Luna^{AVR}

Objectoriented programming language
for AVR Microcontrollers.

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

Language Reference

Credits for the translation goes to following authors:

Matthias Walter (main part)
Frank Baumgart
Klaus Illig
Ulrich Rosowski

Luna (AVR)

This is the documentation for the programming language Luna for Atmel® AVR® microcontroller.

Luna is an object-oriented, modern Basic/Pascal-like programming language, the structure and syntax are similar to current, modern development tools. Luna has a well-designed and understandable syntax, which helps the developer to avoid and find errors in the source code (comparable to Modula vs. C). It also offers more complex technical possibilities with Pascal and C/C ++. It is therefore suitable for the efficient and time-saving development of small to large, sophisticated software projects for AVR microcontrollers.

Luna consists of an integrated development environment (IDE), a preprocessor, compiler and assembler. Software can be optionally written in the Luna-IDE or in a normal text editor. The Luna-IDE provides advanced features such as syntax coloring, auto-indentation, source text inspector with auto-completion, automatic source text editing including collapse and collapse functions, and much more. In addition, there is a library editor, controller definitions browser, and many other useful tools.

The generated executable binary code is comparable in execution speed and size to powerful existing high-level languages. There is no restriction on the depth of expressions. It also provides highly-optimized dynamic memory management including dynamic strings, memory blocks, structures, classes, data objects, data structures, as well as direct access to all hardware registers and functions of the controller. The AVR-specific code parts, including the libraries, are accessible in the source code and are written entirely in assembler.

Getting Started

Getting Started

A short overview for beginners with microcontrollers and/or the programming language Luna.

Basics

- What is a microcontroller?

EQUIPMENT

- ISP (In-System-Programmer) AND
- Evaluation board for Atmel microcontrollers OR
- combined Board & ISP, e.g.: **"Starter kit Luna" provided by eHaJo**
  See also: Artikel ISP/Boards (mikrocontroller.net)

Tutorial

- Tutorial based on "Starter kit Luna"

Getting started with Luna

INSTALLATION

1. Download the archive with the actual LunaAVR development environment.
2. Just unpack the archive and save the folder with all files on a place of your choice. The LunaAVR development environment can also be started from USB-stick. There is no need for an installation with changes on your system; therefore an installation-program is not necessary. With unpacking and copying the files to the place of your choice the installation is done. If desired, a shortcut on your desktop can be created easily (e.g. for Windows OS: Select the LunaAVR symbol and drop it on your Windows Desktop while holding the "ALT" key.

FIRST START & SETTING

After starting the program "LunaAVR" adjust the setting for Ide-programmer-uploader.

Other settings are at the moment not required, we are now ready to start programming.

Examples

A Luna program requires a minimum of three definitions:

1. What microcontroller should be used ("device")
2. What clock rate is given by fusebit and hardware (e.g. quartz) ("clock")
3. Stack size ("stack") - What does stack mean?

Learn more: Root class Avr

Example for an empty correct program:

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

halt()
```

Another simple program, an LED connected to PortB.0 (Pin 14) is flashing:
Learn more: #Define, Port-pins of the microcontroller und Wait, Waitms, Waitus

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

#define LED1 as portB.0   'LED1 is defined as label for portB.0
LED1.mode = output,low    'the port is configured as output and set to low

do
  LED1 = 1                'switch LED1 on
  waitms 500              'wait for 500ms
  LED1 = 0                'switch LED1 off
  waitms 500              'wait for 500ms
  'alternatively for the commands above also following commands can be used
  'LED1.toggle            'toggle the actual status of the port pin
  'wait 1                 'wait for 1 second
loop
```

Language Reference

Categories

Basics

- Basics
- Identifiers
- Comments
- expressions
- Syntax
- Operators
- Literals
- Data Types
- Casting

Memory, Data-Structures

- Constants
- Variables
- Pointer
- Structures
- Sets
- Arrays
- Objects

Program-Flow

- Conditions
- Loops
- Interrupts
- Events
- exceptions

Program-Structure

- Methods
- Classes
- Preprocessor

Others

- Interfaces
- Inline-Assembler

BASICS

Luna is an object-oriented programming language for AVR microcontrollers.

Luna supports (with small restrictions)

- Inheritance
- Encapsulation
- Polymorphism
- Method-Overloading
- Operator-Overloading

OpenBook: Objektorientierte Programmierung von Bernhard Lahres, Gregor Rayman

TERMS

Object-oriented programming uses generally known programming terms and additional terms like:

- Variable
- Method
- Object
- Class

GENERAL PROGRAMM STRUCTURE

- Controller definition
- Defines, Declarations
- Configurations, Initializations
- **Main Programm**
- Methods/Interrupts
- User Classes
- Data Objects

IMPORTANT!

- Executable code is encoded in the textual order so that classes and programs must follow in the source code ***after*** the main program.
- The preprocessor determined in a pre-flow automatically the containing subroutines, classes and data objects, so an extra declaration of classes or methods *before the main program is **not** necessary*.

IDENTIFIER

Identifiers are names of Variabls, Objects, Labels, Subroutines or Constants and must meet these requirements:

1. The first letter must be a character
2. Minimum lenght is 1 character
3. It may be made up of characters, numbers and the Undeline ("_".
4. Special characters and umlaut except the underline are not permitted

Examples of legal Identifiers:

- a
- Horst
- MySubRoutine
- hello123
- m1_var

COMMENTS

Comments in Luna starts with the character **'** (also possible: ***//***). The comment ends at the end of the line.Examples

```
' This is a comment
a = 100 'This is also a comment
// Well-known type for comments is also supported
a = 100 // This is also a comment
```

EXPRESSIONS

Expressions are arithmetic or logic constructs, Semantik/Syntax that can be evaluated in a context. Individual arithmetia or logic expressions or combinations thereof and expressions containing a string and their functionality are part of a "Expression".

EXPRESSION LENGTH

The length is not limited. The compiler will optimize complex expressions. Splitting into separate operations will usually not speed up processing.

KEYWORDS

true	Represents the value "true", arithmetically = *<>0* (non-zero). Posted in conditions and boolean expressions.
false	Represents the value "false", arithmetically = *0* (zero). Used in conditions and boolean expressions.
nil	Used to determine if an object is nil (no value).

EXAMPLES OF A EXPRESSION

- **Arithmetic Expressions**
  - *5 * 7*
  - *(13 * (x - a) / b) << 5*
  - *x * 9 = 2 * y*
  - *z = c > 100*
  - *y = Function(x, 23, 42)*
  - etc.
- **Logic Expressions**
  - *a and b*
  - *b or a*
  - *(a or not b) and c*
  - etc.
- **String Expressions**
  - *"Hallo"+**str**(22)*
  - *m.**StringValue**(2,3)+":"*
  - ***right**("00"+hour,2)+":"+**right**("00"+minute,2)+":"+**right**("00"+second,2)*
  - etc.

LUNAAVR SYNTAX

The Syntax is semantics of commands and/or expressions and is similar to existing object-oriented design tools. [1)]

VARIATIONS

Variables

- *Variable* = *Expression*
- *Variable* = *Method(Expression1, Expression2)*
- *Variable* = *Object.Method(Expression)*
- *Variable* = *Object.Object.Property*
- etc.

Methodes

- *Method(Expression)*
- *Call Method(Expression)*
- etc.

Conditions

- *If Expression Then [..] elseIf Expression Then [..] else [..] endIf*
- *Select Variable, Case Constant [..] Caseelse [..] endSelect*
- *While Expression [..] Wend*
- *Do [..] Loop Until Expression*
- etc.

Objects

- *Class.Object.Property = Expression*
- *Object.Property = Expression*
- *Object.Method(Expression)*
- *Object.Objekt.Property = Expression*
- etc.

Declarations/Dimensioning

- *Dim Name[, NameN] as Datatype*
- *Const Name = Constant//*etc.**Object-Based Structures***//data [[bezeichner [..] endData*
- *eeprom Identifier [..] endeeprom*
- etc.

1) Luna does without the curly bracket as a separator.

OPERATORS

GENERAL

- Arithmetic Operators
- Logical and Bit Operators
- Operator Hierarchy

ASSIGNMENT-OPERATORS

- Self-operators arithmetical
- Self-operators logical

LITERALS

In Luna literals are defined for binary values and hexadecimal values. For this types following notation can be used:

Binary: *0b* - like in C and Assembler (standard), also possible is using *&b*

Following example demonstrated an 8 bit binary value:

```
0b10111010
```

Hexadecimal: *0x* - 0x like in C and Assembler (standard), also possible is using *&h*

Following example demonstrated an 16 bit hexadecimal value:

```
0x65b7
```

Note: The standard for the *notation* of binary or hexadecimal values are in big-endian byte order.

DATA TYPES

Luna knows a selection of standard numerical and special data types.

OVERVIEW DATA TYPES

Name	lowest value	highest value	Type	Size
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	Object-Variable	2 Byte
MemoryBlock	nil	MemoryBlock	Object-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
user-defined	?	?	Objekt-Variable	?

See also: SRAM-Variable

USER DEFINED DATA TYPES

The user may define own dta types and structures
See also: Struct-EndStruct

INTRINSIC DATA TYPES

Intrinsic means that the variable value is interpreted as a memory address. Pointer, string variables, and memory blocks are therefore intrinsic variables.

STRING

A string is bound of random characters. Any kind of alphabetical or numerical information can be stored as a string. "Dr. Who", "13.11.1981", "23:42" are examples of strings. In LunaAVR strings can also contains *binary data*, e.g. *zero bytes*.In the source code strings are embedded in quotes. The maximum length of a string in LunaAVR is 254 bytes. The default value is "" (empty string). LunaAVR stores strings in the Pascal format, embedded in a MemoryBlock-Object.

String variables occupy at least 2 bytes in memory (pointer to memoryblock object). However, Eeprom static strings specify the corresponding number of bytes in the eeprom memory. A string-variable dimensioned in the working memory (SRAM) is a 16-bit-pointer to a MemoryBlock of dynamical size.

String constants are stored in the program segment (flash) and occupys the number of bytes of the string-data + one byte at the first for the length (Pascal-String).

MEMORYBLOCK

The data type MemoryBlock is a *direct* Object reference (and also a 16-bit-pointer) to a MemoryBlock-Object. This allows direct access to the reserved memory within the MemoryBlock-Methods and Properties. Example of a variable declaration with different data types in memory:

```
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
myText="Dr. Who"
m = New MemoryBlock(100)
```

POINTER

Pointer are also intrinsic variables like a MemoryBlock (16-Bit-Pointer to a Memory location), but pointers have a special ability: you can assign values like integer variables?and perform calculations with them. Furthermore, the object functions of the MemoryBlocks also applies. Example p.ByteValue (0) = 0x77, etc. is possible.Since the controller has three different memory segments, also three different pointer types have been implemented as a new data type. These are:

- **sptr:** Pointer to location in the working memory (sram segment)
- **dptr:** Pointer to location in the program segment (flash/data segment)
- **eptr:** Pointer to location in the eeprom (eeprom segment)

With Pointer you can access an arbitrary address to the object features. e.g. within a memory block or table in a flash.

1) false = *0*

2) true = *<>0*

TYPE CASTING

Explicit type convertion of a value expression into a specific Data Type. This is often makes sense when a Function requires a specific data type. Some output functions adapt the display format ti the data type passed.

For example the calculation of a bit manipulation will result in the next larger data type if the original data type possibly may not have enough space to accomodate the result (Data types smaller than Long).

Example 1:

```
dim a,b as byte
print hex(a+b)   ' The result will be a Word,
                 ' the Hex-Output function will therefor display a Word value
```

You could force a type conversion to define the resulting data type.

TYPE CONVERSION FUNCTIONS

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

See also:

- Bcdenc()
- BcdDec()
- Ce16()
- Ce32()

Example 2:

```
dim a,b as byte
print hex(byte(a+b))  ' The result will be of type Byte,
                      ' the Hex-Output function will therefor display a Byte value
```

CONSTANTS

Luna supports ordinary numeric constants, strings and Constants-Object. Ordinary constants are any numerical values or strings, that can be used in the source code. Constants can defined only once in a class and are visible in sub-routines.

A special case for constants is the Constants-Object (data structure).Expressions with constants will be calculated into binary code before the program is compiled. Thus the definition of a constant through a mathematical term assigns the result of the calculation to the constant. In mathematical functions within the source code, constants are reduced to their respective range of values. This means the constant ‘100’ will be treated like the data type byte, ‘-100’ like a integer and ’12.34’ like a single.

NOTE

To make pure constant calculations for the preprocessor recognizable, enclose they in brackets if the expression also conatins variables or similar. If the preprocessor can differentiate them this avoids unnecessary binary code.

PREDEFINED CONSTANTS

The following constants are preset with a fixed value in the compiler.

Name	Description	Type
PI	The number of PI (3.1415926)	Single
COMPILER_VERSION_STRING	Full string of the compiler version.	string
COMPILER_MAJOR_NUMBER	The main version of the compiler.	integer
COMPILER_MINOR_NUMBER	The release version of the compiler.	integer
COMPILER_UPDATE_NUMBER	The release-update version of the compiler.	integer
COMPILER_BUILD_NUMBER	The build number of the compiler.	integer

EXAMPLE

Definition und usage of constants:

```
Const Number1 = 100
Const Number2 = 33.891
Const Number3 = (12 + 10) / 7       ' Number3 = 3.14285
Const String1 = "I’m a string"
dim x as single
x=Number1+Number2  ' Assignment of the value 133.891, corresponding to x = 133.891
print String1    ' Output: ‘I’m a string’
```

VARIABLES

Variables in the source code are Identifier named memory cells in RAM (SRAM) or eeprom (eRAM). You can assign values to them and readout their value somewhere else.Detailed description (Wikipedia)

DIMENSIONING

Variables are declared by Dim (RAM) and/or eeDim (eeprom).

METHODS/ATTRIBUTES

Variables have Object features:

features of *numeric* standard variables (boolean, byte, word, integer, long, single, ..)			
Name	**Description**	**Type**	**read/write**
.0-x[1]	Bit n of the variable value	boolean	read+write
.LowNibble	Low-Nibble of the variable value	byte	read+write
.HighNibble	High-Nibble of the variable value	byte	read+write
.Nibble***n***	Nibble ***n*** *(1-8)* of the variable value	byte	read+write
.Byte***n***[2]	Byte ***n*** *(1-4)* of the variable value	byte	read+write
.LowByte[3]	Low-Byte of the variable value	byte	read+write
.HighByte[3]	High-Byte of the variable value	byte	read+write
.LowWord[4]	lower Word of the variable value	word	read+write
.HighWord[4]	higher Word of the variable value	word	read+write
.Reverse[5]	reverse the Byte order, see also Ce16(), Ce32()	word, long	read
.Addr[6]	Address of the variable [7]	word	read
.Ptr[8]	Address of the memory blocks	word	read
.SizeOf	Read the number of Bytes the variable reserves.	byte	read

Methods of Byte and Word arrays	
Name	**Description**
.Sort	Sort array data in ascending order (very fast sorting algorithm)

USER DEFINED DATA TYPES

The elements Struct inherit Methods/Attributes.

STRING AND MEMORYBLOCK

The data type String and MemoryBlock have further Methods and Attributes.

USAGE

Variables must be declared before they can be accessed. Variables store data values of their Data Type.**See also:** Dim, Chapter "Visibility of Variables"

ACCESSING INDIVIDUAL BITS OF A VARIABLE

A ssen above, you can access individual bytes for read and write operations. You may also access a Bit via another variable which has that Bit number:**Example:**

- ***var*** = ***var1.var2***
- ***var1.var2*** = *Expression*

Access via a *constant value* executes faster:**Example:**

- ***var1*** = ***var.3***
- ***var.3*** = *Expression*

EXAMPLE

```
dim a,b as byte
dim c as word
dim s as string
c=&ha1b2       ' Assign value, Hexa decimal (lie in reverse order in memory, Hex always Big-endian)
a=c.LowByte    ' read Low Byte of the variable value, result: &hb2
b=c.HighByte   ' read High-Byte of the variable value, result: &ha1
c.LowByte=b    ' reverse and write
c.HighByte=a
c.0=1          'set Bit 0 in c
a=b.7          'read Bit 7 of b
a=3
c=b.a          ' read Bit 3 of b, Bit number is in a
Print "&h"+hex(c)  ' Display: &hb2a1
s = "Hallo"        ' assign text
a = s.ByteValue(1) ' Read decimal value of the 1. character of the string
```

See also

- DataTypes
- Classes
- Dim
- eeDim
- String
- Swap
- MemoryBlock

[1] All numeric data types

[2] only long, single

[3] only word, integer

[4] only long, longint, single

[5] only long, longint, integer, word

[6] All numeric data types and structures. The start address of Arrays

[7] relevant to the memory block in which the variable lies, RAM or eeprom.

[8] The intrinsic data type string, memoryblock

POINTER (SPTR, DPTR, EPTR)

Pointer are special intrinsic variables. Intrinsic means, that the content of the variable is interpreted as a memory address. String variables and memory blocks are also intrinsic variables.Pointer have a special relevance: Just as with Word variables; you can do calculations with them. In addition they are memory block objects, for example p.ByteValue(0) = 0x77 etc.The controller has three different memory segments, therefore three different pointer types haqve been implemented Data Type. These are:

- **sptr:** Pointer to a RAM address (sram segment)
- **dptr:** Pointer to a Flash address (data segment)
- **eptr:** Pointer to a eeprom address (eeprom segment)[/list]

You may also use a pointer to access any address such as within a memory block or a Flash-Table :**Example in the Data segment (Direct access):**

```
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
```

This is similar to the access via Sram, Flash or eeprom interface, but then you must use absolute addresses instead of pointer (this is not really something new). Pointer offer another speciality: **Super Imposing using Structures.**

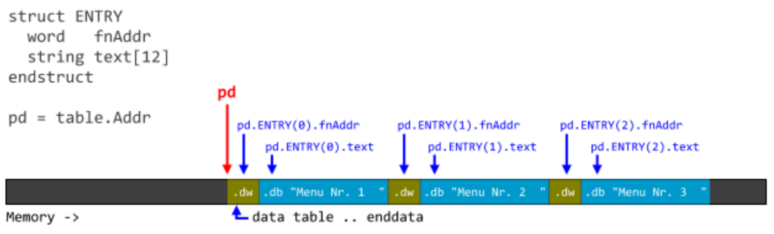
SUPERIMPOSE

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
```

STRUCTURES

STRUCT-ENDSTRUCT

Use Struct/endStruct to define you own structures.

A structure consists of several different Data Types memory object using just one name.

DECLARATION

Syntax (Declaration)

- ***Struct** Definition*
  - *Data Types Definition*
  - *[..]*
- ***endStruct***

VISIBILITY

Declared structures are visible within their class. This means, one in the main program (base class "AVR") declared structure is unknown within another class.

- **Up to version 2014.r2.4:** If you want to swap data using a structure between classes, the structure declaration used for this purpose must be declared in the respective class again. The declarations must be the same, otherwise it causes memory leaks.
- **As of version 2015.r1** structure declarations are also usable with the class name. Is in a class a structure declared, it can also be used in the main program for the dimensioning of structures and vice versa. The above limitation to version 2014.r2.4 is thus canceled.

```
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
```

USAGE

The structure must be declared before it can be accessed. The Declaration uses Struct/endStruct. The *declaration* does *not* use up memory. Memory is reserved after you have actually define a variable structure with Dim in in RAM or eeprom. You can declare a structure within another structure.

Structures themselves are *static*. This means that strings must always have a fixed length (same as when dimensioning in eeprom). Structures reserve as much memory in RAM or eeprom, as the sum of all elements declared.

SUPERIMPOSING

Structure declarations can also be used to simplify data access in Data Objects, or as a memory mask. **See also:** Pointer, chapter "SuperImpose"

STRINGS IN STRUCTURES (DIMENSIONED)

Strings or characters arrays of *dimensioned* structures are automatically allocated as Pascal-String, e.g. the first byte in memory is the length byte. Take this into account when dimensioning a string. Example: A 10 byte long string needs 11 byte in memory.

STRUCTURE-ARRAYS

Structures may be defined as an *Array* , thus making fields with several elemnts per array possible.

Example1:

```
[..]
' Declare Structure
struct point
  byte x
  byte y
endstruct
dim a(3) as point  ' Array with 4 elements of type "point"
a(0).x = 1
a(0).y = 20
a(1).x = 23
a(1).y = 42
[..]
```

Example2:

```
[..]
' Declare Structure
struct buffer
  byte   var(7)
  string text[4](2)  ' String as an array with 3 elements
endstruct
' Declare Structure
struct mydate
  byte    sec
  byte    min
  byte    hour
  byte    month
  word    year
  buffer  buf       ' nested Structure
  string  text[12]  ' Stringspeicher mit max. 11 characters (1 Length + 11 Data)
endstruct

dim d as mydate  ' Declare Structure
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"
[..]
```

SETS

Sets in Luna are special expressions that summarize a collection of constant values ??and facilitate the managing or assigning data collections.

A set stands in the source code within curly braces ***{...}***. If the set is based on a structure declaration, then sets can be embedded within a lot more subsets.

A set basically consists of a so-called ***baseset*** and if necessary other ***subsets***. The first pair of parentheses is called the ***baseset***, in which the values ??and if necessary. other subsets are. Furthermore, it begins with a numeric data type or structure name. ***Subsets*** turn consist only of the parantheses with ***{...}***.

If a set introduced by a structure declaration, the data types specified in the corresponding structure are expected. Structural elements in turn expecting a subset with a matching name structure (nested). See Example 2.

- **Note: Intrinsic data types such as *string, memoryblock, graphics, dptr, eptr, sptr* are *not* allowed.**

EXAMPLES

The following is a baseset in which all values are considered as type "byte". This also applies to the string which in this case is a collection of individual bytes - the letter - represents.

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

The following is a baseset in which all values ??are considered as type "uint24"

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

The following is a baseset in which the values are **different** types. "Which is which" is defined by a structure declaration.

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

mystruct{ { 1, 2 ,3 }, 4, 5 }
```

PRACTICAL APPLICATION

Sets defines the records in the program memory (flash) or the allocation of data to a storage area in memory (sram) or eeprom (eram).

Using data sets/values that would otherwise be present individually are grouped together for easier for people to read the source code. Furthermore, they allow only the assignment of whole records with elements of different types "in one go" to memory areas, eg an array.

SETS IN FLASH MEMORY

EXAMPLE 1

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

data table
  mystruct{ { 1, 2 ,3 }, 4, "hello" } 'Text is defined in the structure declaration, so
  mystruct{ { 5, 6 ,7 }, 8, "bello" } 'defined length of "text" filled with blanks.
enddata
```

The above example is the same as the single statement with db, dw, etc...:

```
data table
  .db 1, 2, 3
  .dw 4
  .db "hallo         "
  .db 5, 6, 7
  .dw 8
  .db "ballo         "
enddata
```

Base sets also allow "normal" data types like *byte, word* and so on, which of course you little to be gained in flash data objects, but it is possible:

```
data table
```

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

The above example is the same as the single statement with db, dw, etc...:

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

EXAMPLE 2

Base set with structure and subset with structure.

```
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
```

ASSIGNMENT TO THE WORK MEMORY/EEPROM

An array of a data type is nothing but concatenated memory cells which can further be addressed individually. They can therefore be considered as a whole as well as a continuous block of memory.

ARRAY

Following is an example to demonstrate an array in one go with fixed values:

```
dim a(7) as byte '8 Elemente
a() = byte{ 11, 22, 33, 44, 55, 66, 77, 88 }
'The array contains for this assignment in the element a (0) is 11, the
'Element a (1) has the value 22, etc.
```

Note: *Watching you have here with the number of items in the set, because the number of the memory location is exceeded by the array be if necessary. subsequent data overwritten. The amount is not directly related to the array. From the perspective of the amount it is just a memory.*
It can also be assigned only a part:

```
dim a(7) as byte '8 Elements
a(3) = byte{ 11, 22 }
'The array contains after this assignment in the element a (3) has the value 11 and in
'Element a (4) the value 22
```

POINTER/MEMORYBLOCK/STRING

Also intrinsic data types such as pointers or memory block containing a pointer to a memory area can be described with data. Here, the destination address is read from the pointer / memory block / string and used as a storage target. That e.g. when a memory block or string is the destination of the data of the memory block which corresponds to the memory block or the string is assigned.

Important: *It must be ensured that the pointer to the memory block or string is a valid memory block or a destination address has been assigned actually!*

```
dim m as MemoryBlock
m = new MemoryBlock(16) 'Creating a memory block with 16 bytes of memory
if m <> nil then          'Check was whether the generated block of memory allocated.
  m = byte{ 11, "Hallo" } 'Writes the byte 11 and the following bytes of text in the memory block.
end if
dim p as sptr            'Creating a Pointer
dim buffer(15) as byte 'A storage area with 16 bytes create (array)

p = buffer(4).Addr       'Assign start address from buffer element (4)
p = byte{ 11, "Hello" }
'the array buffer contains now:
```

```
'  buffer(4) = 11
'  buffer(5) = 72  (ASCII "H")
'  buffer(6) = 101 (ASCII "e")
'  buffer(7) = 108 (ASCII "l")
'  buffer(8) = 108 (ASCII "l")
'  buffer(9) = 111 (ASCII "o")
```

ARRAYS

Arrays are linked DataTypes of the same Type. Numeric Variables, String variables and Structures can be dimensioned as one dimensional arrays. Static means that the number of elements is declared explicitly. You may add a Structure to define sub elements.Add the number of elements in brackets "()" to the Variable to define an array.The number of elements and the first index is *zero based*. The first element of an array has index "0".

NUMBER OF ELEMENTS AS A PROPERTY

You can read the number of elements in the array with ***Ubound***:

```
dim i,a(63) as byte
for i=0 to a().Ubound
  [..]
next
```

INITIALIZATION

Array elements and all other variables in work memory are 0, nil or empty by default. Arrays and variables in eeprom memory are *not* initialized.

Example for dimensioning in memory

```
dim a(99) as byte   ' Byte-Array with 100 elements
dim s(2) as string  ' String-Array with 3 elements
a(22) = 42          ' 23rd element has value 42
Print Str(a(22))    ' Output 22nd element of variable a
s(0) = "Hallo Welt" ' Set text of first element
Print s(0)          ' Output this element
```

Example Dimensioning eeprom

```
eedim a(99) as byte       ' Byte-Array with 100 elements
eedim s[10](2) as string  ' String-Array with 3 elements of 10 Byte length each
                          ' (Strings in eeprom are identical, static and require a length identifier)
a(22) = 42                '  23rd element has value 42
Print Str(a(22))          ' Output 22nd element of variable a
s(0) = "Hallo Welt"       ' Set text of first element
Print s(0)                ' Output this element
```

BUILT-IN OBJECTS

User defined objects are static or dynamic memory objects and can be copied or created for different usages.

- Memory Block - Dynamic memory blocks in RAM
- String - Dynamic variable for character arrays
- Struct-EndStruct - Memory structure in RAM
- Data-EndData - Memory object in Flash
- IncludeData - Memory object in Flash
- Eeprom-EndEeprom - Memory object in Eeprom

CONDITIONS

Use conditions to distinguish beween actions taken.

PREPROCESSOR

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

PROGRAM CODE

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

LOOPS

Three loop-constructs are implemented:

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

These commands are also supported:

- exit/break leave the loop immediately.
- continue continue the loop with the next iteration.

[1)] Initialization value and end condition with automatically an incrementing or decrementing loop counter.

[2)] Optional exit condition at end of loop.

[3)] exit condition check at beginning of loop.

ISR-ENDISR

Isr-endIsr defines a Interrupt-Service routine. The routine can be assigned to one of the Controller's interupts such as a Timer or Serial Port.

Syntax:

- ***Isr*** *Identifier [Switch (see Table)]*
  - Service Routine program code
- ***endIsr***

Switch	Description
save	Compact save all registers (Default)
nosave	No Registers saved, you must handle this in your routine
fastall	Save all Registers (Fast, longer Routine)
fastauto	The registers are saved automatically. Determined automatically by the optimizer by following the program run (fastest version).

EXPLANATION

save/nosave/fastall: Optional parameter control saving of Registers. On an interrupt, the current program execution is interrupted and the Service Routineis executed. Registers will be modified, which must saved on entering and restored on exit. If you have limited memory or a time critical routine; then you can use ***nosave*** to save/restore only Registers that you will modify, or by using ***fastauto*** to let the optimizer determine wich registers must be saved.

SPECIAL FEATURE FASTAUTO

With the option ***fastauto*** the used registers are automatically determined by the optimizer by following the program run.

IMPORTANT NOTES

Interrupt-Service Routines should be kept short without any delay loops, interrupting commands or eeprom access. The Service Routine must be completed before the next Interrupt occurs. Subroutine call are permitted.

Attention when using datatypes who refers to dynamic memory such as *string* **or** *MemoryBlock*. **Here you must ensure that there is no simultaneous access in the interrupts-service and main-program. Manipulation of Strings or MemoryBlocks is time consuming and not recommended!**

EXAMPLE

```
Timer0.isr   = myServiceRoutine  ' Asign Service Routine
Timer0.clock = 1024                 ' Set Prescaler
Timer0.enable                       ' Enable Timer
Avr.Interrupts.enable               ' Enable Globale Interrupts
do
loop

Isr myServiceRoutine
  print "Timer-Interrupt activated"
endIsr
```

EVENT-ENDEVENT

Declares an event. Some Luna objects, such as Twi/Ic2 call a named procedure when certain events occur.**Syntax:**

- ***event*** *eventname*
  - *[Program code]*
- ***endevent***

Example:

```
[..]
event Twierror
  Print "Status = ";str(twi.Status)
endevent
```

EXCEPTION-ENDEXCEPTION

Exceptions let you find errors in the program.Error handling when a Exception is occurred.

Syntax:

- ***exception*** *exceptionName*
  - *Programmcode*
  - *[**Raise**]*
- ***endexception***

Exception Name	Description	Raise
DivisionByZero[1]	Division by Zero occurred in a calculation	yes
OutOfBoundsMemory	MemoryBlock Object Boundry violation	yes
NilObjectexception	Referred object does not exist	yes
OutOfSpaceMemory	Memory full, Object can not be allocated	no
StackOverflow	Stack has overflowed. See also..	no

The exception handler in your code activates the appropriate debug function. Execution speed is reduced in some cases and additional memory in Flash is required.If possible (see Table), **Raise** will continue with code execution. In other cases the program will enter an endless (HALT) loop.
See also: Dump

EXAMPLE

```
' Initialization
[..]
' Main Program
dim a as byte
dim m,m1 as MemoryBlock

m.New(100)           'Allocate a Memory Block with 100 Byte
a=m.ByteValue(100)   'Raises an "out of range" exception
                     '(Access only 0-99, Offsets are zero based)
m1.ByteValue(0)=1    'Object does not exist
m1.New(3000)         'More memory allocated as available (Program will stop)
do
Loop

' Define an Exception Handler (activates the monitoring function)
exception OutOfSpaceMemory
  'Show error message
  Print "*** exception OutOfSpaceMemory ***"
endexception

exception OutOfBoundsMemory
  'Show error message
  Print "*** exception OutOfBoundsMemory ***"
  Raise  'Continue execution
endexception

exception NilObjectexception
  'Show error message
  Print "*** exception NilObjectexception ***"
  Raise  ' Programm fortführen
endexception

exception StackOverflow
  'Show error message
  Print "*** exception StackOverflow ***"
endexception
```

[1] As of Version 2015.r1

METHODS

Methods are subroutines that are executed when called. On completion, the program returns to the code original, and continues exection from there.Luna knows two different methods:

- Procedure-endProc
  (Localized Method and optional parameters)
- Function-endFunc
  (Localized Method with optional parameters and return value)

PROPERTIES

Methods have the property ***.Addr*** to get their address in RAM or Flash.**MyMethod().Addr** returns the **Word-Adresse**.

SEE ALSO

- Call
- Void
- Icall
- Variables in Methods

CLASSES

Luna knows the "AVR" class. You can add new functionality or modularize your program by defining your own classes.

- Basics
- Avr (Basic Class)

USER DEFINED CLASSES

- Class-endClass

PREPROCESSOR (LUNA-SOURCECODE)

The preprocessor is a part of the compiler, which *prepares* the source code for the compile and assemble process. There are two preprocessors in the luna-compiler. One for the Luna-Source and one for the Assembler-Source.

The following parts will be processed by the preprocessor:

- dissolve arithmetic and logic expressions with constants.
- conditional in-/exclusion of sourcecode parts.
- inclusion of external sourcecode and data files
- replacing of defines/macros in the sourcecode.
- processing of inline functions.

PREPROCESSOR FUNCTIONS IN THE LUNA SOURCECODE

DIRECTIVES

- **#Define** - Defines/Aliases.
- **#Undef** - Remove a Define.
- **#pragma, #pragmaSave, #pragmaRestore, #pragmaDefault** - control compiler-internal processes
- **#If-#else-#endif** - conditional compilation.
- **#select-#case-#default-#endselect** - conditional compilation.
- **#macro** - Macros
- **#Include** - includes sourcecode.
- **#IncludeData** - includes binary data into the flasch.
- **#Library** - Import a external library.
- **#cdecl, #odecl, #idecl** - parameter definition for implicit calls.
- **#error** - Put a error message (interrupts the compile process).
- **#warning** - Put a warning message.
- **#message** - Put a info message (without line number and origin).

FUNCTIONS

- **Defined()** - check wether there is a constant or a symbol defined.
- **lo8()** resp. **low()** - Low-Byte of a 16-Bit-value
- **hi8()** resp. **high()** - High-Byte of a 16-Bit-value
- **byte1()** - 1. Byte of a 32-Bit-value
- **byte2()** - 2. Byte of a 32-Bit-value
- **byte3()** - 3. Byte of a 32-Bit-value
- **byte4()** - 4. Byte of a 32-Bit-value
- **byte()** - Typecasting into an 8-Bit-Integer.
- **word()** - Typecasting into a 16-Bit-Integer.
- **integer()** - Typecasting into a signed 16-Bit-Integer.
- **long()** - Typecasting into a 32-Bit-Integer.
- **longint()** - Typecasting into a signed 32-Bit-Integer.
- **single()** - Typecasting into a signed 32-Bit-Float.
- **float()** - Typecasting into a signed 32-Bit-Float.

- **odd()** - check wether the value is odd.
- **even()** - check wether the value is even.
- **chr()** - conversion into binary string (byte).
- **mkb()** - conversion into binary string (byte).
- **mki()** - conversion into binary string (integer).
- **mkw()** - conversion into binary string (word).
- **mkl()** - conversion into binary string (long).
- **mks()** - conversion into binary string (single).
- **strfill()** - fill string with string.
- **hex()** - conversion into hexadecimal notation.
- **str()** - conversion into decimal notation.
- **bin()** - conversion into binary notation.
- **asc()** - conversion of the first character of a string into its numeric equivalent.
- **min()** - arithmetic function.
- **max()** - arithmetic function.

- **format()** - formatted, numeric decimal notation.

BUILT-IN INTERFACES

In Luna, the most common hardware controller functions or interfaces are available as compiler-internal or external interface or module libraries. The interfaces / modules support the developer during configuration, as well as hardware and software accesses to interfaces or protocols. In addition, various software implementations of, for example, Interfaces / protocols.

GENERAL

In general, controller functionalities can be used via the direct access to the configuration and data ports. The configuration or access is based on the port names and configuration bits according to the data sheet of the controller.

The following modules/interfaces are implemented directly in the compiler itself.

See also: external libraries

- Avr - **Basis (The Controller)**
- Eeprom - Eeprom memory/-Objects
- Sram - Work Memory (RAM)
- Uart0-3 - Uart-Interfaces (tiny,mega)
- SoftUart - Software-Uart-Interface

IN ADDITION FOR ATXMEGA

- **Universal-Interfaces** - The universal hardware interface for Atxmega controller.
- Usart - Usart-Interfaces (atxmega)

ASM-ENDASM

Insert assembler sourcecode at the current position.**Syntax:**

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

INLINE-ASSEMBLER

Strings and comments of the inline assembler source code follow the same LunaAVR syntax. Comments in "normal" assembler source code are marked with a **;** and in inline assembler with a **'** or **//**. Strings are always enclosed with double quotes.

GENERAL

Some additional features are implemented in addition to the standard opcodes of the respective controller:

- Standard-Defines of the registers in Luna

- Assembler Commands
- Preprocessor
- Conditions
- Macros

- List of the global constants
- StdLib.interface (Standard-Library)

LABELS IN THE ASSEMBLER SOURCE CODE

The labels in the luna source code are associated with the class they were created. E.g. a label "mylabel" in the main program will be expanded to "classAvrMylabel" and "mylabel" in its own class "myclass" to "classMyclassMylabel". This also applies to labels in the inline assembler source code.This name expansion is important to avoid conflicts between different name spaces and to enable access from within the luna source code to the inline assembler labels.

EXAMPLE 1

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

EXAMPLE 2

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

EXPRESSIONS/COMMANDS

Expressions and operators can be used in conjunction with constants.
The following additional commands are implemented (only in the assembler sourcecode):

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
.importClass	import avr class. sets all defines and constants of the selected avr controller inclusive .device	.importClass atmega32

.importObject	import a library object	.importObject Macro_Delay
.importUsedObjects	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
low()/lo8()	Low-Byte of a value	ldi R16,low(33000)
high()/hi8()	High-Byte of a value	ldi R16,high(33000)
byte*n*()	Byte of a value 1-4	ldi R16,byte1(33000)
single()/float()	Ieee Single-format of a value	.set value = single(33000)
.if .else .endif	Preprocessor case differentiation on assembler level	
defined()	check whether there is a symbol defined.	

EXAMPLE 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
```

Preprocessor (Luna-Sourcecode)

The preprocessor is a part of the compiler, which *prepares* the source code for the compile and assemble process. There are two preprocessors in the luna-compiler. One for the Luna-Source and one for the Assembler-Source.

