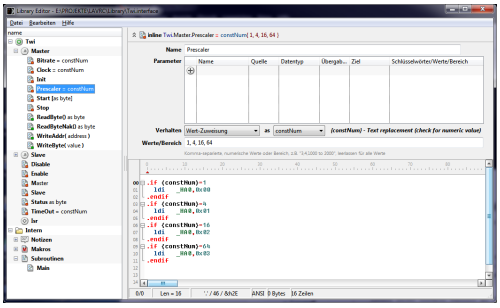
```
avr.device = atmega328p
avr.clock = 20000000
avr.stack = 64

uart0.baud = 19200  'baud rate
uart0.Recv.enable  'enable receive
uart0.Send.enable  'enable send

#library "library/custom/mymodule.module"


print "myAdd() = ";str(myAdd(23,42))    ' = 65
print "myLen() = ";str(myLen("Hello World")) ' = 11

halt()
```

BIBLIOTHEKEN - INTERFACE ERSTELLEN

Eine Interface-Bibliothek ist vergleichbar mit einer Klasse in Luna. Alle in einem Interface enthaltenen Elemente sind innerhalb des Interface eingeschlossen und Funktionen können von Eigenschaften innerhalb eines Interface abhängig sein. Beispielsweise die Zuweisung eines Portpins vor dem Aufruf einer Initialisierungsfunktion und anschließender Verwendung von Funktionen.

Zumeist werden Interfaces klassisch als Schnittstelle (Interface) zur Ansteuerung von Hardware-Komponenten verwendet. Weitere Einsatzmöglichkeiten sind jede Art von in sich geschlossenen Anwendungen, bspw. eines in Software emulierten Protokolls, oder einer gekapselten mathematischen Funktion mit vorherigen Eingabeparametern. Gegenüber den normalen Klassen in Luna können bei Interfaces auch besondere Eigenschaften zugewiesen und/oder abgefragt werden wie z.B. Port-Pins oder eine von einem Schlüsselwort abhängige Funktion, vergleichbar mit den in Luna eingebauten Interfaces für z.B. Ports, Timer o.Ä.

Folgendes Bild zeigt eine Inline-Methode, die nur die Zuweisung von ganz bestimmten Konstantwerten erlaubt:



```
#library "Library/example.interface"
'[...]
example.myaction(123)
```

MÖGLICHE ELEMENTE IN EINEM INTERFACE

- Bibliothekselemente und resultierende Luna-Syntax

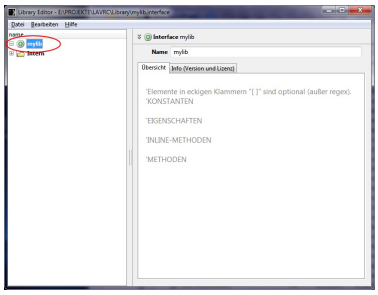
- Schlüsselwort
- Konstante
- Eigenschaft
- Inline
- Methode

- Notizen
- Makros
- Unterfunktionen

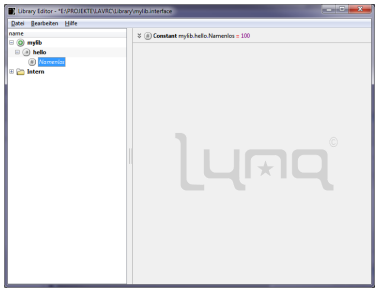
Elemente und resultierende Luna-Syntax

Die Art und Ordnung der Elemente in einer ***Interface-Bibliothek*** bestimmen die resultierende Luna-Syntax. Grundsätzlich ist dies vergleichbar mit einem Dateipfad.

Nach dem Erstellen einer Bibliothek, existiert nur ein Wurzelelement:



Die Ordnung und Gruppierung bestimmt, wie der Zugriff auf ein Element im Luna-Code erfolgt. Die Elemente sind automatisch im Autovervollständigen für den Luna-Programmierer verfügbar:

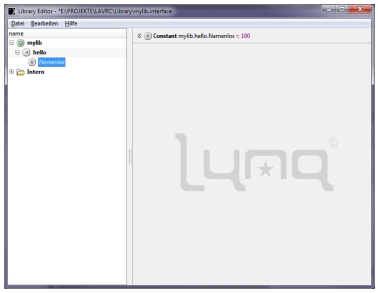


Der Zugriff auf diese Konstante wäre im Luna-Code:

```
a = mylib.hello.Namenlos
```

Bibliothekselement - Schlüsselwort

Ein Schlüsselwort ist wie ein Ordner zu verstehen. Es *gruppiert* gewünschte Funktionen *syntaktisch* und hat sonst keine weitere Funktion.



Bibliothekselement - Konstante

Konstanten in Bibliotheken dienen dazu, dem Programmierer vorbelegte Werte verfügbar zu machen.

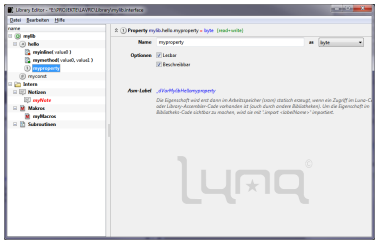
Erstellte Bibliothekskonstanten sind innerhalb der Bibliothek ansprechbar.Der Name einer Konstante innerhalb des Assemblerquelltextes setzt sich zusammen aus

- _

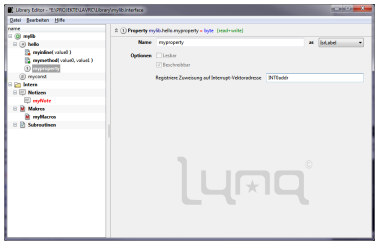
Eine in einer Bibliothek erstellten Konstante mit dem Namen "myValue" ist daher unter dem Namen "MYLIB_MYVALUE" ansprechbar.

Bibliothekselement - Eigenschaft

Eigenschaften können vom Luna-Programmierer - je nach Konfiguration wie Variablen lesend und schreibend genutzt werden. Innerhalb des Quelltextes einer Bibliothek, kann man auf Variablen zugreifen. Das entsprechende Label wird auf der Bearbeitungsseite der Eigenschaft angezeigt.



Die Zuweisung eines Interrupt-Vektors kann hier ebenfalls als Sonderform konfiguriert werden. Der Interrupt wird dann entsprechend mit der vom Luna-Programmierer zugewiesenen ISR-Methode belegt.

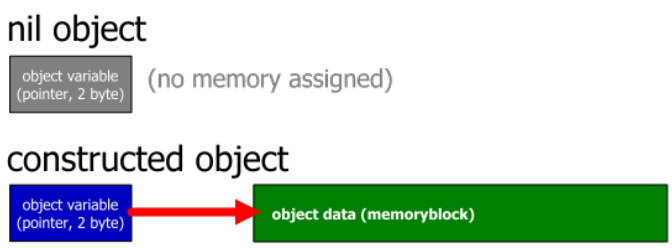


BIBLIOTHEKEN - OBJECT ERSTELLEN

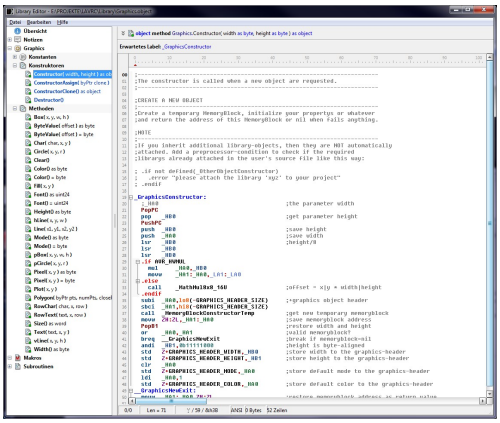
Implementiert ab Version	2015.r1

Eine Objekt-Bibliothek ist eine instanzierbare Klasse in Form einer Bibliothek. Sie fügt einen neuen Objekt-Datentyp hinzu, welcher erzeugt/instanziert werden muss, bevor man ihn verwenden kann. Der in Luna fest eingebaute Typ "MemoryBlock" ist z.B. ein solcher Objekt-Datentyp. "String" ist ebenfalls ein Objekt-Datentyp, der vom Objekt "MemoryBlock" abgeleitet ist.
Das Library-Objekt kann je nach Gestaltung wie z.B. auch ein MemoryBlock entsprechende Funktionen zur Verfügung stellen, die auf das Objekt angewendet werden können. Jedes Objekt besitzt nach dem Erzeugen seinen eigenen, lokalen *dynamischen* Speicher. Objekte können demnach mehrfach zur gleichen Zeit im Speicher existieren. Die Anzahl der möglichen parallelen Instanzen ist nur duch den Arbeitsspeicher begrenzt.

Ein Objekt wird in einem MemoryBlock gespeichert, daher belegt es bis auf die eigentliche Variable keinen Speicherplatz solange es nicht erzeugt/initialisiert wurde. Es kann während der Programmlaufzeit also *konstruiert* und *zerstört* werden.



Das vormals fest eingebaute Objekt "Graphics" wurde in ein Library-Objekt ausgelagert. Es dient auch als Beispiel für eigene Implementationen. Weiterhin finden sie im Ordner /Library/Example/ die Bibliothek *xstring.object*, welche einen eigenes String-Objekt zur Verfügung stellt.



Im Gegensatz zu einem Interface oder Modul, kann in einem Objekt keine verschachtelte Syntax mit Schlüsselwörtern definiert werden. Die Syntax ergibt sich aus der Funktionalität des Objekts und der möglicherweise zusätzlich implementierten weiteren Objekte und deren Funktionen.

In einem Library-Objekt gibt es nur *eine Basisebene* in welcher sich die eigentlichen Funktionen (Methoden) befinden, die der Luna-Programmierer nutzen kann.

Im Verzeichnis *Konstruktoren* befinden sich die *notwendigen* Objekt-Handler (Basisfunktionen) die jedes Objekt aufweisen muss, damit der Compiler das Objekt erzeugen, zerstören, zuweisen oder in Ausdrücke einbetten kann.

Für die Nutzung des Objekts in Ausdrücken mit Operatoren - z.B. "+", "-", "and", wird die Funktionalität "Operator-Überladen" unterstützt. D.h. es ist optional möglich Handler (Basisfunktionen) für Operatoren zu erstellen, sodass man z.B. mit zwei verschiedenen Objekten eine Addition o.Ä. durchführen kann.

```
#library "Library/Graphics.object"
'[...]
dim g as Graphics

g = new Graphics(64,32)
if g<>nil then
  g.Font=font.Addr  'set font
  g.Text("Hello",1,1)
end if

'[...]

#includeData font,"fonts\vdi6x8.lf"
```

ELEMENTE IN EINEM OBJECT

- Konstanten
- Konstruktoren
- Operator-Handler
- Methode
- Inline

- Notizen
- Makros
- Unterfunktionen

ANWENDUNGSBEISPIEL

Siehe hierzu Beispiel-Bibliothek ***xstring.object*** im Ordner */Library/Example/* .Die Beispiel-Bibliothek ist mit entsprechenden Beschreibungen ausgestattet. Beim Anlegen z.B. einer Methode oder eines Konstruktors ist automatisch eine Kurzerklärung vorhanden.

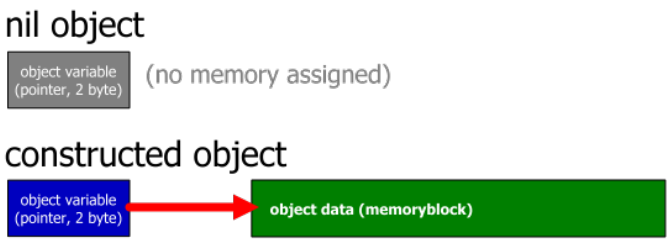
Code-Beispiel:

```
#library "Library/Example/xstring.object"

avr.device = atmega328p
avr.clock = 20000000
avr.stack = 96

uart.baud = 19200    ' Baudrate
uart.recv.enable    ' Senden aktivieren
uart.send.enable    ' Empfangen aktivieren

dim value as word
dim a,b,r as xstring

print 12
wait 1

a = new xstring("This is")
b = new xstring(" just ")

'the address of the objects data blocks (memoryblock)
print "a.Ptr = 0x";hex(a.Ptr)
print "b.Ptr = 0x";hex(b.Ptr)

'the content of the objects
print "a = ";34;a;34
print "b = ";34;b;34

r = a + b + "amazing!"

print "r.Ptr = 0x";hex(r.Ptr)
print "r = ";34;r;34

'calling a function of the object
print "r.reverse = ";34;r.reverse;34

'calling a function who returns a object
print "return value of function 'test()' = ";34;test();34

'destroy a object
r = nil


halt()

'object as return value
function test() as xstring
  return new xstring("test")
endfunc
```

Bibliotheken - Object Konstruktor

Ein Konstruktor für eine Object-Bibliothek ist eine Methode die zum Erzeugen eines Objekts benötigt wird. Die Konstruktor-Methode wird vom Compiler aufgerufen, wenn ein Objekt erzeugt/initialisiert werden soll. Bei Objekt-Bibliotheken ist mindestens ein *Konstruktor* zum Erzeugen (Constructor) notwendig. Die Konstruktoren zur *Zuweisung* auf eine Objekt-Variable (ConstructorAssign) oder *Kopie* (ConstructorClone), sowie der *Destruktor* (Destructor) sind optional.

Wird eine Variable vom Typ der Objekt-Bibliothek dimensioniert, ist sie undefiniert (nil). Es handelt sich - wie bereits beschrieben - um einen Zeiger auf einen dynamischen Speicherblock (MemoryBlock). Sobald ein Objekt mit dem ***new***-Operator erzeugt wurde, zeigt diese Variable auf den nun zugewiesenen Speicherblock.



Wieviel Speicher ein solches Objekt nach dem Erzeugen belegt, wird durch den Autor der Objekt-Bibliothek festgelegt.

Ein Konstruktor zum Erzeugen (new) muss daher enthalten:

1. Freigabe eines Ggf. bereits zugewiesenen MemoryBlock.
2. Erzeugen eines MemoryBlock mithilfe der Funktionen aus der Standardbibliothek in gewünschter Größe.
3. Initialisieren des Speicherblocks mit den gewünschten Daten.
4. Rückgabe der Adresse des neuen MemoryBlock oder nil bei einem Fehler.

Konstruktor-Arten

Bei einer Objekt-Bibliothek gibt es drei verschiedene Arten von Konstruktoren, welche je nach Art der Verarbeitung aufgerufen werden (Erzeugen, Zuweisen, Kopieren).

1. ***Constructor*** - Wird aufgerufen zum Erzeugen eines Objekts: ***new*** *myobject(...)*. Das Erzeugte Objekt liegt dann temporär im Speicher und ist keiner Variablen zugewiesen. (mehr als ein Konstruktor erlaubt)
2. ***ConstructorAssign*** - Wird aufgerufen bei einer Zuweisung: *myobj1* = . Hiermit wird ein temporär erzeugtes Objekt einer Variablen zugewiesen.
3. ***ConstructorClone*** - Wird aufgerufen wenn ein bestehendes Objekt instanziert (geklont/kopiert) werden soll: *myobj1* = *myobj2*. Hierzu wird ein neues, temporäres Objekt erzeugt und die Daten aus dem Quellobjekt werden kopiert. Bei der Zuweisung von ***nil*** wird automatisch der Destructor aufgerufen.

Parameter eines Konstruktors

Die Parameter die für einen Konstruktor definiert werden, steuern wie und welche Art von Eingabedaten zum Erzeugen eines Objekts möglich sind. Da mehrere Konstruktoren angelegt werden dürfen, können sich Diese auch unterscheiden.

Beispiel 1

Constructor()

```
dim var as myobj
var = new myobject()
```

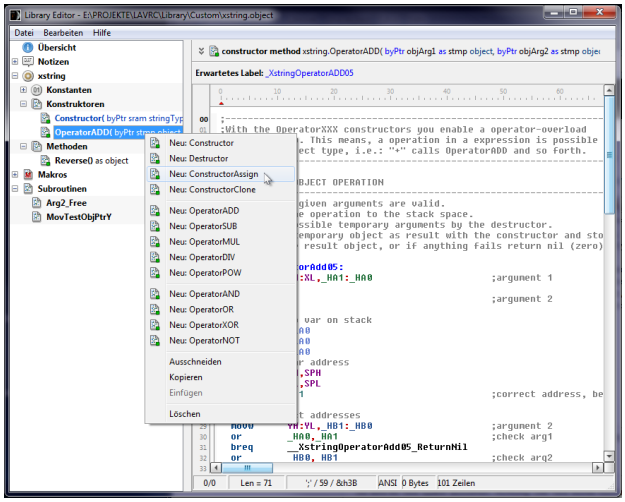
Beispiel 2

Constructor(size as byte)

```
dim var as myobj
var = new myobject(123)
'following term selects and calls automatically the
'best-fit constructor with the value as parameter (like above):
var = 123
```

Bibliotheken - Operator-Handler

Ein Operator-Handler für eine TYPE- oder OBJECT-Bibliothek ist eine (Inline-) Methode, welche die Funktion für einen Operator zur Verfügung stellt. Ein Operator-Handler wird aufgerufen/eingebettet, wenn ein Objekt durch einen Operator, z.B. "+" verarbeitet werden soll.



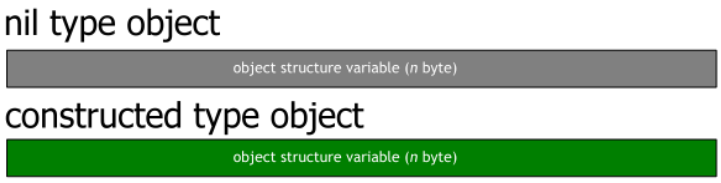
Verwendung

Die Verarbeitung von Type und Object Operator-Handlern sind unterschiedlich. Wenn sie einen neuen Operator-Handler anlegen, wird automatisch eine entsprechende Beschreibung eingefügt. Die Beschreibung erklärt den Aufbau und die Verwendung eines Operator-Handlers.

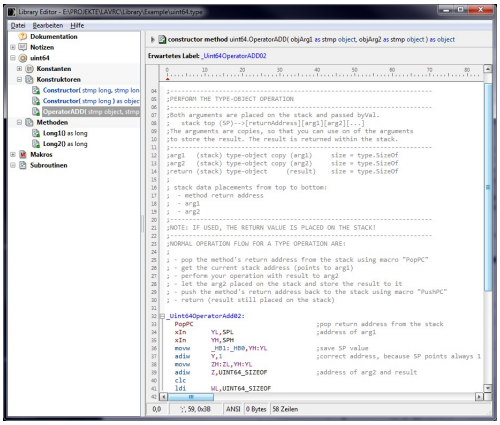
BIBLIOTHEKEN - TYPE ERSTELLEN

Implementiert ab Version	2015.r1

Eine Type-Bibliothek ist eine statisches Speicherobjekt mit Funktionen in Form einer Bibliothek. Sie fügt einen neuen Struktur-Datentyp hinzu. Eine Type-Bibliothek (Type-Objekt) ist vergleichbar mit einer Struktur in Luna, welches zusätzliche Funktionen bereitstellen, sowie in auch Ausdrücken verwendet werden kann.



Das Type-Objekt kann daher je nach Gestaltung entsprechende Funktionen zur Verfügung stellen, die auf das Objekt angewendet werden können. Jedes Objekt besitzt nach dem Erzeugen seinen eigenen, lokalen *statischen* Speicher. Type-Objekte verhalten sich im Speicher wie dimensionierte Variablen (lokal/temporär oder global).



Im Gegensatz zu einem Interface oder Modul, kann in einem Objekt keine verschachtelte Syntax mit Schlüsselwörtern definiert werden. Die Syntax ergibt sich aus der Funktionalität des Objekts und der möglicherweise zusätzlich implementierten weiteren Objekte und deren Funktionen.

In einem Type-Objekt gibt es nur *eine Basisebene* in welcher sich die eigentlichen Funktionen (Methoden) befinden, die der Luna-Programmierer nutzen kann.

Im Verzeichnis *Konstruktoren* befinden sich die *optionalen* Objekt-Handler (Basisfunktionen) die jedes Type-Objekt aufweisen kann, damit der Compiler das Objekt initialisieren, zuweisen oder in Ausdrücke einbetten kann. ***Konstruktoren*** sind bei Type-Objekten optional. Sind keine vorhanden, wird ein interner Standardkonstruktor verwendet. Ein ***Destructor*** ist nicht vorhanden, da der Speicher eines Type-Objekts wie eine (Struktur-) Variable dimensioniert ist.

Für die Nutzung des Objekts in Ausdrücken mit Operatoren - z.B. "+", "-", "and", wird die Funktionalität "Operator-Überladen" unterstützt. D.h. es ist optional möglich Handler (Basisfunktionen) für Operatoren zu erstellen, sodass man z.B. mit Type-Objekten eine Addition o.Ä. durchführen kann.

ELEMENTE IN EINEM TYPE-OBJECT

- Konstante
- Konstruktoren
- Operator-Handler
- Methode
- Inline

- Notizen
- Makros
- Unterfunktionen

ANWENDUNGSBEISPIEL

Siehe hierzu Beispiel-Bibliothek ***unt64.type*** im Ordner */Library/Example/* .Die Beispiel-Bibliothek ist mit entsprechenden Beschreibungen ausgestattet. Beim Anlegen z.B. einer Methode oder eines Konstruktors ist automatisch eine Kurzerklärung vorhanden.

Code-Beispiel:

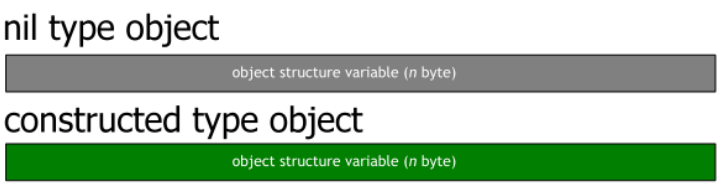
```
#library "Library/Example/uint64.type"

avr.device = atmega328p
avr.clock = 20000000
avr.stack = 96

uart.baud = 19200    ' Baudrate
uart.recv.enable    ' Senden aktivieren
uart.send.enable    ' Empfangen aktivieren

dim value as word
dim a,b,r as uint64

print 12
```

```
a = 0x30303030

a = new uint64(0x30303030,0x30303030)
b = new uint64(0x31313131,0x31313131)

'print the values of the objects using the object's functions
print "value of a = 0x";hex(a.long2);hex(a.long1)
print "value of b = 0x";hex(b.long2);hex(b.long1)

'math operation with this objects
print "math operation 'r = a + b + new uint64(0x32323232,0x32323232)'"
r = a + b + new uint64(0x32323232,0x32323232)
print "result: r = 0x";hex(r.long2);hex(r.long1);" - expected: 0x9393939393939393"

print "call method 'test1(r,a,b)'"
test1(r,a,b)
print "result: r = 0x";hex(r.long2);hex(r.long1);" - expected: 0x6161616161616161"

halt()


'argument test
procedure test1(byRef r as uint64, byRef a as uint64, byRef b as uint64)

  r = a + b

endproc
```

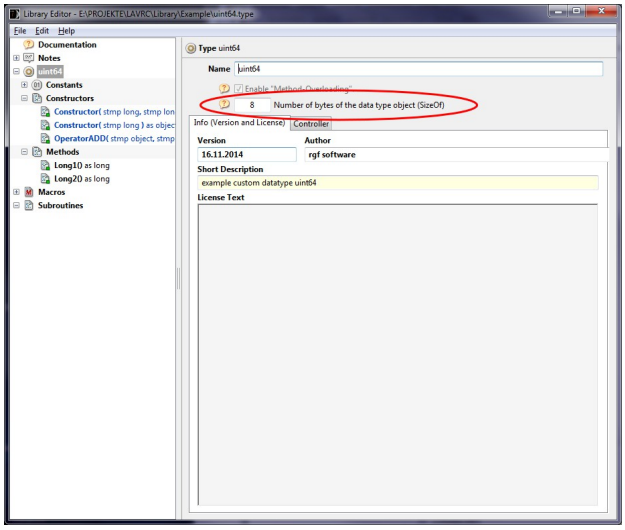
Bibliotheken - Type Konstruktor

Ein Konstruktor für eine Type-Bibliothek ist eine (Inline-) Methode die zum Erzeugen eines Objekts verwendet wird. Der Konstruktor wird aufgerufen/eingebettet, wenn ein Objekt erzeugt/initialisiert werden soll. Bei Type-Bibliotheken ist ein Konstruktor zum Erzeugen (Constructor) *optional*, da der Speicher des Objekts wie eine (Struktur-) Variable dimensioniert ist.

Wird eine Variable vom Typ der Type-Bibliothek dimensioniert, belegt sie daher bereits den Speicherplatz in der festgelegten Größe.



Wieviel Speicher ein solches Objekt belegt, wird durch den Autor der Type-Bibliothek festgelegt.



Konstruktor-Arten

Bei einer Type-Bibliothek gibt es nur einen optionalen Standardkonstruktor (Constructor).

1. ***Constructor*** - Wird aufgerufen zum Erzeugen eines Objekts: ***new*** *myobject(...)*. Das Erzeugte Objekt liegt dann temporär im Speicher (auf dem Stack) und ist keiner Variablen zugewiesen. Es ist mehr als ein Konstruktor erlaubt.

Die Zuweisung auf eine Variable oder das Kopieren (Klonen) wird durch intern eingebaute Funktionen automatisch vorgenommen.

Parameter eines Konstruktors

Die Parameter die für einen Konstruktor definiert werden, steuern wie und welche Art von Eingabedaten zum Erzeugen eines Objekts möglich sind. Da mehrere Konstruktoren angelegt werden dürfen, können sich Diese auch unterscheiden.

Beispiel 1

Constructor()

```
dim var as myobj
var = new myobject()
```

Beispiel 2

Constructor(size as byte)

```
dim var as myobj
var = new myobject(123)
'following term selects and calls automatically the
'best-fit constructor with the value as parameter (like above):
var = 123
```

STDLIB.INTERFACE (STANDARD-BIBLIOTHEK)

Compiler-Standardbibliothek (automatisch eingebunden).

Die Standardbibliothek *StdLib.interface* enthält alle Compiler-intern verwendeten Deklarationenund Funktionen. Ohne Standardfunktionen kann der Compiler kein funktionierendes Kompilat erzeugen.**An der Standardbibliothek sollten sie daher keine Änderungen durchführen, außer sie wurden durcheine Errata oder durch einen Supporthinweis ausdrücklich empfohlen.**

Die Standardbibliothek enthält fest eingebauten Komponenten:

- Eingebaute Objekte

Darüberhinaus stellt sie *globale* Funktionen und Makros bereit, die auch in externen Bibliotheken oder Inline-Assembler genutzt werden können.

- Liste der globalen Konstanten
- Liste der öffentlichen globalen Makros
- Liste der öffentlichen globalen Funktionen

LISTE DER KONSTANTEN/SYMBOLE

In dieser Liste sind globale Konstanten und Symbole aufgeführt, welche in Assembler-Sourcecode in Bibliotheken oder Luna-Inline-Asm verwendet werden können.

KONSTANTEN

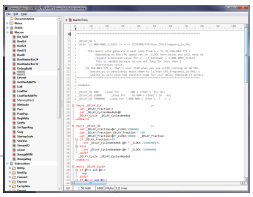
Name	Beschreibung
AVR_ADDRWRAP	Ungleich Null wenn Adress-Wrap unterstützt (Relative Sprünge über Flash-Ende auf Anfang und umgedreht).
AVR_CORE	AVR Core-Typ (Zahl)
AVR_CLOCK, _CLOCK	Definierte Taktrate
AVR_CODE_START_ADDR	Definierte Code-Startadresse im Flash.
AVR_DEVICE, DEVICE	Controllerbezeichnung (Zeichenkette)
AVR_EEPROM_ACCESS	Ungleich Null wenn Eepromzugriffe im Luna-Source erfolgen.
AVR_EEPROM_ACCESS_HIGH	Ungleich Null wenn der Eeprom-Speicher größer 255 Bytes ist.
AVR_HWMUL	Ungleich Null wenn Hardware-Multiplikation unterstützt.
AVR_HWJMP	Ungleich Null wenn direkte Sprünge/Aufrufe unterstützt (call,jmp,..)
AVR_MEGA	Ungleich Null wenn Atmega-Controller
AVR_XMEGA	Ungleich Null wenn Atxmega-Controller
AVR_PC_SIZE	Anzahl Bytes die bei einem Unterprogrammaufruf auf dem Stack als Rücksprungadresse abgelegt werden.
AVR_STACK_SIZE	Definierte Stackgröße
AVR_STACK_END	Stack-Endadresse, zeigt auf letztes Stack-Byte (SRAMEND-AVR_STACK_SIZE).

SYMBOLE

Die folgenden Symbole sind nur dann vorhanden, wenn die entsprechende Eigenschaft zutrifft. Sie sind mittels derPräprozessorfunktion ***defined()*** auf Vorhandensein zu prüfen und *nicht auf Wert*!

Name	Beschreibung
AVR_PC3	3-Byte-Programmcounter (PC). z.B. direkte (Rücksprung-)Adressen (call/ret) haben eine Größe von 3 Byte.

GLOBALE MAKROS (STDLIB.INTERFACE)



Liste der global verfügbaren Makros, die in externen Bibliotheken oder Inline-Assembler genutzt werden können bzw. müssen. Die Einträge der Liste beziehen sich auf die Ordnerstruktur in der Bibliothek (siehe Bild). Die Ggf. erwarteten Parameter oder benutzten Register entnehmen sie bitte der Beschreibung bzw. dem Code innerhalb des jeweiligen Eintrags.

MAKROS/DELAY/

- **_DELAY_CLK** - *Wartefunktion Anzahl Cycles (Takte)*
- **_DELAY_US** - *Wartefunktion Anzahl Mikrosekunden*

MAKROS/EXTEND/

- **Extend(Group,SourceType,DestinationType)** - *Universelle Konvertierung (Casting) von numerischen Werten in einer Registergruppe.*

MAKROS/PUSHPOP/

- **PopPC, PushPC** - *Rücksprungadresse vom/auf Stack (Automatisch 2 oder 3 Byte-Programmcounter).*

- **PopLA, PushLA** - *Registergruppe LA (_LA3::_LA0) vom/auf Stack (4 Byte).*
- **PopLA1, PushLA1** - *Registergruppe LA-Low(_LA1::_LA0) vom/auf Stack (2 Byte).*
- **PopLA2, PushLA2** - *Registergruppe LA-High (_LA3::_LA2) vom/auf Stack (2 Byte).*

- **PopLB, PushLB** - *Registergruppe LB (_LB3::_LB0) vom/auf Stack (4 Byte).*
- **PopLB1, PushLB1** - *Registergruppe LB-Low(_LB1::_LB0) vom/auf Stack (2 Byte).*
- **PopLB2, PushLB2** - *Registergruppe LB-High (_LB3::_LB2) vom/auf Stack (2 Byte).*

- **PopA, PushA** - *Registergruppe A (_HA3::_HA0) vom/auf Stack (4 Byte).*
- **PopA1, PushA1** - *Registergruppe A-Low(_HA1::_HA0) vom/auf Stack (2 Byte).*
- **PopA2, PushA2** - *Registergruppe A-High (_HA3::_HA2) vom/auf Stack (2 Byte).*
- **PopA24, PushA24** - *Registergruppe A 24-Bit (_HA2::_HA0) vom/auf Stack (3 Byte).*

- **PopB, PushB** - *Registergruppe B (_HB3::_HB0) vom/auf Stack (4 Byte).*
- **PopB1, PushB1** - *Registergruppe B-Low(_HB1::_HB0) vom/auf Stack (2 Byte).*
- **PopB2, PushB2** - *Registergruppe B-High (_HB3::_HB2) vom/auf Stack (2 Byte).*
- **PopB24, PushB24** - *Registergruppe B 24-Bit (_HB2::_HB0) vom/auf Stack (3 Byte).*

- **PopA2Z, PushA2Z** - *Register (ZH::_HA0) vom/auf Stack (16 Byte).*
- **PopAll, PushAll** - *Alle Register vom/auf Stack (34 Byte, im Loop: kurze Version).*
- **PopAllFast, PushAllFast** - *Register (ZH::_HA0) vom/auf Stack (34 Byte, direkt - schnell).*

- **PopW, PushW** - *Pointer-Register W (WH:WL) vom/auf Stack (2 Byte).*
- **PopX, PushX** - *Pointer-Register X (XH:XL) vom/auf Stack (2 Byte).*
- **PopY, PushY** - *Pointer-Register Y (YH:YL) vom/auf Stack (2 Byte).*
- **PopZ, PushZ** - *Pointer-Register Z (ZH:ZL) vom/auf Stack (2 Byte).*

MAKROS/SETTYPEREG/

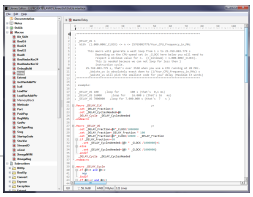
- **SetTypeReg_SramTemp** - *Objekttyp in Register _HA2 setzen für SRAM/STMP (Rückgabeparameter).*
- **SetTypeReg_Flash** - *Objekttyp in Register _HA2 setzen für FLASH (Rückgabeparameter).*
- **SetTypeReg_Eeprom** - *Objekttyp in Register _HA2 setzen für EEPROM (Rückgabeparameter).*

MAKROS/SREG/

- **SregSave, SregRestore** - *Statusregister auf Stack sichern/wiederherstellen, globale Interrupts blockieren/wiederherstellen.*
- **SregSaveRS, SregRestoreRS** - *Statusregister auf Stack sichern/wiederherstellen (Angabe des Registers), globale Interrupts blockieren/wiederherstellen.*

- **xIn, xOut** - *Automatisches verwenden von **in/out** für Port-Registerzugriffe.*
- **xCbi, xSbi** - *Automatisches verwenden von **cbi/sbi** für Port-Registerzugriffe.*
- **gCbi, gSbi** - *Automatisches verwenden von **cbi/sbi** für Port-Registerzugriffe (Ersatz verändert Register WL).*
- **xSbic, xSbis** - *Automatisches verwenden von **sbic/sbis** (Ersatz verändert Register _TMPD).*
- **gSbic, gSbis** - *Automatisches verwenden von **sbic/sbis** (Ersatz verändert Register WL).*

GLOBALE FUNKTIONEN (STDLIB.INTERFACE)



Liste der global verfügbaren Funktionen, die in externen Bibliotheken oder Inline-Assembler genutzt werden können bzw. müssen. Die Einträge der Liste beziehen sich auf die Ordnerstruktur in der Bibliothek (siehe Bild). Die Ggf. erwarteten Parameter oder benutzten Register entnehmen sie bitte der Beschreibung bzw. dem Code innerhalb des jeweiligen Eintrags.

