User Manual

netSCRIPT

Programming Language for Serial Communication

V1.3.x.x

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC090801UM06EN | Revision 6 | English | 2010-07 | Public | Released

Table of Contents

1 Introduction

1.1 About the User Manual

This user manual describes the programming language netSCRIPT. This is used to transfer data between different network protocols within Hilscher devices.

netSCRIPT is based on the script language Lua and extended by special communication functions.

netSCRIPT programs are created with the software SYCON.net and are transferred into the target device. In the target device these are processed cyclically with an interpreter.

The debugging of the script file is possible with the additional software net-SCRIPT_Debugger.exe and can be used on a standard PC. This program has to be installed separately to the configuration program SYCON.net.

1.1.1 List of Revisions

Index	Date	Chapter	Revision
4	2009-12-16	13.1.9	Section *Exit the* Debugger added
			Example Time adapted
5	2010-05-21		netScript Version 1.2.x.x
		5.1.8	Comments to the example
		6.2, 6.3, 6.5	Error corrections
		7.1	Additions
		8.3	Additions
		8.3.5 8.3.6	port.STA_PATTERN_MATCHED → port.STA_PATTERN_MATCH
		9.1	Additions
		10	Corrections
		10.9.3.1 10.9.3.2	Sequence of sections exchanged
		11.1	Additions
		11.2	New
		12.1.2	Additions
		14.7	Additions
6	2010-07-13		netScript Version 1.3.x.x
		10	Chapter Functions for the Communication with the superordinated I/O Network with new BusIO communication without handshake

Table 1: List of Revisions

1.1.2 Reference on netSCRIPT, Hardware, Software and Firmware

Note: The listed hardware revisions, firmware and driver versions or versions of the configuration software and SYCON.net configuration tool functionally belong together.
For existing hardware installation updates the firmware, the driver and the configuration software.

netSCRIPT runs on the following hardware

Device	Revision
NT 100-RE-RS	1
NT 100-DP-RS	1
NT 100-CO-RS	1
NT 100-DN-RS	1
NT 100-DN-RS	1

Table 2: Reference on Hardware

netSCRIPT

netSCRIPT	Version
Lua	5.1
netSCRIPT	1.2.x.x

Table 3: Reference on netSCRIPT

Software

Software	Software Version
SYCON.net	1.210.x.x
netSCRIPT_Debugger.exe	1.0.xxxx

Table 4: Reference on Software

Firmware

Firmware File	Firmware Version
NTxxxNSC.NXF	1.3.x.x

Table 5: Reference on Firmware

1.1.3 Conventions in this Manual

Operation instructions, a result of an operation step or notes are marked as follows:

Operation Instructions:

➢ < instruction >

Or

1. < instruction >
2. < instruction >

Results:

↛ < result >

Notes:

Important: <important note>

Note: <note>

<note, were to find further information>

netSCRIPT notation in this manual:

Syntax	Meaning
command	netSCRIPT commands are written with small letters in general. Functions can contain capital letters or can be written completely in capital letters. netSCRIPT distinguishes between small and capital letters (case sensitive)
var1	Variable name
_par	Parameter are shown with small letters
_par$_i$	Parameter with subscripted index means that a list of operants can be specified
Value	The value of variables / parameter is shown in italic
[ ]	Alternatively commands are between simple square brackets

Table 6: Script Description Syntax

When in the text a bold text between quotation marks is written „**command**“, then this is a direct reference to the described function respectively its parameter.

If a “**:**“ is at the beginning of the function name, then add the the prefix of the corresponding Open function in front of the “**:**“ of the call of this function.

1.1.3.1 Documentation

The following documentation overview gives information, for which items you can find further information in which manual.

All these documents are available on the CD delivered with the device underneath the directory **Documentation**, in Adobe Acrobat® Reader format (PDF).

Manual	Content	Document name
User manual	Configuration of NT 100 devices	netTAP100_usermanual_en.pdf

Table 7: Documentation

Further information about the script language Lua can be found on the following Internet pages:

http://www.lua.org

and documentation:

http://www.lua.org/manual/5.1/

http://lua.gts-stolberg.de/

1.2 Legal Notes

1.2.1 Copyright

1.2.2 Important Notes

The user manual, accompanying texts and the documentation were created for the use of the products by qualified experts, however, errors cannot be ruled out. For this reason, no guarantee can be made and neither juristic responsibility for erroneous information nor any liability can be assumed. Descriptions, accompanying texts and documentation included in the user manual do not present a guarantee nor any information about proper use as stipulated in the contract or a warranted feature. It cannot be ruled out that the user manual, the accompanying texts and the documentation do not correspond exactly to the described features, standards or other data of the delivered product. No warranty or guarantee regarding the correctness or accuracy of the information is assumed.

We reserve the right to change our products and their specification as well as related user manuals, accompanying texts and documentation at all times and without advance notice, without obligation to report the change. Changes will be included in future manuals and do not constitute any obligations. There is no entitlement to revisions of delivered documents. The manual delivered with the product applies.

Hilscher Gesellschaft für Systemautomation mbH is not liable under any circumstances for direct, indirect, incidental or follow-on damage or loss of earnings resulting from the use of the information contained in this publication.