The following parts will be processed by the preprocessor:

- dissolve arithmetic and logic expressions with constants.
- conditional in-/exclusion of sourcecode parts.
- inclusion of external sourcecode and data files
- replacing of defines/macros in the sourcecode.
- processing of inline functions.

PREPROCESSOR FUNCTIONS IN THE LUNA SOURCECODE

DIRECTIVES

- **#Define** - Defines/Aliases.
- **#Undef** - Remove a Define.
- **#pragma, #pragmaSave, #pragmaRestore, #pragmaDefault** - control compiler-internal processes
- **#If-#else-#endif** - conditional compilation.
- **#select-#case-#default-#endselect** - conditional compilation.
- **#macro** - Macros
- **#Include** - includes sourcecode.
- **#IncludeData** - includes binary data into the flasch.
- **#Library** - Import a external library.
- **#cdecl, #odecl, #idecl** - parameter definition for implicit calls.
- **#error** - Put a error message (interrupts the compile process).
- **#warning** - Put a warning message.
- **#message** - Put a info message (without line number and origin).

FUNCTIONS

- **Defined()** - check wether there is a constant or a symbol defined.
- **lo8()** resp. **low()** - Low-Byte of a 16-Bit-value
- **hi8()** resp. **high()** - High-Byte of a 16-Bit-value
- **byte1()** - 1. Byte of a 32-Bit-value
- **byte2()** - 2. Byte of a 32-Bit-value
- **byte3()** - 3. Byte of a 32-Bit-value
- **byte4()** - 4. Byte of a 32-Bit-value
- **byte()** - Typecasting into an 8-Bit-Integer.
- **word()** - Typecasting into a 16-Bit-Integer.
- **integer()** - Typecasting into a signed 16-Bit-Integer.
- **long()** - Typecasting into a 32-Bit-Integer.
- **longint()** - Typecasting into a signed 32-Bit-Integer.
- **single()** - Typecasting into a signed 32-Bit-Float.
- **float()** - Typecasting into a signed 32-Bit-Float.

- **odd()** - check wether the value is odd.
- **even()** - check wether the value is even.
- **chr()** - conversion into binary string (byte).
- **mkb()** - conversion into binary string (byte).
- **mki()** - conversion into binary string (integer).
- **mkw()** - conversion into binary string (word).
- **mkl()** - conversion into binary string (long).
- **mks()** - conversion into binary string (single).
- **strfill()** - fill string with string.
- **hex()** - conversion into hexadecimal notation.
- **str()** - conversion into decimal notation.
- **bin()** - conversion into binary notation.
- **asc()** - conversion of the first character of a string into its numeric equivalent.
- **min()** - arithmetic function.
- **max()** - arithmetic function.
- **format()** - formatted, numeric decimal notation.

Preprocessor (Assembler)

The preprocessor is a part of the Assembler, which *prepares* the source code for the assemble process.

The following parts will be processed by the preprocessor:

- dissolve arithmetic and logic expressions with constants.
- conditional inclusion of sourcecode parts.
- inclusion of external sourcecode and data files
- replacing of defined text phrases in the sourcecode.
- processing of inline functions.
- processing of macro functions.

PREPROCESSOR FUNCTIONS IN THE ASSEMBLER SOURCECODE

DIRECTIVES

- **.if .elseif .else .endif** - conditional compilation.
- **.select .case .default .endselect** - conditional compilation.
- **.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)]	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

Example for Object-Datablock

```
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 classServiceudpDatagram
```

FUNCTIONS

Preprocessor functions expects only constant parameters.

Functions from the Luna instruction set, depicted in the preprocessor.

- **lo8()** resp. **low()** - Low-Byte of a 16-Bit-value
- **hi8()** resp. **high()** - High-Byte of a 16-Bit-value
- **byte1()** - 1. Byte of a 32-Bit-value
- **byte2()** - 2. Byte of a 32-Bit-value
- **byte3()** - 3. Byte of a 32-Bit-value
- **byte4()** - 4. Byte of a 32-Bit-value
- **byte()** - Typecasting into an 8-Bit-Integer.
- **word()** - Typecasting into a 16-Bit-Integer.
- **integer()** - Typecasting into a signed 16-Bit-Integer.
- **long()** - Typecasting into a 32-Bit-Integer.
- **longint()** - Typecasting into a signed 32-Bit-Integer.
- **int8()** - Typecasting into a signed 8-Bit-Integer.
- **int16()** - Typecasting into a signed 16-Bit-Integer.
- **int24()** - Typecasting into a signed 24-Bit-Integer.
- **int32()** - Typecasting into a signed 32-Bit-Integer.
- **uint8()** - Typecasting into a 8-Bit-Integer.
- **uint16()** - Typecasting into a 16-Bit-Integer.
- **uint24()** - Typecasting into a 24-Bit-Integer.
- **uint32()** - Typecasting into a 32-Bit-Integer.
- **single()** - Typecasting into a signed 32-Bit-Float.
- **float()** - Typecasting into a signed 32-Bit-Float.
- **odd()** - check wether the value is odd.
- **even()** - check wether the value is even.
- **chr()** - conversion into binary string (byte).
- **mkb()** - conversion into binary string (byte).
- **mki()** - conversion into binary string (integer).
- **mkw()** - conversion into binary string (word).
- **mkl()** - conversion into binary string (long).
- **mks()** - conversion into binary string (single).
- **strfill()** - fill string with string.
- **hex()** - conversion into hexadecimal notation.
- **str()** - conversion into decimal notation.
- **bin()** - conversion into binary notation.
- **asc()** - conversion of the first character of a string into its numeric equivalent.
- **min()** - arithmetic function.
- **max()** - arithmetic function.
- **format()** - formatted, numeric decimal notation.

Special functions, implemented only in the preprocessor.

- **MakeIdentifier()** - Ein Symbol aus einem String erstellen.
- **Defined()** - check wether there is a constant or a symbol defined.
- **Replace()** - Replaces the first occurrence of a string with another string.
- **ReplaceAll()** - Replaces all occurrences of a string with another string.

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.

Libraries (external)

LunaAVR uses external libraries to extend the functionality. External libraries are visible in the code and can be edited or re-created with a comfortable, graphical editor. New libraries can be created and / or existing libraries can be edited.

- **Library-Choose**
- **Library-Documentations**
- **Create Libraries**

- Bibliotheken von Anwendern

LIBRARIES (LIST)

Libraries in the folder

- **"/LibraryStd"** - are implemented automatically.
  - **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

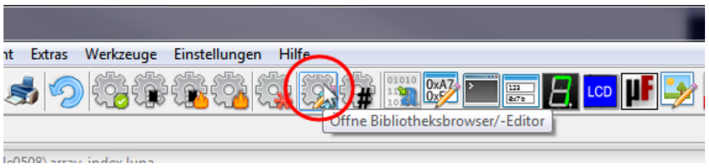
- **"/Library"** - are implemented by command #library.
  - **/Custom** - User created extra libraries.
  - **/Example** - Example libraries for self creating.
  - Adc.interface
  - ClkSys.interface
  - Crc16.interface
  - Crc8.interface
  - DCF77.interface
  - Encodings.module
  - FFT.interface
  - 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

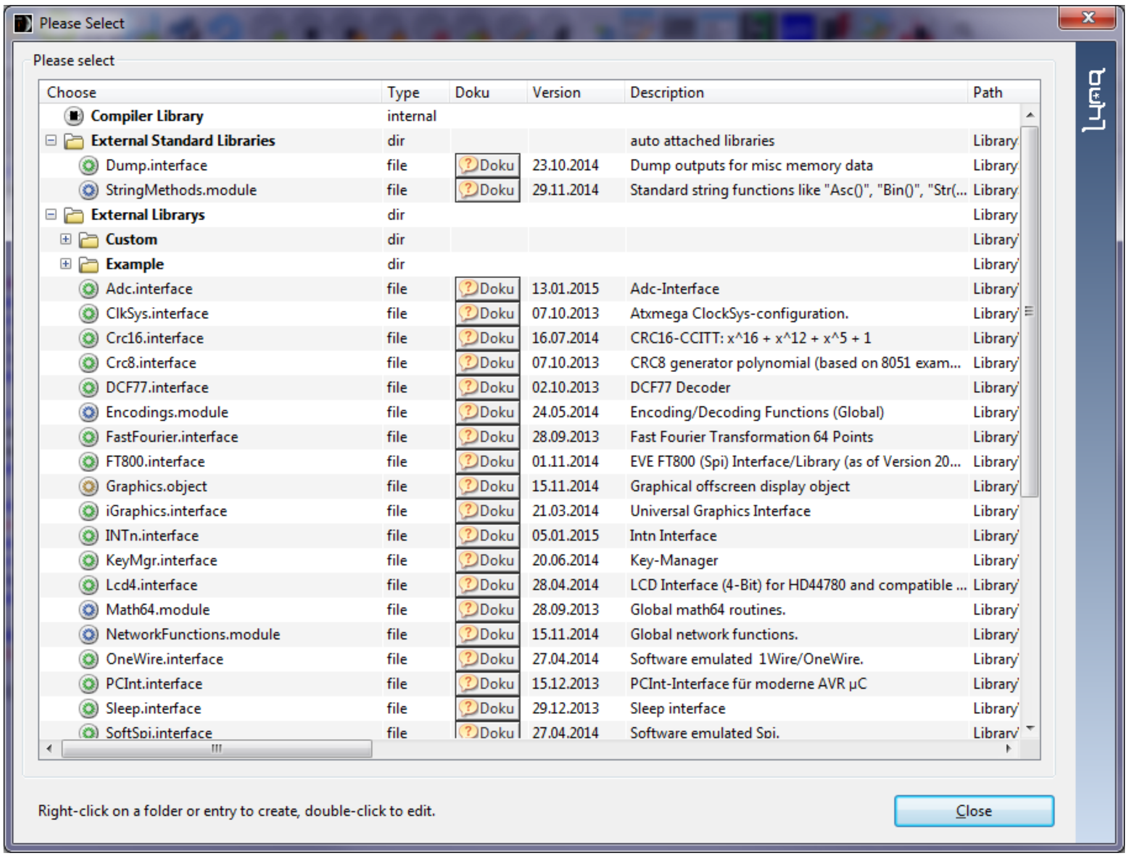
LIBRARIES (CHOOSE)

See also: External Libraries

An overview of the existing, external libraries can be obtained by clicking on the appropriate button or by selecting the menu item *Library Browser/Editor* menu *Tools*.



Subsequently, the library selection opens:



Please Select

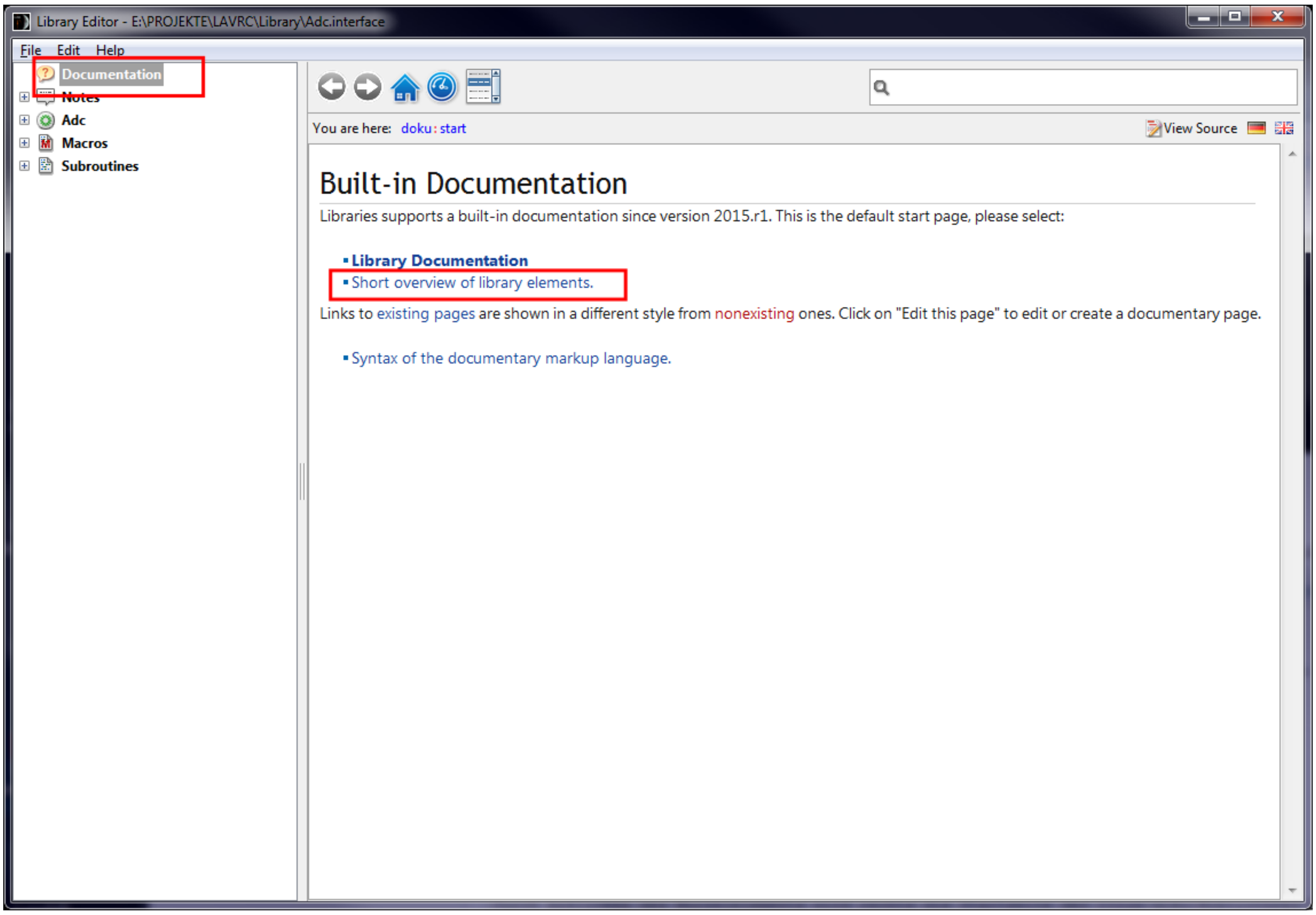
Please select

Choose	Type	Doku	Version	Description	Path
Compiler Library	internal				
External Standard Libraries	dir			auto attached libraries	Library
Dump.interface	file	Doku	23.10.2014	Dump outputs for misc memory data	Library
StringMethods.module	file	Doku	29.11.2014	Standard string functions like "Asc()", "Bin()", "Str(...	Library
External Librarys	dir				Library
Custom	dir				Library
Example	dir				Library
Adc.interface	file	Doku	13.01.2015	Adc-Interface	Library
ClkSys.interface	file	Doku	07.10.2013	Atxmega ClockSys-configuration.	Library
Crc16.interface	file	Doku	16.07.2014	CRC16-CCITT: x^16 + x^12 + x^5 + 1	Library
Crc8.interface	file	Doku	07.10.2013	CRC8 generator polynomial (based on 8051 exam...	Library
DCF77.interface	file	Doku	02.10.2013	DCF77 Decoder	Library
Encodings.module	file	Doku	24.05.2014	Encoding/Decoding Functions (Global)	Library
FastFourier.interface	file	Doku	28.09.2013	Fast Fourier Transformation 64 Points	Library
FT800.interface	file	Doku	01.11.2014	EVE FT800 (Spi) Interface/Library (as of Version 20...	Library
Graphics.object	file	Doku	15.11.2014	Graphical offscreen display object	Library
iGraphics.interface	file	Doku	21.03.2014	Universal Graphics Interface	Library
INTn.interface	file	Doku	05.01.2015	Intn Interface	Library
KeyMgr.interface	file	Doku	20.06.2014	Key-Manager	Library
Lcd4.interface	file	Doku	28.04.2014	LCD Interface (4-Bit) for HD44780 and compatible ...	Library
Math64.module	file	Doku	28.09.2013	Global math64 routines.	Library
NetworkFunctions.module	file	Doku	15.11.2014	Global network functions.	Library
OneWire.interface	file	Doku	27.04.2014	Software emulated 1Wire/OneWire.	Library
PCInt.interface	file	Doku	15.12.2013	PCInt-Interface für moderne AVR µC	Library
Sleep.interface	file	Doku	29.12.2013	Sleep interface	Library
SoftSpi.interface	file	Doku	27.04.2014	Software emulated Spi.	Library

Right-click on a folder or entry to create, double-click to edit.

Close

By double clicking the name you get into the library editor. By clicking on the root element, the start page of the library documentary is shown. The start page also contains a link to the auto-generated short description.


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.

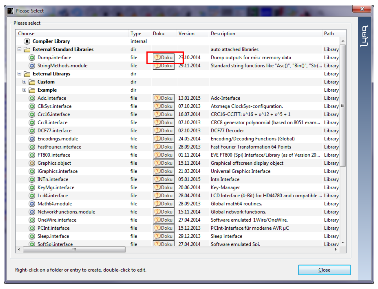
Links to existing pages are shown in a different style from nonexisting ones. Click on "Edit this page" to edit or create a documentary page.

- Syntax of the documentary markup language.

LIBRARY-DOCUMENTARY

The library documentary is stored in each library itself.

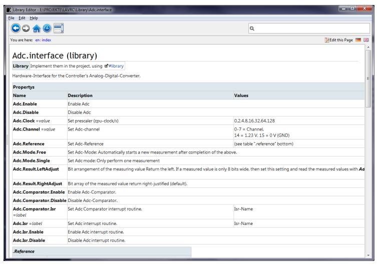
To show the documentary of a library click the button "doku" from the library selection. The library editor will open in the documentation mode, which shows the documentation index page instead of the home page.



Choose	Type	Doku	Version	Description	Path
Compiler Library	internal				
External Standard Libraries	dir			auto attached libraries	Library
Dump.interface	file	Doku	23.10.2014	Dump outputs for misc memory data	Library
StringMethods.module	file	Doku	29.11.2014	Standard string functions like "Asc()", "Bin()", "Str(...	Library
External Librarys	dir				Library
Custom	dir				Library
Example	dir				Library
Adc.interface	file	Doku	13.01.2015	Adc-Interface	Library
ClkSys.interface	file	Doku	07.10.2013	Atxmega ClockSys-configuration.	Library
Crc16.interface	file	Doku	16.07.2014	CRC16-CCITT: x^16 + x^12 + x^5 + 1	Library
Crc8.interface	file	Doku	07.10.2013	CRC8 generator polynomial (based on 8051 exam...	Library
DCF77.interface	file	Doku	02.10.2013	DCF77 Decoder	Library
Encodings.module	file	Doku	24.05.2014	Encoding/Decoding Functions (Global)	Library
FastFourier.interface	file	Doku	28.09.2013	Fast Fourier Transformation 64 Points	Library
FT800.interface	file	Doku	01.11.2014	EVE FT800 (Spi) Interface/Library (as of Version 20...	Library
Graphics.object	file	Doku	15.11.2014	Graphical offscreen display object	Library
iGraphics.interface	file	Doku	21.03.2014	Universal Graphics Interface	Library
INTn.interface	file	Doku	05.01.2015	Intn Interface	Library
KeyMgr.interface	file	Doku	20.06.2014	Key-Manager	Library
Lcd4.interface	file	Doku	28.04.2014	LCD Interface (4-Bit) for HD44780 and compatible ...	Library
Math64.module	file	Doku	28.09.2013	Global math64 routines.	Library
NetworkFunctions.module	file	Doku	15.11.2014	Global network functions.	Library
OneWire.interface	file	Doku	27.04.2014	Software emulated 1Wire/OneWire.	Library
PCInt.interface	file	Doku	15.12.2013	PCInt-Interface für moderne AVR µC	Library
Sleep.interface	file	Doku	29.12.2013	Sleep interface	Library
SoftSpi.interface	file	Doku	27.04.2014	Software emulated Spi.	Library

Right-click on a folder or entry to create, double-click to edit.

The index page:



Adc.interface (library)

Library Implement them in the project, using #library

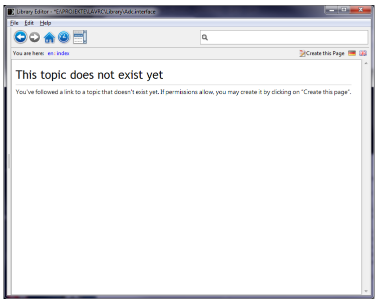
Hardware-Interface for the Controller's Analog-Digital-Converter.

Propertys

Name	Description	Values
Adc.Enable	Enable Adc	
Adc.Disable	Disable Adc	
Adc.Clock =*value*	Set prescaler (cpu-clock/x)	0,2,4,8,16,32,64,128
Adc.Channel =*value*	Set Adc-channel	0-7 = Channel, 14 = 1.23 V, 15 = 0 V (GND)
Adc.Reference	Set Adc-Reference	(see table ".reference" bottom)
Adc.Mode.Free	Set Adc-Mode: Automatically starts a new measurement after completion of the above.	
Adc.Mode.Single	Set Adc mode: Only perform one measurement	
Adc.Result.LeftAdjust	Bit arrangement of the measuring value Return the left. If a measured value is only 8 bits wide, then set this setting and read the measured values with ***Ad***	
Adc.Result.RightAdjust	Bit array of the measured value return right-justified (default).	
Adc.Comparator.Enable	Enable Adc-Comparator.	
Adc.Comparator.Disable	Disable Adc-Comparator.	
Adc.Comparator.Isr =*label*	Set Adc Comparator interrupt routine.	Isr-Name
Adc.Isr =*label*	Set Adc interrupt routine.	Isr-Name
Adc.Isr.Enable	Enable Adc interrupt routine.	
Adc.Isr.Disable	Disable Adc interrupt routine.	

.Reference

With the national flags you can choose between different languages. If there is no page created yet, an appropriate notice page will be displayed:

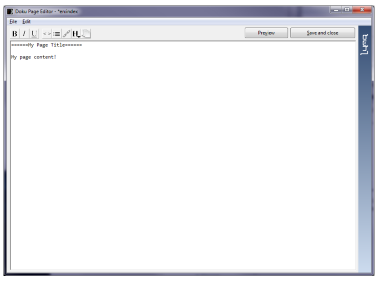


This topic does not exist yet

You've followed a link to a topic that doesn't exist yet. If permissions allow, you may create it by clicking on "Create this page".

By clicking "View Source" to "create this page" or "edit this page", a page can be displayed or edited. **The page will be deleted if it is stored empty.**

Edit the contents of a page:



```
======My Page Title======

My page content!
```

The description of the syntax can be found on the home page. The home page can be opened by clicking on the "Home" button.

CREATING LIBRARIES

GENERAL

- Basics
- Naming conventions
- Sequence of including of library code by the compiler/assembler
- Using external functions from the standard library or other libraries

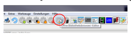
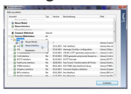
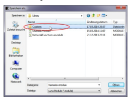
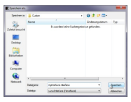
DESCRIPTION/TURORIAL

There are the following types of libraries:

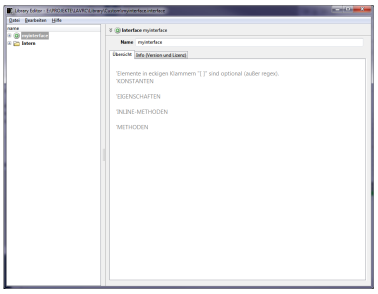
- **Modul** - Function Collection
- **Interface** - Interface (static "Class")
- **Object** - Dynamic memory object with functions.
- **Type** - Static memory object (structure data type) with functions.

CREATE NEW LIBRARY (IDE)

To create a library using the example of an interface, proceed as follows:

- 1. Open Library Browser.

- 2. Right-click on an entry and select "New Interface".

- 3. Open in the File Selection the folder "Custom" (Folder "/Library/Custom" in the Luna-Home-Dir).

- 4. Give the interface a unique name, such as "myinterface" and choose "save".


The interface will be created and appears in a new library editor window. In the list on the right you can edit the name and information of the interface. The interface name is the base name - always specified in Luna code (similar to ***Spi.[..]***). By right-clicking on the interface name or subordinate entries left in the list, you can now create constants, properties, inlines, methods, etc.


Library Editor - E:\PROJEKTE\LAVRC\Library\Custom\myinterface.interface
Datei
Bearbeiten
Hilfe
name
myinterface
Intern
Interface myinterface
Übersicht
Info (Version und Lizenz)

Name: myinterface

```
'Elemente in eckigen Klammern "[ ]" sind optional (außer regex).
'KONSTANTEN

'EIGENSCHAFTEN

'INLINE-METHODEN

'METHODEN
```

CREATE LIBRARY - BASICS

The external libraries are processed through this extra-existing library editor and / or created. The library editor supports the author of a library graphically by overviews and simple choices when creating methods and properties.

The library functions are classically programmed in **assembler**.

In the source code of the interfaces and modules, functions and macros of the standard library can be used. A external label can imported by ***.import*** . The use of functions of other libraries is possible. By using cleverly named constants can be queried whether the Luna programmer has also incorporated the necessary library.

To avoid name collisions, all identifiers must be unique and unambiguous within a library.

It can be used all general Präprozessorfunktionen of Luna assembler as in Inline-Assembler possible, e.g. ***hi8()***, ***lo8()***, shift-commands and so on. In addition, special preprocessor commands are available that make sense only in the creation of libraries (see below).

See also:

- Preprocessor
- Standard-Defines (Names) of the assembler registers

SPECIAL PRÄPROZESSORFUNKTION(S)

MAKEIDENTIFIER(STRING)

Created from the string passed an identifier (symbol). The string is then no longer considered by the assembler as a string, but as a constant name (symbol). If this constant name already has a value, it can then be used as Such in the assembler source code.

Example

Allocation 0xf0 on the already existing register symbol *PORTB*. The constant *MyPort* then refers to the constant (symbol) *PORTB* so that the value of *PORTB* is used in the result here.

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

The following code demonstrates the assigning of a specific port-pin in the Luna-code and the further processing in the assembler source code in an interface.

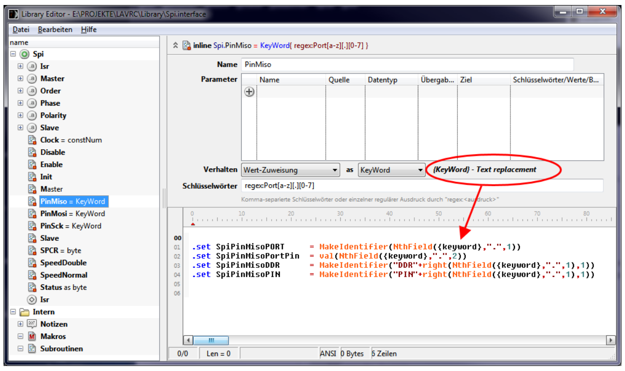
Luna-Code

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

Interface Inline-Method "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))
```

In the Library-Editor



inline Spi.PinMiso = KeyWord{ regex:Port[a-z][.][0-7] }

Name: PinMiso

Parameter:

Verhalten: Wert-Zuweisung as KeyWord

Schlüsselwörter: regex:Port[a-z][.][0-7]

Komma-separierte Schlüsselwörter oder einzelner regulärer Ausdruck durch "regex:<ausdruck>"

```
.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))
```

In such assignments which is a bit like word processing. It is the input of the programmer prepared so that you can work on the assembly level useful with the information. The declaration of this inline method in turn ensures that the programmer without receiving an error message that can only enter what he can add to. Here, this is ensured by the regular expression.

In this example, the four constants then contain the following values:

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

METHODS AND INLINE-METHODS

- **Inline-Methods**
  The source code included is used directly at the point of use. There is *no* subroutine call. The first parameter or a real value assignment is stored to the register block A, so register _HA0 to _HA3 (depending on the value of width). All further on the stack. The parameters are retrieved in the order of declaration (from left to right) from the stack.

- **Methoden**
  The source code included is treated as *sub routine*. There is a call to the corresponding labels as subroutine calls. The first parameter or a real value assignment is stored to the register block A, so register _HA0 to _HA3 (depending on the value of width). All further on the stack. The parameters are retrieved in the order of declaration (from left to right) from the stack. Before you pick up the other parameters from the stack, you have to pop/push the return address of the subroutine call first with the built-in-assembler macros "**PopPC**" and "**PushPC**" (see standard library).

Note

If a real value assignment (no text replacement) used in a method/inline-method, it will displace the first parameter store. That the first parameter will no longer end up in the register block A. The first parameter are stored to the stack like all the other parameters. The register block A is then the value assignment.

Assembler Register defines

The defines (name aliases) for registers 0-31 are predefined for for inline assembler. This can be changed with the directive ***.def***..Registers 0-31 are divided into 8 registers groups. Luna uses none of the registers as constantly entrained status or pointer. It can therefore be used and changed all the registers at any time.

The Register Groups:

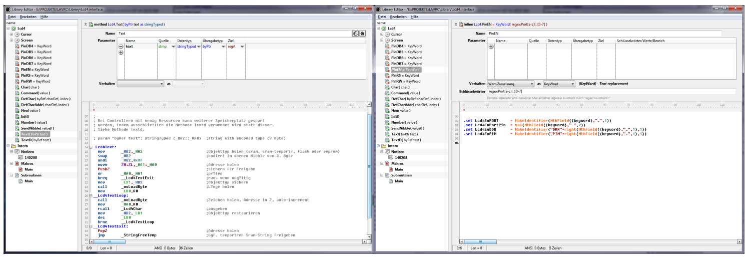
Group	Register	Name
Low A	R0	_LA0
	R1	_LA1
	R2	_LA2
	R3	_LA3
Low B	R4	_LB0
	R5	_LB1
	R6	_LB2
	R7	_LB3
Temp 1	R8	_TMP0
	R9	_TMP1
	R10	_TMP2
	R11	_TMP3
Temp 2	R12	_TMPA
	R13	_TMPB
	R14	_TMPC
	R15	_TMPD
High A	R16	_HA0
	R17	_HA1
	R18	_HA2
	R19	_HA3
High B	R20	_HB0
	R21	_HB1
	R22	_HB2
	R23	_HB3
Low Pointer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

Using of the Groups

- **LA,LB,Temp1 and Temp2:** Be used for temporary data within the internal functions.
- **HA and HB:** Be used as general working registers.
- **Low/High-Pointer:** Are classified as Type Flags, array index, as an intermediate for shifting on the stack, as a classical pointer to access memory areas, as well as the call and used within internal functions.

The naming conventions for source within a library are:

- **Constants/Defines (.def, .set, .equ):**
- **Main-Label: _[]**
- **Sub-Label: __[]**



method Lcd4.Text(byPtr text as stringTyped)

```
;
; Bei Controllern mit wenig Resourcen kann weiterer Speicherplatz gespart
; werden, indem ausschlie?lich die Methode Textd verwendet wird statt dieser.
; Siehe Methode Textd.

; param "byRef text": stringTyped (_HA2::_HA0)  ;string with encoded type (3 Byte)

_Lcd4Text:
  mov       _HB2,_HA2                      ;Objekttyp holen (sram, sram-tempor?r, flash oder eeprom)
  swap      _HB2                           ;kodiert im oberen Nibble vom 3. Byte
  andi      _HB2,0x0F
  movw      ZH:ZL,_HA1:_HA0                ;Adresse holen
  PushZ                                    ;sichern f?r Freigabe
  or        _HA0,_HA1                      ;pr?fen
  breq      __Lcd4TextExit                 ;raus wenn ung?ltig
  mov       _LB1,_HB2                      ;Objekttyp sichern
  call      _ooLoadByte                    ;L?nge holen
  mov       _LB0,R0
__Lcd4TextLoop:
  call      _ooLoadByte                    ;Zeichen holen, Adresse in Z, auto-increment
  mov       _HA0,R0
  rcall     _Lcd4Char                      ;ausgeben
  mov       _HB2,_LB1                      ;Objekttyp restaurieren
  dec       _LB0
  brne      __Lcd4TextLoop
__Lcd4TextExit:
  PopZ                                     ;Adresse holen
  jmp       _StringFreeTemp                ;Ggf. tempor?ren Sram-String freigeben
```

inline Lcd4.PinEN = KeyWord(regex:Port[a-z][.][0-7])

Name: PinEN

Verhalten: Wert-Zuweisung as KeyWord (KeyWord) - Text replacement

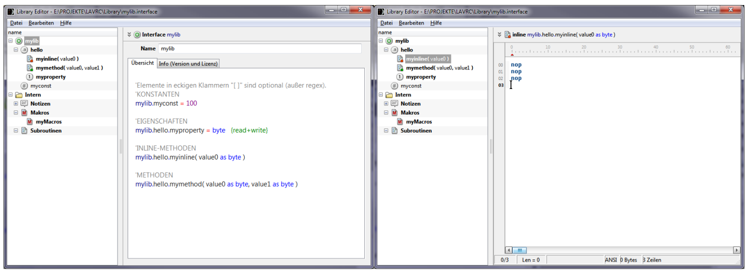
Schlüsselwörter: regex:Port[a-z][.][0-7]

```
.set Lcd4EnPORT    = MakeIdentifier(NthField({keyword},".",1))
.set Lcd4EnPortPin = val(NthField({keyword},".",2))
.set Lcd4EnDDR     = MakeIdentifier("DDR"+right(NthField({keyword},".",1),1))
.set Lcd4EnPIN     = MakeIdentifier("PIN"+right(NthField({keyword},".",1),1))
```

SEQUENCE LIBRARY CODE

The compiler / assembler binds inline code (the source of inline methods) right there, where it appears as text in the Luna-code. Before including macros and text replacement to be processed.

The library source code (methods and subroutines) can be integrated together presented in the order as in the Library Editor. This unused sources are hidden.



