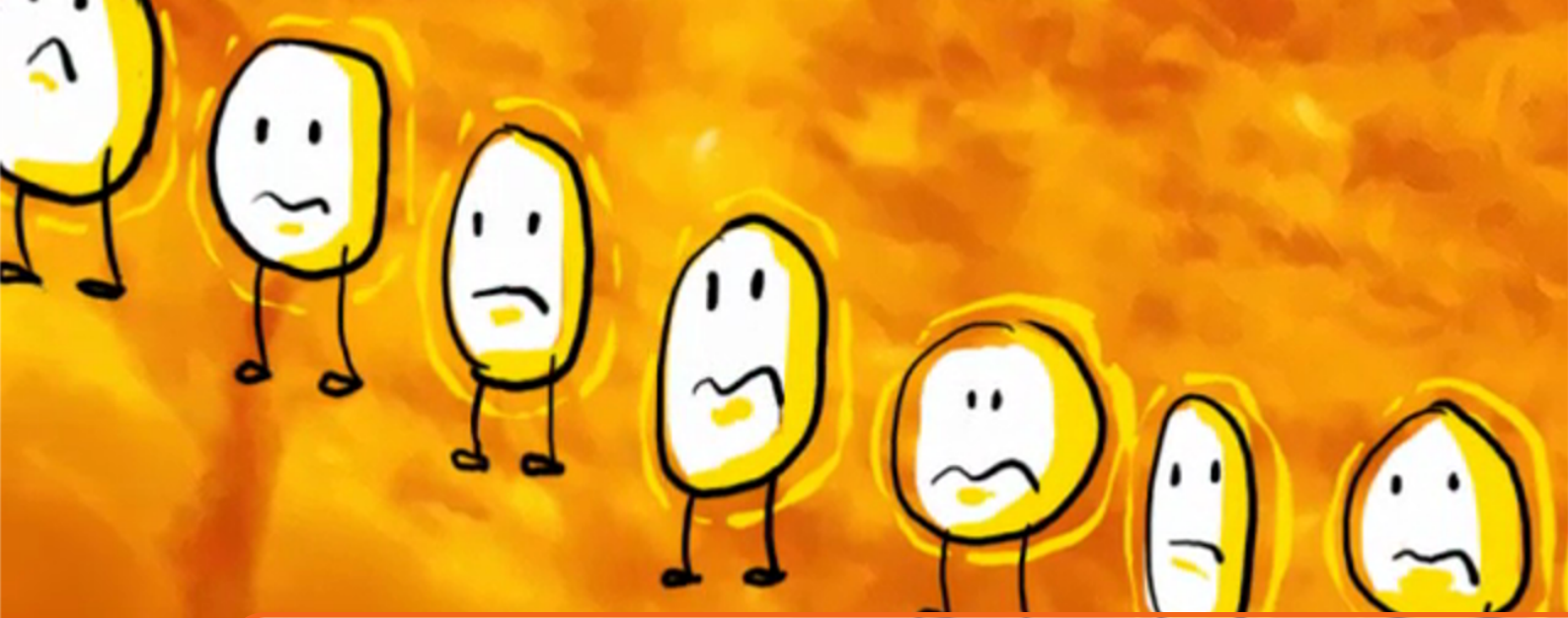
EMDO Energy Manager User Manual

Installation, Configuration, Basic Programming, Reference Applications

EMDO can do



Contents

I Part Main

1 Installation 7

1.1 EMDO101/102/103/104 7

1.1.1 Features 7

1.2 Pin Diagrams 9

1.2.1 EMDO101 9

1.2.2 EMDO102 10

1.2.3 EMDO103 13

1.2.4 EMDO104 13

1.2.5 Expansion boards 13

2 Webinterface 17

3 Reference Application 27

3.1 EMDO101 27

3.2 EMDO102 27

3.3 EMDO103 29

3.4 EMDO104 29

4 Basic Interpreter 33

4.1 Basic introduction 33

4.1.1 Variables 33

4.1.2 Loops 34

4.1.3 Functions 34

4.1.4 If . . . 35
4.1.5 Libraries . . . 36
4.1.6 Scheduler . . . 36
4.2 EMDO command and function overview 37
4.3 EMDO addon board commands 72
4.4 EMDO libraries 73
4.5 Sys API 74
4.6 Command Line Interface 81

5 Configuration . . . 103
5.0.1 Graphics . . . 103
5.0.2 Certificates . . . 103

II Appendix

6 Specification . . . 107

7 Credits and Licenses . . . 109
7.1 Developers 109

Bibliography . . . 111
7.2 References 111
References 111

Index . . . 113

Part Main

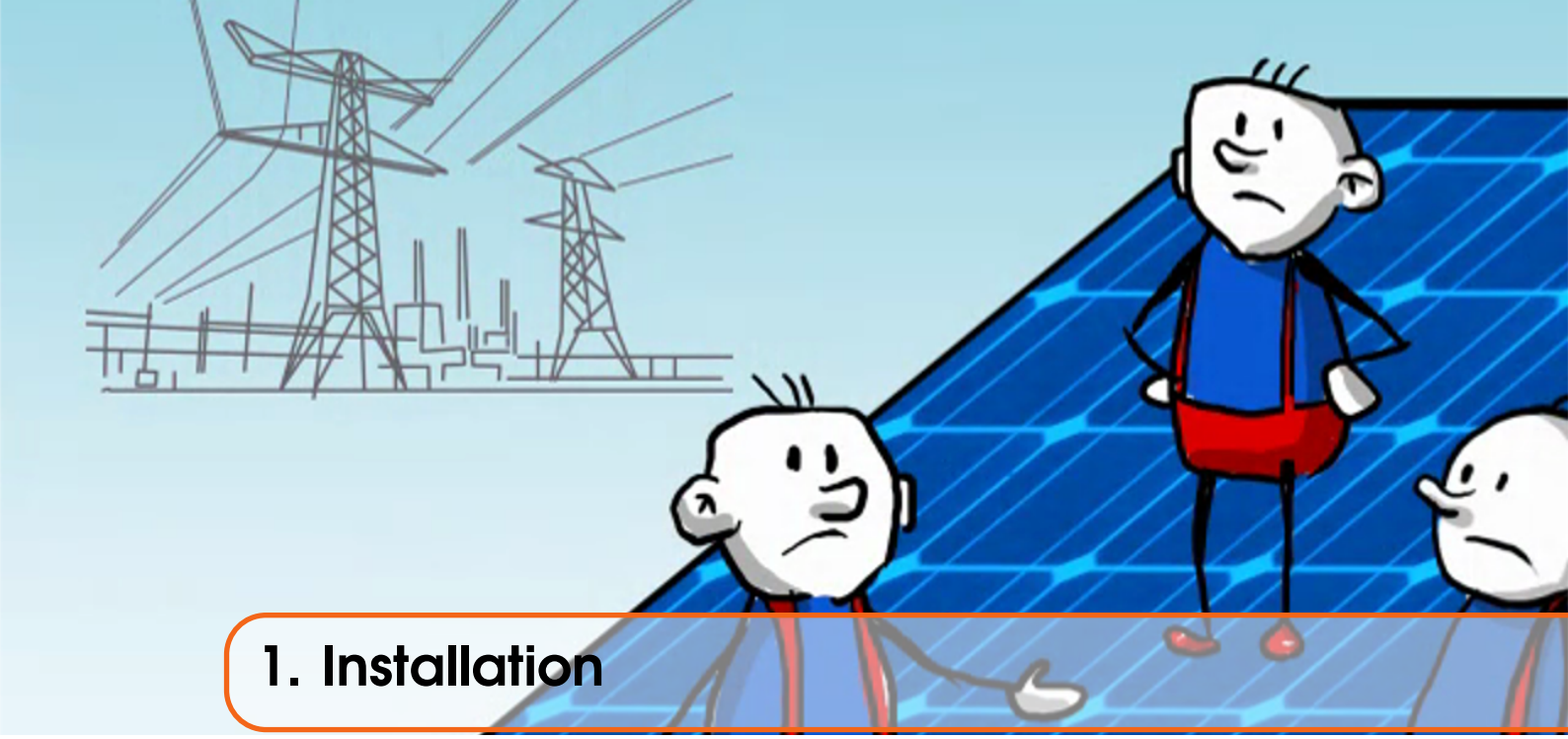
1 Installation 7
1.1 EMDO101/102/103/104
1.2 Pin Diagrams

2 Webinterface 17

3 Reference Application 27
3.1 EMDO101
3.2 EMDO102
3.3 EMDO103
3.4 EMDO104

4 Basic Interpreter 33
4.1 Basic introduction
4.2 EMDO command and function overview
4.3 EMDO addon board commands
4.4 EMDO libraries
4.5 Sys API
4.6 Command Line Interface

5 Configuration 103



1. Installation

1.1 EMDO101/102/103/104

1.1.1 Features

The EMDO device is a versatile tool which allows the connection of components such as inverters, chargers, home automation, heat pumps and heaters at your home. It is a versatile piece of electronics that glues everything in the energy infrastructure together. The EMDO has interfaces already built into the device to facilitate the variety of interconnections with external equipment. The following table lists the interfaces available on the EMDO range of devices:

	EMDO101	EMDO102	EMDO103	EMD104
D0 magnet mount	✓	-	-	-
DIN rail mount		✓	✓	✓
RS485	✓(1)	✓(2)	✓(1)	✓(1)
D0 interface	✓	Option available		
M-Bus interface			✓	
KNX interface				✓
Relay output driver	✓(1)	✓(2)	✓(2)	✓(2)
S0 optical input	✓			
S0 electrical inputs	✓(2)	✓(2)	✓(2)	✓(2)
S0 electrical outputs	✓(2)	✓(2)	✓(2)	✓(2)
Expansion Board connector	✓	✓	✓	✓
Board temperature sensor	✓	✓	✓	✓
Ethernet	✓	✓	✓	✓
USB for firmware update	✓	✓	✓	✓
USB Power supply	✓			
DIN Power supply		✓	✓	✓
Default reset button	✓	✓	✓	✓

1.2 Pin Diagrams

1.2.1 EMDO101

EMDO101 classic is directly mounted on a utility meter on the D0 Interface (IEC-62056-21). The following figure shows the D0 reading head on the enclosure 1.2.1. The device is mounted on the utility meter with the ethernet cables pointing toward the ground.

Figure 1.2.1: *EMDO101 D0 interface*

The EMDO101 is powered over the USB port. Ethernet is required for network connectivity. The status LEDs and firmware reset to default button are also accessible from the connector side. See figure 1.2.2 for details.

Figure 1.2.2: *EMDO101 ethernet power*

The figure 1.2.3 shows the push-in terminals. The pinout of the terminals is described in the table 1.2.1.

Figure 1.2.3: *EMDO101 terminal*

Table 1.2.1: *EMDO101 device pins*

Pin	Function	Description
1	S0 IN 1 +	S0 input with galvanic isolation (5-24V, external resistors required), see chapter 3 for details. This port can be used as gpio input or S0 signal counter from an energy meter. **Wrong external wiring might damage devices.**
2	S0 IN 1 -	
3	S0 IN 2 +	
4	S0 IN 2 -	
5	S0 OUT 1 +	S0 output with galvanic isolation (5-24V, external resistors required), see chapter 3 for details. This port can be used as gpio output or S0 output generator for an energy meter. **Wrong external wiring might damage devices.**
6	S0 OUT 1 -	
7	S0 OUT 2 +	
8	S0 OUT 2 -	
9	GND	external power supply up to 200mA
10	5V Out	
11	RS485 A	RS485 transceiver, see chapter 3 for details. **Wrong external wiring might damage devices.**
12	RS485 B	
13	Relay OUT	Relay driver output (5-24V), see chapter 3 for details. **Wrong external wiring might damage devices.**

1.2.2 EMDO102

EMDO102 is mounted on a DIN Rail (see figure 1.2.4). The EMDO102 has two RS485 ports. Optionally, the secondary RS485 port can be connected to an external D0 reading head (IEC-62056-21).

Figure 1.2.5 show the bottom connectors for power supply, 1 Wire and relay drivers (A). Ethernet is required for network connectivity.
USB is only for firmware update. The status LEDs and firmware reset to default button are accessible from inside the enclosure.

Figure 1.2.6 shows the top connectors (B and C).

Figure 1.2.4: *EMDO102 DIN rail mount*

Figure 1.2.5: *EMDO102 bottom*

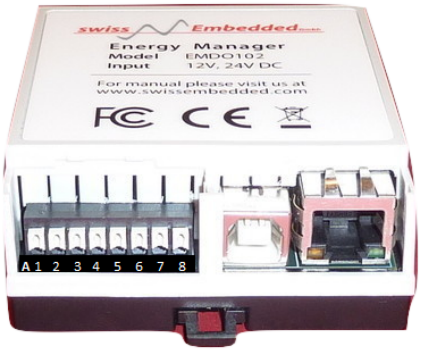


Figure 1.2.6: *EMDO102 top*

Table 1.2.2 describes the pinout of the terminals A, B and C. Refer to chapter 3 for Application.

Table 1.2.2: *EMDO102 device pins*

Pin	Function	Description
A1	24V IN (or 12V)	Voltage range 10-26V
A2	GND	
A3	GND	External power supply up to 0.8A (shared with other port)
A4	24V Out	
A5	Relay Out 1	Relay driver output (5-24V), see chapter 3 for details.
A6	Relay Out 2	**Wrong external wiring might damage devices.**
A7	GND (1 Wire only)	1-Wire transceiver, see chapter 3 for details.
A8	1 Wire	**Wrong external wiring might damage devices.**
B1	S0 In 1 +	S0 input with galvanic isolation (5-24V, external resistors required), see chapter 3 for details.
B2	S0 In 1 -	This port can be used as gpio input or S0 signal counter from an energy meter.
B3	S0 In 2 +	
B4	S0 In 2 -	**Wrong external wiring might damage devices.**
B5	5V Out	Internal fuse protected, approx. 0.8 A
B6	GND	
B7	RS485 A2	RS485 transceiver, see chapter 3 for details.
B8	RS485 B2	**Wrong external wiring might damage devices.**
C1	S0 Out 1 +	S0 output with galvanic isolation (5-24V, external resistors required), see chapter 3 for details.
C2	S0 Out 1 -	This port can be used as gpio output or S0 output generator for an energy meter.
C3	S0 Out 2 +	
C4	S0 Out 2 -	**Wrong external wiring might damage devices.**
C5	5V Out	Internal fuse protected, approx. 0.8 A
C6	GND	
C7	RS485 A	RS485 transceiver, see chapter 3 for details.
C8	RS485 B	**Wrong external wiring might damage devices.**

1.2.3 EMDO103

EMDO103 is connected to the meter over the MBUS interface. Table 1.2.3 shows the pinout. Refer to chapter 3 for Application.

Table 1.2.3: *EMDO103 device pins*

Pin	Function	Description
A1	24V IN (or 12V)	Voltage range 10-26V
A2	GND	
A3	GND	External power supply up to 0.8A (shared with other port)
A4	24V Out	
A5	Relay Out 1	Relay driver output (5-24V), see chapter 3 for details.
A6	Relay Out 2	**Wrong external wiring might damage devices.**
A7	GND (1 Wire only)	1-Wire transceiver, see chapter 3 for details.
A8	1 Wire	**Wrong external wiring might damage devices.**
B1	S0 In 1 +	S0 input with galvanic isolation (5-24V, external resistors required), see chapter 3 for details.
B2	S0 In 1 -	This port can be used as gpio input or S0 signal counter from an energy meter.
B3	S0 In 2 +	
B4	S0 In 2 -	**Wrong external wiring might damage devices.**
B5	5V Out	Internal fuse protected, approx. 0.8 A
B6	GND	
B7	MBUS A	M-Bus transceiver, see chapter 3 for details.
B8	MBUS B2	**Wrong external wiring might damage devices.**
C1	S0 Out 1 +	S0 output with galvanic isolation (5-24V, external resistors required), see chapter 3 for details.
C2	S0 Out 1 -	This port can be used as gpio output or S0 output generator for an energy meter.
C3	S0 Out 2 +	
C4	S0 Out 2 -	**Wrong external wiring might damage devices.**
C5	5V Out	Internal fuse protected, approx. 0.8 A
C6	GND	
C7	RS485 A	RS485 transceiver, see chapter 3 for details.
C8	RS485 B	**Wrong external wiring might damage devices.**

1.2.4 EMDO104

EMDO104 is connected to the meter over KNX bus. Table 1.2.4 shows the pinout. Refer to chapter 3 for Application.

1.2.5 Expansion boards

Expansions board are available for the EMDO for a various radio standards:

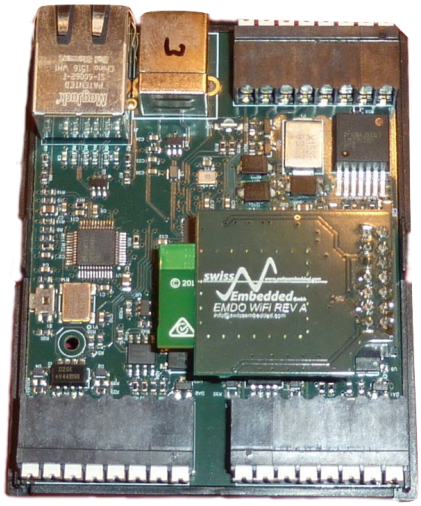
Table 1.2.4: *EMDO104 device pins*

Pin	Function	Description
A1	24V IN (or 12V)	Voltage range 10-26V
A2	GND	
A3	GND	External power supply up to 0.8A (shared with other port)
A4	24V Out	
A5	Relay Out 1	Relay driver output (5-24V), see chapter 3 for details.
A6	Relay Out 2	**Wrong external wiring might damage devices.**
A7	GND (1 Wire only)	1-Wire transceiver, see chapter 3 for details.
A8	1 Wire	**Wrong external wiring might damage devices.**
B1	S0 In 1 +	S0 input with galvanic isolation (5-24V, external resistors required), see chapter 3 for details.
B2	S0 In 1 -	This port can be used as gpio input or S0 signal counter from an energy meter.
B3	S0 In 2 +	
B4	S0 In 2 -	**Wrong external wiring might damage devices.**
B5	5V Out	Internal fuse protected, approx. 0.8 A
B6	GND	
B7	KNX A	KNX transceiver, see chapter chapter 3 for details.
B8	KNX B	**Wrong external wiring might damage devices.**
C1	S0 Out 1 +	S0 output with galvanic isolation (5-24V, external resistors required), see chapter 3 for details.
C2	S0 Out 1 -	This port can be used as gpio output or S0 output generator for an energy meter.
C3	S0 Out 2 +	
C4	S0 Out 2 -	**Wrong external wiring might damage devices.**
C5	5V Out	Internal fuse protected, approx. 0.8 A
C6	GND	
C7	RS485 A	RS485 transceiver, see chapter 3 for details.
C8	RS485 B	**Wrong external wiring might damage devices.**

- LoRaWAN
- WMBUS
- Enocean
- Zigbee
- WiFi

For installation or deinstallation of the expansion board, disconnect power first, open EMDO enclosure by removing the two black skews on the D0 read head side (EMDO101) or remove cover by pushing the two black snap in (DIN rail enclosure EMDO102-104).
Put the expansion board on the connector as shown in the figure 1.2.7. Make sure the connectors are correctly aligned. Close the enclosure again.
The radio module performance depends on the environment. It is important that electromagnetic radiation is not obscured by walls, metal enclosures,etc.

Figure 1.2.7: *EMDO102 expansion boards*



2. Webinterface

The EMDO comprises of a highly versatile energy system manager, accessible through a web interface. It is used to configure how the EMDO should work in the renewable energy system. The interface contains all the necessary components for setting up, controlling and tracking energy production and consumption.

The EMDO has a built-in DHCP client. If no DHCP server is not available in the network, a configurable fixed IP address can be used. If the settings are invalid and EMDO can't be reached at the configured fixed IP, the EMDO can be reset to factory defaults.

Access to the web interface can be password protected, per default there is password active.

The following figure 2.0.1 shows the start page.

The EMDO device has a build in 32MByte flash with file system. Data from various sources can be logged at runtime and stored for years on the flash. It highly depends on the logging frequency and amount of data per log entry. The flash is self contained and garbage collection is executed automatically in the background. The graphical user interface is configurable by the user. The following figure 2.0.2 shows the default configuration of the EMDO.

The configuration of the graphical user interface and the basic program that runs on the EMDO are both configurable in an editor. The figure 2.0.3 shows the editor tab. Library and basic programs can be uploaded in this tab. The program execution can be stopped, paused and restarted. It is also possible to slow down program execution to better check the program flow. Variable can be monitored at program runtime. Details about the configuration are described in chapter 5.0.1.