CONVERT/ATOI/

- **AtoI** - Umwandlung Dezimalzahl (ASCII) in 32 Bit Integer (int32).

CONVERT/BIN/

- **ConvBin8** - Umwandlung 8 Bit Integer (uint8) nach ASCII Binärdarstellung.
- **ConvBin16** - Umwandlung 16 Bit Integer (uint16) nach ASCII Binärdarstellung.
- **ConvBin24** - Umwandlung 24 Bit Integer (uint24) nach ASCII Binärdarstellung.
- **ConvBin32** - Umwandlung 32 Bit Integer (uint32) nach ASCII Binärdarstellung.

CONVERT/DEC/

- **ConvDec8s** - Umwandlung 8 Bit Integer (int8) nach ASCII Dezimaldarstellung.
- **ConvDec8u** - Umwandlung 8 Bit Integer (uint8) nach ASCII Dezimaldarstellung.
- **ConvDec16s** - Umwandlung 16 Bit Integer (int16) nach ASCII Dezimaldarstellung.
- **ConvDec16u** - Umwandlung 16 Bit Integer (uint16) nach ASCII Dezimaldarstellung.
- **ConvDec24s** - Umwandlung 24 Bit Integer (int24) nach ASCII Dezimaldarstellung.
- **ConvDec24u** - Umwandlung 24 Bit Integer (uint24) nach ASCII Dezimaldarstellung.
- **ConvDec32s** - Umwandlung 32 Bit Integer (int32) nach ASCII Dezimaldarstellung.
- **ConvDec32u** - Umwandlung 32 Bit Integer (uint32) nach ASCII Dezimaldarstellung.

CONVERT/XTOA/

- **ConvHex8** - Umwandlung 8 Bit Integer (uint8) nach ASCII Hexdarstellung.
- **ConvHex16** - Umwandlung 16 Bit Integer (uint16) nach ASCII Hexdarstellung.
- **ConvHex24** - Umwandlung 24 Bit Integer (uint24) nach ASCII Hexdarstellung.
- **ConvHex32** - Umwandlung 32 Bit Integer (uint32) nach ASCII Hexdarstellung.

EEPROM/

- **Read** - Byte aus Eeprom lesen.
- **Write** - Byte in Eeprom schreiben.

MAINSTD/

- **ArrayClearEram** - Speicherbereich nullen (Eeprom).
- **ArrayClearMemoryBlock** - Speicherbereich nullen (Memoryblock).
- **ArrayClearSram** - Speicherbereich nullen (SRAM).
- **ArrayReverse** - Speicherbereich reversieren/umdrehen (SRAM).

- **MemCmp** - Speicherbereiche vergleichen (SRAM).
- **MemCpy** - Speicherbereich kopieren (SRAM).
- **MemRev** - Speicherbereich reversieren/umdrehen (SRAM).
- **MemSort** - Speicherbereich aufwärts sortieren (SRAM).

- **QSort8u** - Speicherbereich aufwärts sortieren, 8-Bit (SRAM).
- **QSort16u** - Speicherbereich aufwärts sortieren, 16-Bit (SRAM).
- **QSort16s** - Speicherbereich aufwärts sortieren, 16-Bit mit Vorzeichen (SRAM).

- **StackSpaceAssignRestore** - Speicherbereich auf Stack reservieren/freigeben.

- **Wait** - Delay in Sekunden.

- **Waitms** - Delay in Millisekunden.

MEMORYBLOCK/

- **Clear** - MemoryBlock nullen.
- **ClrVar** - MemoryBlock-Pointervariable auf **nil** setzen, Ggf. vorhandener Speicherblock wird freigegeben.
- **ClrVarAddr** - Rückreferenz auf Pointervariable im MemoryBlock löschen.
- **Compare** - MemoryBlocks vergleichen.
- **Constructor** - MemoryBlock anlegen.
- **Constructor_FromVarRef** - MemoryBlock anlegen mit Pointervariablenadresse (byRef), automatischer Zuweisung und Ggf. Freigabe eines bereits zugewiesenen MemoryBlock.
- **Destructor** - MemoryBlock freigeben, eine Ggf. referenzierte Objektvariable wird auf **nil** gesetzt.
- **DestructTypedZ** - MemoryBlock freigeben in Abhängigkeit von TypeFlags, eine Ggf. referenzierte Objektvariable wird auf **nil** gesetzt.
- **DestructTypedTwo** - Zwei MemoryBlocks freigeben in Abhängigkeit von TypeFlags, Ggf. referenzierte Objektvariablen werden auf **nil** gesetzt.
- **FindByte** - Ein Byte im MemoryBlock finden.
- **GarbageCollection** - Komplette Garbage-Collection durchführen.
- **SetVar** - Neue Adresse in MemoryBlock-Pointervariable setzen. Nur bei Zuweisung von **nil** wird ein Ggf. bereits zugewiesener Speicherblock freigegeben.
- **SetVarAddr** - Neue Referenzadresse auf Pointervariable im MemoryBlock setzen.
- **Size** - Größe eines MemoryBlock ermitteln (Datenbereich).
- **SizeAll** - Größe aller MemoryBlocks im Speicher ermitteln (komplett, inklusive Header).

#LIBRARY

Bindet eine externe Bibliothek in den Sprachumfang des Projektes ein. **Siehe auch:** **Bibliotheken**

Syntax:

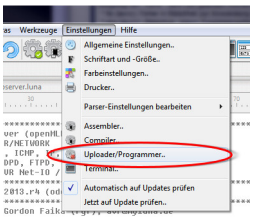
- **#library** *"Dateipfad"*

BEISPIEL

```
const F_CPU = 8000000
avr.device = atmega328p
avr.clock  = F_CPU
avr.stack = 84

#library "Library/TaskTimer.Interface"

#define LED1 as PortD.5
#define LED2 as PortD.6

LED1.mode = output,low
LED2.mode = output,high

TaskTimer.Init(Timer1,2,500) '2 Tasks, 500 ms
TaskTimer.Task(0) = mytask0().Addr
TaskTimer.Task(1) = mytask1().Addr

avr.Interrupts.Enable

do
loop

procedure mytask0()
  LED1.Toggle
endproc

procedure mytask1()
  LED2.Toggle
endproc
```

IDE

Programmer/Uploader



Der verwendete Programmer/Uploader kann in den Einstellungen der Luna-IDE geändert werden. Die Luna-IDE bindet zum Upload der kompilierten Dateien die Software Avrdude ein. Alle von AvrDude unterstützten Programmer können daher verwendet werden.

Die Einstellung erfordert das korrekte Setzen der Programmdatei, die verwendete Schnittstelle - z.B. "com1" oder "usb", der verwendete Programmer und die Übertragungsgeschwindigkeit. Die Luna-IDE stellt hier nur eine für den Benutzer vereinfachte Konfigurationsmöglichkeit zur Verfügung. Weitere Informationen bitte der Dokumentation zu AvrDude entnehmen.

AVRDUDE UND USB

Bei Verwendung der USB-Schnittstelle ist für AvrDude die Installation von **libusb** und die anschließende Registrierung des Programmers mit libusb notwendig.

- Offizielle Webseite (Englisch)
- Download libusb-win32 (Windows XP, 7, Vista, ..)
- Download libusb (Linux/Andere)

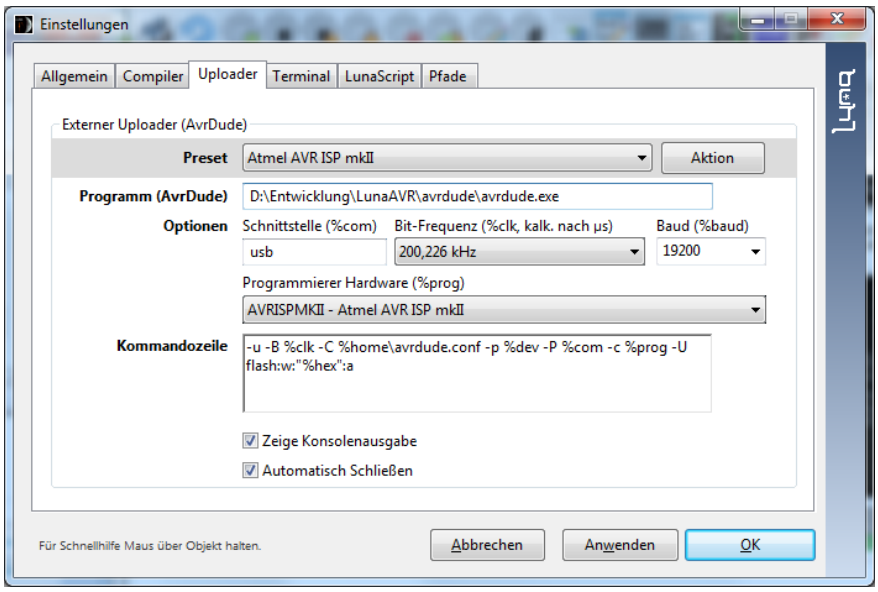
Eine Ausnahme gibt es. Wird ein Programmer genutzt der über eine com-Schnittstelle arbeitet, wie z.B. mySmartUSB light oder die Bootloader für Arduino Nano V3 und MEGA2560 muss kein **libusb** installiert werden. Aber dafür muss eine avrdude Version die OHNE libusb-Support kompiliert wurde genutzt werden.

Siehe auch: #ide - IDE-Steuerung im Source

BEISPIELE

ATMEL AVR ISP MKII

Einstellungen für Atmel AVR ISP mkII

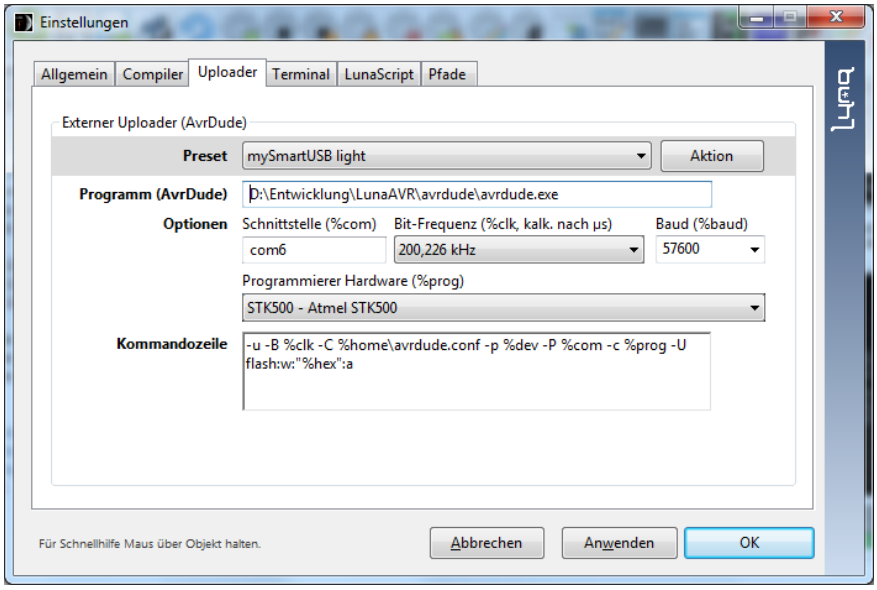


"-u -B %clk -C %home\avrdude.conf -p %dev -P %com -c %prog -U flash:w:"%hex":a"

Gebenenfalls noch den absoluten Pfad zu **avrdude.exe** einstellen:

MYSMARTUSB LIGHT

Einstellungen für mySmartUSB light(wobei die Schnittstelle die vom Adapter ist):

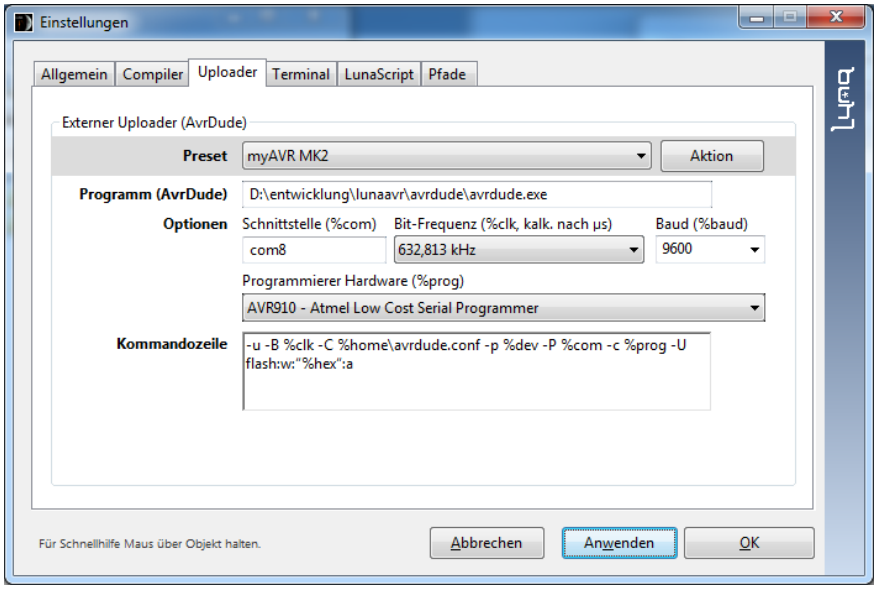


"-u -B %clk -C %home\avrdude.conf -p %dev -P %com -c %prog -U flash:w:"%hex":a"

MYSMART MK2

Einstellungen für mySmart MK2, mit AVR910/AVR 911 spezifischem Protokolle.

Siehe auch: Webseite des Herstellers

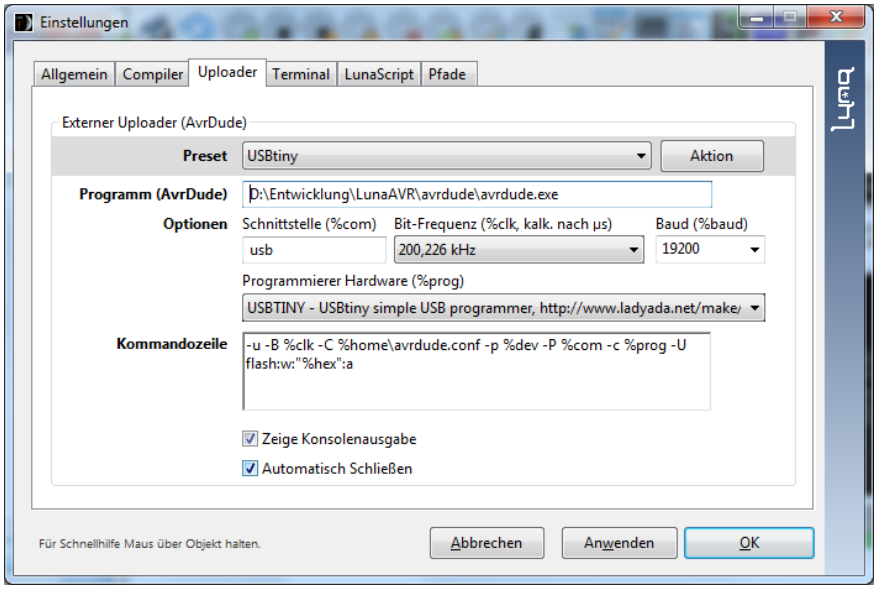


"-u -B %clk -C %home\avrdude.conf -p %dev -P %com -c %prog -U flash:w:"%hex":a"

USBTINY

Einstellungen für USBTiny:

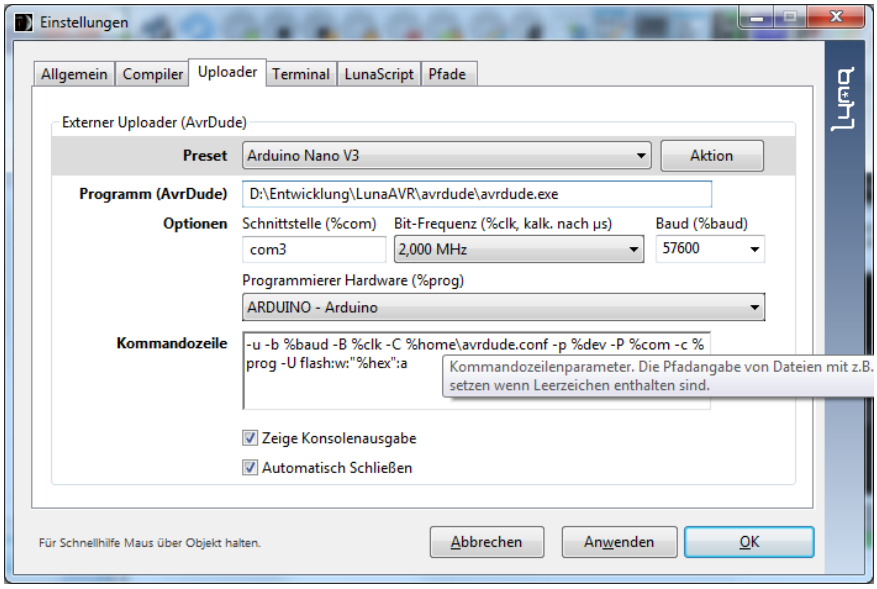
Für Treiber-Installation siehe:Webseite von LadyAda und hier: AVR-ISP-Stick Wiki von ehajo



"-u -B %clk -C %home\avrdude.conf -p %dev -P %com -c %prog -U flash:w:"%hex":a"

ARDUINO NANO V3

Einstellungen für Arduino Nano V3 Bootloader (die COM-Schnittstelle ist die vom Nano V3):

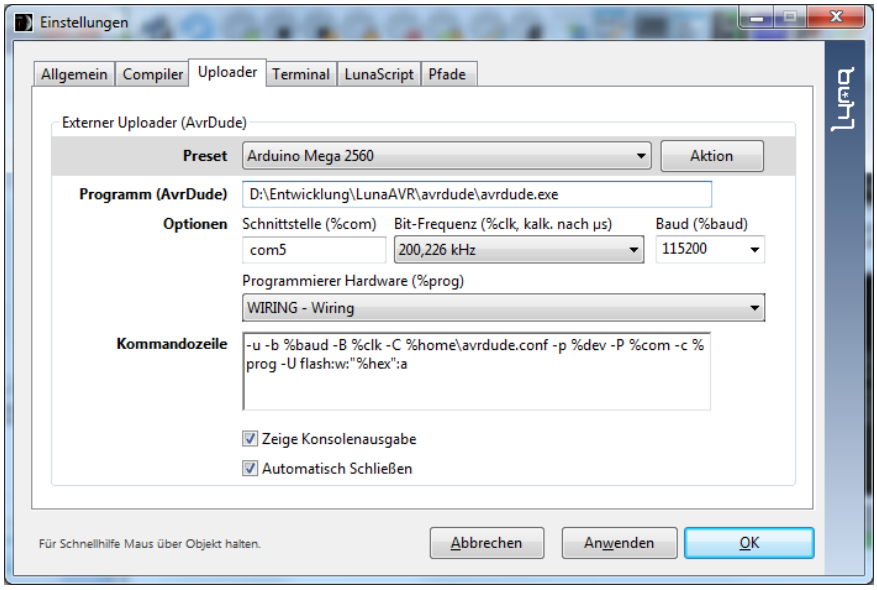


Wichtig: Baud muss auf 57600 stehen und Programmierer Hardware auf ARDUINO.

"-u -b %baud -B %clk -C %home\avrdude.conf -p %dev -P %com -c %prog -U flash:w:"%hex":a"

ARDUINO MEGA2560

Einstellungen für Arduino MEGA2560 Bootloader (die COM-Schnittstelle ist die vom MEGA2560):



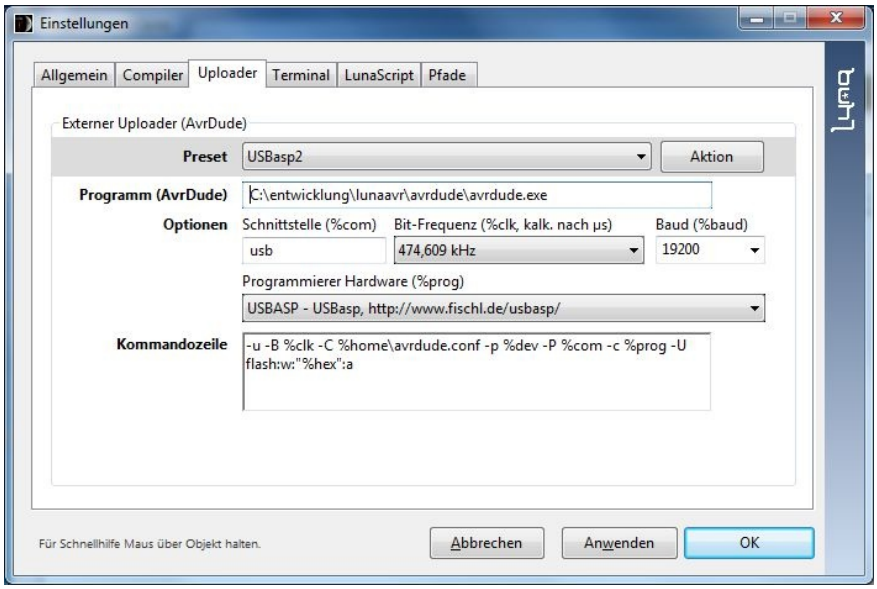
Wichtig: Baud muss auf 115200 stehen und Programmierer Hardware auf WIRING.

"-u -b %baud -B %clk -C %home\avrdude.conf -p %dev -P %com -c %prog -U flash:w:"%hex":a"

USBASP

Einstellungen für USBasp von Thomas Fischl:

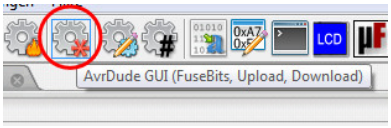
Für Treiber, Firmware und Installation siehe: Webseite von Thomas Fischl



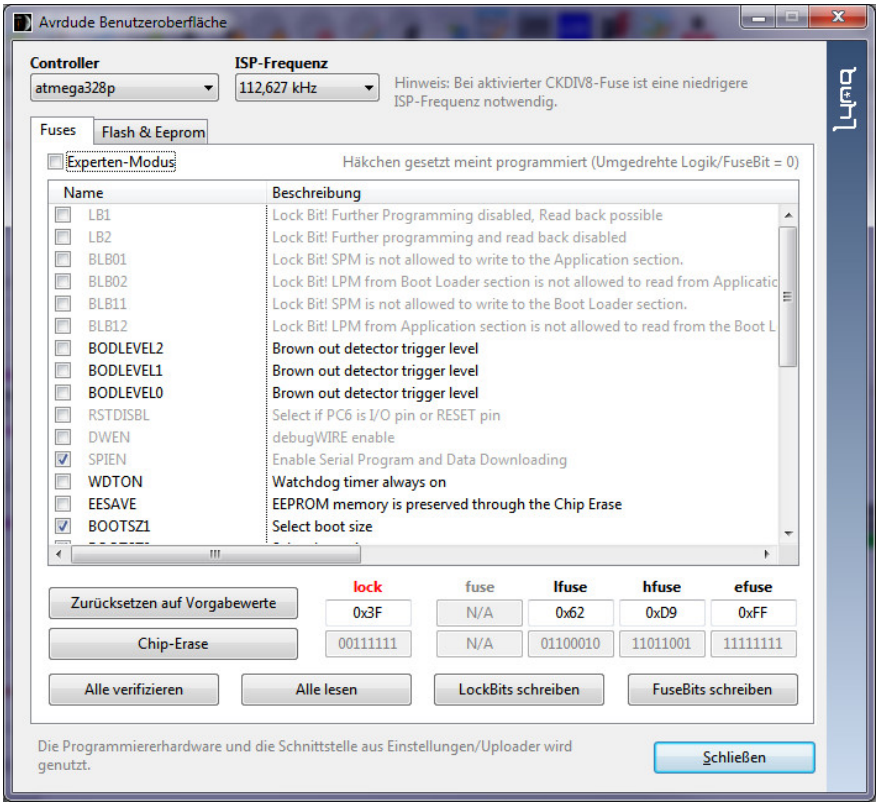
"-u -B %clk -C %home\avrdude.conf -p %dev -P %com -c %prog -U flash:w:"%hex":a"

Fusebits

Aufruf der ***"AvrDude GUI"*** im Menü ***"Werkzeuge"*** oder durch Klick auf den entsprechenden Button:

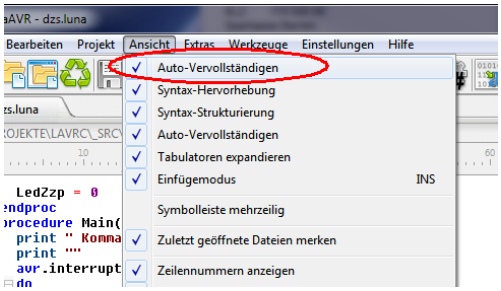


Es öffnet sich das GUI-Fenster:



Alle besonders "gefährlichen" Fusebits sind nur bei aktiviertem ***"Experten-Modus"*** anwählbar. Welche Bedeutung die einzelnen Fusebits haben, entnimmt man dem Datenblatt des Controllers. Bei neuen/jungfräulichen Controllern ist oft das CKDIV8-Fusebit aktiv. Hier wird dann der Controllertakt durch 8 geteilt. In diesem Fall wählt man eine niedrige ISP-Frequenz um mit dem Controller kommunizieren zu können.

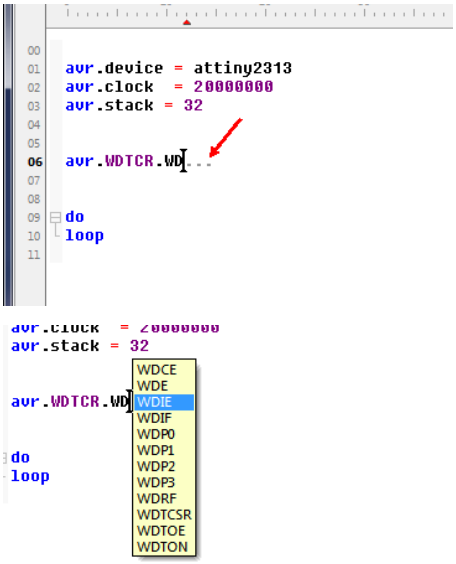
Autovervollständigen



Der Editor in der Luna-IDE enthält eine Funktionalität zum automatischen Vervollständigen zahlreicher Befehle, Befehlsgruppen oder controllerspezifischer Namen und Werte. Das Auto-Vervollständigen funktioniert am Ende einer eingegebenen Zeichenkette. Innerhalb einer Zeichenkette werden keine Vorschläge gemacht.

Die vom Programmierer eingegebene Zeichenkette wird während der Eingabe geprüft. Ist ein Term bekannt, wird er als Vorschlag grau dargestellt und kann mit TAB bestätigt werden. Sind mehr als eine Möglichkeit vorhanden, werden drei graue Punkte dargestellt. Durch betätigen der TAB-Taste erscheint ein Auswahlmenü. Ein erneutes betätigen der TAB-Taste bestätigt die selektierte Auswahl. Escape bricht die Auswahl ab.

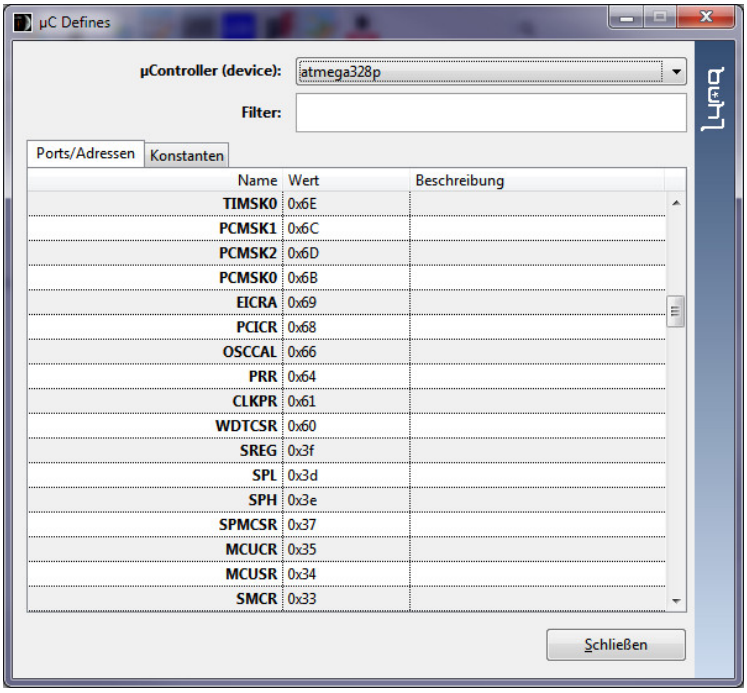
Dem Auto-Vervollständigen sind sogutwie alle Befehlsgruppen und deren Eigenschaften oder Funktionen bekannt. Auch die vom Hersteller für den verwendeten Controller vorbelegten Konstanten und Registernamen sind für den Entwickler vollständig verfügbar.



Controllerwerte

Für die Konfiguration spezieller Hardware oder wenn man bestimmte Controllerfunktionen von Hand konfigurieren möchte, benötigt man die verfügbaren Namen der Ports und Konstanten die vom Hersteller für den jeweiligen Controller vordefiniert wurden. Hierfür steht in der Luna-IDE ein entsprechender Browser/Editor zur Verfügung. Dieser ist über ***"µC-Defines Browser/Editor"*** im Menü ***"Werkzeuge"*** oder über den entsprechenden Knopf erreichbar:

Es öffnet sich das Übersichtsfenster:



Der Zugriff im Sourcecode erfolgt über die Klasse "Avr" bzw. **bei Atxmega-Controllern ist zusätzlich das Universal-Interface** verfügbar.

Die Werte können auch editiert bzw. können neue Controller angelegt werden.

