KUNBUS
industrial communication

User-Manual
KUNBUS-Scripter

PR100087R03_US • 6/12/2019

Table of Contents

1 Overview

1.1 Overview

With the KUNBUS Scripter you can exchange data in any format with any devices via a serial interface.

The KUNBUS-Scripter is divided into three components:

KUNBUS Script Wizard

The Script Wizard allows you to create a script without requiring any programming skills. We provide you with predefined templates for query-response telegrams. Furthermore, you have the option to create your own templates for commands.

KUNBUS Script Editor

With the Script Editor you can edit your script like in text software. The Editor supports you by means of:

- auto-complete
- syntax highlighting

Both components have a trace. This allows you to trace which values the variables accept in the script and where an error occurs.

Once you have created a script in the Script Wizard or Script Editor, you can transfer the script to the module.

Interpreter on the Module

The interpreter on the module processes the transferred scripts.

1.2 Functionality

Communication Sequence

Once you have created a script, it is saved as a PBS file. The abbreviation PBS stands for **P**rotocol **B**uilder **S**cript.

During compilation, a binary file is generated from the script. This binary file is transferred to the module via the Common Debug Interface (CDI) and stored in the flash. The binary file is processed in the module.

The trace is sent back to the PC via the CDI interface. As a result, you can view the current value of the variables in the Script Wizard and Script Editor.

Script structure and execution

The script consists of several sections. Each of these sections has a specific task.

The script starts in the CONFIG section via the INIT section after RUN, or several RUN sections are continued. The script is executed cyclically.

- SECTION_CONFIG
  Commands in this section are for the unique configuration of the module. Here, you can define the following settings on the serial interface:
  Bitrate
  byte sequence for the protocol
  declaration of the variables
- SECTION_INIT
  Commands in this section are for the initialization of the module. Here, you can set variables and Modbus registers to start values.
- SECTION_RUN
  The script can contain several pairs from SECTION_RUN and SECTION_ERROR that are executed sequentially. After the last SECTION_RUN, the protocol jumps again to the first SECTION_RUN.
- SECTION_ERROR
  If an error occurs in SECTION_RUN, the following SECTION_ERROR is executed. Afterwards, the script continues running in the following SECTION_RUN.
  If an error reoccurs in SECTION_ERROR, the SECTION_FATAL will be executed next.
- SECTION_FATAL
  Fatal errors occur in this section. These errors could not be resolved in the SECTION_ERROR.
  The script with SECTION_INIT is restarted after the code of SECTION_FATAL.
  You can stop the execution of the script manually using the stop() command. The script then restarts after the next module reset.

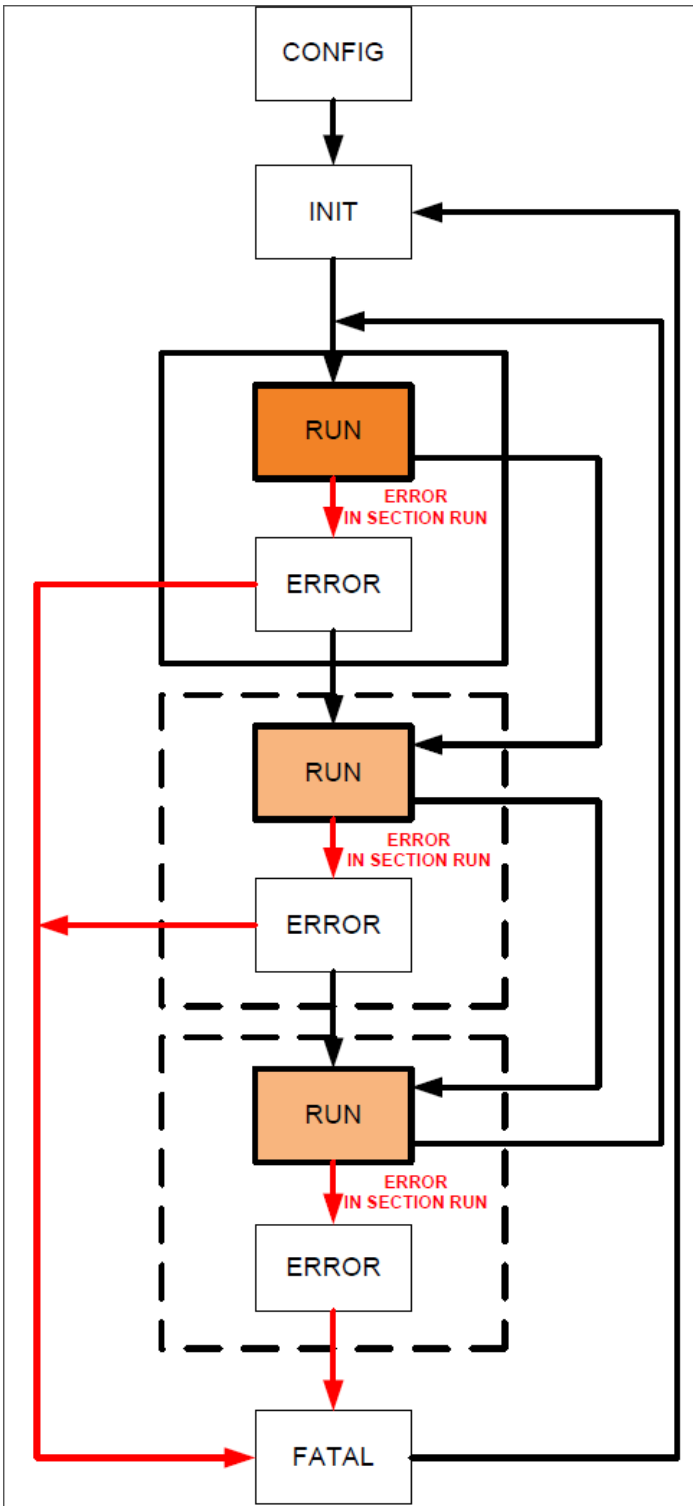


Illustration 1: Script structure

The current implementation is based on a two-stage error or exception model. This model controls the individual sections by implicit jumps. If the script is currently located in the section RUN, it jumps to the following section ERROR if an error occurs (first error stage). Once the commands have been processed in this section, the

script jumps to the start of the next section RUN. If the current section RUN is the last sequence, the script then jumps to the first section RUN.

The section INIT does not normally contain any critical commands that interact with the target application. If an error should occur in this section, however, the script jumps to the section FATAL (second error stage). An error within the section ERROR also causes a jump to FATAL. After the section FATAL, the script is restarted, i.e. it is executed from the section INIT. To prevent a restart, e.g. because continuing a script in the event of an error would not make any sense, a stop() command can be inserted.

1.3 Start Menu

Please visit our Homepage (www.kunbus.com). Here, you will find the File "Setup_KUNBUS_Scripter.exe".

- Download the file "Setup_KUNBUS_Scripter.exe".
- Double-click to start the file

⇨ The start menu opens

In the start menu of the KUNBUS-Scripter you can select how you would like to create the protocol.

– The "Script Wizard" function requires very few programming skills. You can use a script template or compile a script yourself from existing modules.

– You write the script in a text editor using the "Script Editor" function. Here, we provide several features to make the work easier for you.

- Select the required function and confirm it by pressing "OK"

Illustration 2: Start Menu

2 KUNBUS Script Editor

2.1 Overview

Script Editor Overview

The desktop of the Script Editor is divided into three windows.

- In the Script Editor (5) you write your script
- The display window for messages (9) contains information on the status of your script
- On the Trace-Monitor (7) you can trace the sequence of your script.

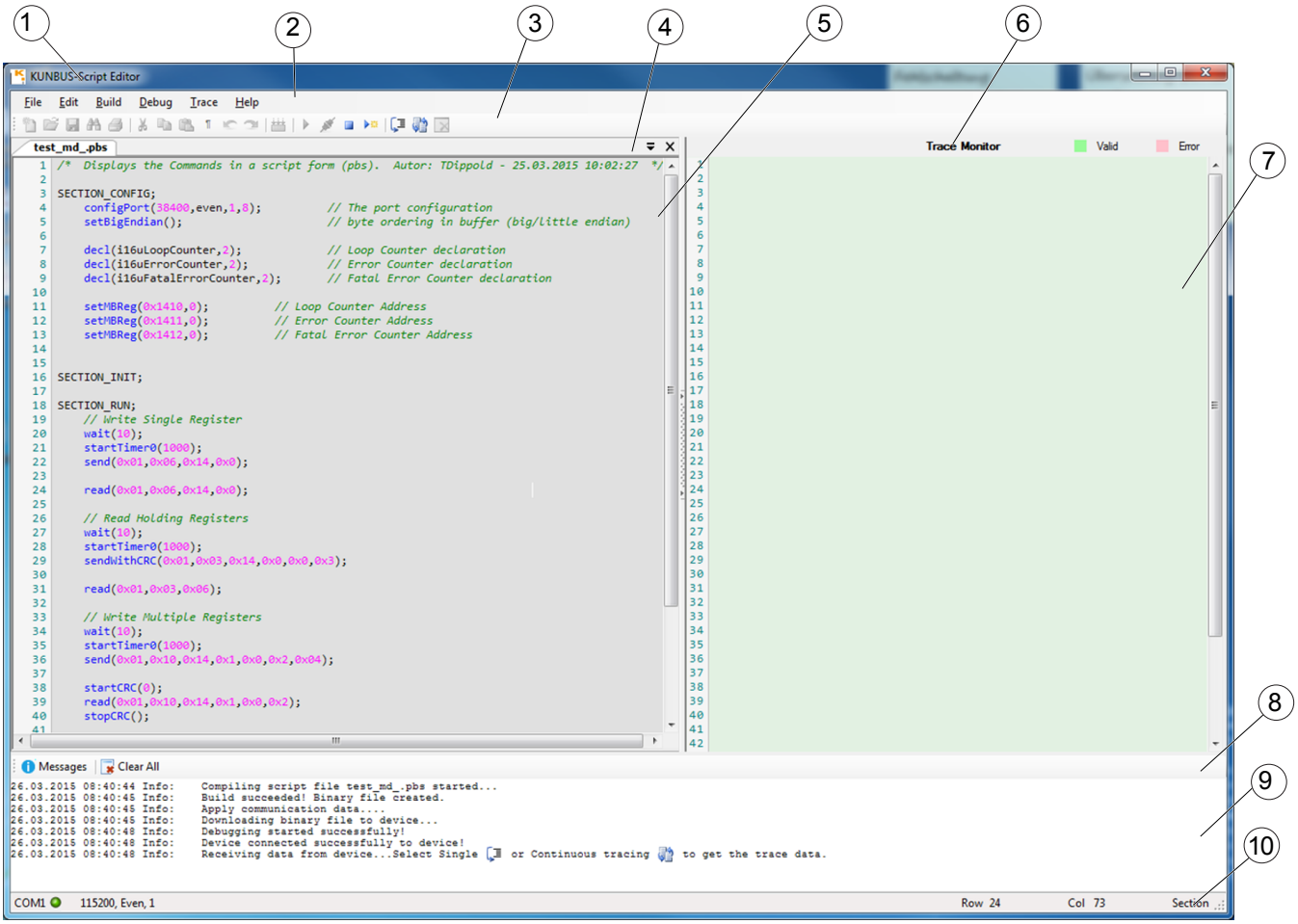


Illustration 3: Protocol builder

1	Window bar	2	Menu bar
3	Toolbar	4	Window bar editor
5	Script editor	6	Trace status bar
7	Trace monitor	8	Toolbar of the message window
9	Display window for messages	10	Status bar

2.2 Menu bar

This section describes the individual functions in the menu bar.