```
'Elemente in eckigen Klammern "[ ]" sind optional (außer regex).
'KONSTANTEN
mylib.myconst = 100

'EIGENSCHAFTEN
mylib.hello.myproperty = byte   {read+write}

'INLINE-METHODEN
mylib.hello.myinline( value0 as byte )

'METHODEN
mylib.hello.mymethod( value0 as byte, value1 as byte )
```

```
inline mylib.hello.myinline( value0 as byte )
nop
nop
nop
```

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

uart0.baud = 19200
uart0.Recv.enable
uart0.Send.enable

#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()
```

The resulting assembly code:

```
;{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

[...]
```

LIBRARIES - EXTERNAL FUNCTIONS

External functions and macros can be used in the source code within a library. An importation is not usually necessary because the compiler / assembler recognizes appropriate dependencies. If a dependency is not recognized, the import of a label can be obtained by

```
.import _externalLabel
```

be instructed.
Mathematical functions, String functions, Conversion routines and much more. must therefore not be rewritten.

LIBRARIES - CREATE MODULE

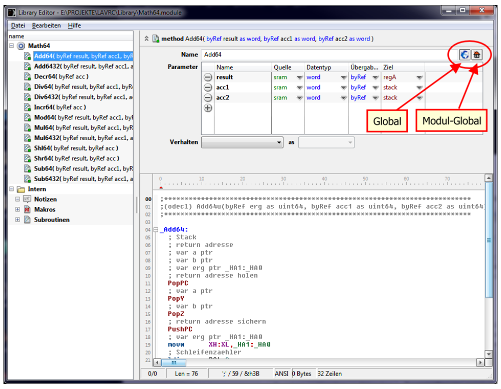
A module-library is a function collection (collection of methods). For a module, you can define the access type in the luna code for each function individually. The two variants of this are

- **Global**
- **Module-Global**

"Global" means that an applied in a module function (method) directly with the function name in the Luna-code can be used. The use would be as with the built-in functions such as Luna ***format()***, ***str()***,***mul()*** and so on. Modules with global functions so add general functions (in writing) and can not be distinguished syntactically to the built-in functions.

"Module-Global" means that the method can be used only in connection with the module name. for example ***Math64.Add()*** and so on.

The behavior can be per method in the declaration set separately:



Library Editor - E:\PROJEKTE\LAVRC\Library\Math64.module
Datei
Bearbeiten
Hilfe
name
Math64
Add64(byRef result, byRef acc1, byR
Add6432(byRef result, byRef acc1, a
Decr64(byRef acc)
Div64(byRef result, byRef acc1, byR
Div6432(byRef result, byRef acc1, a
Incr64(byRef acc)
Mod64(byRef result, byRef acc1, byR
Mul64(byRef result, byRef acc1, byR
Mul6432(byRef result, byRef acc1, a
Shl64(byRef result, byRef acc)
Shr64(byRef result, byRef acc)
Sub64(byRef result, byRef acc1, byR
Sub6432(byRef result, byRef acc1, a
Intern
Notizen
Makros
Subroutinen
Verhalten
as
Global
Modul-Global

method Add64(byRef result as word, byRef acc1 as word, byRef acc2 as word)

Name: Add64

Parameter:

Name	Quelle	Datentyp	Übergab...	Ziel
result	sram	word	byRef	regA
acc1	sram	word	byRef	stack
acc2	sram	word	byRef	stack

```
;***************************************************************************
;(odecl) Add64u(byRef erg as uint64, byRef acc1 as uint64, byRef acc2 as uint64)
;***************************************************************************

_Add64:
  ; Stack
  ; return adresse
  ; var a ptr
  ; var b ptr
  ; var erg ptr _HA1:_HA0
  ; return adresse holen
  PopPC
  ; var a ptr
  PopY
  ; var b ptr
  PopZ
  ; return adresse sichern
  PushPC
  ; var erg ptr _HA1:_HA0
  movw      XH:XL,_HA1:_HA0
  ; Schleifenzaehler
```

0/0 | Len = 76 | ';' / 59 / &h3B | ANSI | 0 Bytes | 32 Zeilen

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

ELEMENTS OF A MODULE

- Inlines
- Methods

- Notes
- Macros
- Subroutines

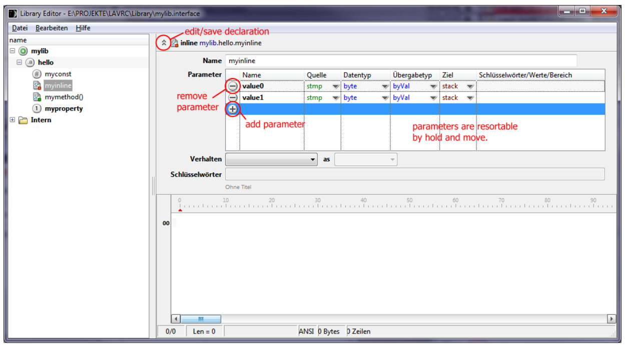
EXAMPLE-METHODS

- Two numerical parameters and return value
- String paramaters and return value

library item - Inline

The source code included is used directly at the point of use. There is *no* subroutine call. The first parameter or a real value assignment is made in the register block A, so register _HA0 to _HA3 (depending on the value of width). All further on the stack. These are fetched in the order of declaration (from left to right) from the stack.

Parameter



Name	Quelle	Datentyp	Übergabetyp	Ziel	Schlüsselwörter/Werte/Bereich
value0	stmp	byte	byVal	stack	
value1	stmp	byte	byVal	stack	

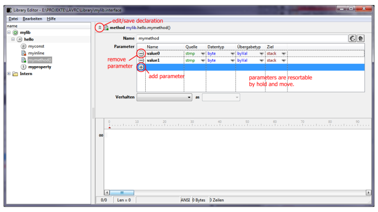
Note

If a real value assignment (no text replacement) used in a method / inline method, the displaced These, if necessary. available first parameter. That The first parameter will no longer end up in the register block A but like all the other parameters on the stack. The register block A is then the value assignment.

The graphical parameter editor automatically changes a corresponding target (stack / RegA).

Library item - Method

The source code included is treated as *sub program*. There is a call to the corresponding labels as subroutine calls. The first parameter or a real value assignment is made in the register block A, so register _HA0 to _HA3 (depending on the value of width). All further on the stack. These are fetched in the order of declaration (from left to right) from the stack. Before you pick up the other parameters, you have to pop/push the return address with the built-assembler macros "**PopPC**" and "**PushPC**" (see standard library) from the stack and back.

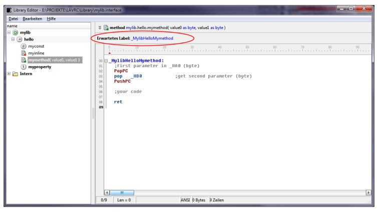


method mylib.hello.mymethod()

Name: mymethod

Name	Quelle	Datentyp	Übergabetyp	Ziel
value0	stmp	byte	byVal	stack
value1	stmp	byte	byVal	stack

In one method, the indication of the correct label is necessary. More than one parameter (Counted including a defined assignment if necessary) is popped from the stack:



method mylib.hello.mymethod(value0 as byte, value1 as byte)

Erwartetes Label: _MylibHelloMymethod

```
_MylibHelloMymethod:
  ;first parameter in _HA0 (byte)
  PopPC
  pop   _HB0           ;get second parameter (byte)
  PushPC

  ;your code

  ret
```

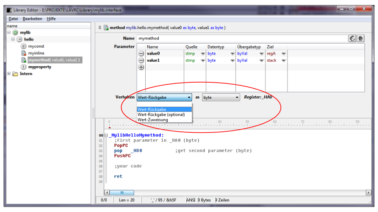
Note

If a real value assignment (no text replacement) used in a method / inline method, the displaced These, if necessary. available first parameter. That The first parameter will no longer end up in the register block A but like all the other parameters on the stack. The register block A is then the value assignment (see below).

The graphical parameter editor automatically changes a corresponding target (stack / RegA).

Method with return value

A return value is expected in the register block A. To declare a method as a function with return value, choose the desired option and the data type. An optional return is also possible. Here, the compiler ignores then the Luna-code the return value if it is not used.

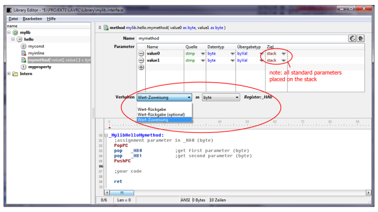


Verhalten: Wert-Rückgabe as byte Register: _HA0

```
_MylibHelloMymethod:
  ;first parameter in _HA0 (byte)
  PopPC
  pop   _HB0           ;get second parameter (byte)
  PushPC

  ;your code

  ret
```

Method with assignment

Alternatively, an assignment can be configured. It should be noted that it is the actual parameters (if any), are stored on the stack. The value of the assignment is stored in the register block A.



method mylib.hello.mymethod(value0 as byte, value1 as byte)

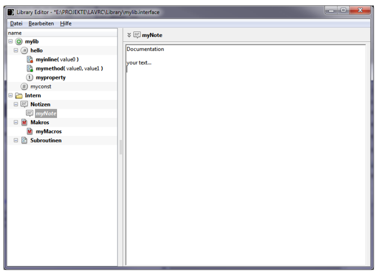
Name	Quelle	Datentyp	Übergabetyp	Ziel
value0	stmp	byte	byVal	stack
value1	stmp	byte	byVal	stack

Verhalten: Wert-Zuweisung as byte Register: _HA0

```
_MylibHelloMymethod:
  ;assignment parameter in _HA0 (byte)
  PopPC
  pop   _HB0           ;get first parameter (byte)
  pop   _HB1           ;get second parameter (byte)
  PushPC

  ;your code

  ret
```

Library Item - Notes

Notes are useful to the programmer documentation of important information.



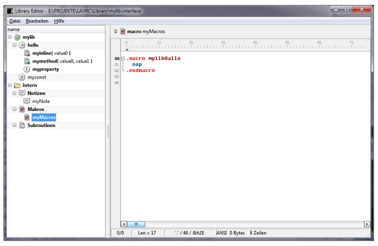
Documentation

your text...

Library Item - Macros

Macros are wide library available and can be stored for a better overview in the specified folder "macros".

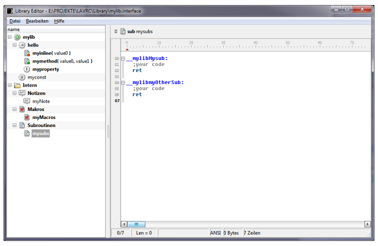
Important: choose a unique name for the macro, eg "mylib_myname"!



```
.macro mylibHallo
  nop
.endmacro
```

Library Item - Sub-functions

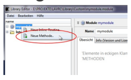
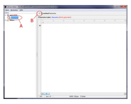
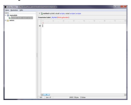
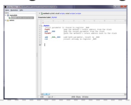
Sub-functions are for the Luna programmers invisible, internal Bibliotheks-Methoden/Routinen which library wide can be used. They are integrated method automatically when using a used (inline).



```
__mylibMysub:
  ;your code
  ret

__mylibmyOtherSub:
  ;your code
  ret
```

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


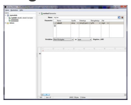
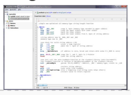
```
_MyAdd:
  ;first parameter is stored in register _HA0 by the compiler at call
  PopPC                 ;pop the method's return address from the stack
  pop    _HB0            ;pop the second parameter from the stack
  PushPC                ;push the methotd's return address back to the stack

  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, Absatz "Modul")

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.

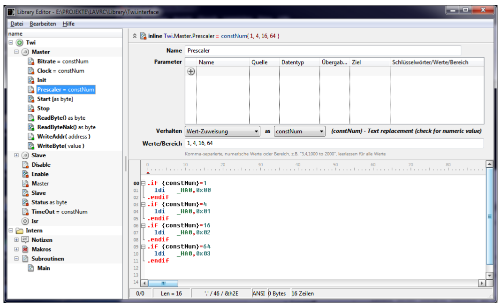
```
avr.device = atmega328p
avr.clock = 20000000
avr.stack = 64

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()
```

LIBRARIES - CREATE INTERFACE

An interface library is similar to a class in Luna. All elements contained in an interface are included within the interface and features may be dependent on properties within an interface. For example, the assignment of a port pin before calling an initialization function and then using functions.

Mostly classical interfaces can be used to control hardware components. Other possible uses include any type of self-contained applications, eg. An emulated software protocol, or an encapsulated mathematical function with more special input parameters. Compared to the normal classes in Luna, special properties can be assigned and/or retrieved - such as Port pins or a keyword-dependent function. Similar to the built-in Luna Interfaces for example, Ports, timers or others.

The following picture shows an inline method that allows only the allocation of certain constant values:



inline Twi.Master.Prescaler = constNum{ 1, 4, 16, 64 }

Name: Prescaler

Parameter

Verhalten: Wert-Zuweisung as constNum {constNum} - Text replacement (check for numeric value)

Werte/Bereich: 1, 4, 16, 64

Komma-separierte, numerische Werte oder Bereich, z.B. "3,4,1000 to 2000", leerlassen für alle Werte

```
.if {constNum}=1
  ldi    _HA0,0x00
.endif
.if {constNum}=4
  ldi    _HA0,0x01
.endif
.if {constNum}=16
  ldi    _HA0,0x02
.endif
.if {constNum}=64
  ldi    _HA0,0x03
.endif
```

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

ELEMENTS OF A INTERFACE

- Library elements and resulting Luna syntax

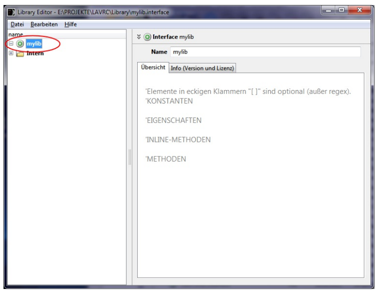
- Keywords
- Constants
- Propertys
- Inlines
- Methods

- Notes
- Macros
- Subroutines

Items and resulting Luna syntax

The type and order of items in a ***interface-library*** determine the resulting Luna syntax. In principle this is similar to a file path.

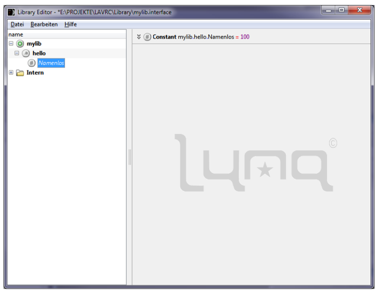
After you create a library, there is only one root element:


Library Editor - E:\PROJEKTE\LAVRC\Library\mylib.interface
Datei
Bearbeiten
Hilfe
name
mylib
Intern
Interface mylib
Übersicht
Info (Version und Lizenz)

Name: mylib

```
'Elemente in eckigen Klammern "[ ]" sind optional (außer regex).
'KONSTANTEN

'EIGENSCHAFTEN

'INLINE-METHODEN

'METHODEN
```

The order and grouping determines how to access an item in Luna code. The elements are automatically in the AutoComplete for the Luna programmer available:



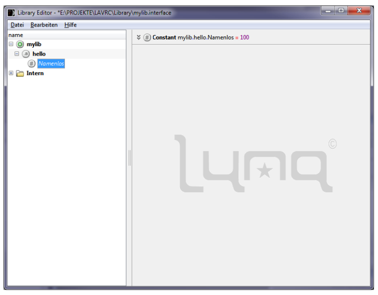
Constant mylib.hello.Namenlos = 100

Access to this constant would be the Luna Code:

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

Library item - Keyword

A keyword is to be understood as a folder. It groups desired functions syntactic / textual and serves no other function.



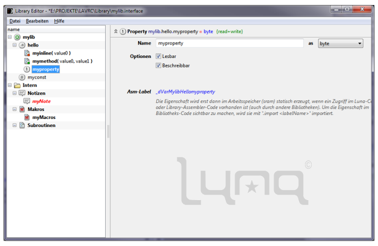
Constant mylib.hello.Namenlos = 100

Library item - Constant

A constant as an element in a library serves the Luna programmers provide fixed predetermined values.

Library Item - Property

Properties from Luna programmers - are used to read and write, depending on the configuration as variables. Within the source code of a library, you can access variables. The corresponding label is displayed on the edit page of the property.



Property mylib.hello.myproperty = byte {read+write}

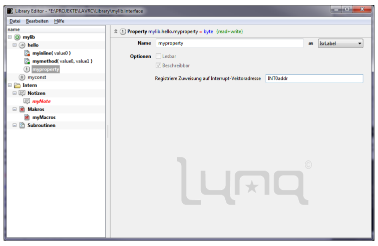
Name: myproperty as byte

Optionen: [✓] Lesbar, [✓] Beschreibbar

Asm-Label: _dVarMylibHellomyproperty

Die Eigenschaft wird erst dann im Arbeitsspeicher (sram) statisch erzeugt, wenn ein Zugriff im Luna-C
oder Library-Assembler-Code vorhanden ist (auch durch andere Bibliotheken). Um die Eigenschaft im
Bibliotheks-Code sichtbar zu machen, wird sie mit ".import <labelName>" importiert.

The assignment of an interrupt vector can also be configured as a special form here. The interrupt is then assigned according to the assigned by Luna programmers ISR method.



Property mylib.hello.myproperty = byte {read+write}

Name: myproperty as IsrLabel

Optionen: [ ] Lesbar, [✓] Beschreibbar

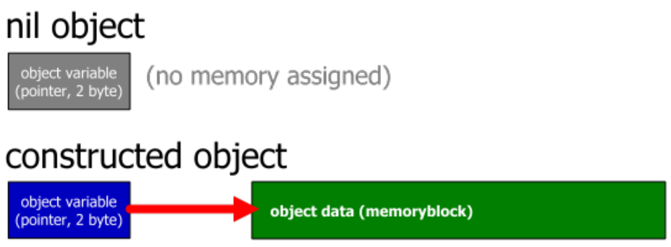
Registriere Zuweisung auf Interrupt-Vektoradresse: INT0addr

LIBRARIES - CREATE OBJECT

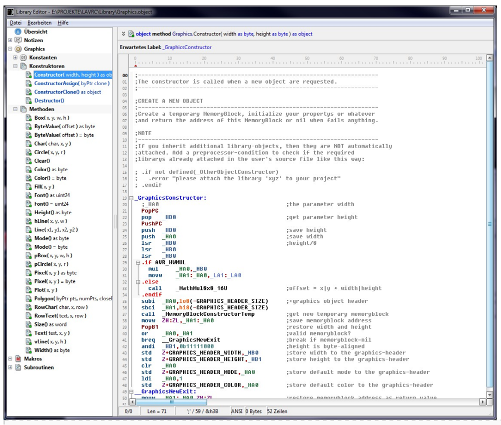
Implemented as of Version	2015.r1

A Objekt-Library is an instantiable class in the form of a library. It adds a new object data type, which must be constructed/instantiated before you can use it. The built-in Luna type "memoryblock" is an example such an object data type. "String" is also an object data type that is derived from the object "MemoryBlock".
The library object, depending on the design such as a memoryblock provide corresponding functions that can be applied to the object. Each object has its private memory after construction. Objects can thus repeatedly exist at the same time in memory. The number of parallel instances is limited only by the available memory.

The object-data is stored in a memoryblock, so the object-variable is *nil* as long as it was not created/initialized. A object can be so *constructed* during the program period and *destroyed*.


nil object
object variable (pointer, 2 byte)
(no memory assigned)
constructed object
object variable (pointer, 2 byte)
object data (memoryblock)

The formerly internal object "Graphics" was outsourced to a library object. It is also a example for a library object implementation. Furthermore, you will find another example *xstring.object*, which represents a separate string object in the folder */Library/Example/*



object method Graphics.Constructor(width as byte, height as byte) as object

Erwartetes Label: _GraphicsConstructor

```
;---------------------------------------------------------------------------
;The constructor is called when a new object are requested.
;---------------------------------------------------------------------------

;CREATE A NEW OBJECT
;---------------------------------------------------------------------------
;Create a temporary MemoryBlock, initialize your propertys or whatever
;and return the address of this MemoryBlock or nil when fails anything.

;NOTE
;---------------------------------------------------------------------------
;If you inherit additional library-objects, then they are NOT automatically
;attached. Add a preprocessor-condition to check if the required
;librarys already attached in the user's source file like this way:

; .if not defined(_OtherObjectConstructor)
;   .error "please attach the library 'xyz' to your project"
; .endif

_GraphicsConstructor:
  ;_HA0                                 ;the parameter width
  PopPC
  pop   _HB0                            ;get parameter height
  PushPC
  push  _HB0                            ;save height
  push  _HA0                            ;save width
  lsr   _HB0                            ;height/8
  lsr   _HB0
  lsr   _HB0
 .if AVR_HWMUL
   mul     _HA0,_HB0
   movw    _HA1:_HA0,_LA1:_LA0
 .else
   call    _MathMul8x8_16U              ;offset = x|y * width|height
 .endif
  subi  _HA0,lo8(-GRAPHICS_HEADER_SIZE)  ;+graphics object header
  sbci  _HA1,hi8(-GRAPHICS_HEADER_SIZE)
  call  _MemoryBlockConstructorTemp     ;get new temporary memoryblock
  movw  ZH:ZL,_HA1:_HA0                 ;save memoryblock address
  PopB1                                 ;restore width and height
  or    _HA0,_HA1                       ;valid memoryblock?
  breq  __GraphicsNewExit               ;break if memoryblock=nil
  andi  _HB1,0b11111000                 ;height is byte-aligned
  std   Z+GRAPHICS_HEADER_WIDTH,_HB0    ;store width to the graphics-header
  std   Z+GRAPHICS_HEADER_HEIGHT,_HB1   ;store height to the graphics-header
  clr   _HA0
  std   Z+GRAPHICS_HEADER_MODE,_HA0     ;store default mode to the graphics-header
  ldi   _HA0,1
  std   Z+GRAPHICS_HEADER_COLOR,_HA0    ;store default color to the graphics-header
__GraphicsNewExit:
```

0/0 | Len = 71 | ';' / 59 / &h3B | ANSI | 0 Bytes | 52 Zeilen

In contrast to an interface or module, no nested syntax can be defined by keywords in an object. The syntax is derived from the functionality of the object and may additionally implemented further objects and their functions.

In a library object, there is only a *root-level* in which the actual functions (methods) are accessible by the Luna developers.

In the folder *Constructors* are the necessary object handlers (basis functions) they must have each object, so that the compiler can create the object, destroy, assign or embed it in expressions.

For the use of the object in expressions with operators - eg *"+", "-", "and"*, the functionality "Operator Overloading" is supported. This optional handlers (basis functions) you can create for operators allows you - in example - to made a addition with your object or similar.

```
#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"
```

ELEMENTS OF A OBJECT

- Constants
- Constructors
- Operator-Handlers
- Methods
- Inlines

- Notes
- Macros
- Subroutines

USAGE EXAMPLE

See also example library ***xstring.object*** in folder */Library/Example/* .The example library is stocked with corresponding descriptions. When creating e.g. a method or constructor, a short explanation is automatically available as comment.

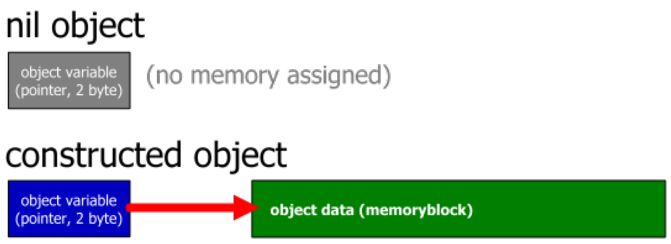
Code-Example:

```
#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
```

Libraries - Object Constructor

A Constructor for a Object-Library is a method which is needed for creating an object. The constructor method is called by the compiler when an object is to be created/initialized. In object libraries at least one constructor for creating is necessary. The constructors for assignment to an object variable (ConstructorAssign) or copy (ConstructorClone), as well as the destructor are optional.

If a variable dimensioned by the type of the object library, it is undefined (nil). It is - as described - a pointer to a dynamic memoryblock. Once an object with the ***new***-operator is created, this variable points to the allocated memory block.


nil object
object variable (pointer, 2 byte)
(no memory assigned)
constructed object
object variable (pointer, 2 byte)
object data (memoryblock)

How Much Memory is allocated for the object after creation, is defined by the author of the object library.

A constructor for creating (new) must therefore contain:

1. If necessary, release of a already allocated memoryblock.
2. Creating a memoryblock using the functions in the standard library in the desired size.
3. Initialize the memoryblock with your data.
4. Return of the address of the new memoryblock, or nil on error.

Constructor types

There are three different types of constructors in an object library that can be executed depending on the type of processing (creating, assigning, copying).

1. ***Constructor*** - Called to create an temporary object: ***new*** *myobject(...)*. The created object is then temporarily stored in memory and is not assigned to a variable. (more than one constructor allowed)
2. ***ConstructorAssign*** - Called when an assignment: *myobj1 =* . This assigns a temporary created object to a variable.
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.

Parameters of a constructor

The parameters that are defined for a constructor that control how and what kind of input data to generate an object are possible. Since multiple constructors may be applied, this may also differ.

Example 1

Constructor()

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

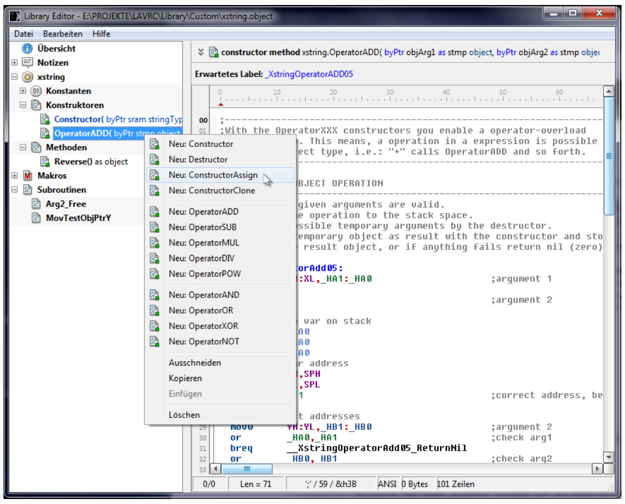
Example 2

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
```

Libraries - Operator-Handler

An operator-handler for a TYPE or OBJECT Library is a (inline) method, which provides the function of an operator. An operator handler is called / embedded when an object is to be processed by an operator, eg "+".



constructor method xstring.OperatorADD(byPtr objArg1 as stmp object, byPtr objArg2 as stmp objec

Erwartetes Label: _XstringOperatorADD05

```
;-----------------------------------------------------------------------
;With the OperatorXXX constructors you enable a operator-overload
. This means, a operation in a expression is possible
ect type, i.e.: "+" calls OperatorADD and so forth.
-----------------------------------------------------------------------

BJECT OPERATION
-----------------------------------------------------------------------
given arguments are valid.
e operation to the stack space.
ssible temporary arguments by the destructor.
emporary object as result with the constructor and sto
 result object, or if anything fails return nil (zero)

orAdd05:
:XL,_HA1:_HA0                   ;argument 1

                                ;argument 2

 var on stack
A0
A0
A0
r address
,SPH
,SPL
1                               ;correct address, be

t addresses
movw        YH:YL,_HB1:_HB0     ;argument 2
or          _HA0,_HA1           ;check arg1
breq        __XstringOperatorAdd05_ReturnNil
or           HB0, HB1           ;check arg2
```

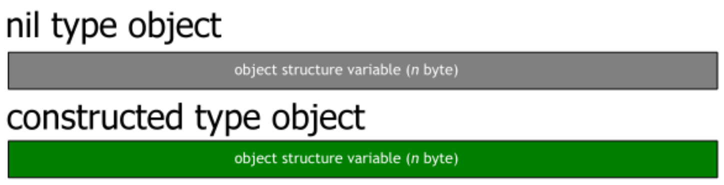
Usage

Processing Type and Object operator handlers are different. When you create a new operator-handler, an appropriate description will be added automatically. The description explains the structure and the use of an operator-handler.

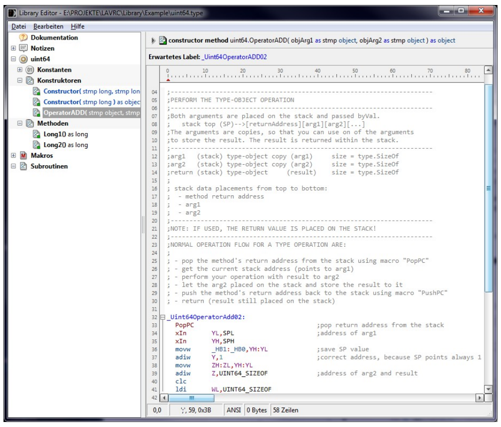
LIBRARIES - CREATE TYPE

Implemented as of Version	2015.r1

A Type Library is a static memory object with functions in the form of a library. It adds a new structure data type. A Type Library (Type object) is similar to a structure in Luna, which provide additional functions and can also be used in expressions.


nil type object
object structure variable (n byte)
constructed type object
object structure variable (n byte)

The Type object can therefore make according to design appropriate functions that can be applied to the object. Each object has its own local and static memory. Therefore Type objects are similar to dimensioned variables (local/temporary or global).


Library Editor - E:\PROJEKTE\LAVRC\Library\Example\uint64.type
Datei
Bearbeiten
Hilfe
Dokumentation
Notizen
uint64
Konstanten
Konstruktoren
Constructor(stmp long, stmp lon
Constructor(stmp long) as objec
OperatorADD(stmp object, stmp
Methoden
Long1() as long
Long2() as long
Makros
Subroutinen

constructor method uint64.OperatorADD(objArg1 as stmp object, objArg2 as stmp object) as object

Erwartetes Label: _Uint64OperatorADD02

```
;-----------------------------------------------------------------------
;PERFORM THE TYPE-OBJECT OPERATION
;-----------------------------------------------------------------------
;Both arguments are placed on the stack and passed byVal.
;   stack top (SP)-->[returnAddress][arg1][arg2][...]
;The arguments are copies, so that you can use on of the arguments
;to store the result. The result is returned within the stack.
;-----------------------------------------------------------------------
;arg1   (stack) type-object copy (arg1)     size = type.SizeOf
;arg2   (stack) type-object copy (arg2)     size = type.SizeOf
;return (stack) type-object     (result)    size = type.SizeOf
;
; stack data placements from top to bottom:
;  - method return address
;  - arg1
;  - arg2
;-----------------------------------------------------------------------
;NOTE: IF USED, THE RETURN VALUE IS PLACED ON THE STACK!
;-----------------------------------------------------------------------
;NORMAL OPERATION FLOW FOR A TYPE OPERATION ARE:
;
; - pop the method's return address from the stack using macro "PopPC"
; - get the current stack address (points to arg1)
; - perform your operation with result to arg2
; - let the arg2 placed on the stack and store the result to it
; - push the method's return address back to the stack using macro "PushPC"
; - return (result still placed on the stack)

_Uint64OperatorAdd02:
  PopPC                            ;pop return address from the stack
  xIn     YL,SPL                   ;address of arg1
  xIn     YH,SPH
  movw    _HB1:_HB0,YH:YL          ;save SP value
  adiw    Y,1                      ;correct address, because SP points always 1
  movw    ZH:ZL,YH:YL
  adiw    Z,UINT64_SIZEOF          ;address of arg2 and result
  clc
  ldi     WL,UINT64_SIZEOF
```

0,0 | ';', 59, 0x3B | ANSI | 0 Bytes | 58 Zeilen

In contrast to an interface or module, no nested syntax can be defined by keywords in an object. The syntax is derived from the functionality of the object and may additionally implemented further objects and their functions.

In a Type object there is only a root-level in which the actual functions (methods) are intended for use by the Luna developers.

In the folder *Constructors* you find the *optional* usable object-handlers (basis functions), they can have each object for initialize, assign or embedding in expressions. **For type objects it is possible to have no handlers. In this case the compiler will automatically use the default Constructor/Destructor for type objects.***

For the use of the object in expressions with operators - eg. *"+", "-", "and"*, the function "operator overloading" is supported. This optional handlers (basis functions) for operators allows you, for example, an Addition with two different objects.

ELEMENTS OF A TYPE OBJECT

- Constants
- Constructors
- Operator-Handlers
- Methods
- Inlines

- Notes
- Macros
- Subroutines

USAGE EXAMPLE

See also example library ***xstring.object*** in folder */Library/Example/*.The example library is stocked with corresponding descriptions. When creating e.g. a method or constructor, a short explanation is automatically available as comment.

Code-Example:

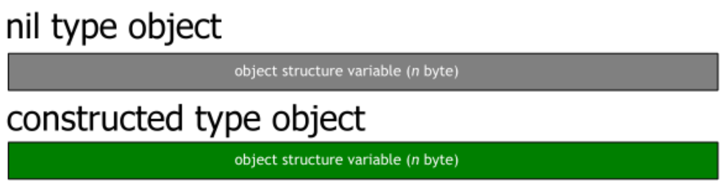
```
#library "Library/Example/uint64.type"

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
```

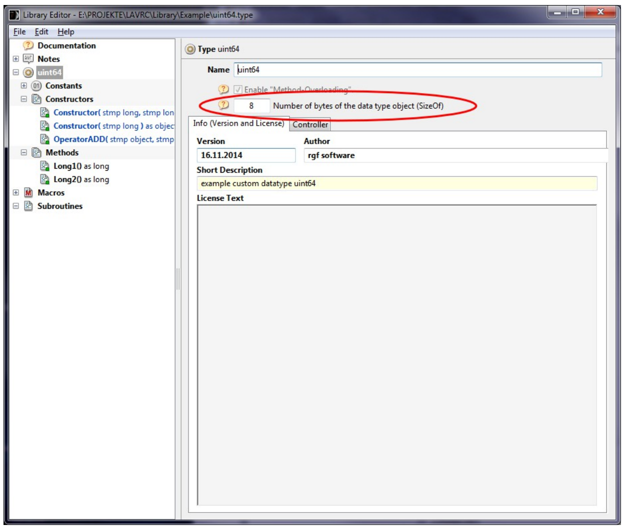
Libraries - Type Constructor

A constructor for a type library is a method that is used for generating an object. The constructor method is called by the compiler when an object is to be created / initialized. For Type Libraries a constructor is *optional*, since the memory of the object is static as in a structure.

If a variable dimensioned by the type of the type library, it is already the space in the specified size.


nil type object
object structure variable (*n* byte)
constructed type object
object structure variable (*n* byte)

How Much Memory an type-object allocates is defined by the author of the Type Library.



Library Editor - E:\PROJEKTE\LAVRC\Library\Example\uint64.type

Documentation
Notes
uint64
Constants
Constructors
Constructor(stmp long, stmp lon
Constructor(stmp long) as objec
OperatorADD(stmp object, stmp
Methods
Long1() as long
Long2() as long
Macros
Subroutines

Type uint64
Name: uint64
[✓] Enable "Method-Overloading"
8 Number of bytes of the data type object (SizeOf)
Version: 16.11.2014
Author: rgf software
Short Description: example custom datatype uint64
License Text

Constructor types

In the type library, there is only an optional standard constructor.