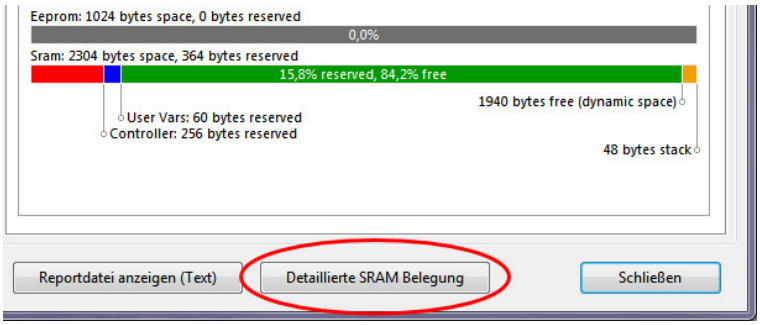
- Siehe hierzu: Noch nicht vorhandenen, neuen Controller hinzufügen

BEISPIEL

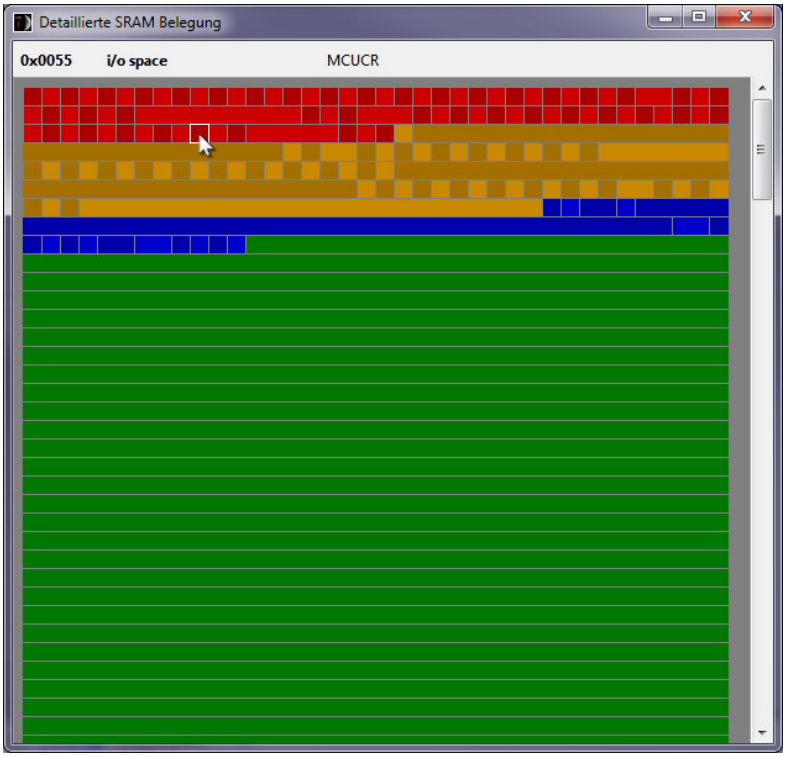
```
avr.TWCR = ((1<<TWINT) or (1<<TWSTA) or (1<<TWEN))

' wait until transmission completed
while ( 0 = (avr.TWCR and (1<<TWINT)) )
wend

'Ein an das Register angehängtes Schlüsselwort eines numerischen Datentyps
'wie z.B. "word" weist den Compiler an den Wert mit der entsprechenden
'Datenbreite in das Register zu schreiben. Vorgabe ist "byte".
'Auch möglich: "int24","uint24","long",... usw.

avr.TCNT1.word = 0
```

Speicherbelegung

Mit der Version 2013.R1 ist der Build-Report um eine detaillierte Ansicht der Belegung von Variablen, Objekten, Controller-Ports usw. erweitert worden.Wenn man sich eine Übersicht über diese Belegung verschaffen möchte, wählt man ***"Report anzeigen"*** im Menü ***"Projekt"*** und klickt unten auf den Knopf ***"Detaillierte SRAM Belegung"***:



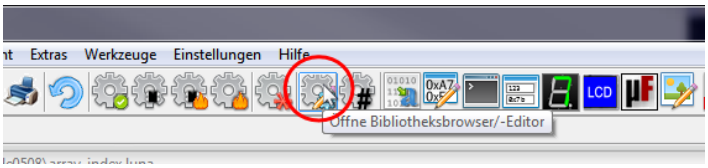
Nach dem Anklicken erscheint ein Fenster mit einer Kacheldarstellung aller Speicherzellen in verschiedenen Farben. Jede Kachel spiegelt hierbei ein einzelnes Byte wieder. Schwebt man mit der Maus über einer Kachel, wird oben Adresse, Bereichsname sowie der Name der zugehörigen Variable oder des Ports usw. dargestellt:



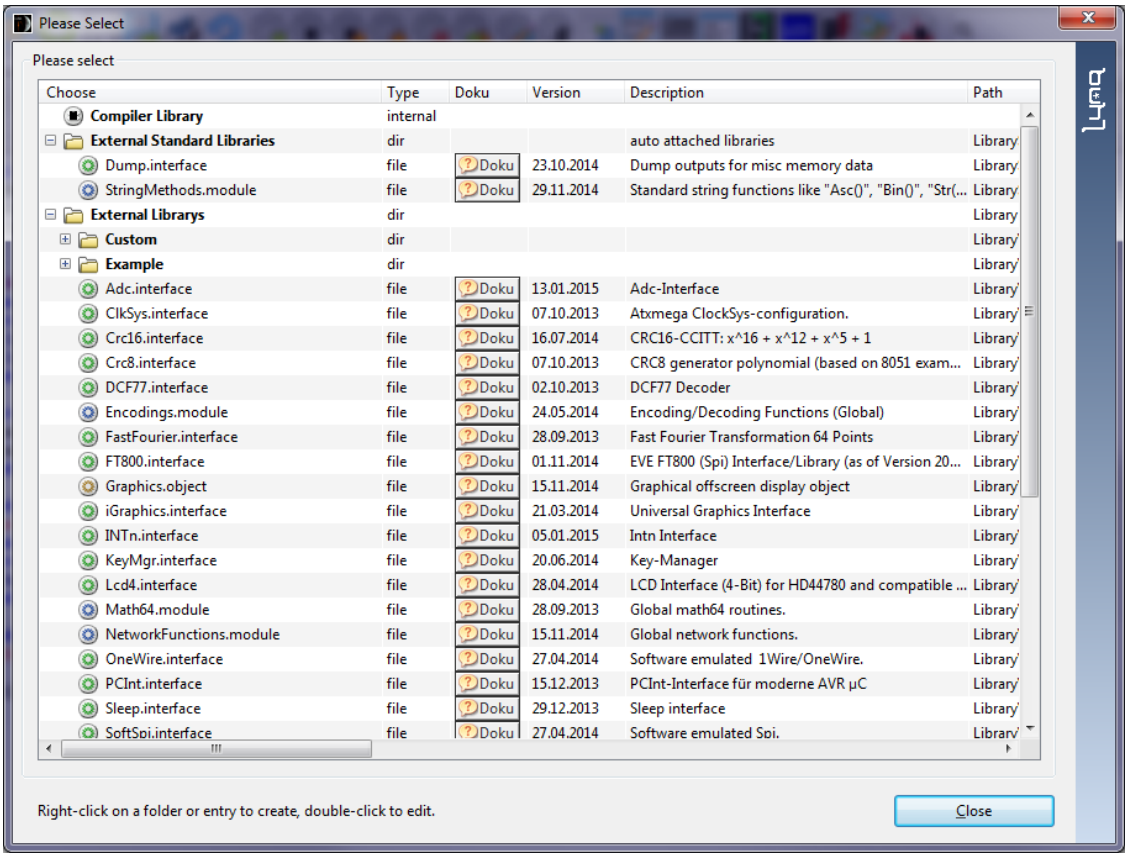
Bibliotheks-Auswahl

Siehe auch: **Externe Bibliotheken**

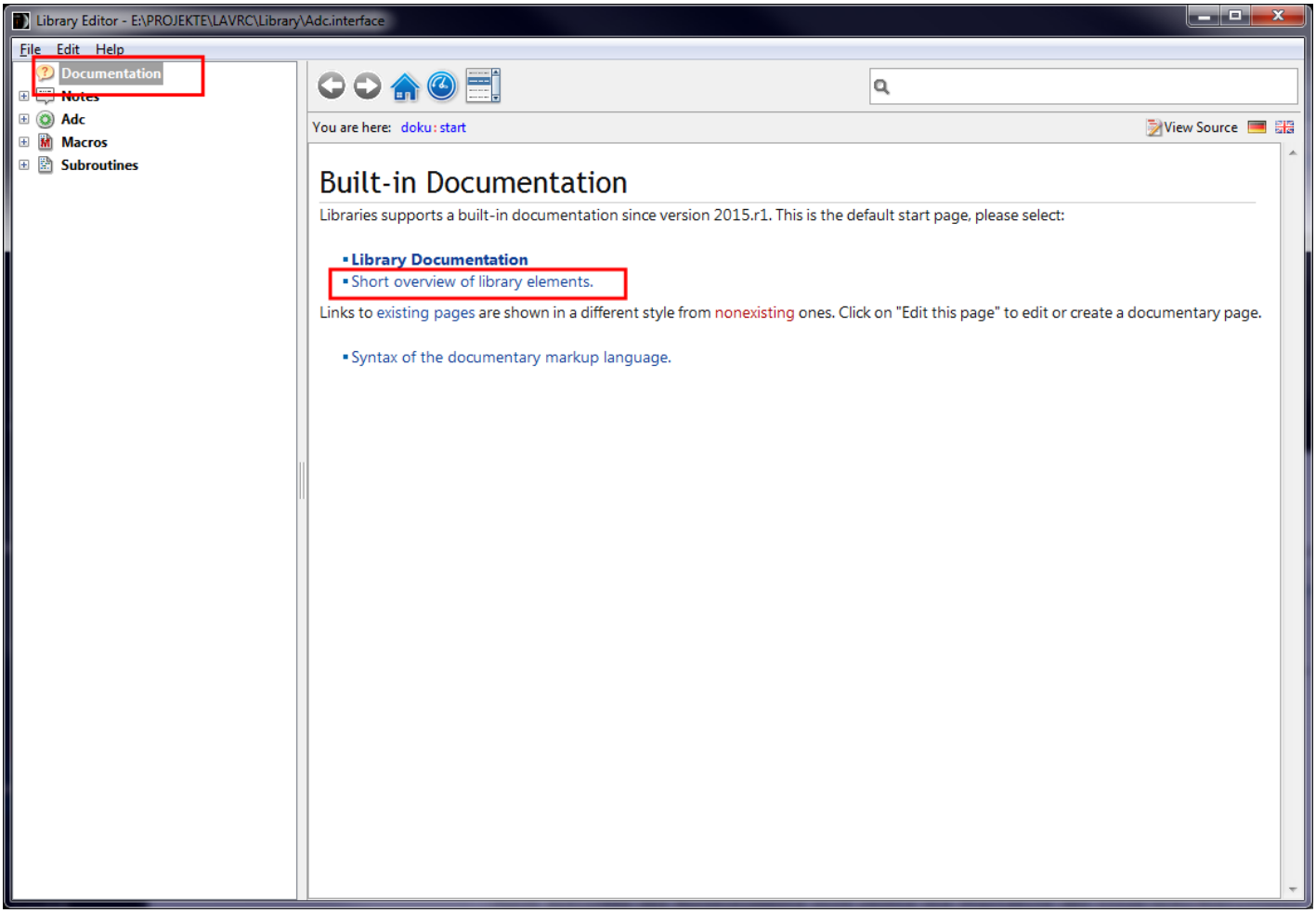
Eine Übersicht über die vorhandenen, externen Bibliotheken erhält man durch Klick auf den entsprechenden Button oder durch Anwahl des Menüpunktes *Bibliotheksbrowser/-Editor* im Menü *Werkzeuge*.



Anschließend öffnet sich die Bibliotheksauswahl:



Ein Doppelklick auf den Namen öffnet den Bibliotheks-Editor.Durch Anklicken des Wurzelelements wird rechts die Startseite der Bibliotheks-Dokumentation angezeigt. Sie enthält einen Link auf die Seite der automatisch erzeugten Kurzübersicht.

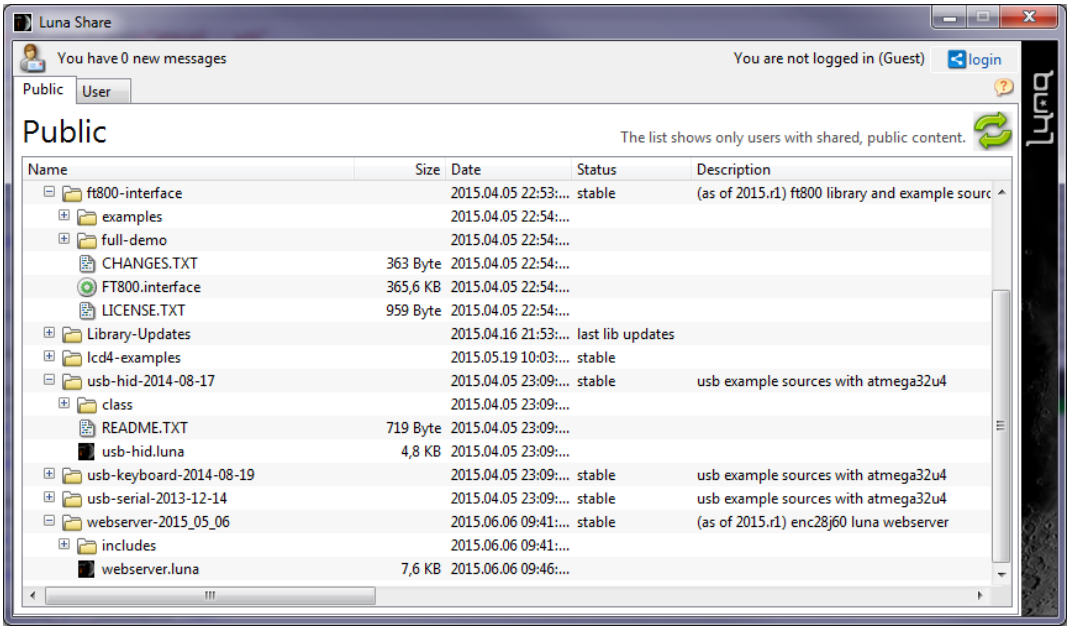

Library Editor - E:\PROJEKTE\LAVRC\Library\Adc.interface
File Edit Help
Documentation
Notes
Adc
Macros
Subroutines
You are here: doku : start
View Source
Built-in Documentation
Libraries supports a built-in documentation since version 2015.r1. This is the default start page, please select:
Library Documentation
Short overview of library elements.
Links to existing pages are shown in a different style from nonexisting ones. Click on "Edit this page" to edit or create a documentary page.
Syntax of the documentary markup language.

Luna-Share

Mit Luna-Share können Sourcen, Bibliotheken und andere Projektdateien geteilt werden. Öffentlich gestellte Dateien sind dann für Alle Anwender in Luna-Share sichtbar und können heruntergeladen werden. Durch anlegen eines Accounts können sie eigene Inhalte verwalten. **Zum Herunterladen der öffentlichen Inhalte benötigen sie keinen Account.**

Luna-Share kann geöffnet werden über das Menü "Tools", oder durch klicken auf das Symbol in der Toolbar:

Das Hauptfenster von Luna-Share:



Durch Rechtsklick auf einen Eintrag können sie den gesamten Ordner oder einzelne Dateien herunterladen. In den Einstellungen zu Luna-Share (Siehe Menü "Einstellungen") können sie festlegen, welche Dateitypen bei einem Doppelklick automatisch geöffnet werden sollen. Durch einen Doppelklick geöffnete Dateien werden heruntergeladen (temporär) und über das Betriebssystem angezeigt, z.B. Bilder oder Texte.

Technisches

Byteorder

Auf dem AVR wird **Little-Endian**-Byteorder verwendet, d.h. das niederwertige Byte liegt an *niederer* Speicheradresse. Nimmt man einen 32-Bit-Wert (4 Bytes), liegen die einzelnen Bytewerte in folgender Richtung im Speicher:

Speicher ?			
0x00	**0x01**	**0x02**	**0x03**
byte0	byte1	byte2	byte3

WICHTIGER HINWEIS

Als Konstanten im Quelltext hexadezimal oder binär ***geschriebene*** Werte sind in **Big-Endian-Schreibweise standardisiert** (sie sind damit besser für Menschen lesbar). Dies ist *unabhängig* von der darunterliegenden Architektur bzw. der Zielarchitektur. Beispiel: Das geschriebene Wort "MOON" in falscher Notation liegt dann rückwärts (falsch) im Speicher:

Geschrieben: 0x4c4f4f41 (falsch)			
0x4d	0x4f	0x4f	0x4e
"M"	"O"	"O"	"N"
Geschrieben: 0x414f4f4e (richtig)			
0x4d	0x4f	0x4f	0x4e
"N"	"O"	"O"	"M"
Speicher ?			
0x00	**0x01**	**0x02**	**0x03**
0x4e	0x4f	0x4f	0x4d
"M"	"O"	"O"	"N"

Für die AVR-Controller hat Little-Endian-Byteorder gewisse Vorteile. Um eine Zwei-Byte-Zahl in eine 4-byte-Zahl zu wandeln, muss man nur zwei mit Null gefüllte Bytes am Ende einfügen, ohne das sich dabei die Adresse verändert. Bei Big-Endian-Byteorder muss der Wert zuvor um zwei Bytes im Speicher verschoben werden.

Siehe auch: Byte-Reihenfolge (Wikipedia)

Typkonvertierung (Casting)

Explizites konvertieren bzw. festlegen eines Wertes oder Ausdruckergebnisses in/auf einen bestimmten Datentyp. Dies ist manchmal sinnvoll, wenn eine bestimmte Funktion anhand des verwendeten Datentyps eine entsprechend angepasste Funktonalität aufweist. Einige Ausgabefunktionen passen z.T. die Art der Ausgabe an den übergebenen Wert an.

Beispielsweise erfolgen mathematische Berechnungen oder Bitmanipulationen im nächst größeren Datentyp, wenn der aktuelle Datentyp das Ergebnis möglicherweise nicht aufnehmen kann (Datentypen kleiner als Long).

Beispiel 1 zur Veranschaulichung:

```
dim a,b as byte
print hex(a+b)  ' Das Ergebnis ist vom Datentyp word,
                ' die Hex-Ausgabefunktion gibt daher einen word-Hexwert aus
```

In obigem Beispiel kann man nun mit einer expliziten Typkonvertierung festlegen von welchem Datentyp das Ergebnis sein soll.

Funktionen der Typkonvertierung

- ***byte****(Ausdruck)*
- ***int8****(Ausdruck)*
- ***uint8****(Ausdruck)*
- ***integer****(Ausdruck)*
- ***word****(Ausdruck)*
- ***int16****(Ausdruck)*
- ***uint16****(Ausdruck)*
- ***int24****(Ausdruck)*
- ***uint24****(Ausdruck)*
- ***long****(Ausdruck)*
- ***longint****(Ausdruck)*
- ***int32****(Ausdruck)*
- ***uint32****(Ausdruck)*
- ***single****(Ausdruck)*

Siehe auch:

- BcdEnc()
- BcdDec()
- CE16()
- CE32()

Beispiel 2:Beispiel 1 zur Veranschaulichung:

```
dim a,b as byte
print hex(byte(a+b))  ' Das Ergebnis ist vom Datentyp byte,
                      ' die Hex-Ausgabefunktion gibt daher einen byte-Hexwert aus
```

Variablen in Methoden

VARIABLEN IN METHODEN

Implementiert man eine Methode mit Parametern und/oder innerhalb der Methode zusätzliche Variablen, *sind Diese lokal auf die Methode bezogen* (sie sind nur in der Methode selbst sichtbar). Parameter und temporär dimensionierte Variablen, belegen nur während der Ausführung der Methode den benötigten Speicherplatz im Arbeitsspeicher. Zusätzliche dimensionierte Variablen können in Methoden auch *statisch* dimensioniert werden.

Siehe hierzu: dim, static

VERANSCHAULICHUNG

Nehmen wir an es wurde folgene Methode implementiert:

```
procedure test(a as byte, b as integer)
  dim c,d as byte
  [..]
endproc
```

Diese Methode reserviert bei Aufruf vier verschiedene temporäre Variablen im Arbeitsspeicher. Als Erstes landen die Parameter auf dem Stack. Dies geschieht *von rechts nach links*, also umgekehrt wie in der Parameterdefinition der Methode schriftlich angegeben.

Der Ablauf beim Aufruf einer Methode

1. Rücksprungadresse auf den Stack legen
2. Parameter "b" auf den Stack legen
3. Parameter "a" auf den Stack legen
4. Speicherplatz für Variable "d" auf dem Stack reservieren
5. Speicherplatz für Variable "c" auf dem Stack reservieren
6. Aktuelle Stackposition merken (Zeiger)

Der Compiler weiß beim Erstellen des Programms wo und wieviele Variablen auf dem Stack abgelegt werden und passt die entsprechenden Aufrufe innerhalb der Methode an.

Wird in der Methode auf eine temporäre, lokale Variable zugegriffen, passiert Folgendes:

1. Zeiger holen
2. Variablenposition hinzurechnen
3. Zugriff durchführen

Bei globalen Variablen, sowie bei in der Methode ***statisch*** dimensionierten Variablen entfallen die ersten beiden Schritte, wodurch Zugriffe schneller sind. Bei zeitkritischen Zugriffen, sind also globale oder statische, lokale Variablen vorzuziehen, sofern dies technisch möglich ist.

Verlassen der Methode

Beim Verlassen der Methode wird der Stack auf die ursprüngliche Position zurückgesetzt, wobei die Rücksprungadresse auf dem Stack verbleibt. Der abschließende Maschinenbefehl "ret" (Return) holt sich diese Rücksprungadresse vom Stack und springt dann zu der Position hinter dem Methodenaufruf zurück. Damit ist der auf dem Stack vorübergehend belegte Speicher wieder freigegeben.

Speicherverwaltung

Für die Verwaltung des Arbeitsspeichers (SRAM) steht in Luna eine Speicherverwaltung zur Verfügung. Diese Speicherverwaltung erlaubt das dynamische Unterteilen des Arbeitsspeichers in einzelne Blöcke verschiedener Größe. Ein solcher Block nennt sich in Luna MemoryBlock und besitzt Objekteigenschaften.

Bei der in Luna implementierten Speicherverwaltung handelt es sich um eine auf hohe Effizienz ausgelegte, integrierte, automatische, generationelle Speicherbereinigung (Garbage Collection). In der Praxis ist die Lebensdauer von Objekten meist sehr unterschiedlich. Auf der einen Seite existieren Objekte, die die gesamte Laufzeit der Applikation überleben. Auf der anderen Seite gibt es eine große Menge von Objekten, die nur temporär für die Durchführung einer einzelnen Aufgabe benötigt werden. Mit jeder Anwendung des Freigabe-Algorithmus werden langlebige Objekte in eine höhere Generation verschoben. Der Vorteil liegt darin, dass die Speicherbereinigung für niedrige Generationen häufiger und schneller durchgeführt werden kann, da nur ein Teil der Objekte verschoben und deren Zeiger verändert werden müssen. Höhere Generationen enthalten mit hoher Wahrscheinlichkeit nur lebende (bzw. sehr wenige tote) Objekte und müssen deshalb seltener bereinigt werden. Die Besonderheit des in Luna implementierten Algorithmus ist, dass er keine Zähler für die Lebensdauer der einzelnen Objekte benötigt und alle Objekte Rückwärtsreferenzen aufweisen. Die Rückwärtsreferenzen erlauben es "tote" Speicher-Objekte und fehlerhafte Objektvariablen zu erkennen. Zugriffe auf ungültige Objekte können mittels einer Exception erkannt werden.

Genutzt wird die Speicherverwaltung in Programmen immer dann, wenn instrinsische Variablen mit Verweisen auf dynamischen Speicher genutzt werden, wie z.B. ***string*** oder ***MemoryBlock***. Die Speicherverwaltung kann jedoch aus technischen Gründen nur sinnvoll genutzt werden, wenn mehr als 64 Bytes Arbeitsspeicher *frei verfügbar* sind.

Ist die Speicherverwaltung nicht nutzbar, sind Stringvariablen und darauf zugreifende Stringfunktionen, sowie MemoryBlocks im Programm zu vermeiden.

Typische Speicheraufteilung in Luna

Aufbau eines Speicherblocks

Da die Speicherverwaltung wissen muss wo und wieviel Speicher belegt ist, benötigt jeder Speicherblock (MemoryBlock) einige Bytes um Verwaltungsdaten zu speichern (Header).

Ein MemoryBlock besteht aus einem Header (5 bytes) und den Nutzdaten:

Name	Wert	Typ	Beschreibung
Magic	0xAA	byte	Erkennungsbyte
Length	Number	word	Anzahl der Nutzdaten in Bytes
Ref	Address	word	Rückreferenz auf die Objekt-Variable mit welcher der Speicherblock aktuell verknüpft ist oder Null.
Data			Nutzdaten

Ist zu wenig Arbeitsspeicher vorhanden, kann der Speicher selbst verwaltet werden. Hierfür steht der direkte Zugriff auf den gesamten Arbeitsspeicher durch das Objekt sram zur Verfügung.

Veranschaulichung

Man nehme an, es wurden mehrere Stringvariablen dimensioniert:

```
dim datum,zeit as string
```

Bei der Zuweisung, oder bei der Verwendung von Stringfunktionen, wird das Ergebnis in einem MemoryBlock gespeichert und die Startadresse der Daten im MemoryBlock als Zeiger (Pointer) der entsprechenden Variable zugewiesen. Es wird hierbei absichtlich nicht die eigentliche Startadresse des MemoryBlocks als Zeiger zugewiesen, sondern der Beginn der Daten. Andernfalls müsste bei jedem Zugriff die Größe des Headers hinzugerechnet werden, was ineffizienter wäre.

Nehmen wir ebenfalls an, dass der Arbeitsspeicher mit der Adresse 100 (dezimal) beginnt. Die Variablen *datum* und *zeit* sind 16-Bit-Zeiger und belegen damit jeweils 2 Byte im Arbeitsspeicher:

Speicher ->			
100	101	102	103
Variable *datum*		Variable *zeit*	
0 (nil)		0 (nil)	

Hinter diesen Variablen beginnt der dynamische (frei verfügbare) Arbeitsspeicher und endet beim Stack-Ende [1]. Wurde den Stringvariablen noch kein Wert zugewiesen, haben sie den Wert *nil* [2].

Durch Zuweisung eines Wertes, wird ein MemoryBlock im Arbeitsspeicher erzeugt:

```
dim datum,zeit as string
```

```
datum="13.11.1981"
```

Im Speicher sieht es nun folgendermaßen aus:

Speicher ->																			
100	**101**	**102**	**103**	**104**	**105**	**106**	**107**	**108**	**109**	**110**	**111**	**112**	**113**	**114**	**115**	**116**	**117**	**118**	**119**
109		0 (nil)		0xAA	11		100		10	'1'	'3'	'.'	'1'	'1'	'.'	'1'	'9'	'8'	'1'
Variable *datum*		Variable *zeit*		Header MemoryBlock					Daten MemoryBlock										

Wie man erkennen kann, ist die Variable *zeit* unbelegt und die Variable *datum* hat als Wert nun die Anfangsadresse der Daten vom MemoryBlock. Die Daten der Zeichenkette sind wiederum als Pascal-String abgelegt, damit verarbeitende Funktionen die Länge der Zeichenkette ermitteln können. Dies ist wichtig, da nur die unsichtbar im Hintergrund agierende Speicherverwaltung selbst auf den Header des MemoryBlocks zugreift und zugreifen darf.

Weiterhin ist zu erkennen, dass im Header des MemoryBlocks nun die Adresse der Variable gespeichert ist. Variable und MemoryBlock verweisen also aufeinander. Dies ist wichtig, da bei einer Freigabe eines MemoryBlocks und dem damit verbundenem Aufräumen des Speichers (defragmentieren/komprimieren) sich die Adresse der nachfolgend noch vorhandenen MemoryBlöcke verändert (sie rücken auf). Die Speicherverwaltung passt dann alle Variablen an, die einen Verweis (Zeiger) auf einen MemoryBlock beinhalten.

Vor- und Nachteile

Jede Art der Speicherverwaltung besitzt ihre Vor- und Nachteile. Ist keine Speicherverwaltung vorhanden, muss sich der Programmierer penibel selbst darum kümmern den Speicher sinnvoll aufzuteilen. Dies schließt den Nachteil mit ein, dass beispielsweise Strings mit einer festen Länge dimensioniert werden müssen (statische Strings). Kann also ein String innerhalb des Programms bis zu 60 Bytes groß werden, muss von Anfang an dieser Speicher dafür reserviert sein, auch wenn im überwiegenden Teil des Programms nur z.Bsp. 10 Bytes benötigt werden. Vorteilig ist diese Variante dann, wenn nur wenige und kleine Stringoperationen erfolgen und wenig Arbeitsspeicher zur Verfügung steht.

Eine Speicherverwaltung ist dann effizienter und z.T. trotz der zusätzlichen Verwaltungsdaten gar speicherplatzsparender, wenn es sich um mehrere zu verwaltende Strings handelt und Daten in eigenen Speicherbereichen verwaltet werden müssen. Zudem ist es einfacher dynamische Daten zuzuweisen. Bei wenig Speicher (weniger als 128 Bytes freiem Speicher) ist sie wiederum ineffizient durch die entstehenden Verwaltungsdaten.

Luna versucht beides dadurch zu kombinieren, indem zusätzlich die Möglichkeit besteht durch Strukturen statische Strings bzw. Datenblöcke anlegen zu können. Beschränkt man sich auf Ein- und Ausgabe und vermeidet Stringfunktionen wie Left(), Right() oder Mid() usw., oder bildet sie selbst für die eigenen Speicherstrukturen, vereint dies beide Varianten der Verwaltung des Arbeitsspeichers.

1) Der Stack startet am Ende des Arbeitsspeichers und wächst Richtung Anfang. Die Angabe avr.Stack legt fest wieviel Bytes der Stack belegt

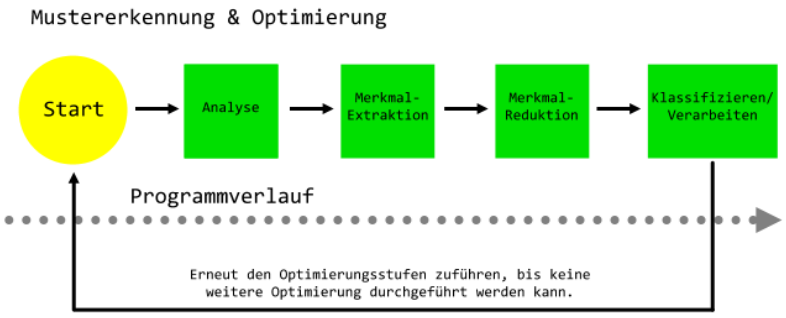
2) Unbelegte Objektvariablen wie String oder MemoryBlock sind dann *nil* wenn ihnen noch kein Objekt (MemoryBlock) zugewiesen wurde

Optimierer

Der Luna-Compiler/Assembler enthält viele verschiedene Stufen in unterschiedlichen Prozessbereichen, welche den aus dem Quelltext erzeugten Zwischencode und den erzeugten Assemblercode optimieren. Hierbei wird der erzeugte Maschinencode auf Geschwindigkeit und Größe während des Kompiliervorgangs nachbearbeitet.

Die verschiedenen Optimierungsstufen bestehen aus Analysen über die Zugriffe von Speicherbereichen, Sprüngen und arithmetischen oder logischen Operationen. Weiterhin werden Sprunganweisungen und Unterprogrammaufrufe möglichst mit direkter Adressierung durch die kürzere und schnellere relative Adressierung ersetzt. Vergleiche werden so umstrukturiert, dass sie schnell erfolgen können. Berechnungen werden aufgelöst und zusammengefasst bzw. so weit wie möglich reduziert.

Eine weitere Optimierungsstufe ist die Mustererkennung. Diese analysiert beispielsweise geladene Werte in Registern. Sie erkennt bestimmte Strukturen und kann ihre logische Abfolge analysieren. Dies verhindert das ein bereits geladener Zeiger in einer nachfolgenden Anweisungsgruppe erneut geladen wird. Sie erkennt also, ob bestimmte Registerwerte zwischenzeitlich durch den Programmverlauf geändert wurden.



Die Optimierungsstufen sind per Vorgabe eingeschaltet. Sie können mittels Compiler-Kommandozeilenoption komplett oder mittels pragma für Teilbereiche deaktiviert werden.

Luna Picture

LUNA PICTURE (*.LP)

Das Luna-Picture-Format ist ein propritäres Bildformat, welches direkt von der IDE mit einem Editor/Konverter unterstützt wird. Durch den simplen Aufbau eignet es sich für die Verwendung in Mikrocontroller-Anwendungen.

Man kann jedoch natürlich jedes andere Format in seine Programme einbinden und mit einer entsprechenden Dekodierungsroutine verarbeiten. Beispielsweise unterstützt das Standard-Windows-Format "bmp" ebenfalls RLE-Kompression und eine Farbtiefe bis 24 Bit. Eine weitere Alternative ist "png", wobei hier jedoch aufwändigere Dekodierungsroutinen notwendig werden.

Im Gegensatz zum im Mikrocontroller-Bereich bekannten "BGF"-Format, führt es nicht zu Dekodierungsfehlern wenn bestimmte Byte-Folgen im Bild auftreten. Zudem besitzt es eine Endmarkierung.

FORMATBESCHREIBUNG

Das Luna-Picture-Format besteht aus einem Header, den mit RLE gepackten Daten und einer Endmarkierung.

Offset	Beschreibung	Name
0x00	Breite in Pixel	Width
0x01	Höhe in Pixel	Height
0x02	Bittiefe (1)	Depth
0x03	unbelegt/reserviert (0)	reserved
0x04..n	RLE-Komprimierte Bilddaten	Data
0x77:0x00	Endmarkierung (2-Byte-Token)	EndMark

Die Bilddaten: Erster Pixel ist an Position links oben (x=0,y=0)

- 1Bit: Jedes Byte speichert 8 Pixel x,y+0..7 (bit)
- 8Bit: Jedes Byte speichert 1 pixel x,y (byte)

Die RLE-Kodierung
Man liest ein Byte und prüft ob es sich um den Wert **0x77** handelt (Magic-Byte). Dieses markiert den Beginn eines 3-Byte-Tokens: ***0x77***:*count:value*, Beispiel: Für "0x77 0xFF 0x13" ist 255 x der Wert 0x13 einzusetzen. Das wars auch schon und das ist wohl auch der einzige Vorteil dieses simplen Formats: Einfach und und vor allem schnell auf Mikrocontrollern zu verarbeiten.

Beispiel-Dekodierungsroutine

```
' Initialisierung
avr.device = atmega168
avr.clock = 20000000          ' Quarzfrequenz
avr.stack = 32                ' Bytes Programmstack (Vorgabe: 16)
uart.baud = 19200             ' Baudrate
uart.Recv.enable              ' Senden aktivieren
uart.Send.enable              ' Empfangen aktivieren

' Hauptprogramm
BildSenden(Bild.Addr) ' Bild dekodieren und auf serieller Schnittstelle ausgeben

do
loop

Procedure BildSenden(imgAddr as word)
    dim i,a,cc as byte
    dim c,w,h,size as word
    ' RLE compressed luna lcd picture (bulled proof)
    ' format description
    ' -- header (4 bytes) ------------------
    ' byte   width in pixel
    ' byte   height in pixel
    ' byte   depth (1,8)
    ' byte   reserved (0)
    ' -- data (x bytes) --------------------
    ' First pixel ist left top x=0,y=0
    '  1 bit: pixels scanned from x to x+width-1 and 8 rows per width
    '         each byte contains 8 pixel x,y+0..7 (bit)
    '  8 bit: pixels scanned from x to x+width-1 and 1 row per width
    '         each byte contains 1 pixel x,y      (byte)
    ' RLE: magic = 0x77:count:value (3-byte token)
    ' example: 0x77 0xFF 0x13  is 255 x the value 0x13
    ' -- end mark --------------------------
    ' 0x77 0x00 (zero count, no value byte)
    ' routine for monochrome pictures
    w = flash.ByteValue(imgAdr)
    incr imgAdr
    h = flash.ByteValue(imgAdr)
```

```
    add imgAdr,3
    size = h / 8
    size = size * w
    decr size
    clr c
    do
      a = flash.ByteValue(imgAdr)
      incr imgAdr
      if a = &h77 then
        cc = flash.ByteValue(imgAdr)
        if cc then
          incr imgAdr
          a = flash.ByteValue(imgAdr)
          incr imgAdr
          for i = 1 to cc
            Uart.Write a
            incr c
          next
        else
          exit
        end if
      else
        Uart.Write a
        incr c
      end if
    loop until c > size or c > 1023
  endproc
EndProc
#includeData Bild,"pictures\meinbild.lp"
```

Luna Font

LUNA FONT FORMAT (*.LF)

Das Luna Font-Format ist ein propritäres Font-Format für Mikrocontroller-Anwendung.Es wird in der IDE direkt durch einen entsprechenden Fonteditor unterstützt, mit dem sich problemlos eigens erstellte Schriftarten in den eigenen Mikrocontroller-Anwendungen verwenden lassen.