1 File

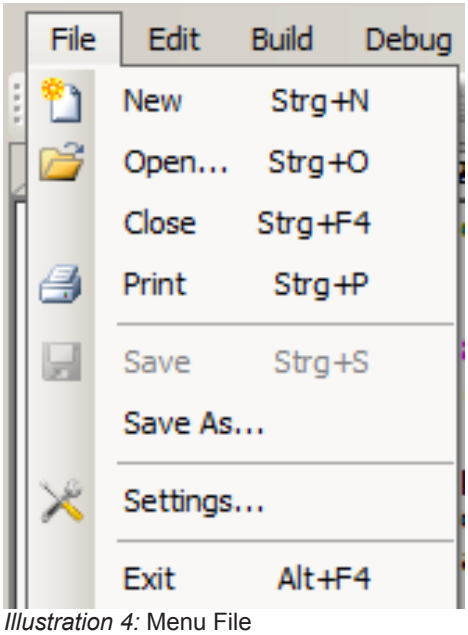


Illustration 4: Menu File

1.1 New:

Open an empty script.

1.2 Open:

Open an existing script.

1.3 Close:

Close the opened script.

1.5 Print:

Print script.

1.6 Save:

Save as PBS file.

1.7 Save As:

Save as PBS file. You can define the file name and storage location.

1.8 Settings

1.8.1 Connection Settings

Define communication settings between PC and module.

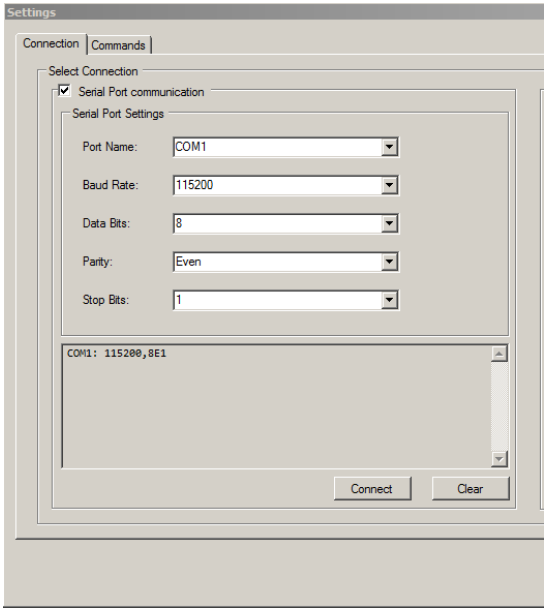


Illustration 5: Connection settings

1.8.2 Connection Commands

Overview of the permitted syntax.
Info!: This menu is purely informative. You cannot assign any of your own values here**.**

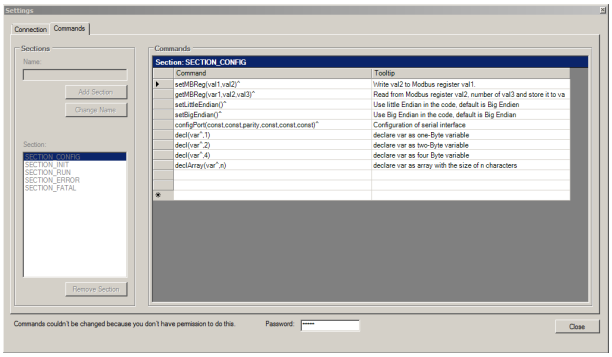
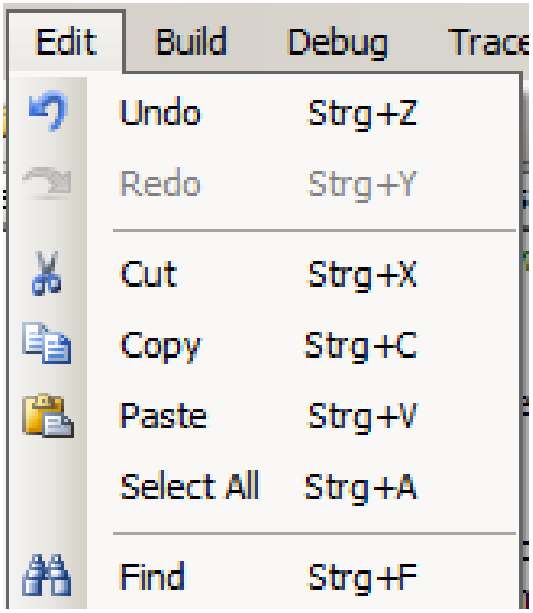


Illustration 6: Command settings

1.9 Exit:

Exit program.

2. Edit



2.1 Undo:

Undo last entry.

2.2 Redo:

Redo last entry.

2.3 Cut:

Cut a selected area.

2.4 Copy:

Copy a selected area.

2.5 Paste:

Insert a previously copied or cut area at the selected position.

2.6 Select All:

Select all the editing area.

2.7 Find:

Find a term in the editor.

3. Build

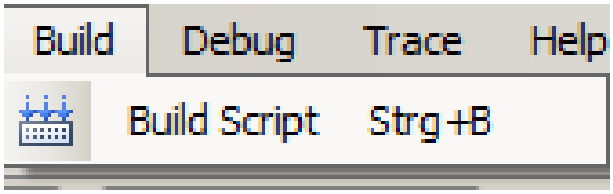


Illustration 7: Build Menu

3.1 Build Script:

Check script for errors and generate a binary file.

4. Debug

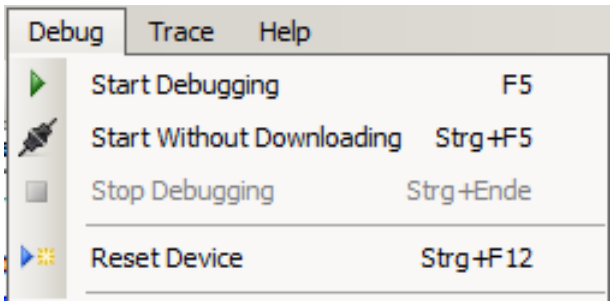


Illustration 8: Debug menu

4.1 Start Debugging:

Check script for errors and generate a binary file. If the file is error-free, it is loaded into the module and executed. A debug connection to the module exists from now on. The trace can be displayed.

4.2 Start Without Downloading:

Check whether the script in the editor and the script on the module are identical. If this is the case, a trace connection is established. The trace can be displayed from now on. The script will not be interrupted or restarted thereby.

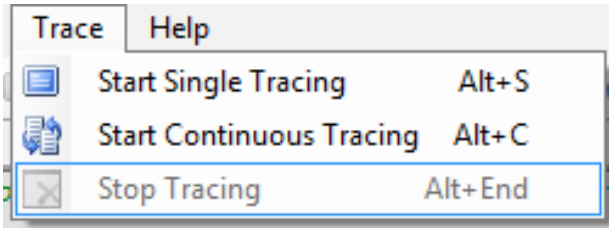
4.3 Stop Debugging:

Interrupt the communication with the module.

4.4 Reset Device:

Restart module.

5. Trace



5.1 Start Single Tracing:

Display sequence of the script in the trace monitor. In Single Tracing mode a single script sequence is polled. This function is useful if you want to exactly trace a specific position in the script sequence.

5.2 Start Continuous Tracing:

Display sequence of the script in the trace monitor. In Continuous Tracing mode the trace is polled cyclically.

5.3 Stop Tracing:

Stops the trace connection.

6. Help

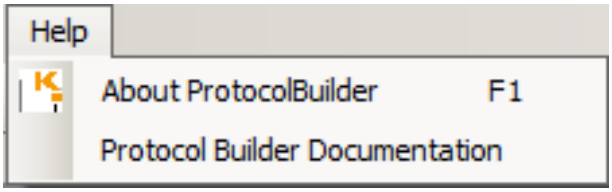


Illustration 9: Help menu

6.1 About KUNBUS Scripter:

Displays system information about KUNBUS-Scripter.

6.2 KUNBUS Scripter Documentation:

Displays help topics about the Scripter.

Among other things, you will also find the syntax here.

2.3 Toolbar

You have quick access to the following operations via the toolbar:

New

Open an empty script.

Open

Open an existing script.

Save

Save as PBS file.

Find

Find a term in the editor.

Print

Print script.

Cut

Cut a selected area.

Copy

Copy a selected area.

Paste

Insert a previously copied or cut area at the selected position.

Control character

Displays or hides control character.

Undo entry

Undo last entry.

Redo entry

Redo last entry.

Compile script

Check script for errors and generate a binary file.

Start Debug

Check script for errors and generate a binary file. If the file is error-free, it is loaded into the module and executed. A debug connection to the module exists from now on. The trace can be displayed.

Start Without Downloading

Check whether the script in the editor and the script on the module are identical. If this is the case, a trace connection is established. The trace can be displayed from now on. The script will not be interrupted or restarted thereby.

Stop Debug

Stops the debug connection.

Reset Module

Restart module.

Start Single Tracing

Display sequence of the script in the trace monitor. In Single Tracing mode a single script sequence is polled. This function is useful if you want to exactly trace a specific position in the script sequence.

Start Continuous Tracing

Display sequence of the script in the trace monitor. In Continuous Tracing mode the trace is polled cyclically.

Stop Tracing

Stops the trace connection.

2.4 Script Editor

In the text editor you can write your script.

Please pay attention to the predefined Functionality [▶ 4] and Overview of the Script Commands [▶ 41] used

Navigation and orientation aids

Navigation

You have the option of opening several scripts in the Editor. A new tab appears with each script in the navigation bar at the upper window edge of the editor. You can switch between these views by:

- clicking on the required tab or
- by clicking on the down arrow on the right-hand side and then clicking on the desired script.