1. ***Constructor*** - Called to create/initialize an type-object: ***new*** *myobject(...)*. The created object is then temporarily stored in memory (on the stack) and is not assigned to a variable. More than one constructor are allowed.

The assignment to a variable, copying (cloning) or destroying is performed automatically by internal built-in functions.

Parameters of a constructor

The parameters that are defined for a constructor controls how and what kind of input data are possible while creating an type-object . Since multiple constructors may be applied, this may also differ.

Example 1

Constructor()

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

Example 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-LIBRARY)

Compiler Standard library (automatically included).

The standard library *StdLib.interface* contains all declarations and functions the compiler used internally. Without standard functions, the compiler can not build a functional binary. **It is strongly recommended that you should not make any changes on the standard library, except it were through an errata or through a support note.**

The standard library contains the following components:

- Built-in Objects

Moreover, it provides *globale* functions and macros that can be used in external libraries or inline assembler.

- List of public global constants
- List of public global macros
- List of public global functions

LIST OF CONSTANTS/SYMBOLS

This list shows global constants and symbols, which can be used in assembler source code in libraries or Luna-inline-Asm.

CONSTANTS

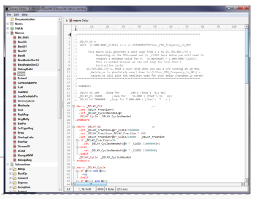
Name	Description
AVR_ADDRWRAP	Nonzero if the address wrap support (Relative jumps over flash end to the beginning and flipped).
AVR_CORE	AVR Core-Type (Number)
AVR_CLOCK, _CLOCK	Defined Clock
AVR_CODE_START_ADDR	Defined Code start address in the flash.
AVR_DEVICE, DEVICE	Controller name (string)
AVR_EEPROM_ACCESS	Nonzero if EEPROM accesses in Luna source.
AVR_EEPROM_ACCESS_HIGH	Non-zero when the EEPROM memory is greater than 255 bytes.
AVR_HWMUL	Nonzero if hardware multiplication are supported.
AVR_HWJMP	Nonzero if direct jumps/call are supported (call, jmp, ..)
AVR_MEGA	Nonzero if Atmega controller
AVR_XMEGA	Nonzero if Atxmega controller
AVR_PC_SIZE	Number of bytes that are stored in a subroutine call on the stack as a return address.
AVR_STACK_SIZE	Defined stack size (bytes)
AVR_STACK_END	Stack end address pointed to last byte stack (SRAMEND-AVR_STACK_SIZE).

SYMBOLS

The symbols are only available if the corresponding property is true. You are using thepreprocessor function ***defined()*** to check for the presence and *not by value*!

Name	Description
AVR_PC3	3-byte program counter (PC). e.g. direct (return-) addresses (call / ret) have a size of 3 bytes.

GLOBAL MACROS (STDLIB.INTERFACE)



List of globally available macros that can be used in external libraries or inline assembler or need. The entries in the list refer to the folder structure in the library (see picture). The possibly expected parameters or registers used please refer to the description or the code within the respective entry.

MACROS / DELAY /

```
 * **_DELAY_CLK** //- Waiting Number Cycles (cycles) //
 * **_DELEY_US** //- Waiting number of microseconds //
```

MACROS/EXTEND/

- **Extend(Group,SourceType,DestinationType)** - *Universal Conversion (Casting) of numeric values in a tab group.*

MACROS/PUSHPOP/

- **PopPC, PushPC** - *Return address to / from Stack (Automatic 2 or 3 byte program counter).*

- **PopLA, PushLA** - *register group LA (_LA3::_LA0) from/to Stack (4 Byte).*
- **PopLA1, PushLA1** - *register group LA-Low(_LA1::_LA0) from/to Stack (2 Byte).*
- **PopLA2, PushLA2** - *register group LA-High (_LA3::_LA2) from/to Stack (2 Byte).*

- **PopLB, PushLB** - *register group LB (_LB3::_LB0) from/to Stack (4 Byte).*
- **PopLB1, PushLB1** - *register group LB-Low(_LB1::_LB0) from/to Stack (2 Byte).*
- **PopLB2, PushLB2** - *register group LB-High (_LB3::_LB2) from/to Stack (2 Byte).*

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

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

- **PopA2Z, PushA2Z** - *Register (ZH::_HA0) from/to Stack (16 Byte).*
- **PopAll, PushAll** - *All Register from/to Stack (34 Byte, with loop, short Version).*
- **PopAllFast, PushAllFast** - *Register (ZH::_HA0) from/to Stack (34 Byte, direct - fast).*

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

MACROS/SETTYPEREG/

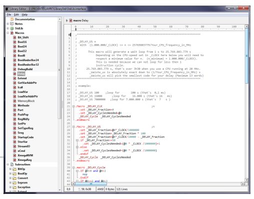
- **SetTypeReg_SramTemp** - *Object type in Register _HA2 set for SRAM/STMP (return values).*
- **SetTypeReg_Flash** - *Object type in Register _HA2 set for FLASH (return values).*
- **SetTypeReg_Eeprom** - *Object type in Register _HA2 set for EEPROM (return values).*

MACROS/SREG/

- **SregSave, SregRestore** - *Status Register on Stack Backup/Restore, global Interrupts disable/restore.*
- **SregSaveRS, SregRestoreRS** - *Status Register on Stack Backup/Restore (Indication of the register), global Interrupts disable/restore.*

- **xIn, xOut** - *Automatic use of **in/out** for Port register access.*
- **xCbi, xSbi** - *Automatic use of **cbi/sbi** for Port register access.*
- **gCbi, gSbi** - *Automatic use of **cbi/sbi** for Port register access (Replacement modifies Register WL).*
- **xSbic, xSbis** - *Automatic use of **sbic/sbis** (Replacement modifies Register _TMPD).*
- **gSbic, gSbis** - *Automatic use of **sbic/sbis** (Replacement modifies Register WL).*

GLOBAL FUNCTIONS (STDLIB.INTERFACE)



List of globally available functions that can be used in external libraries or inline assembler or need. The entries in the list refer to the folder structure in the library (see picture). The possibly expected parameters or registers used please refer to the description or the code within the respective entry.

CONVERT/ATOI/

- **AtoI** - Converting decimal (ASCII) in 32-bit integer (int32).

CONVERT/BIN/

- **ConvBin8** - Converting 8 Bit Integer (uint8) to ASCII binary.
- **ConvBin16** - Converting 16 Bit Integer (uint16) to ASCII binary.
- **ConvBin24** - Converting 24 Bit Integer (uint24) to ASCII binary.
- **ConvBin32** - Converting 32 Bit Integer (uint32) to ASCII binary.

CONVERT/DEC/

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

CONVERT/XTOA/

- **ConvHex8** - Converting 8 Bit Integer (uint8) to ASCII hex.
- **ConvHex16** - Converting 16 Bit Integer (uint16) to ASCII hex.
- **ConvHex24** - Converting 24 Bit Integer (uint24) to ASCII hex.
- **ConvHex32** - Converting 32 Bit Integer (uint32) to ASCII hex.

EEPROM/

- **Read** - Read Byte from Eeprom.
- **Write** - Write Byte into Eeprom.

MAINSTD/

- **ArrayClearEram** - Zero memory area (Eeprom).
- **ArrayClearMemoryBlock** - Zero memory area (Memoryblock).
- **ArrayClearSram** - Zero memory area (SRAM).
- **ArrayReverse** - Memory area reverse/turn (SRAM).

- **MemCmp** - Compare memory areas (SRAM).
- **MemCpy** - Copy memory area (SRAM).
- **MemRev** - Memory area reverse/turn (SRAM).
- **MemSort** - Sort bytes of memory area ascending (SRAM).

- **QSort8u** - Sort value array ascending, 8-Bit (SRAM).
- **QSort16u** - Sort value array ascending, 16-Bit (SRAM).
- **QSort16s** - Sort value array ascending, 16-Bit signed (SRAM).

- **StackSpaceAssignRestore** - Reserve/Release memory space on stack.

- **Wait** - Delay in Seconds

- **Waitms** - Delay in Milliseconds

MEMORYBLOCK/

- **Clear** - Zero MemoryBlock-Data.
- **ClrVar** - Set MemoryBlock-Pointer variable to **nil**,If necessary, existing memory block is released.
- **ClrVarAddr** - Clear backreference to pointer variable in memory block.
- **Compare** - Compare MemoryBlocks.
- **Constructor** - Construct MemoryBlock.
- **Constructor_FromVarRef** - Construct MemoryBlock with Pointer variable-address (byRef), auto-assigned and if necessary, Releasing an already assigned memoryblock.
- **Destructor** - Release MemoryBlock, If necessary, a referenced object variable is set to **nil**.
- **DestructTypedZ** - Memory block release depending on the typeFlags, If necessary, a referenced object variable is set to **nil**.
- **DestructTypedTwo** - Release of two Memoryblocks depending on the typeFlags, If necessary, referenced object variables are set to **nil**.
- **FindByte** - Find a byte in memoryblock.
- **GarbageCollection** - Perform complete garbage collection.
- **SetVar** - Setting new address in memory block pointer variable. Assigning of **nil** releases a existing MemoryBlock.
- **SetVarAddr** - Setting new reference address to pointer variable in the MemoryBlock.
- **Size** - Read Size of a memoryblock (data area).
- **SizeAll** - Size of all memory blocks in memory (complete, including headers).

#LIBRARY

Imports the functionality of a external library to the project.

Syntax:

- **#library** *"Filepath"*

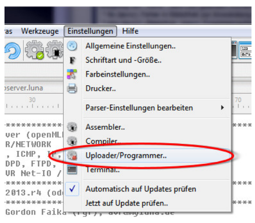
See also: Compiler Sequence

EXAMPLE

```
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



The used Programmer / uploader may be changed in the settings of Luna IDE. The Luna IDE binds the software Avrdude to upload compiled files. All supported by avrdude programmer can therefore be used.

The setting requires the correct setting of the program file, the used interface eg "com1" or "usb", the programmer and the transmission speed. The Luna-IDE provides only a simplified options for the user configuration. For more information, please refer to the documentation for avrdude.

AVRDUDE AND USB

When using the USB interface to install **libusb** and the subsequent registration of the programmer with libusb is necessary for avrdude.

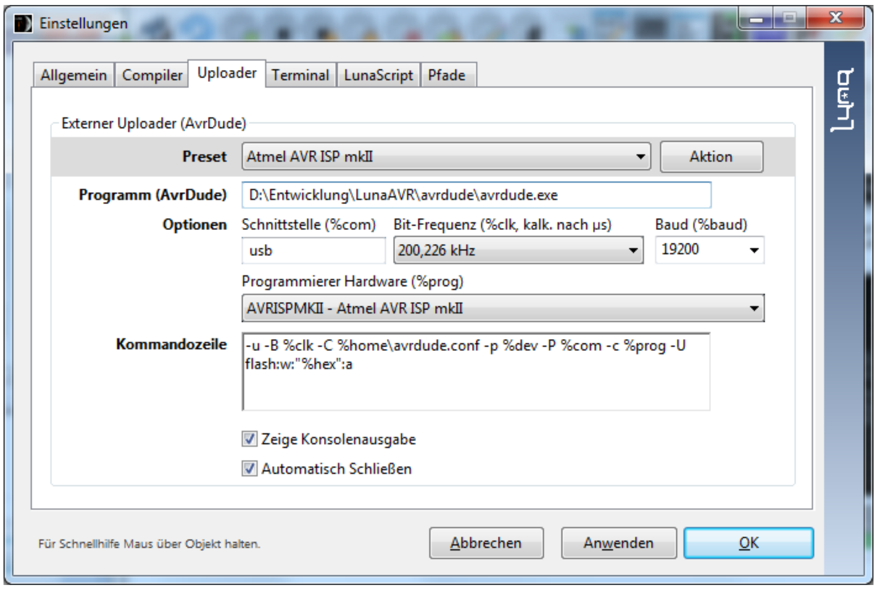
- Official Website
- Download libusb-win32 (Windows XP, 7, Vista, ..)
- Download libusb (Linux/Other)

There is one exception. If a programmer used operating over a com port, such as mySmartUSB light or the bootloader for Arduino Nano V3 and MEGA2560 must be installed no libusb. But for a avrdude version must WITHOUT libusb support was compiled to be used.

EXAMPLES

ATMEL AVR ISP MKII

Settings for Atmel AVR ISP mkII



Externer Uploader (AvrDude)

Preset: Atmel AVR ISP mkII

Programm (AvrDude): D:\Entwicklung\LunaAVR\avrdude\avrdude.exe

Optionen: Schnittstelle (%com): usb | Bit-Frequenz (%clk, kalk. nach µs): 200,226 kHz | Baud (%baud): 19200

Programmierer Hardware (%prog): AVRISPMKII - Atmel AVR ISP mkII

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

[✓] Zeige Konsolenausgabe

[✓] Automatisch Schließen

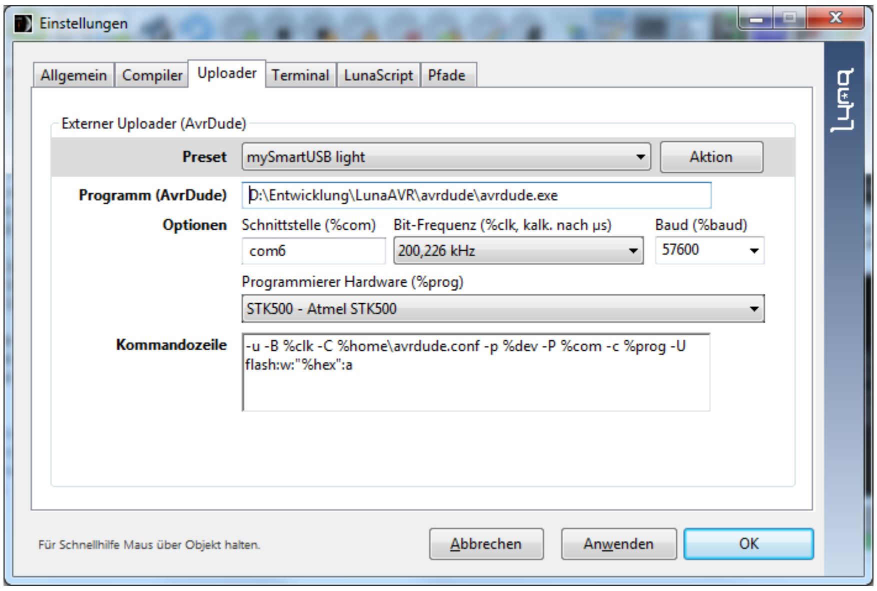
Für Schnellhilfe Maus über Objekt halten.

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

If not, enter the absolute path to **avrdude executable**:

MYSMARTUSB LIGHT

Settings for mySmartUSB light (where the interface is from the adapter):



Externer Uploader (AvrDude)

Preset	mySmartUSB light
Programm (AvrDude)	D:\Entwicklung\LunaAVR\avrdude\avrdude.exe
Schnittstelle (%com)	com6
Bit-Frequenz (%clk, kalk. nach µs)	200,226 kHz
Baud (%baud)	57600
Programmierer Hardware (%prog)	STK500 - Atmel STK500
Kommandozeile	-u -B %clk -C %home\avrdude.conf -p %dev -P %com -c %prog -U flash:w:"%hex":a

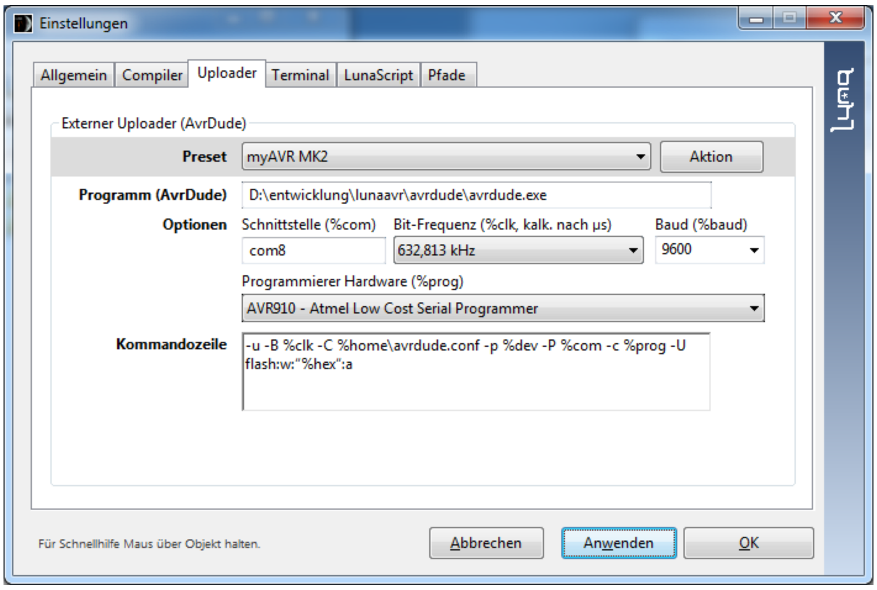
Für Schnellhilfe Maus über Objekt halten.

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

MYSMART MK2

Settings for mySmart MK2, with AVR910/AVR 911 Protokoll.

See also: Website of the manufacturer



Externer Uploader (AvrDude)

Preset	myAVR MK2
Programm (AvrDude)	D:\entwicklung\lunaavr\avrdude\avrdude.exe
Schnittstelle (%com)	com8
Bit-Frequenz (%clk, kalk. nach µs)	632,813 kHz
Baud (%baud)	9600
Programmierer Hardware (%prog)	AVR910 - Atmel Low Cost Serial Programmer
Kommandozeile	-u -B %clk -C %home\avrdude.conf -p %dev -P %com -c %prog -U flash:w:"%hex":a

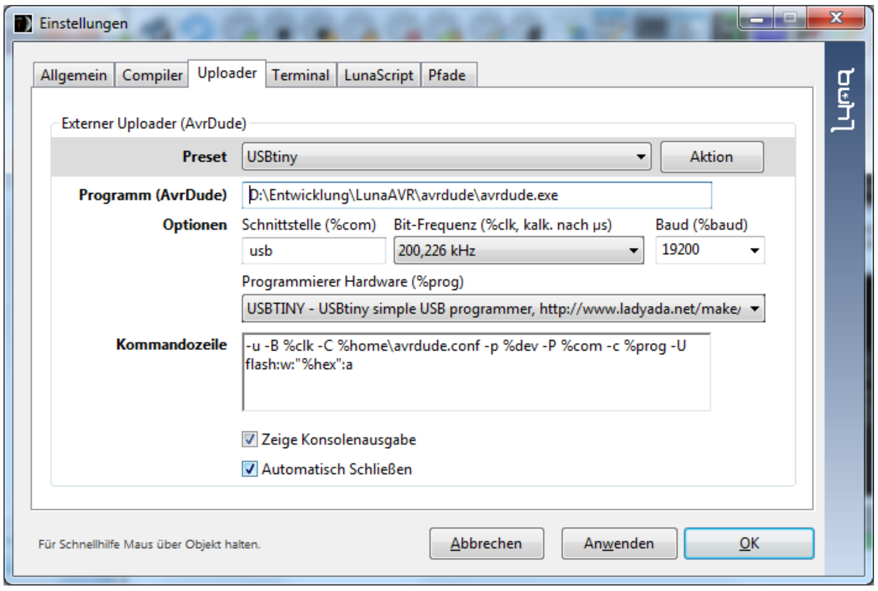
Für Schnellhilfe Maus über Objekt halten.

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

USBTINY

Settings for USBTiny:

For drivers, firmware and installation see also:Website LadyAda


Einstellungen

Externer Uploader (AvrDude)

Preset USBtiny

Programm (AvrDude) D:\Entwicklung\LunaAVR\avrdude\avrdude.exe

Optionen Schnittstelle (%com): usb; Bit-Frequenz (%clk, kalk. nach µs): 200,226 kHz; Baud (%baud): 19200

Programmierer Hardware (%prog): USBTINY - USBtiny simple USB programmer, http://www.ladyada.net/make/

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

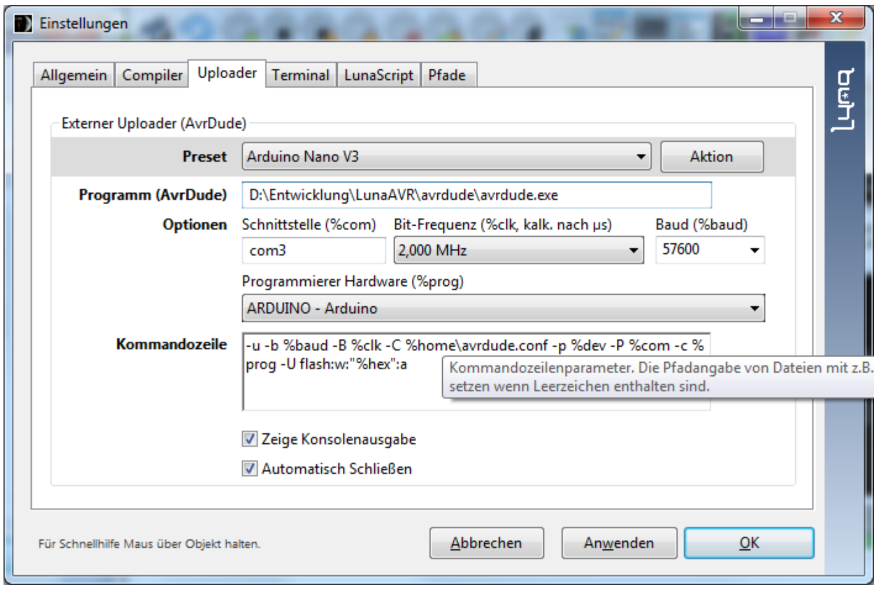
☑ Zeige Konsolenausgabe
☑ Automatisch Schließen

Für Schnellhilfe Maus über Objekt halten.

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

ARDUINO NANO V3

Settings for Arduino Nano V3:


Einstellungen

Externer Uploader (AvrDude)

Preset Arduino Nano V3

Programm (AvrDude) D:\Entwicklung\LunaAVR\avrdude\avrdude.exe

Optionen Schnittstelle (%com): com3; Bit-Frequenz (%clk, kalk. nach µs): 2,000 MHz; Baud (%baud): 57600

Programmierer Hardware (%prog): ARDUINO - Arduino

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

Kommandozeilenparameter. Die Pfadangabe von Dateien mit z.B. setzen wenn Leerzeichen enthalten sind.

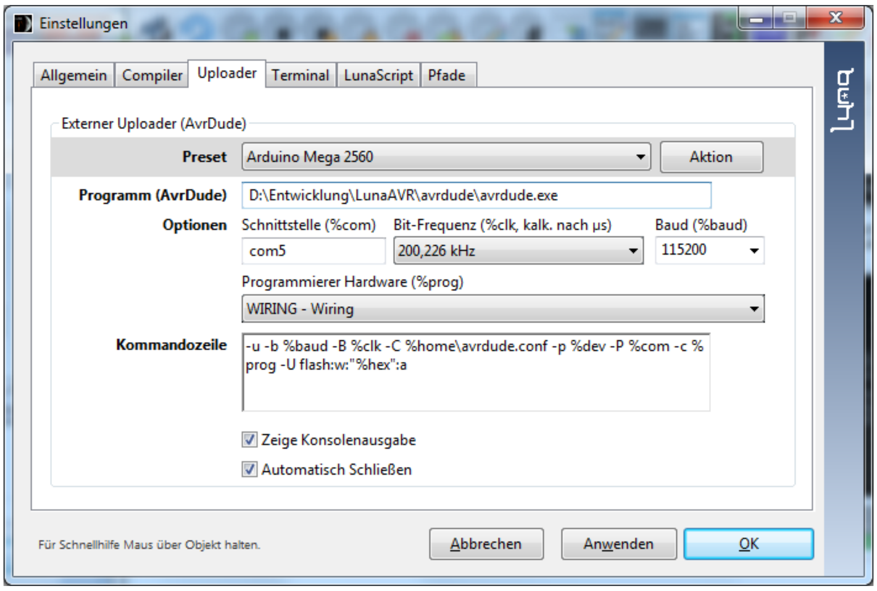
☑ Zeige Konsolenausgabe
☑ Automatisch Schließen

Für Schnellhilfe Maus über Objekt halten.

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

ARDUINO MEGA2560

Settings for Arduino MEGA2560:



Externer Uploader (AvrDude)

Preset Arduino Mega 2560

Programm (AvrDude) D:\Entwicklung\LunaAVR\avrdude\avrdude.exe

Optionen Schnittstelle (%com): com5 · Bit-Frequenz (%clk, kalk. nach µs): 200,226 kHz · Baud (%baud): 115200

Programmierer Hardware (%prog): WIRING - Wiring

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

☑ Zeige Konsolenausgabe

☑ Automatisch Schließen

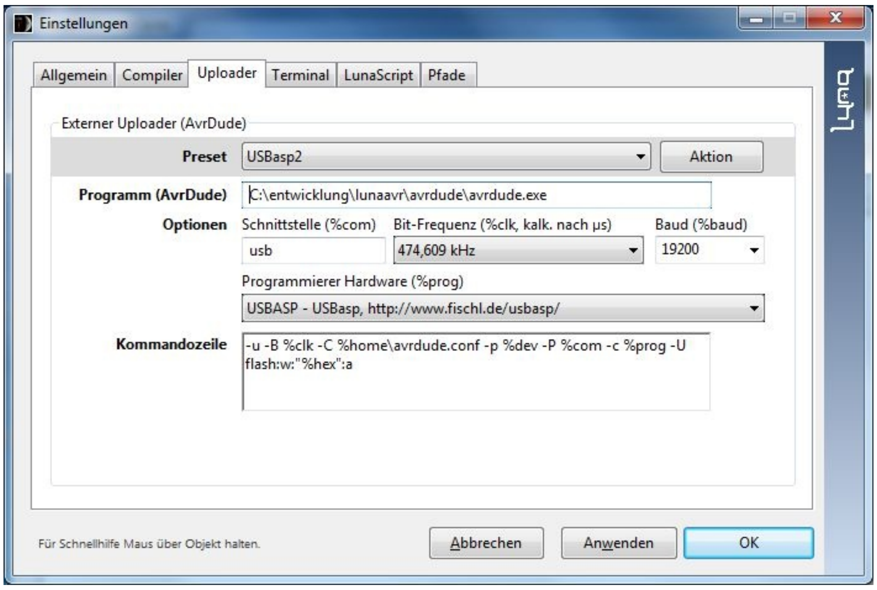
Für Schnellhilfe Maus über Objekt halten.

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

USBASP

Settings for USBasp from Thomas Fischl:

For drivers, firmware and installation see also: Website Thomas Fischl.



Externer Uploader (AvrDude)

Preset USBasp2

Programm (AvrDude) C:\entwicklung\lunaavr\avrdude\avrdude.exe

Optionen Schnittstelle (%com): usb · Bit-Frequenz (%clk, kalk. nach µs): 474,609 kHz · Baud (%baud): 19200

Programmierer Hardware (%prog): USBASP - USBasp, http://www.fischl.de/usbasp/

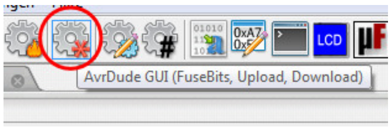
Kommandozeile -u -B %clk -C %home\avrdude.conf -p %dev -P %com -c %prog -U flash:w:"%hex":a

Für Schnellhilfe Maus über Objekt halten.

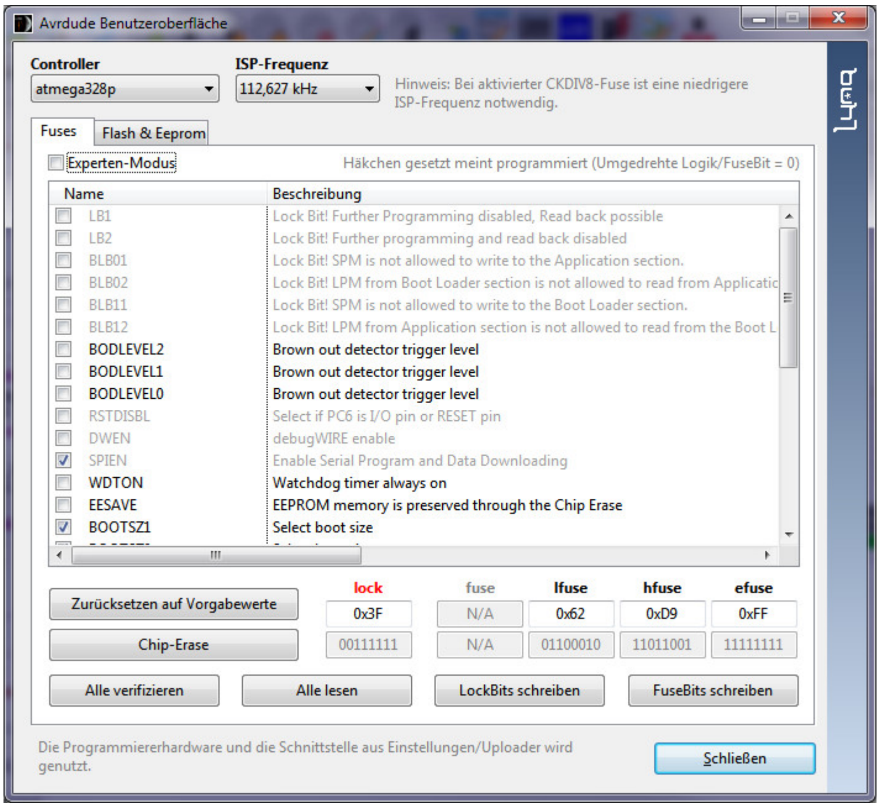
"-u -B %clk -C %home\avrdude.conf -p %dev -P %com -c %prog -U flash:w:"%hex":a"

Fusebits

The ***"AvrDude GUI"*** can be opened using the entry in menu ***"Tools"*** or using the appropriate button:



Window with AvrDude GUI:



Avrdude Benutzeroberfläche

Controller: atmega328p

ISP-Frequenz: 112,627 kHz

Hinweis: Bei aktivierter CKDIV8-Fuse ist eine niedrigere ISP-Frequenz notwendig.

Fuses | Flash & Eeprom

☐ Experten-Modus

Häkchen gesetzt meint programmiert (Umgedrehte Logik/FuseBit = 0)

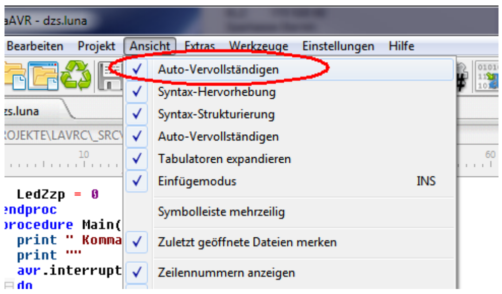
	Name	Beschreibung
☐	LB1	Lock Bit! Further Programming disabled, Read back possible
☐	LB2	Lock Bit! Further programming and read back disabled
☐	BLB01	Lock Bit! SPM is not allowed to write to the Application section.
☐	BLB02	Lock Bit! LPM from Boot Loader section is not allowed to read from Applicatio
☐	BLB11	Lock Bit! SPM is not allowed to write to the Boot Loader section.
☐	BLB12	Lock Bit! LPM from Application section is not allowed to read from the Boot L
☐	BODLEVEL2	Brown out detector trigger level
☐	BODLEVEL1	Brown out detector trigger level
☐	BODLEVEL0	Brown out detector trigger level
☐	RSTDISBL	Select if PC6 is I/O pin or RESET pin
☐	DWEN	debugWIRE enable
✓	SPIEN	Enable Serial Program and Data Downloading
☐	WDTON	Watchdog timer always on
☐	EESAVE	EEPROM memory is preserved through the Chip Erase
✓	BOOTSZ1	Select boot size

	lock	fuse	lfuse	hfuse	efuse
Zurücksetzen auf Vorgabewerte	0x3F	N/A	0x62	0xD9	0xFF
Chip-Erase	00111111	N/A	01100010	11011001	11111111

Die Programmiererhardware und die Schnittstelle aus Einstellungen/Uploader wird genutzt.

All fusebits where changes are considered to be risky are only available when ***"Expert-Mode"*** is activated. For the function of the fusebits, please refer to the respective datasheet of the microcontroller. Microcontrollers were often shipped with active CKDIV8 fusebit. Thereby the controller clock is divided by 8. In this case, a low IPS frequency may be necessary to ensure proper communication with the microcontroller.

Autocomplete



aAVR - dzs.luna

Bearbeiten Projekt Ansicht Extras Werkzeuge Einstellungen Hilfe

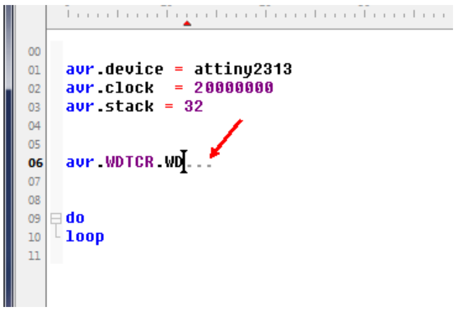
- ✓ Auto-Vervollständigen
- ✓ Syntax-Hervorhebung
- ✓ Syntax-Strukturierung
- ✓ Auto-Vervollständigen
- ✓ Tabulatoren expandieren
- ✓ Einfügemodus INS
- Symbolleiste mehrzeilig
- ✓ Zuletzt geöffnete Dateien merken
- ✓ Zeilennummern anzeigen

```
 Led2zp = 0
endproc
procedure Main(
 print " Komma
 print ""
 avr.interrupt
do
```

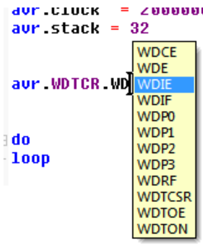
The editor of the Luna-IDE includes an autocomplete feature for numerous commands or controller-specific names and values. The auto-complete function works at the end of the typed string. Inside of a string no proposals are offered.

The string entered by the programmer is checked during entry. If a term is recognized, a proposal is given in grey colour which can be accepted with the TAB-key. If more than one suggestion is possible, three grey dots are displayed. After pressing the TAB-key a menu for selection of the appropriate proposal is displayed. Pressing the TAB-key again will confirm the selected proposal; the ESC-key will cancel the selection.

Almost all command-groups and their properties are implemented in the autocomplete feature. Also constant and register names predefined by the controller manufacturer are completely accessible for the programmer.