FORMATBESCHREIBUNG

Offset	Beschreibung	Name
0x00	Anzahl Zeilen pro Zeichen	RowsPerChar
0x01	Anzahl Bytes je Zeile	BytesPerRow
0x02	Anzahl Bytes je Zeichen gesamt	BytesPerChar
0x03	Extended Modes (Siehe Tabelle)	Mode
0x04..n	Daten für die Zeichen ab ASCII 32	Data

Erstes Pixel eines Zeichens ist links oben. Für den Zugriff auf die Daten eines bestimmten Zeichens ist nur folgende simple Kalkulation notwendig (char = Ascii-Wert des Zeichens):

- ***offset = 4 + (char - 32) * BytesPerChar***

Bei kurzem Zeichenset (Bit 0 in Extended Mode gesetzt) wird der Zugriff auf Zeichen oberhalb von ASCII 127 in den Luna-internen Graphikfunktionen ignoriert. Bei eigener Anwendung/Dekodierung muss eine entsprechende Fallabfrage vorgesehen werden.

Jedes Byte einer Zeichen-Zeile kodiert 8 Pixel in Y-Richtung.

Extended Modes		
Bit	**Beschreibung**	**Vorgabe**
0	Kurzes Zeichenset (nur Zeichen 32-127)[1)]	0
1-7	Reserviert	0

1) Ab Version 2013.r3

lavrc parameter

COMPILER/ASSEMBLER LAVRC

Der Compiler/Assembler "lavrc" ist ein Kommandozeilenprogramm. Das Programm beinhaltet den Luna-Compiler *lavrc* und den Luna-Assembler *lavra*. Der Luna-Assembler verarbeitet den vom Compiler übersetzten Luna-Source inklusive vorhandenem Inline-Assembler im Luna-Source.

Unter den verschiedenen Betriebssystemen lauten die Dateinamen der ausführbaren Datei:

Betriebssystem	Dateiname
Windows	lavrc.exe
Linux	lavrc

KOMMANDOZEILENPARAMETER

- **-i** *dir* - Fügt das Verzeichnis 'dir' als Include-Pfad hinzu.
- **-v** - Schaltet die Textausgabe ein.
- **-c** - Prüft nur die Syntax (Parser), kein Assembliervorgang
- **-z***[switch]* - Assemblercode-Optimierungen:
  - **0** - ausgeschaltet
  - **1** - eingeschaltet (Vorgabe)
- **-o***[type]* - Ausgabeoptionen des Compilers:
  - **b** - Binärdatei schreiben (*.bin)
  - **e** - Eeprom-Datei schreiben (*.eep)
  - **h** - Hex-Datei schreiben (*.hex)
  - **y** - Zwischencode vom Präcompiler schreiben (*.zc)
  - **z** - Zwischencode vom Präassembler schreiben (*.za)
  - **a** - Assembler-Ausgabe schreiben (*.s)
  - **s** - Assembler-Ausgabe schreiben, inkl. Bibliothekscode (*.s)
  - **r** - Report-Datei schreiben (*.rpt)
- **-w***[level]* - Warnungslevel:
  - **0** - Warnungen deaktivieren.
  - **1** - Nur Speicherüberlauf-Warnungen ausgeben.
  - **2** - Alle Warnungen ausgeben (Vorgabe)
- **-k** - Luna-Schlüsselwörter exportieren in 'keywords.txt'
- **-h** - Hilfe anzeigen

BEISPIELAUFRUF

Windows

```
c:\lunaavr\lavrc.exe -v -ohbera -w1 "C:\Projekte\helloworld.luna"
```

Linux

```
/home/user/lunaavr/lavrc -v -ohbera -w1 "/home/user/projekte/helloworld.luna"
```

Basisbefehlssatz

ALPHABETISCHE ÜBERSICHT

#

- #Define - Defines/Aliase.
- #Undef - Defines/Aliase entfernen.
- #ide - IDE-Steuerung im Source.
- #if #elseif #else #endif - Bedingtes Kompilieren
- #select #case #default #endselect - Bedingtes Kompilieren
- #macro #endmacro - Luna-Code Makros
- #Include - Quelltextdateien einbinden.
- #IncludeData - Binärdaten in den Flash einbinden.
- #Library - Externe Bibliothek einbinden.
- #pragma, #pragmaSave, #pragmaRestore, #pragmaDefault - Compiler-Steuerung im Source.
- #cdecl, #odecl, #idecl - Parameterdefinition für indirekte Aufrufe.
- #error - Fehlermeldung ausgeben (Kompiliervorgang abbrechen).
- #warning - Warnmeldung ausgeben.
- #message - Nachricht ausgeben.

A

- Abs() - Absolutwert ermitteln.
- Add - Integer-Addition (Befehls-Syntax).
- Asc() - Liefert von allen Zeichen die Wertigkeit, z.B. asc(" ") = 32.
- Asr - Arithmetisches Bitschieben nach rechts.
- Asm-EndAsm - Inline Assembler.
- Avr - Basisklasse AVR-Controller.

B

- Break - Vorzeitiges Verlassen von Schleifen.
- BcdEnc() - BCD Kodierung.
- BcdDec() - BCD Dekodierung.
- Bin() - Umwandlung in Binär-Textdarstellung (String).
- Byte() - Konvertierung von Datentypen.
- byte1() - 1. Byte eines 16 oder 32-Bit-Wertes
- byte2() - 2. Byte eines 16 oder 32-Bit-Wertes
- byte3() - 3. Byte eines 32-Bit-Wertes
- byte4() - 4. Byte eines 32-Bit-Wertes
- ByVal, byRef - (Schlüsselwort)
- ByteToInt16 - Vorzeichenkonvertierung.
- ByteToInt32 - Vorzeichenkonvertierung.

C

- Call - Methode oder Inline-Assembler-Unterprogramm anspringen.
- Ce16() - Konvertierung zwischen LittleEndian/BigEndian.
- Ce32() - Konvertierung zwischen LittleEndian/BigEndian.
- Chr() - ASCII-Zeichen erzeugen (String).
- Cli - Globale Interrupts ausschalten.
- Class-EndClass - Benutzerdefinierte Klassen.
- Continue - Schleife fortsetzen mit nächster Iteration.
- CountFields() - Anzahl Elemente einer separierten Zeichenkette ermitteln.
- Clr/Clear - Variableninhalt zurücksetzen auf Null/Nil.
- Cos() - Integer Cosinus.
- Const - Konstante anlegen.

D

- .db - Datenblock einleiten (Byte).
- .dw - Datenblock einleiten (Word).
- .dl - Datenblock einleiten (Long).
- Data-EndData - Datenobjekt anlegen (Flash).
- Descriptor() - Aktuelle Zeigerposition des Assemblers lesen.
- Defined() - Prüfen ob eine Konstante oder ein Symbol definiert ist.
- Deklaration - (Begriff)
- Decr - Schnelles Dekrementieren.
- Dim - Variablendimensionierung im Arbeitsspeicher.
- Div - Integer-Division (Befehls-Syntax).
- Do-Loop - Schleife mit optionaler Endbedingung.
- eeDim - Variablen dimensionieren (Eeprom)

E

- Eeprom-EndEeprom - Datenobjekt anlegen (Eeprom).
- Exception-EndException - Debugfunktion.
- Exception - (Begriff)
- Exit - Vorzeitiges Verlassen von Schleifen.
- Even() - Integerwerte auf gerade prüfen.
- Event-EndEvent - Event anlegen.

F

- Fabs() - Fließkomma - Absolutwert ermitteln.
- Facos() Fließkomma - ArcusCosinus berechnen (Radiant)
- Fadd() Fließkomma - Addition (Befehls-Syntax/Funktion)
- Fdiv() Fließkomma - Division (Befehls-Syntax/Funktion)
- Fmul() Fließkomma - Multiplikation (Befehls-Syntax/Funktion)
- Fsub() Fließkomma - Subtraktion (Befehls-Syntax/Funktion)
- Fasin() Fließkomma - ArcusSinus berechnen (Radiant)
- Fatan() Fließkomma - ArcusTangens berechnen (Radiant)
- Fcbrt() Fließkomma - Kubikwurzel berechnen
- Fceil() Fließkomma - Auf Ganzzahl aufrunden
- Fcosh() Fließkomma - Hyperbolischen Cosinus berechnen (Radiant)
- Fcos() Fließkomma - Cosinus berechnen (Radiant)
- Fdeg() Fließkomma - Umrechnung Radiant nach Grad
- Feven() Fließkomma - Prüfen auf gerade.
- Fexp() Fließkomma - Exponentialwert berechnen
- Fix() Fließkomma - In Ganzzahl wandeln.
- Flexp() Fließkomma - Umkehrfunktion von Frexp
- Flog10() Fließkomma - Logarithmus zur Basis 10 berechnen
- Flog() Fließkomma - Natürlichen Logarithmus berechnen
- Floor() Fließkomma - Abrunden zur nächsten Ganzzahl, abhängig vom Vorzeichen
- Fodd() Fließkomma - Prüfen auf ungerade.
- Format() - Formatierter Dezimalstring.
- For-Next - Schleife mit Zähler.
- Fpow() Fließkomma - Potenz y von x berechnen
- Frad() Fließkomma - Umrechnung Grad nach Radiant
- Frac() Fließkomma - Vorkommastellen abschneiden
- Frexp() Fließkomma - Wert in Mantisse und Exponent aufteilen
- Fround() Fließkomma - Arithmetisch runden
- Fsine() Fließkomma - 360° Wellenfunktion, Winkel in Grad
- Fsinh() Fließkomma - Hyperbolischen Sinus berechnen
- Fsin() Fließkomma - Sinus berechnen (Radiant)
- Fsplit() Fließkomma - Wert in Ganzzahl und rationale Zahl aufteilen
- Fsqrt() Fließkomma - Quadratwurzel berechnen
- Fsquare() Fließkomma - Quadrat berechnen
- Ftanh() Fließkomma - Hyperbolischen Tangens berechnen
- Ftan() Fließkomma - Tangens berechnen (Radiant)
- Ftrunc() Fließkomma - Abrunden zur nächsten Ganzzahl.
- Function-EndFunc - Methode mit Rückgabewert anlegen.
- Fval() - Zeichenkette mit dezimaler Fließkommazahl nach Binärwert konvertieren.

G

- GarbageCollection() - Manuelle GarbageCollection.

H

- Halt() - Endlosschleife ohne Inhalt.
- Hex() - Wertkonvertierung nach Hexdarstellung (String).
- HexVal() - Umwandlung Hexadezimalwert aus Text in Integerwert.

I

- Icall - Indirekter Funktionsaufruf über Funktionspointer.
- Idle-EndIdle - Idle-Event anlegen.
- If-ElseIf-Else-Endif - Fallunterscheidung durch Bedingungen.
- Incr - Schnelle Inkrementierung.
- InpStr - Eingabestring von der 1. seriellen Schnittstelle lesen.
- Instr() - Text in Text suchen (String).
- Isr-EndIsr - Interrupt-Serviceroutine anlegen.

J

- Jump - Label oder Adresse anspringen.

L

- Label - (Begriff)
- Left() - Linken Teil eines Textes lesen (String).
- Len() - Länge eines Textes lesen (String).
- Lower() - Text in Kleinbuchstaben wandeln (String).
- Long() - Konvertierung von Datentypen.

M

- Max() - Größten Wert aus einer Liste von Werten ermitteln.
- Median16u() - Median Berechnung.
- Median16s() - Median Berechnung (Vorzeichenbehaftet).
- MemCmp() - Speicherbereiche im Arbeitsspeicher vergleichen.
- MemCpy() - Speicherbereiche im Arbeitsspeicher kopieren (schnell).
- MemSort() - Alle Bytes eines Speicherbereichs aufwärts sortieren.
- MemRev() - Alle Bytes eines Speicherbereichs reversieren.
- MemoryBlock - MemoryBlock-Objekt.
- Mid() - Teil eines Textes lesen (String).
- Min() - Kleinsten Wert aus einer Liste von Werten ermitteln.
- Mod - Modulo-Operator
- Mul - Integer-Multiplikation (Befehls-Syntax).
- Mkb() - Wert (byte) zu String konvertieren.
- Mkw() - Wert (word) zu String konvertieren.
- Mkt() - Wert (uint24) zu String konvertieren.
- Mkl() - Wert (long) zu String konvertieren.
- Mks() - Wert (single) zu String konvertieren.

N

- Nop - Leerbefehl.
- NthField() - Element separierter Zeichenketten lesen (String).

O

- Odd() - Integerwert auf ungerade prüfen.

- Operatoren - Liste der Operatoren

P

- Part-EndPart - Optisches Hervorheben von Codebereichen (IDE).
- Pascal-String - (Begriff)
- Pointer - Pointer (Sram, Flash und Eeprom).
- Präprozessor - Präprozessor (Beschreibung)
- Print - Universelle Ausgabefunktion auf serieller Schnittstelle.
- Procedure-EndProc - Methode anlegen.
- Push8()/PushByte() - 8-Bit-Wert auf dem Stack ablegen.
- Push16() - 16-Bit-Wert auf dem Stack ablegen.
- Push24() - 24-Bit-Wert auf dem Stack ablegen.
- Push32() - 32-Bit-Wert auf dem Stack ablegen.
- Pop8()/PopByte() - 8-Bit-Wert vom Stack holen.
- Pop16() - 16-Bit-Wert vom Stack holen.
- Pop24() - 24-Bit-Wert vom Stack holen.
- Pop32() - 32-Bit-Wert vom Stack holen.

R

- Replace() - Zeichenkette in einem String suchen und erste Vorkommende ersetzen.
- ReplaceAll() - Zeichenkette in einem String suchen und alle Vorkommenden ersetzen.
- Reset - Controller neustarten.
- Return - Rückkehrbefehl aus Methoden.
- Right() - Rechten Teil eines textes lesen (String).
- Rinstr() - Text in einem Text suchen (String).
- Rnd() - Zufallszahl erzeugen.
- Rol - Bitweises rotieren nach links.
- Ror - Bitweises rotieren nach rechts.
- Rnd() - Pseudo-Zufallszahl 8 Bit.
- Rnd16 - Pseudo-Zufallszahl 16 Bit.
- Rnd32 - Pseudo-Zufallszahl 32 Bit.

S

- Seed - Pseudo-Zufallszahl 8 Bit (Initialisierung).
- Seed32 - Pseudo-Zufallszahl 32 Bit (Initialisierung).
- Select-Case-Default-EndSelect - Schnelle Fallunterscheidung.
- Sei - Globale Interrupts einschalten.
- Shl - Bitweises schieben nach links.
- Shr - Bitweises schieben nach rechts.
- Sin() - Integer Sinus.
- Sine() - Schnelle Integer-Sinuswellenfunktion.
- Single() - Konvertierung von Datentypen.
- Spc() - Leerzeichen erzeugen (String).
- Sram - Arbeitsspeicher als Datenobjekt.
- Static - Schlüsselwort bei Dimensionierung von Variablen.
- Struct-EndStruct - Benutzerdefinierte Strukturen deklarieren.
- Str() - Binärwert in Dezimalzahl konvertieren (String).
- StrFill() - Text mit Text füllen (String).
- String - (Begriff)
- Sub - Integer-Subtraktion (Befehls-Syntax).
- Sqrt() - Integer-Wurzel.
- Swap - Tauschen von Variablenwerten oder Variableninhalten.

T

- Trim() - Führende/Abschließende nicht sichtbare Zeichen entfernen (String).

U

- Upper() - Text in Großbuchstaben wandeln (String).

V

- Val() - Zeichenkette mit Dezimalzahl zu Integerwert konvertieren.
- Void - Schlüsselwort für Funktionsaufruf.
- Void() - Dummymethode.

W

- Wait - Wartefunktion.
- Waitms - Wartefunktion.
- Waitus - Wartefunktion.
- When-Do - Zeilenbezogene Kurzbedingung.
- While-Wend - Schleife mit Startbedingung.
- Word() - Konvertierung von Datentypen.
- WordToInt16 - Vorzeichenkonvertierung.
- WordToInt32 - Vorzeichenkonvertierung.

#Define

Mit ***#Define*** können Befehle, Befehlskombinationen oder Ausdrücke im Sinne eines **Alias** mit einem Bezeichner verknüpft werden. Der Bezeichner kann dann wiederum im Sourcecode so genutzt werden, als würde es sich um den zugewiesenen Ausdruck handeln. Anstelle des Platzhalter setzt der Compiler die dem Namen zugewiesenen Ausdruck während des Kompiliervorgangs automatisch ein.

#Define dient der besseren Lesbarkeit und Anpassungsmöglichkeit von Programmen oder Programmteilen. Sollen zum Beispiel zu einem späteren Zeitpunkt zugewiesene Ports für ein anderes Projekt geändert werden, müssen nur die Ports der ***Defines*** geändert werden und nicht der gesamte Quelltext. Auch können wiederkehrende Ausdrücke als kürzere Makrofunktion eine nachträgliche Änderung erleichtern und die Lesbarkeit verbessern.

DEFINE-FUNKTIONEN

Es können zusätzlich auch sog. **Define-Funktionen** erstellt werden. D.h. es kann ein *virtueller* Funktionsname anstatt eines Ausdrucks verwendet werden. Die Parameter werden dann durch Textersetzung in *Ausdruck* rechts eingesetzt. Siehe hierzu Beispiel 2 und 3.

Hinweise

- Defines sind immer global, wirken sich also auf den gesamten Quelltext aus!
- Bereits erstellte Defines können wie Konstanten erneut Definiert werden. Die alte Definition wird dabei durch die Neue ersetzt.

SYNTAX

- ***#Define*** *Bezeichner* ***as*** *Ausdruck*
- ***#Define*** *Bezeichner(parameter)* ***as*** *Ausdruck*

Siehe auch: Ablauf des Kompiliervorgangs

BEISPIEL 1

```
#define Taster as Portb.4
Taster.mode = Output, pulldown
```

BEISPIEL 2

```
#define BV(n) as (1 << (n))
a and= BV(3)  'wird zu a and= (1 << (3))
```

BEISPIEL 3

```
#define myfunc(a,b) as ((a + b) * a)
dim var1,var2,result as byte
result = myfunc(var1,var2)  '"myfunc(var1,var2)" wird zu ((var1 + var2) * var1)
```

#Undef

Implementiert ab Version	2015.r2

Mit ***#Undef*** werden erstellte Defines entfernt. Unbekannte Defines erzeugen eine Warnung.

SYNTAX

- ***#Undef*** *Bezeichner*

BEISPIEL

```
#define Taster as Portb.4

Taster.mode = output, low

#undef Taster
Taster.mode = output, low  'Fehler, "Taster" ist hier nicht mehr bekannt
```

#ide

#ide leitet ein IDE-Kommando ein. IDE-Anweisungen werden vom Compiler ignoriert.

Syntax: ***#ide***

ANWEISUNGEN

UploaderPreset="*Presetname*"	Anweisung zur Auswahl eines Presets des Uploaders/Programmers. Der Text ist die Bezeichnung des Presets in der Auswahl.

Präprozessor-Bedingungen

Dient dem fallunterschiedenen Kompilieren von Luna-Programmteilen. Die Ausdrücke erwarten Werte aus Konstanten bzw. Funktionen welche solche Verarbeiten.

#IF #ELSEIF #ELSE #ENDIF

- ***#if*** *Ausdruck*
  - Assemblercode
- ***#elseif*** *Ausdruck* [1)]
  - Assemblercode
- ***#else***
  - Assemblercode
- ***#endif***

#SELECT #CASE #DEFAULT #ENDSELECT

- ***#select*** *Ausdruck*
- ***#case*** *Ausdruck [, Ausdruck1 , Ausdruck2* ***to*** *Ausdruck3, ..]*
  - Assemblercode
- ***#default***
  - Assemblercode
- ***#endselect***

Siehe auch: defined(), Ablauf des Kompiliervorgangs

1) implementiert ab Version 2014.r1

Präprozessor - Makros (Luna)

Implementiert ab Version	2015.r1

Deklaration

- ***#Macro*** *Bezeichner[(**parameter1**, **parameter2**, ..)]*
  - (Luna-Code/Makroaufrufe)
- ***#EndMacro***

Parameterzugriff im Makro

- **@parameter1** = Parameter1
- **@parameter2** = Parameter2
- [..]

BEISPIEL

Dies deklariert ein Makro, welches 2 Parameter erwartet

```
#macro muladd(arg1,arg2)
  @arg1 = @arg1*@arg2+@arg2
#endmacro
```

Aufruf im Luna-Quelltext:

```

dim a,b as word

muladd(a,b)
```

#Include

(Hinweis: Befehl in Versionen vor 2012.r7.1 noch ohne "#" am Anfang, Syntax geändert.)

Einbinden extern gespeicherter Sourcode-Dateien an aktueller Position.

Syntax: ***#Include*** *"Dateipfad"*

Dateipfad: Zeichenkette (Konstante) mit absolutem oder relativem Dateipfad zur einzubindenden Datei. Es sind "/" oder "\" im Pfadnamen erlaubt.

Beispiel:

```
#include "Bibliotheken/ExtraFunktionen.luna"
```

#IncludeData

Dient dem Einbinden von Binärdaten aus einer Datei als objektbasierte Datenstruktur im Programmsegment (Flash).

Syntax:

- **#IncludeData** Bezeichner,"Dateipfad" *[**at** Adresskonstante]*[1)]

Die Daten werden mit dem angegebenen Bezeichner verknüpft und können wie ein normales Datenobjekt benutzt werden.

Optionale Adresskonstante siehe Datenobjekt (Flash)

1) Setzen an feste Adresse ab 2015.R1

#pragma, #pragmaSave/Restore/Default

Implementiert ab Version	2015.r1

Präprozessordirektiven zur Steuerung von Compiler-Internen Vorgängen.Die Pragma-Direktive wird *textuell* wie alle anderen Direktiven im Quelltext verarbeitet.

Syntax

- ***#pragma***
- ***#pragmaSave*** - Aktuelle Werte aller Pragmas zwischenspeichern.
- ***#pragmaRestore*** - Zwischengespeicherte Pragmawerte wiederherstellen.
- ***#pragmaDefault*** - Vorgabewerte Werte aller Pragmas wiederherstellen.

Bei ***#pragmaSave/Restore*** ist eine Verschachtelung möglich. Die Tiefe ist dabei nicht begrenzt. ***#pragmaDefault*** wirkt sich nicht auf zwischengespeicherte Pragmawerte aus.

ANWEISUNGEN

Name	Beschreibung	Vorgabewert
DebugBuild	Wenn aktiviert, wird der Source als **Debugbuild** kompiliert, aktuelle Code-Zeile und Datei können im Code gelesen werden, z.B. zur Anzeige eines Fehlers in einer Exception.	false
OptimizeCode	Schaltet die Code-Optimierung ein oder aus.	true
MethodSetupFast	Wenn aktiviert, wird die Methodeninitialisierung geschwindigkeitsoptimiert direkt in der Methode durchgeführt, anstatt eine Unterfunktion aufzurufen. Führt zur Erzeugung von mehr Maschinencode.	false
MethodLocalVarInit	Aktiviert oder Deaktiviert die Initialisierung lokaler, temporärer Variablen auf ***0*** oder ***nil*** in Methoden. **Wirkt sich bei Deaktivierung nur in Verbindung mit *MethodSetupFast* aus.**	true
MemoryBlockGarbageCollection	Aktiviert oder Deaktiviert die automatische Garbage-Collection des dynamischen verwalteten Arbeitsspeicherbereich der für Strings und MemoryBlocks verwendet wird. Wenn deaktiviert ist es sehr wichtig die Funktion GarbageCollection() manuell aufzurufen, um einen Speicherüberlauf zu vermeiden!	true
MemoryBlockProcessAtomic	Aktiviert oder Deaktiviert die atomare Verarbeitung der Speichermanagementroutinen. Die atomare Verarbeitung deaktiviert die globalen Interrupts während der Ausführung einer Management-Routine. Dies kann deaktiviert werden wenn keine reentrante Speicherverwaltung notwendig ist oder um Jitter von Interrupts zu vermeiden.	true

#cdecl, #odecl, #idecl

Direktive zur Definition einer Parameterliste für den indirekten Aufruf von Methoden mittels lcall.Die Direktive definitiert wieviele und welche Art Parameter einer Parameterübergabe aussehen.

Dabei bedeutet:

- **#cdecl:** Alle Parameter werden über den Stack übergeben.
- **#odecl:** 1. Parameter wird über Registerblock A übergeben, Nachfolgende über den Stack **(Luna Vorgabe)**.
- **#idecl:** 1. und 2. Parameter werden über Registerblock A und B übergeben, Nachfolgende über den Stack. Jedoch darf der erste Parameter hierbei *kein* Ausdruck sein, sondern nur eine einzelne Variable, Adresse, Konstante o.Ä.

Syntax: ***#xdecl*** *Bezeichner([] , ..)*

Ein Beispiel für indirekte Aufrufe ist in der Beschreibung zum Befehl lcall zu finden.

Icall()

Icall() Ist eine Funktion zum indirekten Aufruf von Methoden über *Funktionspointer*.

SYNTAX

- **Icall**(*Adresse*[, *Parameterdefinition(Parameter1, Parameter2, ..)*])

Die **Parameterdefinition** wird genutzt, wenn Parameter der Methode übergeben werden sollen.**Adresse** ist die Adresse der Methode im Programmspeicher (Flash).

WICHTIGE HINWEISE

- Für indirekte Aufrufe von Luna-Methoden im Programm, wird ***#odecl*** erwartet.
- Bei Controllern mit mehr als 128k Flash wird das Bank-Register "RAMPZ" und der Assemblerbefehl "eicall" verwendet. "RAMPZ" speichert auf dem Avr die oberen Bits (16-24) einer Flashadresse. D.h. verwendet die aufgerufene Methode *"von Hand" erstellte Assembler-Funktionen/Befehle zum Flashzugriff* oder kann durch solche Funktionen unterbrochen werden (ISR mit solchen Funktionen), muss am Beginn der aufgerufenen Methode das Register "RAMPZ" auf Null zurückgesetzt werden. Nach Rückkehr vom Aufruf wird das Register "RAMPZ" jedoch automatisch zurückgesetzt.

Siehe auch: #cdecl, #odecl, #idecl

BEISPIEL

```
' ------------------------------------------
' ICALL Example
' Beispiel zur Nutzung von Funktionspointern
' ------------------------------------------
' System Settings
' ------------------------------------------
const F_CPU  = 20000000
avr.device = atmega328p
avr.clock = F_CPU
avr.stack = 48

uart.baud = 19200
uart.Recv.Enable
uart.send.enable

#define LED1 as portd.5
#define LED2 as portd.6
#define LED3 as portd.7

'Funktions prototyp
'Deklariert wieviele und welche Art Parameter einer Parameterübergabe aussehen
'Dabei bedeutet:
' #cdecl = Alle Parameter werden über den Stack übergeben.
' #odecl = 1. Parameter wird über Registerblock A übergeben,
'          Nachfolgende über den Stack.
' #idecl = 1. und 2. Parameter werden über Registerblock A und B übergeben,
'          Nachfolgende über den Stack. Jedoch darf der erste Parameter hierbei KEIN
'          Ausdruck sein, sondern nur eine einzelne Variable, Adresse, Konstante o.Ä.
' Syntax: #cdecl  Bezeichner( [] , .. )

#odecl byte func_p(byte)
#odecl void proc_p(byRef byte,word)
'        |     |     |     |    |
'        |     |     |     |    +Datentyp 2.Parameter
'        |     |     |     +Datentyp 1.Parameter
'        |     |     +Übergabetyp (optional), Referenz oder Wert(Vorgabe)
'        |     +Bezeichner der Parameterdeklaration
'        +Rückgabetyp, bei Proceduren "void"
'
LED1.mode = output
LED2.mode = output
LED3.mode = output

dim a,b as byte
dim ptr1,ptr2,ptr3 as word

print 12;

'Funktionspointer der Methoden erstellen
ptr1=test().Addr
ptr2=myproc.Addr
ptr3=myfunc.addr

print "ptr1 = 0x";hex(ptr1)
print "ptr2 = 0x";hex(ptr2)
print "ptr3 = 0x";hex(ptr3)

do
```

```
  icall(ptr1)
  icall(ptr2,proc_p(a,b))
  b=icall(ptr3,func_p(b))
  print 13;
loop

procedure test()
  LED1.toggle
  print " | test()";
  waitms 300
endproc

function myfunc(a as byte) as byte
  LED3.toggle
  incr a
  print " | myfunc(): a=0x";hex(a);
  waitms 300
  return a
endfunc

procedure myproc(byRef a as byte,b as word)
  LED2.toggle
  incr a
  print " | myproc(): a=0x";hex(a);", b=0x";hex(b);
  waitms 300
endproc
```

#error, #warning, #message

Direktive zur Ausgabe einer Fehler- Warn- oder Infonachricht.

Syntax	Hinweis
#error *"Nachricht"*	Bricht den Kompiliervorgang ab.
#warning *"Nachricht"*	
#message *"Nachricht"*	Ausgabe ohne Zeilennummer und Quelle.

Abs()

Abs() gibt den Absolutwert des übergebenen Integerwertes zurück.

Syntax: ***integer*** = ***Abs***(*Integer*)

Beispiel:

```
dim a as integer
a = -123
a = Abs(a)  ' Ergebnis: 123
a = 123
a = Abs(a)  ' Ergebnis: 123
```

Add,Sub,Mul,Div

Arithmetische Funktionen in Befehlsform mit einer Variable (SRAM) und einer Konstante (technisch kein Unterschied zu Ausdrücken).

Syntax

- ***Add*** *Variable, Konstante*
- ***Sub*** *Variable, Konstante*
- ***Mul*** *Variable, Konstante*
- ***Div*** *Variable, Konstante*

Beispiel:

```
dim a as integer
add a, 1000  ' Gleichbedeutend zu a += 1000, oder a = a + 1000
```

Asc()

Asc() gibt den Ascii-Bytewert des ersten Zeichens der übergebenen Zeichenkette zurück.

Präprozessor	Die Funktion ist zusätzlich im Präprozessor verfügbar, führt mit ausschließlich Konstanten als Parameter also nicht zur Erzeugung von Maschinencode.

Syntax: ***byte*** = ***Asc****(a as string)*

Beispiel:

```
dim a as byte
a = asc("A") ' Ergebnis: 65
```

Shl, Shr, Asr

Shl, Shr: *Logisches* Bit-Shiften nach links (Shl) oder rechts (Shr). Bei Integer-Datentypen, wie z.B. byte, word , usw. führt das Verschieben der Bits um eine Stelle nach links zu einer Multiplikation mit 2, Verschieben der Bits um eine Stelle nach rechts zur Division mit 2. **Asr**: *arithmetische* Variante von **Shr**. *Bei vorzeichenbehafteten* Integer-Datentypen wie z.B. int8, int16, integer, usw. notwendig, *damit das Vorzeichenbit nicht zerstört wird*.

SYNTAX:

- **Shl** *Variable, Bits*
- **Shr** *Variable, Bits*
- **Asr** *Variable, Bits*

Für ***Variable*** ist ausschließlich eine ***Integer-Variable aus dem Arbeitsspeicher*** erlaubt. Für Integer-Variablen ohne Vorzeichen können alternativ auch die Bitschiebe-Operatoren "<<" bzw. ">>" verwendet werden.

BEISPIEL:

```
dim a as word
a=0b0000000100000000
Shl a, 4  ' 4 Bits nach links, Ergebnis: 0b0001000000000000
```

Siehe auch: Add, Sub, Mul, Div, Bitschiebe-Operatoren "<<" bzw. ">>"

Avr

Avr ist die Basis-Klasse (der Controller). **Die Eigenschaft "Device" muss als *erste* Eigenschaft vor allen Anderen gesetzt werden.**

Eigenschaften (nur schreiben)		
Name	**Beschreibung**	**Wert**
Avr.Device=Controllername	Controller-Klasse wählen[1)]	Name
Avr.Clock=Konstante	Taktrate in Hz [2)]	> 0
Avr.Stack=Konstante	Stackgröße in Bytes.	> 1
Avr.Isr.=*Isr-Label*	Direktes Setzen einer Interrupt-Serviceroutine.[3)]	Adresse
Avr.CodeStartAddr=Adresse	Startadresse des Codes setzen, z.Bsp. für Bootloader "avr.CodeStartAddr = LARGEBOOTSTART". Führt dazu, dass der erzeugte Maschinencode ab der angegebenen Adresse im Programmspeicher geschrieben wird.	Adresse (Wordorientiert)
Eigenschaften (nur lesen)		
Name	**Beschreibung**	**Typ**
Ergebnis = Avr.StackPointer	Aktuelle Position des Stackpointers	word
Eigenschaften (lesen/schreiben)		
Name	**Beschreibung**	**Typ**
Avr.[.]	Direkter Zugriff auf sämtliche Ports des Controllers. Die optionale Angabe eines Datentyps (byte, word, integer, long, ..) spezifiziert die Zugriffsart beim lesen/schreiben.	Vorgabe: byte
Avr..	Direkter Zugriff auf die Bits des angegeben Ports.	byte
Methoden		
Name	**Beschreibung**	
Avr.Interrupts.enable	Globale Interrupts *ein*schalten	
Avr.Interrupts.disable	Globale Interrupts *aus*schalten	
Avr.Idle	Ruft das Idle-Event auf, sofern vorhanden.	

BEISPIEL

```
avr.device = atmega32 // Atmega32
avr.clock = 8000000   // 8 Mhz Taktrate
avr.stack = 32        // 32 Bytes Programmstack
// Programmcode
```

Stack-Größe

Die Größe des benötigten Stacks richtet sich nach der Anzahl der Parameter und lokalen Variablen, sowie der Verschachtelungstiefe von Methodenaufrufen. Bei Verschachtelten Aufrufen addieren sich die benötigten Anzahl bytes. Eine Methode benötigt mindestens 2 bytes (Rücksprungadresse). Hinzu kommen die Parameter und deklarierte Variablen (sofern vorhanden).

Controller-Ports und -Konstanten

Neben den abgebildeten Objekten und Funktionen kann man wie in C/Assembler auf die Ports des Controllers **direkt** zugreifen. Weiterhin sind alle im Datasheet benannten controllerspezifischen Bitnamen bzw. Eigenschaften der jeweiligen Ports standardmäßig definiert. Im Editor der Luna-IDE werden sie bei korrekter Schreibweise/Existenz auf dem aktuell im Source definierten Controller farblich hervorgehoben. Die Ports und Namen werden beim Einladen und Speichern der Source-Datei anhand der **avr.device**-Eigenschaft aktualisiert.