Syntax highlighting

Functions, comments, values and variables are displayed in colour for easier orientation

- Functions: blue
- Values: pink
- Variables: black
- Comments: green

```
test_md_.pbs
1  /*  Displays the Commands in a script form (pbs).  Autor: TDippold - 25.03.2015 10:02:27  */
2
3  SECTION_CONFIG;
4      configPort(38400,even,1,8);          // The port configuration
5      setBigEndian();                      // byte ordering in buffer (big/little endian)
6
7      decl(i16uLoopCounter,2);             // Loop Counter declaration
8      decl(i16uErrorCounter,2);            // Error Counter declaration
9      decl(i16uFatalErrorCounter,2);       // Fatal Error Counter declaration
10
11     setMBReg(0x1410,0);          // Loop Counter Address
12     setMBReg(0x1411,0);          // Error Counter Address
13     setMBReg(0x1412,0);          // Fatal Error Counter Address
14
15
16 SECTION_INIT;
17
18 SECTION_RUN;
19     // Write Single Register
20     wait(10);
21     startTimer0(1000);
22     send(0x01,0x06,0x14,0x0);
23
24     read(0x01,0x06,0x14,0x0);
25
26     // Read Holding Registers
27     wait(10);
28     startTimer0(1000);
29     sendWithCRC(0x01,0x03,0x14,0x0,0x0,0x3);
30
31     read(0x01,0x03,0x06);
32
33     // Write Multiple Registers
34     wait(10);
35     startTimer0(1000);
36     send(0x01,0x10,0x14,0x1,0x0,0x2,0x04);
37
38     startCRC(0);
39     read(0x01,0x10,0x14,0x1,0x0,0x2);
40     stopCRC();
41
42     add(i16uLoopCounter,1);
43     setMBReg(0x1410,i16uLoopCounter);
44
45 SECTION_ERROR;
46     add(i16uErrorCounter,1);
47     setMBReg(0x1411,i16uErrorCounter);
48
49 SECTION_FATAL;
50     add(i16uFatalErrorCounter,1);
51     setMBReg(0x1412,i16uFatalErrorCounter);
```

Illustration 10: Syntax highlighting

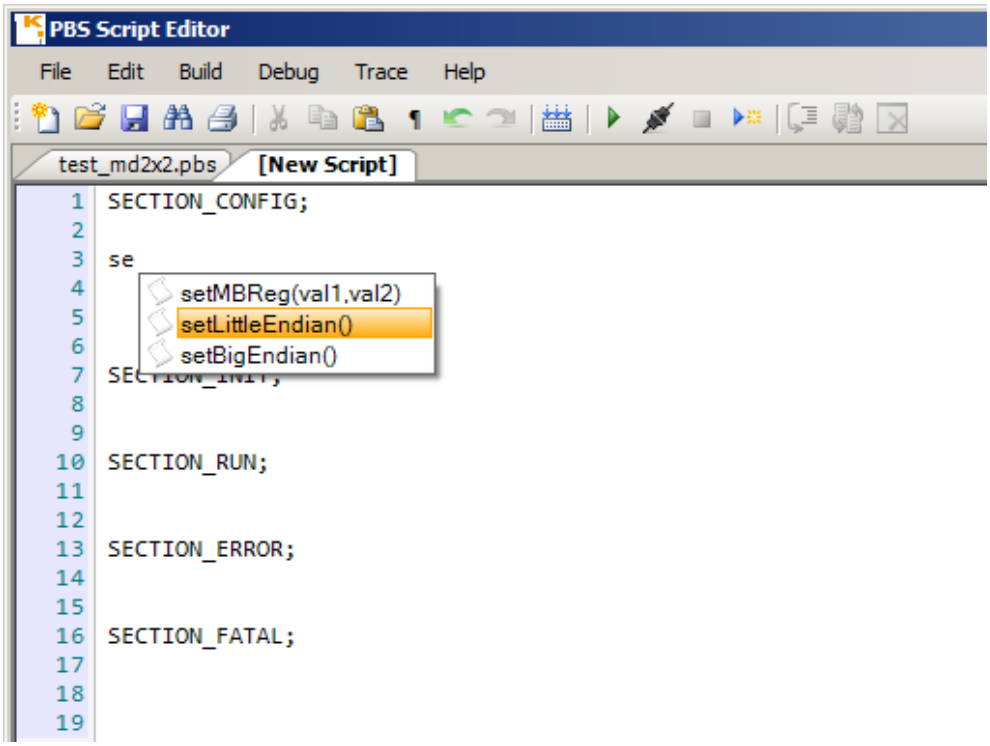
Labor-saving functions

Automatic text completion

The KUNBUS-Scripter supports you by completing the text of the permissible syntax automatically.

- Write the first letters in the Editor of the desired syntax in the Editor.
  - ⇨ A selection window will already open after the first letter.
- Click on the desired syntax.

⇨ The selected syntax is accepted by the Editor.



Indentation of text blocks

- Select the required text block
- Select "AutoIndent selected text" in the context menu

⇨ The selected text will now be displayed indented.

2.5 Starting script

Prerequisite: You have created a script in the Editor or opened an existing script.

- In the toolbar, select the function "Connect"
  - ⇨ A connection to the module is established
- In the toolbar, select the function "Compile File"
  - ⇨ With this function you compile the script.
- In the toolbar, select the function "Start Debug"
  - ⇨ With this function you start the script.

⇨ Terminate the connection by selecting

2.6 Trace monitor

On the Trace-Monitor you have the possibility to trace the sequence of your script. The Trace-Monitor window will open on the right-hand side next to the Editor window. The window opens automatically as soon as you select a trace function.

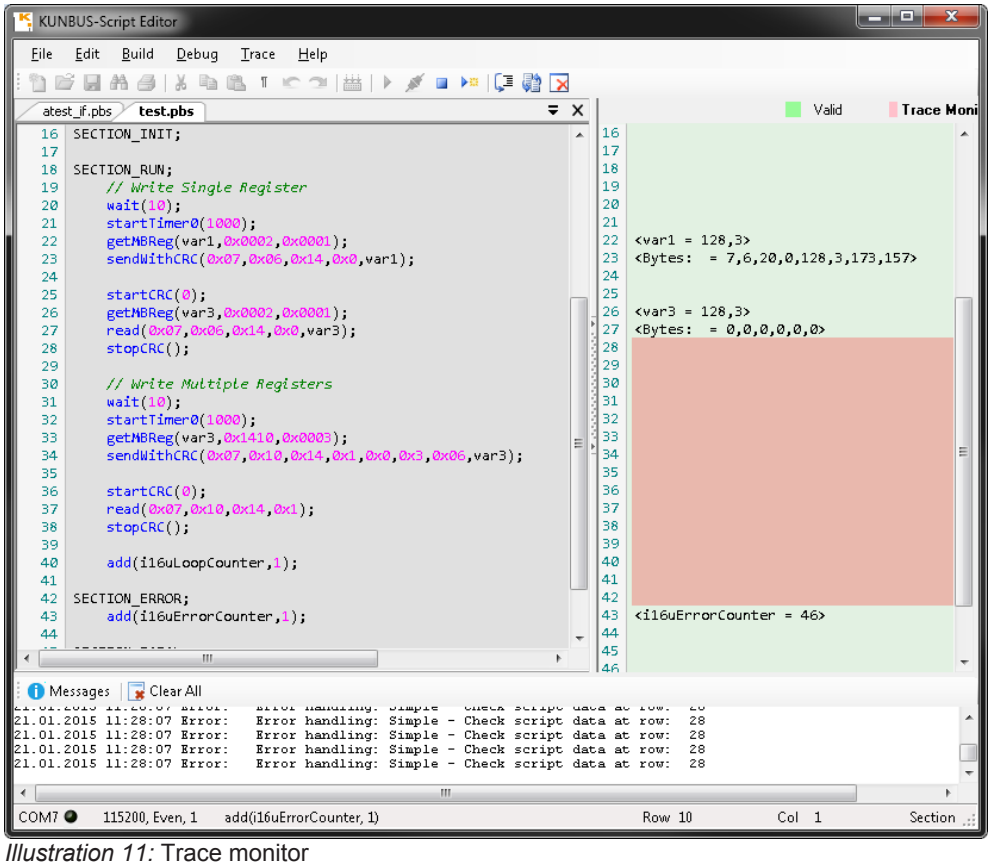


Illustration 11: Trace monitor

Representation

– In the bar on the left side of the window you can trace in which line of the script a message appears.

– Lines of the script that have been executed are highlighted in green.
– Lines of the script that have not been executed are highlighted in red.

Tracing functions

In the navigation bar in the "Trace" menu item, you will find all tracing functions:

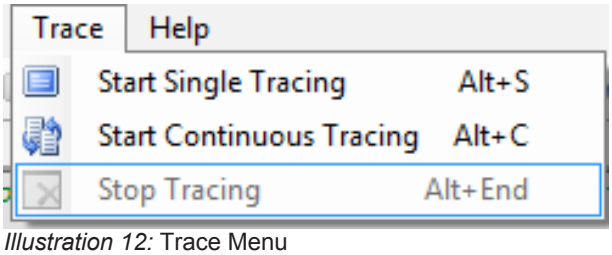
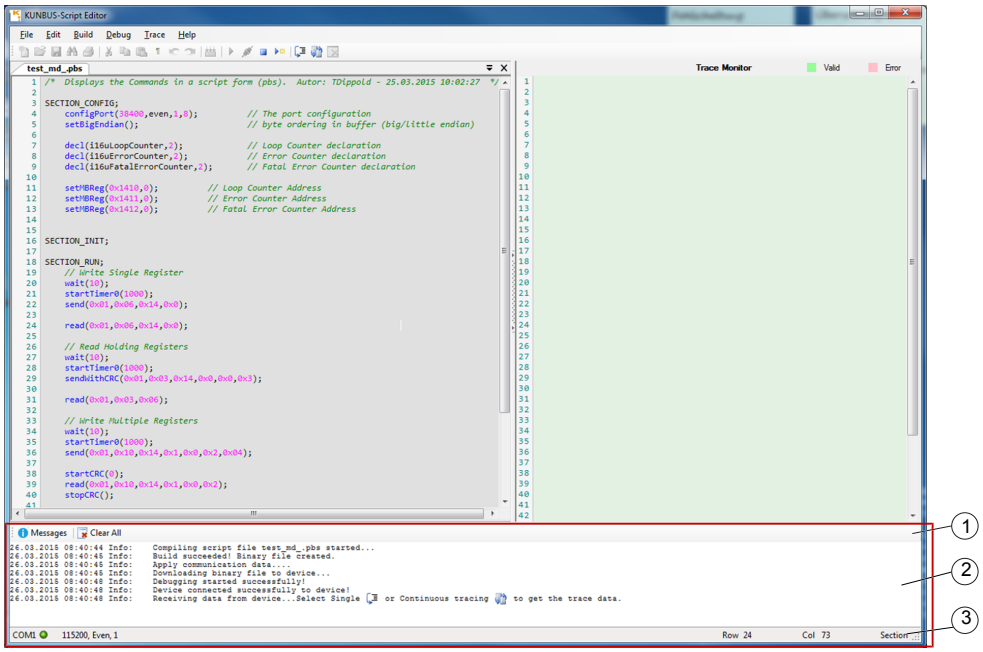


Illustration 12: Trace Menu

Start Single Tracing	With this function you have the possibility to trace the sequence of your script step-by-step.
Start Continuous Tracing	With this function you have the possibility to trace the sequence of your script. In this mode, the script is executed cyclically.
Stop Tracing	This function terminates the tracing.