The following configuration menu show the current system settings of the hardware. The settings can be overwritten in BASIC at runtime or can be changed from web interface. It is important to ensure that setting changes during runtime in the web interface do not interfere with the BASIC

Figure 2.0.1: *Webinterface Welcome*

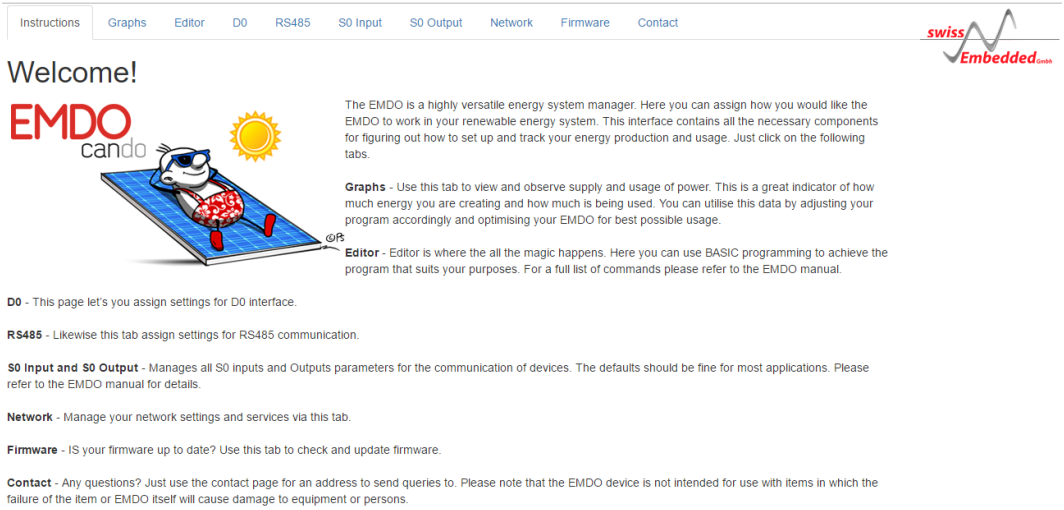
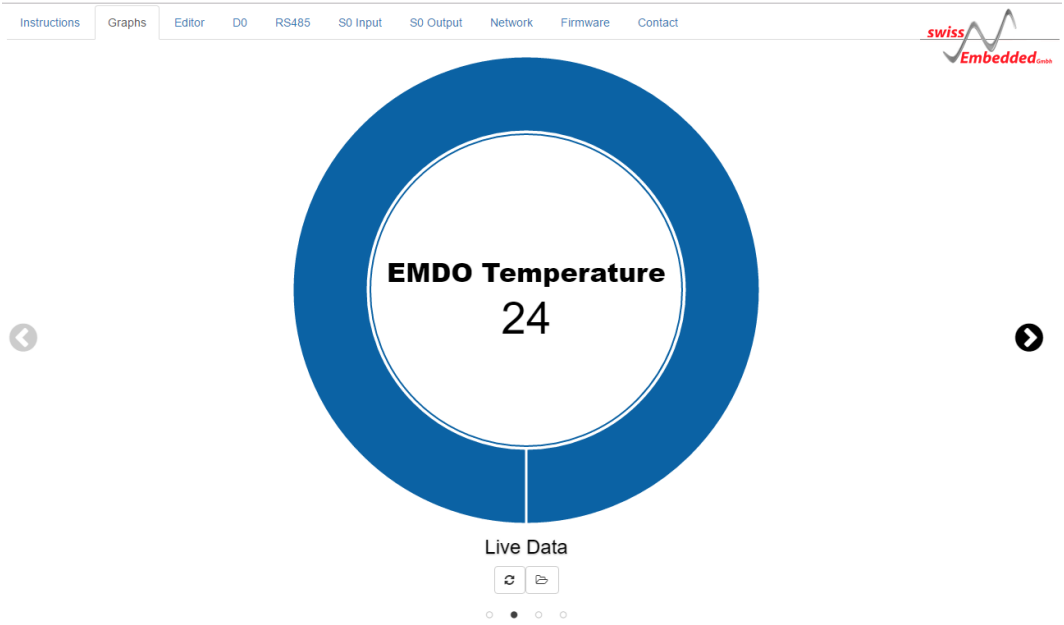


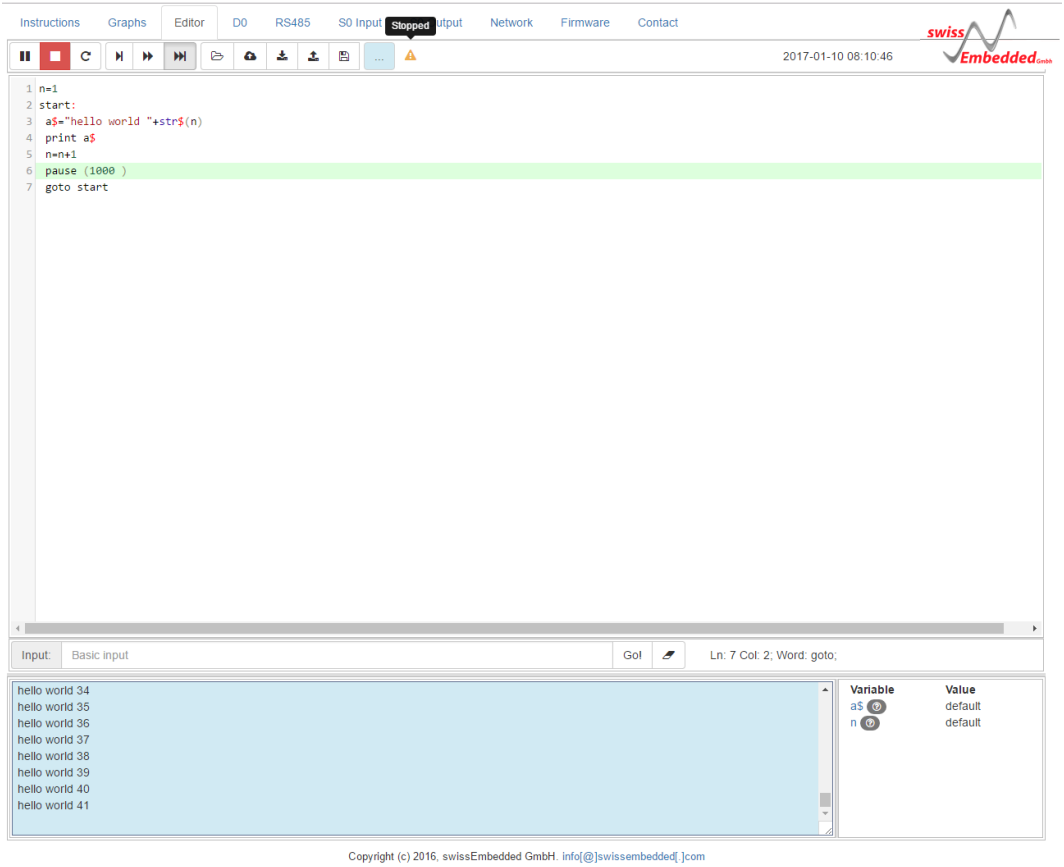
Figure 2.0.2: *Webinterface Graphics*



program. The following figures show the settings: D0 (figure 2.0.4), RS485 (figure 2.0.5), S0 Input (figure 2.0.6), S0 Output tab (figure 2.0.7).

The network menu setting (figure 2.0.8) allow the configuration of the DHCP client or static network ip (IP, netmask, gateway, DNS server), NTP server configuration (network time protocol), MQTT broker (mqtt.ednme.com), remote syslog server (rsyslog) and configuration for local or remote. In addition the RSH (remote shell), TCP to D0, RS485, MBUS, KNX network ports and Webinterface authentification can be enabled through this menu (disabled by default for security reasons).

Figure 2.0.3: *Webinterface Editor*



NTP is very important for the EMDO as many mechanism rely on it.

The firmware update menu shows information about various system properties and assembled add on boards (figure 2.0.9).
In case of any firmware problems, the System log might show any valuable information for the firmware development team and can be easy taken by simply clicking on the button.
Prior to any firmware update, it is important to get a snapshot of the system flash content. This includes all the configuration, basic program and logged data.
. The microprocessor firmware and the flash disk content can be updated independently. Update firmware first (.hex.gz), then the flash disk (.tar).

The contact menu shows the contact and support url (figure 2.0.10). In case of any problem it is important for engineering to get as much as possible information to reproduce the problem. Please try to send us the information as short as possible but with as much relevant information as possible. Credits to all the people contributing to the EMDO project can be found at chapter 7.