Verwendung

Allgemein gilt:

- *Ohne* Angabe des Klassennamens "Avr" werden sämtliche vom Hersteller vordefinierten Ports und Konstanten als *Konstante* interpretiert
- *Mit* Angabe des Klassennamens "Avr", werden Ports wie Variablen behandelt (die Portinhalte werden gelesen oder geschrieben). Controller-Konstanten werden als normale Konstanten behandelt. Was davon Ports und was Konstanten sind, entnimmt man entweder dem Datenblatt oder in der IDE der entsprechenden Übersicht.

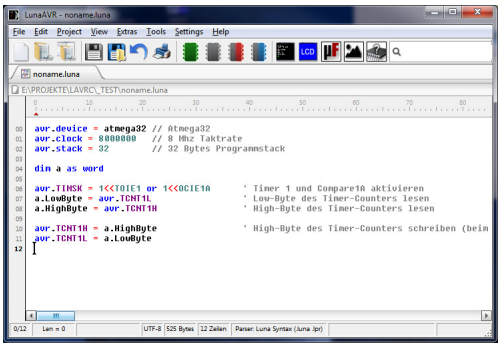
SYNTAX

```
value = avr.SREG        'default datentyp ist byte
avr.TCNT1.Word = value  'wortbreiten Zugriff spezifizieren (schreibend)
```

```
value = avr.TCNT1.Word  '(lesend)
```

Siehe Hierzu: Ermitteln von Controllerspezifischen Werten und Konstanten.

Beispiel




```
avr.device = atmega32 // Atmega32
avr.clock = 8000000   // 8 Mhz Taktrate
avr.stack = 32        // 32 Bytes Programmstack

dim a as word
dim b as byte

avr.TIMSK = 1<<TOIE1 or 1<<OCIE1A      ' Timer 1 und Compare1A aktivieren
a.LowByte = avr.TCNT1L                 ' Low-Byte des Timer-Counters lesen
a.HighByte = avr.TCNT1H                ' High-Byte des Timer-Counters lesen

avr.TCNT1H = a.HighByte                ' High-Byte des Timer-Counters schreiben (beim Schreiben immer High-byte zuerst!)
avr.TCNT1L = a.LowByte                 ' Low-Byte des Timer-Counters schreiben
avr.DDRA.PINA0 = 1
b = avr.PORTA.PINA0
avr.Isr.ICP1Addr = myIsr

do
loop

isr myIsr
 [..]
endisr
```

1) Avr.Device ist auch lesbar, gibt jedoch einen String als Konstante zurück

2) Das setzen dieser Eigenschaft beeinflusst nicht die physikalische Taktrate, sondern definiert sie für verschiedene darauf zugreifende Funktionen, damit diese wie erwartet funktionieren können.

3) ab Version 2012.r4. ***InterrupAdr*** ist die Hardware-Interruptadresse im Interruptvektor des Controllers, z.Bsp. ***ICP1Addr***.

Exit/Break

Befehl zum vorzeitigen Verlassen einer Schleife, unabhängig von den normalen Schleifenbedingungen.

Beispiel1:

```
' Variante 1
for i=0 to 10
  if i>6 then
    exit
  endif
next

' Variante 2
for i=0 to 10
  when i>6 do exit
next
```

Beispiel2:

```
' Variante 1
while a>0
  if value=20 then
    exit
  endif
wend

' Variante 2
while a>0
  when value=20 do exit
wend
```

Beispiel3:

```
' Variante 1
do
  a = uart.read
  if a=13 then
    exit
  endif
loop

' Variante 2
do
  a = uart.read
  when a=13 do exit
loop
```

BcdEnc()

BcdEnc() konvertiert einen 16 Bit Integerwert in einen gepackten 16 Bit Bcd-Wert (4 Dezimalstellen) bzw. einen 8 Bit Integerwert in einen gepackten 8 Bit Bcd-Wert (2 Dezimalstellen).

Syntax: *byte* = ***BcdEnc****(a as word)*

Beispiel:

```
dim a as byte
dim b as word
a=BcdEnc(23)
b=BcdEnc(2342)
a = BcdDec(a)
b = BcdDec(b)
```

Siehe auch: BcdDec(), Aufbau des BCD-Wertes

BcdDec()

BcdDec() konvertiert einen gepackten 16 Bit Bcd-Wert (4 Dezimalstellen) in einen 16 Bit Integerwert (word) bzw. einen gepackten 8 Bit Bcd-Wert (2 Dezimalstellen) in einen 8 Bit Integerwert (byte).

Syntax: *word* = ***BcdDec****(a as word)*

Beispiel:

```
dim a as byte
dim b as word
a=BcdEnc(23)
b=BcdEnc(2342)
a = BcdDec(a)
b = BcdDec(b)
```

Siehe auch: BcdEnc(), Aufbau des BCD-Wertes

Bin()

Zahl in Zeichenkette mit Binärdarstellung konvertieren. Konvertiert automatisch zur Bitlänge des verwendeten Datentyps. Die Ausgabe erfolgt standardisiert wie auch bei der hexadezimalen Darstellung mit geringster Wertigkeit rechts und steigender Wertigkeit nach links (Big-Endian).

Präprozessor	Die Funktion ist zusätzlich im Präprozessor verfügbar, führt mit ausschließlich Konstanten als Parameter also nicht zur Erzeugung von Maschinencode.

Syntax: *String* = ***Bin****(Ausdruck)*

Beispiel:

```
dim s as string
s = Bin(14)  // Ergebnis: "00001110"
```

byte1()..byte4()

Präprozessor	Ausschließlich eine Funktion des Präprozessors

Gibt die einzelnen Bytes der niederwertigsten Stelle bis zur höherwertigsten Stelle vom 32-Bit-Wert einer Konstante oder eines Symbols zurück.

Syntax:

- *byteWert* = **byte1(** *value* **)** - niederwertiges Byte aus niederwertigem Word
- *byteWert* = **byte2(** *value* **)** - höherwertiges Byte aus niederwertigem Word
- *byteWert* = **byte3(** *value* **)** - niederwertiges Byte aus höherwertigem Word
- *byteWert* = **byte4(** *value* **)** - höherwertiges Byte aus höherwertigem Word

Siehe auch: Direktiven, Präprozessor

Beispiel:

```
asm
  '32-Bit-Wert laden
  '
  ' Speicher ->
  ' 4e 61 bc 00
  '  :  :  :  :
  '  :  :  :  +-byte4()
  '  :  :  +-byte3()
  '  :  +-byte2()
  '  +-byte1()
  '
  ldi _HA0,byte1(12345678)
  ldi _HA1,byte2(12345678)
  ldi _HA2,byte3(12345678)
  ldi _HA3,byte4(12345678)
endasm
```

byVal, byRef

byVal und byRef sind Schlüsselwörter die bei der Deklaration von Methoden verwendet werden können.

byVal

Der nachfolgende Parameter soll als Kopie der Methode übergeben werden (Vorgabe). Das Schlüsselwort "byVal" ist damit im Grunde obsolet, da die Übergabe ohne Angabe von byVal oder byRef immer als Kopie erfolgt.

Kopie meint, dass die Quelle von der der Wert stammt unangetastet bleibt.

byRef

Der nachfolgende Parameter soll als Referenz der Methode übergeben werden.

Referenz meint, dass ein Adresszeiger auf das übergebene Objekt/Variable als Parameter der Methode übergeben wird und man den Wert des Objekts/der Variable innerhalb der Methode ändern kann.

Beim Aufruf dürfen dann als Übergabeparameter an die Methode nur einzelnde Objekte, Variablen oder Datenstrukturen übergeben werden. Ausdrücke können verständlicherweise nicht Referenziert werden.

Doppelte Deklarationen in Klassen

Werden Klassen verwendet, ist die doppelte Deklaration einer gleichnamigen Struktur in der Klasse und außerhalb der Klasse erlaubt, da es sich um getrennte Namensräume handelt.

Wird jedoch in der Klasse eine Methode mit byRef Statement verwendet, kann der Compiler nicht erkennen, um welche Struktur sich das übergebene Objekt handelt und meldet daher einen Kollisionsfehler. In solchen Fällen die *Strukturdeklaration* eines referenzierten Strukturtyps ausschließlich im Hauptprogramm vornehmen.

Parameter als Kopie übergeben

Beispiel 1:

```
' Hauptprogramm
dim wert as integer
dim s as string

ausgabe("Meine Zahl: ",33) ' Aufruf des Unterprogramms
' Weitere Aufrufmöglichkeit
wert = 12345
s = "Andere Zahl: "
call ausgabe(s,wert)

do
loop

' Unterprogramm
procedure ausgabe(text as string, a as byte)
  dim x as integer  ' lokal gültige Variable
  x=a*100
  print text+str(x)
endproc
```

Parameter als Referenz übergeben

Beispiel 1: Variablen

```
dim a as byte

a=23
test(a)
print "a = ";str(a)    ' Ausgabe: a = 42

do
loop

procedure test(byRef c as byte)
  print "c = ";str(c)  ' Ausgabe: c = 23
  c = 42
endproc
```

Beispiel 2: Strukturobjekte

```
struct point
```

```
  byte x
  byte y
endstruct

dim p as point

p.x=23
test(p)
print "p.x = ";str(p.x)    ' Ausgabe: p.x = 42

do
loop

procedure test(byRef c as point)
  print "c.x = ";str(c.x)  ' Ausgabe: c.x = 23
  c.x = 42
endproc
```

Beispiel 3: Konstantenobjekte

```


test(text)       ' Ausgabe: c.PString(0) = "hallo"
test("ballo")    ' Ausgabe: c.PString(0) = "ballo"

do
loop

procedure test(byRef c as data)
  print "c.PString(0) = ";34;c.PString(0);34
endproc

data text
  .db 5,"hallo"
enddata
```

Beispiel 4: Arbeitsspeicherobjekte

```


dim a(127),i as byte

for i=0 to a.Ubound
  a(i)=i
next

test(a())

do
loop

' über Umweg Direktzugriff auf außenstehendes Array
procedure test(byRef c as sram)
  dim i as byte
  for i=0 to 63
    print str(c.ByteValue(i))
  next
endproc
```

Vorzeichenkonvertierung

BYTETOINT16(), WORDTOINT16(), BYTETOINT32(), WORDTOINT32()

Diese Funktionen konvertieren den Übergabewert explizit in einen vorzeichenbehafteten Wert. Diese Funktion wird benötigt, wenn man beispielsweise einen nicht vorzeichenbehafteten Variablen- oder Rückgabewert Vorzeichenrichtig weiterbearbeiten möchte. Dies ist z.Bsp. der Fall, wenn eine Byte-Variable empfangen worden ist, die statt 0-255 den vorzeichenbehafteten Wertebereich von -127 bis +128 darstellt. Da der Datentyp ***Byte*** nicht vorzeichenbehaftet ist, wären Rechnungen und Darstellung damit ebenfalls nicht vorzeichenbehaftet.

SYNTAX

- *integer* = **ByteToInt16**(*value as byte*)
- *integer* = **WordToInt16**(*value as word*)

- *longint* = **ByteToInt32**(*value as byte*)
- *longint* = **WordToInt32**(*value as word*)

Beispiel:

```
const F_CPU=20000000
avr.device = atmega328p
avr.clock  = F_CPU
avr.stack = 64

uart.baud = 19200
uart.recv.enable
uart.send.enable

dim a as byte
dim b as word

a=0xff
b=0xffff
print str(a)                'Ausgabe: "255"
print str(b)                'Ausgabe: "65535"
print str(ByteToInt16(a))   'Ausgabe: "-1"
print str(WordToInt16(b))   'Ausgabe: "-1"
```

Call

Aufruf eines Unterprogramms (Procedure oder Inline-Assembler-Label).

Syntax:

1. ***Call*** *ProzedurName([Parameter1, Parameter 2, ..])*

Der Aufruf von Prozeduren muss nicht zwingend mit dem "Call"-Befehl erfolgen. folgende Syntax hat denselben Effekt:

1. ***ProzedurName****([Parameter1, Parameter 2, ..])*

Siehe auch: Void

CE16(), CE32()

CE16() konvertiert einen 16 Bit Integerwert, CE32() einen 32 Bit Integerwert von LittleEndian nach BigEndian und umgedreht. Die Byte-Anordnung des Wertes wird umgedreht. Selbe Funktion wie bei der Objektmethode **.Reverse** der Datentypen Integer, Word und Long (siehe Variablen).

Syntax: *word* = ***CE16****(a as word)*
Syntax: *long* = ***CE32****(a as long)*

Beispiel:

```
dim a as word
dim b as long
a=0x1122
b=0x11223344
a = CE16(a)     ' Ergebnis: 0x2211
b = CE32(b)     ' Ergebnis: 0x44332211
```

Chr(), Mkb(), Mkw(), Mkt(), Mkl(), Mks()

Die Funktionen erstellen aus einem numerischen Ausdruck einen String mit den binären Daten des numerischen Wertes im Little-Endian-Format[1].

Präprozessor	Die Funktion ist zusätzlich im Präprozessor verfügbar, führt mit ausschließlich Konstanten als Parameter also nicht zur Erzeugung von Maschinencode.

Syntax

- *string* = ***chr***(*value* ***as byte***)
- *string* = ***mkb***(*value* ***as byte***)
- *string* = ***mkw***(*value* ***as word***)
- *string* = ***mkt***(*value* ***as uint24***)
- *string* = ***mkl***(*value* ***as long***)
- *string* = ***mks***(*value* ***as single***)

Hinweis:
chr() und *mkb()* sind dieselben Funktionen in verschiedenen Schreibweisen.

mks() erzeugt einen String, der den Single-Binärwert nach IEEE-Standard enthält, d.h. jeder beliebige übergebene Wert wird in einen Single-Wert konvertiert und als String zurückgegeben.

Beispiel:

```
[..]
dim a as word
dim b as long
dim s as string

a=1234
b=4567

s = mkw(a)   ' Word-Binärwert als String
s = mkl(b)   ' Long-Binärwert als String
s = mkb(65)  ' Byte-Binärwert als String, dasselbe wie chr(65)
s = mks(a)   ' Single-Binärwert von "a" als String

' Dump-Ausgabe des String-Inhaltes
print "s = ";
for i=1 to len(s)
  print "0x";hex(s.ByteValue(i));" ";
next
print

[..]
```

[1] Auf den Avr-Controllern übliche Wert-Formatierung.

Cli, Sei

Cli und *Sei* sind die Kurzform von Avr.Interrupts.Enable und Avr.Interrupts.Disable.

- ***Cli*** schaltet die Interrupts global aus
- ***Sei*** schaltet die Interrupts global ein

Benutzerdefinierte Klassen

In LunaAVR lassen sich eigene Klassen definieren, mit denen Funktionen im Sinne eines Moduls oder einer in sich geschlossenen Bibliothek nachgerüstet werden können.

Eine solche Klasse kann als Bestandteil des Hauptprogramms implementiert werden und ist in sich geschlossen. Dies bedeutet, dass die enthaltenen Komponenten nur durch den Zugriff über den Namen der Klasse erreichbar sind (getrennter Namensraum: "KlassenName.xxx"). Der getrennte Namensraum erlaubt es daher Bezeichner/Namen für die Komponenten zu verwenden, die auch im Hauptprogramm existieren.

Eine Klasse kann folgende *Komponenten* enthalten:

- Konstanten
- Variablen (Arbeitsspeicher & Eeprom)
- Strukturen
- Proceduren
- Funktionen
- Interrupt-Serviceroutinen
- Konstanten-Objekte
- Eeprom-Objekte

Die einzelnen Komponenten werden innerhalb der Klasse wie im Hauptprogramm deklariert und *sind dann Teil der Klasse*.

Syntax

- ***Class** Bezeichner*
  - *Konstantendeklaration*
  - *Definitionen/Aliase*
  - *Strukturdeklaration*
  - *Variablendimensionierung*
  - *Interrupt-Serviceroutinen*
  - *Funktionen und Proceduren*
  - *Datenstrukturen*
- ***EndClass***

Anwendung von Klassen im Programmcode
Von Außen sind Zugriffe die Komponenten einer Klasse erlaubt:

- Strukturdeklarationen (zur Dimensionierung)
- Konstanten (lesen)
- Variablen (lesen und schreiben)
- Prozeduren (aufrufen)
- Funktionen (aufrufen)
- Daten (lesen und Ggf. schreiben)

Beispiel

```
' Initialisierung
[..]
dim var as single
'Aufruf irgendwo im Programmcode
print test.Version       ' Ausgabe
test.Init()              ' Procedure aufrufen
var = test.GetData(0,2)  ' Funktion aufrufen
test.a=123               ' Wert zuweisen
print str(test.a)        ' Wert lesen und ausgeben

' Hauptschleife
do
loop

' Deklaration
Class test
  const Version = "1.0"
  dim a as byte

  Procedure Init()
    print "Initialisierung"
    print "a = "+str(a)
  EndProc

  Function GetData(x as byte, y as byte) as single
    mul y,4
```

```
    y=y+x
    return tabelle.SingleValue(y)
  EndFunc

  data tabelle
    .dw &h1234,&h5678,&h9aab
    .dw &h1234,&h5678,&h9aab
    .dw &h1234,&h5678,&h9aab
    .dw &h1234,&h5678,&h9aab
  enddata

EndClass
```

Continue

Befehl zum Fortsetzen einer Schleife mit der nächsten Iteration. Alle Befehle die im Schleifenkörper nach **continue** stehen werden übersprungen. Hierbei wird je nach Schleifentyp zur entsprechenden Schleifenbedingung gesprungen.

Beispiel:

```
print "for-next"
for i=0 to 3
  if i=2 then
    continue  'springt zu 'for ..'
  end if
  print str(i)
next

print "while-wend"
clr i
while i<3
  incr i
  if i=2 then
    continue  'springt zu 'while ..'
  end if
  print str(i)
wend

print "do-loop"
clr i
do
  incr i
  if i=2 then
    continue  'springt zu 'loop ..'
  end if
  print str(i)
loop until i>2
```

CountFields()

Anzahl Elemente einer durch Separatoren unterteilten Zeichenkette ermitteln.

Syntax: *byte* = ***CountFields****(source as string, separator as string)*

- **source:** Zeichenkette in der gesucht/gelesen werden soll.
- **separator:** Zeichenkette welche die Elemente untereinander abgrenzen.

Siehe auch: NthField()

BEISPIEL

```
dim a as string
a = "Dies | ist | ein | Beispiel"
print str(CountFields(a,"|")   ' Ergebnis: 4
```

Clr, Clear

Schnelles setzen des Inhalts einer Variable, Arrays, Struktur, Struktureigenschaft auf 0.

Obacht bei Datentyp MemoryBlock! Hier sollte die entsprechende Freigabemethode des Objekts aufgerufen werden anstatt nur die Referenz zu löschen, da sonst Speicherleichen zurückbleiben.

Syntax:

1. ***Clr*** *Variable* - Variablenwert auf Null setzen
2. ***Clr*** *Variable()* - Alle Elementwerte des Arrays auf Null setzen

Info

Variable schließt auch Variablen ein, die als Struktur dimensioniert wurden. In diesem Fall werden alle Werte in der Struktur gelöscht. Auch einzelne Eigenschaften einer solchen Struktur können auf Null gesetzt werden, sowie auch Arrayelemente oder ganze Arrays einer Struktur.

Beispiel1:

```
dim a(100) as byte
dim b as integer
dim s as string

clr a(4)  ' element 5 des Arrays "a" auf 0 setzen
clr a()   ' gesamtes Array "a" auf 0 setzen
clr b     ' b auf 0 setzen
clr s     ' string freigeben/auf "" setzen
```

Beispiel2:

```
' Struktur deklarieren
struct date
  byte hour
  byte minute
  byte second
  byte wert(5)
endstruct
' Variable als Struktur dimensionieren (anlegen)
dim d as date
clr d.hour    ' hour auf 0 setzen
clr d         ' alle Werte auf 0 setzen
clr d.wert(3) ' element 4 des array "wert" auf 0 setzen
clr d.wert()  ' ganzes Array "wert" auf 0 setzen
```

Cos()

Cos() ist eine schnelle 16 Bit Integer-Cosinus-Funktion. Als Parameter erwartet sie einen Winkel in Grad multipliziert mit 10. Das Ergebnis ist ein 16 Bit Integer-Wert von -32768 bis +32767 als Äquivalent zum Cosinuswert von -1 bis +1. Die Funktion nutzt den Cordic Algorithmus zur Berechnung.

Syntax: *integer* = ***Cos****(a as integer)*

Beispiel:

```
dim winkel as byte
dim ergebnis as integer
winkel=23
ergebnis = Cos(winkel*10)
```

.db

Leitet einen Byte-orientierten Datenblock ein. Kann nur in Datenstrukturen verwendet werden.

Beispiel 1, Datenstruktur (Flash):

```
data test
  .db 1,2,"Hallo",0
enddata
```

Beispiel 2, Datenstruktur (Eeprom):

```
eeprom test
  .db 1,2,"Hallo",0
endeeprom
```

.dw

Leitet einen Word-orientierten Datenblock ein. Kann nur in Datenstrukturen verwendet werden.

Beispiel 1, Datenstruktur (Flash):

```
data test
  .dw 1,2,12345,&h4711
enddata
```

Beispiel 2, Datenstruktur (Eeprom):

```
eeprom test
  .dw 1,2,12345,&h4711
endeeprom
```

.dl

Leitet einen Long-orientierten Datenblock ein (4 Bytes). Kann nur in Datenstrukturen verwendet werden.

Beispiel 1, Datenstruktur (Flash):

```
data test
  .dl 1,2,12345,&h4711aabc
enddata
```

Beispiel 2, Datenstruktur (Eeprom):

```
eeprom test
  .dl 1,2,12345,&h4711aabc
endeeprom
```

Data-EndData

Dient der Definition einer objektbasierten Datenstruktur im Programmsegment (Flash).

Syntax:

- ***Data*** *Bezeichner* ***[at*** *Adresskonstante]*[1)]
  - *Daten*
- ***EndData***

Methoden (nur lesbar)		
Name	**Beschreibung**	**Rückgabetyp**
.ByteValue(offset)	Byte lesen	byte
.WordValue(offset)	Word lesen	word
.IntegerValue(offset)	Integer lesen	integer
.LongValue(offset)	Long lesen	long
.SingleValue(offset)	Single lesen	single
.StringValue(offset,bytes)	String lesen mit Längenvorgabe	string
.PString(offset)	Pascal-String lesen (Start-Byte ist Länge)	string
.CString(offset)	C-String lesen (Nullterminiert)	string
.Addr	Adresse der Datenstruktur im Flash	word
.SizeOf	Vom Objekt belegte Anzahl Bytes lesen.	byte

- **offset:** *Byte-Position* innerhalb der Daten mit Basis Null.
  Anm.: Greift man zu Beispiel auf nacheinanderliegende Word-Daten zu (16-Bit-Werte), dann ist der Offset mit 2 zu multiplizieren.
- **bytes:** Anzahl zu lesender Bytes.

DATEN AN FESTER ADRESSE

Das Setzen der Daten ab der angegebenen ***Word-Adresse im Flash***. Eine falsche oder fehlerhafte Adresse wird mit einer Assembler-Fehlermeldung quittiert. Achten sie darauf das es sich hier um eine *Word*-Adresse handelt, wie z.B. die Bootloaderadresse *FIRSTBOOTSTART*. Die Byte-Adresse wäre *(wordAddresse*2)*

Mit dieser Funktionalität ist es möglich Daten an einer ganz bestimmten, festen Position im Programmspeicher (Flash) abzulegen. Wird das kompilierte Programm größer oder gleich dieser Adresse, werden die Daten überschrieben. Daher ist über die Speicherbelegung zu prüfen ob genügend Freiraum vorhanden ist.

DATENABLAGE

Die Direktiven zur Ablage von Daten innerhalb eines Datenobjektes.

- .db - (byte) 8 Bit Werte inkl. Zeichenketten
- .dw - (word) 16 Bit Werte
- .dt - (triple) 24 Bit Werte
- .dl - (long) 32 Bit Werte

Hinweis
Der Zugriff über die Objektgrenzen hinaus ist möglich und wird nicht geprüft.

SUPERIMPOSING

Bei Datenobjekten können Strukturdeklarationen für eine vereinfachte Ablage und auch Zugriff verwendet werden. **Siehe hierzu:** Pointer, Absatz "SuperImpose"

Beispiel

```
' Initialisierung
[..]
' Hauptprogramm
dim a as byte
dim s as string

a=tabelle1.ByteValue(4)         ' Byte lesen von tabelle1 + 4, Ergebnis: 5
s=tabelle1.CString(11)          ' C-String lesen, Ergebnis: "Hallo"
s=tabelle1.StringValue(11,2)    ' String lesen mit Längenvorgabe: Ergebnis: "Ha"
s=tabelle1.PString(17)          ' Pascal-String lesen, erstes Byte ist Länge(10). Ergebnis: "i love you"
Print "&h"+Hex(tabelle1.Addr)   ' Adresse der Datenstruktur im Flash anzeigen

do
```

```
loop

' Datenstruktur im Flash definieren
data tabelle1
  .db 1,2,3,4,5
  .dw &h4af1,&hcc55,12345
  .db "Hallo",0
  .db 10,"i love you",0
enddata
```

STRUKTUREN

Daten ablegen anhand einer Strukturdeklaration.

Beispiel:

```
struct ENTRY
  word   fnAddr
  string text[12]
endstruct

[..]

data table
  ENTRY { myfunc1, "Menu Nr. 1" }
  ENTRY { myfunc2, "Menu Nr. 2" }
  ENTRY { myfunc3, "Menu Nr. 3" }
enddata
```

1) Setzen an feste Adresse ab 2013.R1

Descriptor()

Präprozessor Ausschließlich eine Funktion des Präprozessors

Die Funktion gibt die aktuelle Position des Assembler-Deskriptors zurück, d.h. die aktuelle ***Byte-Adresse*** im Flash, bis zu welcher der Assembler den Assemblercode bereits übersetzt hätte. Mit dieser Funktion kann man vor einer Änderung des Deskriptors mittels der Direktive ".org" die Position lesen und anschließend wiederherstellen.

Syntax:

- **Konstante = descriptor()**

Siehe auch: Präprozessor (Assembler)

BEISPIEL

```
.set OLD_CPC = descriptor()/2 ;aktuelle Position merken
.org  FOURTHBOOTSTART  ;nachfolgenden Asm-Code ab neuer Adresse ablegen
jmp  THIRDBOOTSTART
.org OLD_CPC                    ;vorherige Position wiederherstellen
                                ;nachfolgender Asm-Code wird wieder ab alter Adresse abgelegt.
```

Defined()

Präprozessor	Ausschließlich eine Funktion des Präprozessors

Ermittelt ob ein Symbol oder eine Konstante definiert ist. Die Präprozessorfunktionen sind für #if..#endif-Strukturen vorgesehen und ermöglichen das bedingte Kompilieren von Programmcode in Abhängigkeit vom *Vorhandensein* einer Konstante. **Defined()** ist vergleichbar mit der **#ifdef**-Direktive in C, jedoch hier als Funktion implementiert.

Erlaubt sind:

- Konstanten, auch Prozessorkonstanten wie *avr.TIMSK*
- Label aus dem Assembler-Quelltext
- Bibliotheksnamen wie z.B. *Graphics.interface*

Syntax:

- **defined(** *symbol* **)**
  Das Ergebnis ist wahr, wenn *symbol* definiert ist.

Siehe auch: Direktiven, Ablauf des Kompiliervorgangs

Beispiel

Luna

```
#if defined(avr.TIMSK0)
  'Programmcode
#endif
#if defined(Graphics.interface)
  'Programmcode
#endif
#if defined(mylabel)
  'Programmcode
#endif
```

Assembler

```
.if defined(avr.TIMSK0)
  'Programmcode
.endif
.if defined(Graphics.interface)
  'Programmcode
.endif
.if defined(mylabel)
  'Programmcode
.endif
```

Deklaration (Begriff)

Eine reine Deklaration ist eine Anweisung für den Compiler, wie er beispielsweise ein unter einem Bezeichner beschriebenes Objekt oder eine Struktur zu verstehen hat. Die Deklaration ist somit nicht zwangsläufig eine Anweisung um einen Speicherbereich zu reservieren. Sie dient der semantischen Definition von z.Bsp. Strukturen oder Parameterfolgen.

In Luna werden viele Elemente deklariert und auch gleichzeitig angelegt. Dies trifft z.B. auf Variablen und Methoden zu. Es gibt jedoch auch nur reine Deklarationen wie z.Bsp. Strukturen.

Incr, Decr

Schnelle Inkrementation oder Dekrementation um 1 (x = x +/- 1). Geschwindigkeitsvorteil bei den Datentypen byte, word, integer, long, bei Arrayzugriffen und bei Zugriffen auf Variablen-Eigenschaften die als Struktur dimensioniert wurden. Verarbeitet auch Fließkommawerte, jedoch ohne nennenswerten Geschwindigkeitsvorteil.

Syntax

1. ***Incr*** *Variable*
2. ***Decr*** *Variable*

Syntax (alternativ)

1. ***Variable***++
2. ***Variable***--

Info

Variable schließt auch numerische Eigenschaften von Variablen ein, die als Struktur dimensioniert wurden.

Beispiel1:

```
dim a(100) as byte
dim b as integer

incr a(4)   ' Wert des Elements 5 vom Arrays "a" um 1 erhöhen
decr a(4)   ' Wert des Elements 5 vom Arrays "a" um 1 vermindern
incr b      ' Wert von "b" um 1 erhöhen
```

Beispiel2:

```
' Struktur deklarieren
struct date
  byte hour
  byte minute
  byte second
  byte wert(5)
endstruct
' Variable als Struktur dimensionieren (anlegen)
dim d as date

incr d.hour     ' Wert der Eigenschaft "hour" um 1 erhöhen
incr d.wert(3)  ' Wert des Elements 4 vom Array "wert" um 1 erhöhen
decr d.wert(3)  ' Wert des Elements 4 vom Array "wert" um 1 vermindern
```

Dim, eeDim

Mittels Dim (Arbeitsspeicher) und eeDim (Eeprom) werden Variablen dimensioniert/angelegt.

Syntax:

- ***Dim*** *Name[(Elemente)] [, NameN[(Elemente) as* ***[static]*** *Datatype* [***= InitialValue***][1)]
- ***eeDim*** *Name[(Elemente)] [, NameN[(Elemente) as Datatype* [***= InitialValue***][1)]

Sichtbarkeit von Variablen

- Innerhalb einer Klasse dimensionierte Variablen sind in der gesamten Klasse *lokal* und statisch (static). D.h. z.Bsp. im Hauptprogramm (Klasse "Avr") dimensionierte Variablen sind auch in Methoden des Hauptprogramms sichtbar, sofern keine gleichnamige Variable innerhalb der Methode dimensioniert wurde. Jedoch sind sie nicht in einer anderen Klasse sichtbar. Auf Elemente anderer Klassen kann jedoch zugegriffen werden durch ***Klassenname.Eigenschaft bzw. Methode***

- In Methoden dimensionierte Variablen sind innerhalb der Methode *lokal* (also nur in der Methode sichtbar) und wahlweise temporär oder statisch (***static***).

- Eeprom-Variablen können *nur* in Klassen, z.Bsp. im Hauptprogramm (der Klasse "Avr") dimensioniert werden, nicht in Unterprogrammen. Sie sind ebenfalls statisch.

Namenskollisionen

Wird in einer Klasse, z.Bsp. im Hauptprogramm eine Variable mit dem Namen "myVar"" und in einer seiner Methoden ebenfalls eine Variable mit dem Namen "myVar" dimensioniert (als Parameter oder lokale Variable), dann wird die Variable der Methode bevorzugt behandelt. D.h. die Variable der Klasse wird innerhalb der Methode ausgeblendet (versteckt).

Static

Mit ***static*** werden die Variablen in einer Methode statisch angelegt. D.h. sie sind schon ab Beginn des gesamten Programms im Arbeitsspeicher dimensioniert und werden beim Aufruf einer Methode (im Gegensatz zu temporären Variablen) *nicht* initialisiert[2)] (nur beim Start des Controllers) und behalten somit ihre Werte vom letztmaligem Ausführen der Methode. Die Ausführungsgeschwindigkeit ist um Einiges schneller als bei temporären Variablen. Hierbei ist jedoch zu beachten, dass man die Methode dann nicht parallel mehrfach ausführen sollte, da die parallel ausgeführten Methoden dann auf dieselbe Variable zugreifen. Dies kann zu Fehlern führen, die schwer zu finden sind.

Standardmäßig werden die Variablen temporär innerhalb einer Methode angelegt, d.h. sie existieren nur solange die Methode ausgeführt wird.

InitialValue

Implementiert ab Version	2015.r1

InitialValue ist eine Funktionalität, mit der man den numerischen- und string-Variablen bei der Dimensionierung optional einen Startwert zuweisen kann (auch in Methoden). Hierbei werden alle angegebenen Variablen, auch Arrays mit dem Initialwert belegt.

```
dim a,b(7) as byte = 42
dim s as string = "hallo"
```

Hinweise

ALLGEMEIN

- Bei der Dimensionierung von Strings ist zu beachten, dass Stringdaten im Arbeitsspeicher *dynamisch* sind und im Eeprom *statisch*. Dimensionierung von Stringvariablen im Eeprom bedürfen daher der zusätzlichen Angabe der statischen Länge (Anzahl Zeichen).

EEPROM

- Das Lesen und Schreiben von Eeprom-Variablen ist sehr langsam und die Anzahl der möglichen Schreibzugriffe durch die Hardware begrenzt. Daher Eeprom-Variablen nicht in Schleifen dauerhaft beschreiben!
- Eepromvariablen werden zu Programmstart logischerweise **nicht** wie Arbeitsspeicher-Variablen mit dem Wert 0 initialisiert.