2.7 Message Window

The message window can be found below the editor window.



1	Toolbar
2	Display window for messages
3	Status bar

Display window for messages

In the message box you will find the following information:

- Status of the script
- Error notifications
- Further instructions

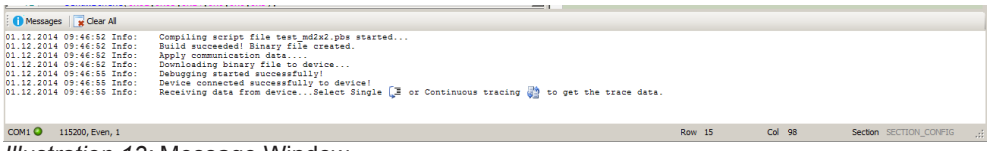


Illustration 13: Message Window

Tip!: By clicking in the message line the cursor jumps to the corresponding line in the script.

Toolbar

In the window bar, you can clear all messages. To do this, click on "Clear All".

Status bar

In the footer bar you will find information about the communication settings for the module.

The colored status indicator indicates whether a connection to the module exists:

Red	No connection to the module
Green	Connection to the module

In the example shown here, you can see in the footer bar that a connection to the module exists. The communication settings are: 115200 bit/s, Even Parity, 1 Stop Bit.

3 KUNBUS Script Wizard

3.1 Overview

With the Script Wizard you can compile a script. You do not need any programming skills to do this.

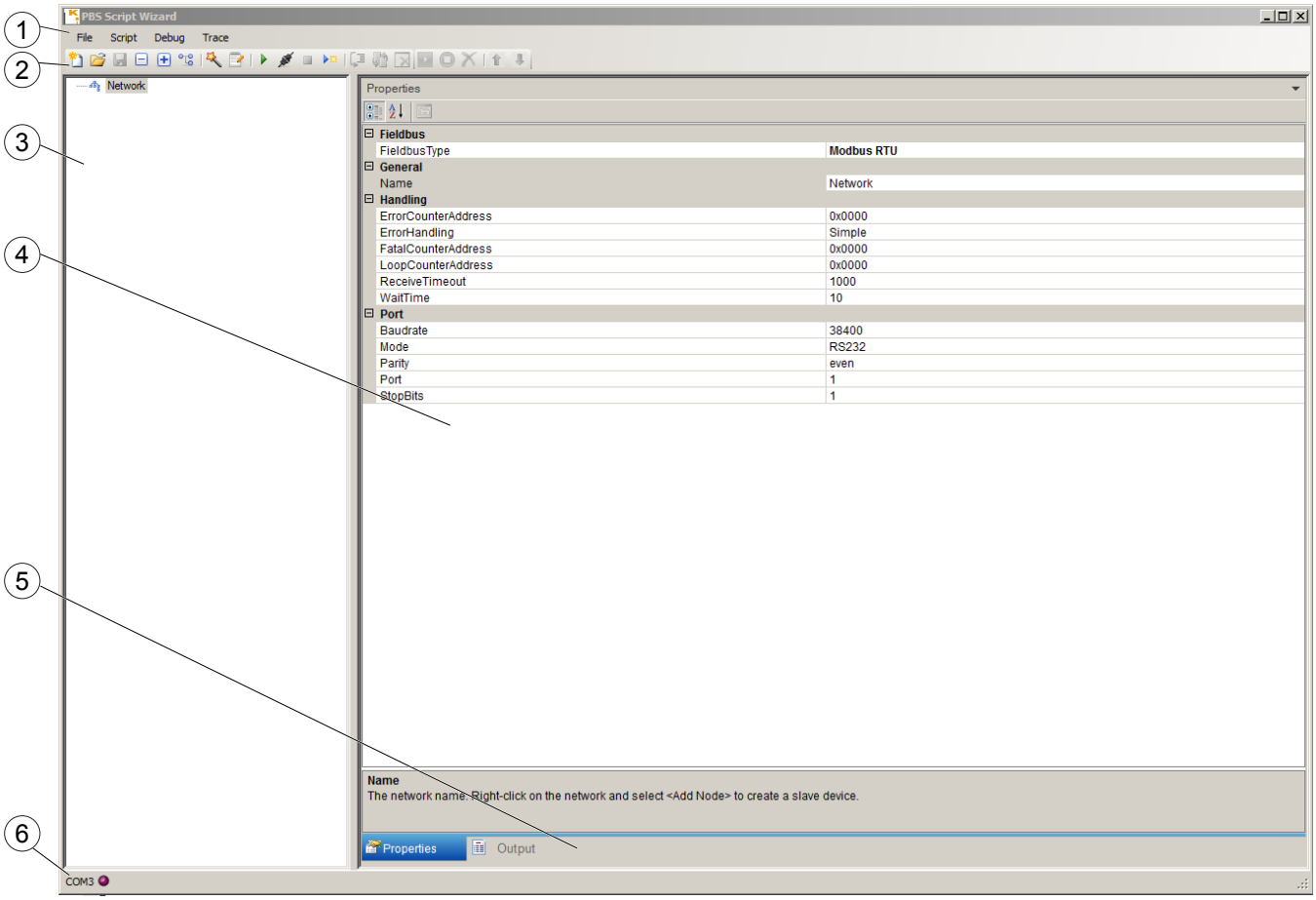


Illustration 14: Start view

1	Menu bar
2	Toolbar
3	Structure tree (Tree view)
4	Work area
5	Register bar
6	Status bar

3.2 Menu bar

This section describes the individual functions in the menu bar.

1 Menu File

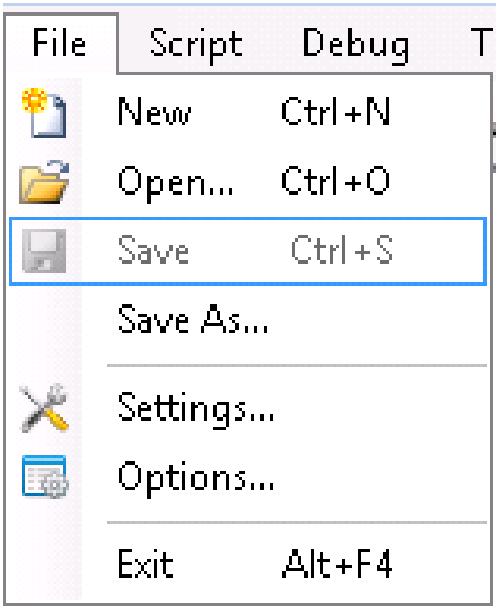


Illustration 15: Menu File

1.1 New:

Open an empty script.

1.2 Open:

Open an existing script.

1.3 Save:

Save Wizard project as XML file.

1.4 Save As:

Save Wizard project as XML file and the script as PBS file. You can also determine the storage location and filename.

1.5 Settings:

1.5.1 Connection Settings

Define communication settings between PC and module.

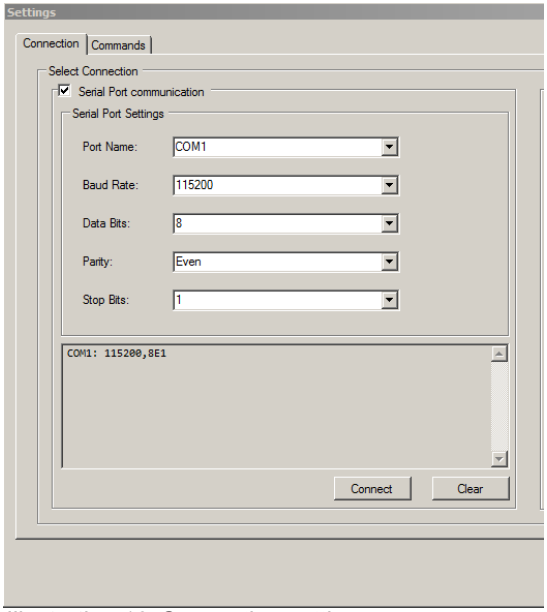


Illustration 16: Connection settings

1.5.2 Command Settings

Overview of the permitted syntax.
Info!: This menu is purely informative. You cannot assign any of your own values here**.**

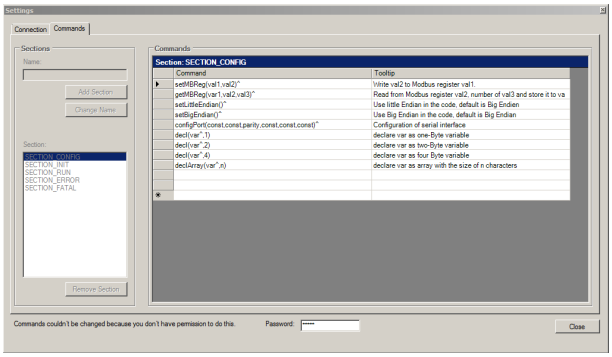
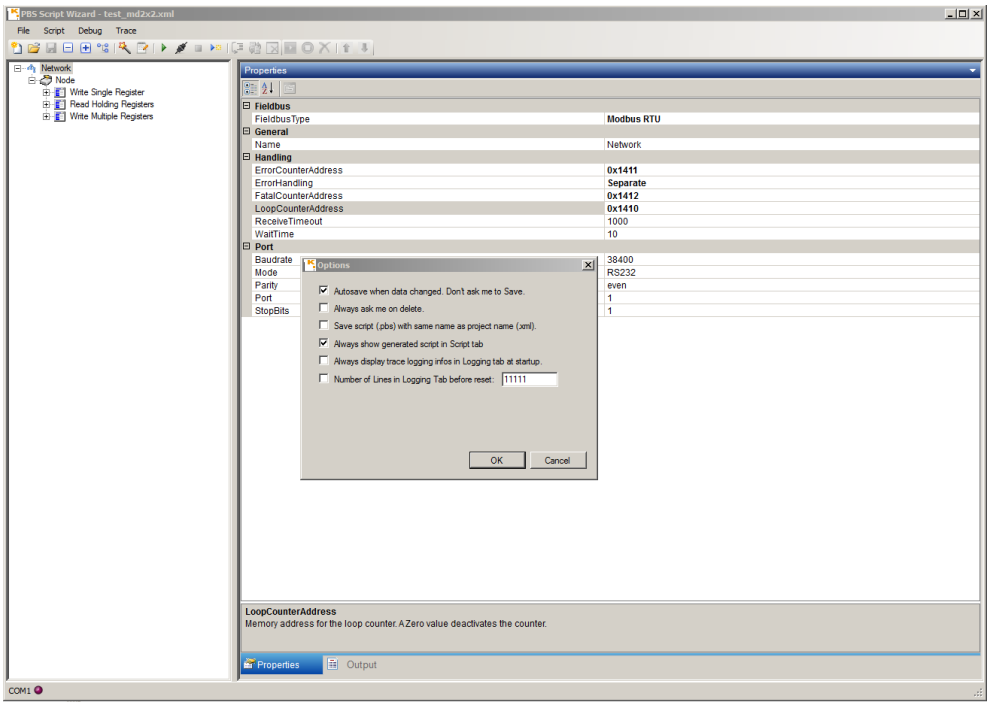


Illustration 17: Command settings

1.6 Options:

Define presettings:

- Automatic saving after changing the data
- Confirmation prompt before deleting
- Automatic name allocation when saving
- Once the script has been generated, open it in a separate tab.
- Display trace logging when starting
- Maximum number of lines after which logging should be deleted.