```

avr.device = attiny2313
avr.clock  = 20000000
avr.stack = 32


avr.WDTCR.WD...


do
loop

```

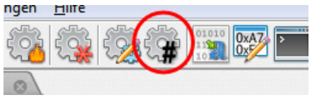


```
avr.clock  = 20000000
avr.stack = 32


avr.WDTCR.WD


do
loop
```

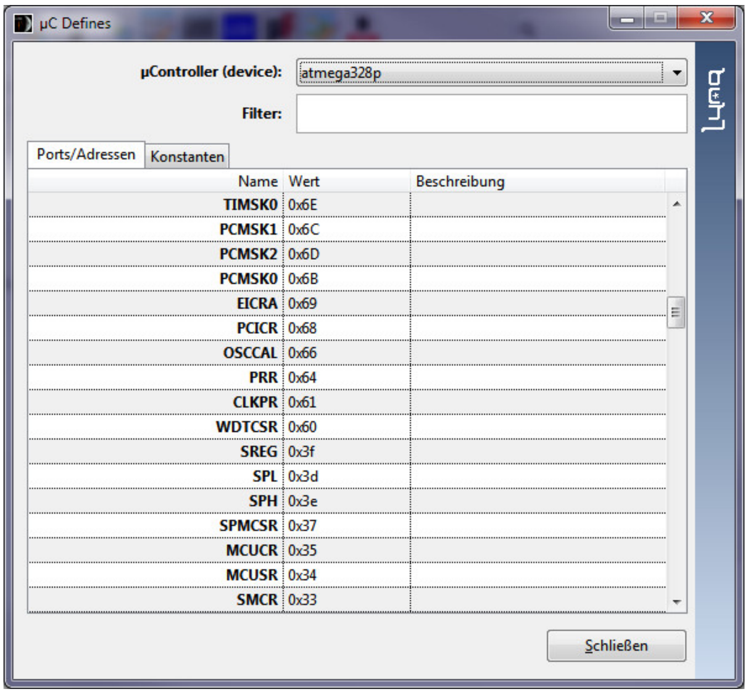
- WDCE
- WDE
- WDIE
- WDIF
- WDP0
- WDP1
- WDP2
- WDP3
- WDRF
- WDTCSR
- WDTOE
- WDTON

Controller Values

For the configuration of particular hardware or when dedicated controller function should be configured manually, the available port names and constants (predefined by the manufacturer of the controller) are necessary. Therefore the Luna-IDE provides an overview, accessible through ***"Zeige alle µC-Defines"*** ("Show all µC-Defines") in menu ***"Projekt"*** or through clicking the corresponding button:



After clicking the butten the overview-window pop up:



µController (device): atmega328p

Filter:

Name	Wert	Beschreibung
TIMSK0	0x6E	
PCMSK1	0x6C	
PCMSK2	0x6D	
PCMSK0	0x6B	
EICRA	0x69	
PCICR	0x68	
OSCCAL	0x66	
PRR	0x64	
CLKPR	0x61	
WDTCSR	0x60	
SREG	0x3f	
SPL	0x3d	
SPH	0x3e	
SPMCSR	0x37	
MCUCR	0x35	
MCUSR	0x34	
SMCR	0x33	

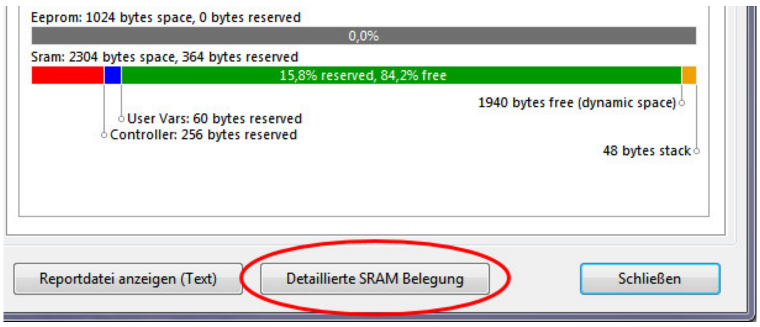
In source-code the access is given via the class "Avr".

BEISPIEL

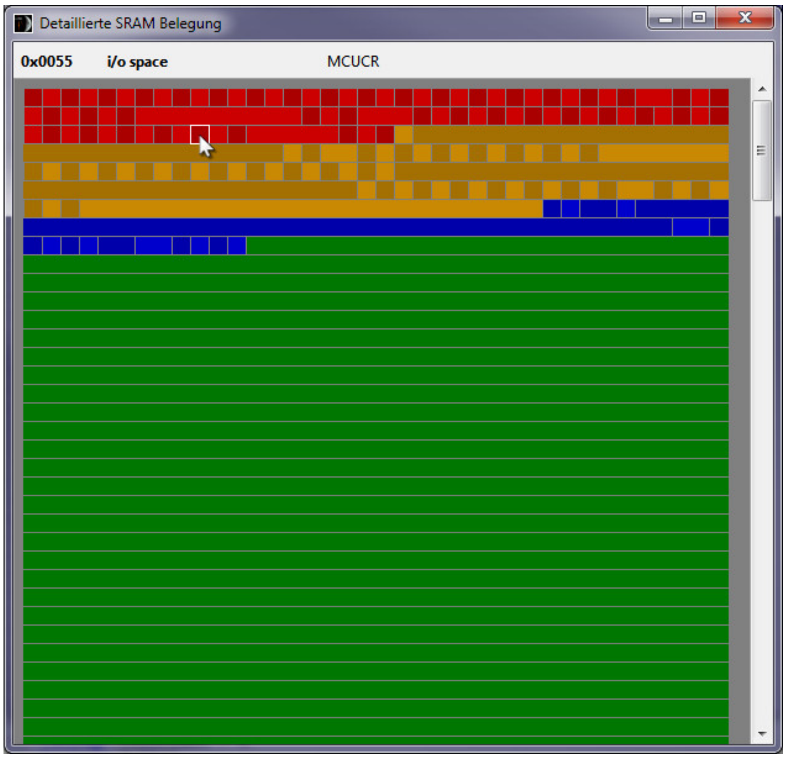
```
avr.TWCR = ((1<<TWINT) or (1<<TWSTA) or (1<<TWEN))

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

Memory Assignment

Since version 2013.R1 a more detailed view concerning the allocation of variables, object, controller-ports etc. in the memory is implemented in the build-report.If one wants to get an overview of this allocation, choose ***"Report anzeigen"*** in menu ***"Projekt"*** and click on the button ***"Detaillierte SRAM Belegung"***:


Eeprom: 1024 bytes space, 0 bytes reserved
0,0%
Sram: 2304 bytes space, 364 bytes reserved
15,8% reserved, 84,2% free
1940 bytes free (dynamic space)
User Vars: 60 bytes reserved
Controller: 256 bytes reserved
48 bytes stack
Reportdatei anzeigen (Text)
Detaillierte SRAM Belegung
Schließen

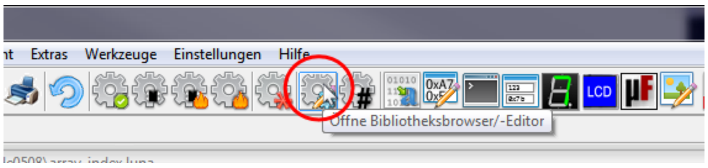
After clicking a new window pops up displaying all memory cells in different colors. Each cell is corresponding to a single byte of the memory. When moving the mouse-pointer to a dedicated cell, in top of the window, the address, section-name, name of the assigned variable port etc. is shown.



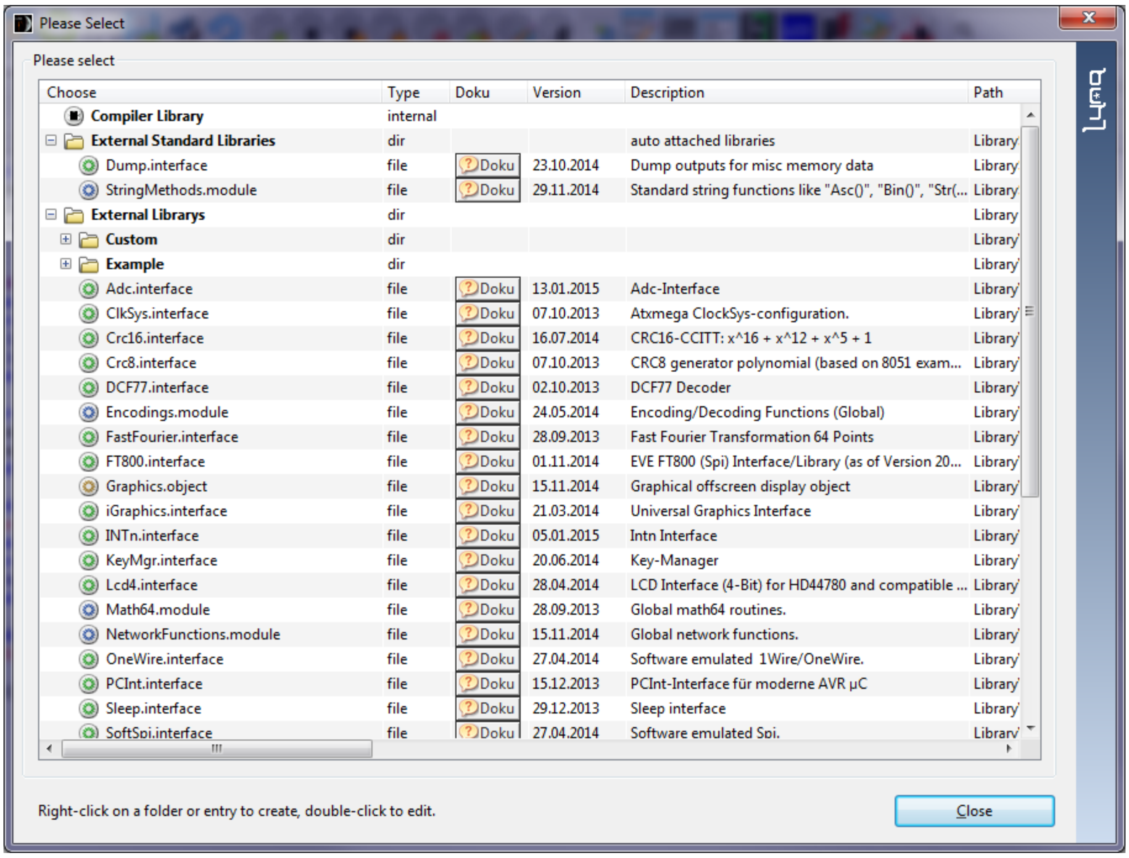
Libraries (Choose)

See also: External Libraries

An overview of the existing, external libraries can be obtained by clicking on the appropriate button or by selecting the menu item *Library Browser/Editor* menu *Tools*.



Subsequently, the library selection opens:



Please Select

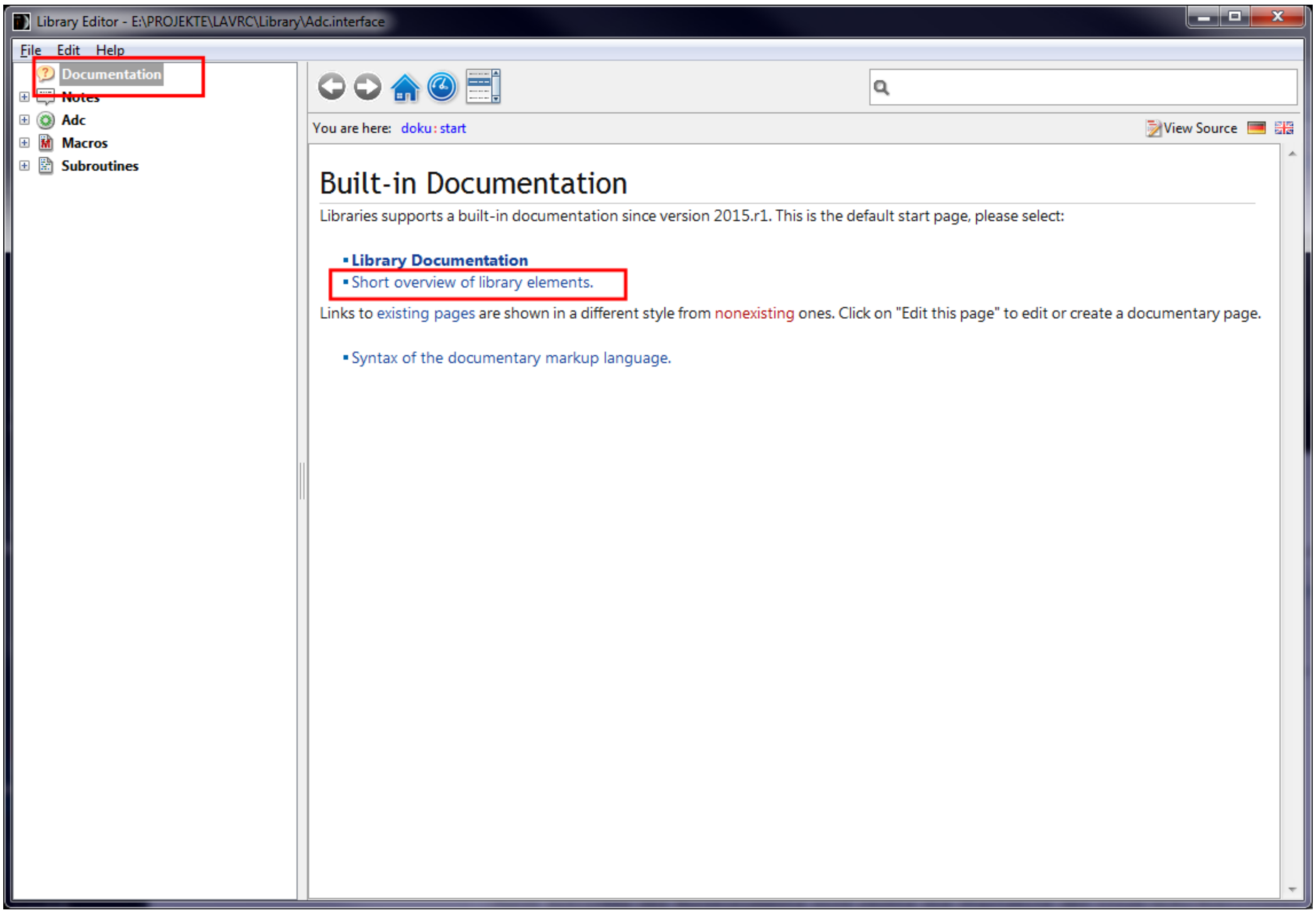
Please select

Choose	Type	Doku	Version	Description	Path
Compiler Library	internal				
External Standard Libraries	dir			auto attached libraries	Library
Dump.interface	file	Doku	23.10.2014	Dump outputs for misc memory data	Library
StringMethods.module	file	Doku	29.11.2014	Standard string functions like "Asc()", "Bin()", "Str(...	Library
External Librarys	dir				Library
Custom	dir				Library
Example	dir				Library
Adc.interface	file	Doku	13.01.2015	Adc-Interface	Library
ClkSys.interface	file	Doku	07.10.2013	Atxmega ClockSys-configuration.	Library
Crc16.interface	file	Doku	16.07.2014	CRC16-CCITT: x^16 + x^12 + x^5 + 1	Library
Crc8.interface	file	Doku	07.10.2013	CRC8 generator polynomial (based on 8051 exam...	Library
DCF77.interface	file	Doku	02.10.2013	DCF77 Decoder	Library
Encodings.module	file	Doku	24.05.2014	Encoding/Decoding Functions (Global)	Library
FastFourier.interface	file	Doku	28.09.2013	Fast Fourier Transformation 64 Points	Library
FT800.interface	file	Doku	01.11.2014	EVE FT800 (Spi) Interface/Library (as of Version 20...	Library
Graphics.object	file	Doku	15.11.2014	Graphical offscreen display object	Library
iGraphics.interface	file	Doku	21.03.2014	Universal Graphics Interface	Library
INTn.interface	file	Doku	05.01.2015	Intn Interface	Library
KeyMgr.interface	file	Doku	20.06.2014	Key-Manager	Library
Lcd4.interface	file	Doku	28.04.2014	LCD Interface (4-Bit) for HD44780 and compatible ...	Library
Math64.module	file	Doku	28.09.2013	Global math64 routines.	Library
NetworkFunctions.module	file	Doku	15.11.2014	Global network functions.	Library
OneWire.interface	file	Doku	27.04.2014	Software emulated 1Wire/OneWire.	Library
PCInt.interface	file	Doku	15.12.2013	PCInt-Interface für moderne AVR µC	Library
Sleep.interface	file	Doku	29.12.2013	Sleep interface	Library
SoftSpi.interface	file	Doku	27.04.2014	Software emulated Spi.	Library

Right-click on a folder or entry to create, double-click to edit.

Close

By double clicking the name you get into the library editor. By clicking on the root element, the start page of the library documentary is shown. The start page also contains a link to the auto-generated short description.


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.

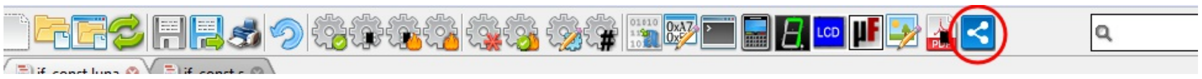
Links to existing pages are shown in a different style from nonexisting ones. Click on "Edit this page" to edit or create a documentary page.

- Syntax of the documentary markup language.

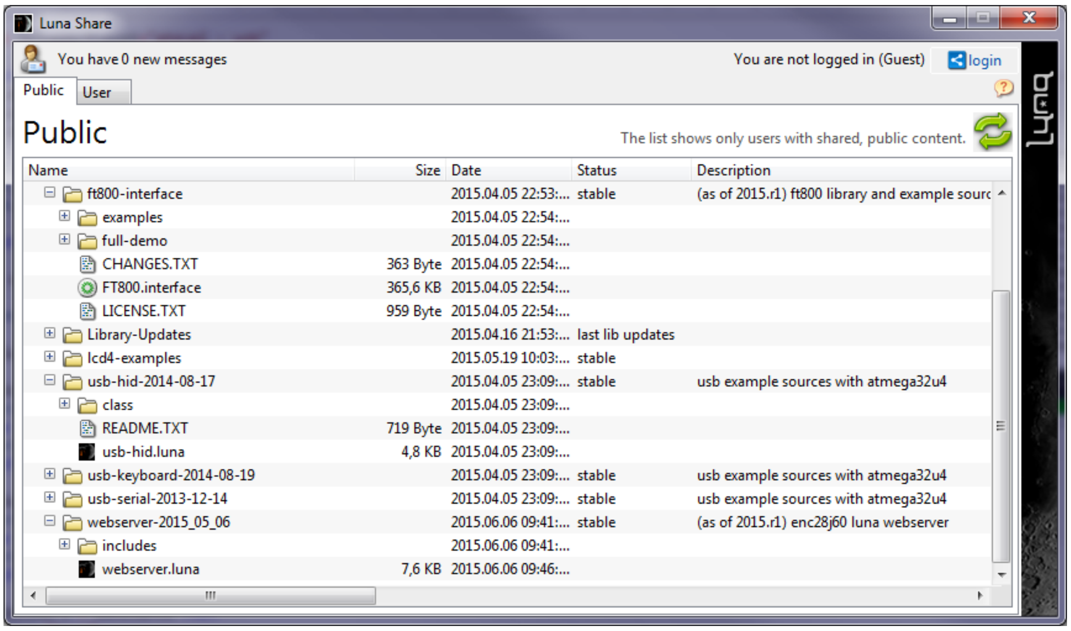
Luna-Share

With Luna-Share you can share or download sources, libraries and other project files. Public files are visible and can be downloaded from all luna users. By creating an account, you can manage your own content. **To download public content you not need an account.**

Luna-Share can be opened via the "Tools" menu, or by clicking the icon in the toolbar:



The main window of Luna Share:



Luna Share

You have 0 new messages

You are not logged in (Guest) login

Public User

Public

The list shows only users with shared, public content.

Name	Size	Date	Status	Description
ft800-interface		2015.04.05 22:53:...	stable	(as of 2015.r1) ft800 library and example sourc
examples		2015.04.05 22:54:...		
full-demo		2015.04.05 22:54:...		
CHANGES.TXT	363 Byte	2015.04.05 22:54:...		
FT800.interface	365,6 KB	2015.04.05 22:54:...		
LICENSE.TXT	959 Byte	2015.04.05 22:54:...		
Library-Updates		2015.04.16 21:53:...	last lib updates	
lcd4-examples		2015.05.19 10:03:...	stable	
usb-hid-2014-08-17		2015.04.05 23:09:...	stable	usb example sources with atmega32u4
class		2015.04.05 23:09:...		
README.TXT	719 Byte	2015.04.05 23:09:...		
usb-hid.luna	4,8 KB	2015.04.05 23:09:...		
usb-keyboard-2014-08-19		2015.04.05 23:09:...	stable	usb example sources with atmega32u4
usb-serial-2013-12-14		2015.04.05 23:09:...	stable	usb example sources with atmega32u4
webserver-2015_05_06		2015.06.06 09:41:...	stable	(as of 2015.r1) enc28j60 luna webserver
includes		2015.06.06 09:41:...		
webserver.luna	7,6 KB	2015.06.06 09:46:...		

By right-clicking on an entry you can download the entire folders or individual files. In the settings to Luna Share (see "Settings" menu) you can define which file types should be opened automatically when you double-click. By double-clicked files are downloaded (temporarily) and displayed via the operating system, for example, Images or text.

Technical

Byte-order

The AVR uses the little endian Byte-order, e.g. the lower byte is located at the lower memory address. A 32 Bit value(4 Bytes) has this order in memory:

Memory ?			
&h00	**&h01**	**&h02**	**&h03**
byte0	byte1	byte2	byte3

IMPORTANT NOTE

As constants in the source code hexadecimal or dual (binary) ***written*** values are always in **Big-endian notation** (they are then better readable for humans). Eg if you wrote the word "MOON" in wrong notation, then the bytes are stored in reverse (wrong) order in memory:

Written: 0x4c4f4f41 (wrong)			
0x4d	0x4f	0x4f	0x4e
"M"	"O"	"O"	"N"
Written: 0x414f4f4e (correct)			
0x4d	0x4f	0x4f	0x4e
"N"	"O"	"O"	"M"
Memory ?			
0x00	**0x01**	**0x02**	**0x03**
0x4e	0x4f	0x4f	0x4d
"M"	"O"	"O"	"N"

The Little-endian Byte-order has some advantages for the AVR controller. Only two zeros have to be added to convert a 2 Byte number into a 4 Byte number, without modifiying the memory address. With Big-endian Byte-order; the memory address must be moved by 2.See also: Byte-order (Wikipedia)

Type Casting

Explicit type convertion of a value expression into a specific Data Type. This is often makes sense when a Function requires a specific data type. Some output functions adapt the display format ti the data type passed.

For example the calculation of a bit manipulation will result in the next larger data type if the original data type possibly may not have enough space to accomodate the result (Data types smaller than Long).

Example 1:

```
dim a,b as byte
print hex(a+b)   ' The result will be a Word,
                 ' the Hex-Output function will therefor display a Word value
```

You could force a type conversion to define the resulting data type.

Type Conversion Functions

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

See also:

- Bcdenc()
- BcdDec()
- Ce16()
- Ce32()

Example 2:

```
dim a,b as byte
print hex(byte(a+b))   ' The result will be of type Byte,
                       ' the Hex-Output function will therefor display a Byte value
```

Variables in Methods

VARIABLEN IN METHODEN

Parameter und lokale Variablen der Methoden werden gegenüber den globalen Variablen in Bezug zur Methode gesetzt. Globale Variablen stehen demgegenüber in Bezug zur Klasse in der sie dimensioniert wurden. Im Hauptprogramm wäre dies die Basisklasse "Avr".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 auch:** dim, static

KONZEPTE

Für temporär existierende Variablen gibt es verschiedene Konzepte, wie man die dafür notwendigen lokalen Speicherbereiche verwaltet. Jedes dieser Konzepte hat entsprechende Vor- und Nachteile.eines der Konzepte ist, zusätzlich zum normalen Stack einen extra-Stack für lokalen Speicher zu verwalten. Dies macht es jedoch notwendig, dass ein programmweit verfügbarer Zeiger mitgeführt werden muss und die Gefahr besteht, dass beide Stacks kollidieren (sich gegenseitig überschreiben). Zusätzlich werden zumeist eine Konfiguration zur Größe des extra-Stack und extra verwaltende (interne) Unterprogramme notwendig.ein weiteres Konzept ist die Nutzung des normalen Programmstack. Auch hier ist ein Zeiger notwendig, jedoch ist eine Kollision verschiedener Stacks nicht möglich. Zudem werden Konfigurationsfehler eines zweiten Stacks vermieden.Luna nutzt letzteres Konzept für Parameter und temporäre, lokale Variablen.

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

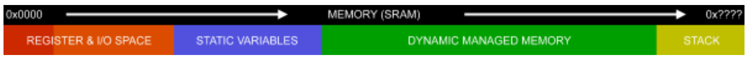
Bei globalen Variablen, sowie bei in der Methode ***statisch*** dimensionierten Variablen entfallen die ersten beiden Schritte, wodurch Zugriffe schneller sind. Bei zeitkritischen Zugriffen, sind also globale oder statische, lokale Variablen vorzuziehen, sofern dies technisch möglich ist.**Verlassen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

Memory Management

MEMORY MANAGEMENT

See Memory Management. The managment routine allows you to dynamically split SRAM into blocks of different sizes. Luna calls such a block Memory Block and is an Object.Memory Management is dynamic and automatically cleans up memory Garbage Collection. Memory management comes in whenever you use strings or string functions or when memory is allocated.Memory management requires at least 64 Byte of available memory.Do not use strings or string functions if this is not the case.

TYPICAL MEMORY ALLOCATION IN LUNA



MEMORY BLOCK STRUCTURE

Every memory block requires a 5 Byte to manage the block (Header).

Name	Value	Typ	Description
Magic	0xAA	byte	ID
Length	Number	word	Number of data byte
VarAddr	Address	word	Address of linked variable, or Nil
	Data		Your data

If memory is very limited then you can manage it yourself. Use direct access commands to the memory object sram.

EXAMPLE

Assuming that you have dimensioned some strings:

```
dim date,time as string
```

When allocating a string or string function, the result will be stored in a memory block and its starting address (Pointer) is returned to the respective variable. The Pointer points to the data starting address and not the block address to avoid havint to take the header size into account. This would be inefficient.Lets also assume that you RAM starts at Address 100 (decimal). The variables *date* and *time* are 16 Bit pointer and require 2 Byte each:

Memory ->			
100	**101**	**102**	**103**
Variable *date*		Variable *time*	
0 (nil)		0 (nil)	

The available memory starts right behind these variables and ends at stack end. [1]. As long as you have not assigned data to the string, it will point to *nil*.Allocating data to the string variable will create a memory block in your RAM:

```
dim date,time as string
date="13.11.1981"
```

The memory now looks like this:

Memory ->																			
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 *date*		Variable *tme*		Header Memory Block					Data Memory Block										

As you can see *zeit* is not allocated, *date* now has the starting address of your data in the memory block. The string data is stored as a Pascal-String to allow string functions to read the string length. This is important because only the background memory management may access the memory block's header data.You will also notice that only the variable's address is stored in the memory block header. This is important because when the memory block is released, (defragmentation/compression), all addresses of the following blocks will change (they will move up). Memory management will now adjust all addresses of variables that have a pointer to a memory block.

PRO AND CON

Without memory management, the programmer himself must carefully look after his memeory allocation. This would also include the disadvantage that all strings would have to have a fixed length (statis Strings). If a string mit sometime grow upto 60 Byte, then you would have to reserve 60 Byte, even if most of the time the string might only need 10 Byte. This is advantageous if you only have a few short string operations and you have limited memory available.Memory management ist advantageous is more efficient and even saves memory inspite of the header dta required if you have to manage many strings an blocks. It is also easier to allocate dynamic data. If memory is limited (less than 128 Byte), it will become inefficient due to the header data. Luna tries to combine both by allowing you to use Strukturen to allocate static strings or data blocks. If you limit your usage to in- / output and avoid string functions like Left(), Right() or Mid() etc., or define your own memory structure, both types of memory management will be combined.

1) The stack grows from the RAM end towards RAM start. The definition avr.Stack defines the number of bytes reservedfor the stack

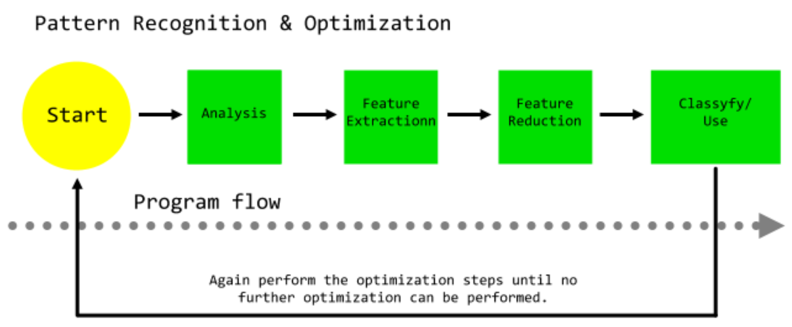
Optimizer

OPTIMIZER

The Luna Compiler/Assembler has many different stages of translation / optimization when compiling your source code. The machine code will be optimized for speed and size while compiled.

The optimization stages include a analysis of memory block access, jumps and mathematical or logicaloperations. Jumps and calls will be done by faster and shorter direct adrressing instead of realtive addresses . Comparisons will be restructured to increase speed. Calculations are soolved and reduced as fas as possible.

A other optimization stage is the pattern recognition. It will analyze data in registers. It recongnizes patterns and will analyze the logic order. This will avoid loading a pointer again that is alredy loaded. It will also recognize if a register value has been modified by the program sequence.


Pattern Recognition & Optimization
Start
Analysis
Feature Extractionn
Feature Reduction
Classyfy/ Use
Program flow
Again perform the optimization steps until no further optimization can be performed.

The optimization levels are turned on by default. You can using compiler command-line option to disable the optimization completely or using pragma to deactivate it for parts.

Luna Picture

LUNA PICTURE (*.LP)

Luna uses a proprietary picture format which the IDE supports directly. Its simple structure is ideal for microcontroller applications.You may also integrate any other picture format and decode it yourself. The BMP format also supports RLe-Compression und up to 24 Bit color depth. "png" is also suitable although decoding is more complicated.Contrary to the popular microcontroller "BGF"-Format, "LP" will not lead to decoding errors when some specific byte sequences are encountered. "LP" also has an "end" marker.

FORMAT DESCRIPTION

The Luna-Picture-Format is made up of a header, the RLe packed data end an end marker.

Offset	Description	Name
0x00	Width in Pixel	Width
0x01	Height in Pixel	Height
0x02	Bit Depth(1)	Depth
0x03	Reserved (0)	reserved
0x04..n	RLe-Compressed Picture Data	Data
0x77:0x00	End marking (2-Byte-Token)	endMark

tHE pICTURE dATA: first Pixel is at top left (x=0,y=0)

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

The RLe-Coding

Read a byte and check if it is **0x77**, the (Magic-Byte). This preceeds a 3 byte token: ***0x77***:*count*:*value*, Example: "0x77 0xFF 0x13" represents 255 times the value 0x13. Tha tis all there is to it, the simplicity is ideal for processing on micro controller.**Example of a decoding routine**

```
' Initialization
avr.device = atmega168
avr.clock = 20000000         ' Quartz frequency
avr.stack = 32               ' Bytes Program stack (Default: 16)
uart.baud = 19200            ' Baudrate
uart.Recv.enable             ' Send active
uart.Send.enable             ' Receive aktive
' Main Program
SendPicture(Picture.Addr) ' Decode picture and send to serial port
do
loop

Procedure SendPicture(imgAddr as word)
    dim i,a,cc as byte
    dim c,w,h,size as word
    ' RLe compressed luna lcd picture
    ' format description
    ' -- header (4 byte) ------------------
    ' byte   width in pixels
    ' byte   height in pixels
    ' byte   depth (1,8)
    ' byte   reserved (0)
    ' -- data (x bytes) --------------------
    ' First pixel at top left 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\mypicture.lp"
```

Luna Font

LUNA FONT FORMAT (*.LF)

the Font-Format is a propritatary format for microcontroller applications.The IDE font editor directly supports this format and makes creating your own font easy, mit dem sich problemlos eigens erstellte Schriftarten in den eigenen Mikrocontroller-Anwendungen verwenden lassen.

FORMAT DESCRIPTION

Offset	Description	Name
0x00	Lines per character	RowsPerChar
0x01	Byte per line	BytesPerRow
0x02	Total number of byte per character	BytesPerChar
0x03	extended Modes (See Table)	Mode
0x04..n	Character data >= ASCII 32	Data

First character pixel is at top left. This simple equation gives you access to the character data (char = Ascii-value of the character):

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

Bit 0 in extended Mode is set when the short character set is used, ignoring characters above ASCII 127. If you use your own decoding, make sure to test for this.Every byte of a character row codes 8 vertical pixels.

Extended Modes		
Bit	**Description**	**Default**
0	Short character set (ASCII 32 to 127)	0
1-7	Reserved	0

Compiler Parameters

COMPILER/ASSEMBLER LAVRC

The compiler/assembler "lavrc" is a command line program. The program includes the Luna-compiler *lavrc* and the Luna-assembler *lavra*. The Luna-assembler assembles the compiled code including inline-assembler of the Luna source code.The names of the executable for different operating systems are:

Operation System	Filename
Windows	lavrc.exe
Linux	lavrc

COMMAND LINE PARAMETERS AND OPTIONS

- **-i** *dir* - add the directory 'dir' as include-path.
- **-v** - enables the text output (verbosity).
- **-c** - Check syntax only (Parser), no assembling.
- **-z***[switch]* - Assembler code-optimizations:
  - **0** - disabled
  - **1** - enabled (default)
- **-o***[type]* - Output options of the compiler:
  - **b** - write binaery file (*.bin)
  - **e** - write binaery eeprom file (*.eep)
  - **h** - write intel hex file (*.hex)
  - **y** - write precode of the precompiler (*.zc)
  - **z** - write precode of the preassembler (*.za)
  - **a** - write assembler output (*.s)
  - **s** - write assembler output, incl. library code (*.s)
  - **r** - write report file (*.rpt)
- **-w***[level]* - Warning level:
  - **0** - Disable warnings.
  - **1** - Only memory space warnings
  - **2** - All warnings (default)
- **-k** - export Luna-keywords to 'keywords.txt'
- **-h** - This help.