Beispiele

Beispiel Dimensionierung Arbeitsspeicher

```
dim a,b as byte
dim c(19) as integer     ' Word-Array mit 20 Elementen
dim s1 as string
dim s2(4) as string      ' String-Array mit 5 Elementen
```

```
procedure hallo(a as byte)
  dim c(19) as static integer     ' statisches Word-Array mit 20 Elementen
  dim s1 as string
endproc
```

Beispiel Dimensionierung Eeprom

```
eedim a,b as byte
eedim c(19) as integer       ' Word-Array mit 20 Elementen
eedim s1[20] as string       ' String mit 20 Zeichen Speicherplatz
eedim s2[10](4) as string    ' String-Array mit 5 Elementen, jeweils 10 Zeichen Speicherplatz
```

1) as of Version 2015.r1

2) Luna initialisiert Variablen, Objekte und Strukturen die Werte speichern mit dem Wert ***nil***, was dem Zahlenwert 0 entspricht.

Do-Loop

Schleife mit optionaler Bedingung zum Verlassen. Die Do-Loop-Schleife wird immer betreten und abhängig der optional angegebenen Bedingung am Schleifenende verlassen.

Syntax:

- ***Do***
  - *Programmcode*
- ***Loop** [**Until** Ausdruck]*

Beispiel 1

```
Do
  Print "Hallo"
Loop
```

Beispiel 2

```
dim a as byte
Do
  Incr a
Loop Until a>10  ' Verlassen der Schleife wenn a größer 10
```

Eeprom-EndEeprom

Dient der Definition einer Objekt-Datenstruktur im Eeprom (Syntax 1) bzw. als Ausdruck (Syntax 2+3) dem Zugriff auf den kompletten Eeprom-Speicher.

Bei Schreibzugriffen über Objekt-Datenstrukturen ist darauf zu achten, dass die Grenzen der Struktur nicht überschritten werden, da sonst möglicherweise nachfolgende Daten überschrieben werden.

Syntax 1:

- ***Eeprom*** *Bezeichner*
  - *Daten*
- ***EndEeprom***

Syntax 2:

- **Eeprom**.*Methode/Eigenschaft* = Ausdruck

Syntax 3:

- *Ergebnis* = **Eeprom**.*Methode/Eigenschaft*

Methoden (nur lesbar)		
Name	**Beschreibung**	**Rückgabetyp**
.ByteValue(offset)	Byte lesen	byte
.WordValue(offset)	Word lesen	word
.IntegerValue(offset)	Integer lesen	integer
.LongValue(offset)	Long lesen	long
.SingleValue(offset)	Single lesen	single
.StringValue(offset,bytes)	String lesen mit Längenvorgabe	string
.PString(offset)	Pascal-String lesen (Start-Byte ist Länge)	string
.CString(offset)	C-String lesen (Nullterminiert)	string
.Addr	Adresse der Datenstruktur im Flash	word
.SizeOf	Vom Objekt belegte Anzahl Bytes lesen.	byte

- **offset:** Byte-Position innerhalb der Struktur mit Basis Null.
- **bytes:** Anzahl zu lesender Bytes.

DATENABLAGE

Die Direktiven zur Ablage von Daten innerhalb eines Datenobjektes.

- .db - 8 Bit Werte inkl. Zeichenketten
- .dw - 16 Bit Werte
- .dt - 24 Bit Werte
- .dl - 32 Bit Werte

Hinweis
Der Zugriff über die Objektgrenzen hinaus ist möglich und wird nicht geprüft.

Beispiel

```
dim a as byte
dim s as string

a=tabelle1.ByteValue(4) ' Byte lesen von tabelle1+4, Ergebnis: 5
s=tabelle1.CString(12)  ' C-String lesen, Ergebnis: "Hallo"

tabelle1.ByteValue(1)=7                          ' Byte schreiben
tabelle1.CString(0)="Ich bin ein Eeprom-String"  ' String schreiben

a=Eeprom.ByteValue(4)    ' Byte direkt aus Eepromspeicher lesen
Eeprom.ByteValue(4)=a    ' Byte direkt in den Eepromspeicher schreiben

' Datenstruktur im Eeprom definieren
' HINWEIS:
' Die im Code definierten Eeprom-Werte und Zeichenketten werden vom Compiler
' mit entsprechender Ausgabeoption in der Datei *.eep gespeichert.
' Sie müssen ebenfalls auf den Controller hochgeladen werden.
```

```
eeprom tabelle1
  .db 1,2,3,4,5
  .dw &h4af1,&hcc55,12345
  .db "Hallo",0
endeeprom
```

Exception

Eine ***Exception*** bezeichnet in Luna einen Fehler, der durch den vom Programmierer geschriebenen Programmcode ausgelöst wurde. Wird z.Bsp. ein Speicherblock angefordert der größer ist als der noch verfügbare freie Arbeitsspeicher, kann dieser nicht reserviert werden und die Anforderung schlägt fehl. Vergisst der Programmierer nun durch eine Prüfung, ob der angeforderte Speicher wirklich alloziert werden konnte, werden möglicherweise bereits belegte Speicherbereiche beschädigt und das Programm stürzt ab.

Mit der Deklaration von Exceptions kann während der Programmentwicklung und Fehlersuche geprüft werden, ob das Programm bei entsprechend kritischen Bereichen fehlerfrei ausgeführt wird.

Even()/Odd()

Even() ermittelt ob der übergebene Integer-Wert *gerade* ist.
Odd() ermittelt ob der übergebene Integer-Wert *ungerade* ist.

Präprozessor	Die Funktion ist zusätzlich im Präprozessor verfügbar, führt mit ausschließlich Konstanten als Parameter also nicht zur Erzeugung von Maschinencode.

Syntax: *boolean* = ***Even****(Ausdruck)*
Syntax: *boolean* = ***Odd****(Ausdruck)*

Beispiel:

```
dim a as byte
a=7
if Even(a) then
 print "Die Zahl ";str(a);" ist gerade."
else
  print "Die Zahl ";str(a);" ist ungerade."
end if
```

Siehe auch: Fodd, Feven

Fabs()

Fabs() gibt den Absolutwert des übergebenen Fließkommawertes zurück.

Syntax: ***single*** = ***Fabs****(value as single)*

Beispiel:

```
dim a as single
a = -123.456
a = Fabs(a)  ' Ergebnis: 123.456
a = 123.456
a = Fabs(a)  ' Ergebnis: 123.456
```

Facos()

Fasin() berechnet den Arcus-Cosinus vom übergebenen Wert (Radiant).

Syntax: ***single*** = ***Facos****(value as single)*

Beispiel:

```
dim a as single
a=0.5
a = Facos(a)  ' Ergebnis: 1.04719
```

Fadd, Fsub, Fmul, Fdiv

Fließkomma-Arithmetik als Funktionen (technisch kein Unterschied zu Ausdrücken).

Syntax 1: ***Fadd*** *Variable, Konstante*
Syntax 2: *Ergebnis =* ***Fadd****(arg1 as single,arg2 as single)*

Fsub, Fmul und Fdiv analog zu Fadd.

BEISPIEL SYNTAX 1

```
dim a as single

fadd a, 1000  ' Gleichbedeutend zu a = Single(a + 1000)
```

BEISPIEL SYNTAX 2

```
dim a as integer
dim b as single

b=fmul(a,b)  ' Gleichbedeutend zu b = Single(a * b)
```

Fasin()

Fasin() berechnet den Arcus-Sinus vom übergebenen Wert (Radiant).

Syntax: ***single*** = ***Fasin****(value as single)*

Beispiel:

```
dim a as single
a=0.5
a = Fasin(a)  ' Ergebnis: 0.52359
```

Fatan()

Fatan() berechnet den Arcus-Tangens vom übergebenen Wert (Radiant).

Syntax: ***single*** = ***Fatan****(value as single)*

Beispiel:

```
dim a as single
a=0.5
a = Fatan(a)  ' Ergebnis: 0.46364
```

Fcbrt()

Fcbrt() berechnet die Wurzel aus 3 (Kubikwurzel) vom übergebenen Wert.

Syntax: ***single*** = ***Fcbrt****(value as single)*

Beispiel:

```
dim a as single
a = 27
a = Fcbrt(a)  ' Ergebnis: 3
```

Siehe auch: Fsqrt()

Fceil()

Fceil() rundet zur nächsten Ganzzahl. Bei negativen Zahlen in Richtung 0, bei positiven Zahlen weg von 0.

Syntax: ***Ergebnis*** **=** ***Fround****(value as single)*

Beispiel:

```
dim a as single
a=100.5
a = Fround(a)  ' Ergebnis: 101.0
a=-100.5
a = Fround(a)  ' Ergebnis: -100.0
```

Siehe auch: Floor(), Fround(), Fix(), Frac()

Fcosh()

Fcosh() berechnet den hyperbolischen Cosinus vom übergebenen Wert (Radiant).

Syntax: ***single*** = ***Fcosh****(value as single)*

Beispiel:

```
dim a as single
a=0.5
a = Fcosh(a)  ' Ergebnis: 1.12762
```

Fcos()

Fcos() berechnet den Cosinus vom übergebenen Wert (Radiant).

Syntax: ***single*** = ***Fcos****(value as single)*

Beispiel:

```
dim a as single
a=0.5
a = Fcos(a)  ' Ergebnis: 0.87758
```

Fdeg()

Fdeg() rechnet den übergebenen Winkel von Radiant nach Grad um.

Syntax: ***single*** = ***Fdeg****(angleRad as Single)*

Beispiel:

```
dim winkel,winkelDeg as single
winkel=0.5
winkelDeg = Fdeg(winkel)
```

Siehe auch: Frad()

Feven()/Fodd()

Sonderfunktion zum prüfen von Fließkommawerten auf gerade/ungerade.

Feven() prüft auf ***gerade*** und **Fodd()** auf ***ungerade***.

Syntax: *byte* = ***Feven****(value as single)*
Syntax: *byte* = ***Fodd****(value as single)*

RÜCKGABEERGEBNIS

- **0** = Falsch
- **1** = Wahr
- **2** = Keins von Beiden

Das Prüfen auf gerade/ungerade funktioniert nur mit ganzen Zahlen. Durch eine Typkonvertierung auf einen Integerwert kann man den Teil vor dem Komma mit den Funktionen für Integerwerte auf gerade/ungerade prüfen. Dabei bleibt jedoch der Nachkommaanteil unbeachtet.

Die Sonderfunktionen *Fodd()* und *Feven()* beachten hierbei jedoch auch den Nachkommaanteil. Ist der Nachkommaanteil ungleich 0, dann wird dies durch einen zusätzlichen Rückgabewert signalisiert.

Beispiel:

```
dim a as single
a=28.22

select case Fodd(a)
case 0
  print "Die Zahl ";str(a);" ist gerade."
case 1
  print "Die Zahl ";str(a);" ist ungerade."
case 2
  print "Die Zahl ";str(a);" ist weder gerade noch ungerade"
end select

select case Feven(a)
case 0
  print "Die Zahl ";str(a);" ist ungerade."
case 1
  print "Die Zahl ";str(a);" ist gerade."
case 2
  print "Die Zahl ";str(a);" ist weder gerade noch ungerade"
end select
```

Siehe auch: Odd, Even

Fexp()

Fexp() ist die Exponentialfunktion (e-Funktion) mit der Eulerschen Zahl als Basis.

Syntax: ***single*** = ***Fexp****(value as single)*

Beispiel:

```
dim a as single
a = 2
a = Fexp(a)  ' Ergebnis: 7.38905
```

Fix()

Fix() gibt den Vorkommaanteil einer Fließkommazahl als Integerwert zurück.

Syntax: ***int32*** **=** ***Fix****(value as single)*

Beispiel:

```
dim a as single
dim b as long
a = 100.5
b = Fix(a)  ' Ergebnis: 100
```

Siehe auch: Floor(), Fround(), Fceil(), Frac()

Flexp()

Flexp() ist die Umkehrfunktion von Frexp().

Das Rückgabeergebnis ist die Multiplikation der Mantisse ***m*** mit 2, potenziert mit dem Exponent **e**.

Syntax: ***single*** = ***Flexp****(value as single, e as word)*

Beispiel:

```
dim e as word
dim a as single
a = 100
a = Frexp(a,e)  ' Ergebnis: 0.78125 und in e: 7
a = Flexp(a,e)  ' Ergebnis: 100.0
```

Siehe auch: Frexp()

Flog10()

Flog10() berechnet den Logarithmus zur Basis 10 vom übergebenen Wert.

Syntax: ***single*** = ***Flog10****(value as single)*

Beispiel:

```
dim a as single
a = 2
a = Flog10(a)  ' Ergebnis: 0.30103
```

Flog()

Flog() berechnet den natürlichen Logarithmus vom übergebenen Wert.

Syntax: ***single*** = ***Flog****(value as single)*

Beispiel:

```
dim a as single
a = 2
a = Flog(a)  ' Ergebnis: 0.69314
```

Floor()

Floor() rundet auf die nächste Ganzzahl ab, jedoch abhängig vom Vorzeichen.

Syntax: ***Ergebnis*** **=** ***Floor****(value as single)*

Beispiel:

```
dim a as single
a=123.50011
a = Floor(a)  ' Ergebis: 123.0
a=-123.50011
a = Floor(a)  ' Ergebis: -124.0
```

Siehe auch: Ftrunc(), Fround(), Fix(), Frac()

Format()

Format() konvertiert numerische Eingabewerte in einen formatierten Dezimal-String. Der übergebene Wert wird wie bei str() analog zum übergebenen Datentyp zu einer Zeichenkette (string) mit Dezimaldarstellung konvertiert. Die Formatierungsanweisung weist die Funktion an, wie der Wert formatiert werden soll.

Präprozessor	Die Funktion ist zusätzlich im Präprozessor verfügbar, führt mit ausschließlich Konstanten als Parameter also nicht zur Erzeugung von Maschinencode.

SYNTAX

string = **format**(*formatSpec*, *Ausdruck*)

- **formatSpec:** Zeichenkette (Konstante), welche die Formatierungsanweisungen enthält.
- **Ausdruck:** Ausdruck mit numerischem Ergebnis.

Im Folgenden bezeichnet ***"Dezimalstring"*** den in der Funktion konvertierten, numerischen Eingangswert in Dezimaldarstellung.

Formatierungszeichen	
Zeichen	**Beschreibung**
0	Platzhalter für eine Zahl aus dem Dezimalstring
.	Platzhalter für den Dezimalpunkt. **Bei Integerwerten** führt dieser Platzhalter zum Auftrennen der Zahlendarstellung an dieser Position. **Bei Fließkommawerten** wird die Zahl an dieser Position anhand ihrer Kommastelle ausgerichtet. Fehlt der Punkt, wird bei Fließkommazahlen nur die Ganzzahl dargestellt.
+	Platzhalter für das Vorzeichen. Ist die Zahl negativ, wird "**-**" eingesetzt, ist sie positiv, wird "**+**" eingesetzt.
-	Platzhalter für das negative Vorzeichen. Ist die Zahl positiv, wird ein Leerzeichen eingesetzt.

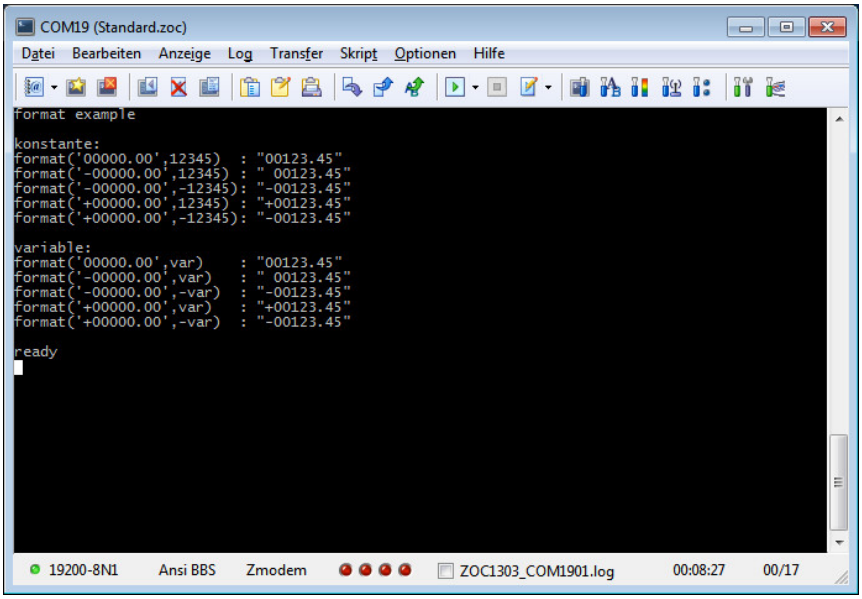
Sonstige Zeichen in ***formatSpec*** werden nicht interpretiert und in den Ergebnisstring übernommen.

Die Formatierung ist **rechtsbündig**, erfolgt also beginnend von der niederwertigsten zur höherwertigsten Dezimalstelle.

BEISPIEL

```
const F_CPU=20000000
avr.device = atmega328p
avr.clock  = F_CPU
avr.stack = 64

uart.baud = 19200
uart.recv.enable
uart.send.enable

dim var as integer

print 12;"format example"
print

print "konstante:"
print "format('00000.00',12345)  : ";34;format("00000.00",12345);34
print "format('-00000.00',12345) : ";34;format("-00000.00",12345);34
print "format('-00000.00',-12345): ";34;format("-00000.00",-12345);34
print "format('+00000.00',12345) : ";34;format("+00000.00",12345);34
print "format('+00000.00',-12345): ";34;format("+00000.00",-12345);34

print
print "variable:"
var=12345
print "format('00000.00',var)    : ";34;format("00000.00",var);34
print "format('-00000.00',var)   : ";34;format("-00000.00",var);34
print "format('-00000.00',-var)  : ";34;format("-00000.00",-var);34
print "format('+00000.00',var)   : ";34;format("+00000.00",var);34
print "format('+00000.00',-var)  : ";34;format("+00000.00",-var);34

print
print "ready"
do
loop
```

AUSGABE


COM19 (Standard.zoc)
Datei Bearbeiten Anzeige Log Transfer Skript Optionen Hilfe
format example
konstante:
format('00000.00',12345) : "00123.45"
format('-00000.00',12345) : " 00123.45"
format('-00000.00',-12345): "-00123.45"
format('+00000.00',12345) : "+00123.45"
format('+00000.00',-12345): "-00123.45"
variable:
format('00000.00',var) : "00123.45"
format('-00000.00',var) : " 00123.45"
format('-00000.00',-var) : "-00123.45"
format('+00000.00',var) : "+00123.45"
format('+00000.00',-var) : "-00123.45"
ready
19200-8N1 Ansi BBS Zmodem ZOC1303_COM1901.log 00:08:27 00/17

For-Next

Schleife mit Schleifenzähler, automatisch inkrementierend oder dekrementierend.Die Schleife wird nur betreten, wenn der Schleifenzähler den Endwert *erreichen kann* undwird verlassen, wenn der Schleifenzähler den Endwert überschreitet.

Die Berechnung und Prüfung des Schleifenzählers erfolgt bei der For-Schleife am Schleifenanfang.

Syntax:

- ***For*** *Variable = Ausdruck* ***to|downto*** *[* ***step*** *Konstante]*[1)]
  - *Programmcode*
- ***Next***

Mit dem Schlüsselwort **to** wird der Schleifenzähler inkrementiert, mit **downto** wird er dekrementiert.
Das optionale Schlüsselwort **step** setzt die Schrittweite eines Schleifendurchgangs. Es wird ein positiver Konstantwert erwartet. Vorgabe ist 1.

Siehe auch: Continue

Beispiel 1

```
dim i as byte
For i=1 to 10
  Print "Hallo"   ' 10 x Ausgabe von "Hallo"
Next
```

Beispiel 2

```
dim i as byte
For i=10 downto 1  ' Schleifenzähler rückwärts
  Print "Hallo"    ' 10 x Ausgabe von "Hallo"
Next
```

Beispiel 3

```
dim i,a,b as byte
a=1
b=10
For i=a to a+b
  Print "Hallo"   ' 11 x Ausgabe von "Hallo"
Next
```

Beispiel 4

```
dim i,a as byte
a=0
For i=1 to a     ' Schleife wird nicht betreten, da Schleifenzähler Endwert nicht erreichen kann
  Print "Hallo"
Next
```

1) **step** implementiert ab 2012.r7.2.build 3875

Fpow()

Fpow() potenziert den übergebenen Wert ***a*** mit dem Exponenten ***b***.

Syntax: ***single*** = ***Fpow***(*a as single, b as single*)

Beispiel:

```
dim a,b as single
a = 1.23
b = 4.32
a = Fpow(a,b)  ' Ergebnis: 2.44652
```

Frad()

Frad() rechnet den übergebenen Winkel in Grad nach Radiant um.

Syntax: ***single*** = ***Frad****(angleDeg as Single)*

Beispiel:

```
dim winkel,winkelRad as single
winkel=40
winkelRad = Frad(winkel)
```

Siehe auch: Fdeg()

Frac()

Frac gibt die Nachkommastellen eines Fließkommawertes (Single) zurück.

Syntax: ***Ergebnis*** = ***Frac****(value as single)*

Beispiel:

```
dim a as Single
a=23.00423
a = Frac(a) ' Ergebnis: 0.00423
a=-23.00423
a = Frac(a) ' Ergebnis: -0.00423
```

Siehe auch: Ftrunc(), Fround(), Fix(), Floor()

Frexp()

Frexp() teilt die Fließkommazahl ***a*** in Mantisse und Exponent auf. Der Exponent wird in der Variable ***e*** gespeichert. Der zweite Parameter ***e*** darf daher kein Ausdruck sein, sondern nur eine einzelne Byte-Variable im Arbeitsspeicher (SRAM) auf die sich dann die Referenz beziehen kann.

Der Rückgabeparameter ist die Mantisse der Fließkommazahl ***a***.

Syntax: ***single*** = ***Frexp****(a as Single, byRef e as word)*

Beispiel:

```
dim e as word
dim a as single
a = 100
a = Frexp(a,e)  ' Ergebnis: 0.78125 und in e: 7
```

Siehe auch: Flexp()

Fround()

Fround() rundet zur nächsten Ganzzahl weg von Richtung Null (kaufmännisches Runden).

Syntax: ***Ergebnis*** **=** ***Fround****(value as single)*

Beispiel:

```
dim a as single
a=100.5
a = Fround(a)  ' Ergebis: 101.0
a=-100.5
a = Fround(a)  ' Ergebis: -101.0
```

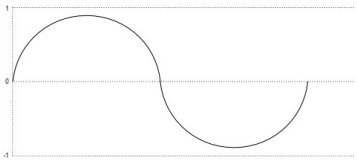
Siehe auch: Floor(), Fceil(), Fix(), Frac()

Fsine()

Fsine() ist eine schnelle 360° Wellenfunktion. Als Parameter erwartet sie einen Winkel in Grad von 0-359. Das Ergebnis ist ein Single-Wert von -1 bis +1 als Äquivalent zum jeweiligen Winkel.

Syntax: ***Ergebnis*** **=** ***Fsine****(Ausdruck)*

Mit Fsine lassen sich auf einfache Weise entsprechende geometrische oder mathematische Sinuswellen erzeugen, da hier keine einzelne Sinus/Cosinus-Berechnung notwendig ist. Die Berechnung des Fsine()-Wertes ist bis zu 20x schneller gegenüber der klassischen Fließkommaberechnung, da sie eine Tabelle verwendet. Jedoch sind hierbei nur ganze Grad-Zahlen möglich.



Grad	Ergebnis
0	0
45	0.7071443
90	1
180	0
270	-1

Beispiel:

```
dim winkel as byte
dim ergebnis as single
winkel=90
ergebnis = Fsine(winkel)
```

Siehe auch: Sine()

Fsinh()

Fsinh() berechnet den hyperbolischen Sinus vom übergebenen Wert (Radiant).

Syntax: ***single*** = ***Fsinh****(value as single)*

Beispiel:

```
dim a as single
a=0.5
a = Fsinh(a)  ' Ergebnis: 0.52109
```

Fsin()

Fsin() berechnet den Sinus vom übergebenen Wert (Radiant).

Syntax: ***single*** = ***Fsin****(Single)*

Beispiel:

```
dim a as single
a=0.5
a = Fsin(a)  ' Ergebnis: 0.47942
```

Fsplit()

Fsplit() teilt die Fließkommazahl ***a*** in den Bereich vor dem Komma und nach dem Komma auf. Es entsteht eine Ganzzahl und eine rationale Zahl. Der Ganzzahlanteil wird in der Variable ***b*** gespeichert. Der zweite Parameter ***b*** darf daher kein Ausdruck sein, sondern nur eine einzelne Variable im Arbeitsspeicher (SRAM) auf die sich dann die Referenz beziehen kann.

Der Rückgabeparameter ist der Nachkommaanteil der Fließkommazahl ***a***. Das Vorzeichen wird für die aufgeteilten Zahlenanteile beibehalten.

Syntax: ***single*** = ***Fsplit****(a as Single, byRef b as single)*

Beispiel:

```
dim a,b as single
a = 100.23
a = Fsplit(a,b)  ' Ergebnis: 0.23 und in b: 100.0
```

Fsqrt()

Fsqrt gibt die Quadratwurzel eines Fließkommawertes (Single) zurück.

Syntax: ***Ergebnis*** = ***Fsqrt****(value as single)*

Beispiel:

```
dim a,b as Single
b=1.4
a=b^2        ' Ergebnis: 1.959
a = Fsqrt(a) ' Ergebnis: 1.4 (1.399)
```

Siehe auch: Sqrt

Fsquare()

Fsquare() berechnet das Quadrat vom übergebenen Wert.

Syntax: ***single*** = ***Fsquare****(value as single)*

Beispiel:

```
dim a as single
a=4.2
a = Fsquare(a)   ' Ergebnis: 17.64
```

Ftanh()

Ftanh() berechnet den hyperbolischen Tangens vom übergebenen Wert (Radiant).

Syntax: ***single*** = ***Ftanh****(value as single)*

Beispiel:

```
dim a as single
a=0.5
a = Ftanh(a)  ' Ergebnis: 0.46211
```

Ftan()

Ftan() berechnet den Tangens vom übergebenen Wert (Radiant).

Syntax: ***single*** = ***Ftan****(Single)*

Beispiel:

```
dim a as single
a=0.5
a = Ftan(a)  ' Ergebnis: 0.5463
```

Ftrunc()

Ftrunc() rundet den übergebenen Wert zur nächsten Ganzzahl, jedoch nicht größer als der Wert vor dem Komma.

Syntax: ***Ergebnis*** **=** ***Ftrunc****(value as single)*

Beispiel:

```
dim a as single
a=123.5
a = Ftrunc(a)  ' Ergebis: 123.0
a=-123.5
a = Ftrunc(a)  ' Ergebis: -123.0
```

Siehe auch: Floor(), Fround(), Fix(), Frac()

Function

Eine Funktion ist ein Unterprogramm wie eine Prozedur, gibt jedoch einen Wert zurück.

Syntax:

- ***Function*** *Name(Var* ***as*** *Datatype[, VarN* ***as*** *Datatype])* ***as*** *Datatype*
  - *[Programmcode]*
  - ***Return*** *Ausdruck*
  - *[Programmcode]*
- ***EndFunc***

- **Name:** Bezeichner des Unterprogramms
- **Var:** Bezeichner der Parameter-Variable

Beispiel für eine Funktionsdeklaration und Aufruf:

```
// Hauptprogramm
dim ergebnis as word
ergebnis=Addition(112,33) // Aufruf der Funktion
function Addition(var1 as word, var2 as byte) as word
  return var1+var2
endfunc
```

Fval()

Zeichenkette mit Fließkomma-Dezimalzahl in Binärwert konvertieren. Das Ergebnis ist 32 Bit Fließkomma vorzeichenbehaftet (single/float). Ignoriert führende Leerzeichen. Das Dezimaltrennzeichen ist der Punkt ".".

Syntax: *single =* ***Fval****(text as string)*

Hinweis: *Es wird ein SRAM-String erwartet, EEPROM ist nicht zulässig, übergabe mit Konstanter Zeichenkette ala* ***fval("12345")*** *wird durch den Präprozessor bearbeitet.*

Beispiel:

```
dim s as string
dim value as single
s = "123.45"
value = Fval(s)
```

GarbageCollection()

Implementiert ab Version:	2015.r1

GarbageCollection() räumt den dynamisch verwalteten Arbeitsspeicher auf. Alle als gelöscht markierten MemoryBlocks (auch Strings und Objekte die auf MemoryBlock basieren) werden entfernt und der Speicher defragmentiert.

Syntax: ***GarbageCollection****()*

Hinweis

Per Vorgabe wird der dynamisch verwaltete Arbeitsspeicher automatisch während der Laufzeit aufgeräumt. Mit dem pragma MemoryBlocksGarbageCollection kann das automatische Aufräumen jedoch deaktiviert werden. Dies kann notwendig sein um in bestimmten Prozessabschnitten eine schnellere Bearbeitung zu erhalten, weil während einer Verarbeitung mit Strings oder MemoryBlocks Ggf. keine automatische Bereinigung gestartet wird. Hier ist es dann notwendig die Bereinigung manuell steuern zu können. **Ist die automatische Bereinigung aktiviert (Vorgabe), dann ist der Aufruf dieser Funktion zwecklos.**

Halt()

Erzeugt eine leere Endlosschleife.

Syntax: ***Halt()***

Beispiel:

```
Halt()
' obiger Ausdruck ist dasselbe wie:
do
loop
```

Hex()

Zahl in Hexadezimale Zeichenkette konvertieren. Konvertiert automatisch analog zum verwendeten Datentyp.

Präprozessor	Die Funktion ist zusätzlich im Präprozessor verfügbar, führt mit ausschließlich Konstanten als Parameter also nicht zur Erzeugung von Maschinencode.

Syntax: *String* = ***Hex****(Ausdruck)*

Beispiel:

```
dim s as string
s = Hex(41394)  ' Ergebnis: "A1B2"
```

HexVal()

Zeichenkette mit Hexadezimalzahl in Integer-Binärwert konvertieren. Die Konvertierungsroutine erkennt Hexadezimalzahlen die mit ***0x*** oder ***&h*** beginnen. Ob Groß- oder Kleinschreibung ist nicht relevant. Führende nicht sichtbare Zeichen (ASCII 0-32) werden ignoriert/übersprungen.

Das Ergebnis ist 32 Bit integer ohne Vorzeichen (long/uint32). Wird das Ergebnis jedoch einem vorzeichenbehafteten 32-Bit-Datentyp wie *longint* oder *int32* zugewiesen, wird der Wert vorzeichenbehaftet.

Syntax: *long* = ***HexVal****(text as string)*

Beispiel:

```
dim result as long
dim s as string
result = HexVal("0xa")
result = HexVal("0xab12")
result = HexVal("&hab12")
result = HexVal("0xabcd1234")
s = "1a2c"
result = HexVal("0x"+s)
```

Idle-EndIdle

Idle-EndIdle ist ein ***globales Event***. Immer wenn bei Aufruf von wartenden Befehlen nichts zu tun ist, wird dieses Event ausgeführt. Ist ***Idle-EndIdle*** im Sourcecode nicht definiert, entfallen in den einzelnen Warteschleifen der Luna-internen Methoden automatisch auch die Aufrufe dieses Events.

Syntax:

- ***Idle***
  - Programmcode
- ***EndIdle***

Methoden die dieses Event aufrufen während sie warten:

- Uart Read-Funktionen
- Uart Write-Funktionen
- InpStr

Das Event kann z.Bsp. in eigenen Schleifen direkt aufgerufen werden, um Rechenzeit abzugeben:

- Avr.Idle

Beispiel:

```
avr.device = attiny2313
avr.clock = 20000000
avr.stack = 12

Uart.Baud = 19200
Uart.Recv.enable
Uart.Send.enable

do
  a = Uart.ReadByte                     ' auf Zeichen warten, dann lesen
  Print "Zeichen empfangen: "+34+a+34    ' Ausgabe z.Bsp. 'Zeichen empfangen: "A"'
loop

' wird aufgerufen während Uart.Read auf ein Zeichen wartet
Idle
  print "nichts zutun"
EndIdle
```

If-ElseIf-Else-EndIf

Bedingte Verzweigung und Programmausführung.

Syntax:

- ***If** Ausdruck1 **Then***
  - Programmcode wenn Ausdruck1 wahr
- ***[ElseIf** Ausdruck2 **Then]***
  - Programmcode wenn Ausdruck1 unwahr und Ausdruck2 wahr
- ***[Else]***
  - Programmcode wenn Ausdruck1 und Ausdruck2 unwahr
- ***EndIf***

Beispiel:

```
dim a,b as byte
if a>b or a=100 then
  [..]
elseif b=1000 then
  [..]
elseif b=2000 and meineFunktion(a+b) then
  [..]
else
  [..]
endif
```

InpStr

!! ACHTUNG !!	Veraltete Syntax! Nur verfügbar bis Version 2013.r3 Als Ersatz siehe Interface-Methode *.InpStr()* der Interfaces Uart und SoftUart.

Zeichenkette von der einer seriellen Schnittstelle lesen, optional mit wiederkehrender Ausgabe eines Prompts. Bei Eingabe eines Zeichens wird Dieses automatisch zurückgesendet (Echo).

Syntax: ***InpStr*** *[Prompt,] ZielString*

- **Prompt (optional):** Zeichenkette (Konstante) die vor der Eingabe ausgegeben wird.
- **ZielString:** String-Variable (Arbeitsspeicher) die das Ergebnis speichert.

Ohne Angabe der Schnittstelle bezieht sich die Funktion auf Uart0.

Die Eingabefunktion wartet bis ein CR (Ascii-13) eintrifft. Nullbytes und Steuerzeichen bis auf Backspace werden ignoriert. Backspace wird interpretiert und der Eingabestring entsprechend verkürzt.

Eigenschaften von InpStr (Global)	
Name	**Beschreibung**
InpStr.echo.Enable	Aktiviert das Eingabeecho (Vorgabe).
InpStr.echo.Disable	Deaktiviert das Eingabeecho.

Mittels der Anweisung ***InpStr.Echo.Disable*** lässt sich das per Vorgabe eingeschaltete Echo der eingegebenen Zeichen abschalten. Die Eigenschaft wirkt Global und wird bei Programmstart definiert (Schalter).