1.7 Exit:

Exit program.

2. Menu Script

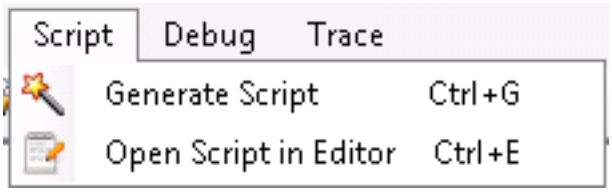


Illustration 18: Menu Script

2.1 Generate Script

Generate Script.

2.2 Open Script in Editor

Open the generated script in the Script Editor.

3 Debug menu

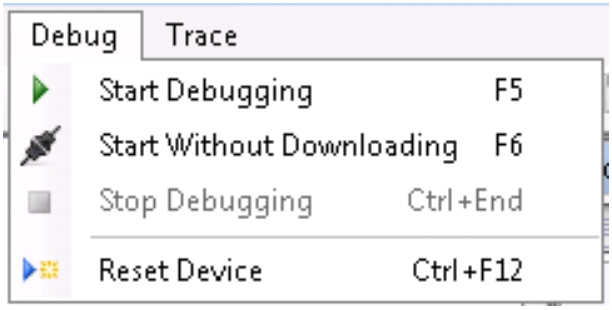


Illustration 19: Debug menu

3.1 Start Debugging

This function executes the following actions:

– Generate script

- Compile script
- Transfer script to the module
- Starting script
- Open trace connection

3.2 Start Without Downloading

Check whether the script in the editor and the script on the module are identical. If this is the case, a trace connection is established. The trace can be displayed from now on. The script will not be interrupted or restarted thereby.

3.3 Stop Debugging

Stops the trace connection.

3.4 Reset Module

Restart module.

4. Trace Menu

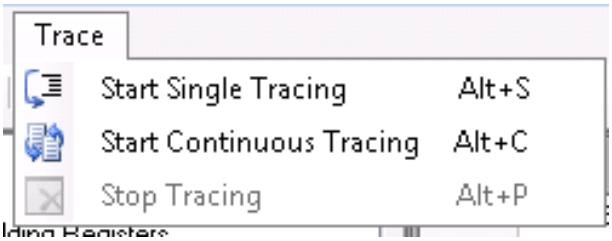


Illustration 20: Trace Menu

4.1 Start Single Trace

Load a trace from the module. The status indicator in the structure tree indicates which parts of the scripts were executed successfully. If the status message is green, the execution was successful. A red status message indicates an error.

4.2 Start Continuous Trace

Poll trace cyclically. The result is displayed in the structure tree.

4.3 Stop Trace

Stops the trace connection.

3.3 Toolbar

You have quick access to the following operations via the toolbar:

New

Open an empty script.

Open

Open an existing script.

Save

Save as PBS file.

Collapse

Collapse a selected command in the structure tree.

Expand

A selected command in the structure tree is expanded.

Collapse/expand All

Collapse/Expand all commands in the structure tree.

Generate script

Generate a script.

Switch to the Editor

Open script in the Script Editor.

Start Debug

This function executes the following actions:

- Generate script
- Compile script
- Transfer script to the module
- Starting script
- Open trace connection

Start Without Downloading 

Check whether the script in the editor and the script on the module are identical. If this is the case, a trace connection is established. The trace can be displayed from now on. The script will not be interrupted or restarted thereby.

Stop Debug 

Stops the trace connection

Reset Module 

Restart module.

Start Single Tracing 

Load a trace from the module. The status indicator in the structure tree indicates which parts of the scripts were executed successfully. If the status message is green, the execution was successful. A red status message indicates an error.

Start Continuous Tracing 

Load trace from the module cyclically. The result is displayed in the structure tree.

Stop Tracing 

Stops the trace connection.

Start Logging 

Start logging. The logging in the trace is displayed continuously in the logging window.

Stop Logging 

Stop logging.

Clear Logging 

Clear logging.

Jump to the top 

Jump to the first line of the logging.

Jump to the bottom 

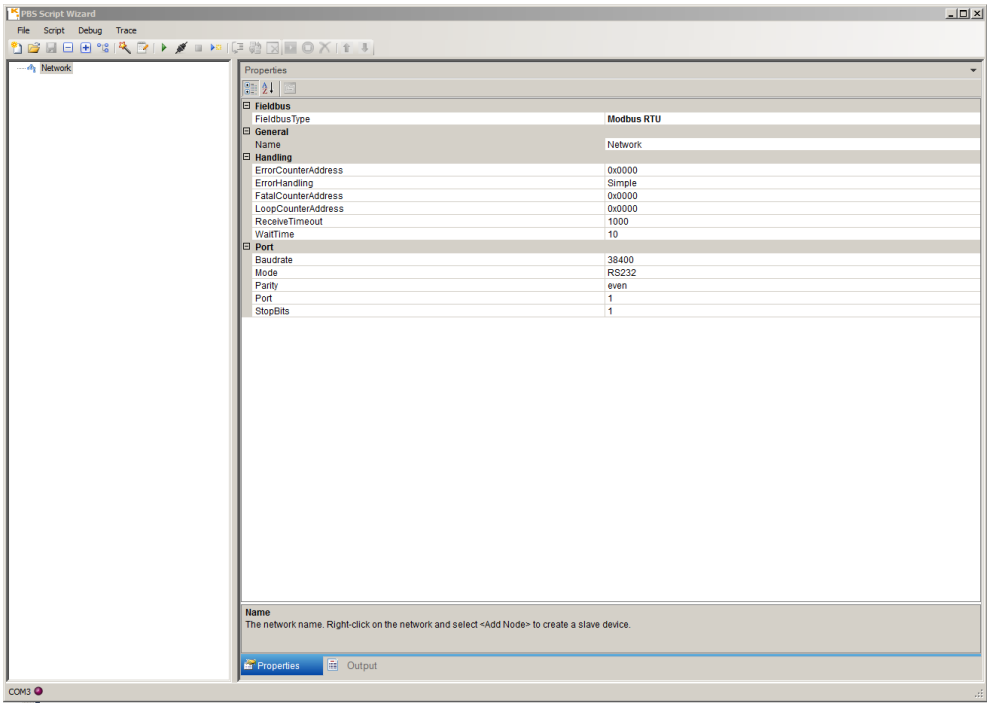
Jump to the last line of the logging.

3.4 Creating Script with the KUNBUS-Wizard

Open the KUNBUS-Wizard

Properties

In the work window you can see the Properties tab Here, you have to define the settings for the communication with the module:



Field	Setting	Value Range
FieldbusType	Enter the fieldbus that you are using.	
Name	Assign a name for the network	
ErrorCounterAddress	Enter the destination register for the error counter here.	SDI Input data range 0x1401-0x1480 To deactivate the function, enter 0x000 in this field
ErrorHandling	Enter the mode of the error indicator	Each command has its own error handling
FatalCounterAddress	Enter the destination register for the fatal error counter here.	SDI Input data range 0x1401-0x1480 To deactivate the function, enter 0x000 in this field
LoopCounterAddress	Enter the destination register for the script loop counter here.	SDI Input data range 0x1401-0x1480 To deactivate the function, enter 0x000 in this field

Field	Setting	Value Range
ReceiveTimeout	Enter the wait time until the response of the module here	Wait time in ms
Wait Time	Enter the wait time between the script sequences.	Wait time in ms
Baudrate	Enter the baudrate	2400 4800 9600 19200 38400 57600 115200
Parity	Enter the parity.	Even, odd, none
StopBits	Enter the number of Stop Bits.	0-2

Add a device

- Right-click on the "Network" line in the tree view
- Select "Add node".

⇨ You have added a new device.

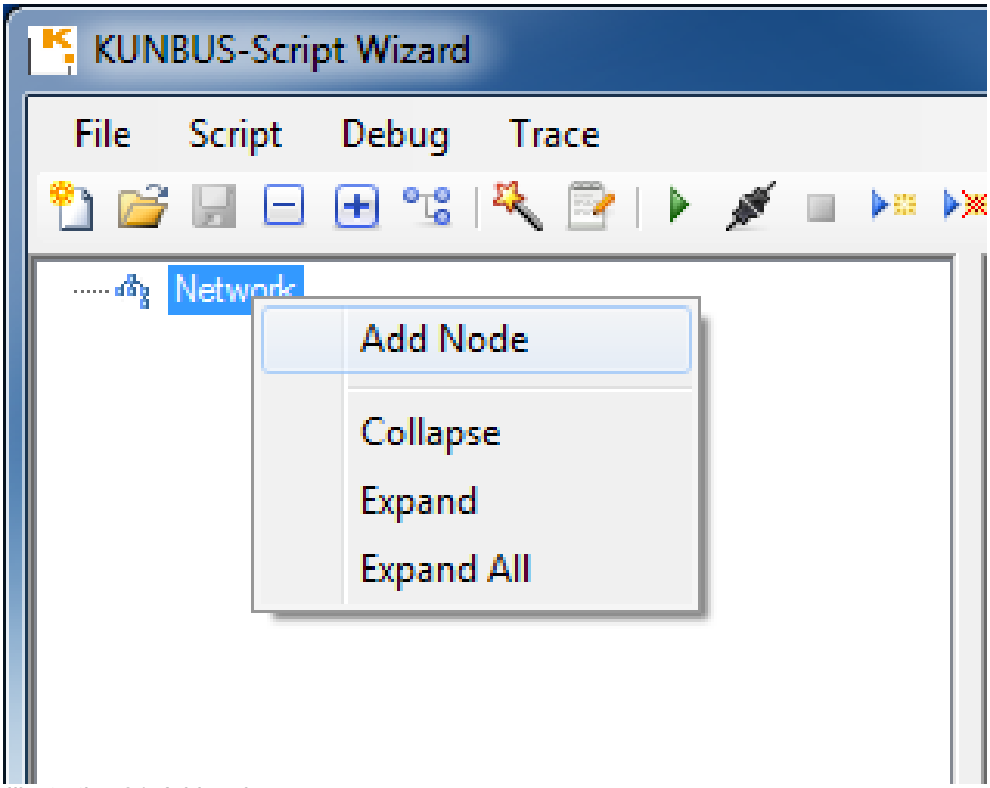


Illustration 21: Add node

In the "Properties" window you can define an address and a name for the device. Commands can be assigned to the device with this address.