Figure 2.0.4: *Webinterface D0 interface*

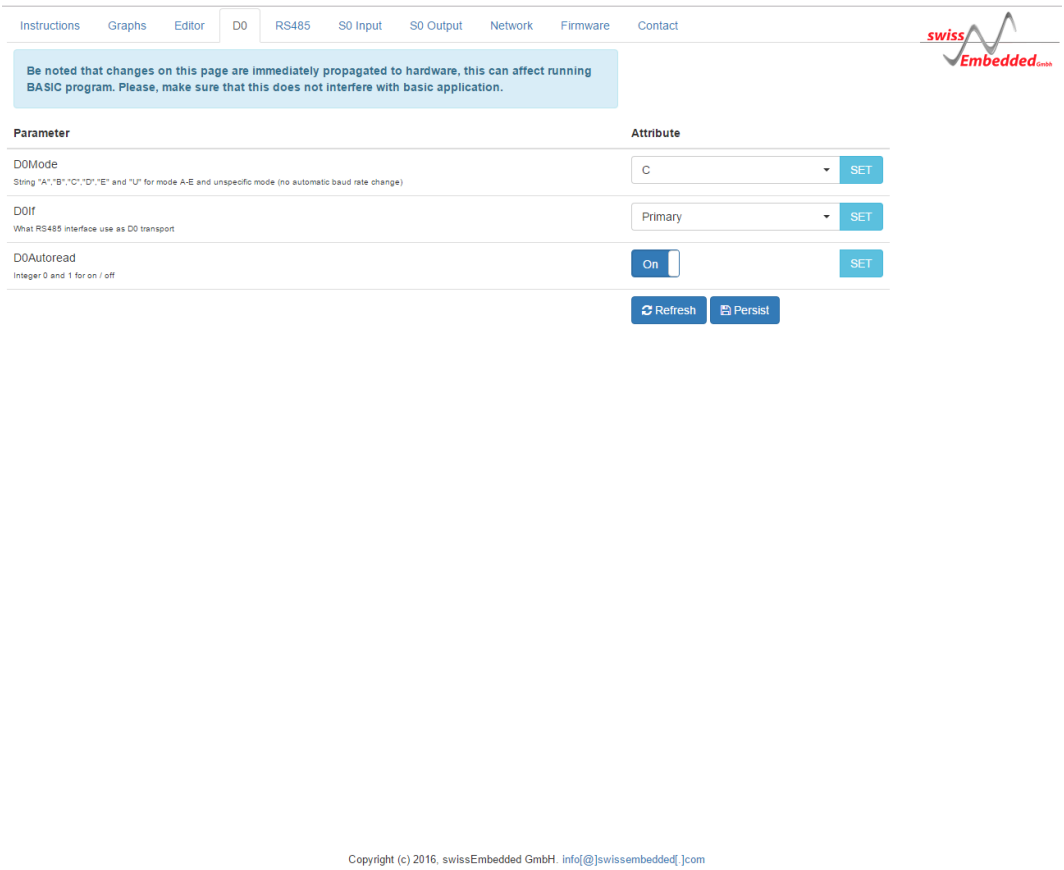


Figure 2.0.5: *Webinterface RS485 interface*

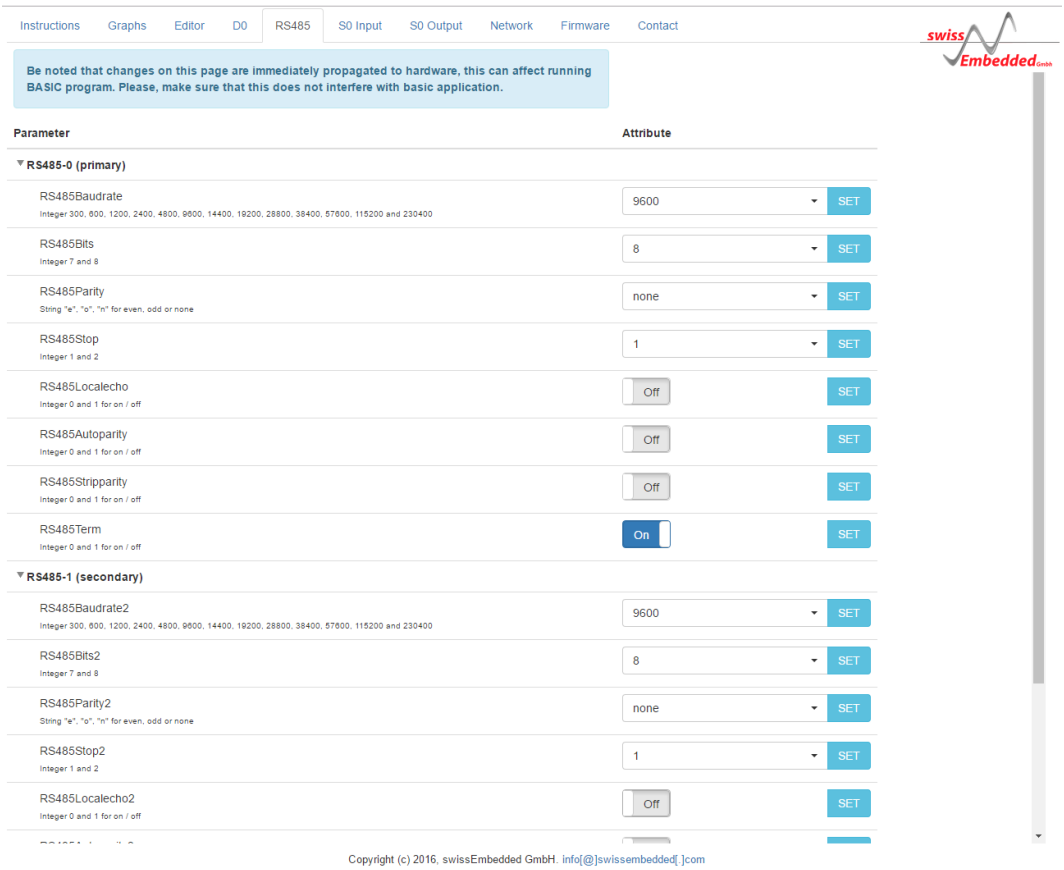


Figure 2.0.6: *Webinterface S0 input*

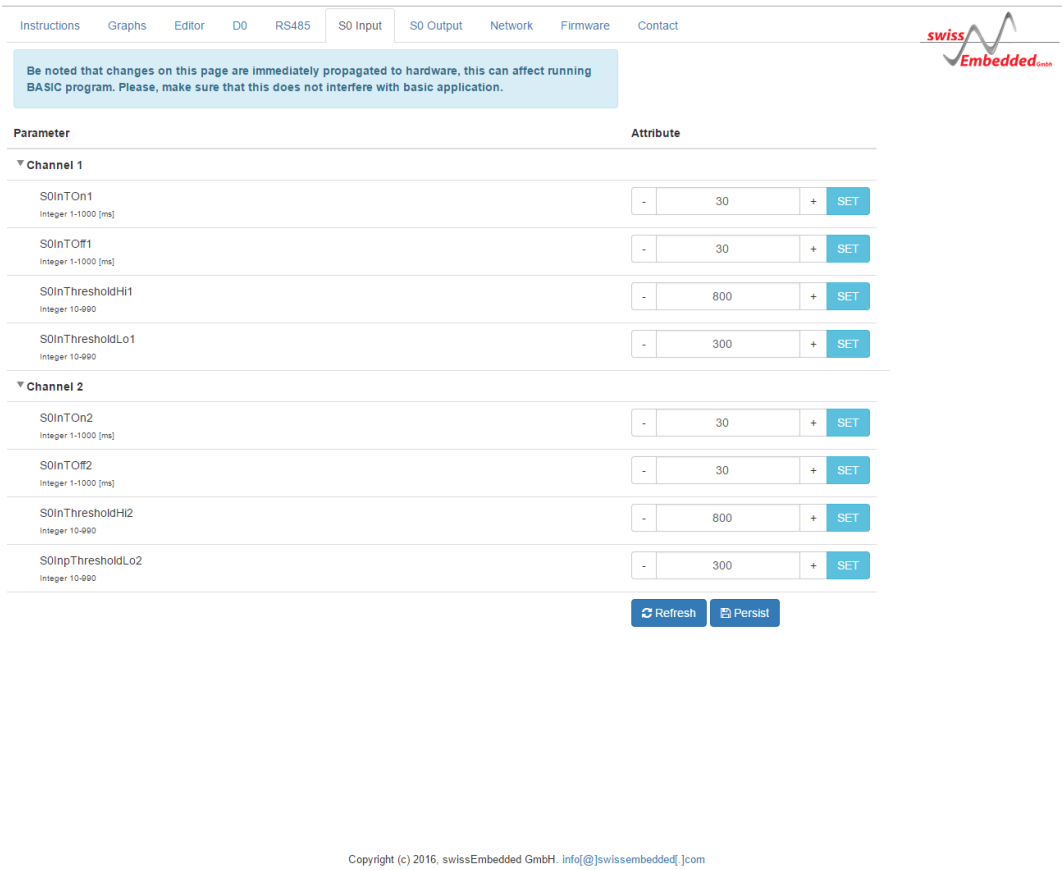


Figure 2.0.7: *Webinterface S0 output*

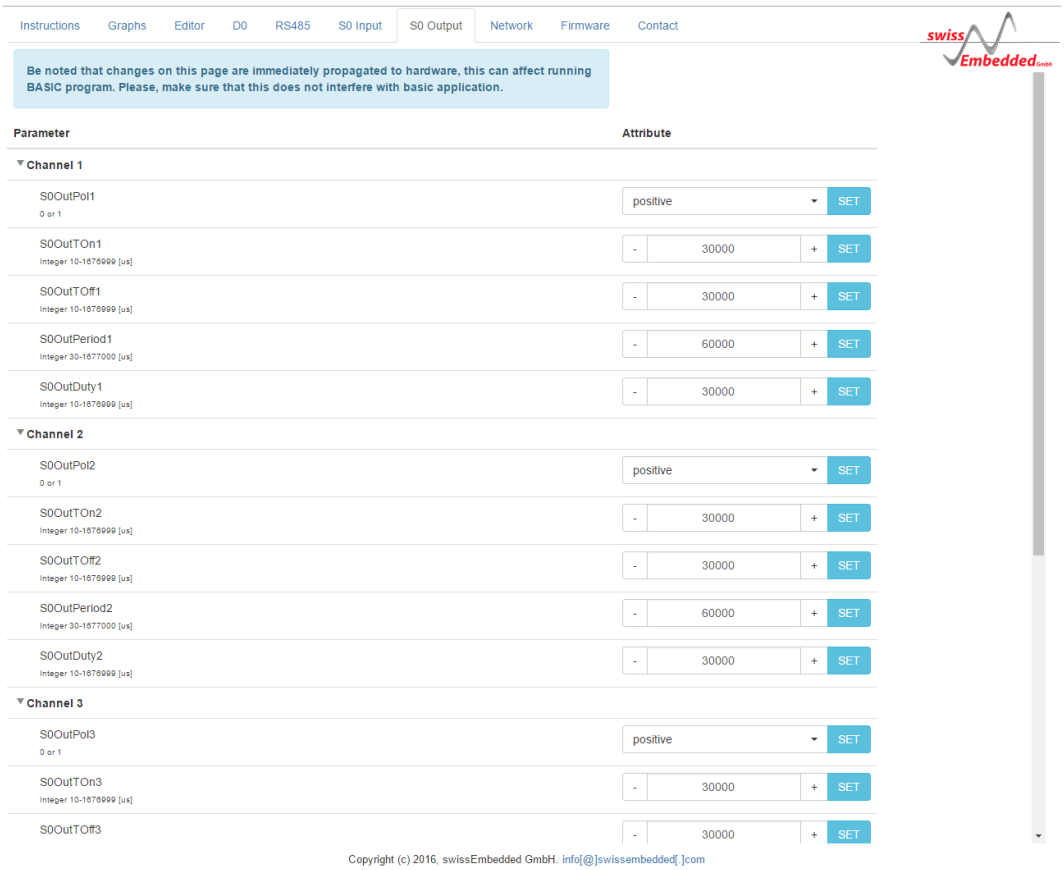


Figure 2.0.8: *Webinterface Network*

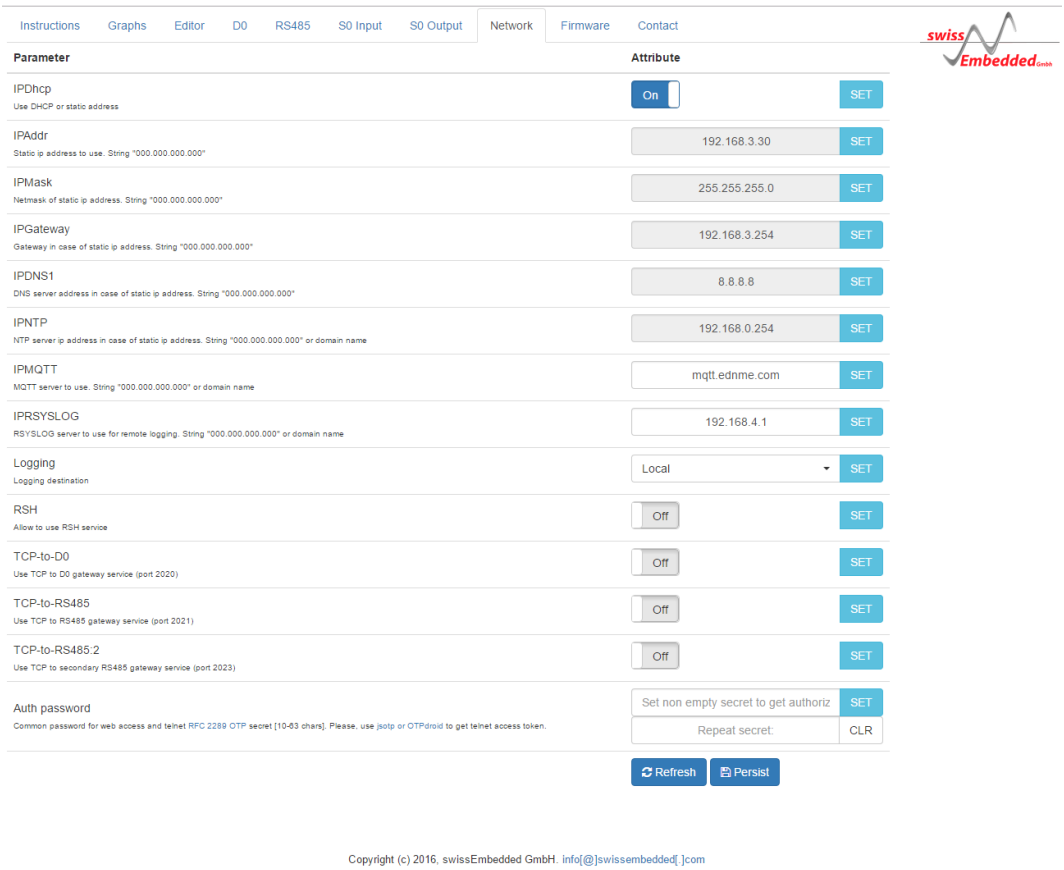


Figure 2.0.9: *Webinterface Firmware*

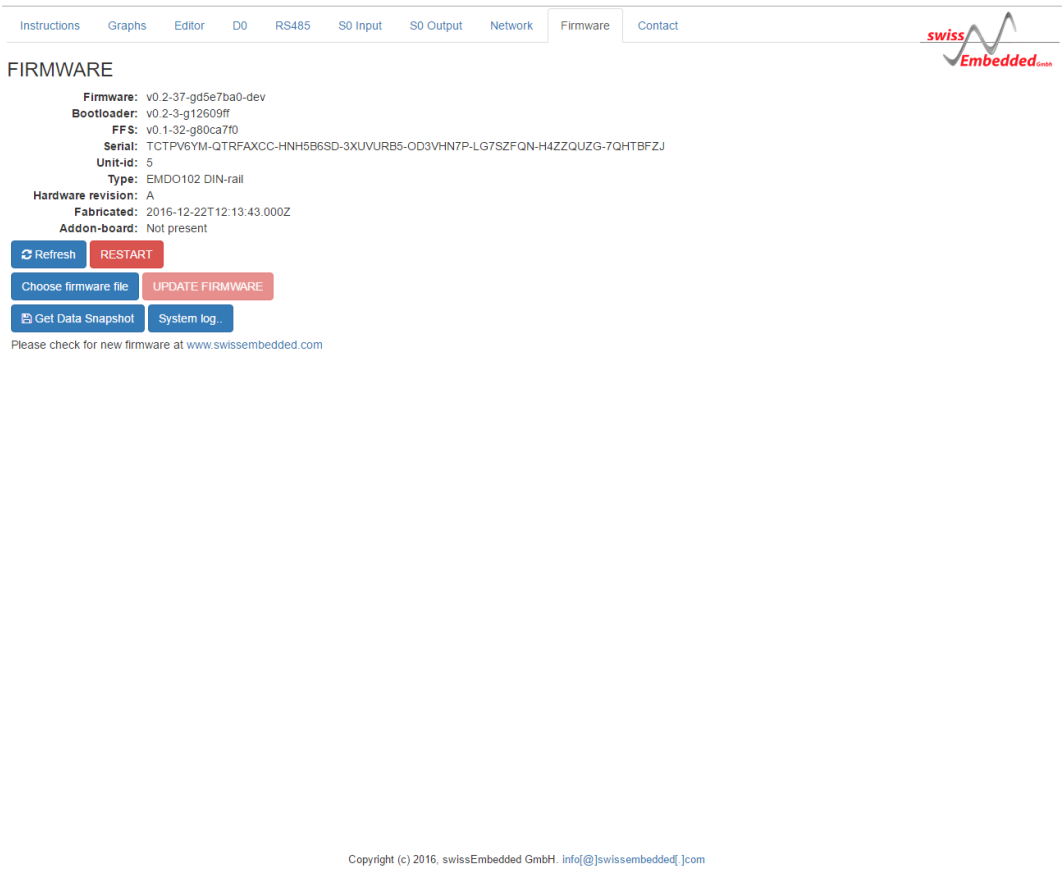
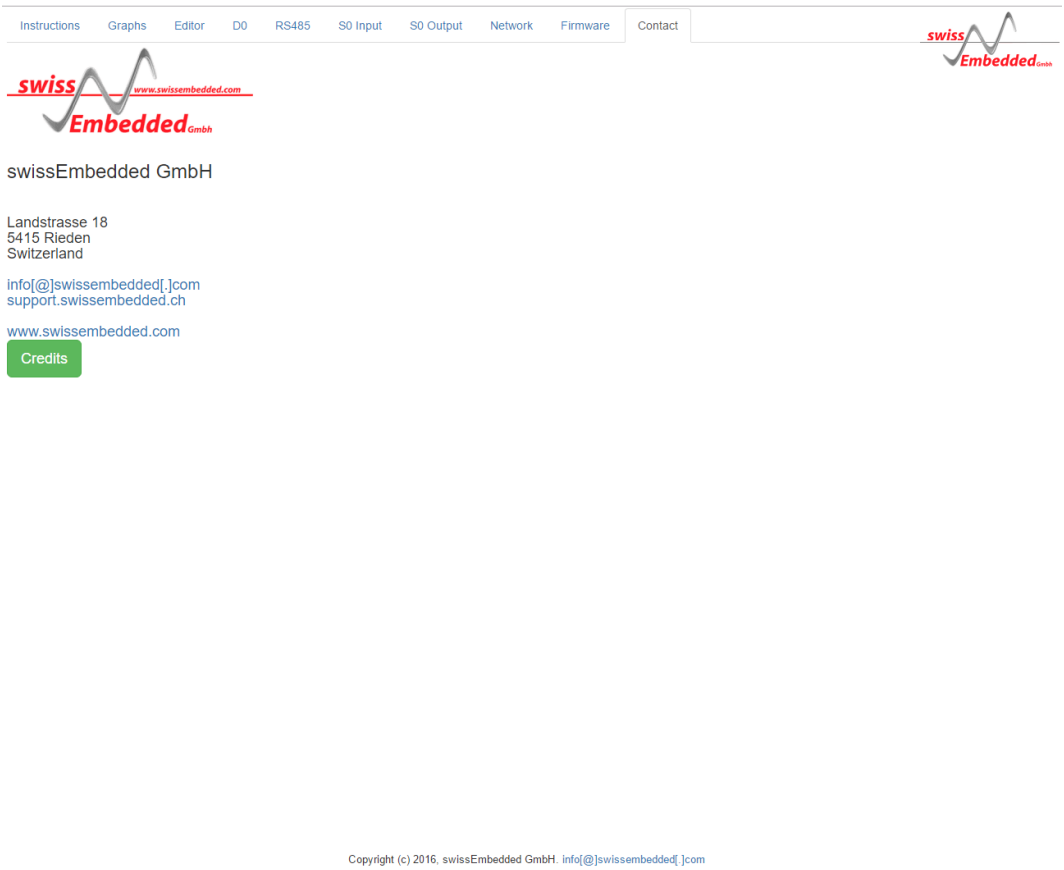
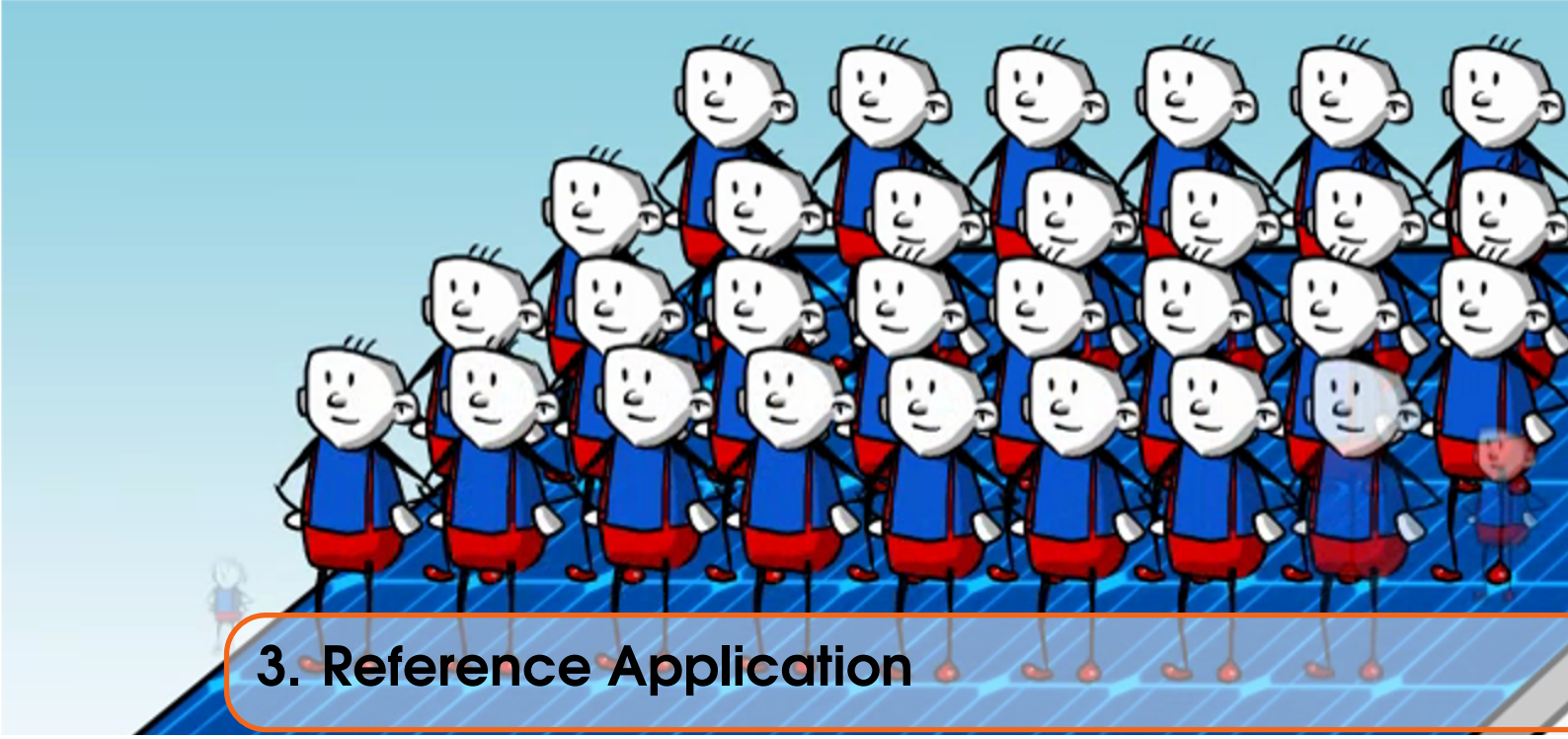


Figure 2.0.10: *Webinterface Contact*





3. Reference Application

3.1 EMDO101

The following figure 3.1.1 shows the EMDO101 setup with the following devices:

- **S0 RX** devices such as an inverter input, controlling inverter's power sourcing to the house and grid, e.g. for grid excess power control (reduction).
- **S0 TX** devices such as an energy meter, producing e.g. 1000 pulses per kWh.
- **Relay Out** for relay or solid state devices, e.g. switching a load such as a water heater or heat pump on and off.

In the next section 3.2 examples for the EMDO102 are shown. These examples also apply to the EMDO101, except the 1-wire example, as the EMDO101 does not have a 1-wire interface due to the smaller form factor.

3.2 EMDO102

The EMDO din rail devices have a build in 1-wire interface. In the following figure 3.2.1, a 1-wire temperature sensor based on DS18S20 chip is shown. Such devices are low cost and available from many sources. The wire colors might be manufacturer and model dependent.
The example shows the parasitic power wiring of the chip. The chip can also be sourced from 5V, available on the connector on the top row.
Please be aware that 1-Wire is not a differential bus, it is not made for longer cables. Cables must be kept below 1 meter. Wiring should be done linear from one device to the next without any star topology, as it introduces additional signal reflections in the line.
The figure 3.2.2 shows an RFID card reader example with power supply for the card reader from the EMDO. The device has two LEDs which can be driven directly by the EMDO.

The figure 3.2.3 shows the wiring of the RS485. RS485 uses a twisted pair cable and 120Ohm termination resistors on both ends. The EMDO has a built in termination resistor that can be activated by software in the RS485 settings of the corresponding port.

Figure 3.1.1: *EMDO101 classic wiring example*

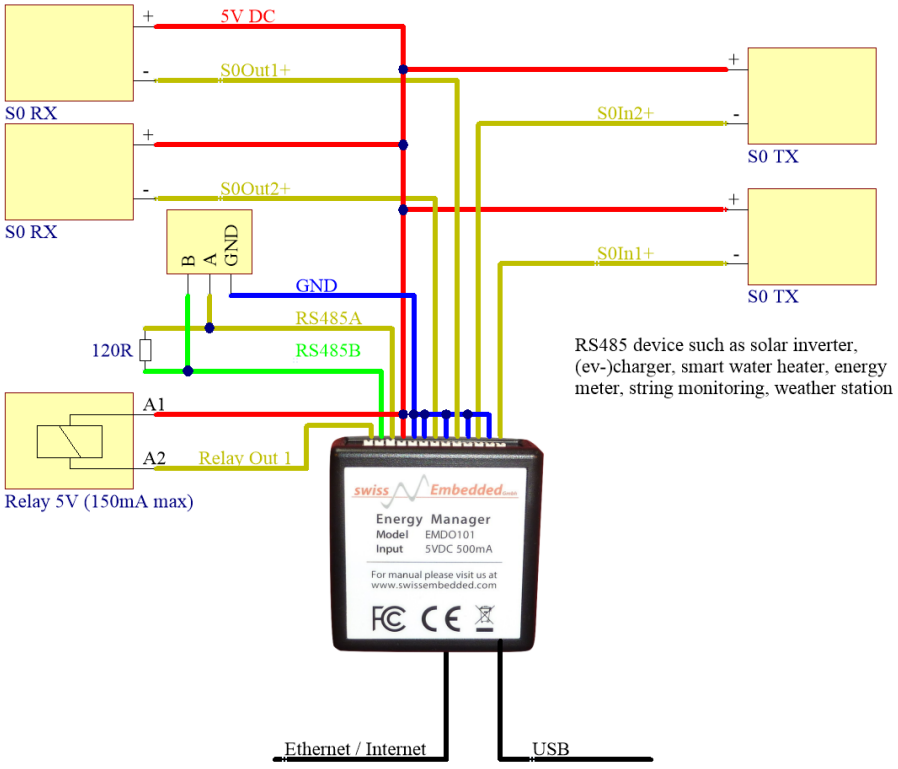


Figure 3.2.1: *EMDO102 temperature sensor example*

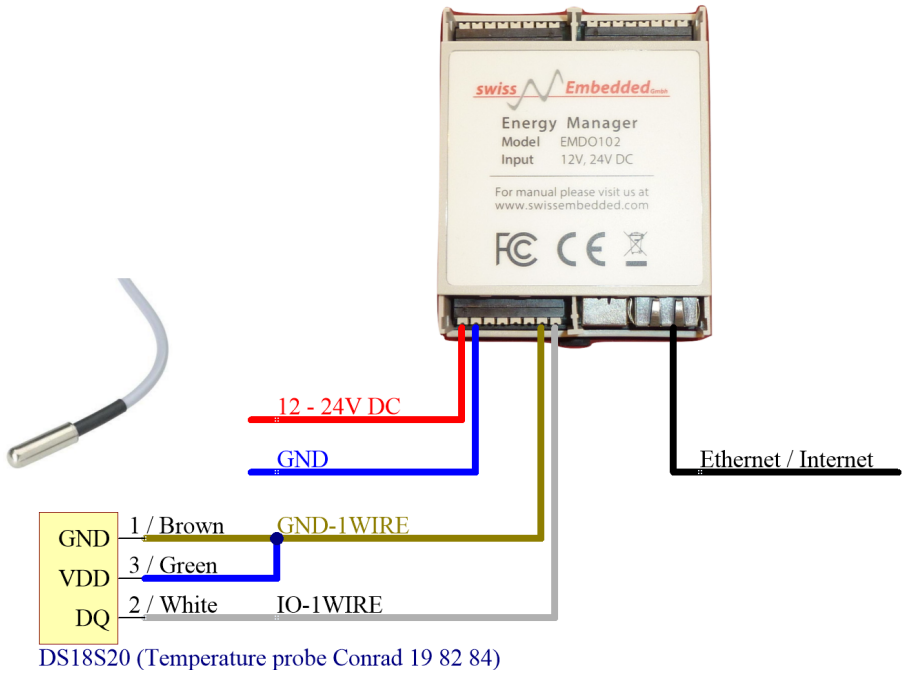
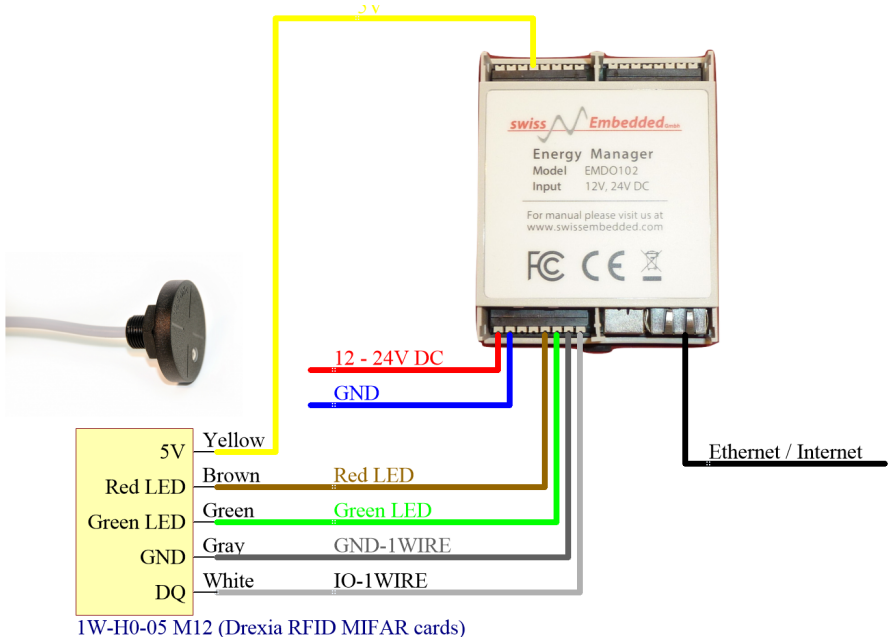


Figure 3.2.2: *EMDO102 RFID card reader example*



In some situations the 120Ohm termination is not enough. This depends on the wiring and cable length. In that case we recommend to add an external termination resistor and disable the internal termination.

The figure 3.2.4 shows the possible wiring of the EMDO with relay, relay contractor, zero crossing solid state relay and single / three phase angle controller. Thyristor based controller may introduce switching noise, external filters might be required. We recommend to contact local power power company about installation rules of phase angle controllers and allower maximal power.
Please refer to section 3.1 for a description of the figure 3.2.5. The EMDO family share the same S0 hardware.

3.3 EMDO103

Instead of the second RS485 port, the EMDO103 has a MBUS transceiver.

3.4 EMDO104

Instead of the second RS485 port, the EMDO104 has a KNX slave transceiver.

Figure 3.2.3: *EMDO102 RS485 example*

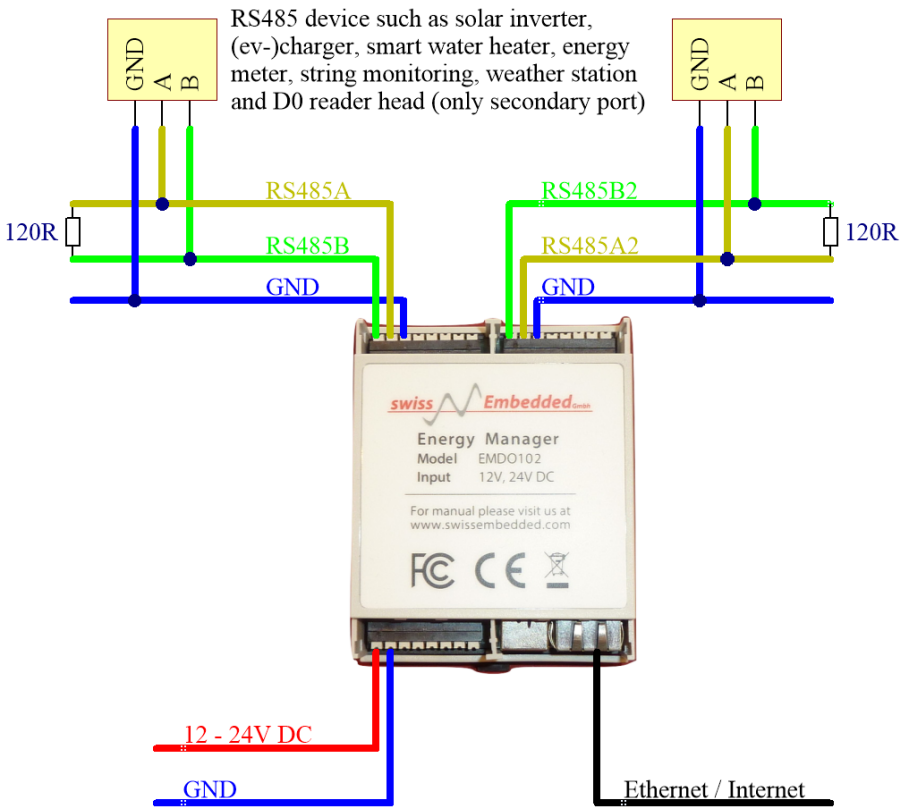


Figure 3.2.4: *EMDO102 relay driver example*

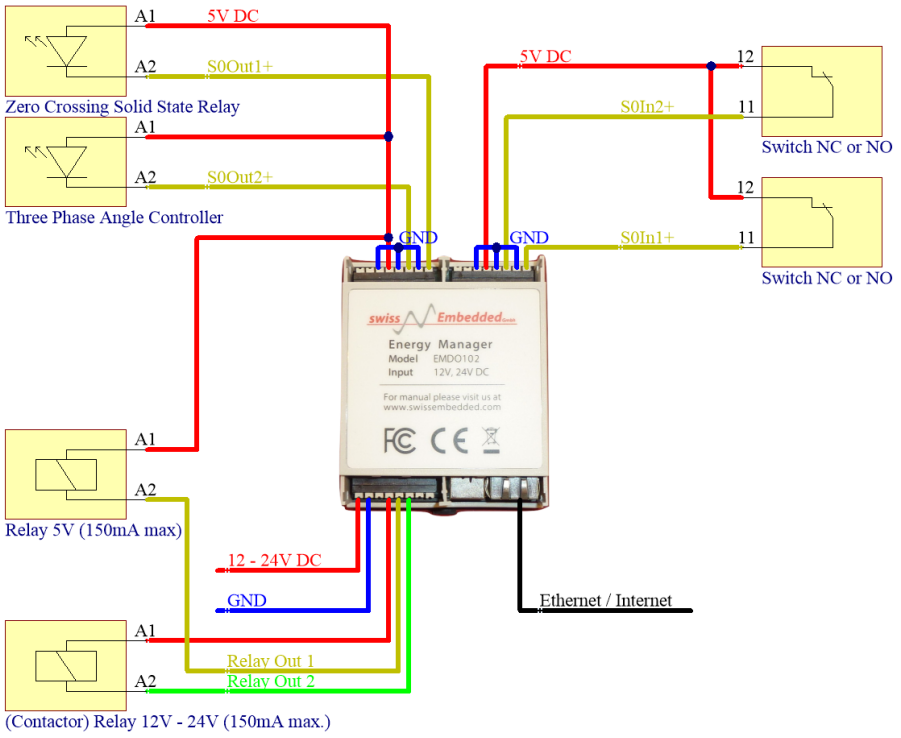
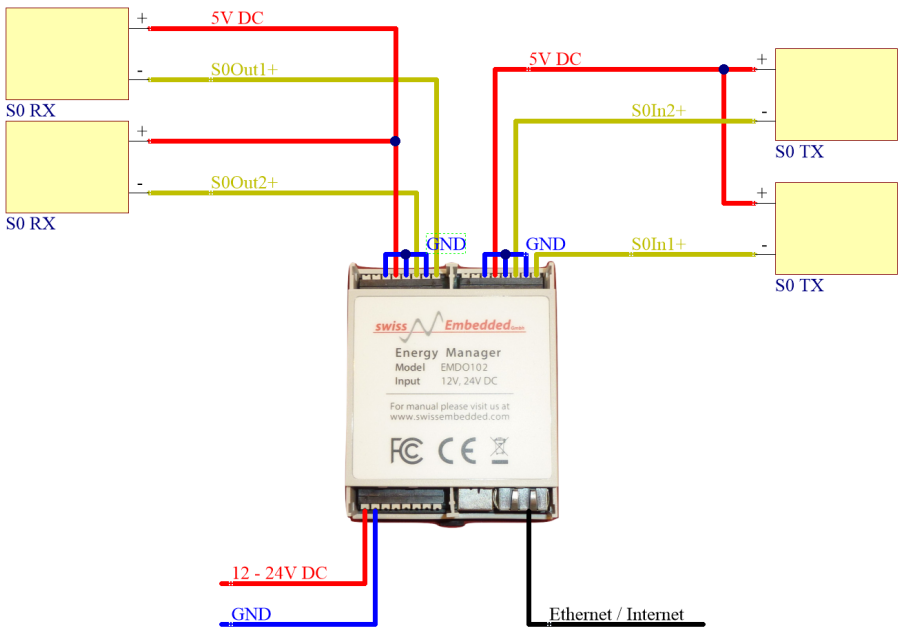


Figure 3.2.5: *EMDO102 S0 input and output example*



4. Basic Interpreter

4.1 Basic introduction

BASIC (**B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode) is an easy to learn programming language. The EMDO allows the programming through its build in editor. BASIC is a interpreted language and does not require a compiler.