Siehe auch: Idle-EndIdle

Beispiel 1:

```
  dim eingabe as string
  do
    InpStr " Bitte Text eingeben > ",eingabe
    Print "  Sie haben eingegeben: ";34;eingabe;34
  loop
```

Im Terminal zu sehen:

```
Bitte Text eingeben > hallo
  Sie haben eingegeben: "hallo"
Bitte Text eingeben > |
```

Beispiel 2:

```
  dim eingabe as string
  do
    InpStr eingabe
    Print "Sie haben eingegeben: ";34;eingabe;34
  loop
```

Instr()

Zeichenkette in einer anderen Zeichenkette suchen und Positon zurückliefern.Die Suche unterscheidet zwischen Groß- und Kleinschreibung.

Syntax: *byte* = ***Instr****([startPos,]*[1)] *Source as string, Find as string)*

- **startPos:** Startposition der Suche, beginnend mit 1 von links (optional)
- **Source:** Zeichenkette in der gesucht werden soll
- **Find:** Gesuchte Zeichenkette
- **Rückgabe:** Position beginnend mit 1 von links, 0 = nicht gefunden

Beispiel:

```
dim s as string
dim pos as byte
s = "Hallo Welt"
pos = Instr(s,"lo")     ' Ergebnis: 4
pos = Instr(s,"welt")   ' Ergebnis: 0 (nicht gefunden)
```

[1)] Ab Version 2013.R1

Jump

Direkter Sprung zu einem Label oder einer Adresse.

Syntax: ***Jump*** *LabelName/Adresse*

Beispiel 1:

```
   jump test // springe zu "test" und führe die Befehle dort weiter aus
   Print "1" // wird nicht ausgeführt
test:
   Print "2"
```

Beispiel 2:

```
jump &h3c00 ' Bootloader anspringen & Reset (atmega32)
```

Label

Als **Label** werden werden Sprungadressen im Luna- oder Assembler-Code bezeichnet, die mittels entsprechender Anweisungen angesprungen/aufgerufen werden können.

EIGENSCHAFTEN

Labels besitzen die Eigenschaft ***.Addr***, die man zur Ermittlung der Adresse des Labels im Programmspeicher (Flash) benötigt. Mittels **MeinLabel.Addr** lässt sich dieser Wert ermitteln.

SIEHE AUCH

- Jump
- Call
- Void
- Icall

Left()

Linken Teil einer Zeichenkette bis zu einer bestimmten Positon zurückliefern.

Syntax: *string* = ***Left****(Source as string, Position as byte)*
Position: Beginnend mit 1 von links

Beispiel:

```
dim s as string
s = Left("Hallo Welt",5)  ' Ergebnis: "Hallo"
```

Len()

Länge eines Strings ermitteln. Gibt die Anzahl Zeichen zurück, die in einem String aktuell gespeichert sind.

Syntax: *Anzahl* = ***Len****(Ausdruck)*

Beispiel:

```
dim a as byte
dim s as string

s = "Hallo Welt"
a = Len(s)          ' Ergebnis: 10
```

Lower()

Zeichenkette in Kleinbuchstaben wandeln, beachtet sämtliche Umlaute.

Syntax: *string* = ***Lower****(Source as string)*

Beispiel:

```
dim s as string
s = Lower("ABCDEFG")   ' Ergebnis: "abcdefg"
```

Min(), Max()

Min() gibt aus einer Liste von mehreren Werten den kleinsten Wert zurück. Max() den größten Wert. Die Funktion kann eine variable Anzahl an Parametern verarbeiten. Es können bis zu 255 verschiedene Werte angegeben werden. Es werden jedoch mindestens zwei Parameter erwartet.

Der Datentyp des ersten Parameters bestimmt mit welchem Datentyp die Vergleiche durchgeführt werden. Alle weiteren Parameter werden auf Basis dieses Datentyps verglichen und Ggf. vorher auf den Datentyp konvertiert.

Präprozessor	Die Funktion ist zusätzlich im Präprozessor verfügbar, führt mit ausschließlich Konstanten als Parameter also nicht zur Erzeugung von Maschinencode.

Syntax:

- ***Ergebnis*** = ***Min***(*Wert1,Wert2 [,WertN]*)
- ***Ergebnis*** = ***Max***(*Wert1,Wert2 [,WertN]*)

Beispiel:

```
const F_CPU = 8000000
avr.device = attiny2313
avr.clock  = F_CPU
avr.stack = 8

uart.baud = 19200
uart.recv.enable
uart.send.enable

dim i as byte
dim a,b,c as integer

print 12;"min()/max() example"
print

a = -170
b = 8011
c = 230

'min()/max() besitzt eine variable Anzahl Parameter. Minimum sind 2 Parameter.
'Der Datentyp des ersten Parameters bestimmt mit welchem Datentyp gearbeitet wird.
'Alle weiteren Parameter werden dann auf Basis dieses Datentyps verglichen.
print str(min(a,b,c))
print str(max(a,b,c))

print "ready"

do
loop
```

Median16s(), Median16u()

Die Funktion ermittelt aus einer Folge von Eingabewerten den Median. Der Median ist die Zahl, welche an der mittleren Stelle steht, wenn man die Werte nach Größe sortiert. Er kann als Medianfilter zur Verarbeitung von Eingabedaten verwendet werden (Rangordnungsfilter). Beispielsweise beid er Messung von Spannungswerten mittels ADC liefert ein Medianfilter wesentlich bessere Ergebnisse, weil hier keine sehr großen Sprünge auftreten können.

Es sind zwei Varianten für 16-Bit breite Werte definiert. Einmal ohne und einmal mit Beachtung des Vorzeichens.

Syntax: ***word*** = **Median16u**(valuesAddr as word, valueCount as byte)
Syntax: ***integer*** = **Median16s**(valuesAddr as word, valueCount as byte) (Vorzeichenbehaftete Werte)

Als Parameter wird eine Adresse auf einen Speicherbereich im Arbeitsspeicher (z.B. ein Array), sowie die Anzahl der Werte erwartet. Zur Berechnung werden die Eingangswerte temporär auf dem Stack abgelegt. Dies bei der Dimensionierung des Stacks beachten.

BEISPIEL

```
const F_CPU=20000000
avr.device = atmega16
avr.clock  = F_CPU
avr.stack = 64

uart.baud = 19200
uart.recv.enable
uart.send.enable

dim i as byte
dim b(4) as word
dim mv as long

print 12;"median example"
print
wait 1

'Die Eingabewerte
b(0) = 55
b(1) = 99
b(2) = 44
b(3) = 22
b(4) = 11

print
print "Median() = ";str(Median16u( b().Addr, b().Ubound+1 ))
print

print "Median von Hand:"
'Eingabewerte aufsteigend sortieren
b().Sort
'Zur Ansicht ausgeben
for i=0 to b().Ubound
  print "b(";str(i);") = ";str(b(i))
next
print
i = b().Ubound/2 ' mittleres Element ist der Median
mv = b(i)
'Wenn die Anzahl der Elemente gerade ist, gibt es kein mittleres Element, hier
'spricht man dann von Low- und High-Median. Aus beiden wird dann ein Mittelwert
'gebildet.
if even(b().Ubound+1) then
  mv = (mv + b(i+1)) / 2
end if

print "Median() = ";str(mv)
print
print "ready"
do
loop
```

MemCmp()

Die Funktion **MemCmp()** vergleicht zwei Speicherbereiche im Arbeitsspeicher (SRAM) miteinander.

Syntax: *result* = **MemCmp(** *s1Adr as word, s2Adr as word, numBytes as word* **)**

- **s1Adr/s2Adr:** Anfangsadresse der beiden zu vergleichenden Speicherbereiche im Arbeitsspeicher (SRAM).
- **numBytes:** Die Anzahl Bytes die miteinander verglichen werden sollen.
- **result:** Integer-Wert des Vergleichsergebnisses, dabei bedeutet:
  - **= 0:** s1 = s2
  - **> 0:** s1 > s2
  - **< 0:** s1 < s2

BEISPIEL

```
const F_CPU  = 20000000
avr.device = atmega328p
avr.clock  = F_CPU
avr.stack  = 32

uart.baud  = 19200
uart.send.enable
uart.recv.enable

print 12;"MemCmp() Example"

dim i,a as byte
dim m as memoryblock

m.New(60)

m.CString(0)="Hallo"
m.CString(20)="Hallo"

print "MemCmp(): ";str(memCmp(m.Ptr+0,m.Ptr+20,5))  'Ergebnis: 0 (gleich)

print
print "ready"

do
loop
```

MemCpy()

Die Funktion **MemCpy()** kopiert geschwindigkeitsoptimiert Daten zwischen Speicherbereichen im Arbeitsspeicher (SRAM).

Syntax: **MemCpy(** *srcAddr as word, dstAddr as word, numBytes as word* **)**

- **srcAddr:** Anfangsadresse des Quell-Speicherbereichs im Arbeitsspeicher (SRAM).
- **dstAddr:** Anfangsadresse des Ziel-Speicherbereichs im Arbeitsspeicher (SRAM).
- **numBytes:** Die Anzahl Bytes die kopiert werden sollen.

BEISPIEL

```
const F_CPU  = 20000000
avr.device  = atmega328p
avr.clock  = F_CPU
avr.stack  = 32

uart.baud  = 19200
uart.send.enable
uart.recv.enable

print 12;"MemCpy() Example"

dim i,a as byte
dim m1,m2 as memoryblock

m1.New(30)
m2.New(30)

m1.CString(0)="Hallo Welt"

memCpy(m1.Ptr,m2.Ptr,11)  'kopiert Den String inkl. Nullbyte in den 2. Speicherblock

print "m2.CString(0) = ";34;m2.CString(0);34 'Ausgeben aus 2. Speicherblock

print
print "ready"

do
loop
```

MemSort()

Die Funktion **MemSort()** sortiert alle Bytes eines Speicherbereichs im Arbeitsspeicher aufwärts.

Syntax: **MemSort(** *sAdr as word, numBytes as word* **)**

- **sAdr:** Anfangsadresse des Speicherbereichs im Arbeitsspeicher (SRAM).
- **numBytes:** Die Anzahl Bytes die sortiert werden sollen.

BEISPIEL

```
const F_CPU  = 20000000
avr.device = atmega328p
avr.clock  = F_CPU
avr.stack  = 32

uart.baud  = 19200
uart.send.enable
uart.recv.enable

print 12;"MemSort() Example"

dim i,a as byte
dim m as MemoryBlock

m.New(80)
m.PString(0)="Horst rast über völlig verwahrloste Straßen quer durch Bützow."
print "vorher: ";34;m.PString(0);34

MemSort(m.Ptr+1,m.ByteValue(0))

print "nachher: ";34;m.PString(0);34
'Ausgabe: "        .BHSaaabcdeeeeeghhilllnoooqrrrrrrrrssstttttuuvvwwzßöüü"

print
print "ready"

do
loop
```

MemRev()

Die Funktion **MemRev()** ordnet alle Bytes eines Speicherbereichs im Arbeitsspeicher rückwärts an.

Syntax: **MemRev(** *sAdr as word, numBytes as word* **)**

- **sAdr:** Anfangsadresse des Speicherbereichs im Arbeitsspeicher (SRAM).
- **numBytes:** Die Anzahl Bytes die reversiert werden sollen.

BEISPIEL

```
const F_CPU  = 20000000
avr.device  = atmega328p
avr.clock  = F_CPU
avr.stack  = 32

uart.baud  = 19200
uart.send.enable
uart.recv.enable

print 12;"MemRev() Example"

dim i,a as byte
dim m as MemoryBlock

m.New(80)
m.PString(0)="Horst rast über völlig verwahrloste Straßen quer durch Bützow."
print "vorher: ";34;m.PString(0);34

MemRev(m.Ptr+1,m.ByteValue(0))

print "nachher: ";34;m.PString(0);34
'Ausgabe: ".woztüB hcrud reuq neßartS etsolrhawrev gillöv rebü tsar tsroH"

print
print "ready"

do
loop
```

MemoryBlock

Ein MemoryBlock ist ein Speicherbereich im Arbeitsspeicher (SRAM), genauer ein Speicher-Objekt im dynamisch verwalteten Speicherbereich von LunaAVR. Beim Erzeugen eines MemoryBlock wird die gewünschte Anzahl Bytes im Arbeitsspeicher reserviert und kann dann zur Speicherung von beliebigen Daten verwendet werden.

Der Zugriff auf den MemoryBlock bzw. dessen Daten darf ausschließlich nur über die Objekt-Methoden erfolgen, da ein MemoryBlock dynamisch im Speicher verwaltet wird.

Um einen MemoryBlock zu erzeugen, dimensioniert man eine Variable vom Typ MemoryBlock und ruft die "New"-Methode mit der gewünschten anzahl Bytes als Parameter auf. Die maximale Größe eines MemoryBlocks beträgt 65 kb.

ERZEUGEN EINES NEUEN MEMORYBLOCK

ACHTUNG! Ab Version 2013.r6 korrekte objektorientierte Notation:

```
dim myvar as MemoryBlock
myvar = New MemoryBlock(numBytes)
```

FREIGABE EINES MEMORYBLOCK

Die Freigabe eines Speicherblocks erfolgt automatisch bei Zuweisung eines neuen Speicherblocks, sowie beim Verlassen einer Methode, *außer* es handelt sich um eine globale Variable, eine Referenz ("byRef"-Parameter) oder eine statische Variable ("dim x as *static* ...").**Code-Beispiele zur Erläuertung siehe unten.**

METHODEN/EIGENSCHAFTEN

Methoden (aufrufen)	
Name	**Beschreibung**
.New(length)[1)]	Neuen MemoryBlock allozieren und zuweisen, ein vorhandener Block wird freigegeben.
.Free[2)]	Ein vorhandener Block wird freigegeben.

- **length:** zu allozierende Anzahl Bytes

Methoden (nur lesen)		
Name	**Beschreibung**	**Rückgabetyp**
.Addr	Adresse der Variable im Arbeitsspeicher	word
.Ptr	Adresse des zugewiesenen Speicherblocks im Arbeitsspeicher	word
.Size[3)]	Gesamtanzahl Bytes (Größe) des allozierten Speicherblocks.	word
Methoden (lesen und schreiben)		
Name	**Beschreibung**	**Rückgabetyp**
.ByteValue(offset)	Byte lesen/schreiben	byte
.WordValue(offset)	Word lesen/schreiben	word
.IntegerValue(offset)	Integer lesen/schreiben	integer
.Int24Value(offset)[4)]	int24 lesen/schreiben	int24
.Uint24Value(offset)[4)]	uint24 lesen/schreiben	uint24
.LongValue(offset)	Long lesen/schreiben	long
.LongIntValue(offset)[4)]	LongInt lesen/schreiben	longint
.SingleValue(offset)	Single lesen/schreiben	single
.StringValue(offset,bytes)	String lesen/schreiben mit Längenvorgabe	string
.PString(offset)	Pascal-String lesen/schreiben (Start-Byte ist Länge)	string
.CString(offset)	C-String lesen/schreiben (Nullterminiert)	string

- **offset:** Byte-Position innerhalb der Struktur mit Basis Null.
- **bytes:** Anzahl zu lesender Bytes.

HINWEIS

- **Der Zugriff über die Objektgrenzen** hinaus ist ohne deklarierte Exception möglich und wird nicht geprüft (schneller).

Siehe auch: Speicherverwaltung, Sram, Dump

Beispiele

```
dim a as byte
dim s as string
dim m as MemoryBlock

m = New MemoryBlock(100) ' MemoryBlock erzeugen und zuweisen, Ggf. vorhandener Block wird freigeben
if m <> nil then                    ' prüfen ob der Speicherblock alloziert wurde
  m.ByteValue(0)=7                  ' Byte schreiben
  m.CString(1)="Ich bin ein String"  ' String schreiben

  a=m.ByteValue(0) ' Byte lesen, Ergebnis: 7
  s=m.CString(1)   ' C-String lesen, Ergebnis: "Ich bin ein String"

  m=nil           ' Objekt und belegten Speicher freigeben
end if
```

Beispiele zur automatischen Freigabe

```
procedure test(m as MemoryBlock)
  'm ist byVal und wird nach Verlassen zerstört
endproc
```

```
procedure test()
  dim m as MemoryBlock
  'm ist byVal und wird nach Verlassen zerstört
endproc
```

```
procedure test(byRef m as MemoryBlock)
  'm ist byRef und bleibt unangetastet
endproc
```

```
dim m as MemoryBlock

procedure test()
  'm gehört nicht der Methode und bleibt unangetastet
  'Grund:
  'Sichtbarkeit von globalen Variablen in Methoden solange keine eigene
  'Variable gleichen Namens in der Methode erzeugt wurde
endproc
```

```
function test(byRef m as MemoryBlock) as MemoryBlock
  'm ist byRef und bleibt unangetastet
  return m 'es wird eine Instanz von m erzeugt
endfunc
```

```
function test() as MemoryBlock
  dim m as MemoryBlock
  return m 'der Speicherblock wird von "m" dereferenziert (losgelöst) und zurückgegeben
endfunc
```

1) bis Version 2013.r5

2) bis Version 2014.r2.4, obsolet ab Version 2015.r1: Definiertes Freigeben durch Zuweisung von ***nil***, Sonstige automatisch.

3) Ab Version 2013.r4

4) ab Version 2014.r1.8

Mid()

Teil einer Zeichenkette ab einer bestimmten Positon mit bestimmter Länge zurückliefern.

Syntax: *string* = ***Mid****(Source as string, Position as byte[, Length as byte])*

- **Position:** Beginnend mit 1 von links
- **Length (optional):** Anzahl Zeichen die zurückgegeben werden sollen.

Beispiel:

```
dim s as string
s = Mid("Hallo Welt",3,3)  ' Ergebnis: "llo"
```

Arithmetische Operatoren

+ (PLUS)

Dieser Operator wird verwendet um zwei Zahlen zu **summieren** und auch zum **Zusammenfügen** von Zeichenketten.
Ergebnis = Ausdruck1 + Ausdruck2
Folgendes Beispiel addiert verschiedene Zahlen:

```
dim x as integer
x = 1+2+3
```

Folgendes Beispiel fügt verschiedene Zeichenketten zusammen:

```
dim a as string
a = "Ich bin "+" eine "+" Zeichenkette"
a = a + " - 123"
```

- (MINUS)

Dieser Operator wird verwendet um zwei Zahlen zu **subtrahieren**.
Ergebnis = Ausdruck1 - Ausdruck2
Dieses Beispiel subtrahiert verschiedene Zahlen:

```
dim x as integer
x = 10-2-4
```

* (STERN)

Dieser Operator wird verwendet um zwei Zahlen zu **multiplizieren**.
Ergebnis = Ausdruck1 * Ausdruck2
Dieses Beispiel multipliziert verschiedene Zahlen:

```
dim x as integer
x = 1*2*3
```

/ (SCHRÄGSTRICH)

Dieser Operator wird verwendet um zwei Zahlen zu **dividieren**.
Ergebnis = Ausdruck1 / Ausdruck2
Dieses Beispiel dividiert verschiedene Zahlen:

```
dim x as integer
x = 9 / 3
```

^ (CIRCUMFLEX/HUT)

Dieser Operator wird verwendet um zwei Zahlen zu **potenzieren**.
Ergebnis = Ausdruck1 ^ Ausdruck2

ACHTUNG! Diese Funktion wird in Fließkomma-Arithmetik ausgeführt, so wie die Funktion *pow()* in C!Integerargumente werden automatisch konvertiert.

Dieses Beispiel potenziert eine Zahl mit 2:

```
dim x as single
x = 1.234 ^ 2
```

MOD

Dieser Operator führt eine Modulo-Operation zweier Zahlen durch.
Ergebnis = Zahl1 MOD Zahl2
Der MOD-Operator arbeitet mit ganzen Zahlen (Zahlen ohne Nachkommastellen). Fließkommazahlen werden zuerst zu ganzen Zahlen reduziert. Ist eine der beiden Parameter vom Wert 0, dann ist das Ergebnis nicht definiert.

```
dim x as integer
x = 5 mod 2     ' Ergebnis: 1
x = 5 mod 1.999 ' Ergebnis: 0
```

Nop

Keine Operation, beispielsweise zur Verwendung in Warteschleifen. Benötigt 1 Takt.

Syntax: ***Nop***

Beispiel:

```
// 3 Takte warten
nop
nop
nop
```

NthField()

Teilstring einer Zeichenkette lesen, welche zueinander durch einen Separator unterteilt sind.

Syntax: *string* = ***NthField****(source as string, separator as string, index as byte)*

- **source:** Zeichenkette in der gesucht/gelesen werden soll.
- **separator:** Zeichenkette welche die Elemente untereinander abgrenzen.
- **index:** Element das gelesen werden soll, beginnend mit 1 von links.

Siehe auch: CountFields()

BEISPIEL

```
dim a as string
a = "Dies | ist | ein | Beispiel"
print NthField(a,"|",3)   ' Ergebnis: " ein "
```

Part-Endpart

Part/Endpart dient der *optischen* Strukturierung von Sourcecode und besitzt keine Maschinenfunktion. Durch die Ein-/Ausklappfunktion im Editor können damit entsprechende Codebereiche optisch zusammengeschlossen werden.

Syntax:

- **part** *Beschreibung*
  - [..]
- **endpart**

Pascal-String

In Luna werden Strings standardmäßig als Pascal-String gespeichert (ob Variable, Strukturelement oder Konstante). Ein Pascal-String setzt sich aus einem führenden Byte als Längenangabe und den Daten zusammen. Hierdurch lassen sich auch binäre Daten in einem String speichern.

Aufbau des Pascal-Strings
Der String "hallo" wird im Speicher folgend abgelegt:

Speicher ?					
0x05	0x68	0x61	0x6c	0x6c	0x6f
(länge)	h	a	l	l	o

Print

Kurzform von Uartn.Print
Syntax-Beschreibung gilt ebenfalls für SoftUart.Print

Print ist eine der umfangreichsten Ausgabefunktionen. Mit Print können folgend aufgelistete Werte auf den seriellen Schnittstellen des Controllers ausgegeben werden:

- Einzelne Variablen jedes Typs
- Ganze Array-Inhalte
- Zeichenketten
- Konstanten
- Rückgabewerte von Funktionen oder Objekten

Werte separieren

Einzelne Werte werden durch das Semikolon voneinander separiert und dann in der vorhandenen Reihenfolge **von links nach rechts** ausgegeben:

- ***Print*** *Ausdruck1;Ausdruck2*

Befindet sich am Ende des Gesamtausdrucks ein Semikolon, unterdrückt dies die automatische Ausgabe eines Zeilenumbruchs mit ASCII 13 und ASCII 10 (CRLF).

- ***Print*** *Ausdruck1;Ausdruck2;*

Ohne Werte gibt Print nur einen Zeilenumbruch aus:

- ***Print***

Dieser Ausdruck gibt demnach nichts aus:

- ***Print*** *;*

Jeder einzelne Ausdruck wird für sich interpretiert und die Ergebnisse nacheinander ausgegeben:

- ***Print*** *(numVar+1);(numVar1*numVar2);"Hallo"+stringVar*

Hier werden die Ergebnisse der mathematischen Ausdrücke ausgegeben und die zusammengefügte Zeichenkette aus "Hallo" und stringVar (ohne extra Speicherbedarf). Dabei ist zu beachten, dass die Ergebnisse der mathematischen Ausdrücke bei Addition und Multiplikation den nächst größeren Datentyp ergeben. Möchte man also aus einer Berechnung den binären Byte-Wert ausgeben statt eines Word, teilt man dies dem Compiler durch explizite Anweisung über eine Typkonvertierung mit:

- ***Print*** *Byte(numVar+1);numVar1*numVar2;"Hallo"+stringVar*

Werte ausgeben

Einzelne Angaben von Variablen bzw. Array-Elementen (mit Index) gibt ihren Inhalt aus:

- ***Print*** *numVar* - gibt den Inhalt der Variable "numVar" aus
- ***Print*** *numVar(1)* - gibt den Inhalt des Array-Elements "numVar(1)" aus
- ***Print*** *stringVar* - gibt den Inhalt der Variable "stringVar" aus

Einzelne Angaben von Konstanten, gibt ihren Wert abhängig vom Wertebereich aus:

- ***Print*** *22* (gibt den Wert 22 als byte-Wert aus)
- ***Print*** *1234* (gibt den Wert 1234 als word-Wert aus)
- ***Print*** *-1234* (gibt den Wert -1234 als integer-Wert aus)
- ***Print*** *12.34* (gibt den Wert 12.34 als single-Wert aus)
- ***Print*** *1234567* (gibt den Wert 1234567 als long-Wert aus)
- ***Print*** *"text"* (gibt die Zeichenkette "text" aus)

Ausgabe von Rückgabewerten von (Objekt/Interface-) Methoden bzw. Eigenschaften:

- ***Print*** *funktion1(parameter1, parameter2, ..)*
- ***Print*** *PortB*
- ***Print*** *Timer1.Value*
- ***Print*** *str(numVar)*

Beispiel:

```
dim var,c(4),a,b as byte
var=97 ' ASCII-Zeichen "a"
c(0)=asc("H")
c(1)=asc("a")
c(2)=asc("l")
c(3)=asc("l")
c(4)=asc("o")
a=100
b=5
Print "Hallo Welt ";str(12345)   ' Ausgabe: "Hallo Welt 12345"
Print "Hallo Welt ";65           ' Ausgabe: "Hallo Welt A"
Print "Hallo Welt ";Str(65)      ' Ausgabe: "Hallo Welt 65"
Print "Hallo Welt ";c(4)         ' Ausgabe: "Hallo Welt o"
Print "Hallo Welt ";str(a/b)     ' Ausgabe: "Hallo Welt 20"
Print var                        ' Ausgabe: "a"
```

Procedure

Eine Prozedur ist ein Unterprogramm mit lokalem Speicher und mit optionalen Parametern. Sollen Variablen und/oder Werte im Unterprogramm verarbeitet werden, können sie dem Unterprogramm beim Aufruf übergeben werden. Alternativ kann man globale Variablen des Hauptprogramms/der übergeordneten Klasse verwenden.

Siehe auch: Was ist ein Unterprogramm?

Lokale Variablen

Es können benötigte Variablen innerhalb eines Unterprogramms lokal deklariert werden, die dann nur solange existieren bis die Methode verlassen wurde. Beim Aufruf ist zu beachten, dass durch die Übergabe der Parameter zusätzlicher Speicher benötigt wird, bis das Unterprogramm abgearbeitet ist. *Ohne Parameter und lokale Variablen ist das nur die Rücksprungadresse (2 bytes).*

Siehe auch: Dim, Abschnitt Sichtbarkeit von Variablen und Schlüsselwort "static"

Typumwandlung

Ist der übergebene Datentyp ungleich dem des Parameters im Unterprogramm, findet bei numerischen Datentypen eine *automatische Typumwandlung* statt. D.h. übergibt man eine Variable des Typs "word" als Parameter und ist der Parameter als "byte" deklariert, wird der Wert der "word"-Variable vor der Übergabe in einen Byte-Wert gewandelt.

Parameter-Übergabe

Parameter können als Kopie oder Referenz[1)] dem Unterprogramm übergeben werden. Die Parameterübergabe erfolgt über den normalen Programm-Stack (siehe avr.Stack).

Als Übergabeparameter per Kopie (byVal) sind erlaubt:

- Konstanten
- Variablen (außer Strukturen und komplette Arrays)
- Ausdrücke
- Objekteigenschaften und -Methoden mit Rückgabewert
- Methoden mit Rückgabewert

Als Übergabeparameter per Referenz (byRef) sind erlaubt:

- Konstanten-Zeichenketten
- Konstanten-Objekt "data"
- Arbeitsspeicher-Objekte "sram" (Variablen, Arrays, Strukturen, ..)
- Variablen inkl. Strukturen

Das Schlüsselwort **"byVal"** muss nicht zwingend angegeben werden. Vorgabe ist die Übergabe per Kopie.

InitialValue Parameter

InitialValue ist eine Funktionalität, mit der man Parametern optional einen Startwert zuweisen kann:

```
procedure test(a as byte, b as word = 456)
endproc
procedure test1(a as byte = 123, b as word)
endproc
```

Der Aufruf kann dann durch den dann optionalen Parameter verschieden erfolgen, wobei der dann weggelassene Parameter beim Aufruf durch InitialValue automatisch ersetzt wird:

```
test(123,333)
test(100) 'der optionale zweite Parameter erhält den Initialwert
test1(,444) 'der erste Parameter erhält den Initialwert
```

Methoden-Überladen

Methoden-Überladen ist eine Funktionalität, bei der mehrere Methoden gleichen Namens mit verschiedenen Parametertypen und -Anzahl bzw. Rückgabewert definiert werden kann. Der Aufruf wird dann durch die Art- Anzahl und Typ der Parameter bestimmt.

```
test(123)     'methode #1 wird aufgerufen
test(123,456) 'methode #2 wird aufgerufen
a = test(123) 'methode #3 wird aufgerufen

'methode #1
procedure test(a as word)
```

```
endproc
'methode #2
procedure test(a as byte, b as word)
endproc
'methode #3
function test(a as byte) as byte
endfunc
```

Assignment (Zuweisung)

Mit dem Schlüsselwort ***assigns*** beim letzten Parameter einer Methode legt man fest, dass der Methode (Procedure) im Quelltext ein Wert zugewiesen werden kann.

```
test(100) = 200

'[...]
procedure test(a as byte, assigns b as word)
  print "parameter  = ";str(a)
  print "assignment = ";str(b)
endproc
```

Inline

Das optionale Schlüsselwort ***inline*** führt zur Verwendung als Inline-Methode.

Siehe hierzu: http://de.wikipedia.org/wiki/Inline-Ersetzung

Namensraum

In Methoden hat man Zugriff auf globale Variablen der übergeordneten Klasse. Eine im Hauptprogramm deklarierte Variable "a" kann also in einer Prozedure/Funktion verwendet werden. Deklariert man jedoch in einer Prozedure/Funktion eine gleichnamige Variable oder definiert sie als Parameter, **wird Diese bevorzugt behandelt**.

Rekursive/Verschachtelte Aufrufe

Methoden sind in Luna eintrittsinvariant (reentrant), können also verschachtelt oder rekursiv verwendet, oder in ein- oder mehreren Interrupts bzw. im Hauptprogramm aufgerufen werden.

Syntax

Syntax:

- ***[inline] Procedure*** *Name([byVal/byRef] [Var* ***as*** *Datatype[, [byVal/byRef] VarN* ***as*** *Datatype)*
  - *Programmcode*
  - *[Return]*
  - *Programmcode*
- ***EndProc***

- **Name:** Bezeichner des Unterprogramms
- **Var:** Bezeichner der Parameter-Variable

Beispiel 1:

```
' Hauptprogramm
dim wert as integer
dim s as string

ausgabe("Meine Zahl: ",33) ' Aufruf des Unterprogramms
' Weitere Aufrufmöglichkeit
wert = 12345
s = "Andere Zahl: "
call ausgabe(s,wert)

do
loop

' Unterprogramm
procedure ausgabe(text as string, a as byte)
  dim x as integer  ' lokal gültige Variable
  x=a*100
  print text+str(x)
endproc
```

Beispiel 2:

```
struct point
  byte x
  byte y
endstruct

dim p as point

p.x=23
test(p)
print "p.x = ";str(p.x)    ' Ausgabe: p.x = 42

do
loop

procedure test(byRef c as point)
  print "c.x = ";str(c.x)   ' Ausgabe: c.x = 23
  c.x = 42
endproc
```

Beispiel 3:

```
test(text)       ' Ausgabe: c.PString(0) = "hallo"
test("ballo")    ' Ausgabe: c.PString(0) = "ballo"

do
loop

procedure test(byRef c as data)
  print "c.PString(0) = ";34;c.PString(0);34
endproc

data text
  .db 5,"hallo"
enddata
```

1) ab Version 2012.r4

Push/Pop Funktionen

Pushxx***()*** legt den übergebenen Wert auf dem Stack ab.
Popxx***()*** holt den Wert wieder vom Stack.

SYNTAX

- ***Push8/PushByte****(value as uint8)*
- ***Push16****(value as uint16)*
- ***Push24****(value as uint24)*
- ***Push32****(value as uint32)*

- ***uint8*** = ***Pop8/PopByte****()*
- ***uint16*** = ***Pop16****()*
- ***uint24*** = ***Pop24****()*
- ***uint32*** = ***Pop32****()*

WICHTIG!

Die Funktionen wirken exakt wie die Maschinenbefehle Push/Pop. Werden bytes auf dem Stack abgelegt, muss zwingend auch die gleiche anzahl Bytes *innerhalb der gleichen Quelltextebene* wieder vom Stack gelesen werden.

BEISPIEL

```
PushByte(avr.SREG)
cli
MeineFunktion(1,2,3)
avr.SREG = PopByte()
```

Replace(), ReplaceAll()

Hinweis Nur im Präprozessor verfügbar!

Zeichenkette in einem String suchen und ersetzen.

SYNTAX

- ***Replace****(source* ***as string****, find* ***as string****, replacement* ***as string****)*
  Zeichenkette in einem String suchen und *erste* Vorkommende ersetzen.

- ***ReplaceAll****(source* ***as string****, find* ***as string****, replacement* ***as string****)*
  Zeichenkette in einem String suchen und *alle* Vorkommenden ersetzen.

BEISPIEL

```
result=Replace("Hello world! Good morning world!","world","earth") ;returns "Hello earth! Good morning world!"
result=ReplaceAll("Hello world! Good morning world!","world","earth") ;returns "Hello earth! Good morning earth!"
```

Reset

Software-Reset durchführen. Es werden alle Arbeitsspeicher-Variablen zurückgesetzt, die Interrupts abgeschaltet und ein Neustart von Adresse 0x0000 durchgeführt.

Syntax: ***Reset***

Beispiel:

```
Reset
```

Return

Rückkehrbefehl für Unterprogramme (Methoden). In Funktionen mit Parameter.

Syntax 1 (in Funktionen): ***Return*** *Ausdruck*
Syntax 2 (in Prozeduren): ***Return***

Siehe auch: Methoden

Beispiel:

```
Function test(a as byte)
  return a  ' Rückgabe des Funktionswertes
EndFunc
Procedure test1(a as byte)
  if a>0 then
    Return  ' vorzeitiges Verlassen der Methode
  else
    Print str(a)
  end if
EndProc
```

Right()

Rechten Teil einer Zeichenkette bis zu einer bestimmten Positon zurückliefern.