Command

A command consists of one request and one response respectively. The request is sent from the script to the node. A response is then awaited. Since a command is mostly used multiple times with various parameters, the Command Editor contains predefined commands that you can select.

Add a new command

✓ You have already added a new device (node) in the structure tree

◦ In the tree view, right-click on the device that you want to assign a command to.

◦ Select "Add Command"

⇨ The Command Editor opens. In the next section you will learn how to create a command.

Command Editor

In the Command Editor you can select commands for your script.

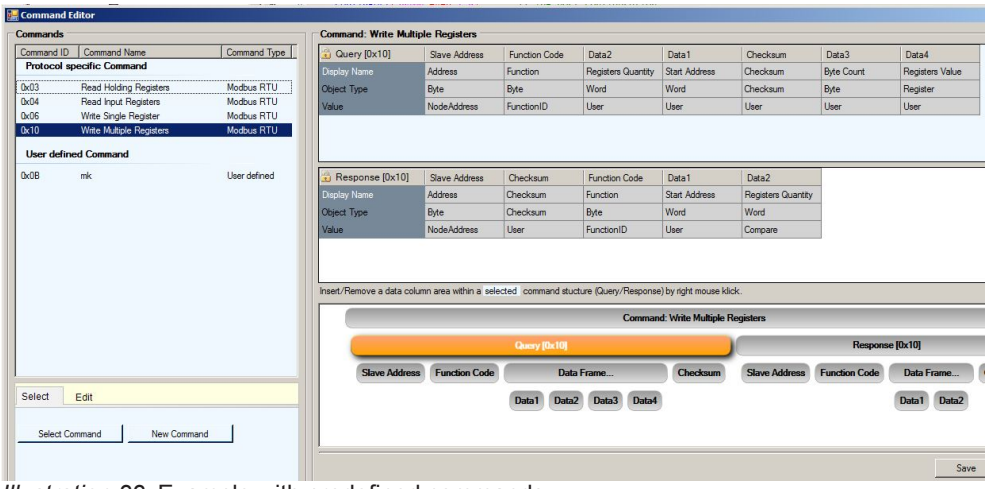


Illustration 22: Example with predefined commands

In the "Commands" window you will find the "Protocol specific Command" section. The commands listed here have already been predefined. They can vary depending on the fieldbus used. In this example, we will use Modbus RTU as protocol.

Info!: The fieldbus used can be saved in the Properties, in the "FieldbusType" field.

Predefined Commands

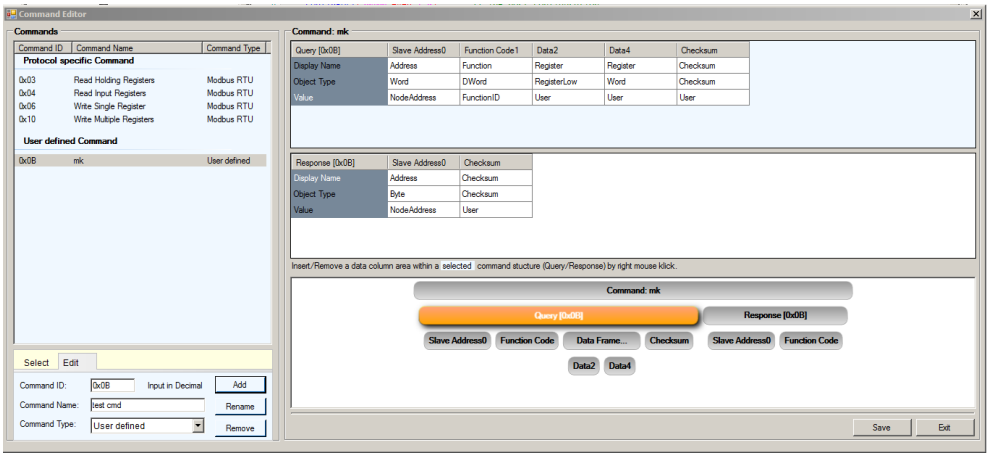
◦ Select the required command from the "Protocol specific Command" section

⇨ The predefined command is displayed in the "Command" window. It is divided into the query and response area. The query area represents a query telegram that is sent to the module. The response area is the response telegram that transfers the values from the module.

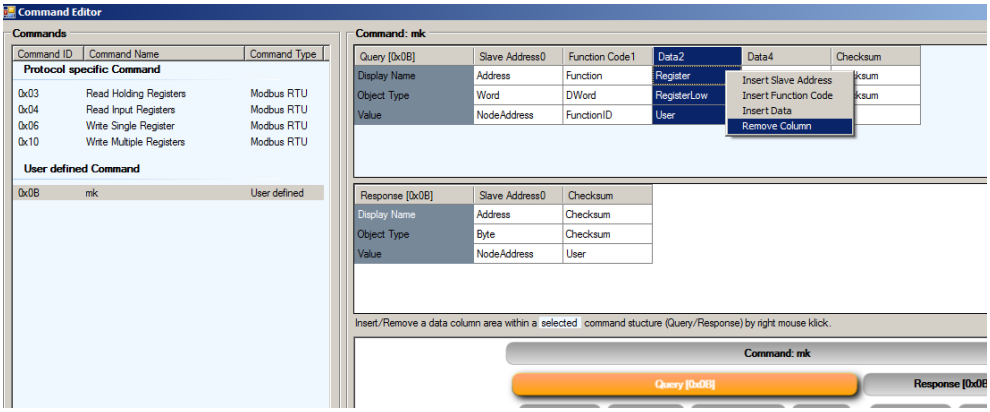
The commands perform the following functions:

Function	Description
Read Holding Registers	Reads measured values
Read Input Registers	Reads input registers
Write Single Registers	Writes on a single register
Write Multiple Registers	Writes on multiple registers
User defined	User defined

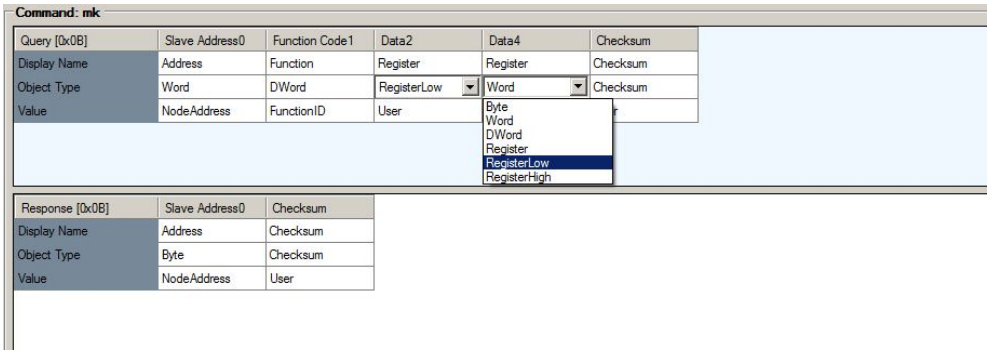
Create user defined commands



- Select the "Edit" tab in the "Commands" window
- Enter a new command ID.
- Enter a command name.
- Select "User defined" to define any command you wish. If you select "Modbus RTU" here, the command must begin with slave address and function code and must end with a checksum. The fields in between can be defined as you wish.
- Click on "Add" to define your script in the "Command" window.
  Tip!: In the Edit tab you can delete the created command once again or rename it.
- Add the required elements in the "Command" field. Right-click in the "Query" or "Response" area.
  ⇨ A selection box with the available elements will open.
- Select the desired elements.



- Define the values of your elements. We also support you here by means of predefined options.



- Save your command with "Save".
- In the left "Commands" column, select a command. The "Select Command" button allows you to add the command to your script in the main window.
- Click on "Exit" to exit the Command Editor.

Function and valid Values of the Commands

Command	Value	Function
Display Name	Freely definable	Name that the field is displayed with in the main window.
Object Type	Here, you can select the field type.	
	Byte	The field includes one byte with a fixed value.
	Word	The field includes 2 bytes with a fixed value.
	DWord	The field includes 4 bytes with a fixed value.
	Register	The field includes 2 bytes that are taken from a memory register or are written to it, depending on whether the field is used in the query or response.
	RegisterHigh	This field includes 1 byte, whereby only the Highbyte of the register is used.
	RegisterLow	This field includes 1 byte, whereby only the Lowbyte of the register is used.
	Checksum	A checksum of the telegram is inserted or expected.

Command	Value	Function
Value	This line defines where the values come from or where they are written. Here, the option is strongly dependent on the type and whether the field is used in the Query or Response. The following are possible:	
	User	In the main window, you have to specify the numerical value to be transferred.
	Assign	The value is assigned to a register
	NodeAddress	The address of the device is entered.
	FunctionID	The command Id is entered.

Define Command

In the structure tree you now see the created command with all associated fields. Values have already been entered for some fields. These values are transferred automatically from the device or from the command.

✓ In the Command Editor you have compiled or selected a command.

◦ Click on a field.

⇨ In the "Properties" window you can now define the values for this field.

◦ Check all fields and supplement the missing values.

⇨ You have now defined your command. In the next step you can generate your script.

Field	Meaning
Address	Address of the device (node)
Function	Number of the ModbusRTU function
Register Address	Address from where the register is read/written
Register Value	Definition of the source or destination from where the values are read/written
Checksum	Returns a checksum
Quantity	Number of registers that should be read or written
Byte Count	Number of bytes read/written

Table 1: Input value of the command fields

Example

✓ The input registers 4-7 are to be read from a ModbusRTU server with the device address 7 and written in the module to the memory register 0x1410-0x1413.

✓ The registers are to be read every 20 ms.

✓ How often the register was read should be counted in the register 0x1405.

✓ Any errors that occur should be counted in the register 0x1406.

◦ Open a new project with "File/New".

◦ Assign the following properties in the "Properties" window:
- ErrorCounterAddress: 0x1406
- LoopCounterAddress: 0x1405
-WaitTime: 20 (ms)

◦ Right-click on the "Network" element.

◦ Select "Add Node" in the context menu

⇨ You have now added a new device.