4.1.1 Variables

Programming is the same as maths; it uses variables to represent data. Variables are names given for locations in memory where data is stored. The language supports variables with integer (whole numbers), floating point and strings (text). In addition it is possible to define these types of variables in multiple dimensions array.

```
' str is a variable contains the string EMDO
str$ = "EMDO"

' intNum is a variable contains the integer 50
intNum%=50

' floatNum is a variable contains the float 40.2
floatNum=40.2

' this is a string
a$="Hello world"

' This is a array of type string which is declared with DIM statement
dim b$(2)
b$(0)="this is a text"
b$(1)="this is another text"

' Print the variables in the web browser output field of the EMDO
```

```
print str$ intNum% floatNum a$ b$(0) b$(1)

' It is also possible to calculate things with numerical variables
intNum%=intNum%+1
floatNum=floatNum + 0.5

' It is also possible to add a string to another string Hello World!
a$=a$+"!"
```

4.1.2 Loops

Sometimes task must be repeated several times. A loop is a structure that allows a piece of code to be executed one or more times. Before or after each iteration a loop condition is checked and iteration is confinued only if this condition is true. There are different types of loops. **FOR NEXT**, **DO WHILE** and **DO LOOP UNTIL** loops.
DO WHILE loop and **DO LOOP UNTIL** both iterate due to a certain condition. The only difference is that **DO WHILE** loop checks the condition before entering the loop. **DO LOOP UNTIL** checks the condition after finishing each iteration. This allows the **DO LOOP UNTIL** to enter the loop at least once even if the condition is false, however, the **DO WHILE** may never enter the loop under the same condition when false.
FOR NEXT statement allows to count a number within a specified range.

```
' This is a for loop counting from -100 to 100 in steps of 2 and print the
    number out.
a=10.0
for i = -100 to 100 step 2
 a=a*10.0
 print i a
next i


' This is a while loop where the sum is the summation of the count values as
    long as the count is smaller then 10. The condition is checked BEFORE the
    summation is done prior to each iteration of the loop.
count%=0
sum%=0
do while count% < 10
 sum% = sum% + count%
 count% = count% + 1
loop

' This is a do while loop where the sum is the summation of the count values
    until the count reaches 0 or below zero. The condition is checked AFTER the
    summation is done.
do
 sum% = sum% + count%
 count% = count% - 1
loop until count% > 0
```

4.1.3 Functions

When writing code, a program quickly gets more than 20 lines. Functions break the code down into small parts. Each part (function) does a specific task. This way, the code become more readable

and usable for many different tasks.
The function might need some information to do its job, these info is given as parameters (arguments) on the call. They are variables that are required as inputs for the function to do its job. However, some functions do not need them.
Variables declared inside the function are **LOCAL** variables that are only visible within the function and will be destroyed (removed from its memory location) once the function ends. If a variable is set inside the function without **LOCAL** declaration, the variable will be globally valid. If this variable exists outside the function, the value will be overwritten. It is good practive to deliver a variable as a argument which will be then modified by the function.
A function can return a value to be stored in the variable which call the function. The return value is assigned to the function's name.
It is good programming practice to document the functions.

```
' The variable 'consumedPower' which calls the function 'calculatePower' stores
    the returned value of the function
consumedPower = calculatePower(2.2, 1.1)
print consumedPower

' This function calculates power from volts and amps
function calculatePower (volts, amps)
 ' power is a local variable which is not "visible" outside of the function
 local power
 power = volts * amps
 calculatePower = power
end function
```

Subroutines are a special type of function which does not return any result.

```
calcPower 2.2, 1.1
print "Power = Voltage * Current = " power

' This subroutine prints the value of the power (instead of returning it)
sub calcPower (volts, amps)
 ' this variable is globally "visible" outside of the subroutine
 power = volts * amps
end sub
```

4.1.4 If

IF Statements (Conditional Control Flow):
An **IF** statement is a piece of code that is executed only if a specified condition is true.

```
power = calculatePower (voltage, current)

' IF statement switches light on when power is in the range of 50 to 1000,
    switches light off when power more then 1000, or prints a string if power is
    less then 50
if power >= 50 and power <= 1000 then
 switchLight(1)
elseif power > 1000 then
 switchLight(0)
else
 print "Warning Power is less than 50"
endif
```

4.1.5 Libraries

A library is a group of functions and subroutines that are used by other programs. By including a library, functions defined in the library can be called from the application or from another library. Libraries can be loaded/unloaded during the runtime. Library files have the extension '.blib'.

```
' Loading libraries allow using functions without defining them
library load "switchlight"
library load "powermeter"
power = readMeter()
if power >= 50 and power <= 1000 then
 switchLight(1)
else
 switchLight(0)
endif
```

Here is an example of a distributed system with master and slave EMDO. Master EMDO is reading the power consumption from the meter and send that info to a slave EMDO which switches the light on and off accordingly.

```
'''''''''''''' Master : Meter ''''''''''''''
library load "powermeter"
library load "slavecontrol"

power = readMeter()
if power >= 50 then
 ' send state to the slave (virtual switch)
 switchSlaveSend(1)
endif

'''''''''''''' Slave : Switch ''''''''''''''
library load "slavecontrol"
library load "switchlight"

do
 ' Receiveing state from master about the virtual switch
 receivedMasterState%=switchSlaveReceive()
 ' Sending state to the switch
 switchLight(receivedMasterState%)
loop
```

4.1.6 Scheduler

CRON is a statement that is used to schedule functions. An example of a logger, that stores the readings of power from an inverter and stores the reading every 15 minutes on flash, is shown below.

```
library load "inverter"
library load "logger"

' Add Cron schedule every minute
cron%=CrontabAdd("*/15 * * * *")
if cron% < 0 then error "Failed to add CRON entry"
' Dispatch the cron every 1 hour
on cron cron% quarterCronHandler
```

```
do
 ' sleep a second, read current inverter value, dispatch will execute cron if
    expired
 producedPower=inverterRead()
 dispatch 1000
loop

' This function is called every 15 minutes
function quarterCronHandler (cron%)
  logger(producedPower)
end function
```

4.2 EMDO command and function overview

The EMDO Basic interpreter has been ported from Geoff Graham's famous MMBASIC interpreter. The original code has been extensively tested, bugfixed and enhanced in functionality, especially in regards of energy management and network communication. We recommend the MMBASIC manual for further reading, see page 111.
MMBASIC supports three types of variables. The type is indicated by the suffix of the variable name.

- A variable with a $ suffix (e.g. *string$*) indicates a string variable. A string has a maximal length of 255 characters.
- A variable with a % suffix (e.g. *integer-value%*) indicates a integer variable. This is a 64bit signed value.
- A variable without a suffix (e.g. *float-value*) indicates a float variable (64bit). This is IEEE754 compliant. The EMDO has a built in floating point unit.

In the following tables, an overview of all EMDO instructions is given. Each BASIC instruction is shown with parameters and return value (if it is a function). Square brackets indicate that a parameter or characters is optional.
The EMDO stores the basic program in the file *start.bas*. This file can be edited through the EMDO BASIC editor in the web interface. The EMDO has also library support. These files are stored at *library-name.blib*. An rich set of libraries is available at github, please see subsection **??** for details.

A variable or state is testet in a program (e.g. if excess power is larger than 1kW). The following table 4.2.1 describes the conditional control flow.

Table 4.2.1: *Conditional control flow*

Basic Interpreter
If *expr* **Then** <statement> **Else** <statement> Evaluates the expression *expr* and performs the **Then** statement if it is true or skips to the next line if false. The optional **Else** statement is the reverse of the **Then** test. This type of **If** statement is all on one line. The **Then** statement construct can be also replaced with: **Goto** *linenumber* or **Goto** *label*. A *linenumber* is a increasing number that is add in front of each line of code. A *label* is the keyword *label* : (name followed by :) that is placed in front of a statement.
Continued on next page

Table 4.2.1

Basic Interpreter, continued from previous page

If *expr* **Then**
<statements>
[**Else**
<statements>]
[**Elseif** *expr* **Then**
<statements>]
Endif

The **Elseif** statement (if present) is executed if the previous condition is false and it starts a new **If** chain with further **Else** and / or **Elseif** statements as required.
One **Endif** is used to terminate the multiline **IF**.
Elseif and **Else If** are equivalent.
Endif and **End If** are equivalent.

Select Case expression
Case expression
<statements>
[**Case** expression
<statements>
Case Else
<statements>]
End Select

The **Select Case** statement is similar to the **If** statement. It allows to compare an expression to one or more expressions. The optional **Case Else** matches if none of the previous **Case** statements matches.
The **Select Case** must always be terminated with a **End Select** statement.

Variable can be tested and manipulated in various way. Table 4.2.2 shows these operators. The operators are shown in the next table in the order of precedence. Operators that are on the same level (for example + and -) are processed with a left to right precedence as they occur on the program line.
Logical and string operators are used in **If Then** statements. Arithmetic and logical and operators are used to do calculations operations (e.g. *a*%=*a*%+1).

Table 4.2.2: *Operators*

Basic Interpreter
Arithmetic operators: **^** exponentiation ***** multiplication **/** division **** integer division **Mod** modulus (remainder) **+** addition **-** subtraction
Logical operators: **Not** Logical inverse of a value **«** Shift left **»** Shift right **<>** inequality **<** less then **>** greater than **<=** less than or equal to **=<** less than or equal to (alternative version) **>=** greater than or equal to **=>** greater than or equal to (alternative version) **=** equality **AND** conjunction **OR** disjunction **XOR** exclusive or **~** binary not
String operators: **+** join two strings **<>** inequality **<** less than **>** greater than **<=** less than or equal to **=<** less than or equal to (alternative version) **>=** greater than or equal to **=>** greater than or equal to (alternative version) **=** equality

Any kind of calculation usually involves looping over a dataset. The table 4.2.3 all looping commands.

Table 4.2.3: *Looping*

Basic Interpreter
For *counter%* = *start%* **To%** *finish* [**Step** *increment%*] [<statement> **Exit For**] **Next** [*counter%*, *counter2%*, ...] Initiates a **For**-**Next** loop with the *counter%* initially set to *start%* and incrementing in *increment%* steps (default is 1) until 'counter' equals *finish%*. The *increment%* must be an integer value, but may be negative. If multiple **For**-**Next** are cascaded, one **Next** statement can be used with the list of counter variables. It is recommended to state the counter variable for better readability of the code. **Exit For** provides a early exit for the loop. Execution will be continued after the **Next**.
Do [<statements> **Exit Do**] **Loop** This structure will loop forever; the **Exit Do** or only **xit** can be used to terminate the loop or control must be explicitly transferred outside of the loop by commands like **Goto** or **Return** (if in a subroutine).
Do While *expression* <statements> **Loop** **Do** <statements> **Loop Until** *expression* Loops while *expression* is true (this is equivalent to the older **While**- **Wend** loop, also implemented in MMBasic). If, at the start, the expression is false the statements in the loop will not be executed, even once. Loops until the expression following **Until** is true. Because the test is made at the end of the loop the statements inside the loop will be executed at least once, even if the expression is false.

The program flow can involve branching, the simplest form are shown in the next table 4.2.4.

Table 4.2.4: *Branching*

Basic Interpreter
Continue Resume running a program that has been stopped by an **End** statement, an error, or web interface stop button. The program will restart with the next statement following the previous stopping point.
Gosub target Initiates a subroutine call to the target, which can be a line number or a label. The subroutine must end with **Return**.
Goto target Branches program execution to the target, which can be a line number or a label.
On expression **Gosub** target **On** expression **Goto** target **On CRON** clid% handler [**Detach**] **On TIMER** clid% handler [**Detach**] If the expression evaluates to true, the **Gosub / Goto** is executed. Special functionality offers **On CRON** and **On TIMER**. This registers a handler that is executed if the cron or timer are scheduled. The optional **Detach** statement unregisters the handler again.

BASIC supports functions, which will return an argument, and subroutines. Both take parameters on call. These parameters can be modified within in routine, if the are referenced through a variable name. Functions and subroutine are used to "do something" and go back to continue the program flow to where it has been called from. See table 4.2.5 for details.

Table 4.2.5: *Functions and Subroutines*

Basic Interpreter
Local *variable* [,*variable*] Defines a list of variable names as local to the subroutine or function. *variable* can be an array and the array will be dimensioned just as if the
Continued on next page

Table 4.2.5

Basic Interpreter, continued from previous page

Function *function-name* ([*arg1*, *arg2*, . . .])
[<statements>
xxx = <return value> **Exit Function**]
End Function

Defines a callable function. A function is a piece of code that takes some arguments and returns a result at the end.
function-name is the function name and it must meet the specifications for naming a variable. *arg1*, *arg2*, etc are the arguments or parameters to the function.
To set the return value of the function you assign the value to the function's name. For example:
Function SQUARE(a)
SQUARE = a * a
End Function
Every definition must have one **End Function** statement. When this is reached the function will return its value to the expression from which it was called. The command **Exit Function** can be used for an early exit.
You use the function by using its name and arguments in a program just as you would a normal MMBasic function. For example:
Print SQUARE(56.8)
When the function is called each argument in the caller is matched to the argument in the function definition. These arguments are available only inside the function.
Functions can be called with a variable number of arguments. Any omitted arguments in the function's list will be set to zero or a null string. Arguments in the caller's list that are a variable (ie, not an expression or constant) will be passed by reference to the function. This means that any changes to the corresponding argument in the function will also be copied to the caller's variable.
You must not jump into or out of a function using commands like **Goto**, **Gosub**, etc. Doing so will have undefined side effects including the possibility of ruining your day.

Continued on next page

Table 4.2.5

Basic Interpreter, continued from previous page

Sub *sub-name* ([*arg1*,*arg2*, . . .])
[<statements>
Exit Sub]
End Sub

Defines a callable subroutine. This is the same as adding a new command to MMBasic while it is running your program.
sub-name is the subroutine name and it must meet the specifications for naming a variable. *arg1*, *arg2*, etc are the arguments or parameters to the subroutine.
Every definition must have one **End Sub** statement. When this is reached the program will return to the next statement after the call to the subroutine. The command **Exit Sub** can be used for an early exit. You use the subroutine by using its name and arguments in a program just as you would a normal command.
For example:
MySub a1, a2
When the subroutine is called each argument in the caller is matched to the argument in the subroutine definition. These arguments are available only inside the subroutine. Subroutines can be called with a variable number of arguments. Any omitted arguments in the subroutine's list will be set to zero or a null string.
Arguments in the caller's list that are a variable (ie, not an expression or constant) will be passed by reference to the subroutine. This means that any changes to the corresponding argument in the subroutine will also be copied to the caller's variable and therefore may be accessed after the subroutine has ended.

It is also possible to define multidimensional arrays in BASIC. The following table 4.2.6 shows more sophisticated variable handling commands.

Table 4.2.6: *Special Variable commands*

Basic Interpreter
Const *variable* = *constant-value* Sets a *variable* to a *constant-value*, which can't be overwritten within the program.
Data *constant*[,*constant*]... Stores numerical and string constants to be accessed by **Read**. In general string constants should be surrounded by double quotes (""). An exception is when the string consists of just alphanumeric characters that do not represent MMBasic keywords (such as **Then**, **While**, etc). In that case quotes are not needed. Numerical constants can also be expressions such as 5 * 60.

Continued on next page

Table 4.2.6

Basic Interpreter, continued from previous page

Restore

Resets the line and position counters for **Data** and **Read** statements to the top of the program file.

Read *variable*[, *variable*]...

Reads values from **Data** statements and assigns these values to the named variables. Variable types in a **Read** statement must match the data types in **Data** statements as they are read. See also **Data** and **Restore**.

Dim *var(dim)* , [*var(dim)*]...
Dim *var$(dim)* **Length** n

Specifies a variable that is an array with one or more dimensions. The variables can be numbers or strings with multiple declarations separated by commas.
dim is a bracketed list of numbers separated by commas. Each number specifies the number of elements in each dimension. Normally the numbering of each dimension starts at 0 but the **OPTION BASE** command can be used to change this to 1.
For example: **Dim** nbr(10, 20) specifies a two dimensional array with 11 elements (0 to 10) in the first dimension and 21 (0 to 20) in the second dimension. The total number of elements is 231 and because each number requires 4 bytes a total of 924 bytes of memory will be allocated.
String arrays will by default be allocated memory for 255 characters for each element and this can quickly use up memory. In that case the **Length** keyword can be used to specify the amount of memory to be allocated to each element. This allocation (*n*) can be from 1 to 255 characters.
For example: **Dim** str$(5, 10) will declare a string array with 66 elements consuming 16,896 bytes of memory while:
Dim str$(5, 10) **Length** 20
Will only consume 1,386 bytes of memory. Note that the amount of memory allocated for each element is n + 1 as the extra byte is used to track the actual length of the string stored in each element.
If a string longer than *n* is assigned to an element of the array an error will be produced. Other than this string arrays created with the **Length** keyword act exactly the same as other string arrays.
Examples:
Dim nbr(50)
Dim str$(20)
Dim a(5,5,5), b(1000)
Dim str$(200) **length** 20

Let *variable* = *expression*

Assigns the value of *expression* to the *variable*. **Let** is automatically assumed if a statement does not start with a command.

Continued on next page

Table 4.2.6

Basic Interpreter, continued from previous page

Dim command had been used.

A local variable will only be visible within the procedure and will be deleted (and the memory reclaimed) when the procedure returns. If the local variable has the same name as a global variable (used before any subroutines or functions were called) the global variable will be hidden by the local variable while the procedure is executed.

Option Base 0
Option Base 1

Set the lowest value for array subscripts to either 0 or 1. The default is 0. This must be used before any arrays are declared.

Option Error Continue
Option Error Abort

Set the treatment for errors in file input/output. The option **CONTINUE** will cause MM-Basic to ignore file related errors. The program must check the variable **MM.ERRNO** to determine if and what error has occurred.
The option **Abort** sets the normal behaviour (ie, stop the program and print an error message). The default is **ABORT**.
Note that this option only relates to errors reading from or writing to NOR flash. It does not affect the handling of syntax and other program errors.

Option Prompt string$

Sets the command prompt to the contents of *string$* (which can also be an expression which will be evaluated when the prompt is printed).
For example:
Option Prompt “Ok “
Option Prompt TIME$ + “: "
Option Prompt CWD$ + “: “
Maximum length of the prompt string is 48 characters. The prompt is reset to the default (“> “) on power up but you can automatically set in the first lines of the program.

Continued on next page

Table 4.2.6

Basic Interpreter, continued from previous page

SetVarAccess *http-json-access-rights$*

Scalar variables (not arrays) in the BASIC application can be modified over http json requests. The *http-json-access-rights$* set the default to **PROTECTED**, **READONLY**, **WRITEONLY** or **READWRITE** for all variables that are initialized in the application. Already initialized variables are not affected by the new default. The interpreter starts with **READONLY** default. So all variables are readable over http json.
Read a variable from EMDO:

```
function get_error( title, jqXHR, textStatus ) {
   var sjqMsg;
   if (jqXHR != void 0) {
      if ( jqXHR.hasOwnProperty('responseText') ) {
         if (jqXHR.responseText.indexOf('<meta property="emdo:pageid"') != -1)
            sjqMsg = textStatus; // avoid refer to the same page
         else sjqMsg = jqXHR.responseText;
      } else sjqMsg = textStatus;
   } else sjqMsg = textStatus;
    return title + sjqMsg;
}

function get_variable(device_address, varname)
{
 var udata = JSON.stringify({
  Method: "GetParameters",
  params: {BasicVariable:varname}
 });
 var promise = new Promise(function(resolve, reject) {
   $.ajax({
      url : device_address+'?json='+encodeURIComponent(udata),
      type : 'GET',
      dataType : 'json',
      timeout : 5000,
      cache: false
   }).done(function(response) {
      resolve( response["result"] );
   }).fail(function(jqXHR, textStatus){
      reject( "Failed to query: ", get_error( jqXHR, textStatus ) );
   });
 });
 return promise;
}

function get_multi_var(device_address, vars)
{
   var udata = {
      Method: "GetBasicMulti",
      params:[]
   };

   var i = 0, p;
   for ( p in vars ) {
      udata.params[i]=vars[p];
      i++;
   }
   udata= JSON.stringify(udata);
   var promise = new Promise(function(resolve, reject) {
      $ajax({
```

BASIC has a rich set of mathematical functions. The following table 4.2.7 shows the commands.

Table 4.2.7: *Math functions*

Basic Interpreter
absolute-value = **Abs**(*number*) Returns the *absolute-value* of the argument *number* (ie, any negative sign is removed and the positive number is returned).
arccosinus = **Acos**(*number*) Returns the arccosinus of a *number* (0 to π) in radians
cosinus = **Cos**(*number*) Returns the *cosinus* of the argument *number* in radians.
arcsinus = **Asin**(*number*) Returns the *arcsinus* of a *number* in radians ($-\frac{\pi}{2}$ to $\frac{\pi}{2}$) in radians.
sinus = **Sin**(*number*) Returns the *sinus* of the argument *number* in radians.
arctangent = **Atn**(*number*) Returns the *arctangent* of a *number* in radians ($-\frac{\pi}{2}$ to $\frac{\pi}{2}$) in radians.
tangent = **Tan**(*number*) Returns the *tangent* of the argument *number* in radians.
integer-value = **Int**(*number*) Truncate an expression to the next whole number less than or equal to the argument (*integer-value*). For example 9.89 will return 9 and -2.11 will return -3. This behaviour is for Microsoft compatibility, the **Fix**() function provides a true integer function. See also **Cint**() .
rounded-value = **Cint**(*number*) Round numbers with fractional portions up or down to the next whole number or integer (*rounded-value*). For example: 45.47 will round to 45 45.57 will round to 46 -34.45 will round to -34 -34.55 will round to -35 See also **Int**() and **Fix**().
Continued on next page

Table 4.2.7

Basic Interpreter, continued from previous page

degrees = **Deg**(*radians*)

Converts *radians* to *degrees*.

radians = **Rad**(*degrees*)

Converts *degrees* to *radians*.

exponential-value = **Exp**(*number*)

Returns the *exponential-value* of *number*.

truncated-value = **Fix**(*number*)

Truncate a number to a whole number by eliminating the decimal point and all characters to the right of the decimal point.
For example 9.89 will return 9 and -2.11 will return -2 (*truncated-value*).
The major difference between **Fix** and **Int** is that **Fix** provides a true integer function (ie, does not return the next lower number for negative numbers as **Int**() does). This behaviour is for Microsoft compatibility. See also **Cint**() .

natural-logarithm-value = **Log**(*number*)

Returns the natural logarithm (*natural-logarithm-value*) of the argument *number*.

pi-constant = **Pi**()

Returns the value of pi.

pseudo-random-number = **Rnd**(*number*)
Rnd

Returns a *pseudo-random-number* in the range of 0 to 0.999999. The 'number' value is ignored if supplied. This function is based on the EMDO crypto hardware module and does not need randomization.

rounded-value = **Round**(*number*, *places*)

Round the number to *number* to *places* number of fractional digits (*rounded-value*).

sign-value = **Sgn**(*number*)

Returns the sign of the argument *number*, +1 for positive numbers, 0 for 0, and -1 for negative numbers (*sign-value*).

square-root-value = **Sqr**(*number*)

Returns the square root (*square-root-value*) of the argument *number*.

The following string manipulation routines are available, see table 4.2.8.