Syntax: *Zeichenkette* = ***Right****(Zeichenkette,Position)*Position: Beginnend mit 1 von rechts

Beispiel:

```
dim s as string
s = Right("Hallo Welt",4)  ' Ergebnis: "Welt"
```

Rinstr()

Zeichenkette *beginnend vom Ende* in einer anderen Zeichenkette suchen und Positon zurückliefern. Die Suche unterscheidet zwischen Groß- und Kleinschreibung.

Syntax: *Position* = ***Rinstr****(Source,Find)*

- **Position:** Beginnend mit 1 von links, 0 = nicht gefunden
- **Source:** Zeichenkette in der gesucht werden soll
- **Find:** Gesuchte Zeichenkette

Beispiel:

```
dim s as string
dim pos as byte
s = "Hallo Welt"
pos = rinstr(s,"l")    ' Ergebnis: 9
pos = rinstr(s,"lo")   ' Ergebnis: 4
```

Seed, Rnd()

Gibt eine Zufallszahl zwischen 0 und 255 zurück.

Der in LunaAVR implementierte Pseudo-Zufallszahlengenerator basiert auf einem linear rückgekoppeltem Schieberegister (LFSR).

Vor dem *ersten* Aufruf von Rnd() muss der Generator mit **Seed** initialisiert werden. Der sporadisch im Programmcode verteilte Aufruf von **Seed** erhöht die Zufallsrate. Es kann bei einem solchen Pseudo-Zufallsgenerator vorkommen, dass nur Nullwerte zurückgegeben werden wenn der LFSR durch bestimmte Umstände in einen Ruhezustand springt. Es ist daher z.T. notwendig die Position und den Seed-Startwert im Quelltext zu verändert bis eine gewünschte, gleichmäßige Verteilung der Zufallszahlen eintritt.

Syntax: ***Seed*** *Ausdruck*
Syntax: *ergebnis =* ***Rnd****()*

Beispiel:

```
[..]
' zuerst Zufallszahlengenerator mit einem beliebigem Startwert laden
Seed 11845
' permanent Zufallszahl ausgeben
do
  print str(Rnd())
  waitms 200
loop
```

Rol

Logisches Bit-Rotieren nach links.

Syntax: ***Rol*** *Variable, Bits*

Beispiel:

```
dim a as integer
Rol a, 4  // Bitweises Rotieren um 4 Bits nach links
```

Ror

Logisches Bit-Rotieren nach rechts.

Syntax: ***Ror*** *Variable, Bits*

Beispiel:

```
dim a as integer
Ror a, 4  // Bitweises Rotieren um 4 Bits nach rechts
```

Seed16, Rnd16()

Gibt eine Zufallszahl zwischen 0 und 65.535 zurück.

Der implementierte Pseudo-Zufallszahlengenerator basiert auf einem linear rückgekoppeltem Schieberegister (LFSR).

Vor dem *ersten* Aufruf von Rnd16() muss der Generator mit **Seed16** initialisiert werden. Der sporadisch im Programmcode verteilte Aufruf von **Seed16** erhöht die Zufallsrate. Es kann bei einem solchen Pseudo-Zufallsgenerator vorkommen, dass nur Nullwerte zurückgegeben werden wenn der LFSR durch bestimmte Umstände in einen Ruhezustand springt. Es ist daher z.T. notwendig die Position und den Seed-Startwert im Quelltext zu verändert bis eine gewünschte, gleichmäßige Verteilung der Zufallszahlen eintritt.

Syntax: ***Seed16*** *Ausdruck*
Syntax: *ergebnis =* ***Rnd16****()*

BEISPIEL

```
[..]
' zuerst Zufallszahlengenerator mit einem beliebigem Startwert laden
Seed16 0x1a74
' permanent Zufallszahl ausgeben
do
  print str(Rnd16())
  waitms 200
loop
```

Seed32, Rnd32()

Gibt eine Zufallszahl zwischen 0 und 4.294.967.295 zurück.

Der implementierte Pseudo-Zufallszahlengenerator basiert auf einem linear rückgekoppeltem Schieberegister (LFSR).

Vor dem *ersten* Aufruf von Rnd32() muss der Generator mit **Seed32** initialisiert werden. Der sporadisch im Programmcode verteilte Aufruf von **Seed32** erhöht die Zufallsrate. Es kann bei einem solchen Pseudo-Zufallsgenerator vorkommen, dass nur Nullwerte zurückgegeben werden wenn der LFSR durch bestimmte Umstände in einen Ruhezustand springt. Es ist daher z.T. notwendig die Position und den Seed-Startwert im Quelltext zu verändert bis eine gewünschte, gleichmäßige Verteilung der Zufallszahlen eintritt.

Syntax: ***Seed32*** *Ausdruck*
Syntax: *ergebnis =* ***Rnd32****()*

BEISPIEL

```
[..]
' zuerst Zufallszahlengenerator mit einem beliebigem Startwert laden
Seed32 0x1a74f9d3
' permanent Zufallszahl ausgeben
do
  print str(Rnd32())
  waitms 200
loop
```

Select-Case-Default-EndSelect

Schnelle bedingte Verzweigung und Programmausführung mit Konstantenvergleich. Ist ein Vergleich zutreffend, wird die Sektion nach ausführen des enthaltenen Programmcodes automatisch Verlassen und springt zum ende des gesamten Bedingungsblocks.

Syntax:

- ***Select Case*** *Ausdruck*
- ***Case*** *Konstante[, KonstanteN]*
  - Programmcode wenn Vergleich wahr
- ***[Case*** *Konstante[, KonstanteN*
  - Programmcode wenn Vergleich wahr
- ***[Default]***
  - Programmcode wenn alle Anderen Vergleiche unwahr.
- ***EndSelect***

Ab 2013.R1 zusätzliche Bereichsprüfung möglich:

```
select case a
case 100 to 200,33,600 to 1000
  ...
end select
```

BEISPIELE

```
  dim a,b as byte
  dim c as word
  dim d as long

' Manchmal kann es sinnvoll sein, einen Datentyp vorzugeben, z.B. Byte()
'  select case Byte(a+b)
  select case (a+b) 'Ausdrücke erlaubt
  case 1
    ' Programmcode
  case "A","B"  ' auch Zeichenketten zulässig (1 Zeichen)
    ' Programmcode
  case 2,3,0b10101010
    ' Programmcode
  case 0x33
    ' Programmcode
  default
    ' Programmcode
  endselect

  'word vergleichen
  select case c
  case 1
    ' Programmcode
  case "AB","xy"  ' auch Zeichenketten zulässig ab build 3815 (2 Zeichen)
    ' Programmcode
  default
    ' Programmcode
  endselect

  'long vergleichen
  select case c
  case 1
    ' Programmcode
  case "ABCD","wxyz"  ' auch Zeichenketten zulässig ab build 3815 (4 Zeichen)
    ' Programmcode
  default
    ' Programmcode
  endselect
```

```
  dim s as string

  select case s
  case "A"
    ' Programmcode
  case "B","C","D"
    ' Programmcode
  case "Super!"
    ' Programmcode
  default
    ' Programmcode
  endselect
```

Sin()

Sin() ist eine schnelle 16 Bit Sinus-Funktion. Als Parameter erwartet sie einen Winkel in Grad multipliziert mit 10. Das Ergebnis ist ein 16 Bit Integer-Wert von -32768 bis +32767 als Äquivalent zum Sinuswert von -1 bis +1. Die Funktion nutzt den Cordic Algorithmus zur Berechnung.

Syntax: ***Ergebnis*** = ***Sin****(Ausdruck)*

Beispiel:

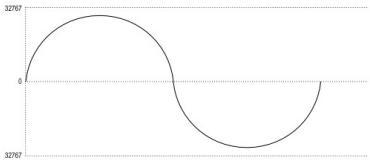
```
dim winkel as byte
dim ergebnis as integer
winkel=23
ergebnis = Sin(winkel*10)
```

Sine()

Sine() ist eine schnelle 16 Bit 360° Wellenfunktion. Als Parameter erwartet sie einen Winkel in Grad von 0-359. Das Ergebnis ist ein 16 Bit Integer-Wert von -32768 bis +32767 als Äquivalent zum jeweiligen Winkel.

Syntax: ***Ergebnis*** = ***Sine****(Ausdruck)*

Mit Sine lassen sich auf einfache Weise entsprechende geometrische oder mathematische Sinuswellen erzeugen, da hier keine einzelne Sinus/Cosinus-Berechnung notwendig ist. Die Berechnung des Sine()-Wertes ist bis zu 10x schneller gegenüber der klassischen Berechnung, da sie eine Tabelle verwendet. Jedoch sind hierbei nur ganze Grad-Zahlen möglich.



Grad	Ergebnis
0	0
45	23171
90	32767
180	0
270	-32768

Beispiel:

```
dim winkel as byte
dim ergebnis as integer
winkel=90
ergebnis = Sine(winkel)
```

Siehe auch: Fsine()

Spc()

Erzeugt eine Zeichenkette mit der angegeben Anzahl Leerzeichen (ASCII 32).

Syntax: *Zeichenkette* = ***Spc****(Anzahl)*

Beispiel:

```
dim s as string
s = Spc(8)  ' Ergebnis: "        "
```

Sram

Sram ist das Objekt "Arbeitsspeicher" der übergeordneten Klasse "Avr".

Eigenschaften (nur lesen)		
Name	**Beschreibung**	**Rückgabe**
.StartAddr oder .Addr	Phys. Startadresse des nutzbaren Arbeitsspeichers.	word
.EndAddr	Phys. Endadresse des nutzbaren Arbeitsspeichers.	word
.Length	Anzahl Bytes des nutzbaren Arbeitsspeichers.	word
.DynAddr	Startadresse des dynamisch nutzbaren Arbeitsspeichers.	word
.DynLength	Anzahl Bytes des dynamischen Arbeitsspeichers.	word
.Space	Anzahl Bytes des unbenutzten dynamischen Arbeitsspeichers.	word

Methoden (aufrufen)	
Name	**Beschreibung**
.ClearAll	Physikalisch nutzbaren Arbeitsspeicher löschen (mit Nullbytes füllen).[1)]
.Dump	Hexdumpausgabe des gesamten physikalisch nutzbaren Arbeitsspeichers auf erster serieller Schnittstelle. [2)]
.DumpBlocks	Hexdumpausgabe aller aktuell allozierten MemoryBlocks. [2)]

Direktzugriff auf Sram

Der Direktzugriff auf den Arbeitsspeicher vergleichbar mit den Objekten Flash- oder EEprom. Durch diese Funktionen kann der Zugriff auf den Arbeitsspeicher "von Hand" vorgenommen werden. D.h. man hat Zugriff vom Speicherbereich der Variablen bis hin zum Stack (der gesamte Arbeitsspeicher). Das Schreiben in den Arbeitsspeicher sollte daher nur dann erfolgen, wenn man MemoryBlock und Stringfunktionen von LunaAvr vermeidet, da diese den Speicher für ihre Dateninhalte selbst verwalten. Besitzt der Controller wenig Arbeitsspeicher, ist es unter Umständen notwendig den Speicher selbst zu verwalten. Wofür diese Funktionen implementiert sind.

Methoden (lesen und schreiben)		
Name	**Beschreibung**	**Rückgabetyp**
.ByteValue(offset)	Byte lesen/schreiben	byte
.WordValue(offset)	Word lesen/schreiben	word
.IntegerValue(offset)	Integer lesen/schreiben	integer
.LongValue(offset)	Long lesen/schreiben	long
.SingleValue(offset)	Single lesen/schreiben	single
.StringValue(offset,bytes)	String lesen/schreiben mit Längenvorgabe	string
.PString(offset)	Pascal-String lesen/schreiben (Start-Byte ist Länge)	string
.CString(offset)	C-String lesen/schreiben (Nullterminiert)	string

Beispiel 1

```
print "Sram.startaddr: "+hex(Sram.startaddr)
print "Sram.endaddr: "+hex(Sram.endaddr)
print "Sram.length: "+str(Sram.length)
print "Sram.dynaddr: "+hex(Sram.dynaddr)
print "Sram.dynlength: "+str(Sram.dynlength)
print "Sram.space: "+str(Sram.space)

Sram.ClearAll  ' komplett löschen (mit Nullbytes füllen)
```

Beispiel 2:

```
avr.device = atmega32
avr.clock = 20000000          ' Quarzfrequenz
avr.stack = 32                ' Bytes Programmstack (Vorgabe: 16)

uart.baud = 19200             ' Baudrate
uart.Recv.enable              ' Senden aktivieren
uart.Send.enable              ' Empfangen aktivieren

dim a,b,i,j as byte
dim offset as word
dim s as string

print 12;"*******************************************"
print "* sram direct access"
```

```
print

a=&h11
b=&h22
s="hallo"

print
print "**** Sram dump from variable space to end of RAM ***"
print "sram.Addr      = 0x";hex(sram.Addr)
print "sram.StartAddr = 0x";hex(sram.StartAddr)
print "sram.EndAddr   = 0x";hex(sram.EndAddr)
print "sram.Length    = ";str(sram.Length)
print

offset = 0
do
  print "0x";hex(word(sram.Addr+offset));": ";
  for i=0 to 23
    when offset > sram.length do exit
    print hex(sram.ByteValue(offset));" ";
    incr offset
  next
  sub offset,24
  print "  ";
  for i=0 to 23
    when offset > sram.length do exit
    a=sram.ByteValue(offset)
    if a>31 then
      print a;
    else
      print ".";
    end if
    incr offset
  next
  print
loop until offset > sram.Length

do
loop
```

1) Inklusive Variablen, Objekte und Stack! Wird am absoluten Anfang jedes Programms automatisch vorgenommen.

2) Debugfunktion, benötigt zusätzlichen Speicherplatz im Flash.

Str()

Zahl in dezimale Zeichenkette konvertieren. Konvertiert automatisch analog zum verwendeten Datentyp.

Präprozessor	Die Funktion ist zusätzlich im Präprozessor verfügbar, führt mit ausschließlich Konstanten als Parameter also nicht zur Erzeugung von Maschinencode.

Syntax: *Zeichenkette* = ***Str****(Ausdruck)*

Beispiel:

```
dim s as string
s = Str(-23.42)  ' Ergebnis: "-23.42"
```

StrFill()

Erzeugt einen String (Zeichenkette) der mit dem Inhalt eines Quellstrings in gewünschter Anzahl aufgefüllt ist.

Präprozessor	Die Funktion ist zusätzlich im Präprozessor verfügbar, führt mit ausschließlich Konstanten als Parameter also nicht zur Erzeugung von Maschinencode.

Syntax: *String* = ***StrFill****(Source,Anzahl)*

- **Source:** Zeichenkette mit der aufgefüllt wird
- **Anzahl:** Anzahl Wiederholungen

Beispiel 1:

```
dim a as string

a = StrFill("Hallo",3)    ' Ergebnis: "HalloHalloHallo"
```

Beispiel 2:

```
dim char as byte
dim a as string

char=65                   ' ASCII-Zeichen "A"
s = StrFill(chr(char),5)  ' Ergebnis: "AAAAA"
```

String (Variable)

Ein String ist eine Kette von beliebigen Zeichen. Jede Art von alphabetischer oder numerischer Informationen kann als Zeichenfolge gespeichert werden. "Heinz Ehrhardt", "13.11.1981", "23.42" sind Beispiele für Strings. In LunaAVR können Strings auch binäre Daten aufnehmen, z.Bsp. Nullbytes.

Im Sourcecode werden Zeichenketten in Anführungsstriche eingebettet. Die maximale Länge eines Strings in LunaAVR beträgt 254 Bytes. Der Standardwert eines Strings ist "" (Leerstring). LunaAVR legt Strings im Pascal-Format, eingebettet in ein MemoryBlock-Objekt ab.

Stringvariablen belegen im Arbeitsspeicher mindestens 2 Bytes (Zeiger auf MemoryBlock-Objekt). Eeprom-Strings jedoch die entsprechend statisch angegebene Anzahl Bytes im Eeprom. Ein als Arbeitsspeicher-Variable deklarierter String ist ein 16-Bit-Pointer auf einen MemoryBlock mit dynamischer Größe.

Eigenschaften (nur lesen)		
Name	**Beschreibung**	**Rückgabetyp**
.Addr	Adresse der Variable im Arbeitsspeicher	word
.Len	Stringlänge lesen (Anzahl Zeichen)	byte
.Ptr	Adresse des zugewiesenen MemoryBlocks im Arbeitsspeicher	word

Methoden (lesen und schreiben)		
Name	**Beschreibung**	**Rückgabetyp**
.ByteValue(offset)	Einzelnes Byte aus gespeichertem Inhalt[1)]	byte

- **offset:** Byte-Position innerhalb des zugewiesenen MemoryBlocks mit Basis Null. Das erste Byte ist das Längenbyte des Strings (offset = 0).

Hinweis
Der Zugriff über die Objektgrenzen hinaus ist ohne deklarierte Exception möglich und wird nicht geprüft (schneller).

Beispiel

```
dim a as bte
dim s as string

s="Hallo"
a=s.ByteValue(0) ' Längenbyte lesen, Ergebnis: 5
s=s.ByteValue(1) ' 1. Zeichen vom Text lesen, Ergebnis: 72 (Ascii-"H")
```

1) Bei leerem String, gibt die Funktion den Wert "0" zurück

Sqrt()

Sqrt gibt die Quadratwurzel eines Integer-Wertes zurück.

Syntax: ***Ergebnis*** = ***Sqrt****(Ausdruck)*

Beispiel:

```
dim a,b as word
b=8
a=b^2        ' Ergebnis: 64
a = Sqrt(a)  ' Ergebnis: 8
```

Siehe auch: Fsqrt

Swap

Swap dient zum vertauschen von Sram-Variablenwerten (auch Array-Elemente). Der Befehl erfüllt die Funktion zum vertauschen der Werte ***zwei verschiedener*** Variablen ***oder*** der ***Low/High-Werte einer*** Variable.

Syntax 1: - vertauscht Low/High-Werte abhängig vom Datentyp

- ***Swap*** *Variable*

Syntax 2: - vertauscht die Werte zweier Variablen

- ***Swap*** *Variable1, Variable2*

Beispiel zu Syntax 1:

```
dim a as byte
dim b as word
dim c as long

a=&hab
b=&haabb
c=&haabbccdd

print "a = 0x";hex(a)  ' Ausgabe: "a = 0xAB"
print "b = 0x";hex(b)  ' Ausgabe: "a = 0xAABB"
print "c = 0x";hex(c)  ' Ausgabe: "a = 0xAABBCCDD"

' Low/High-Werte der Variable tauschen
swap a  ' tauscht Low/High-Nibble
swap b  ' tauscht Low/High-Byte
swap c  ' tauscht Low/High-Word

print "a = 0x";hex(a)  ' Ausgabe: "a = 0xBA"
print "b = 0x";hex(b)  ' Ausgabe: "a = 0xBBAA"
print "c = 0x";hex(c)  ' Ausgabe: "a = 0xCCDDAABB"
```

Beispiel zu Syntax 2:

```
dim a,b,v(4) as byte
dim c,d as word

a=1
b=2
c=3
d=4

' Variablenwerte vertauschen
swap a,b       ' vertauscht die Werte von a und b
swap c,d       ' vertauscht die Werte von c und d
swap v(2),v(4) ' vertauscht die Werte von Element 3 und 5 des Arrays
```

Trim

Entfernt führende und abschließende nicht sichtbare Zeichen einer Zeichenkette, optional Zeichen aus einem Konstanten-String oder Datenobjekt.

Per Vorgabe gefilterte Zeichen: 0,9,32,13,10

Syntax: *Zeichenkette = **Trim**(Ausdruck, [trimChars])*

- **trimChars**: optionaler Konstanten-String oder Datenobjekt (erstes Byte ist Anzahl bytes) mit Zeichen die gefiltert werden sollen.

Beispiel:

```
dim s as string

s = "   hallo  "
s = Trim(s)  ' Ergebnis: "hallo"
```

Upper()

Zeichenkette in Großbuchstaben wandeln, beachtet sämtliche Umlaute.

Syntax: *Zeichenkette = **Upper**(Zeichenkette)*

Beispiel:

```
dim s as string
s = Upper("abcdefg")   ' Ergebnis: "ABCDEFG"
```

Val()

Zeichenkette mit Dezimalzahl in Binärwert konvertieren. Das Ergebnis ist 32 Bit vorzeichenbehaftet (longint/int32). Ignoriert führende Leerzeichen.

Syntax: *int32* = ***Val****(text as string)*

Beispiel:

```
dim a as integer
dim b as longint
a = Val("12345")
b = Val("12345")
```

Void

Void ist ein Schlüsselwort zum Aufruf von Funktionen. Der Rückgabewert der Funktion wird durch dieses Schlüsselwort verworfen.

Void() ist eine Dummymethode. Sie erzeugt keinen Code und dient als Platzhalter z.B. bei Fallunterschiedenen Defines.

Syntax: ***Void*** *Funktionsname(Parameter)*

BEISPIEL

```
dim a as word

a=setChannel(0)
void setChannel(2)

[..]

function setChannel(kanal as byte) as word
  if kanal then
    adc.channel = kanal
  end if
  return adc.Value
endfunc
```

BEISPIEL DUMMYMETHODE

```
const USE_DEBUGPRINT = 0   '1: Debugfunction on, 0: Debugfunction off

#if USE_DEBUGPRINT
  #define DEBUGPRINT(s) as do_print(s)
#else
  #define DEBUGPRINT(s) as avr.void()
#endif

[..]

DEBUGPRINT("show debug print")

[..]

halt()

procedure do_print( s as string )
  print s
  'more code
endproc
```

Siehe auch: Methoden

Wait, Waitms, Waitus

Wartet eine bestimmte Zeit und führt erst nach Ablauf dieser Zeit das Programm fort. Dies ist eine blockliernde Funktion, d.h. die Programmausführung wird für den angebenen Zeitraum komplett blockiert. Die Interrupts sind davon nicht beeinflusst.

Erfordert korrekte Definition des Systemtaktes über Avr.Clock.

Die Genauigkeit aller Wait-Funktionen **mit Konstanten als Parameter** liegt bei rund **1 µs**[1)].

Syntax:

1. ***wait*** *Ausdruck* (Sekunden)
2. ***waitms*** *Ausdruck* (Millisekunden)
3. ***waitus*** *Konstante* (Mikrosekunden)

Die Funktionen **wait** und **waitms** können **ab Version 2012r2** auch mit Variable aufgerufen werden. Die Genauigkeit liegt hier dann bei rund **100 µs**[1)].

Beispiel:

```
dim a a byte
a=200
wait 3       ' 3 Sekunden warten
waitms 100   ' 100 ms warten
waitus 100   ' 100 µs warten
waitms a     ' dynamische Zeit warten, Wartezeit in a
```

1) Wert bezieht sich auf eine Taktrate des Controllers von 20 Mhz. Je niedriger die Taktrate, um so ungenauer ist die Wartezeit.

When-Do

Bedingte Verzweigung und Programmausführung, auf eine Zeile beschränkt.

Syntax:

- ***When*** *Ausdruck* ***Do/Then*** *Ausdruck*

Es kann wahlweise "Do" oder "Then" als zweites Schlüsselwort verwendet werden.

Beispiel1:

```
[..]

when a>b or a=100 do PortB.0 = 1

' obige Syntax ist dasselbe wie:
if a>b or a=100 then
  PortB.0 = 1
endif
```

While-Wend

Schleife mit Startbedingung zum Betreten und Verlassen. Die While-Wend-Schleife wird nur betreten und so lange ausgeführt wie der Ausdruck wahr ist.

Syntax:

- ***While*** *Ausdruck*
  - *Programmcode*
- ***Wend***

Beispiel 1

```
dim i as integer
i=10
While i>0
  Decr i
  Print "Hallo"
Wend
```

Assemblerbefehle

Übersicht über die vom Luna-Assembler (lavra) unterstützten AVR-Assemblerbefehle. Der Luna-Assembler ist im Compiler integriert. Nicht alle Befehle sind auf jedem Controller verfügbar, siehe Datasheet.

Kürzel	Beschreibung
r	Quell-/Zielregister
rh	Obere Quell-/Zielregister (R16-R31)
rd	Doppelregister R24:25(W), R26:27(X), R28:29(Y), R30:31(Z)
rp	Pointerregister X,Y, Z
ry	Pointerregister Y, Z
p	Port
pl	Port an unterer Adresse 0 bis 31
b7	Bitnummer 0 bis 7
k63	Konstante 0 bis 63
k127	Konstante -64 bis +63
k255	Konstante 0-255
k4096	Konstante -2048 bis +2047
k65535	Konstante 0 bis 65535

Gruppe	Funktion	Befehl	Flags	Clk
Leerbefehl	No Operation	NOP		1
Powermanagement	Sleep	SLEEP		1
Überwachung	Watchdog Reset	WDR		1
Register setzen	0	CLR r	Z N V	1
	255	SER rh		1
	Konstante	LDI rh,k255		1
Kopieren	Register >> Register	MOV r,r		1
	SRAM >> Register, direkt	LDS r,k65535		2
	SRAM >> Register	LD r,rp		2
	SRAM >> Register mit INC	LD r,rp+		2
	DEC, SRAM >> Register	LD r,-rp		2
	SRAM, indiziert >> Register	LDD r,ry+k63		2
	Port >> Register	IN r,p		1
	Stack >> Register	POP r		2
	Programmspeicher(Z) >> R0	LPM		3
	Programmspeicher(Z) >> Register	LPM r,Z		3
	Programmspeicher(Z) mit INC >> Register	LPM r,Z+		3
	Programmspeicher(RAMPZ:Z)	ELPM		3
	Register >> SRAM, direkt	STS k65535,r		2
	Register >> SRAM	ST rp,r		2
	Register >> SRAM mit INC	ST rp+,r		2
	DEC, Register >> SRAM	ST -rp,r		2
	Register >> SRAM, indiziert	STD ry+k63,r		2
	Register >> Port	OUT p,r		1
	Register >> Stack	PUSH r		2
Addition	8 Bit, +1	INC r	Z N V	1
	8 Bit	ADD r,r	Z C N V H	1
	8 Bit+Carry	ADC r,r	Z C N V H	1
	16 Bit, Konstante	ADIW rd,k63	Z C N V S	2
	8 Bit, -1	DEC r	Z N V	1
	8 Bit	SUB r,r	Z C N V H	1

Subtraktion	8 Bit, Konstante	SUBI rh,k255	Z C N V H	1
	8 Bit - Carry	SBC r,r	Z C N V H	1
	16 Bit, Konstante	SBIW rd,k63	Z C N V S	2
	8 Bit - Carry, Konstante	SBCI rh,k255	Z C N V H	1
Multiplikation	Integer ohne Vorzeichen	MUL r,r	Z C	2
	Integer mit Vorzeichen	MULS r,r	Z C	2
	Integer mit/ohne Vorzeichen	MULSU r,r	Z C	2
	Fließkomma ohne Vorzeichen	FMUL r,r	Z C	2
	Fließkomma mit Vorzeichen	FMULS r,r	Z C	2
	Fließkomma mit/ohne Vorzeichen	FMUL r,r	Z C	2
Schieben	Logisch, links	LSL r	Z C N V	1
	Logisch, rechts	LSR r	Z C N V	1
	Rotieren, links über Carry	ROL r	Z C N V	1
	Rotieren, rechts über Carry	ROR r	Z C N V	1
	Arithmetisch, rechts	ASR r	Z C N V	1
	Nibbletausch	SWAP r		1
Binär	Und	AND r,r	Z N V	1
	Und, Konstante	ANDI rh,k255	Z N V	1
	Oder	OR r,r	Z N V	1
	Oder, Konstante	ORI rh,k255	Z N V	1
	Exklusiv-Oder	EOR r,r	Z N V	1
	Einer-Komplement	COM r	Z C N V	1
	Zweier-Komplement	NEG r	Z C N V H	1
Bits ändern	Register, Setzen	SBR rh,k255	Z N V	1
	Register, Rücksetzen	CBR rh,255	Z N V	1
	Register, Kopieren nach T-Flag	BST r,b7	T	1
	Register, Kopie von T-Flag	BLD r,b7		1
	Port, Setzen	SBI pl,b7		2
	Port, Rücksetzen	CBI pl,b7		2
Statusbits	Zero-Flag	SEZ	Z	1
	Carry Flag	SEC	C	1
	Negativ Flag	SEN	N	1
	Zweierkompliment Überlauf Flag	SEV	V	1
	Halbübertrag Flag	SEH	H	1
	Signed Flag	SES	S	1
	Transfer Flag	SET	T	1
	Interrupt Enable Flag	SEI	I	1
	Zero-Flag	CLZ	Z	1
	Carry Flag	CLC	C	1
	Negativ Flag	CLN	N	1
	Zweierkompliment Überlauf Flag	CLV	V	1
	Halbübertrag Flag	CLH	H	1
	Signed Flag	CLS	S	1
	Transfer Flag	CLT	T	1
	Interrupt Enable Flag	CLI	I	1
Vergleiche	Register, Register	CP r,r	Z C N V H	1
	Register, Register + Carry	CPC r,r	Z C N V H	1
	Register, Konstante	CPI rh,k255	Z C N V H	1
	Register, ≤0	TST r	Z N V	1
	Sprung relativ	RJMP k4096		2
	Sprung Adresse direkt	JMP k65535		3

Verzweigung	Sprung Adresse indirekt (Z)	IJMP		2
	Sprung Adresse indirekt (EIND:Z)	EIJMP		2
	Unterprogramm, relativ	RCALL k4096		3
	Unterprogramm, Adresse direkt	CALL k65535		4
	Unterprogramm, Adresse indirekt (Z)	ICALL		4
	Unterprogramm, Adresse indirekt (EIND:Z)	EICALL		4
	Return vom Unterprogramm	RET		4
	Return vom Interrupt	RETI	I	4
Bedingte Verzweigung	Statusbit gesetzt	BRBS b7,k127		1/2
	Statusbit rückgesetzt	BRBC b7,k127		1/2
	Springe bei gleich	BREQ k127		1/2
	Springe bei ungleich	BRNE k127		1/2
	Springe bei Überlauf	BRCS k127		1/2
	Springe bei Carry=0	BRCC k127		1/2
	Springe bei gleich oder größer	BRSH k127		1/2
	Springe bei kleiner	BRLO k127		1/2
	Springe bei negativ	BRMI k127		1/2
	Springe bei positiv	BRPL k127		1/2
	Springe bei größer oder gleich (Vorzeichen)	BRGE k127		1/2
	Springe bei kleiner Null (Vorzeichen)	BRLT k127		1/2
	Springe bei Halbübertrag	BRHS k127		1/2
	Springe bei HalfCarry=0	BRHC k127		1/2
	Springe bei gesetztem T-Bit	BRTS k127		1/2
	Springe bei gelöschtem T-Bit	BRTC k127		1/2
	Springe bei Zweierkomplementüberlauf	BRVS k127		1/2
	Springe bei Zweierkomplement-Flag=0	BRVC k127		1/2
	Springe bei Interrupts eingeschaltet	BRIE k127		1/2
	Springe bei Interrupts ausgeschaltet	BRID k127		1/2
Bedingte Sprünge (überspringe Folgebefehl wenn)	Registerbit=0	SBRC r,b7		1/2/3
	Registerbit=1	SBRS r,b7		1/2/3
	Portbit=0	SBIC pl,b7		1/2/3
	Portbit=1	SBIS pl,b7		1/2/3
	Vergleiche, Sprung bei gleich	CPSE r,r		1/2/3

Frequently Asked Questions

- **Was kostet Luna?**
  Die Philosophie der Programmiersprache Luna ist, Anwendern ein innovatives Werkzeug für die Softwareentwicklung **kostenlos** zur Verfügung zu stellen. **Erst bei einer kommerziellen Verwendung werden Ggf. Lizenzgebühren fällig. Siehe hierzu: LunaAVR lizenzieren**

- **Gibt es einen Support?**
  Für Firmen die Luna kommerziell einsetzen, besteht die Möglichkeit unkompliziert einmalig oder dauerhaft einen Support zu buchen bzw. Dienstleistungen zur Entwicklung von z.B. Schnittstellen oder Bibliotheken in Anspruch zu nehmen. Weiterhin steht für alle Anwender das Luna-Forum für die freiwillige Hilfe untereinander zur Verfügung.

- **Kann ich mit Luna selbst entwickelte Programme verkaufen oder kommerziell verwerten?**
  Ja, sie dürfen als Privatperson oder auch als Firma die von ihnen selbst mit Luna entwickelten Programme verkaufen und kommerziell verwerten (z.Bsp. Klassen für die Ansteuerung spezieller Hardware). Die Lizenzbestimmungen legen sie selbst fest.

- **Warum ist die Dokumentation als PDF und nicht als Windows-Hilfedatei verfügbar?**
  LunaAVR ist nicht nur für die Windows-Plattform, sondern auch für Linux und Mac-OS verfügbar. Eine Windows-Hilfedatei macht auf den anderen Plattformen wenig Sinn. Das PDF-Format ist auf allen Systemen lesbar, zudem kann im kompletten Dokument gesucht werden, was z.Bsp. bei einer lokalen HTML-basierten Dokumentation nicht möglich wäre.

- **Ist Luna ein Basic?**
  Nein, sie ist wie Pascal "Basic-ähnlich" und anfängerfreundlich, jedoch kein einfaches Basic. Luna ist eine eigenständige Sprache, welche viele praktische Elemente aus anderen Sprachen beinhaltet.

Kontakt

- **LunaAVR Lizenzierung/Unterstützung**

ENTWICKLER

Entwicklung und Programmierung: Richard Gordon Faika (rgf software)
Mail: avr@myluna.de
Plattform: Windows, Linux, Mac-OS
http://www.rgfsoft.com
http://avr.myluna.de

ÜBERSETZER (ENGLISCH)

Dokumentation/Wiki: by Matthias Walter (listrik) http://www.listrik.de