EXAMPLE CALL

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"
```

Base Command Set

ALPHABETICAL OVERVIEW

#

- #Define - Defines/Aliases.
- #Undef - Defines/Aliase entfernen.
- #ide - IDE commands in the source code.
- #if #elseif #else #endif - Conditional compiling of source code.
- #select #case #default #endselect - Conditional compiling of source code.
- #macro #endmacro - Luna-code macros
- #Include - Include source code files.
- #IncludeData - Include binary data files into flash.
- #Library - Import a external library.
- #pragma, #pragmaSave, #pragmaRestore, #pragmaDefault - Control compiler-internal processes.
- #cdecl, #odecl, #idecl - Parameter definition for indirect call.
- #error - Show error message (interrupt compile process).
- #warning - Show warning message.
- #message - Show info message.

A

- Abs() - Absolute Value.
- Add - Integer Addition (Command-Syntax).
- Asc() - Returns the numerical value of characters, example asc(" ") = 32.
- Asr - Arithmetic Bit shift right.
- Asm-EndAsm - Inline Assembler.
- Avr - Root class AVR-Controller.

B

- Break - Exit a loop.
- BcdEnc() - BCD Coding.
- BcdDec() - BCD Decoding.
- Bin() - Display a value or string in binary format.
- Byte() - Convert data types.
- byte1() - 1. Byte of a 16 or 32-bit-Value
- byte2() - 2. Byte of a 16 or 32-bit-Value
- byte3() - 3. Byte of a 32-bit-Value
- byte4() - 4. Byte of a 32-bit-Value
- ByVal, byRef - (Key word)
- ByteToInt16 - Convert numeric sign.
- ByteToInt32 - Convert numeric sign.

C

- Call - Call a Methode or Inline Assembler Subroutine.
- Ce16() - Convert between LittleEndian/BigEndian.
- Ce32() - Convert between LittleEndian/BigEndian.
- Chr() - Convert numeric value to ASCII.
- Cli - Disable global Interrupts.
- Class-EndClass - User defined class.
- Continue - Continue loop with next iteration.
- CountFields() - Determine number of elements in a separated string.
- Clr/Clear - reset a Variable to Zero/Nil.
- Cos() - Integer Cosine.
- Const - Define a constant.

D

- .db - Start of Data Block (Byte).
- .dw - Start of Data Block (Word).
- .dl - Start of Data Block (Long).
- Data-EndData - Define dta object (Flash).
- Declaration - (Definition)
- Descriptor() - Current assembling position.
- Decr - Fast decrement.
- Defined() - Check existance of a constant or symbol.
- Dim - Define a variable in RAM.
- Div - Integer Division (Command-Syntax).
- Do-Loop - Loop with optional exit criteria.
- eeDim - Define variable in Eeprom

E

- Eeprom-EndEeprom - Create Data Object (Eeprom).
- Exception-EndException - Debug Function.
- Exception - (Definition)
- Exit - Exit a loop.
- Even() - Check if Integer value is even.
- Event-EndEvent - Create Event.

F

- Fabs() - Floating - Find absolute value.
- Facos() Floating - Calculate ArcusCosine (Radiant)
- Fadd() Floating - Addition (Command-Syntax/Function)
- Fdiv() Floating - Division (Command-Syntax/Function)
- Fmul() Floating - Multiplication (Command-Syntax/Function)
- Fsub() Floating - Subtraction (Command-Syntax/Function)
- Fasin() Floating - Calculate ArcusSinus (Radiant)
- Fatan() Floating - Calculate ArcusTangens (Radiant)
- Fcbrt() Floating - Calculate Cubic Root
- Fceil() Floating - Round up to whole number
- Fcosh() Floating - Calculate Hyperbolic Cosine (Radiant)
- Fcos() Floating - Calculate Cosine (Radiant)
- Fdeg() Floating - Convert Radiant to Degrees
- Feven() Floating - Check if even.
- Fexp() Floating - Calculate Exponential value
- Fix() Floating - Convert to whole number.
- Flexp() Floating - Inverse of Frexp
- Flog10() Floating - Calculate Logarithm to Basis 10
- Flog() Floating - Calculate Natural Logarithm
- Floor() Floating - Round down to next whole number, depending on sign
- Fodd() Floating - Check if odd.
- Format() - Format Decimal String.
- For-Next - Loop with Counter.
- Fpow() Floating - Calculate x to power of y
- Frad() Floating - Convert from Degrees to Radiant
- Frac() Floating - Extract Fractional part of a number
- Frexp() v - Convert Value to Mantissa and Exponent
- Fround() Floating - Arithmetic Round Down
- Fsine() Floating - 360° Wave Function, Angle to Degrees
- Fsinh() Floating - Calculate Hyperbolic Sinus
- Fsin() Floating - Calculate Sinus (Radiant)
- Fsplit() Floating - Split Value into whole and rational part
- Fsqrt() Floating - Calculate Square Root
- Fsquare() Floating - Calculate Square
- Ftanh() Floating - Calculate Hyperbolic Tangent
- Ftan() Floating - Calculate Tangent (Radiant)
- Ftrunc() Floating - Round dowwn to next whole number.
- Function-EndFunc - Create a Function with return value.
- Fval() - Convert a String with a decimal floating value to a binary value.

G

- GarbageCollection() - Manual GarbageCollection.

H

- Halt() - Permanent empty loop. Same as Do ... Loop.
- Hex() - Convert value to a hexadecimal number (String).
- HexVal() - Converts hexadecimal number (String) to a integer value.

I

- Icall - Indirect Function Call via Pointer.
- Idle-EndIdle - Create Idle Event.
- If-ElseIf-Else-Endif - Condition Branching.
- Incr - Fast Increment.
- Instr() - Find Text in Text (String).
- Integer() - Convert Data Types.
- Isr-EndIsr - Create Interrupt Service Routine.

J

- Jump - Jump to Label or Address.

L

- Label - (Expression)
- Left() - Returns left part of string (String).
- Len() - Returns length of string (String).
- Lower() - Convert Text to lower case (String).
- Long() - Type convert a value to a Long Integer. (32 Bit)

M

- Max() - Find largest value in a list of values.
- Median16u() - Calculate Median Value.
- Median16s() - Calculate Signed Median Value.
- MemCmp() - Compare RAM Memory.
- MemCpy() - Copy Memory within RAM (schnell).
- MemSort() - Sort all Bytes of Memory in ascending order.
- MemRev() - Invert Bytes of Memory.
- MemoryBlock - Memory Block Object.
- Mid() - Read part of a Text (String).
- Min() - Find smallest value in a list of values.
- Mod - Modulo-Operator
- Mul - Integer Multiplication (Command-Syntax).
- Mkb() - Convert Byte Value to String.
- Mkw() - Convert Word Value to String.
- Mkt() - Convert Uint24 Value to String.
- Mkl() - Convert Long Value to String.
- Mks() - Convert Single Value to String.

N

- Nop - Empty command.
- NthField() - Find a field in an element separated String (String).

O

- Odd() - Check if Integer is odd.

- Operators - List of Operators

P

- Part-EndPart - Highlight Code in the IDE.
- Pascal-String - (Definition)
- Pointer - Pointer (Sram, Flash and Eeprom).
- Preprocessor - Preprocessor (Definition)
- Print - Universal Output to a serial port.
- Procedure-EndProc - Create Method.
- Push8()/PushByte() - Push 8-Bit-Value onto the Stack.
- Push16() - Push 16-Bit-Value onto the Stack.
- Push24() - Push 24-Bit-Value onto the Stack.
- Push32() - Push 32-Bit-Value onto the Stack.
- Pop8()/PopByte() - Pop 8-Bit-Value from the Stack.
- Pop16() - Pop 16-Bit-Value from the Stack.
- Pop24() - Pop 24-Bit-Value from the Stack.
- Pop32() - Pop 32-Bit-Value from the Stack.

R

- Replace() - Replaces the first occurrence of a string with another string.
- ReplaceAll() - Replaces all occurrences of a string with another string.
- Reset - Restart Controller.
- Return - Return Command from a Method.
- Right() - Read right part of a Text (String).
- Rinstr() - Find a text within a text (String).
- Rnd() - Create pseudo random number 8 bit.
- Rnd16() - Create pseudo random number 16 bit.
- Rnd32() - Create pseudo random number 32 bit.
- Rol - Shift bitweise to left.
- Ror - Shift bitweise to right.

S

- Seed - Initiate random number generator 8 bit.
- Seed32 - Initiate random number generator 32 bit.
- Select-Case-Default-EndSelect - Fast Case Distinction.
- Sei - Enable global interrupts.
- Shl - Shift byte to left.
- Shr - Shift byte to right.
- Sin() - Integer Sinus.
- Sine() - Fast Integer Sinewave function.
- Single() - Convert data type to single.
- Spc() - Create space characters (String).
- Sram - RAM as a data object.
- Static - Keyword when Dimensioning a variable.
- Struct-EndStruct - User defined struct declaration.
- Str() - Convert binary value to a decimal value (String).
- StrFill() - Fill a Text with another Text (String).
- String - (Definition)
- SoftUart - SoftUart Interface.
- Sub - Integer subtraction (Command-Syntax).
- Sqrt() - Integer square root.
- Swap - Swap variables or values.

T

- Trim() - Clip leading or trailing non printable characters (String).

U

- Upper() - Convert Text to captial letters (String).

V

- Val() - Convert a decimal number string to a Integer value.
- Void - Keyword for a Function call.
- Void() - Dummy method.

W

- Wait - Delay Function (Seconds).
- Waitms - Delay Function (Millieconds).
- Waitus - Delay Function (Microseconds).
- When-Do - Conditional exection for one line only.
- While-Wend - Conditional exection (Loop).
- Word() - Convert a value to Word Type.
- WordToInt16 - Convert a Word value to an 16 Bit Integer.
- WordToInt32 - Convert a Word value to an 32 Bit Integer.

#Define

#Define links commands or expressions similar to an **Alias** with a Identifier. The Source code can use this identifier as if it were an Expression. The compiler will replace the place holder with the corresponding expression.

#Define improves readability and adaptability of the programm or parts thereof. If you want to modify Ports for a different project; then just modify the ports only and not the whole source.

DEFINE FUNCTIONS

You can define **Define-Functions**. The define function is a *virtual* function instead of a simple expression. The parameters of this virtual function are text-replaced in *Expression* on the right. See Example 2 and 3.

Notes

- Defines are always global!
- Already created Defines can be redefined like Constants. The old definition are then replaced by the new definition.

SYNTAX

- ***#Define*** *Identifier* ***as*** *Expression*

See also: Compiler Sequence

EXAMPLE 1

```
avr.device = attiny85
avr.clock = 8000000
avr.stack = 2
#define Button as Portb.4
dim a as byte
Button.mode = Output, pulldown
do
  if Button.debounceHigh then
    ' put your key pressed code here
    do
      waitms 100
    loop until Button.debounceLow
  end if
loop
```

EXAMPLE 2

```
#define BV(n) as (1 << (n))

a and= BV(3)  'wird zu a and= (1 << (3))
```

EXAMPLE 3

As of 2013.r6: Define-Functions

```
avr.device = attiny85
avr.clock = 8000000
avr.stack = 2

#define myfunc(a,b) as ((a + b) * a)

dim var1,var2,result as byte

result = myfunc(var1,var2)  '"myfunc(var1,var2)" would ((var1 + var2) * var1)

do
loop
```

#Undef

Implemented as of Version	2015.r2

#Undef removes a defined #Define. A unknown define throws a warning.

SYNTAX

- ***#Undef*** *Identifier*

EXAMPLE

```
#define Taster as Portb.4

Taster.mode = output, low

#undef Taster
Taster.mode = output, low  'Error, "Taster" is here unknown/undefined
```

#ide

#ide initiates an IDE command. IDE instructions are ignored by the compiler.

Syntax: ***#ide***

COMMAND

UploaderPreset="*Presetname*"	Instructions for selecting a preset from the uploader/programmer. The text is the name of the preset in the selection.

Preprocessor-Conditions

Used to process the assembler source conditionally. Only expression with constants are allowed.

.IF .ELSEIF .ELSE .ENDIF

- ***.if*** *Expression*
  - Assembler code
- ***.elseif*** *Expression*
  - Assembler code
- ***.else***
  - Assembler code
- ***.endif***

.SELECT .CASE .DEFAULT .ENDSELECT

- ***.select*** *Ausdruck*
- ***.case*** *Expression [, Expression1 , Expression2* ***to*** *Expression3, ..]*
  - Assemblercode
- ***.default***
  - Assemblercode
- ***.endselect***

See also: defined(), Compiler Sequence

Preprocessor - Macros (Luna)

Implemented as of Version	2015.r1

Declaration

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

Parameter access in the macro

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

EXAMPLE

This declares a macro that excepts 2 parameters.

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

Use in Luna-Source:

```

dim a,b as word

muladd(a,b)
```

#Include

*(Note: Command without the leading "#" command feore Versionen 2012.r7.1, Syntax modified.)*Include a external source code file at this position.**Syntax:**
#Include *"FilePath"*
FilePath: is a constant string with an absolute or relative path to the file. "/" or "\" are permitted in the path name.**Example:**

```
#include "Libraries/NewFunction.luna"
```

#IncludeData

*(Notice: Command prior to Versionen 2012.r7.1 without the leading "#")*Include binary data from a file as a object based data structure into the program segment (Flash).**Syntax:**

- **#IncludeData** Identifier,"File path" *[**at** Addressconstant]*[1)]

The data will be linked to the Identifier and can be managed as a normal Data Object.
Optional address constant siehe Data objekt (Flash)

1) set data to a specified address as of 2015.R1

#pragma, #pragmaSave/Restore/Default

Implemented as of Version	2015.r1

Preprocessor directive to control compiler-internal processes.The pragma directive is processed *textually* like all other directives in the source code.

Syntax

- ***#pragma***
- ***#pragmaSave*** - Save all current pragma values.
- ***#pragmaRestore*** - Restore saved pragma values.
- ***#pragmaDefault*** - Set all pragma values to default.

#pragmaSave/Restore can be stacked. The depth is not restricted. ***#pragmaDefault*** Does not touch the saved pragma values.

COMMANDS

Name	Description	Default Value
DebugBuild	If activated, then the source is compiled as **Debugbuild**, current Code-Line and -File can be read in the source, eg to display additional informations in a Exception.	false
OptimizeCodeEnabled	Activates or deactivates the Code-Optimizing.	true
MethodSetupFast	The methods-initialization is speed optimized and written out directly to the method-code instead of calling a sub-function. Generates more machine code.	false
MethodLocalVarInit	Activates or deactivates the initialization of local, temporary variables to ***0*** or ***nil***. Affects only if ***MethodSetupFast*** is active.	true
MemoryBlockGarbageCollection	Activates or deactivates the automatic garbage collection of the dynamic managed work memory space, used for strings and memoryblocks. If deactivated it is highly important to call the GarbageCollection() function manually to avoid memory overflow!	true
MemoryBlockProcessAtomic	Activates or deactivates the atomic processing of the memory management routines. The atomic processing disables the global interrupts during process of a management routine. This can be disabled if no reentrant memory management is necessary or to avoid jitter of interrupts.	true

#cdecl, #odecl, #idecl

Directive to define a Parameter list for an indirect call of methods using Icall.The directive defines number and type of parameters of parameters passed.Description:

- **#cdecl:** All parameters are passed via the stack.
- **#odecl:** 1. Parameter will be passed via Register block A, all others via the stack **(Luna default)**.
- **#idecl:** 1. and 2. Parameter will be passed via Register block A and B, all others via the stack. However, the first parameter may *not* be an expression. It must be a single variable, address, constant or similar.

Syntax: ***#xdecl*** *Identifier([] , ..)*See Icall for an example.

Icall()

Icall() Makes an indirect call of methods via a function pointer.

Syntax: Icall(*Address*[, *Parameter definition(Parameter1, Parameter2, ..)*])

Use **Parameter definition** to pass parameters to the method.

- **Address** is the method address in memory (Flash).

Note: A indirect call to Luna methods expects ***#odecl***.

See also: #cdecl, #odecl, #idecl

EXAMPLE

```
' ---------------------------------------------
' ICALL example
' Example using a function pointer
' ---------------------------------------------
' 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
'Function prototype declares number and type of paramters to pass
'Meaning:
' #cdecl = All Parameter are passed via stack.
' #odecl = 1. Parameter is passed via Register A, the rest via stack
' #idecl = 1. and 2. Parameter passed via Register A and B respectively,
the rest via stack. The 1. Parameter must be a Variable, address, constant etc.
' Syntax: #cdecl  Identifier( [] , .. )
#odecl byte func_p(byte)
#odecl void proc_p(byRef byte,word)
'        |     |     |     |    |
'        |     |     |     |    +Data Type 2.Parameter
'        |     |     |     +Data Type 1.Parameter
'        |     |     +Type (optional), Reference or value(Default)
'        |     +Identifier of the parameter declaration
'        +Return Type when Procedure "void"
'

LeD1.mode = output
LeD2.mode = output
LeD3.mode = output
dim a,b as byte
dim ptr1,ptr2,ptr3 as word
print 12;
'Function pointer of Method
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

Directive to put a error- warning- or info-message.

Syntax	Note
#error *"message"*	Interrupts the compile process.
#warning *"message"*	
#message *"message"*	Output without line number and origin.

Abs()

Abs() returns the absolute value of the integer passed.**Syntax:** ***integer*** = ***Abs(*** *Integer* ***)***

Example:

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

Add,Sub,Mul,Div

Arithmetical functions in command form with a variable (SRAM) and a Constant (technical no difference to expressions).

Syntax

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

Example:

```
dim a as integer
add a, 1000  ' same like: a += 1000, or a = a + 1000
```

Asc()

Asc() returns the ASCII-Byte value of the first charater passed.

Pre Processor	The function is also available in the Preprocessor, using constants only; it will not result in machine code.

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

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

Shl, Shr, Asr

Shl, Shr: *Logic bit shift* to the left (Shl) or right (Shr). For integer data types, e.g. Byte, word, etc.: Shifting the bits by one position to the left leads to a multiplication by 2, shifting the bits by one position to the right leads to divide by 2. **Asr**: *arithmetical* shift-right. for signed integer data types such as Int8, Int16, integer, etc.: .

SYNTAX:

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

For ***variable***, only a ***integer variable from working memory*** is permitted. Alternatively, the bit shift operators "<<" or ">>" can also be used for unsigned integer variables.

EXAMPLE:

```
dim a as word
a=0b0000000100000000
Shl a, 4  ' shift 4 bits to the left, result: 0b0001000000000000
```

See also: Add, Sub, Mul, Div, Bitshift-Operators "<<" bzw. ">>"

Avr

Avr is the root class (the Controller). **Die Property "Device" must be declared *before* all others.**

Property (write only)		
Name	**Description**	**Value**
Avr.Device=Controllername	selects controller class [1]	Name
Avr.Clock=Constant	Clock rate in Hz [2]	> 0
Avr.Stack=Constant	Stack size in Byte.	> 1
Avr.Isr.=*Isr-Label*	Define the Interrupt service routine.[3]	Address
Avr.CodeStartAddr=Address	Start address of the code, e.g. for Bootloader "avr.CodeStartAddr = LARGEBOOTSTART". This will write the machine code to the specifies address in memory.	Address (Word oriented)
Property (read only)		
Name	**Description**	**Type**
Result = Avr.StackPointer	Current stack pointer position	word
Property (read/write)		
Name	**Description**	**Type**
Avr.	Direct access to all ports.	byte
Avr..	Direct access to the bits of the specified port.	byte
Method		
Name	**Description**	
Avr.Interrupts.enable	Enable global interrupts *ON*	
Avr.Interrupts.disable	Disable global interrupts *OFF*	
Avr.Idle	will call Idle-Event if available.	

EXAMPLE

```
avr.device = atmega32 // Atmega32
avr.clock = 8000000   // 8 Mhz clock rate
avr.stack = 32        // 32 Byte program stack
// Program code
```

Stack-Size

The required stack size depends on the number of parameters and local variables and cascading level of method calls. Cascaded calls add their number of required byte. A method requires at least 2 byte (Return address). Parametera and declared variables come on top.

Controller-Ports and -Constants

Besides using ojects and functions, you may also access the ports **directly** (same as in C/Assembler). All controller specific Bit names or properties of the specific port are defined. Luna-IDE will highlight correctly written and existent names. Ports and names of the **avr.device**-Properties will be updated when loading or saving the source code.

Usage

Generally speaking:

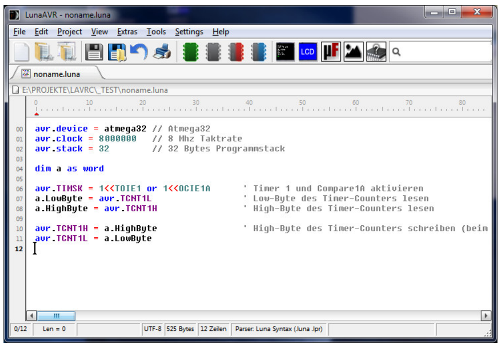
- *Without* specifying the class name "Avr", all ports and constants defined by the manufacturer will be understood as *constants*
- *When* specifying the class name "Avr", all ports and constants will be interpreted as variables (read and write). Controller constants will be interpreted as normal constants. See the product datasheet or IDE to know what are ports or constants.

SYNTAX

```
value = avr.SREG         'default data type is byte
avr.TCNT1.Word = value   'define word access (write)
value = avr.TCNT1.Word   '(read)
```

See also: Ermitteln_von_controllerspezifischen_werten_und_konstanten

Example



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

dim a as word

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
avr.TCNT1L = a.LowByte
```

0/12 | Len = 0 | UTF-8 | 525 Bytes | 12 Zeilen | Parser: Luna Syntax (.luna .lpr)

```
avr.device = atmega32 // Atmega32
avr.clock = 8000000   // 8 Mhz clock rate
avr.stack = 32        // 32 Byte program stack

dim a as word
dim b as byte

avr.TIMSK = 1<<TOIE1 or 1<<OCIE1A       ' Activate Timer 1 and Compare1A
a.LowByte = avr.TCNT1L                  ' Read Low Byte of timer counter
a.HighByte = avr.TCNT1H                 ' Read High Byte of timer counter

avr.TCNT1H = a.HighByte                 ' Write High-Byte of timer counter (when writing, always High byte first!)
avr.TCNT1L = a.LowByte                  ' Write Low-Byte of timer counter
avr.DDRA.PINA0 = 1
b = avr.PORTA.PINA0
avr.Isr.ICP1Addr = myIsr

do
loop

isr myIsr
 [..]
endisr
```

1) Avr.Device can also be read and will return as string constant

2) Setting this property does not modify the actual clock rate, instead it defines the value for subsequential read functions

3) ***InterrupAdr*** is the hardware interrupt address in the controller's interrupt vector, e.g. ***ICP1Addr***.

Exit/Break

Command to exit a loop immediately, independet of the normal condition.

Example1:

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

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

Example2:

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

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

Example3:

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

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

Bcdenc()

Bcdenc() converts a 16 Bit Integer value into a packed 16 Bit BCD valu (4 Decimals) resp. an 8 Bit Integer value into an 8 Bit BCD value (2 Decimals).

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

Example:

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

See also: BcdDec(), BCD Value Construct

BcdDec()

BcdDec() converts a packed 16 Bit BCD value (4 Decimals) into a 16 Bit Integer value (word) resp. a packed 8 Bit BCD value (2 Decimals) into an 8 Bit Integer value (byte).

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

Example:

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

See also: Bcdenc(), BCD Value Construct

Bin()

Converts a number into a binary formatted string. Converts automatically into the data type dependent bit length. Same as in hexadecimal, the standard output has the least significant bit on the right with ascendig significance to the left (Big-endian).

Preprocessor	The function is also available in the Preprocessor, using constants only will not result in any machine code.

Syntax: *String* = ***Bin***(*Expression*)**Example:**

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

byte1()..byte4()

Präprozessor	Ausschließlich eine Funktion des Preprocessor

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, Luna-pp**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 and byRef are keywords in the declaration of Meths .

byVal

The following parameter is to be passed to the method as a copy (Default). "byVal" is therefore obsolete, because parameters are passed as a copy by default. **Copy** meaning the the source variable is left untouched.

byRef

The following parameter is to be passed to the method by reference. **Reference** meaning that a address pointer to the source object or variable is passed as a parameter and the method can modify the source variable. Expressions can't be passed to the method because they are neither objects, variables or data structures.

Recurrent Declarationen in Classes

When using Classes, you may declare items with the same name again in or out of a Structure because they occupy separate namespaces.However, if you reference a method within a class with **byRef**, the compiler will not be able to distinguish the name structure of the object and will report a collision. In such a case use the *Structur declaration* of a referenced structure in the main program only.

Pass a Parameter as a Copy

Example 1:

```
' Main program
dim myvalue as integer
dim s as string
Display("My Number: ",33) ' Call Subroutine
' Alternative call
Display = 12345
s = "Different Number: "
call Display(s,myvalue)
do
loop

' Subroutine
procedure Display(text as string, a as byte)
  dim x as integer  ' local variable
  x=a*100
  print text+str(x)
endproc
```

Pass a Parameter by Reference

Example 1: Variables

```
dim a as byte
a=23
test(a)
print "a = ";str(a)   ' Output: a = 42
do
loop
procedure test(byRef c as byte)
  print "c = ";str(c)  ' Output: c = 23
  c = 42
endproc
```

Example 2: Object Structure

```
struct point
  byte x
  byte y
endstruct
dim p as point
p.x=23
test(p)
print "p.x = ";str(p.x)   ' Output: p.x = 42
do
loop

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

Beispiel 3: Constant Object

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

Example 4: Memory Object

```
dim a(127),i as byte
for i=0 to a.Ubound
  a(i)=i
next
test(a())
do
loop

' Direct access to external array
procedure test(byRef c as sram)
  dim i as byte
  for i=0 to 63
    print str(c.ByteValue(i))
  next
endproc
```

Convert Numeric Sign

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

Converts the source value into a signed one. This function is reuired if you want to pass on a unsigned variable or return value. Example is converting a Byte function result (0-255) that you want to use as a signed value (-127... +128). ***Byte*** is a unsigned data type, so all subsequent calculations would also be unsigned.

SYNTAX

- *integer* = **ByteToInt16**(*value as byte*)
- *integer* = **WordToInt16**(*value as word*)
- *longint* = **ByteToInt32**(*value as byte*)
- *longint* = **WordToInt32**(*value as word*)

Example:

```
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)                'Output: "255"
print str(b)                'Output: "65535"
print str(ByteToInt16(a))   'Output: "-1"
print str(WordToInt16(b))   'Output: "-1"
```

Call

Calls a Subroutine, Procedure or Inline-Assembler-Label.**Syntax:**

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

Procedures do not have to be called using **Call** as this is the default method. This will also work:

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

See also: Void

Ce16(), Ce32()

Ce16() converts a 16 Bit Integer, Ce32() a 32 Bit Integer, from Littleendian to Bigendian and vice versa. The byte order is reversed. Same function as the object method **.Reverse** for data type Integer, Word and Long (see Typkonvertierung).**Syntax:** *word* = ***Ce16****(a as word)*
Syntax: *long* = ***Ce32****(a as long)*

Example:

```
dim a as word
dim b as long
a=0x1122
b=0x11223344
a = Ce16(a)     ' Output: 0x2211
b = Ce32(b)     ' Output: 0x44332211
```

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

The function create a numeric value in Little-endian-Format[1] from a string with numeric content.

Preprocessor	The function is also available in the Praeprozessor, using constants only will not result in machine code.

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***)

Notice:
chr() und *mkb()* are the same function in different notation.*mks()* creates a string in the Standard IEEE, e.g. any value passed to the function will be converted to a Single and returned as a string.

Example:

```
[..]
dim a as word
dim b as long
dim s as string
a=1234
b=4567
s = mkw(a)   ' Word as a String
s = mkl(b)   ' Long as a String
s = mkb(65)  ' Byte as a String, same as chr(65)
s = mks(a)   ' "a" ==> Single ==> String
' Dump of String content
print "s = ";
for i=1 to len(s)
  print "0x";hex(s.ByteValue(i));" ";
next
print
[..]
```

[1] Avr-Controller standard formatting.

Cli, Sei

Cli und *Sei* are the short term for Avr.Interrupts.enable and Avr.Interrupts.Disable.

- ***Cli*** disables the global interrupts
- ***Sei*** enables the global interrupts

User defined Classes

LunaAVR lets you define your own classes, as a module or library.Such a class can be implemented in the main program. Such a class is self contained. The components are accesssible only via the class name (different namespace: "ClassName.xxx"). The different Namespace allows to use names for the components that also exists in the main program.

A class may contain these *components*:

- Constant
- Variable (RAM and eeprom)
- Structure
- Procedure
- Funktion
- Interrupt-Service Routine
- Constant-Object
- eeprom-Object

The individual components are declared within the class similar to the main program and then be a *part of the class*.

Syntax

- ***Class** Identifier*
  - *Constants declaration*
  - *Definitions/Aliases*
  - *Structure Declarations*
  - *Variable dimensioning*
  - *Interrupt-Service Routines*
  - *Functions and Procedures*
  - *Data structures*
- ***endClass***

Using classes in the program code

You can acces all components of a class from your program code:

- Structure Declarations (for dimensioning)
- Constantes (read)
- Variables (read and write)
- Procedures (call)
- Funktions (call)
- (Data-) Structures (read and write)

Example

```
' Initialization
[..]
dim var as single
'Somewhere in your program code
print test.Version        ' Output
test.Init()               ' Call Procedure
var = test.GetData(0,2)   ' Call Funktion
test.a=123                ' Assign value
print str(test.a)         ' Read value and output

' Main Loop
do
loop

' Declaration
Class test
  const Version = "1.0"
  dim a as byte
  Procedure Init()
    print "Initialization"
    print "a = "+str(a)
  endProc

  Function GetData(x as byte, y as byte) as single
    mul y,4
    y=y+x
    return table.SingleValue(y)
  endFunc

  data table
```

```
    .dw &h1234,&h5678,&h9aab
    .dw &h1234,&h5678,&h9aab
    .dw &h1234,&h5678,&h9aab
    .dw &h1234,&h5678,&h9aab
  enddata
endClass
```

Continue

Continue a loop with the next iteration. All commands in the loop behind **continue** will be skipped. The program will jump to the loop condition command.

Example:

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

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

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

CountFields()

Determine number of elements in a separated string.

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

- **source:** The string in which to search.
- **separator:** String which separates the elements to each other.

See also: NthField()

EXAMPLE

```
dim a as string
a = "This | is | a | example!"
print str(CountFields(a,"|"))  ' Result: 4
```

Clr, Clear

Fast clearing (set to 0) of a variable, array, structure or structure property.Beware of Data Type Memory Block! Here you should use the Release method of that object instead of erasing the reference, otherwise the referenced memory will be inaccessible.**Syntax:**

1. ***Clr*** *Variable* - Set Variable value to zero
2. ***Clr*** *Variable()* - Set all element values of the array to zero

Info

Variable includes declared structures. In this case, all values of that structure will be set to zero.You may also set individual properties of a structure to zero, array elements or a whole array of a structure.

Example1:

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

clr a(4)  ' set 5th element of array "a" to 0
clr a()   ' set whole array "a" to 0
clr b     ' set b to 0
clr s     ' set string to ""
```

Example2:

```
' declare structure
struct date
  byte hour
  byte minute
  byte second
  byte wert(5)
endstruct
' declare such a structure
dim d as date
clr d.hour     ' set hour to 0
clr d          ' set all values to 0
clr d.value(3) ' set 4th element of array "value" to 0
clr d.value()  ' set whole array "value" to 0
```

Cos()

Cos() is a fast 16 Bit Integer Cosine function. You must pass an angle in degrees multiplied by 0. The result is a 16 Bit Integer in the range of -32768 to +32767, equivalent to Cosine -1 to +1. The function utilizes the Cordic Algorithm .**Syntax:** *integer* = ***Cos****(a as integer)*

Example:

```
dim angle as byte
dim result as integer
angle=23
result = Cos(angle*10)
```

.db

Initiates a Byte-oriented data block. May be used in data structures only.

Example 1, Data Structure in Flash:

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

Example 2, Data Structure in eeprom:

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

.dw

Initiates a Word-oriented data block. May be used in data structures only.

Example 1, Data structure in Flash:

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

Example 2, Data structure in eeprom:

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

.dl

Initiates a long-oriented data block. May be used in data structures only.

Example 1, Data structure in Flash:

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

Example 2, Data structure in eeprom:

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

Data-endData

Defines a object data structure in the program segment (Flash).**Syntax:**

- ***Data*** *Identifier [**at** Address constant]*[1)]
  - *Data*
- ***endData***

Method (read only)		
Name	**Description**	**Return Type**
.ByteValue(offset)	Read Byte	byte
.WordValue(offset)	Read Word	word
.IntegerValue(offset)	Read Integer	integer
.LongValue(offset)	Read Long	long
.SingleValue(offset)	Read Single	single
.StringValue(offset,bytes)	Read String of explicit length	string
.PString(offset)	Read Pascal String (Start-Byte is the length)	string
.CString(offset)	Read C-String (Null terminated)	string
.Addr	Data Structure Addresse in Flash	word
.SizeOf	Read Byte size of object	byte

* ***offset:*** Zero based Byte-Position within the structure

* ***bytes:*** Number of Byte to read

Note:

* When accessing Word (16-Bit) Data, then Word_offset = byte_offset * sizeof(Word), e.g. word_offset = byte_offset * 2. * Out of bounds access is possible and is not checked.

DATA TO FIXED ADDRESS

Setting the data from the specified **word address** in the flash. A false or erroneous address results in an assembler error message. Take care that we have here is a **word address**, such as the bootloader address *FIRSTBOOTSTART*. The byte address would be *(word address * 2)*.

With this functionality, it is possible to store data to a specific, fixed position in the program memory (Flash). If the compiled program is greater than or equal to this address it will overwrite the data. You have to check that is enough free space available.

DATA STORAGE

The directives for the storage of data within a data object.

- .db - (byte) 8 Bit values incl. strings
- .dw - (word) 16 Bit values
- .dt - (triple) 24 Bit values
- .dl - (long) 32 Bit values

SUPERIMPOSING

Strukture Declaration simplifies access. **See also:** Pointer, Chapter "SuperImpose"

Example

```
' Initialization
[..]
' Main program
dim a as byte
dim s as string
a=table1.ByteValue(4)         ' Read Byte from table1 + 4, Result: 5
s=table1.CString(11)          ' Read C-String, Result: "Hallo"
s=tabelle1.StringValue(11,2)  ' Read String of explicit length: Result: "Ha"
s=tabelle1.PString(17)        ' Read Pascal String (First-Byte is the length, 10). Result: "i love you"
Print "&h"+Hex(tabelle1.Addr) ' Display data structure address in Flash
do
loop

' Define data structure in Flash
```

```
data table1
  .db 1,2,3,4,5
  .dw &h4af1,&hcc55,12345
  .db "Hallo",0
  .db 10,"i love you",0
enddata
```

STRUCTURE

Save data using the structure.Example:

```
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) at user defined address as of 2013.R1

Declaration (Definition)

A declaration is a compiler command defining how to understand a identifier object or structure.Luna declares and defines many elements (Variable, Method etc.). and a pure dcleration such as Structure.

Descriptor()

Preprocessor	Only a function of the Preprocessor (Assembler)

The function returns the current position of the assembly descriptor, ie the current ***byte address*** in the flash memory, to which the assembly would have been translated to the assembly language code. With this function you can before changing the descriptor by the directive. "Org" read the position and then restore.

Syntax:

- **constant = descriptor()**

see also: Preprocessor (Assembler)

EXAMPLE

```
.set OLD_CPC = descriptor()/2 ;remember current position
.org  FOURTHBOOTSTART  ;take the following asm code from a new address
jmp  THIRDBOOTSTART
.org OLD_CPC                    ;Restore previous position
                                ;following asm code is stored again from old address.
```

Incr, Decr

Fast incrementation or decrementation by 1 (x = x +/- 1). This gives a speed advantage for data types byte, word, integer, long, when accessing arrays and structures. Will also work on floating values, but without a speed advantage.

Syntax

1. ***Incr*** *Variable*
2. ***Decr*** *Variable*

Syntax (alternative)

1. ***Variable***++
2. ***Variable***--

Info

Variable includes numeric properties in structures.