Table 4.2.8: *String manipulation*

Basic Interpreter
ascii-value = **Asc**(*string$*) Returns the ASCII code (*ascii-value*) for the first letter in the argument *string$*.
*one-character$* = **Chr$**(*number*) Returns a *one-character string$* consisting of the character corresponding to the ASCII code indicated by argument *number*.
string-converted$ = **Conv** (*converter$*, *int-to-convert%*) *string-converted$* = **Conv** (*converter$*, *float-to-convert*) *int-converted%* = **Conv** (*converter$*, *string-to-convert$*) *float-converted* = **Conv** (*converter$*, *string-to-convert$*) Data need serialisation and deserialisation when send over data communication interface. The EMDO supports the following data conversions, specified in the *converter*: i16 (signed 16bit), u16 (unsigned 16 bit), i32 (signed 32 bit), u32 (unsigned 32bit), f32 (float 32 bit) and f64 (float 64bit) to / from bbe (binary big endian string) and ble (binary little endian string). Examples: *string-converted$* = **Conv** ("i16/bbe", *int-converted%*) *float-converted* = **Conv** ("ble/f32", *string-to-convert$*)
*crc-sum$* = **CRC$** (*mode%*, *data$*) Calculate the CRC checksum *crc-sum$* of *data$*. The following *mode%* are supported: Modbus CRC16 (use 0), Modbus LRC (use 1), 1-Wire CRC8 (use 2) and 1-Wire CRC16 (use 3).
Continued on next page

Table 4.2.8

Basic Interpreter, continued from previous page

*formatted-string$* = **Format$**(*nbr* [, *fmt$*])

Will return a string representing *nbr* formatted according to the specifications in the string *fmt$*. The format specification starts with a % character and ends with a letter. Anything outside of this construct is copied to the output as is.
The structure of a format specification is:
% [flags] [width] [.precision] type Where *flags* can be:
- Left justify the value within a given field width
0 Use 0 for the pad character instead of space
+ Forces the + sign to be shown for positive numbers
space Causes a positive value to display a space for the sign. Negative values still show the – sign
width is the minimum number of characters to output, less than this the number will be padded, more than this the width will be expanded.
precision specifies the number of fraction digits to generate with an e, or f type or the maximum number of significant digits to generate with a g type. If specified, the precision must be preceded by a dot (.).
type can be one of:
g Automatically format the number for the best presentation.
f Format the number with the decimal point and following digits
e Format the number in exponential format
If uppercase G or F is used the exponential output will use an uppercase E. If the format specification is not specified "%g" is assumed.
Examples:
format$(45) will return 45
format$(45, "%g") will return 45
format$(24.1, "%g") will return 24.1
format$(24.1,"%f") will return 24.100000
format$(24.1, "%e") will return 2.410000e+01
format$(24.1,"%09.3f") will return 00024.100
format$(24.1,"%+.3f") will return +24.100
format$(24.1,"**%-9.3f**") will return **24.100 **

position = **Instr**([*start-position*,] *string-searched$*, *string-pattern$*)

Returns the *position* at which *string-pattern$* occurs in *string- searched$*, beginning at *start-position*.

*lower-case-string$* = **Lcase$**(*string$*)
*upper-case-string$* = **Ucase$**(*string$*)

Returns *string$* converted to lowercase characters (**Lcase$**) or uppercase characters (**Ucase$**).

Continued on next page

Table 4.2.8

Basic Interpreter, continued from previous page

*left-string$* = **Left$**(*string$*, *number-of-chars*)
*right-string$* = **Right$**(*string$*, *number-of-chars*)
*mid-string$* = **Mid$**(*string$*, *start-position-in-string*[, *number-of-chars*])

Returns a substring of *string$* with *number-of-chars* from the left (beginning) of the string (**Left$**), from the right (end) of the string (**Right$**) or *start-position-in-string* in the string (**Mid$**). *start-position-in-string* is in the range 1..**len**(string$)

*trimmed-string$* = **Ltrim$**(*string$*)
*trimmed-string$* = **Rtrim$**(*string$*)
*trimmed-string$* = **Trim$**(*string$*)

Return a *string$* copy without leading spaces on the left (**Ltrim$**), right (**Rtrim$**) or on both ends (**Trim$**).

number-of-characters = **Len**(*string$*)

Returns the *number-of-characters* in 'string$'.

*blank-space-string$* = **Space$**(*number*)
blank-space-string$ = **Spc**(*number*)

Returns a string of blank spaces (*blank-space-string$*), *number* characters long.

*string$* = **Str$**(*number*)

Returns a *string$* in the decimal (base 10) representation of *number*.

*binary-string$* = **Bin$**(*number*)

Returns *binary-string$*, a string giving the binary (base 2) value for the *number*. This function is similar to **Str$**.

*octal-string$* = **Oct$**(*number*)

Returns *octal-string$*, a string giving the octal (base 8) representation of *number*. This function is similar to **Str$**.

Hex-string = **Hex$**(*number*)

Returns *Hex-string*, a string giving the hexadecimal (base 16) value for the *number*. This function is similar to **Str$**.

*substring$* = **Split$** (*index%*, *string$*,*token$*)

Split a *string$* into multiple *substring$* separated by *token$*. The *index%* selects the substring, e.g. 0 would select the first *substring$* of *string$* which is terminated with the *token$*.

Continued on next page

Table 4.2.8
Basic Interpreter, continued from previous page
*string$* = **String$**(*number*, *ascii-value* | *first-char-string$*) Returns a *string$*, *number* bytes long consisting of either the first character of *first-char-string$* or the character representing the ASCII value *ascii-value*.
space-string$ = **Tab**(*number*) Outputs spaces (*space-string$*) until the column indicated by *number* has been reached.
numerical-value = **Val**(string$) Returns the *numerical-value* of the *string$*. If *string$* is an invalid number the function will return zero. This function will recognise the &H prefix for a hexadecimal number, &O for octal and &B for binary.

It is possible to print something in the web interface or get an input from the web interface. The following table describes the commands 4.2.9

Table 4.2.9: *Webinterface IO*

Basic Interpreter
Input ["prompt string$";] list of variables Allows input from the keyboard to a list of variables. The input command will prompt with a question mark (?). The input must contain commas to separate each data item if there is more than one variable. For example, if the command is: **Input** a, b, c And the following is typed on the keyboard: 23, 87, 66 Then a = 23 and b = 87 and c = 66 If the "prompt string$" is specified it will be printed before the question mark. If the prompt string is terminated with a comma (,) rather than the semicolon (;) the question mark will be suppressed.
Inkey$ Checks the input of a character.
Line Input [*prompt$*,] *string-variable$* **Line Input** #*nbr*, string-variable$ Reads entire line from the keyboard into *string-variable$*. If specified the *prompt$* will be printed first. Unlike **Input**, **Line Input** will read a whole line, not stopping for comma delimited data items. A question mark is not printed unless it is part of *prompt$*.
Continued on next page

Table 4.2.9

Basic Interpreter, continued from previous page

Print *expression* [[,;]*expression*] ... etc
? *expression* [[,;]*expression*] ... etc

Outputs text to the browser. Multiple expressions can be used and must be separated by either a:
? Comma (,) which will output the tab character
? Semicolon (;) which will not output anything (it is just used to separate expressions).
? Nothing or a space which will act the same as a semicolon.
A semicolon (;) at the end of the expression list will suppress the automatic output of a carriage return/ newline at the end of a print statement.
When printed, a number is preceded with a space if positive or a minus (-) if negative but is not followed by a space. Integers (whole numbers) are printed without a decimal point while fractions are printed with the decimal point and the significant decimal digits. Large numbers (greater than six digits) are printed in scientific format.
The function **Format$()** can be used to format numbers. The function **Tab()** can be used to space to a certain column and the string functions can be used to justify or otherwise format strings.

With the following commands it is possible to modify a file, see table 4.2.10.

Table 4.2.10: *File IO*

Basic Interpreter
Open *fname$* **For** mode **As** [#]*fnbr* Opens a file for reading or writing. *fname$* is the filename with maximal string length (case-sensitive). It can be prefixed with a directory path. For example: "DIR1/DIR2/FILE.EXT". SPIFFS does not support directories. Directories are only emulated. *mode* is **Input**, **Output**, **Append** or **Random**. **Input** will open the file for reading and throw an error if the file does not exist. **Output** will open the file for writing and will automatically overwrite any existing file with the same name. **Append** will also open the file for writing but it will not overwrite an existing file; instead any writes will be appended to the end of the file. If there is no existing file the **Append** mode will act the same as the **Output** mode (i.e. the file is created then opened for writing). Note: **Random** will open the file for both read and write and will allow random access using the SEEK command. When opened the read/write pointer is positioned at the end of the file. *fnbr* is the file number (1 to 10). The # is optional. Up to 10 files can be open simultaneously. The **Input**, **Line Input**, **Print**, **Write** and **Close** commands as well as the **Eof**() and **Input$**() functions all use *fnbr* to identify the file being operated on. See also **Option Error** and **MM.ERRNO** for error handling.
Close [#]*fnbr* [,[#]*fnbr*] . . . Close the file(s) previously opened with the file number *fnbr*. The # is optional. Also see the **Open** command.
Input$(*nbr*, [#]*fnbr*) indexInput$ Will return a string composed of textifnbr characters read from a file previously opened for INPUT with the file number *fnbr*. This function will read all characters including carriage return and new line without translation. When reading from a serial communications port this will return as many characters as are waiting in the receive buffer up to *nbr*. If there are no characters waiting it will immediately return with an empty string. The # is optional. Also see the **OPEN** command.
Print #*nbr*, *expression* [[,;]*expression*] . . . etc Same as above except that the output is directed to a file previously opened for **Output** or **Append** as *nbr*. See the **Open** command.

There are also some commands to manipulate the files on the SPIFFS filesystem, see table 4.2.11.

Table 4.2.11: *Filesystem manipulation*

Basic Interpreter
Copy *src$* To *dest$* Copy the file named *src$* to another file named *dest$*.
Dir$(*fspec*) *file-name$* = **Dir$**( ) Will search the NOR flash for files and return the names of entries found. *fspec* is a file specification using wildcards the same as used by the **Files** command. E.g., "*.*" will return all entries, "*.TXT" will return text files. The function without argument, will return the first entry found. To retrieve subsequent entries use the function with no arguments. ie, **Dir$**. The return of an empty string indicates that there are no more entries to retrieve. This example will print all the files in a directory: *f$* = **Dir$**("*.*") **Do While** *f$* <> "" **Print** f$ *f$* = **Dir$**() **Loop** SPIFFS does not support directories. Leading path alike pattern will force search within a "directory".
end-of-file = **Eof**([#]*nbr*) Will return true (textitend-of-file) if the file previously opened for **Input** with the file number *nbr* is positioned at the end of the file. If used on a file number opened as a serial port this function will return true if there are no characters waiting in the receive buffer. The # is optional. Also see the OPEN, INPUT and LINE INPUT commands and the **Input$** function.
Files [*fspec$*] Lists files in the current directory of the NOR flash. Optional use of *fspec$*. Question marks (?) will match any character and an asterisk (*) will match any number of characters. If omitted, all files will be listed. For example: *.* Find all entries *.TXT Find all entries with an extension of TXT E*.* Find all entries starting with E X?X.* Find all three letter file names starting and ending with X
Loc([#]*fnbr*) For a file opened as **Random** this will return the current position of the read/write pointer in the file. Note that the first byte in a file is numbered 1. The # is optional.
Lof([#]*fnbr*) For a file this will return the current length of the file in bytes. The # is optional.

Continued on next page

Table 4.2.11
Basic Interpreter, continued from previous page
Kill *file$* Deletes the file specified by *file$*. If there is an extension it must be specified. If *file$* is a string constant quote marks around it are optional at the command prompt but must be specified if the command is used within a program. Example: **Kill** "SAMPLE.DAT"
Name *old$* AS *new$* Rename a file or a directory from *old$* to *new$* Unlike the other commands that work with file names the **NAME** command cannot accept a full pathname (with directories).
Seek [#]*fnbr*, *pos* Will position the read/write pointer in a file that has been opened for **RANDOM** access to the *pos* byte. The first byte in a file is numbered one so **Seek** #5,1 will position the read/write pointer to the start of the file.
current-cursor-position=**Pos**() Returns the *current-cursor-position* in the line in characters.

The BASIC interpreter allows to load program code and libraries at runtime. The following table 4.2.12 shows the commands.

Table 4.2.12: *Program code handling*

Basic Interpreter
Chain *file$* Clear the current program from memory, load the new program (*file$*) into memory and run it starting with the first line. Unlike the **Run** command this command retains the current state of the program (ie, the value of variables, open files, open handles, etc). Cron, Timer, SEMP will be stay active, it is recommended to **Detach** all handlers prior to **Chain** and reattach after it. One program can **Chain** to another which can then chain to another (or back to the original) an unlimited number of times. As long as a program can be broken down into modules this command allows programs of almost unlimited size to be run, even with limited memory. Communication between the modules can be accomplished by assigning values to one or more variables which then can be examined by the new chained program. Note that another way of squeezing a large program into limited memory is to use the **Library** command.
Continued on next page

Table 4.2.12

Basic Interpreter, continued from previous page

Library Load *file$*
Library Unload *file$*

Will load a library file (*file$*) into general memory. Any user defined commands and subroutines in the file will then become available to the running program.
A library file is like any other MMBasic program, with the exception that any programming code outside the user defined commands and subroutines in the file will be ignored. The library is not visible to the user (it is not listed by the LIST command) so it should be tested and debugged as a normal basic program first. Normally a library file has the extension ".BLIB" and that extension will be automatically added if the file name (*file$*) does not include the extension.
Libraries can be loaded and unloaded in any order. Libraries can be loaded from within other libraries and nested to an unlimited extent. Any library file can be unloaded from memory and the memory returned to the general pool by using the LIBRARY UNLOAD command.
Library files must not be unloaded from within a library that is currently being used by the program (the results are undefined but it may crash MMBasic and cause your hair to go prematurely gray).
This command can be used to load specialised libraries to extend the functionality of MMBasic. Examples include device drivers, libraries that provide bit manipulation and libraries of specialised mathematical functions. The library file is only loaded on the first load command encountered so it is acceptable to put the same load command into every part of the program or every subroutine that may need the library.
Another use of the LIBRARY command is to extend the amount of memory available to a program by only loading sections of code as needed and then unloading them when their task is finished so that another function can be loaded.
To prevent fragmentation of memory, functions that use a lot of memory (like arrays) should be declared first before any libraries are loaded and then unloaded.

List
List *line*
List - *lastline*
List *firstline* -
List *firstline* - *lastline*

Lists all lines in a program line or a range of lines. If – *lastline* is used it will start with the first line in the program. If *startline* - is used it will list to the end of the program.

(Pause) *delay*

Halt execution of the running program for *delay* mS. The maximum delay is 2147483647 mS (about 24 days).

Rem string
’ string

Rem allows remarks to be included in a program. Note the Microsoft style use of the single quotation mark to denote remarks is also supported and is preferred for better readability of the code.

Continued on next page

Table 4.2.12
Basic Interpreter, continued from previous page
Run [*line*] [*file$*] Executes the program in memory. If a line number is supplied then execution will begin at that line, otherwise it will start at the beginning of the program. Or, if a file name (*file$*) is supplied, the current program will be erased and that program will be loaded from the current drive and executed. This enables one program to load and run another. Example: **Run** "TEST.BAS" If an extension is not specified ".BAS" will be added to the file name.

There are System specific settings and debug features, which are shown in the following table 4.2.13.

Table 4.2.13: *Some Settings and Debug*

Basic Interpreter
Copyright Show MMBASIC copyright.
Clear Delete all variables and recover the memory used by them. See **Erase** for deleting specific array variables.
Erase *variable*[,*variable*]... Deletes arrayed variables and frees up the memory. Use **Clear** to delete all variables including all arrayed variables.
Error [*errormsg$*] Forces an error and terminates the program. This is normally used in debugging or to trap events that should not occur.
Memory List the amount of memory currently in use. For example: *1kB (1%) Program (9 lines)* *1kB (0%) 5 Variables* *0kB (0%) General* *98kB (99%) Free* Program memory is cleared by the *New* command. Variable and the general memory spaces are cleared by many commands (e.g., **New**, **Run**, **Load**, etc) as well as the specific commands **Clear** and **Erase**. General memory is used by streams, file I/O buffers, etc.
Continued on next page

Table 4.2.13

Basic Interpreter, continued from previous page

MM.VER

The version number of the firmware in the form aa.bbcc where aa is the major version number, bb is the minor version number and cc is the revision number (normally zero but A = 01, B = 02, etc).

MM.FNAME$

The name of the file that will be used as the default for the **Save** command. This is set by **Load**, **Run** and **Save**.

MM.CMDLINE$

The command line used with the implied **Run** command. See the implied **Run** command at the start of the next page for the details.

MM.ERRNO

Is set to the error number if a statement involving the NOR flash fails or zero if the operation succeeds. This value is not supported yet.

Peek(**VAR** *var*, *+-offset*)

Will return a byte within MMBASIC variable memory space.
You can access the memory allocated to a variable by using the variable's name (*var*) preceded by the keyword **VAR**. This can be used to access the individual bytes of a numeric variable or a large segment of RAM allocated to an array (the first element of an array (eg, nbr(0)) is the start of RAM allocated to the whole array). *offset* is the offset in the memory space. This depends on the processor architecture and might be incompatible to future EMDO designs. For this reason, We recommend to use the more versatile **Conv** command for any type of data conversions.

config$=**Sys.Get** (*device$*, *config-item$*)
Sys.Set *device$*, *config$*

The system settings can be modified with the getter and setter function. Please see subsection **??** for an overview. Not all settings can be overwritten. All arguments are handled as strings.
For example: **Sys.Set** "rs485-2", "baud=115200 data=8 stop=1 parity=n"
baudrate% = **Sys.Get** ("rs485", "baud")

Syslog *log-level*, *log*

This mechanism allows to collect logs from EMDO devices centrally and analyze the logs. The open source software Logstash and Kibana is used for this.

System *command$*

Continued on next page

Table 4.2.13
Basic Interpreter, continued from previous page
Execute a command in the EMDO operating system's command line interface, please see subsection **??**. For example: **System** "stat s0 clear"
board-temperature = **Temperature**() This function returns the EMDO mainboard temperature in °Celsius. This temperature reading is biased by the current processor load. It is suitable to measure the enclosure temperature, but not suitable to measure a room temperature. It is recommended to use a 1-wire temperature sensor instead.
Trace On **Tron** **Trace Off** **Troff** **Trace Extended** **Trace List** Turns on/off the trace facility. This facility will print the number of each line (counting from the beginning of the program) in square brackets as the program is executed. This is useful in debugging programs. In some situations more verbose output can be activated with **Extended**. As logging has an impact on program execution speed, it is sometime better to list the trace log when the program is idle (**Trace List**) rather than all the time (**Trace On**).
Wdt Call this command to enable the watchdog. It is disabled by default.

The EMDO supports the OASIS Standard MQTT v3.1.1. The standard is freely available from mqtt homepage, see chapter 7.1.
MQTT (MQ Telemetry Transport) is a protocol for machine to machine communication. It is a publish/subscribe protocol, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimize network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery.
These principles also turn out to make the protocol ideal of the emerging "machine-to-machine" (M2M) or "Internet of Things" world of connected devices. With this connectivity, it is extremely important that modern encryption technologies are used as much as possible, otherwise private data quickly become public.
For certificate generation please see subsection **??**.
The following table 4.2.14 describes all MQTT commands.

Table 4.2.14: *MQTT*

Basic Interpreter
err% = **MQTTConnect**(*server$, port%, id$, connection-flag$, user$, pass$, willtopic$, willmessage$, timeout%*) **MQTTConnectTLS**(*server$, port, id$, connection-flag$, caCert$, cliCert$, cliKey$, user$, pass$, willtopic$, willmessage$, timeout%*) Establish a connection to a MQTT broker. We recommend to use the TLS version of the command if you publish or subscribe private data in the cloud. *server$* server IP or domain name, use IPMQTT default from network tab on empty string. *port* server port, if 0 default TCP port 1883 is used. *id$* this is a unique id for a client, best use EMDO unique id **Sys.Get**(*"asset","serialID"*) *connection-flag$* for example "C0", where C = Clean session R = Will Retain (must be not be set if *willmessage* is empty) 0 = Will QoS0 (at most once delivery) 1 = Will QoS1 (at least once delivery) 2 = Will QoS2 (exactly once delivery) *caCert$* file name of the certification authority signed certificate (TLS only) *cliCert$* file name of the client certificate (TLS only) *cliKey$* file name of the client key (TLS only) *user$* authentification username (empty if unused) *pass$* authentification passwort (empty if unused) *willtopic$* will topic (empty if unused) *willmessage$* will message (empty if unused) *timeout%* communication timeout (in ms, -1 if forever) Return value 0 on success or error code with negative sign.
err% = **MQTTDisconnect**(*timeout*) Wait for disconnect from MQTT broker within the given *timeout%*. Return value 0 on success or error code with negative sign.
err% = **MQTTPublish**(*flag$*, *topic$*, *payload$*, *timeout%*) Publish a message through the MQTT broker within the given *timeout*. *flag$* for example "DR0", where D = DUP flag (this is a possible redelivery) R = Retain flag (message must be stored by broker and delivered to new subscribers) 0 = QoS0 (at most once delivery) 1 = QoS1 (at least once delivery) 2 = QoS2 (exactly once delivery) Return value 0 on success or error code with negative sign.
Continued on next page

Table 4.2.14
Basic Interpreter, continued from previous page
err% = **MQTTSubscribe**(*qos$*, *topic$*, *timeout%*) Subscribe to MQTT broker for *topic$* filter and maximal QoS level of *qos$* that the MQTT broker is allowed to deliver messages within the given *timeout%*. *qos$* for example "0", where 0 = QoS0 1 = QoS1 2 = QoS2 Return value 0 on success or error code with negative sign.
err% = **MQTTUnsubscribe**(*topic$*, *timeout%*) Unsubscribe from MQTT broker topic *topic$* within the given *timeout%*. Return value 0 on success or error code with negative sign.
num-of-messages% = **MQTTSubscription**(*flag$*, *topic$*, *payload$*) Poll a received message from the MQTT message queue. The string variables *flag$*, *topic$* and *payload$* will be overwritten by the message. If the message is longer than the maximal string lenght, it will be truncated. Return value is the number of messages retrieved (*num-of-messages%* is 1 on success).

The D0 interface (IEC62056-2) has several operation modes. Typically the device is used in mode C (active) or mode (U) passive mode. Pls make sure the interface is correctly configured.
newline This example configures the interface to passive mode:
Sys.Set "d0", "mode=U baud=19200 data=8 stop=1 parity=n echo=0 auto-parity=0 strip-parity=0 autoread=0".
If the EMDO is configured as D0 to TCP/IP Gateway, the autoread mode queries a new read cycle automatically on each connection (mode C and E only). With this mechanism it is possible to read D0 meter periodically, e.g. via Openhab or FHEM by a simple query script.
The following table 4.2.15 describes the interface.