◦ Enter "7" as address in the field "Property".

◦ Right-click on the newly generated device (Node) in the structure tree.

◦ Select "Add Command" in the context menu.

⇨ The "Command Editor" window opens.

◦ Click on the predefined command "Read Input Registers"

◦ Click on "Select Command"

⇨ The command now appears in the structure tree

◦ Check all data fields for any missing values. You cannot assign the address and function values. These values are transferred automatically from the device.

✓ Assign the values to the Query in the "Properties" window:

◦ Address: 4

◦ Quantity:4

✓ Assign the values to the Response in the "Properties" window:

◦ Byte Count: 8. In Modbus, 4 registers with two bytes each are read. You have to enter the number of bytes in this field.

◦ Location: 0x1410

◦ Input registers and checksum are returned automatically once the script is running on the module.

◦ Click on the "Start Debugging" button

⇨ You have defined a script. It is now running on your module.

Generate script

You have 2 options for generating a script:

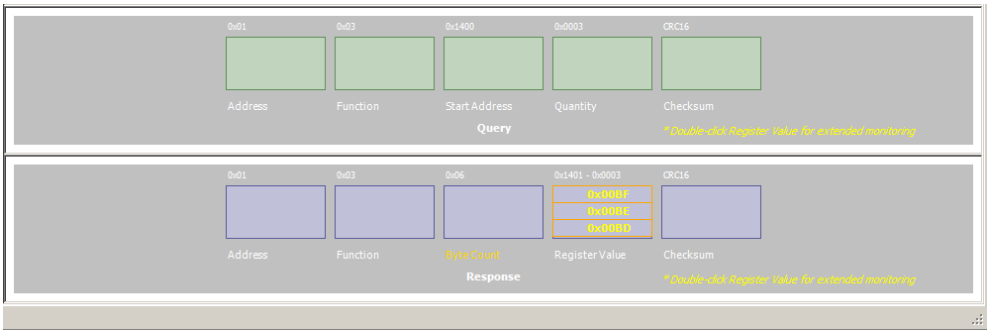
– Select "Script>Generate Script" in the menu bar
– In the toolbar click on 

Checking script

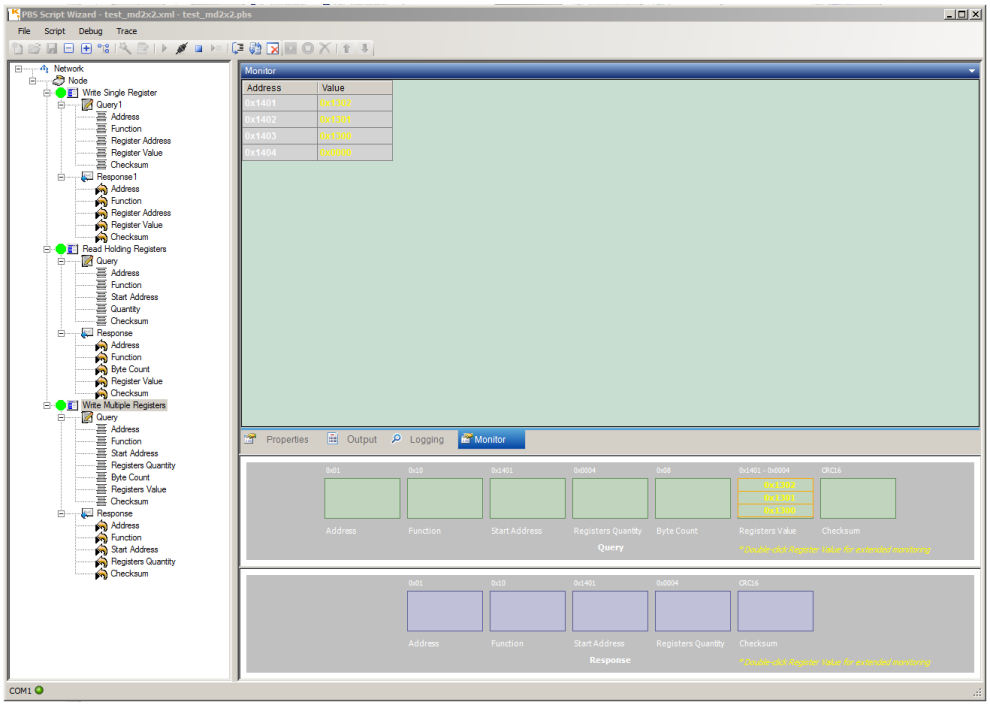
◦ Double-click to select a command in the structure tree.

⇨ The following window will open:

Here, you can trace the script sequence in the memory registers.



Select a register by double-clicking to trace the sequence in the individual registers.



Error in Script

If an error occurs in the script sequence, the Wizard will indicate this:

– The command concerned is highlighted by a red status indicator in the structure tree.

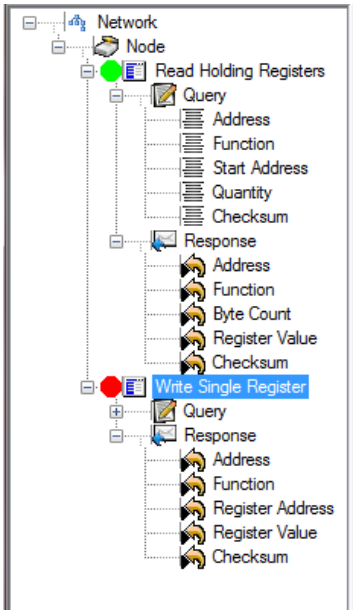


Illustration 23: Error indicator in the structure tree

– The error is highlighted in red in the Trace Monitor and reported in the message window.

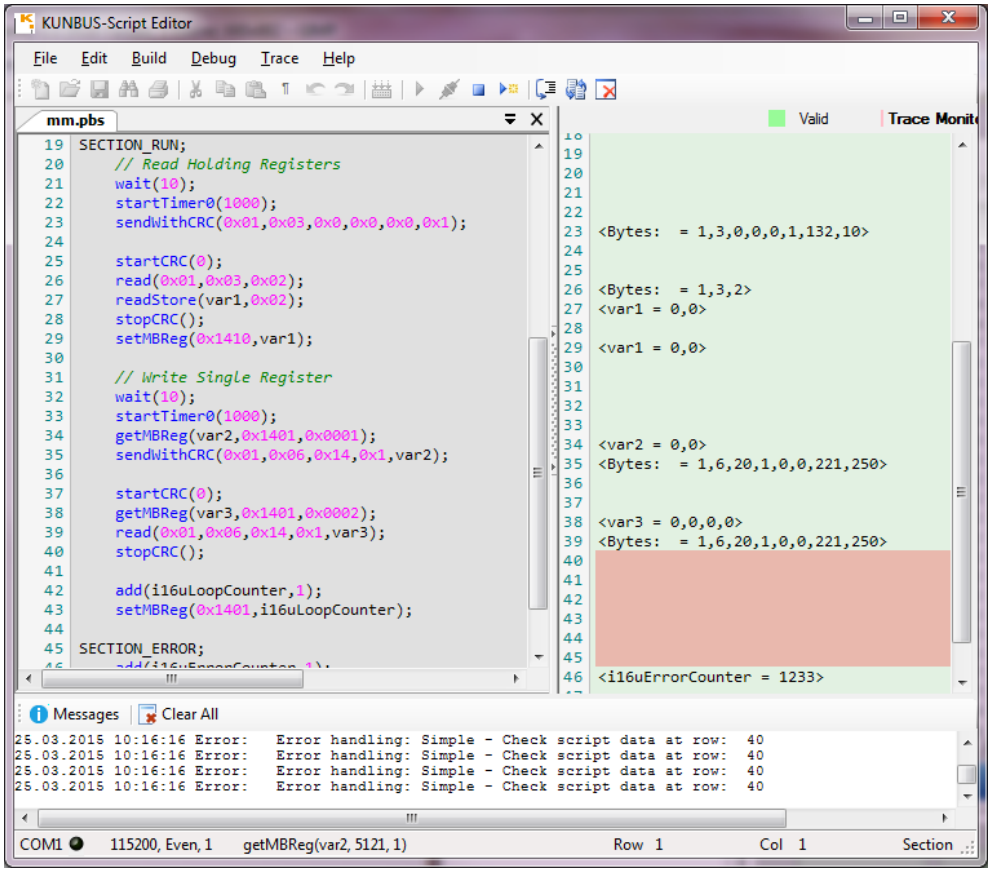


Illustration 24: Error display in the trace monitor message window

Tip!: Switch to the Script-Editor with  . The generated script is displayed there. It can also be compiled and run on the module. When you open the Trace View in the Script Editor, the source code is displayed as well as the values of the variables for the listed lines. Any code not executed is highlighted in red. Frequent error sources are timeouts when waiting for the response and the use of register addresses that do not exist in the module.

Structure tree

The structure tree is a visualization of your script. Here, you can find information on the status, function and content of the individual commands. Everything displayed is dependent on the contents in your script.

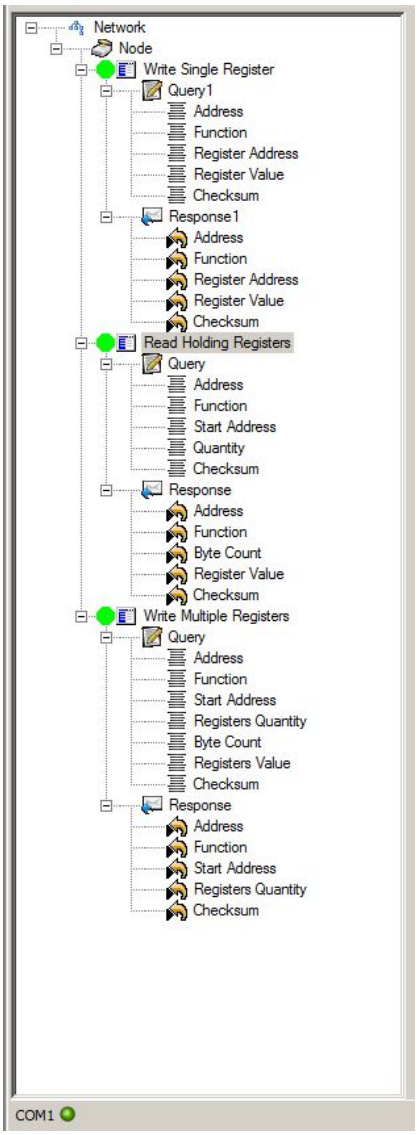


Illustration 25: Structure tree

Status indicator: The colored status indicator before the command is active/updated once a trace has been loaded from the module. It indicates whether the command is being executed or whether an error has occurred.

Green	No Error
Red	Error
Grey	No connection to the network or no information about the command is available because an error occurred previously

Contents of the command: Below the function of the command is a list showing what elements the command consists of.