Example1:

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

incr a(4)   ' increment 5th element of Array "a" by 1
decr a(4)   ' decrement 5th element of Array "a" by 1
incr b      ' increment variable "b" by 1
```

Example2:

```
' Declare structure
struct date
  byte hour
  byte minute
  byte second
  byte value(5)
endstruct
' Dimension d as a structure
dim d as date

incr d.hour     ' increment "hour" by 1
incr d.wert(3)  ' increment 4th element of "value" by 1
decr d.wert(3)  ' decrement 4th element of "value" by 1
```

Defined()

Preprocessor	Solely a Preprocessor function

Checks if a symbol or constant is defined. The Preprocessor functions are for #if..#endif-Structures and allow conditional compiling of program code dependent on the *existance* of a symbol. **Defined()** is similar to the **#ifdef**-Directive in C as a function.**Valid are:**

- Constants, also controller constants like *avr.TIMSK*
- Labels from assembler source
- Library names e.g. *Graphics.interface*

Syntax:

- **defined(** *symbol* **)**
  The result is true, if *symbol* is defined.

See also: Directives, Compile process

Example

Luna

```
#if defined(avr.TIMSK0)
  'more
#endif
#if defined(Graphics.interface)
  'more
#endif
#if defined(mylabel)
  'more
#endif
```

Assembler

```
.if defined(avr.TIMSK0)
  'more
.endif
.if defined(Graphics.interface)
  'more
.endif
.if defined(mylabel)
  'more
.endif
```

Dim, eeDim

Dim (memory) and eeDim (eeprom) define Variables.

Syntax:

- ***Dim*** *Name[(elements)] [, NameN[(elements)] as [**static**] Datatype* [***= InitialValue***][1)]
- ***eeDim*** *Name[(elements)] [, NameN[(elements)] as Datatype* [***= InitialValue***][1)]

Visibility of Variables

- Variables defined in a class are *local* and static to the whole class. For example; Class "Avr", defined in the main program are also visible in in methods of the main program, as long as you have not variables of the same name inside the method. They are not visible in any other class. You can access variables of other classes by using ***ClassName.Property or Method***
- In Methods dimensioned variables are *local* (visible only within the method) and temporary or static (***static***) within the method.
- eeprom-Variables may be dimensioned *only* in classes or main program (of class "Avr"), not in a subroutine. They are static.

Name Collisions

If you define a variable named "myVar" in a class in the main program and a variable with the same name "myVar" in one of its methods (parameter or local variable), then the variable in the method has a higher priority, e.g. the classes variable will be hidden.

Static

static declares variables in a method as static. They are already reserved in memory from program start, will be initialized when the controller starts and will *not* be initialized when the method is first called such as a temporyry variable [2)]. Execution speed is higher as a temporyry variable. Remember that you should not execute this method in parallel because they will all access the same variable. This can cause hard to find errors.**By default, all variables of a method are declared temporary, e.g. they exist only as long as the method is executed.**

InitialValue

Implemented as of Version	2015.r1

InitialValue is a feature with which you can optionally assign an initial value to numerischen- and string variables in the dimensioning (in methods). Here all of the indicated variables, and arrays are filled with the initial value.

```
dim a,b(7) as byte = 42
dim s as string = "hello"
```

Notice

Strings are *dynamic* in memory and *static* in eeprom. When dimensioning a string in eeprom, you must add the string length identifier (number of characters).Read and write of eeprom-Variables is very slow and the number of write accesses is limited by the hardware. Do not permanently write eeprom-Variables!

Example

Example when dimensioning in memory

```
dim a,b as byte
dim c(19) as integer     ' Word-Array of 20 elements
dim s1 as string
dim s2(4) as string      ' String-Array of 5 elements
```

```
procedure hello(a as byte)
  dim c(19) as static integer     ' static Word-Array of 20 elements
  dim s1 as string
endproc
```

Example when dimensioning in eeprom

```
eedim a,b as byte
eedim c(19) as integer      ' Word-Array of 20 elements
eedim s1[20] as string      ' String of 20 byte in memory (1 string length, 19 characters)
eedim s2[10](4) as string   ' String-Array of 5 elements @ 10 byte each (1 string length, 9 characters)
```

[1] as of Version 2015.r1

[2] Luna initializes variables, Objects and structures that can store values with ***nil*** (same as zero

Do-Loop

Loop with a conditional exit. A Do-Loop is always entered and exited dependent on the loop end condition.**Syntax:**

- ***Do***
  - *Program code*
- ***Loop** [**Until** Expression]*

```

```

Example 1

```
Do
  Print "Hello"
Loop
```

Example 2

```
dim a as byte
Do
  Incr a
Loop Until a>10   ' Exit loop when a larger than 10
```

eeprom-endeeprom

Defines a object data structure in eeprom (Syntax 1) or a expression (Syntax 2+3) accessing the whole eeprom-memory.Ensure that you don't violete the structure boundary because this will overwrite the succeeding data.**Syntax 1:**

- ***eeprom*** *Identifier*
  - *Data*
- ***endeeprom***

Syntax 2:

- **eeprom**.*Method/Property* = Expression

Syntax 3:

- *Result* = **eeprom**.*Method/Property*

Method (read only)		
Name	**Description**	**Return Type**
.ByteValue(offset)	Read Byte	byte
.WordValue(offset)	Read Word	word
.IntegerValue(offset)	Read Integer	integer
.LongValue(offset)	Read Long	long
.SingleValue(offset)	Read Single	single
.StringValue(offset,bytes)	Read String of certain length	string
.PString(offset)	Read Pascal-String (Start-Byte is the length)	string
.CString(offset)	Read C-String (Null terminated)	string
.Addr	Address of data structure in Flash	word
.SizeOf	Read Size in Byte of the object	byte

- **offset:** Byte-Position within the structure starting at zero.
- **bytes:** Number of Byte to read.

DATA STORAGE

The directives for the storage of data within a data object.

- .db - 8 Bit values incl. strings
- .dw - 16 Bit values
- .dt - 24 Bit values
- .dl - 32 Bit values

Note

Accessing areas outside the object boundary is possible and not checked.

Example

```
dim a as byte
dim s as string
a=table1.ByteValue(4) ' Read Byte of tabele1+4, reszlt: 5
s=table1.CString(12)  ' Read C-String, result: "Hello"
table1.ByteValue(1)=7 ' Write Byte
table1.CString(0)="I am an eeprom-String"  'Write string
a=eeprom.ByteValue(4)    ' Read a Byte from eeprom memory
eeprom.ByteValue(4)      ' Write a Byte from eeprom memory

' Define Data Structure in eeprom
' Note:
' The compiler stores the eeprom-values and strings in the file *.eep.
' They must also be uploaded to the controller.
eeprom table1
  .db 1,2,3,4,5
  .dw &h4af1,&hcc55,12345
  .db "Hello",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() checks whether the Integer-Value passed is *even*
Odd() checks wether the Integer-Value passed is *odd*.

Preprocessor	The function is also available in the Preprocessor, does not create machine code.

Syntax: *boolean* = ***even***(*Expression*)
Syntax: *boolean* = ***Odd***(*Expression*)**Example:**

```
dim a as byte
a=7
if even(a) then
 print "The number ";str(a);" is even."
else
  print "The number ";str(a);" is odd."
end if
```

See also: Fodd, Feven

Fabs()

Fabs() returns the absolute value of the floating value passed.**Syntax:** ***single*** = ***Fabs***(*Expression*)**Example:**

```
dim a as single
a = -123.456
a = Fabs(a)  ' result: 123.456
a = 123.456
a = Fabs(a)  ' result: 123.456
```

Facos()

Facos() calulates the Arc-Cosine of the value (Radiant) passed.

Syntax: ***single*** = ***Facos****(Single)*

Example:

```
dim a as single
a=0.5
a = Facos(a)   ' result: 1.04719
```

Fadd, Fsub, Fmul, Fdiv

Floating point math functions (technical no difference to expressions).

Syntax 1: ***Fadd*** *Variable, Constant*
Syntax 2: *result* = ***Fadd***(*Expression,Expression*)Fsub, Fmul and Fdiv similar to Fadd.**Example Syntax 1:**

```
dim a as single

fadd a, 1000  ' Similar to a = Single(a + 1000)
```

Example Syntax 2:

```
dim a as integer
dim b as single

b=fmul(a,b)  ' Same as b = Single(a * b)
```

Fasin()

Fasin() calculates the Arc-Sine of the Value (Radiant) passed.**Syntax:** ***single*** = ***Fasin(*** *Single* *)*

Example:

```
dim a as single
a=0.5
a = Fasin(a)  ' result: 0.52359
```

Fatan()

Fatan() calculates the Arc-Tangent of the Vallue (Radiant) passed.**Syntax:** ***single*** = ***Fatan***(*Single*)

Example:

```
dim a as single
a=0.5
a = Fatan(a)  ' result: 0.46364
```

Fcbrt()

Fcbrt() calculates the cubic root of the value passed.**Syntax:** ***single*** = ***Fcbrt(*** *Single)*

Example:

```
dim a as single
a = 27
a = Fcbrt(a)  ' resul: 3
```

See also: Fsqrt()

Fceil()

Fceil() rounds to the next whole number. To zero when negative, away from zero when positive.

Syntax: ***result*** = ***Fround***(*value as single*)

Example:

```
dim a as single
a=100.5
a = Fround(a)   ' result: 101.0
a=-100.5
a = Fround(a)   ' result: -100.0
```

See also: Floor(), Fround(), Fix(), Frac()

Fcosh()

Fcosh() calculates the hyperbolic Cosine of the value (Radiant) passed.

Syntax: ***single*** = ***Fcosh***(*Single*)

Example:

```
dim a as single
a=0.5
a = Fcosh(a)  ' result: 1.12762
```

Fcos()

Fcos() calculates the Cosine of the value (Radiant) passed.

Syntax: ***single*** = ***Fcos***(*Single*)

Example:

```
dim a as single
a=0.5
a = Fcos(a)  ' result: 0.87758
```

Fdeg()

Fdeg() converts the angle from Radiant to Degrees.

Syntax: ***single*** = ***Fdeg****(angleRad as Single)*

Example:

```
dim Angle,angleDeg as single
angle=0.5
angleDeg = Fdeg(angle)
```

See also: Frad()

Feven()/Fodd()

Checks if a floating value is even or odd.

Feven() checks if ***even***

Fodd() checks if ***odd***.

Syntax: *byte* = ***Feven****(a as single)*

Syntax: *byte* = ***Fodd****(a as single)*

RETURN VALUES

- **0** = False
- **1** = True
- **2** = Neither

The function requires whole numbers. The whole part of the number is converted to an integer, therefore the decimal part is not taken into account.The function returns "neither even nor odd" if the decimal part is not zero.
Example:

```
dim a as single
a=28.22

select case Fodd(a)
case 0
  print "The number ";str(a);" is even."
case 1
  print "The number ";str(a);" is odd."
case 2
  print "The number ";str(a);" ist neither even nor odd"
end select

select case Feven(a)
case 0
  print "The number ";str(a);" is odd."
case 1
  print "The number ";str(a);" is even."
case 2
  print "The number ";str(a);" ist neither even nor odd"
end select
```

See also: Odd, even

Fexp()

Fexp() is the exponential function (e-Function) to the mit der Euler's number.

Syntax: ***single*** = ***Fexp(*** *Single)*

Example:

```
dim a as single
a = 2
a = Fexp(a)   ' result: 7.38905
```

Fix()

Fix() returns the whole part of a float number as an signed integer.

Syntax: ***int32*** = ***Fix***(*value as single*)

Example:

```
dim a as single
dim b as long
a = 100.5
b = Fix(a)   ' result: 100
```

See also: Floor(), Fround(), Fceil(), Frac()

Flexp()

Flexp() ist die inverse function of Frexp().The function returns the multiplication of the mantisse ***m*** by 2, raised to the exponent ***e***.

Syntax: ***single*** = ***Flexp***(*m as Single, e as word*)

Example:

```
dim e as word
dim a as single
a = 100
a = Frexp(a,e)   ' result: 0.78125 und in e: 7
a = Flexp(a,e)   ' result: 100.0
```

See also: Frexp()

Flog10()

Flog10() calculates the logarithm, Basis 10.

Syntax: ***single*** = ***Flog10***(*Single*)

Example:

```
dim a as single
a = 2
a = Flog10(a)   ' result: 0.30103
```

Flog()

Flog() calculates the natural logarithm.

Syntax: ***single*** = ***Flog***(*Single*)

Example:

```
dim a as single
a = 2
a = Flog(a)   ' result: 0.69314
```

Floor()

Floor() rounds down to the next whole number, dependent on the sign.

Syntax: ***ergebnis*** = ***Floor***(*value as single*)

Example:

```
dim a as single
a=123.50011
a = Floor(a)   ' result: 123.0
a=-123.50011
a = Floor(a)   ' result: -124.0
```

See also: Ftrunc(), Fround(), Fix(), Frac()

Format()

Format() converts a numeric input into a formatted decimal string. The function uss str(). The formatting expression defines the output format.

Präprozessor	The function is also available in the Preprocessor, uses constansts and does not create any machine code.

SYNTAX

string = **format**(*formatSpec*, *Expression*)

- **formatSpec:** String(Constant), with the formatting instruction.
- **Expression:** Expression with numeric result.

"Decimal String" denotes the converted numerical value.

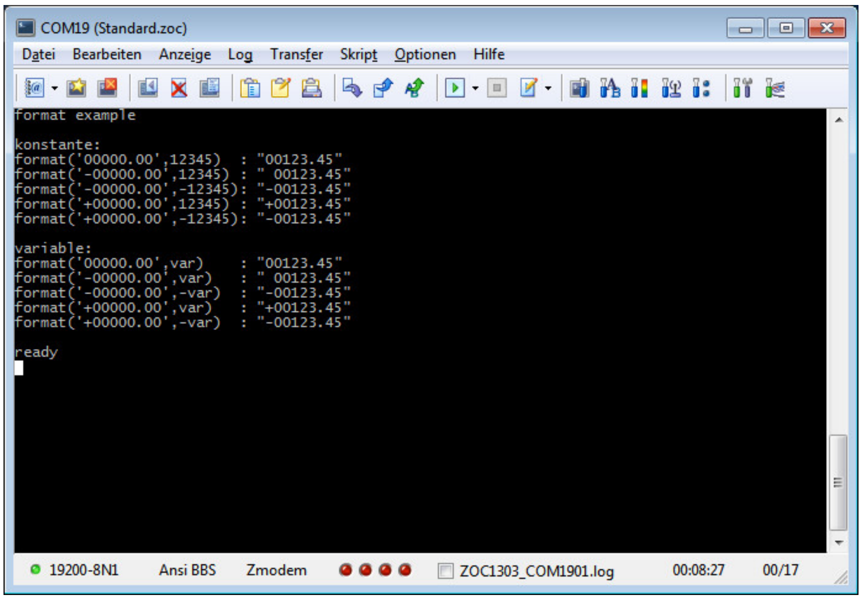
Format description	
Zeichen	**Description**
0	Placeholder for a number in the decimal string
.	Placeholder for a dot. **When Integer** the placeholder will split the numeric display at this position. **When Floating** the decimal point will be at this position. If the dot is omitted, the floating value will show the whole part of the number only.
+	Placeholder for the sign. When negative, a "**-**" is inserted, when positive, a "**+**" is inserted.
-	Placeholder for a negative sign. When negative a blank is inserted.

Other characters in ***formatSpec*** will not be interpreted and passed to the result string.Formatting is **right aligned**, starting from low to high order decimal.

EXAMPLE

```
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 "constant:"
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
```

OUTPUT


COM19 (Standard.zoc)
Datei Bearbeiten Anzeige Log Transfer Skript Optionen Hilfe
19200-8N1 Ansi BBS Zmodem ZOC1303_COM1901.log 00:08:27 00/17

```
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
```

For-Next

Loop with counter, automatically increments or decrements.The loop will be entered if the loop counter can can *reach* the target value and will leave the loop when the target count value is passed.Calculation and comparison of the loop target value is executeted at the loop beginning.**Syntax:**

- ***For** Variable = Expression **to|downto** [**step** Constant]*[1]
  - *Program Code*
- ***Next***

The keyword **to** increments the loop counter, **downto** decrements it.
The optional keyword **step** defines the step of a loop cycle. A positive constant is expected. Default is 1.
See also: Continue

Example 1

```
dim i as byte
For i=1 to 10
  Print "Hello"   ' Print "Hello" 10 times
Next
```

Example 2

```
dim i as byte
For i=10 downto 1   ' Loop counter downwards
  Print "Hello"   ' Print "Hello" 10 times
Next
```

Example 3

```
dim i,a,b as byte
a=1
b=10
For i=a to a+b
  Print "Hello"   ' Print "Hello" 11 times
Next
```

Example 4

```
dim i,a as byte
a=0
For i=1 to a      ' Lopp will be omitted because the the loop counter can not reach the target value
  Print "Hello"
Next
```

[1] **step** as of 2012.r7.2.build 3875

Fpow()

Fpow() exponentiates the value passed ***a*** with the exponent ***b***.**Syntax:** ***single*** = ***Fpow****(a as Single, b as Single)*

Example:

```
dim a,b as single
a = 1.23
b = 4.32
a = Fpow(a,b)  ' output: 2.44652
```

Frad()

Frad() converts the angle from degrees to radiant.**Syntax:** ***single*** = ***Frad****(angleDeg as Single)*

Example:

```
dim angleDeg,angleRad as single
angleDeg=40
angleRad = Frad(angleDeg)
```

See also: Fdeg()

Frac()

Frac returns the decimal part of a floating (Single) value.

Syntax: ***result*** = ***Frac***(*value as single*)

Example:

```
dim a as Single
a=23.00423
a = Frac(a) ' output: 0.00423
a=-23.00423
a = Frac(a) ' output: -0.00423
```

See also: Ftrunc(), Fround(), Fix(), Floor()

Frexp()

Frexp() splits a floating number ***a*** into mantissa and exponent. The exponent is stored in ***e***. ***e*** may therefore not be an expression, but instead it must be a single byte variable in memory (SRAM).The return value is the mantissa of the floating value ***a***.

Syntax: ***single*** = ***Frexp****(a as Single, byRef e as word)*

Example:

```
dim e as word
dim a as single
a = 100
a = Frexp(a,e)  ' output: 0.78125 and e: 7
```

See also: Flexp()

Fround()

Fround() rounds to the next whole number away from zero (bankers' rounding).

Syntax: ***result*** = ***Fround***(*value as single*)

Example:

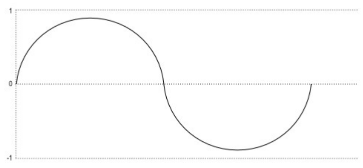
```
dim a as single
a=100.5
a = Fround(a)   ' output: 101.0
a=-100.5
a = Fround(a)   ' output: -101.0
```

See also: Floor(), Fceil(), Fix(), Frac()

Fsine()

Fsine() is a fast 360° Sinus function. Input parameter is an angle (0-359) in degrees. The result is a Single precision value between -1 and +1.

Syntax: ***result*** = ***Fsine***(*Expression*)Fsine does a 20x faster conversion that the equivalent floating point Sinus/Cosine calculation because it uses a lookup table. The function accepts whole numbers only.



Grad	result
0	0
45	0.7071443
90	1
180	0
270	-1

Example:

```
dim angle as byte
dim result as single
angle=90
result = Fsine(angle)
```

See also: Sine()

Fsinh()

Fsinh() returns the hyperbolic Sinus of the passed value (Radiant).

Syntax: ***single*** = ***Fsinh***(*Single*)

Beispiel:

```
dim a as single
a=0.5
a = Fsinh(a)   ' output: 0.52109
```

Fsin()

Fsin() returns the Sine of the passed value (Radiant).

Syntax: ***single*** = ***Fsin****(Single)*

Example:

```
dim a as single
a=0.5
a = Fsin(a)  ' output: 0.47942
```

Fsplit()

Fsplit() splits a floating number ***a*** into a whole number and decimal number. The whole number part is saved to variable ***b***. ***b*** must be a single variable in memory (SRAM).The return value is the decimal part number of ***a***. The sign is maintained.

Syntax: ***single*** = ***Fsplit(*** *a as Single, byRef b as single)*

Example:

```
dim a,b as single
a = 100.23
a = Fsplit(a,b)  ' output: 0.23 and b: 100.0
```

Fsqrt()

Fsqrt returns the square root of a floating number (Single).

Syntax: ***ergebnis*** = ***Fsqrt****(Expression)*

Example:

```
dim a,b as Single
b=1.4
a=b^2         ' output: 1.959
a = Fsqrt(a) ' output: 1.4 (1.399)
```

See also: Sqrt

Fsquare()

Fsquare() returns the square of the value passed.

Syntax: ***single*** = ***Fsquare****(Single)*

Example:

```
dim a as single
a=4.2
a = Fsquare(a)  ' output: 17.64
```

Ftanh()

Ftanh() returns the hyperbolic Tangent of the value passed (Radiant).

Syntax: ***single*** = ***Ftanh***(*Single*)

Example:

```
dim a as single
a=0.5
a = Ftanh(a)  ' output: 0.46211
```

Ftan()

Ftan() returns the Tangent of the value passed (Radiant).

Syntax: ***single*** = ***Ftan****(Single)*

Example:

```
dim a as single
a=0.5
a = Ftan(a)  ' output: 0.5463
```

Ftrunc()

Ftrunc() rounds to the next whole number, not larger than the whole part of the number.

Syntax: ***result*** = ***Ftrunc****(value as single)*

Example:

```
dim a as single
a=123.5
a = Ftrunc(a)  ' output: 123.0
a=-123.5
a = Ftrunc(a)  ' output: -123.0
```

See also: Floor(), Fround(), Fix(), Frac()

Function

A Function is a subroutine similar to a Procedure, returning a value.

Syntax:

- ***Function*** *Name(Var* ***as*** *Datatype[, VarN* ***as*** *Datatype]****)*** ***as*** *Datatype*
  - *[Program Code]*
  - ***Return*** *Expression*
  - *[Program Code]*
- ***endFunc***
- **Name:** Identifier of the subroutine
- **Var:** Identifier of the parameter variable

Example of a function declaration and call to it:

```
// Main program
dim result as word
result=Addition(112,33) // Calls the function

function Addition(var1 as word, var2 as byte) as word
  return var1+var2
endfunc
```

Fval()

Converts a foating value string into a binary value. The result is a signed 32 Bit floating (single/float) value. Ignoriert führende Leerzeichen. The decimal sign is the ".".

Syntax: *single* = ***Fval***(*text as string*)

Note: *The function expects an SRAM-String, EEPROM is not permitted, a constant string parameter eg:* ***fval("12345")*** *are processed by the preprocessor.*

Example:

```
dim s as string
dim value as single
s = "123.45"
value = Fval(s)
```

GarbageCollection()

GarbageCollection() cleans up the dynamically managed memory. All the deleted memoryblocks (also strings and objects based on memoryblocks) are removed.

Syntax: ***GarbageCollection**()*

Note

By default, the dynamically managed memory is automatically cleaned during the processing. But the automatic cleanup can be disabled by the pragma MemoryBlocksGarbageCollection This may be necessary in order to obtain faster processing in certain parts of the program, because during processing with strings or memoryblocks no automatic cleanup is started. Here it is then necessary to be able to control the cleanup manually. **If automatic cleaning is enabled (default), then the call to this function is useless.**

Halt()

Creates an empty continuous loop

Syntax: ***Halt()***

Example:

```
Halt()
' above expression is the same as:
do
loop
```

Hex()

Convert number to a hexadecimal string. Takes the data type into account.

Preprocessor	The function is also available in the Preprocessor, creates constants only, e.g. does not generate machine code.

Syntax: *String* = ***Hex***(*Expression*)

Example:

```
dim s as string
s = Hex(41394)  ' output: "A1B2"
```

HexVal()

Converts a string with hex notation to a unsigned integer. The conversion routine knows the notation literals ***0x*** and ***&h*** and the conversion is not case-sensitive. First invisible chars (ASCII 0-32) are ignored/skipped.

The result is unsigned 32 Bit integer (long/uint32), but if you assign this value to a signed 32 bit datatype such as *longin* or *int32*, the value will become signed.

Syntax: *long* = ***HexVal****(text as string)*

EXAMPLE

```
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 is a ***global event*** which is called when the processor has no commands to process. If ***Idle-endIdle*** is omitted in your source code, wait loops in the Luna internal methods are skipped.**Syntax:**

- ***Idle***
  - Programmcode
- ***endIdle***

Methods that call this event while waiting:

- Uart Read-Funktionen
- Uart Write-Funktionen
- InpStr

You can call this event in your own loops to pass idle time to this event:

- Avr.Idle

Example:

```
avr.device = attiny2313
avr.clock = 20000000
avr.stack = 12
Uart.Baud = 19200
Uart.Recv.enable
Uart.Send.enable
do
  a = Uart.ReadByte                       ' wait for a character, then read it
  Print "Character received: "+34+a+34    '"A" received
loop

' is calle dwhile Uart.Read is waiting for a character
Idle
  print "nothing to do"
endIdle
```

If-elseIf-else-endIf

Conditional branching.**Syntax:**

- ***If*** *Expression1* ***Then***
  - Program code when Expression1 = TRUE and Expression2= TRUE
- ***elseIf*** *Expression2* ***Then***
  - Program code when Expression1 = TRUE
- ***elseIf*** *Expression3* ***Then****]*
  - Program code when Expression1 = TRUE and Expression2 = TRUE
- *[**else**]*
  - Program code when Expression1 = FALSE and Expression2 = FALSE
- ***endIf***

EXAMPLE

```
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
```

Instr()

Find a substring in a string and return its starting position.The function is case sensitive.

Syntax: *byte* = ***Instr****([startPos,][1] Source as string, Find as string)*

- **startPos:** Starting position in the main string, beginning from left at position 1 (optional)
- **Source:** Main character string to search in
- **Find:** Substring to find
- **Returns:** Position from left, starting with 1. 0 = not found.

Example:

```
dim s as string
dim pos as byte
s = "Hello World"
pos = Instr(s,"lo")     ' output: 4
pos = Instr(s,"world")  ' output: 0 (not found)
```

[1] As of 2013.R1

Jump

Direct jump to a Lable or Address.

Syntax: ***Jump*** *LableName/Address*

Example 1:

```
  jump test // Jump to "test" and continue execution there
  Print "1" // will not be executed
test:
  Print "2"
```

Example 2:

```
 jump &h3c00 ' Jump to Bootloader & Reset (atmega32)
```

Label

Label are jump addresses in Luna- or Assembler-Code, that can be called/jumped to.

PROPERTIES

A Label has the property ***.Addr***, that holds the Label's address in program memory (Flash). Use **MyLabel.Addr** to acces this value.

SEE ALSO

- Jump
- Call
- Void
- Icall

Left()

Returns the left part of a string.**Syntax:** *string* = ***Left***(*Source as string, Position as byte*)
Position: Starts with 1 from left**Example:**

```
dim s as string
s = Left("Hello World",5)  ' output: "Hello"
```

Len()

Returns the current number of characters in a string.

Syntax: *Number* = ***Len****(Expression)*

Example:

```
dim a as byte
dim s as string

s = "Hello World"
a = Len(s)          ' output: 11
```

Lower()

Convert string to lower case including Zeichenkette in Kleinbuchstaben wandeln, beachtet sämtliche umlaut.

Syntax: *string* = ***Lower****(Source as string)*

Example:

```
dim s as string
s = Lower("ABCDeFG")  ' output: "abcdefg"
```

Min(), Max()

Min() returns the smallest value from a list of values. Max() returns the largest value. The Function accepts a variable number of parameters (2 to 255). The first value's data type defines the function type. All other parameters are assumed to be of this type and converted to that type if required.

Preprocessor	The function is available in the Preprocessor, and does not create machine code because only constants are used.

Syntax:

- ***Result*** = ***Min***(*Value1*,*Value2* [,*WertN*])
- ***Result*** = ***Max***(*Value1*,*Value2* [,*ValueN*])

Example:

```
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
print str(min(a,b,c))
print str(max(a,b,c))
print "ready"
do
loop
```

Median16s(), Median16u()

Median is a number, in the middle of a row of numbers, sorted by value. Can be used as a Median filter to process Input data (Rank order filter). Voltage readings of an ADC are much more stable using Median because variations are much smaller.

Two types are defined for 16-Bit values. With and without taking the sign into account.

Syntax: ***word*** = **Median16u**(valuesAddr as word, valueCount as byte)
Syntax: ***integer*** = **Median16s**(valuesAddr as word, valueCount as byte) (Signed value)

The parameter expects a memory address (example an Array), and the number of values. The input values are temporarily pushed on the stack. Take this into account when sizing the stack.

EXAMPLE

```
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

'Input data
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 manually:"
'Sort input data in ascending order
b().Sort
'Display it
for i=0 to b().Ubound
  print "b(";str(i);") = ";str(b(i))
next
print
i = b().Ubound/2 ' the middle element is the Median
mv = b(i)
'There is no middle element when the number of elements is even. The Low- and High-Median will be averaged.
if even(b().Ubound+1) then
  mv = (mv + b(i+1)) / 2
end if

print "Median() = ";str(mv)
print
print "ready"
do
loop
```

MemCmp()

MemCmp() compares to areas in memory (SRAM).**Syntax**: *result* = **MemCmp(** *s1Adr as word, s2Adr as word, numBytes as word* **)**

- **s1Adr/s2Adr:** Start addresses of the two memory areas (SRAM).
- **numBytes:** Number of bytes to compare.
- **result:** Integer-value of the result, meaning:
  - **= 0:** s1 = s2
  - **> 0:** s1 > s2
  - **< 0:** s1 < s2

EXAMPLE

```
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)="Hello"
m.CString(20)="Hello"
print "MemCmp(): ";str(memCmp(m.Ptr+0,m.Ptr+20,5))  'output: 0 (gleich)
print
print "ready"
do
loop
```

MemCpy()

MemCpy() does a high speed copy between two memory blocks (SRAM).**Syntax**: **MemCpy(** *srcAddr as word, dstAddr as word, numBytes as word* **)**

- **srcAddr:** Starting address of the source memory block (SRAM).
- **dstAddr:** Starting address of the target memory block (SRAM).
- **numBytes:** Number of Bytes to copy.

EXAMPLE

```
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)="Hello World"
memCpy(m1.Ptr,m2.Ptr,11)  'copies the string including Nullbyte to the 2. memory block
print "m2.CString(0) = ";34;m2.CString(0);34 'output from 2. memory block
print
print "ready"
do
loop
```

MemSort()

MemSort() sorts all Bytes of a memory block in ascending order (SRAM).**Syntax**: **MemSort(** *sAdr as word, numBytes as word* **)**

- **sAdr:** Starting address of the memory block (SRAM).
- **numBytes:** Number of Byte to sort.

EXAMPLE

```
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(10)
m.PString(0)="76193."
print "before: ";34;m.PString(0);34
MemSort(m.Ptr+1,m.ByteValue(0))
print "after: ";34;m.PString(0);34 'output: "    .13679"
print
print "ready"
do
loop
```

MemRev()

MemRev() sorts the Bytes of a memory block in descending order.**Syntax**: **MemRev(** *sAdr as word, numBytes as word* **)**

- **sAdr:** Starting address of the memory block (SRAM).
- **numBytes:** Number of Byte to sort in descending order.

EXAMPLE

```
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(10)
m.PString(0)="76519."
print "before: ";34;m.PString(0);34
MemRev(m.Ptr+1,m.ByteValue(0))
print "after: ";34;m.PString(0);34 'output: "    .15679"
print
print "ready"
do
loop
```

MemoryBlock

is a memory area in the working memory (SRAM), actually it is a memory object in Luna's dynamicly managed memory. Creating a block reserves that number of Byte in memory an can be used to manage your data.Access to the object via the object methods only because it is managed dynamicly, e.g. the addresses change dynamicly.Call the "New"-Method to create a Memory Block. The maximus size are 65KB.

CREATE A NEW MEMORYBLOCK

```
dim myvar as MemoryBlock
myvar = New MemoryBlock(numBytes)
```

RELEASE A MEMORYBLOCK

The release of a memory block occurs automatically when assigning a new memory block, and departure from a method, *except* it is a global variable, a reference ("byRef" parameter) or a static variable ("dim x as *static* ...").**Code-Examples for explanation see bottom.**

Methods (read only)		
Name	**Description**	**Return Type**
.Addr	Address of the variable in memory	word
.Ptr	Address of the allocated memory block	word
.Size	Total number of Bytes of the allocated memory block	word
Methods (read / write)		
Name	**Description**	**Return type**
.ByteValue(offset)	Read/Write Byte	byte
.WordValue(offset)	Read/Write Word	word
.IntegerValue(offset)	Read/Write Integer	integer
.LongValue(offset)	Read/Write Long	long
.SingleValue(offset)	Read/Write Single	single
.StringValue(offset,bytes)	Read/Write String of fixed length	string
.PString(offset)	Read/Write Pascal-String (Start-Byte is the length information)	string
.CString(offset)	Read/Write C-String (Null terminated)	string

- **offset:** Byte-Position within the memory block, basis 0.
- **bytes:** Number of Byte to read.

NOTICE

Memory Block boundry violation is permitted and not validated (faster). You may define a exception.

See also: Memory Management, Sram, Dump

Examples

```
dim a as byte
dim s as string
dim m as MemoryBlock
m.New(100) ' Create Memory Block and release exising block, if defined
if m <> nil then                    ' Check to see if memory block was allocated
  m.ByteValue(0)=7                  ' Write Byte
  m.CString(1)="I am a String"  ' Write String
  a=m.ByteValue(0) ' Read Byte, output: 7
  s=m.CString(1)   ' Read C-String, output: "I am a String"
  m.free           ' Release Objekt memory
end if
```

Examples for automatic release.

```
procedure test(m as MemoryBlock)
  'm is byVal and destroyed at return
endproc
```

```
procedure test()
```

```
  dim m as MemoryBlock
  'm is byVal and destroyed at return
endproc
```

```
procedure test(byRef m as MemoryBlock)
  'm is byRef and remains untouched
endproc
```

```
dim m as MemoryBlock

procedure test()
  'm is not part of the method and remains untouched
  'Reason:
  'Visibility of global variables in methods as long as no separate
  'Variable with the same name was declared in the method
endproc
```

```
function test(byRef m as MemoryBlock) as MemoryBlock
  'm is byRef and remains untouched
  return m 'returns a new instance of m
endfunc
```

```
function test() as MemoryBlock
  dim m as MemoryBlock
  return m 'the memoryBlock is dereferenced (detached) from "m" and returned
endfunc
```

Mid()

Returns a substring of specified length.**Syntax:** *string* = ***Mid****(Source as string, Position as byte[, Length as byte])*

- **Position:** Starts from left with 1
- **Length (optional):** Number of characters to return.

Example:

```
dim s as string
s = Mid("Hello World",3,3)  ' ergebnis: "llo"
```

Arithmetical Operators

+ (ADD)

This operator is used to **add** numerical values or **bind** strings together.

result = Expression1 + Expression2

Following example addiert different values:

```
dim x as integer
x = 1+2+3
```

Following example binds different strings together:

```
dim a as string
a = "i am "+" a "+" string"
a = a + " - 123"
```

- (SUB)

This operator is used to **subtract** two values.

Result = Expression1 - Expression2

This example does subtract different values:

```
dim x as integer
x = 10-2-4
```

* (ASTERIX)

This operator is used to **multipy** two values.

Result = Expression1 * Expression2

This example does multiply different values:

```
dim x as integer
x = 1*2*3
```

/ (SLASH)

This operator is used to **divide** two values.

Result = Expression1 / Expression2

This example does divide different values:

```
dim x as integer
x = 9 / 3
```

^ (CIRCUMFLEX/HAT)

This operator is used to **potentiate** a value.

Result = Expression1 ^ Expression2

This example does potentiate a value with 2:

ATTENTION! This function is processed in floatingh point arithmetic, like the function *pow()* in C. Integer arguments are automatically converted.

```
dim x as single
x = 1.234 ^ 2
```

MOD

This operator performs a modulo operation of a value.

Result = Expression1 MOD Expression2

The MOD-operator works with integers (Values without decimal places). Floating-point values would reduced to integers. Is one of the two arguments of the value = 0, then the result is undefined.