Table 4.2.15: *D0 reader*

Basic Interpreter
success% = **D0Start**(*timeout%*) Start a new D0 active readout cycle within the given *timeout%* (in ms, -1 if forever). Energy meter must be configured in mode A,B,C and E (see IEC62056-2). Mode U is a passive mode and does not require any active readout cycle start. Energy meter periodically outputs all OBIS codes. Return positive value on success.
Continued on next page

Table 4.2.15
Basic Interpreter, continued from previous page
success% = **D0End**(*timeout%*) End currently active D0 readout cycle within the given *timeout%*. Returns positive value on success.
*OBIS-code$* = **D0ReadLn$**(*timeout%*) Return an OBIS code (*OBIS-string$*) per call within the given *timeout%*. Return an empty string on end.
*string$* = **D0Read$**(*num-of-chars%*) Read a string with length *num-of-chars%* from the D0 interface. Return the *string$*. This call is blocking and should only be used in blocking mode.
char-int% = **D0Read**() Read a character from the D0 interface and return an ASCII value. This call is blocking and should only be used in passive mode.
num-of-chars-written% = **D0Write**([*string$*, *char-int%*, *char-float*]) Write the characters from argument to the D0 interface. This call is blocking until all arguments has been sent.

RS485 is a common communication interface. The EMDO does not have any galvanic isolation on the RS485. So make sure that the connected devices have galvanic isolation (most do). If this is not the case, use a RS485 to Ethernet Gateway and connect the device over Ethernet.
RS485 must have termination resistors on both ends of the twisted pair line. EMDO has a software configurable termination resistor incorporated.
RS485 devices use different settings for baudrate, parity and framing. These settings must be configured for all bus participants.
For example **Sys.Set "rs485-2", "baud=115200 data=8 stop=1 parity=n"**.
The following table describes the interface 4.2.16.

Table 4.2.16: *RS485*

Basic Interpreter
*string$*=**RS485Read$**(*device%*, *num-of-chars%*, *timeout%*) Read string of *num-of-chars%* characters from RS485 on port *device%* (1=first, top left in mounted enclosure front view), within *timeout%* ms (in ms, -1 if forever).
Continued on next page

Table 4.2.16
Basic Interpreter, continued from previous page
*string$* = **RS485ReadLn$**(*device%*, *timeout%*, *end-of-line-token$*) Read string terminated with token *end-of-line-token$* from RS485 on port *device%* (1=first, top left in mounted enclosure front view), within *timeout%* ms (in ms, -1 if forever). Some protocol use a fixed line termination like carriage return, line feed at the end of each message.
char-int% = **RS485Read**(*device%*) Read one character from RS485 port *device%* and return character (*char-int%*). If no character is in the queue, return a negative value.
num-of-chars-written% = **RS485Write**(*device%*[, *string$*, *char-int%*, *char-float*]) Write over RS485 *string$*, *char-int%* or *char-float* on port *device%*. Return *num-of-chars-written%*.

Many devices have network connectivity. The EMDO supports TCP and UDP sockets for server and client. The following table 4.2.17 describes the commands.

Table 4.2.17: *Network Socket*

Basic Interpreter
socket% = **SocketServer** (*isTCP%*, *port%*) Opens a TCP (*isTCP%* = 1) or UDP server (*isTCP%* = 0) socket on port *port%* on the EMDO. On success, the connection handle (*socket%*) is returned. Negative value on error. EMDO uses the following ports 23 (telnet TCP), 80 (HTTP TCP), 514 (rsh, TCP), 2020 (TCP to D0 Gateway), 2021 (TCP to RS485 Gateway), 2023 (TCP to RS485-2 Gateway), 9522 (Speedwire, UDP), 1900 (SSDP UDP). These ports are not available.
socket% = **SocketClient**(*isTCP%*, *server$*, *port%*) Opens a TCP (*isTCP%* = 1) or UDP client (*isTCP%* = 0) connection to the remote *server$* (IP or domain name) on port *port%*. On success, the connection handle (*socket%*) is returned. Negative value on error.
is-connected% = **SocketConnected**(*socket%*) Check if the TCP connection *socket%* to the server is still established. Return *is-connected%* 0 if not connected anymore.
char-int% = **SocketRead** (*socket%*) Read one character from *socket%* and return character (*char-int%*).
Continued on next page

Table 4.2.17

Basic Interpreter, continued from previous page

*string$* = **SocketRead$**(*socket%*, *num-of-chars%*)

Read string of *num-of-chars%* characters from *socket%*.

*string$* = **SocketReadLn$**(*socket%*, *timeout%*, *end-of-line-token$*)

Read string terminated with token *end-of-line-token$* from *socket%* within *timeout%* ms (in ms, -1 if forever).

num-of-chars-written% = **SocketWrite**(*socket%*[, *string$*, *char-int%*, *char-float*])

Write over *socket%* the character stream consisting of *string$*, *char-int%* or *char-float*. Return *num-of-chars-written%*.

err% = **SocketOption**(*socket%*, *option$*[, *arguments*, ...])

Configure the *socket%* options. Returns negative value on error *err%*.
Options are:
SO_RCVTIMEO set the *socket%* receive timeout, use argument *rx-timeout%* in miliseconds. Please also refer to **SetTimeout** and **CheckTimeout** for details about monitoring network timeouts. This option is does timeout handling on network stack level, not application level.
SO_SNDTIMEO set the *socket%* transmit timeout, use argument *rx-timeout%* in miliseconds.Please also refer to **SetTimeout** and **CheckTimeout** for details about monitoring network timeouts. This option is does timeout handling on network stack level, not application level.
SO_IP_ADD_MEMBERSHIP add the *socket%* to multicast group, use argument *multicast-group-ip$* of the multicast group the socket has to enter.
SO_IP_DROP_MEMBERSHIP remove the *socket%* from multicast group, use argument *multicast-group-ip$* of the multicast group the socket has to be removed from.

bitset% = **StreamSearch$**(*input-stream-function*, *token-array$*[, *additional-token-arrays$*[, *timeout%*]])

This function reads a data stream from *input-stream-function* by calling the function in a loop until the start token (*token-array$*(0)) and end token *token-array(1)$* are found or the *timeout%* (in ms) has been reached. The found string is stored in the variable where *token-array$*(2) is pointing to. It then returns the *bitset%* of tokens found (1 = first token found, 2= second token found..). A maximum of 32 tokens can be searched at a time. If all tokens are found, this would result in a return value of &HFFFFFFFF.
For example:
DIM *dataSet$*(3) *dataSet$*(0)="start" *dataSet$*(1)="end" *dataSet$*(1)="value" *found%*=**StreamSearch$**(SocketRead(con%), dataSet$,5000) IF found% and 1 THEN print value$ ENDIF

closed-socket% = **SocketClose**(*socket%*)

Closing communication *socket%*. The function returns *closed-socket%* equal to *socket%* on success.

The EMDO has a S0 inputs, S0 outputs and relay output drivers. All these signals are mapped on the following S0 commands, see table 4.2.18. The relay driver are aligned after the S0 outputs on **S0Out**.

Table 4.2.18: *S0 inputs, S0 outputs, relay driver*

Basic Interpreter
pulse-count% = **S0In** (*port%* [, *info$*) *pulse-count%* = **S0In** (*port%* [, *reset-counter%*) The S0 and relay outputs are highly configurable to static and dynamic modes. Prior to using a *port%* it must be configured with **SYS.SET** for its operation. Examples: ' TOn, TOff are the minimal on and off period of the pulse in ms. ThHi and ThLo the threshold values of the ADC for high and low signal detection. If the reset flag is set to 1, the current pulse counter is reset. **SYS.SET** "s0_in0", "TOn=30 TOff=30 ThHi=700 ThLo=400 reset=1" ' Read pulse-count from primary S0 input and reset the pulse-count. *pulse-count%* = **S0In**(0,1) ' Get static state of the first S0 input *input-state* = **S0In**(0,'s') ' Get pulse rise time of the first S0 input *input-state* = **S0In**(0,'T') ' Get pulse fall time of the first S0 input *input-state* = **S0In**(0,'t') ' Get average pulse on time of the first S0 input *input-state* = **S0In**(0,'P') ' Get average pulse off time of the first S0 input *input-state* = **S0In**(0,'p') The average pulse times are interesting to calculate average value e.g. for power production or consumption from the counts.
Continued on next page

Table 4.2.18
Basic Interpreter, continued from previous page
S0Out *port%*, *pulse-count%* [, *blank-period%*] The S0 and relay outputs are highly configurable to several static and dynamic modes. Prior to using a *port%* it must be configured with **SYS.SET** for its operation. Examples: ' set first S0 output to 0 (phototransistor or relay driver conducting) **SYS.SET** "s0_out0", "state=0" ' set second S0 output to 1 (phototransistor or relay driver conducting) **SYS.SET** "s0_out1", "state=1" ' set second S0 output to pulsed mode with TOn and TOff period of 30m. **SYS.SET** "s0_out1", "TOn=30000 TOff=30000 mode=1" ' generate 100 pulses (*pulse-count%*) on second S0 output **S0Out** 1,100 ' generate 2 pulses (*pulse-count%*) on second S0 output with 1 second *blank-period*. This feature is usefull if the device receiving the pulse measures the time between the pulses to calculate a power value from it. **S0Out** 1,2, 1000

The EMDO can plan events in the future. For this purpose a rich set of commands is available. The following table 4.2.19 describes those commands.

Table 4.2.19: *Timer and cron scheduler*

Basic Interpreter
SetTimeout(*timeout%*, *remaining-time%*, *timeout-state%*) For many application it is essential to have some kind of timeout monitoring. Set a *timeout%* for an application in ms. The integer variables *remaining-time%* and *timeout-state%* will be used by the BASIC interpreter to do internal calculations of the timeout. Use **CheckTimeout** to check if the timer has expired.
expired = **CheckTimeout** (*remaining-time%*, *timeout-state%*) Check if the timeout has occured. Function return 0 is no timeout has occured (*expired*). The variables *remaining-time%* and *timeout-state%* must have been initialized with **SetTimeout** prior to calling **CheckTimeout**.
Dispatch *timeout%* This command is similar to the **Pause** but monitors the scheduler for any planned timer or cron calls.
Continued on next page

Table 4.2.19

Basic Interpreter, continued from previous page

object-count = ()
object-property-count = (*object-id$*)
object-property-count = (*object-index*)
object-property = (*object-id$*, *property$*)
object-property = (*object-index*, *property$*)

The OApi (Object API) interface allows the data exchange with the EMDO operating system (e.g. SEMP, Speedwire, 1-Wire). Pls refer to the corresponding libraries for details (**??**).
for example:

```
' Query total objects quantity
@()
' Get object [by name] property [by name]
@("Dishwasher1","SEMP/Identification/DeviceId")
' Get object [by index] property [by name]
@(0,"SEMP/Identification/DeviceId")
' Get object property by name in case of array property
@("Dishwasher1","SEMP/DeviceStatus/PowerConsumption/AveragePower()", 1)
' Get array property length by name
@("Dishwasher1","SEMP/DeviceStatus/PowerConsumption/AveragePower()")
' Get object [by name] property [by index]
@("Dishwasher1",0)
' Get object [by name] property [by index] in case of array property
@("Dishwasher1", 0, 0)
' Get object [by index] property [by index]
@(0,0)
' Get object properties count
@("Dishwasher1")
@(0)
```

Continued on next page

Table 4.2.19

Basic Interpreter, continued from previous page

```
' Set object property
@("Dishwasher1","SEMP/Identification/DeviceId","122200-0000-0000")
' Set object property by name in case of array property
@("Dishwasher1","SEMP/DeviceStatus/PowerConsumption/AveragePower", 1, 1500)
' Set object property by index in case of array property
@("Dishwasher1", 0, 1, 1500)
' Delete object specified by name or index
@("Dishwasher1", "#DELETE")
@(0, "#DELETE")
' Delete property specified by name or index
@("Dishwasher1", "#DELETE", "SEMP/Identification/DeviceId" )
@(0, "#DELETE", 0)
' Query object name
@(0, "#NAME")
' Query property name
@("Dishwasher1", "#NAME", 0)
@(0, "#NAME", 0)
' Query property type
@("Dishwasher1", "#TYPE", "SEMP/Identification/DeviceId")
@("Dishwasher1", "#TYPE", 0)
@(0, "#TYPE", 0)
' Query property length
@("Dishwasher1", "#LEN", "SEMP/Identification/DeviceId")
@("Dishwasher1", "#LEN", 0)
@(0, "#LEN", 0)
```

unix-time-stamp=**Unixtime**()

Return the current time *unix-time-stamp*. The unix time is counting the seconds since 1.1.1970 (UTC). It will overflow at 19.1.2038. The EMDO unix time depends on the global network time protocol which is distributed globally and derived from a precise atomic clock and corrected to universal time (solar time). All logging in the EMDO is done with Coordinated Universal Time (UTC) as it is the most correlated to earth rotation and solar cycle.

ticks = **Ticks**()

This function returns the system ticks. It is a 32 bit register which overflows around every 24 days. For timeout calculation you better use the functions **SetTimeout** and **CheckTimeout**.

*time-stamp$* = **Timestamp$**()

This returns either the unix time, if the EMDO is synchronized to a NTP server, or the seconds since the EMDO has been powered up.

*iso-time-stamp$* = **Date$**()

This function return an ISO 8601 formatted timestring. e.g.1988-04-07T18:39:09-08:00.

Continued on next page

Table 4.2.19

Basic Interpreter, continued from previous page

unix-timestamp%=**TimeFromISO**(*iso-time-stamp$*)TimeFromISO)

Convert a ISO 8601 formatted timestring into *unix-timestamp%*.

week-day%=**Weekday**

This function returns the day of the week (e.g. 1 = monday, ... , 7=sunday).

year-day%=**Yearday**()

This function returns the day of the year (e.g. 1= first day of january, ... 365 = last day of december).

timer-descriptor% = **SetTimer**(*timeout%* [, *one-shot%*])
On Timer *timer-descriptor% time-handler-function* [**Detach**] *err%* = **KillTimer**(*timer-descriptor%*)

Plan returning events in the range of miliseconds to minutes (in ms resolution). With **SetTimer** the timer is configured to a *timeout%*. As an optional argument *one-shot%* must be set to 1 if the timer is a one shot timer (does not retrigger itself). The function returns *timer-descriptor%*. The value is negative on error.
Next the *time-handler-function* is registered, which is called every time the timer expires.
The handler function is called within the **Dispatch** statement.
The timer can be disabled with **KillTimer**. The return value is negative on error. Once the timer is killed, it can be unregistered again with the **DETACH** statement.
The jitter of the timer is a view milliseconds and should be fine for most applications. Example:

```
td%=SetTimer(200)
IF td% < 0 THEN ERROR "Failed to create timer"
ON TIMER td% thandler
Dispatch 1100
sc%=KillTimer(td%)
IF sc%<0 THEN ERROR "Failed to kill timer"
ON TIMER td% thandler DETACH
IF MM.errno < 0 THEN ERROR "Failed to detach handler"
FUNCTION thandler(id%)
IF id% <> td% THEN ERROR "id mismatch"
END FUNCTION
```

Continued on next page

Table 4.2.19

Basic Interpreter, continued from previous page

cron-descriptor% = **CrontabAdd**(*cron-string$*)
On Cron *cron-descriptor% cron-handler-function* [**Detach**]
err% = **CrontabRemove** (*cron-descriptor%*)

Tasks that run on a minute resolution can be triggered as cron jobs. The commands are very similar to the Timer commands. The main difference is that the CRON is not made for milisecond time base, but rather minutes. The *cron-string$* describes the scheduling of the task in analogy to unix cron jobs (see below examples). **CrontabAdd** adds the job to the cron system and the registration of the *cron-handler-function* is done with the *cron-descriptor%* via **ON CRON** command. The unregistration is done the same way with the additional **DETACH** token.
Example:

```
' Minute, hour, day, month, weekday
' * * * * * is every minute
' 17 * * * * hourly at xx:17
' 17 6 * * * daily at 6:17
' 17 18 * * 5 is every friday at 18:17
' 0 0 1 * * is monthly on the 1st
' */5 * * * * every 5 minutes
' * * * * 1 every monday
cd%=CrontabAdd("* * * * *")
IF cd% < 0 THEN ERROR "Failed to create cron"
ON CRON cd% chandler
Dispatch 120000
sc%=CrontabRemove(td%)
IF sc%<0 THEN ERROR "Failed to remove cron"
ON CRON cd% chandler DETACH
IF MM.errno < 0 THEN ERROR "Failed to detach handler"
FUNCTION chandler(id%)
IF id% <> cd% THEN ERROR "id mismatch"
END FUNCTION
```

4.3 EMDO addon board commands

The EnOcean Radio addon board enables two BASIC commands to transceive EnOcean telegrams.
The addon board uses a TCM310 module.
The user manuals of the chip are available from manufacturer `https://www.enocean.com/de/enocean_module/tcm-310/`.
The following table 4.3.1 describes the commands.

Table 4.3.1: *Enocean*

Basic Interpreter
err% = **EnoceanReceive**(*type%*, *data$*, *optdata$*, *timeout%*) Receive an EnOcean telegram from queue within *timeout%* ms. Return value *err%* is 0 on successfull reception of a telegram, negative on error. The basic variables *type%*, *data$* and *optdata$* contain the message on successful reception of a telegram.
err% = **EnoceanTransmit**(*type%*, *data$*, *optdata$*, *timeout%*) Transmit an EnOcean telegram *type%*, *data$* and *optdata$* within *timeout%* ms. Return value is 0 on successful transmission of the telegram, negative on error.

4.4 EMDO libraries

We are still working on many parts of the libraries. The idea is to have a simple way configure most applications for the EMDO. Please see page 111 for github reference.

4.5 Sys API

EMDO firmware allows to adjust its behavior by calling this API. Firmware is naturally built from a set of modules, each one module controlling one part of hardware or software.
Each module has a set of variables which describe this module. Variables can be read-only, writable, write-only, and read-write. The general form of Sys API is

var = Sys.Get(module_name_expression, var_name_expression) Sys.Set module_name_expression, var_value_expression
SYNTAX
Sys.Get(module_name, var_name) module_name is a string expression which specify firmware module name. var_name is a string expression which specify module variable to query.
Sys.Set module_name, var_values_list var_values_list = var_value_expression
{space var_value_expression} var_value_expression = var_name "=" value
var_values_list is a string expression which specify a list of variable with their values

EXAMPLES
baud%=**Sys.Get**("rs485","baud")
Sys.Set "rs485", "baud=115200 data=8 stop=1 parity=n"

EXCEPTIONS
Both Sys.Get function and Sys.Set statement can return error. Non-fatal errors returned in MM.Errno variable.
Fatal ones raises basic error.
MODULES

Table 4.5.1: *EMDO identification module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
uuid	string	RO	-	manufacturer assigned device unique identifier
type	int	RO	-	device type 0 = EMDO101 1 = EMDO102 2 = EMDO103 3 = EMDO104
revision	string	RO	-	PCA revision
mac	string	RO	-	device MAC-address
fabricationDate	string	RO	-	date of device fabrication
serial	string	RO	-	manufacturer assigned device serial number
serialID	string	RO	-	manufacturer assigned device serial ID used for portal authorization
firmwareVersion	string	RO	-	firmware version
ffsVersion	string	RO	-	onboard flash file system version
addon_uuid	string	RO	-	manufacturer assigned addon board unique identifier

Continued on next page

Table 4.5.1				
Sys API, continued from previous page				
addon_type	int	RO	-	addon board type (if any installed) 0 = ENOCEAN
addon_revision	string	RO	-	addon board PCA revision
addonFabricationDate	string	RO	-	date of addon board fabrication
addonSerial	string	RO	-	manufacturer assigned addon board serial number
addonSerialID	string	RO	-	manufacturer assigned addon board serial ID used for portal authorization
addonMAC	string	RO	-	device MAC-address

Table 4.5.2: *Primary RS485 device module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
baud	int	R,W	9600	baudrate, bps 300,600,1200,2400,4800,9600 14400,19200,28800,38400,57600,115200, 128000,230400,256000,460800,500000
data	int	R,W	8	databits = 7|8
parity	string	R,W	n	parity = 'e'|'o'|'n'
stop	int	R,W	1	stop bits = 1|2
echo	int	R,W	0	local echo = 0|1
auto-parity	int	R,W	0	auto generate parity bits = 0|1
strip-parity	int	R,W	0	strip parity bits from data stream = 0|1
term	int	R,W	1	use line termination resistors = 0|1
polarity	int	R,W	1	data polarity = 0|1

Table 4.5.3: *Secondary RS485 device module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
baud	int	R,W	9600	baudrate, bps 300,600,1200,2400,4800,9600, 14400,19200,28800,38400,57600,115200, 128000,230400,256000,460800,500000
data	int	R,W	8	databits = 7|8
parity	string	R,W	n	parity = 'e'|'o'|'n'

Continued on next page

Table 4.5.3				
Sys API, continued from previous page				
stop	int	R,W	1	stop bits = 1|2
echo	int	R,W	0	local echo = 0|1
auto-parity	int	R,W	0	auto generate parity bits = 0|1
strip-parity	int	R,W	0	strip parity bits from data stream = 0|1
term	int	R,W	1	use line termination resistors = 0|1
polarity	int	R,W	1	data polarity = 0|1

Table 4.5.4: *D0 device module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
if	int	R,W	1	RS485 interface to use for D0 head access (available only on DIN-rail devices) = 0|1
mode	string	R,W	C	protocol mode = 'A'|'B'|'C'|'D'|'E'|'U'
baud	int	R,W	19200	baudrate, bps 300,600,1200,2400,4800,9600,19200
data	int	R,W	7	databits = 7|8
parity	string	R,W	n	parity = 'e'|'o'|'n'
stop	int	R,W	1	stop bits = 1|2
echo	int	R,W	0	local echo = 0|1
auto-parity	int	R,W	0	auto generate parity bits = 0|1
strip-parity	int	R,W	0	strip parity bits from data stream = 0|1
autoread	int	R,W	1	use auto read mode in protocol modes C and E

Table 4.5.5: *Temperature onboard temperature sensor module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
val	int	RO	-	onboard temperature

Table 4.5.6: *S0 input 0 device module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
TOn	int	R,W	30	impulse detection ON duration, ms = 1..1000
TOff	int	R,W	30	impulse detection OFF duration, ms = 1..1000
ThHi	int	R,W	800	impulse ON ADC threshold, 1/10 % = 10..990

Continued on next page

Table 4.5.6				
Sys API, continued from previous page				
ThLo	int	R,W	300	impulse OFF ADC threshold, 1/10 % = 10..990
pulses	int	RO	-	detected impulses
reset	int	WO	-	reset detected impulses counter
retrain	int	WO	-	retrain detection

Table 4.5.7: *S0 input 1 device module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
TOn	int	R,W	30	impulse detection ON duration, ms = 1..1000
TOff	int	R,W	30	impulse detection OFF duration, ms = 1..1000
ThHi	int	R,W	800	impulse ON ADC threshold, 1/10 % = 10..990
ThLo	int	R,W	300	impulse OFF ADC threshold, 1/10 % = 10..990
pulses	int	RO	-	detected impulses
reset	int	WO	-	reset detected impulses counter
retrain	int	WO	-	retrain detection