First you will see whether your command sends a query to the module and receives a response from the module.

The level below query and response shows the elements that the query and response consist of.

4 Appendix

4.1 Example Script

```
SECTION_CONFIG;
configPort(38400,even,1,8); // BaudRate, Parity, Stop, Data
setBigEndian(); // set the byte-order to Big Endian
decl(var1, 1); // declare var1 as one-Byte variable
decl(var2, 2); // declare var2 as two-Byte variable
decl(var4, 4); // declare var4 as four Byte variable
declArray(var5, 10); // declare var5 as array with the size of 10 bytes
SECTION_INIT;
setMBReg(0x1401, 1); // set register 0x1401 to 1 while the script runs
set(var1, 0x03); //var1 has the value 3
set(var2, 0x0101);
set(var4, 0x12345678);
SECTION_RUN;
wait(100); // wait 100 ms
startTimer0(100); // expect response in 100 ms or error
add(var2, var1); //add var2 and var1 and save the result in var2
minus(var4, var1);
divide(var4, var1);
multiply(var4, var2);
setArray(var5, 3, 4, 250, var1, var1); // var5 = {3, 4, 250, 3, 3}
sendWithCRC(0x11, 0x1201, var2, var5);
// send data 0x11, 0x12, 0x01, high byte of var2, low byte of var2, 3,
4, 250, 3, 3, CRC, CRC
startCRC(0); // start CRC for subsequent bytes
readStore(var5, 0x05); // store 5 bytes in var5
stopCRC(); // read 2 more bytes and compare them to the calculated
CRC
SECTION_ERROR;
setMBReg(0x1401, 2); // set register 0x1401 to 2 if an error occurred
SECTION_FATAL;
stop(); // stop
```

4.2 Overview of the Script Commands

add(*parameter1, parameter2*);

This command counts the value from *parameter2 to parameter1* and stores it in *parameter1*. In the course of this, *parameter1* must be a variable.

configPort(*Baudrate, Parity, StopBit, DataBit*);

This command configures the serial interface.

The first parameter defines the baudrate. Permissible values are 110, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600 and 115200.

The parity is defined as text. 'even', 'odd' and 'none' are allowed.

The third argument defines the number of stop bits, value range 0 tos 2.

The fourth argument defines the number of data bits, value range 7 to 8.

decl(*name ,size*);

This command declares a variable with name *name* and size *size*. Variables of 1, 2 and 4 bytes in size are possible. A total of up to 20 variables and arrays can be used in a script.

declArray(*name, size*);

This command declares an array name (*name)* and size (*size).*

- The size can have values between 1 and 128.
- All arrays together must not be longer thans 256 bytes.

divide(*parameter1, parameter2*);

This command divides the value of *parameter1 by the value of parameter2.* The result is stored in *parameter1*. *parameter1* must be a variable. If parameter2 has the value 0, an exception is triggered.

getMBReg(*value, address*);
getMBReg(*value, address, number*);

This command stores the value of the memory register with the address *address* in the variable *value*. Value can be a variable or an array. The *number* parameter specifies the number of registers to be read. If it is not specified, one register is read.

getMBRegHigh(*value, address*);
getMBRegHigh(*value, address, number*);
getMBRegLow(*value, address*);
getMBRegLow(*value, address, number*);

This command stores the value of the memory register with the address *address* in the variable *value*. Value can be a variable or an array. The *number* parameter specifies the number of registers to be read. If it is not specified, only one register is read. Only the higher-value or lower-value byte is read from the registers.

If-expression

If par1 cmp par2 then … endif

If par1 cmp par2 then … else … endif

The condition par1 cmp par2 is checked:

- If the condition is true, the commands are executed after then.
- If the condition is false and an else-branch exists, the commands are executed after else.
- The script is then continued after endif.

The condition consists ofs 1 variables or constants with one comparative operator in between. The following operators are possible:

Operator	Description	Example for true	Example for false
==	Equal to	1 == 1	1 == 2
!=	Unequal to	1 != 2	3 != 3
<	Less than	1 < 2	2 < 1
<=	Less than or equal to	1 <= 3	3 <= 1
>	Greater than	3 > 2	2 > 3
>=	Greater than or equal to	3 >= 3	2 >= 3
bs	Bit set	0x02 bs 1	0x02 bs 2
bc	Bit not set	0x10 bc 3	0x10 bc 4

bs (bit set) and bc (bit cleared) check whether or not a bit is set in a value. The first parameter is the value, in which the bit is checked. The second is the bit position, whereby the lowest value bit is checked with 0 and 31 is checked for the highest value bit. The number 0x10 would be 10000b as a binary number, i.e. only bit 4 is set. Thus, (0x10 bs 4) would be true and (0x10 bc 4) false.

incCounter0(*value*);
incCounter1(*value*);

There are 2 counters: Counter0 and Counter1. With the resetCounter0 or resetCounter1 commands you can set the counter to 0. Each time the functions incCounter0 or incCounter1 are invoked, the counter is incremented.

minus(*parameter1, parameter2*);	This command subtracts the value of *parameter2 from parameter1 and stores it in parameter1*. In the course of this, *parameter1* must be a variable. Info!: The value of parameter1 must be greater than the value of parameter2 so that the result is not negative. If the case of a negative result, an exception is triggered.
modulo(*parameter1, parameter2*);	This command divides the value of *parameter1 by parameter2. The division rest is stored in parameter1*. *parameter1 must* be a variable. If parameter2 has the value 0, an exception is triggered.
multiply(*parameter1, parameter2*);	This command multiplies both parameters. The result is stored in *parameter1*. In the course of this, *parameter1* must be a variable.
read (parameter1 , paramter2, …);	This command compares the number of bytes received with the values of the arguments. If the values are different, an exception is triggered. **Info!:**If startTimer0() was invoked previously and not enough bytes were received in the specified time, an exception will also be triggered. **Info!**:The function first receives the specified number of bytes. After that, it compares the data values. Example: read(1,2,3); receives 3 bytes and then checks whether the first byte is 1. If another response is desired, read must be invoked several times: read(1); read(2); read(3); – The function can have between 1 and 16 parameters. – All data types are allowed.
readSkip(*length*);	This command receives *length* bytes. These bytes are taken into account during the checksum calculation.
readStore(*parameter1, parameter2*);	With this command you store the number of bytes to be received in *parameter2* . These are saved in *parameter1* . – *parameter1 must* be a variable or an array.
resetCounter0(); resetCounter1();	See incCounter0

send(parameter1, parameter2, …);

This command sends data via the serial interface.

- The function can have between 1 and 16 parameters.
- All data types are allowed.
- In the case of 2 or 4 byte constants and variables, the bytes are output according to the set sequences (setBigEndian() or setLittleEndian()).
- In the case of arrays, the bytes are sent in the same sequence, in which they are in the array.

sendWithBCC(parameter1, parameter2, …);

This command sends data via the serial interface.

- The function can have between 1 and 16 parameters.
- All data types are allowed.
- In the case of 2 or 4 byte constants and variables, the bytes are output according to the set sequences (setBigEndian() or setLittleEndian()).
- In the case of arrays, the bytes are sent in the same sequence, in which they are in the array.

In addition, a 1-byte checksum is also sent after the data, which is calculated from the data. You can define the type of checksums using the function startBCC().

sendWithCRC(parameter1, parameter2, …);

This command sends data via the serial interface.

- The function can have between 1 and 16 parameters.
- All data types are allowed.
- In the case of 2 or 4 byte constants and variables, the bytes are output according to the set sequences (setBigEndian() or setLittleEndian()).
- In the case of arrays, the bytes are sent in the same sequence, in which they are in the array.

In addition, a 2-byte checksum is also sent after the data, which is calculated from the data with the CRC-16 procedure.

set(*name, value*);

This command assigns the value *value* to the variable *name.* The values can be constants or variables.

setArray(*name, parameter1, paramter2, …*);

The values of the remaining parameters are written in the array *name*. If only one parameter is available, it can be a 1, 2 or 4 byte variable or constant. During writing, the byte sequences defined by setLittleEndian() or setBigEndian() is used. If there is more than one parameter, only 1 byte constants or variables can be specified.

setBigEndian();

This command defines the byte sequence to Big Endian.

setLittleEndian();

This command defines the byte sequence to Little Endian.

setMBReg(*address, value*);

This command writes the value in *value* in the memory address with the address *address*. The *value* must be a constant, variable or an array. The length of the array must be a multiple of 2.
Info!: The memory registers are always two bytes in size. A variable of four bytes or an array that contains more than two bytes is further written in the following register. The *address* is incremented.

setMBRegHigh(*address, value*);
setMBRegLow(*address, value*);

This command writes the value in *value* in the memory address with the address *address*. With setMBRegHigh() only the higher-value byte is written, the lower-value byte remains unchanged. With setMBRegLow() only the lower-value byte is written accordingly.

value can be a constant, variable or an array. With this function one byte each is written in the memory register. A variable or an array that contains more than two bytes is further written in the following registers. The *address* is incremented.

startBCC(*mode*);

This command starts a checksum calculation of the data received. It makes no difference here which read-function receives the data. With *mode* the method of the checksum calculation is selected:

mode = 0 -> ZERO_MINUS_SUM method:

Calculates the sum of all bytes received and subtracts the result from 0.

mode = 1-> BITWISE_NEGATION:

Calculates the sum of all bytes received and negates the result bit by bit.

mode = 2-> ADDITION:

Calculates the sum of all bytes received

Mode = 3 -> XOR

Concatenates all bytes bit by bit with XOR.

startCRC(*mode*);

With this command a checksum calculation starts on the data received. It makes no difference here which read-function receives the data. With *mode* the method of the checksum calculation is selected, however, only 0 is permitted for a CRC16 calculation at the moment.

startTimer0(*value*);	This command is used to limit the wait time of a subsequent read function. A timer is started that runs for the *value* milliseconds. If it expires while a read function is still waiting for data, an error is then triggered and the next SECTION_ERROR is executed.
stop();	This command stops the execution of the script. Restart the module to restart the script.
stopBCC();	**Prerequisite:** You have started a checksum calculation. With this command stopBCC receives a further byte with each incoming read function. stopBCC compares this byte with the calculated checksum. If the checksums do not match, an exception is triggered.
stopCRC();	Prerequisite: You have started a checksum calculation. If, after invoking StartCRC, a byte with a read function is received, the checksum is updated. stopCRC now receives two additional bytes and compares these with the calculated checksum. If these do not match, an error is triggered.
wait(*time*);	This command interrupts the program sequence for the for the specified *time* . The value is specified in milliseconds.
waitSequence(*parameter1, parameter2, …*);	This command interrupts the program sequence until the specified byte sequence is received. Permissible parameters are constants and variables.