```
dim x as integer
x = 5 mod 2     ' Result: 1
x = 5 mod 1.999 ' Result: 0
```

Nop

No operation can be used in wait loops. It requires 1 CPU cycle.

Syntax: ***Nop***

Example:

```
// wait 3 CPU cycle
nop
nop
nop
```

NthField()

Read a substring that is delimited by a separator.

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

- **source:** String to inspect.
- **separator:** Delimiting String.
- **index:** Element index to read, starting from left with 1.

See also: CountFields()

EXAMPLE

```
dim a as string
a = "This | is | an | example"
print NthField(a,"|",3)  ' output: " an "
```

Part-endpart

Use Part/endpart to *visually* structure your source code. It does not create machine code. Visually combine code with the editor's Expand/Reduce function.**Syntax:**

- **part** *Description*
  - [..]
- **endpart**

Pascal-String

Luna uses Pascal-String format as a standard (Variable, Structured element or Constant). A Pascal-String is made up of a leading lenght descriptor Byte and the data. This also allows you to store Binary data in a string.

Pascal-String structure
"hallo" is saved to memory like this:

Memory ?					
0x05	0x68	0x61	0x6c	0x6c	0x6f
(length)	h	a	l	l	o

Print

Short notation for Uartn.Print
This description is also valid for SoftUart.Print
Print is on of the the most comprehensive output functions. Use Print to write data to a serial port:

- individual variables of any type
- complete Array contents
- Strings
- Constants
- Return values of Functions or Objects

Value separation

Values are separated by the Semicolon and then printed in their order from **left to right**:

- ***Print*** *Expression1;Expression2*

If the last character is a semicolon, then the automatic new line function ASCII 13 and ASCII 10 (CRLF) is suppressed.

- ***Print*** *Expression1;Expression2;*

Print will print an empty new line:

- ***Print***

This expression will print nothing:

- ***Print*** *;*

Every expression is interpreted for itself and then the results are printed one after the other:

- ***Print*** *(numVar+1);(numVar1*numVar2);"Hello"+stringVar*

This will print the result of a mathematical expression and the string "Hello" and stringVar (no extra memory required). 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;"Hello"+stringVar*

Output Variables

Print Variable or indexed Array elements:

- ***Print*** *numVar* - prints the content of Variable "numVar"
- ***Print*** *numVar(1)* - prints the content of Array element "numVar(1)"
- ***Print*** *stringVar* - prints the content of Variable "stringVar"

Constants print their typed values :

- ***Print*** *22* (output 22 as a Byte value)
- ***Print*** *1234* (output 1234 as a Word value)
- ***Print*** *-1234* (output -1234 as a Integer value)
- ***Print*** *12.34* (output 12.34 as a Single value)
- ***Print*** *1234567* (output 1234567 as a Long value)
- ***Print*** *"text"* (gibt die Zeichenkette "text" aus)

Print the return value of (Objekt/Interface-) Methods or properties:

- ***Print*** *function1(parameter1, parameter2, ..)*
- ***Print*** *PortB*
- ***Print*** *Timer1.Value*
- ***Print*** *str(numVar)*

Example:

```
dim var,c(4),a,b as byte
var=97 ' ASCII-character "a"
c(0)=asc("H")
c(1)=asc("e")
```

```
    c(2)=asc("l")
    c(3)=asc("l")
    c(4)=asc("o")
    a=100
    b=5
    Print "Hello World ";str(12345)  ' output: "Hello World 12345"
    Print "Hello World ";65          ' output: "Hello World A"
    Print "Hello World ";Str(65)     ' output: "Hello World 65"
    Print "Hello World ";c(4)        ' output: "Hello World o"
    Print "Hello World ";str(a/b)    ' output: "Hello World 20"
    Print var                        ' output: "a"
```

Procedure

A Procedure is a subroutine with local memory and optional parameters. You may pass parameters to the subroutine when calling it. You may also use global Variables of the main program/superior Class.

See also: What is a subroutine?

Local Variables

Lokal variables can be declared in the Method, that are valid only until you exit the Method. Passing parameters requires additional memory until the subroutine is exited. *Without Parameters and/or local variables its just the return address (2 or 3 Byte).***See also: Dim, Chapter visibility of variables and keyword "static"**

Typecasting

If the parameter passed is of a different type than in the subroutine, the numeric data type will be *typecast automatically*. Example, if you pass a "Word" variable and the Subroutine expects a Byte, then the Word will be converted to a Byte.

Parameter Passing

Parameters can be passed byVal Copy or byRef Reference[1] to the Subroutine. The parameters are exchanged via the program stack (see avr.Stack).**Paramater passing per Copy (byVal):**

- Constants
- Variables (except Structures and whole Arrays)
- Expressions
- Object properties and -Methods with return value
- Methods with return value

Paramater passing per Reference (byRef):

- Constants-Strings
- Constant-Object "data"
- Memory-Object "sram" (Variables, Arrays, Structures, ..)
- Variables incl. Structures

The keyword **"byVal"** is the default method and can be omitted.

InitialValue Parameter

InitialValue is a feature with which you can optionally assign an initial value parameters:

```
procedure test(a as byte, b as word = 456)
endproc
procedure test1(a as byte = 123, b as word)
endproc
```

The call can then be different by then the optional parameters, then the omitted parameter when called by the initial value will be replaced automatically:

```
test(123,333)
test(100) 'the optional second parameter is given the initial value
test1(,444) 'the first parameter is replaced by the initial value
```

Method-Overload

Methoden-Overload is a feature in which multiple methods with the same name can be defined with different parameter types or return values. The call is then determined by the number and type of parameters.

```
test(123)     'methode #1 is called
test(123,456) 'methode #2 is called
a = test(123) 'methode #3 is called

'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)
endfunc
```

Assignment

Implemented as of Version	2015.r1

With the keyword ***assigns*** at the last parameter of a method specifies a method (Procedure), a value can be assigned in the source code.

```
test(100) = 200

'[...]
procedure test(a as byte, assigns b as word)
  print "parameter  = ";str(a)
  print "assignment = ";str(b)
endproc
```

Inline

The optional keyword ***inline*** makes a inline-method.

See also: http://en.wikipedia.org/wiki/Inline_expansion

Namespace

Methods can access the global variables of the superior Class. A variable "a", declared in the main program, is also accessible by a Procedure/Function. However, if you declare a variable of the same name in a Procedure/Function or define it as a parameter, then this variable will be the **only one accessible**.

Recursive/Nested Calls

Luna Methods Reentrancy are reentrant and can be nested or called recursively, or called by one or more Interrupts of the main program.

Syntax

Syntax:

- ***[inline] Procedure*** *Name([byVal/byRef] [Var* ***as*** *Data Type[, [byVal/byRef] VarN* ***as*** *Data Type)*
  - *Program Code*
  - *[Return]*
  - *Program Code*
- ***endProc***
- **Name:** Identifier of the Subroutine
- **Var:** Identifier of the Parameter Variable

Example 1:

```
' Main Program
dim value as integer
dim s as string
output("My Number: ",33) ' Call the Subroutine
'also possible
value = 12345
s = "Other Number: "
call output(s,value)
do
loop

' Subroutine
procedure output(text as string, a as byte)
  dim x as integer  ' locally valid Variable
  x=a*100
  print text+str(x)
endproc
```

Example 2:

```
struct point
  byte x
  byte y
endstruct
dim p as point
p.x=23
test(p)
print "p.x = ";str(p.x)   ' output: p.x = 42
do
loop

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

```
endproc
```

Example 3:

```
test(text)      ' output: c.PString(0) = "hello"
test("ballo")   ' output: 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,"hello"
enddata
```

1) as of 2012.r4

Push/Pop Functions

Pushxx() pushes the assigned value onto the Stack.
Popxx() pops the top value from the 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****()*

IMPORTANT!

The function is the same as the machine command Push/Pop. After pushing bytes onto the stack you MUST pop the same number of bytes *in the same level ot code prior to exiting.*

EXAMPLE

```
PushByte(avr.SREG)
cli
MyFunction(1,2,3)
avr.SREG = PopByte()
```

Replace(), ReplaceAll()

Note	Only available in the preprocessor!

Find and replace a String with another string.

SYNTAX

- ***Replace***(*source* ***as string***, *find* ***as string***, *replacement* ***as string***)
  Replaces the first occurrence of a string with another string.

- ***ReplaceAll***(*source* ***as string***, *find* ***as string***, *replacement* ***as string***)
  Replaces all occurrences of a string with another string.

EXAMPLE

```
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

Does a Software-Reset. All memory (SRAM) Variables will be reset, Interrupts tuned off and a Restart at address 0x0000 executed.

Syntax: ***Reset***

Example:

```
Reset
```

Return

Returns from a Subroutine (Method).

Syntax 1 (in Functions): ***Return*** *Expression*

Syntax 2 (in Procedures): ***Return*** See also: MethodsExample:

```
  Function test(a as byte)
    return a  ' Returns with a value
  endFunc

Procedure test1(a as byte)
    if a>0 then
      Return  ' immediate exit of the Procedure
    else
      Print str(a)
    end if
  endProc
```

Right()

Returns the right part of a String starting at a given position.

Syntax: *String* = ***Right***(*String,Position)*Position: Starting with 1 from right

Example:

```
dim s as string
s = Right("Hello World",4)  ' output: "orld"
```

Rinstr()

Find a Substring within a String *starting from the end*. The search is case sensitive.

Syntax: *Position* = ***Rinstr****(Source,Find)*

- **Position:** Starting with 1 from left, 0 = not found
- **Source:** String to search in
- **Find:** String to find

Example:

```
dim s as string
dim pos as byte
s = "Hello World"
pos = rinstr(s,"l")    ' output: 3
pos = rinstr(s,"ld")  ' output: 10
```

Seed, Rnd()

Returns a Random number between 0 and 255.LunaAVR's Pseudo Random Number Generatoris based on a Linear feedback shift register (LFSR).You must initialize the generator with **Seed** prior to your *first* call to Rnd(). A sporadic call to **Seed** increases the probability of a random number. There is a chance of returning a zero value when the LFSR jumps to the idle state. It is therefore recommended to vary the position and Seed starting values in your code until you get a constant and evenenly spread random value.

Syntax: ***Seed*** *Expression*

Syntax: *result* = ***Rnd***()

Example:

```
[..]
' Load the Random Generator with a zuerst Zufallszahlengenerator mit einem arbitrary starting value
Seed 11845
' permanently output random numbers
do
  print str(Rnd())
  waitms 200
loop
```

Seed16, Rnd16()

Returns a Random number between 0 and 65,535.

The Pseudo Random Number Generator is based on a Linear feedback shift register (LFSR).You must initialize the generator with **Seed16** prior to your *first* call to Rnd(). A sporadic call to **Seed16** increases the probability of a random number. There is a chance of returning a zero value when the LFSR jumps to the idle state. It is therefore recommended to vary the position and Seed starting values in your code until you get a constant and evenenly spread random value.

Syntax: ***Seed16*** *Expression*
Syntax: *result =* ***Rnd16****()*

EXAMPLE

```
[..]
' Load the Random Generator with a arbitrary starting value
Seed16 0x1a74
' permanently output random numbers
do
  print str(Rnd16())
  waitms 200
loop
```

Seed32, Rnd32()

Returns a Random number between 0 and 4,294,967,295.

The Pseudo Random Number Generator is based on a Linear feedback shift register (LFSR).You must initialize the generator with **Seed32** prior to your *first* call to Rnd(). A sporadic call to **Seed32** increases the probability of a random number. There is a chance of returning a zero value when the LFSR jumps to the idle state. It is therefore recommended to vary the position and Seed starting values in your code until you get a constant and evenenly spread random value.

Syntax: ***Seed32*** *Expression*
Syntax: *result* = ***Rnd32****()*

EXAMPLE

```
[..]
' Load the Random Generator with a arbitrary starting value
Seed32 0x1a74f9d3
' permanently output random numbers
do
  print str(Rnd32())
  waitms 200
loop
```

Rol

Logical Bit-Rotation too the left.

Syntax: ***Rol*** *Variable, Bits*

Example:

```
dim a as integer
Rol a, 4  // Bitwise Rotate by 4 Bit to left
```

Ror

Logical Bit-Rotation to the right.

Syntax: ***Ror*** *Variable, Bits*

Example:

```
dim a as integer
Ror a, 4  // Bitwise Rotate by 4 Bits to right
```

Select-Case-Default-endSelect

Fast conditional branching and program execution with Constants comparison. If comparison is True, the code of this section is executed and then continued at the end of the Structure.

Syntax:

- ***Select Case*** *Expression*
- ***Case*** *Konstante[, ConstantN]*
  - Program code when comparison is TRUE
- *[**Case** Konstante[, ConstantN*
  - Program code when comparison is TRUE
- ***[Default]***
  - Program code when all other comparisons are FALSE.
- ***endSelect***

As of 2013.R1 addition range check is possible:

```
select case a
case 100 to 200,33,600 to 1000
  ...
end select
```

EXAMPLES

```
  dim a,b as byte
  dim c as word
  dim d as long

' Sometimes it may make sense to define a Data Type, such as Byte()
'  select case Byte(a+b)
  select case (a+b) 'Expressions permitted
  case 1
    ' Program Code
  case "A","B"   ' also Strings permitted (1 character)
    ' Program Code
  case 2,3,0b10101010
    ' Program Code
  case 0x33
    ' Program Code
  default
    ' Program Code
  endselect

  'Word comparison
  select case c
  case 1
    ' Program Code
  case "AB","xy"   ' also Strings permitted, as of build 3815 (2 characters)
    ' Program Code
  default
    ' Program Code
  endselect

 'Long comparison
  select case c
  case 1
    ' Program Code
  case "ABCD","wxyz"   ' also Strings permitted, as of build 3815 (4 characters)
    ' Programmcode
  default
    ' Program Code
  endselect
```

```
  dim s as string
  select case s
  case "A"
    ' Program Code
  case "B","C","D"
    ' Program Code
  case "Super!"
    ' Program Code
  default
    ' Program Code
  endselect
```

Sin()

Sin() is a fast Sinus-Function. It expects an angle in degrees multiplied by 10. The Result is a 16 Bit Integer (-32768 to +32767) equivalent to the Sinus value (-1 to +1). The function the Cordic Algorithm to calculate.

Syntax: ***ergebnis*** = ***Sin****(Expression)*

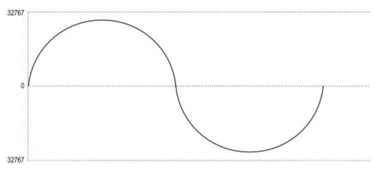
Example:

```
dim angle as byte
dim result as integer
winkel=23
result = Sin(angle*10)
```

Sine()

Sine() is a fast 16 Bit, 360° Wave Function. It expects an angle in degrees (0-359). The Result is a 16 Bit Integer (-32768 to +32767) as the equivalent to the angle.

Syntax: ***result*** = ***Sine***(*Expression*)Use Sine to create geometric or mathematical Sinewaves because no individual Sinus/Cosine is required. The calulation is upto 10x faster than the classic method because it uses a Lookup table. Only whole angles are permitted.



Angle	Result
0	0
45	23171
90	32767
180	0
270	-32768

Example:

```
dim angle as byte
dim result as integer
angle=90
result = Sine(winkel)
```

See also: Fsine()

Spc()

creates a String with the requested number of Spaces (ASCII 32).

Syntax: *String* = ***Spc****(Number)*

Example:

```
dim s as string
s = Spc(8)  ' ouutput: "        "
```

Sram

SRAM is the "Memory" object of the superior class "Avr".

Properties (read only)		
Name	**Description**	**Returns**
.StartAddr oder .Addr	Phys. Start address of the available memory.	word
.endAddr	Phys. End address of the available memory.	word
.Length	Byte size of the available memory.	word
.DynAddr	Start address of the available dynamic memory.	word
.DynLength	Byte size of the available dynamic memory.	word
.Space	Byte size of the unused dynamic memory.	word

Methods (call)	
Name	**Description**
.ClearAll	Clear usable memory (fill with Null-Bytes).[1]
.Dump	Hex dump of the whole physicly accessiblememory to the first serial port. [2]
.DumpBlocks	Hex dump of all currently allocated Memory Blocks. [2]

Direct access to SRAM

The direct access to memory is similar to the Object Flash- or eeprom. Gives you "manual" access to memory. e.g. you have access to Variables and the whole Stack (the whole memory). Write to memory only when you do not use Luna's Memory-Block and String functions, because these function mange memory themselves. Manual memory mangement may make sense on Controller with limited memory.

Methods (read/write)		
Name	**Description**	**Rückgabetyp**
.ByteValue(offset)	Byte read/write	byte
.WordValue(offset)	Word read/write	word
.IntegerValue(offset)	Integer read/write	integer
.LongValue(offset)	Long read/write	long
.SingleValue(offset)	Single read/write	single
.StringValue(offset,bytes)	String read/writeincluding Length	string
.PString(offset)	Pascal-String read/write (Start-Byte is the Length)	string
.CString(offset)	C-String read/write (Null terminated)	string

Example 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  ' complete erase (fill with Null-Byte)
```

Example 2:

```
avr.device = atmega32
avr.clock = 20000000          ' Quartz
avr.stack = 32                ' Bytes for Program stack (Default: 16)
uart.baud = 19200             ' Baudrate
uart.Recv.enable              ' activate transmit
uart.Send.enable              ' activate receive
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="hello"
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) Including Variabls, Objects and Stack! This is executed automatically at the very beginning of every program.

2) Debug function, requires additional space in Flash.

Str()

Convert a number into a String. Allows for the Data-Type.

Präprozessor	The function is also available in the Preprocessor, and does not generate any machine code because only constants are used.

Syntax: *String* = ***Str****(Expression)*

Example:

```
dim s as string
s = Str(-23.42)  ' output: "-23.42"
```

StrFill()

Creates a String with the source string in the requested number of repetitions.

Preprocessor	The Function is also available in the Preprocessor, and does not generate any machine code because only constants are used.

Syntax: *String* = ***StrFill***(*Source,Number*)

- **Source:** String to fill
- **Number:** number of repetitions

Example 1:

```
dim a as string

a = StrFill("Hello",3)     ' output: "HelloHelloHello"
```

Example 2:

```
dim char as byte
dim a as string

char=65                        ' ASCII character "A"
s = StrFill(chr(char),5)       ' output: "AAAAA"
```

String (Variable)

A String is a chain of any characters. Any type of alphabetic or numeric Informationen can be saved as a String. "Heinz Erhardt", "13.11.1981", "23.42" are examples of Strings. LunaAVR also allows you to put binary data in a String such as Null-Bytes.In the source code, include your String in Qquotes. The maximum length in LunaAVR is 254 Byte. The default String content is "" (empty). LunaAVR stores strings in the Pascal-Format, embedded in a MemoryBlock-Object.Strings require at least 2 Byte in memory (Pointer to the MemoryBlock-Object). eeprom-Strings require the number of static Byte defined in the eeprom. A RAM Memory String is a 16-Bit Pointer to a MemoryBlock of dynamic size.

Properties (read only)		
Name	**Description**	**Return Type**
.Addr	Address of the Variable in Memory	word
.Len	Read String length (number of characters)	byte
.Ptr	Address of the allocated Memory-Blocks	word
Methods (read/write)		
Name	**Description**	**Return Type**
.ByteValue(offset)	individual Byte [1]	byte

- **offset:** Byte-Position within the allocated Memory-Block, zero based. The first Byte is the String length identifier (offset = 0).

Notice
You may violate boundaries if exception is not declared. Boundary violation is not tested (faster).

Example

```
dim a as bte
dim s as string
s="Hello"
a=s.ByteValue(0) ' Read Length Byte, output: 5
s=s.ByteValue(1) ' Read 1. character of text, output: 72 (Ascii-"H")
```

[1] an empty String returns "0"

SoftUart

NOTE	This article describes a (discontinued) internal module/interface. As of version 2013.r6 available as External Library, see Libraries

Is LunaAVR's Software-Uart-Interface and ea simplified Uart-Emulation. Any Port-Pin can be assigned to RX/TX. SoftUart uses 1 Start and 1 Stop-Bit (Standard-Configuration). The Baudrate can be set freely. Similar to the Hardware-Uart, you must watchout for the error rate. You should select a Baudrate with a low error rate. Baudrats upto about 1/100 of the CPU Frequency are possible (200000 Baud @ 20 Mhz).
Initialization is done once and is globally valid. Modification to initialization or Pins is not possible at runtime. Initialization must take place textually before any calls are made. The Preprocessor uses constants to optimize speed.

Properties(write only)		
Name	**Description**	**Value**
.Mode =	Signal Mode	normal, inverted
.PinRxd =	Rxd-Pin (Receive) define	Port.n
.PinTxd =	Txd-Pin (Transmit) define	Port.n
.Baud =	Baudrate define	Constant
.AdjustDelay =	Bit-Time adjust[1)]	Constant

Methods			
Name	**Description**	**Type**	**r/w**
.Print	Transmit several values, see Print.	-	write
.InpStr([*echo*,[*breakChar*)	Read max. 254 characters from the port and exits when breakChar (Default: 13) is encountered. If *echo* = 0 (Default: 1), echoing will be suppressed. Both paramters are optional.	string	read
.ReadByte	Byte read (waits for character)	byte	read
.ReadWord	Word read (waits for character)	word	read
.ReadInteger	Integer read (waits for character)	integer	read
.ReadLong	Long read (waits for character)	long	read
.ReadSingle	Single read (waits for character)	single	read
.Read *sramAdresse,Anzahl*	Read number of Byte and save to memory (waits until all characters are read).	-	read
.ReadC *sramAdresse*	Read number of Byte and save to memory (Waits until a Null-Byte is read. The Null-Byte is discarded.).	-	read
.ReadP *sramAdresse*	Read number of Byte and save to memory (The first Byte received holds the number of Byte to read. Waits until all characters are read.).	-	read
.WriteByte	Byte write (waits until transmission is ready)	byte	write
.WriteWord	Word write (waits until transmission is ready)	word	write
.WriteInteger	Integer write (waits until transmission is ready)	integer	write
.WriteLong	Long write (waits until transmission is ready)	long	write
.WriteSingle	Single write (waits until transmission is ready)	single	write
.Write *sramAdresse,Anzahl*	Transmit a number of Byte from memory (waits until all characters have been sent).	-	write
.WriteC *sramAdresse*	Transmit a number of Byte from memory until a Null-Byte is encountered(waits until all characters have been sent including Null-Byte).	-	write
.WriteP *sramAdresse*	Transmit a number of Byte from memory, the first Byte denotes the number of Byte to transmit(waits until all characters have been sent.	-	write
.CWrite *flashAdresseBytes,Anzahl*	Transmit a number of Byte from Flash (waits until all characters have been sent).	-	write
.CWriteC *flashAdresseBytes*	Transmit a number of Byte from Flash until a Null-Byte is encountered (waits until all characters have been sent except the Null-Byte).	-	write
.CWriteP *flashAdresseBytes*	Transmit a number of Byte from Flash, the first Byte denotes the number of Byte to (waits until all characters have been sent.	-	write
.Dump	Hexdump for debuggin puposes, see Dump.	-	call

Notice
Emulation takes place in the software, therefore timing is very important. A interrupt during transmission will influence the timing.

Example

```
avr.device = attiny2313
avr.clock = 20000000
avr.stack = 4

' SoftUart Example: simple terminal
' Setup Pins you want to use and baud rate first
SoftUart.PinRxd = PortB.0    ' Receive Pin: RXD
SoftUart.PinTxd = PortB.1    ' Transmit Pin: TXD
SoftUart.Baud = 19200        ' Baud rate up to 1/100 of avr.clock (200000 baud @ 20 Mhz)

dim a as byte

SoftUart.Write 12            ' clear terminal
SoftUart.Print " > ";        ' the prompt
do
  a=SoftUart.Read            ' wait and get byte from input
  select case a
  case 13
    SoftUart.Write 13        ' CR
    SoftUart.Write 10        ' LF
    SoftUart.Print " > ";    ' the prompt
  default
    SoftUart.Write a
  endselect
loop
```

1) Time in µs that will be subtracted from or added to the Bit-Send Time. This property must be set before setting .baud. Avoids any possible transmission dilatation.

Sqrt()

Sqrt returns the square root of an Integer-Value.

Syntax: ***result*** = ***Sqrt****(Expression)*

Example:

```
dim a,b as word
b=8
a=b^2          ' output: 64
a = Sqrt(a)   ' output: 8
```

See also: Fsqrt

Swap

Swaps SRAM-Variables and Array-Elements. Use the command to swap ***two different*** Variables ***or*** the ***Low/High-Values*** of a Variable.

Syntax 1: - swaps Low/High-Values dependent on Data-Type

- ***Swap*** *Variable*

Syntax 2: - swaps 2 Variables

- ***Swap*** *Variable1, Variable2*

Example for 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)   ' output: "a = 0xAB"
print "b = 0x";hex(b)   ' output: "a = 0xAABB"
print "c = 0x";hex(c)   ' output: "a = 0xAABBCCDD"

swap a   ' swaps Low/High-Nibble
swap b   ' swaps Low/High-Byte
swap c   ' swaps Low/High-Word

print "a = 0x";hex(a)   ' output: "a = 0xBA"
print "b = 0x";hex(b)   ' output: "a = 0xBBAA"
print "c = 0x";hex(c)   ' output: "a = 0xCCDDAABB"
```

Example for Syntax 2:

```
dim a,b,v(4) as byte
dim c,d as word

a=1
b=2
c=3
d=4
' swap Variables
swap a,b        ' swaps values of a and b
swap c,d        ' swaps values of c and d
swap v(2),v(4)  ' swaps values of array elements 3 and 5
```

Trim

Removes whitespaces on start and end of a string, optionalthe chars given by a constant string or data object.

Whitespaces means: 0,9,32,13,10

Syntax: *string* = ***Trim***(*Expression*, *[trimChars]*)

- **trimChars**: optional constant string or data object (first byte is number of bytes) containing the chars do you want to trim.

Example

```
dim s as string

s = "   hello  "
s = Trim(s)  ' result: "hello"
```

Upper()

Converts a string to upper characters, including all umlaut.

Syntax: *String* = ***Upper***(*String*)

Example:

```
dim s as string
s = Upper("abcdefg")  ' output: "ABCDeFG"
```

Val()

Convert a string holding a decimal number into a numeric value. The result is 32 Bit signed (longint/int32). Ignores leading spaces.

Syntax: *int32* = ***Val****(text as string)*

Example:

```
dim a as integer
dim b as longint
a = Val("12345")
b = Val("12345")
```

Void

Void is a keyword for a function call. The return value is discarded by this keyword.

Void() is a dummy method. No code would created and is useful as Placeholder for e.g. conditional used defines.

Syntax: ***Void*** *Functionname(Parameter)*

EXAMPLE

```
dim a as word

a=setChannel(0)
void setChannel(2)
[..]

function setChannel(chan as byte) as word
  if chan then
    adc.channel = chan
  end if
 return adc.Value
endfunc
```

EXAMPLE DUMMY METHOD

```
  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
```

See also: Methods

Wait, Waitms, Waitus

Waits a certain time before continuing program execution. Program execution is blocked completely during the wait, excluding Interrupts. They run anyway.**it requires a korrektly defined system clock speed Avr.Clock.**The precision of the Wait-Function **when using Constant as parameters** lies at around **1 µs** 1).

Syntax:

1. ***wait*** *Expression* (Secunds)
2. ***waitms*** *Expression* (Milliseconds)
3. ***waitus*** *Constant* (Microseconds)

As of **Version 2012r2**, you may also call **wait** and **waitms** with variables. Precision is then around **100 µs** 2).
Example:

```
dim a a byte
a=200
wait 3      ' wait 3 seconds
waitms 100  ' wait 100 ms
waitus 100  ' wait 100 µs
waitms a    ' wait a variable time, time to wait in a
```

1) 20 MHz Controller. The lower the clock speed, the lower the wait time accuracy

2) 20 MHz Controller. The lower the clock speed, the lower the wait time accuracy.

When-Do

Conditional branching and program execution, one line only.

Syntax:

- ***When*** *Expression* ***Do/Then*** *Expression*

you may use "Do" or "Then" as the second keyword.

EXAMPLE

```
[..]

when a>b or a=100 do PortB.0 = 1

' above Syntax is the same as:
if a>b or a=100 then
  PortB.0 = 1
endif
```

While-Wend

Lopp with a starting condition when entering or exiting. The While-Wend-Loop will be entered and executed continuously as long as the condition is True.

Syntax:

- ***While*** *Expression*
  - *Program Code*
- ***Wend***

Example

```
dim i as integer
i=10
While i>0
  Decr i
  Print "Hello"
Wend
```

Assembler Commands

Overview of the luna assembler (lavra) supported avr assembler commands. The luna assembler is part of the luna compiler. Not all commands are available for every controller (see datasheet).

abbr.l	description
r	source-/destination register
rh	upper source-/destination (R16-R31)
rd	double register R24:25(W), R26:27(X), R28:29(Y), R30:31(Z)
rp	pointer register X,Y, Z
ry	pointer register Y, Z
p	port
pl	port at lower adress 0 bis 31
b7	bitnumber 0 bis 7
k63	constant 0 bis 63
k127	constant -64 bis +63
k255	constant 0-255
k4096	constant -2048 bis +2047
k65535	constant 0 bis 65535

group	function	command	flags	clk
empty command	No Operation	NOP		1
power management	Sleep	SLeeP		1
watchdog	Watchdog Reset	WDR		1
set register	0	CLR r	Z N V	1
	255	SeR rh		1
	constant	LDI rh,k255		1
copy	register >> register	MOV r,r		1
	SRAM >> register, direct	LDS r,k65535		2
	SRAM >> register	LD r,rp		2
	SRAM >> register with 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
	program memory(Z) >> R0	LPM		3
	program memory(Z) >> register	LPM r,Z		3
	program memory(Z) mit INC >> register	LPM r,Z+		3
	program memory(RAMPZ:Z)	eLPM		3
	register >> SRAM, direkt	STS k65535,r		2
	register >> SRAM	ST rp,r		2
	register >> SRAM with 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, constant	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

subtraction	8 Bit, constant	SUBI rh,k255	Z C N V H	1
	8 Bit - Carry	SBC r,r	Z C N V H	1
	16 Bit, constant	SBIW rd,k63	Z C N V S	2
	8 Bit - Carry, constant	SBCI rh,k255	Z C N V H	1
multiplication	unsigned integer	MUL r,r	Z C	2
	signed integer	MULS r,r	Z C	2
	unsigned/unsigned integer	MULSU r,r	Z C	2
	unsigned floating point	FMUL r,r	Z C	2
	signed floating point	FMULS r,r	Z C	2
	signed/unsigned floating point	FMUL r,r	Z C	2
rotate	logical, left	LSL r	Z C N V	1
	logical, right	LSR r	Z C N V	1
	rotate, left with carry	ROL r	Z C N V	1
	rotate, right with carry	ROR r	Z C N V	1
	arithmetc, right	ASR r	Z C N V	1
	swap nibbles	SWAP r		1
binary	and	AND r,r	Z N V	1
	and, constant	ANDI rh,k255	Z N V	1
	or	OR r,r	Z N V	1
	or, constant	ORI rh,k255	Z N V	1
	exclusive-or	eOR r,r	Z N V	1
	ones complement	COM r	Z C N V	1
	twos complement	NeG r	Z C N V H	1
bit change	register, set	SBR rh,k255	Z N V	1
	register, reset	CBR rh,255	Z N V	1
	register, copy to T-Flag	BST r,b7	T	1
	register, copy to T-Flag	BLD r,b7		1
	port, set	SBI pl,b7		2
	port, reset	CBI pl,b7		2
status bits	Zero-Flag	SeZ	Z	1
	carry flag	SeC	C	1
	negative flag	SeN	N	1
	twos complement overflow flag	SeV	V	1
	half carry 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
	negative flag	CLN	N	1
	twos complement overflow flag	CLV	V	1
	half carry flag	CLH	H	1
	Signed Flag	CLS	S	1
	Transfer Flag	CLT	T	1
	Interrupt enable Flag	CLI	I	1
compare	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
	jump relativ	RJMP k4096		2
	jump adress absolute	JMP k65535		3
	jump adress implicit (Z)	IJMP		2

branchg	jump adress implicit (eIND:Z)	eIJMP		2
	subroutine, relative	RCALL k4096		3
	subroutine, adress absolute	CALL k65535		4
	subroutine, adress implicit (Z)	ICALL		4
	subroutine, adress implicit (eIND:Z)	eICALL		4
	return from subroutine	ReT		4
	return from interupt	ReTI	I	4
conditional branch	status bit set	BRBS b7,k127		1/2
	status bit reset	BRBC b7,k127		1/2
	jump if equal	BReQ k127		1/2
	jump if not equal	BRNe k127		1/2
	jump if overflow	BRCS k127		1/2
	jump if carry=0	BRCC k127		1/2
	jump if equal or greater	BRSH k127		1/2
	jump if less	BRLO k127		1/2
	jump if negative	BRMI k127		1/2
	jump if positive	BRPL k127		1/2
	jump if greater or equal(signed)	BRGe k127		1/2
	jump if less than zero (Vorzeichen)	BRLT k127		1/2
	jump if half carry	BRHS k127		1/2
	jump if half carry=0	BRHC k127		1/2
	jump if T-Bit is set	BRTS k127		1/2
	jump if T-Bit not set	BRTC k127		1/2
	jump if twos complement overflow	BRVS k127		1/2
	jump if twos complent flag=0	BRVC k127		1/2
	jump if interrupts enabled	BRIe k127		1/2
	jump if interrupts disabled	BRID k127		1/2
conditional jumps (skip next command if)	Register bit=0	SBRC rd,b7		1/2/3
	register bit=1	SBRS r,b7		1/2/3
	port bit=0	SBIC pl,b7		1/2/3
	port bit=1	SBIS pl,b7		1/2/3
	compare, jump if equal	CPSe r,r		1/2/3

Frequently Asked Questions

- **What does Luna cost?**
  The philosophy of programming suite Luna is to provide users an **free** innovative tool for software development. Only for **commercial use is a license purchase required. See also: Luna Licensing**

- **Is there any support?**
  Companies that use Luna commercially can book a support uncomplicated once or continuously. Also services for the development of classes or libraries can be commissioned. Furthermore, there is the forum for all users available for volunteer help each other

- **Can I sell my software commercially?**
  Yes, you are as an individual or as a company sell themselve developed programs and commercial applications (eg. Classes for the control of dedicated hardware, libraries ans so on). You define the license terms themselve.

- **Why is the documentation as a PDF file, rather than a Windows Help file available?**
  LunaAVR is not only available for the Windows platform, but also for Linux and Mac OS. A Windows Help file does not make sense on the other platforms. The PDF format is readable on all systems, also can be searched in the complete document, which eg. at a local HTML-based documentation would not be possible.

- **Is Luna a Basic?**
  No, it's like Pascal "Basic-like" and beginner-friendly, but not a simple Basic. Luna is an independent language which includes many practical elements from other languages.

Contact

Contact

- **LunaAVR Licensing/Donation**

DEVELOPER

Developing and programming by: Richard Gordon Faika (rgf software)
Mail: avr@myluna.de
Platforms: Windows, Linux, Mac-OS
http://www.rgfsoft.com
http://avr.myluna.de

TRANSLATIONS (ENGLISH)

Documentation/Wiki: by Matthias Walter (listrik) http://www.listrik.de