Table 4.5.8

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
TOn	int	R,W	30	impulse detection ON duration, milliseconds = 1..1000
TOff	int	R,W	30	impulse detection OFF duration, milliseconds = 1..1000
ThHi	int	R,W	800	impulse ON ADC threshold, 1/10 % = 10..990
ThLo	int	R,W	300	impulse OFF ADC threshold, 1/10 % = 10..990
pulses	int	RO	-	detected impulses
reset	int	WO	-	reset detected impulses counter
retrain	int	WO	-	retrain detection

Table 4.5.9

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
TOn	int	R,W	30000	impulse ON duration, microseconds = 10..1676999
TOff	int	R,W	30000	impulse OFF duration, microseconds = 10..1676999

Continued on next page

Table 4.5.9				
Sys API, continued from previous page				
period	int	R,W	60000	PWM period duration, microseconds = 30..1677000
duty	int	R,W	30000	PWM duty duration, microseconds = 10..1676999
polarity	int	R,W	0	signal polarity = 0|1
mode	int	R,W	0	output mode = 0..2 0 = digital out 1 = pulsed out 2 = PWM
state	int	R,W	0	output value = 0|1
pulses	int	WO	-	amount of pulses to generate

Table 4.5.10: *S0 output 1 device module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
TOn	int	R,W	30000	impulse ON duration, microseconds = 10..1676999
TOff	int	R,W	30000	impulse OFF duration, microseconds = 10..1676999
period	int	R,W	60000	PWM period duration, microseconds = 30..1677000
duty	int	R,W	30000	PWM duty duration, microseconds = 10..1676999
polarity	int	R,W	0	signal polarity = 0|1
mode	int	R,W	0	output mode = 0..2 0 = digital out 1 = pulsed out 2 = PWM
state	int	R,W	0	output value = 0|1
pulses	int	WO		amount of pulses to generate

Table 4.5.11: *S0 output 2 device module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
TOn	int	R,W	30000	impulse ON duration, microseconds = 10..1676999
TOff	int	R,W	30000	impulse OFF duration, microseconds = 10..1676999
period	int	R,W	60000	PWM period duration, microseconds = 30..1677000
duty	int	R,W	30000	PWM duty duration, microseconds = 10..1676999
polarity	int	R,W	0	signal polarity = 0|1

Continued on next page

Table 4.5.11				
Sys API, continued from previous page				
mode	int	R,W	0	output mode = 0..2 0 = digital out 1 = pulsed out 2 = PWM
state	int	R,W	0	output value = 0|1
pulses	int	WO		amount of pulses to generate

Table 4.5.12: *S0 output 3 device module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
TOn	int	R,W	30000	impulse ON duration, microseconds = 10..1676999
TOff	int	R,W	30000	impulse OFF duration, microseconds = 10..1676999
period	int	R,W	60000	PWM period duration, microseconds = 30..1677000
duty	int	R,W	30000	PWM duty duration, microseconds = 10..1676999
polarity	int	R,W	0	signal polarity = 0|1
mode	int	R,W	0	output mode = 0..2 0 = digital out 1 = pulsed out 2 = PWM
state	int	R,W	0	output value = 0|1
pulses	int	WO		amount of pulses to generate

Table 4.5.13: *S0 output 3 device module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
dhcp	int	R,W	1	Use DHCP
rsh	int	R,W	0	Use RSH service
d0_gw	int	R,W	0	Use TCP-to-D0 gateway service
rs485_gw	int	R,W	0	Use TCP-to-RS485 gateway service
rs485_2_gw	int	R,W	0	Use TCP-to-RS485 gateway service(available only on DIN-rail)
addr	string	RO	-	Current IP address
mask	string	RO	-	Current network mask
gw	string	RO	-	Current gateway address
dns1	string	RO	-	Current DNS server address
addr_cfg	string	R,W	-	Manual IP address to use (valid only if dhcp=0), x.x.x.x

Continued on next page

Table 4.5.13				
Sys API, continued from previous page				
mask_cfg	string	R,W	-	Manual network mask to use (valid only if dhcp=0), x.x.x.x
gw_cfg	string	R,W	-	Manual gateway address to use (valid only if dhcp=0), x.x.x.x
dns1_cfg	string	R,W	-	Manual DNS server address to use (valid only if dhcp=0), x.x.x.x
ntp	string	R,W	-	Manual NTP server address to use (valid only if dhcp=0), FQDN|x.x.x.x
mqtt	string	R,W	-	MQTT server address to use, FQDN|x.x.x.x
rlog	string	R,W	-	RSYSLOG server address to use, FQDN|x.x.x.x

Table 4.5.14: *ENOCEAN addon module*

Sys API Setter and Getter				
Var	**Type**	**Access**	**Def val**	**Description**
id	string	RO	-	ENOCEAN chip id

4.6 Command Line Interface

Command Line Interface (CLI)
We love oldschool freedom :) that is why every single aspect of EMDO can be controlled by comand line. We use this interface during continous testing and production procedures. So, why don't you use it too ;)?
There are two ways of using CLI:
- Telnet session
- RSH session

EMDO fully support TELNET protocol (RFC 854 - https://tools.ietf.org/html/rfc854) on port 23. You can use any telnet client to communicate with EMDO. Telnet sessions quantity limited to one active only.
EMDO fully support RSH protocol on port 514 with RCP limited only to host->device direction. You can use any RSH client to communicate with EMDO. RSH sessions quantity limited to one active only.
Active telnet session also used for local logging console:

EXAMPLE logging: 167385465,+167385465: <2>HTTP

$$80023ce0, src/http/connection.c : 1092$$

Initial: 192.168.4.4, /description.xml
167385466,+1: <2>HTTP

$$80023ce0, src/http/https_rv.c : 377$$

GET: /description.xml
167385469,+3: <2>HTTP

$$80023ce0, src/http/https_rv.c : 172$$

— SESSION CLOSED

EMDO uses ANSI escape sequences for coloring output and cursor positions, so normally you should not disable ANSI colors in your terminal settings.
To execute command you should type it after command line prompt "emdo $" and hit ENTER. EMDO keep command line history limited to 256 bytes buffer. You can use Up and Down keys to move between entries in a history.

KEYS	
Up | Esc+A	Move to previous line in a history
Down | Esc+B	Move to next line in a history
Right | Esc+C	Move cursor to the right by one char
Left | Esc+D	Move cursor to the left by one char
Delete	Delete one char after cursor
Backspace	Delete one char before cursor
Home	Move cursor to the begin of prompt

End	Move cursor to the end of prompt
Tab	Command line completition. You can use it at any time - if CLI already has enough chars to complete command it will enter its full name, otherwise EMDO will show possible commands according to previous text typed.

Every command has its own help that will be shown if you typed not enough arguments for that command.
You can use CLI in BASIC scripts also by using Sys.Get(), Sys.Set() and SYSTEM() API calls.

COMMAND LIST

Table 4.6.2: *Help*

CLI Interpreter
help Shows command list *HelpCommand* Use TAB key for completion. The basic commands are: help - displays this message ...

Table 4.6.3: *Basic Interpreter*

CLI Interpreter
basic mem Dump memory usage: emdo $ **basic mem** 0 0kB (0%) Program (0 lines) 0 0kB 0 Libraries (0 lines) 0 0kB (0%) 0 Variables 0 0kB (0%) General 102400 100kB (100%) Free
basic pause Pause BASIC execution.
basic resume Resume BASIC execution.
basic restart Restart BASIC program.
Continued on next page

Table 4.6.3

CLI Interpreter, continued from previous page

basic stop

Stop BASIC program.

basic status

Dump misc.status.

basic tables

Dump internal tables.
emdo$ **basic tables**

BASIC COMMANDS

Name Tkn Type
1) Memory 0080 C
2) ? 0081 C
3) Clear 0082 C
4) Continue 0083 C
5) Data 0084 C
6) Dim 0085 C
7) Do 0086 C
8) ElseIf 0087 C
9) Else If 0088 C
10) Case Else 0089 C
...

basic vars - dump basic variables
emdo $ **basic vars**

BASIC VARIABLES
Name Type Lvl
1) KWHDAY1 f 0
2) KWHDAY2 f 0
3) STATUS_MSG s 0
4) STATUS2 s 0
...

Table 4.6.4: *Date*

CLI Interpreter
date Shows current date and time. emdo$**date** Fri Jan 6 11:39:22 2017

Table 4.6.5: *Dir*

CLI Interpreter
dir List file system contents. emdo$ **dir** 23493 output/20161216.csv 887 config/private.pem 272 http/pubkey.pem 29689 output/20161228.csv ... 1499332 http/index.htm 629338 http/index.htm.gz 73 config/firmware.cfg 1088 basic/log.blib 2507492 bytes in 42 files

Table 4.6.6: *File system maintenance CLI*

CLI Interpreter
erase File system maintenance CLI.
erase do Execute background file system format. ALL CONTENTS WILL BE ERASED! Normally you should not use this command.

Continued on next page

Table 4.6.6
CLI Interpreter, continued from previous page
erase status Show erase task progress and status. Erase procedure is long - normally it will take more than 15 minutes, so you can use this command to check it. Upon completion of erase its result will be automatically shown in console

Table 4.6.7: *Fsck*

CLI Interpreter
fsck File system optimization and clearance CLI. EMDO has 32Mb onboard SPI-flash IC, which is used to store data. Raw SPI NOR flash space is managed by SPIFFS (SPI Flash File System). Spiffs is designed with following characteristics in mind: - Only big areas of data (blocks) can be erased - An erase will reset all bits in block to ones - Writing pulls one to zeroes - Zeroes can only be pulled to ones by erase - Wear leveling (as flash has limited number of erasal cycles) Erase block operation of SPI-flash is long, due to that SPIFFS will switch to new empty blocks during lifetime and raw space will be more and more fragmented, especially when the free space became low. Previously used blocks can be returned to free space only after erasal of them. We call this process garbage collection. Firmware is automatically watching filesystem state and doing GC from time to time. With that command you can do it manually.
fsck full Do full check This operation is long and executed in background.
fsck gc Do layout optimization This operation is shorter than full optimization and also executed in background.
fsck status Show fsck status Shows current fsck status and progress. Upon completion of fsck its result will be automatically shown in console.

Table 4.6.8: *Get*

CLI Interpreter
get Allows to show system settings. EMDO firmware is naturally built from a set of modules, each one module controlling one part of hardware or software. Each module has a set of variables which describe this module. Variables can be read-only, writable, write-only, and read-write. Please, see more details at the Sys API topic. emdo $ **get** [get command] Called to show settings. get asset - show all asset variables get asset x - show asset.x value get rs485 - show all rs485 variables get rs485 x - show rs485.x value ... get ip - show all ip variables get ip x - show ip.x value get enocean - show all enocean variables ...

Table 4.6.9: *Info*

CLI Interpreter
info Shows quick information about EMDO. Detailed information can be queried by issuing 'get asset' command.

Table 4.6.10: *Log*

CLI Interpreter
log Allows to adjust EMDO logging behavior. Most of firmware modules can do detailed logging according to that module log settings. Log destinations can be local console, remote log server or/and FFS log (output/message.log). There are following log levels exist: 0 = ERROR 1 = INFO 2 = TRACE 3 = TRANSMITTED DATA DUMPING 4 = RECEIVED DATA DUMPING Log level values are not persistent and will be restored to default value (which is OFF) after reboot.
log basic on *level* Switch BASIC logging ON with optional level.
log basic off Switch logging OFF.
log basic level x Adjust logging level.
log cron Toggle CRON logging off and on.
log d0 on *level* Switch logging ON with optional level.
log d0 off Switch logging OFF.
log d0 level x Adjust logging level.
log euart Toggle addon board module logging off and on.

Continued on next page

Table 4.6.10

CLI Interpreter, continued from previous page

log enocean on *level* Switch logging ON with optional level.
log enocean off switch logging OFF.
log enocean level x Adjust logging level.
log flash on *level* Switch logging ON with optional level.
log flash off Switch logging OFF.
log flash level x Adjust logging level.
log http on *level* Switch logging ON with optional level.
log http off Switch logging OFF.
log http level x Adjust logging level.
log mqtt on *level* Switch logging ON with optional level.
log mqtt off Switch logging OFF.

Continued on next page

Table 4.6.10

CLI Interpreter, continued from previous page
log mqtt level x Adjust logging level.
log mqtt tls x Adjust logging level of TLS sessions.
log rs485 Toggle RS485 module logging off and on.
log rs485-2 Toggle RS485 module logging off and on (exists only on DIN-rail editions).
log sntp Toggle SNTP module logging off and on.
log s0 on *level* Switch logging ON with optional level.
log s0 off switch logging OFF.
log s0 level x Adjust logging level.
log 1-wire on *level* switch logging ON with optional level (exists only on DIN-rail editions).
log 1-wire off switch logging OFF (exists only on DIN-rail editions).
log 1-wire level x Adjust logging level (exists only on DIN-rail editions).
Continued on next page

Table 4.6.10

CLI Interpreter, continued from previous page

log status

Shows logging status.
emdo $ **status**

Logging to local

```
          http  = 0.1
         basic  = 0.2
          cron  = 0
            D0  = 0.2
       extUART  = 0
      ENOCEAN   = 0.2
         flash  = 0.2
          MQTT  = 0.2.1
         rs485  = 0
           rsh  = 0
            S0  = 0.4
          sntp  = 0
```

Table 4.6.11: *List*

CLI Interpreter

list

Some firmware modules allows to inquire some extra information by issuing this command.

list cron

Dump crontab:

emdo $ **list cron**

CRONTAB ———

```
 */25 * * * *  fsck_hourly
  */5 * * * *  bas1
 59 23 * * *   bas2
```

list pf

Dump packet filtering table.

Continued on next page

Table 4.6.11

CLI Interpreter, continued from previous page

list semp

Dump SEMP objects information.

list s0_in0

Dump s0_in0 history. You can use this command as impulse debugger.

list s0_in1

Dump s0_in1 history. You can use this command as impulse debugger.

list s0_in2

Dump s0_in2 history. You can use this command as impulse debugger.

list spiffs

Dump SPIFFS layout .
emdo $ **list spiffs**

0 ddddddddddddddddddddddddddddddddddidddddddddddddddddddddddddddddddd
//
//
era_cnt: 1185
1 ///ddddddddddddddddddddidddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddidddddddddddddddddddddddddddddddddddddddddddddd
d/////////////////////

—— ANNOTATIONS ——

_ : free page
/ : deleted page
i : index page
d : data page

Table 4.6.12: *Mem*

CLI Interpreter
mem Shows memory consumption emdo $ **mem** Free memory = 83520 Failed allocations = 0 Free BASIC memory = 68608

Table 4.6.13: *Memoria*

CLI Interpreter
memoria Shows exception or reboot reason if any. emdo $ **memoria** BOOT reason: POR Magic: 0xdead *VALID* Occured: Fri Jan 6 11:19:36 2017 Reason: 0x3 *TLBrefill* RunLevel: 13 Address: 0x9d0ea6a4 Task: Basic RebootCnt: 0

Table 4.6.14: *Mount*

CLI Interpreter
mount Shows SPIFFS mount status.

Continued on next page

Table 4.6.14

CLI Interpreter, continued from previous page

emdo $ **mount**

```
 fs mounted  = yes
       used  = 3493074
      total  = 29752879
      usage  = 11.74%
```

Table 4.6.15: *OTP RFC2289 authentication CLI.*

CLI Interpreter
otp status Show otp status.
otp secret Change secret passphrase interactively.
otp secret x x Change secret passphrase batchmode.
otp secret clear Cleare secret passphrase.

Table 4.6.16: *Packet filtering CLI*

CLI Interpreter
pf add ipaddr Drop all traffic from specified host
pf rm ipaddr Delete specified IP address from blacklist table
pf clear Delete all entries from blacklist table
pf list Show blacklist table contents

Table 4.6.17: *Quit*

CLI Interpreter
quit Close CLI session

Table 4.6.18: *Reboot*

CLI Interpreter
reboot Reboot CLI
reboot do - Perform hardware reboot
reboot do bbl - Perform hardware reboot with bootloader activation

Table 4.6.19: *Rm*

CLI Interpreter
rm Remove files from FFS
rm file *file* *regexp* - remove one or more files matching regular expression or full
path Regular expression can include '*' and '?' characters, where '*' mean any number of chars and '?' mean one char only. Removal any file from config/ directory is prohibited and might ruin your board.

Table 4.6.20: *Save*

CLI Interpreter
save Save full configuration into FFS.

Table 4.6.21: *Set*

CLI Interpreter
set Allows to change any module settings. See also 'get' command and Sys API.
set rs485 x=y Change primary rs485 settings.
set rs485-2 x=y Change secondary rs485 settings (DIN-rail editions only).
set D0 x=y Change D0 settings.
set s0_in0 x=y Change s0_in0 settings.
set s0_in1 x=y Change s0_in1 settings.
set s0_in2 x=y Change s0_in2 settings.
set s0_out0 x=y Change s0_out0 settings.
set s0_out1 x=y Change s0_out1 settings.
set s0_out2 x=y Change s0_out2 settings.
Continued on next page

Table 4.6.21
CLI Interpreter, continued from previous page
set ip x=y change ip settings.

Table 4.6.22: *Stat*

CLI Interpreter
stat Allows to see collected statistics. Every firmware module has its own collected lifetime statistics.
stat basic *clear* Dump/reset MMBASIC stat.
stat cpu Dump CPU stat.
stat cron *clear* Dump/reset CROND stat.
stat D0 *clear* Dump/reset D0 stat.
stat eth *clear* Dump/reset MAC stat.
stat enocean *clear* Dump/reset ENOCEAN addon stat.
Continued on next page

Table 4.6.22

CLI Interpreter, continued from previous page

stat euart

clear

Dump/reset ext.uart stat.

stat flash

clear

Dump/reset flash stat.

stat http

clear

Dump/reset HTTP stat.

stat i2c

clear

Dump/reset I2C stat.

stat igmp

clear

Dump/reset IGMP stat.

stat ip

clear

Dump/reset IP stat.

stat mqtt

clear

Dump/reset MQTT stat.

stat net

Dump network stat.

stat pf

clear

Dump/reset Packet Filter stat.

Continued on next page

Table 4.6.22

CLI Interpreter, continued from previous page

stat rs485

clear

Dump/reset primary RS485 stat.

stat rs485-2

clear

Dump/reset secondary RS485 stat (DIN-rail editions only).

stat rsh

clear

Dump/reset RSH stat.

stat s0

clear

Dump/reset S0 stat.

stat sma

clear

Dump/reset SMA Speedwire stat.

stat sntp

clear

dump/reset SNTP stat.

stat spi

clear

dump/reset SPI stat.

stat spiffs dump SPIFFS stat.

stat upnp

clear

- dump/reset UPnP stat.

Continued on next page

Table 4.6.22

CLI Interpreter, continued from previous page

stat 1-wire

clear

dump/reset 1-Wire stat (DIN-rail editions only)

stat cpu

Show CPU status with various task.
emdo $ **stat cpu**

Task	Pri	S	Abs	Time	%Time Stack
CliSrv	2	R	46488	1%	380
IDLE	0	R	21331262	92%	836
Tmr Svc	2	B	199468	<1%	780
HttpSrv	2	B	145063	<1%	2006
Basic	1	B	517580	2%	3722
OtherSr	2	B	24260	<1%	856
EMAC	2	B	12882	<1%	922
IP-task	2	B	158519	<1%	884
flightm	3	B	855	<1%	898
SysAppT	2	B	541982	2%	944
Enocean	2	B	167	<1%	1126
ISR	4	x	-	-	989

Continued on next page

Table 4.6.22

CLI Interpreter, continued from previous page

stat eth

Show ethernet status.
emdo $**stat eth**

```
Status of the ethernet interface eth0: Up, 100MBps,full-duplex
     bytesReceived = 1248732
         bytesSent = 1853662
         in-unicast = 3404
        in-Nunicast = 305
         out-unicast = 5284
       out-Nunicast = 0

    hw-transmitted = 5283
   hw-okReceived = 3709
        buffersSent = 5284
        buffersDup = 0
       bufReceived = 3709
multiBufReceived = 0

         badPackets = 0
        rxOverflows = 0
          noBuffers = 0
          collisions = 0
           fcsErrors = 0
         alignErrors = 0
        nolinkDrops = 0
 txDmaNoBuffers = 0
 rxDmaNoBuffers = 0
        rxEventLost = 0
     transmitErrors = 0
        pktTxErrors = 0
               isrs-rx = 4284
               isrs-tx = 5284
            isrs-rxErr = 0
           isrs-genErr = 0
     handledErrors = 0
      noAckBuffers = 0102B
  bufNFWRelease = 0
```

Table 4.6.23: *Uptime*

CLI Interpreter
uptime Shows EMDO uptime. emdo $ **uptime** up 00:16:23.

SECURITY

EMDO uses RFC 2289 OTP (https://tools.ietf.org/html/rfc2289) as login mechanism, so clear text passwords are never transmitted by network. After you launch telnet session you'll see the following login prompt

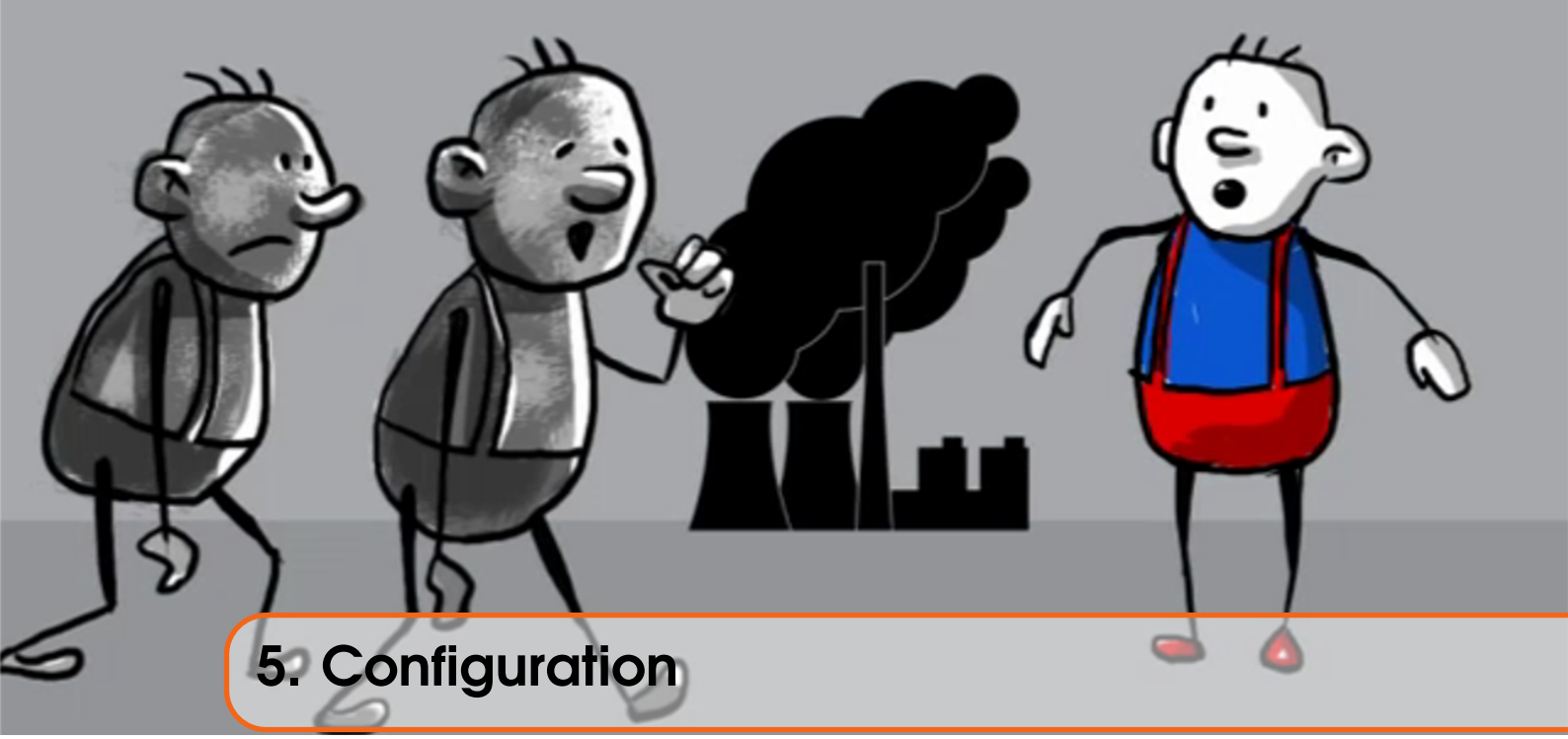
otp-md5 499 emdo-fcae04-1
response:

You can use secure online response calculator
`https://www.ocf.berkeley.edu/~jjlin/jsotp/` or Android app OTPdroid (`https://play.google.com/store/apps/details?id=de.ub0r.android.otpdroid`) or OTP Generator for iPhone (`https://itunes.apple.com/us/app/otp-generator/id294055241?mt=8&at=11l9z8&ign-mpt=uo%3D4`) to calculate response to login prompt.
Challenge is the text after 'otp-md5' word. Type in calculated response and hit Enter. If login is successfull you'll see command prompt:

response: slam make moll bien newt sell

emdo $
You can use both upper and lower case (doesn't matter) or use numeric form of reponse (such as DE56 CAF7 2BDC 3FB2). After 3 unsuccesfull retries IP adress of incoming host will be blacklisted for 60 seconds.
During that period all packets from that host will be silently dropped. EMDO uses same secret phrase for TELNET and HTTP sessions. HTTP sessions use digest authentification mechanism (use login 'admin').
EMDO is shipped with default secret 'emdocando1' and OTP activated.

WARNING: RSH protocol does not allow to use any authentification, so you should disable RSH usage in network settings if your EMDO is located in publically accessable network, or keep it behind firewall. EMDO is shipped with disabled RSH by default. RSH sessions allowed from local subnet only, other sessions will be rejected.



5. Configuration

5.0.1 Graphics

```
{
   "donuts": [
      {
         "Power consumption":"kWhDay1",
         "Power grid feed":"kWhDay2",
         "status":"status_msg$"
      },
      {
         "EMDO Temperature":"Temperature()",
         "status":"status2$"
      }
   ]
}
```

```
' test
for i = 1to 100
 print "hallo"
next i
```

5.0.2 Certificates

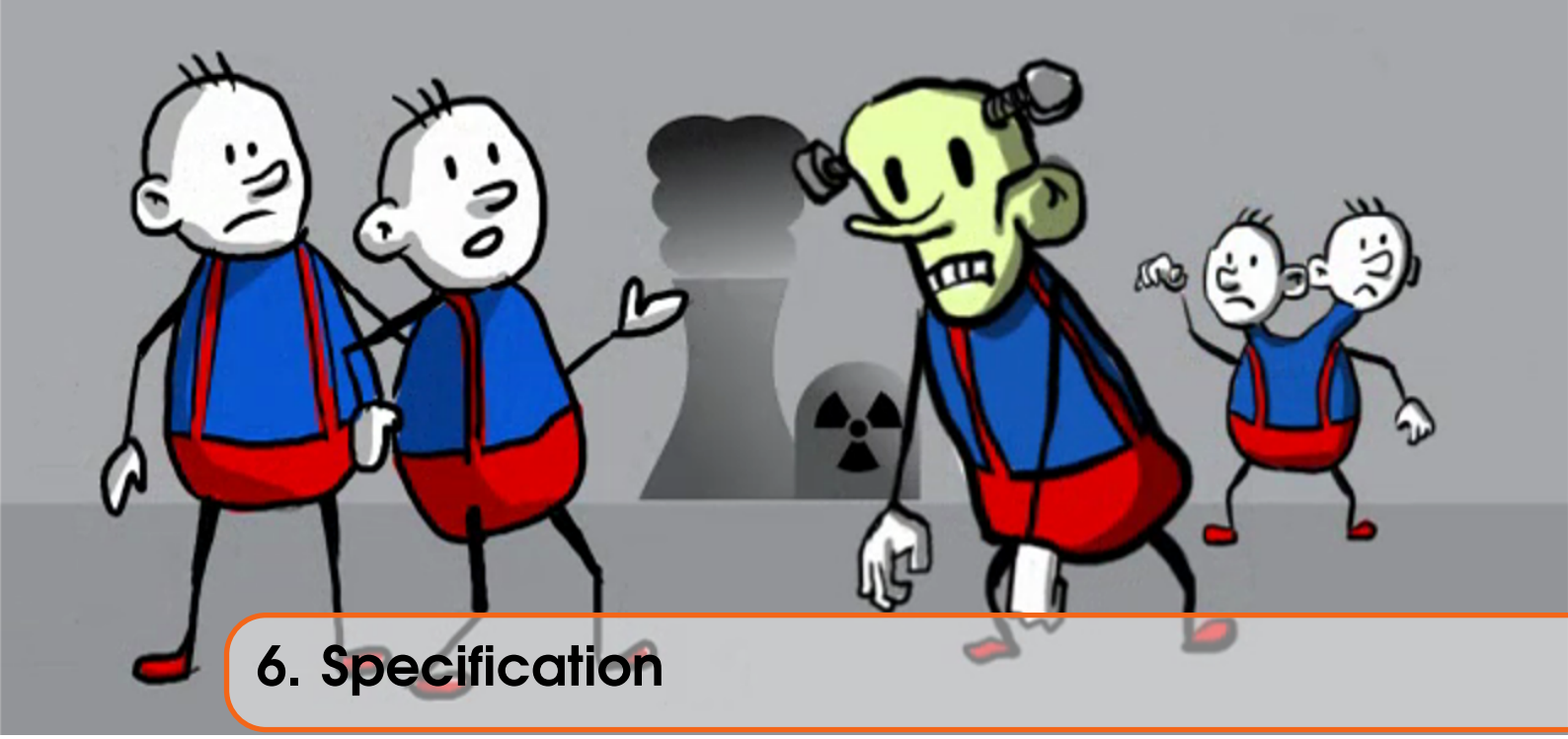
II Appendix

6 Specification 107

7 Credits and Licenses 109
7.1 Developers

Bibliography 111
7.2 References
References

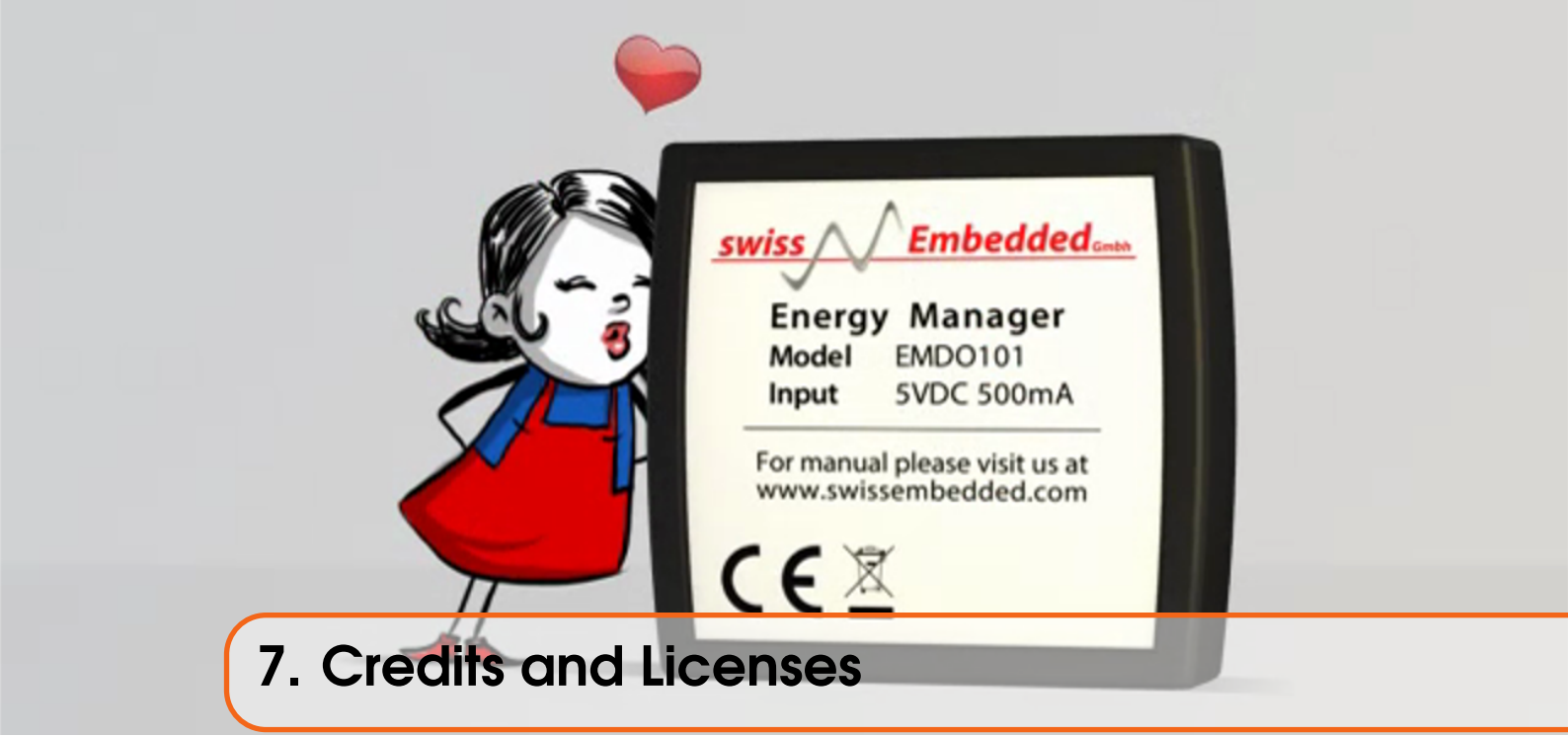
Index 113



6. Specification

Table 6.0.1: *EMDO10x features*

Feature	EMDO101	EMDO102	EMDO103 MBUS	EMDO104 KNX
USB Power Supply	✓	-	-	-
12V/24V Power Supply	-	✓	✓	✓
D0 Interface	✓	-	-	-
RS485	1	2	-	-
MBUS	-	-	✓	-
KNX	-	-	-	✓
S0 Electrical Outputs	2	2	2	2
S0 Electrical Inputs	2	2	2	2
S0 Optical Input	1	-	-	-
Relay Outputs	1	2	2	2
1 Wire	-	1	1	1
Ethernet	1	1	1	1
Power Consumption	< 1Watt			
Dimensions(mm)	67x66x35	92x70x32	92x70x32	92x70x32
Weight(gram)	90	95	95	95



7. Credits and Licenses

7.1 Developers

Founder Daniel Haensse (swissEmbedded GmbH)

Hardware, Firmware, WebUI, Webservice, Basic scripts, Manual (alphabetical order)

- Andrey Ivanov
- Andrey Borovskikh
- Eduardo Jose Ramos
- Eman Abdelmohsen Gaber
- Gabriel Soare
- Gabriel Donari
- Moosa Bonomali
- Mr. Bean *aka Bred Pitt*
- Roman

Kickstarter Campaign

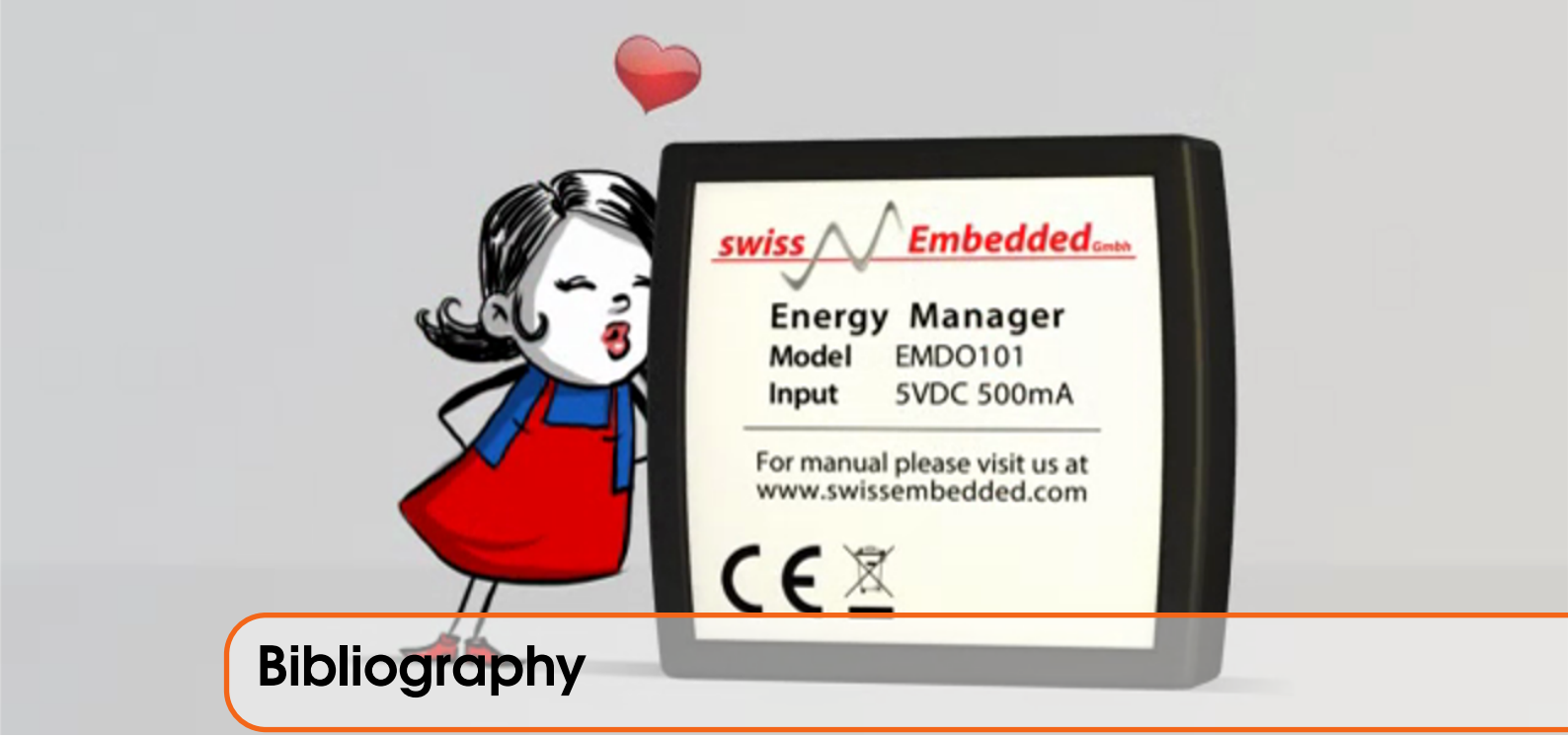
SEO: Sandra Ravioli
EMDO the movie: Petros Sagoridis
Text: Andrew Collingwood
Native speakers: Andrew Collingwood, Tom Ehrhardt
Kickstarters: A humongous thanks for supporting us.

MMBASIC

Author: `http://mmbasic.com` (Geoff Graham)

Special Thanks

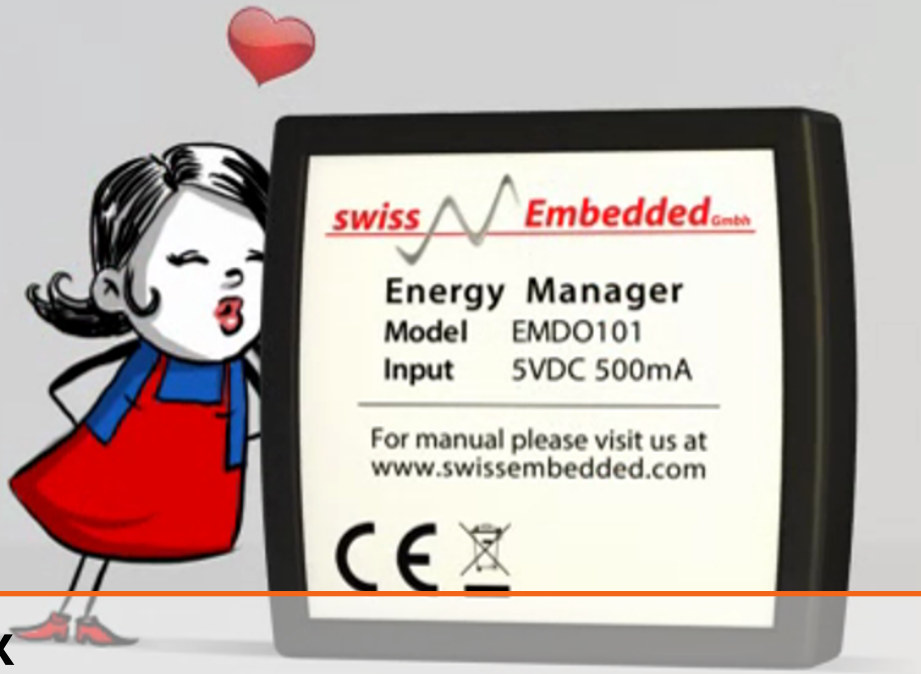
Samoylov Eugene (microrl), Michael Ringgaard, Peter Andersson (spiffs), Hein Tibosch, Richard Barry, Rich Geldreich (gz), Joseph A. Adams (json), and more...



Bibliography

7.2 References

- EMDO github repository with lot of application examples and support libraries `https://github.com/swissembedded/em`.
- Geoff GRAHAM's MMBasic Language Manual `http://geoffg.net/Downloads/Maximite/MMBasic%20Language%20Manual.pdf`
- M-Bus standard `http://www.m-bus.com/`
- Message Queuing Telemetry Transport (MQTT) protocol (OASIS standard) `http://mqtt.org/`
- Modbus `http://www.modbus.org/`
- Sunspec Alliance standard `http://sunspec.org`



Index

(, 70
* (Arithmetic operator), 39
+ (Arithmetic operator), 39
+ (String operator), 39
- (Arithmetic operator), 39
/ (Arithmetic operator), 39
< (Logical operator), 39
< (String operator), 39
« (Logical operator), 39
<= (Logical operator), 39
<= (String operator), 39
<> (Logical operator), 39
<> (String operator), 39
= (Logical operator), 39
= (String operator), 39
=< (Logical operator), 39
=< (String operator), 39
=> (Logical operator), 39
=> (String operator), 39
> (Logical operator), 39
> (String operator), 39
>= (Logical operator), 39
>= (String operator), 39
» (Logical operator), 39
?, 53
(Object API, 68
^(Arithmetic operator), 39
\(Arithmetic operator), 39
~(Logical operator), 39

Abs, 47
Acos, 47
AND (Logical operator), 39
As (Open), 54
Asc, 49
Asin, 47
Atn, 47

Case, 38
Case Else, 38
Chain, 56
CheckTimeout, 67
Chr$, 49
Cint, 47
Clear, 58
Close, 54
Const, 43
Continue, 41
Conv, 49
Copy, 55
Copyright, 58
Cos, 47
CRC$, 49
CrontabAdd, 71
CrontabRemove, 71

D0Read, 63
D0Read$, 63
D0Start, 62
D0Write, 63

Data, 43
Date, 69
Deg, 48
Detach, 41
Dim, 44
Dim Length, 44
Dir, 55
Dispatch, 67
Do, 40
Do While, 40

Else, 37
Elseif, 38
End Function, 42
End Select, 38
End Sub, 43
Endif, 38
EnoceanReceive, 72
EnoceanTransmit, 72
Eof, 55
Erase, 58
Error, 58
Exit Do, 40
Exit For, 40
Exit Function, 42
Exit Sub, 43
Exp, 48

Files, 55
Fix, 48
For, 40
For (Open), 54
Format$, 50
Function, 42

Gosub, 41
Goto, 41

Hex$, 51

If, 37
Inkey, 52
Input, 52
Instr, 50
Int, 47

Kill, 56
KillTimer, 70

Lcase$, 50
Left$, 51
Len, 51
Let, 44
Library Load, 57
Library Unload, 57
Line Input, 52
List, 57
Loc, 55
Local, 41
Lof, 55
Log, 48
Loop, 40
Loop Until, 40
Ltrim$, 51

Memory, 58
Mid$, 51
MM.CMDLINE$, 59
MM.ERRNO, 59
MM.FNAME$, 59
MM.VER, 59
Mod (Arithmetic operator), 39
MQTTConnect, 61
MQTTConnectTLS, 61
MQTTDisconnect, 61
MQTTPublish, 61
MQTTSubscribe, 62
MQTTSubscription, 62
MQTTUnsubscribe, 62

Name, 56
Next, 40
Not (Logical operator), 39

Oct$, 51
On Cron, 41
On Cron [Detach], 71
On Gosub, 41
On Goto, 41
On Timer, 41
On Timer [Detach], 70
Open, 54
Option Base, 45
Option Error Abort, 45
Option Error Continue, 45
Option Prompt, 45
OR (Logical operator), 39

Pause, 57
Peek, 59
Pi, 48
Print, 53

Rad, 48
Read, 44
Rem, 57
Right$, 51
Rnd, 48
Round, 48
RS485Read, 64
RS485Read$, 63
RS485ReadLn$, 64
RS485Write, 64
Rtrim$, 51
Run, 58

S0In, 66
S0Out, 67
Seek, 56
Select Case, 38
SetTimeout, 67
SetTimer, 70
SetVarAccess, 46
Sgn, 48
Sin, 47
SockerRead$, 65
SocketClient, 64
SocketClose, 65
SocketConnected, 64
SocketOption, 65
SocketRead, 64
SocketReadLn$, 65
SocketServer, 64
SocketWrite, 65
Space$, 51
Spc, 51
Split$, 51
Sqr, 48
Str$, 51
StreamSearch$, 65
String$, 52
Sub, 43
Sys.Get, 59
Sys.Set, 59
Syslog, 59
System, 59

Tab, 52
Tan, 47
Temperature, 60
Then, 37
Ticks, 69
Timestamp, 69
Trace Extended, 60
Trace List, 60
Trace Off, 60
Trace On, 60
Trim$, 51
Troff, 60
Tron, 60

Ucase$, 50
Unixtime, 69

Val, 52

Wdt, 60
Weekday, 70

XOR (Logical operator), 39

Yearday, 70