1.2.3 Exclusion of Liability

The software was produced and tested with utmost care by Hilscher Gesellschaft für Systemautomation mbH and is made available as is. No warranty can be assumed for the performance and flawlessness of the software for all usage conditions and cases and for the results produced when utilized by the user. Liability for any damages that may result from the use of the hardware or software or related documents, is limited to cases of intent or grossly negligent violation of significant contractual obligations. Indemnity claims for the violation of significant contractual obligations are limited to damages that are foreseeable and typical for this type of contract.

It is strictly prohibited to use the software in the following areas:

- for military purposes or in weapon systems;
- for the design, construction, maintenance or operation of nuclear facilities;
- in air traffic control systems, air traffic or air traffic communication systems;
- in life support systems;
- in systems in which failures in the software could lead to personal injury or injuries leading to death.

We inform you that the software was not developed for use in dangerous environments requiring fail-proof control mechanisms. Use of the software in such an environment occurs at your own risk. No liability is assumed for damages or losses due to unauthorized use.

1.2.4 Warranty

Although the hardware and software was developed with utmost care and tested intensively, Hilscher Gesellschaft für Systemautomation mbH does not guarantee its suitability for any purpose not confirmed in writing. It cannot be guaranteed that the hardware and software will meet your requirements, that the use of the software operates without interruption and that the software is free of errors. No guarantee is made regarding infringements, violations of patents, rights of ownership or the freedom from interference by third parties. No additional guarantees or assurances are made regarding marketability, freedom of defect of title, integration or usability for certain purposes unless they are required in accordance with the law and cannot be limited. Warranty claims are limited to the right to claim rectification.

1.2.5 Export Regulations

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different counters, in particular those of Germany and the USA. The software may not be exported to countries where this is prohibited by the United States Export Administration Act and its additional provisions. You are obligated to comply with the regulations at your personal responsibility. We wish to inform you that you may require permission from state authorities to export, re-export or import the product.

1.2.6 Registered Trademarks

Windows® 2000 and Windows® XP are registered trademarks of the Microsoft Corporation.

Adobe-Acrobat® is an registered trademark of the Adobe Systems Incorporated.

Rocksoft is an registered trademark of Rocksoft Pty Ltd. Australia.

1.3 Licenses

Lua is free software: it can be used for any purpose, including commercial purposes.

http://www.lua.org

Lua is a scripting language which is developed by

Pontifícia Universidade Católica do Rio de Janeiro- © PUC-RIO – 2009 Rua Marquês de São Vicente, 225, Gávea - Rio de Janeiro, RJ - Brasil - 22453-900; Cx. Postal: 38097 - Telefone: (55 21) 3527-1001

2 Description and Requirements

2.1 Description

netSCRIPT is a Lua based script programming language, which is used for creation and loading of the programs into the target hardware and requires the PC configuration and diagnostic program SYCON.net for it.

Using the netSCRIPT language, it is possible to connect devices which have a serial RS232, RS422, RS485 communication interface (UART port) to a superior control unit via an additional bus system (Bus IO).

netSCRIPT programs are managed with SYCON.net. Editing, syntax check and the storage are done with SYCON.net. The netSCRIPT program is transferred into the target device together with the complete network configuration. The serial protocol and its sequence can be programmed as desired. Functions for send and receive via UART are available. The transfer to the superior bus system is done via data buffers IN and OUT. Additional functions are available for this access. Furthermore status information can be transferred.

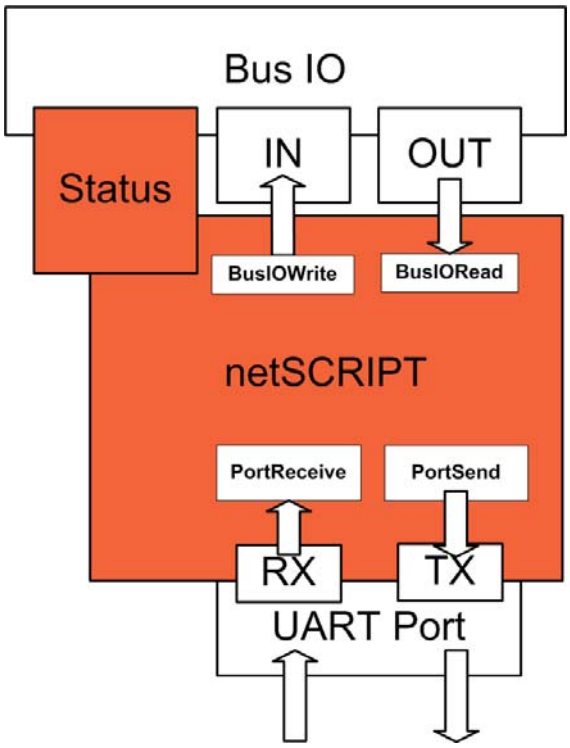


Figure 1: netSCRIPT communication channels

2.2 System Preconditions

1. PC with COM and USB 1.1 interface and CD-ROM drive.
2. Windows® 2000/Windows® XP
3. SYCON.net and netSCRIPT Debugger (programming and debugging tools)
4. A netSCRIPT capable device as target hardware of the netSCRIPT program, e. g. netTAP 100

3 Editor for netSCRIPT

The netSCRIPT editor is a component of the SYCON.net software.

3.1 Invoke the Editor

The script file management and edit function is available via the context menu of the netSCRIPT capable device. In this section this is explained with the NT 100.

3.1.1 Device Selection

The netSCRIPT editor is a component of the SYCON.net software.

- Start the SYCON.net software on your PC. Select the project which contains the device which you want to program.

The following window opens:

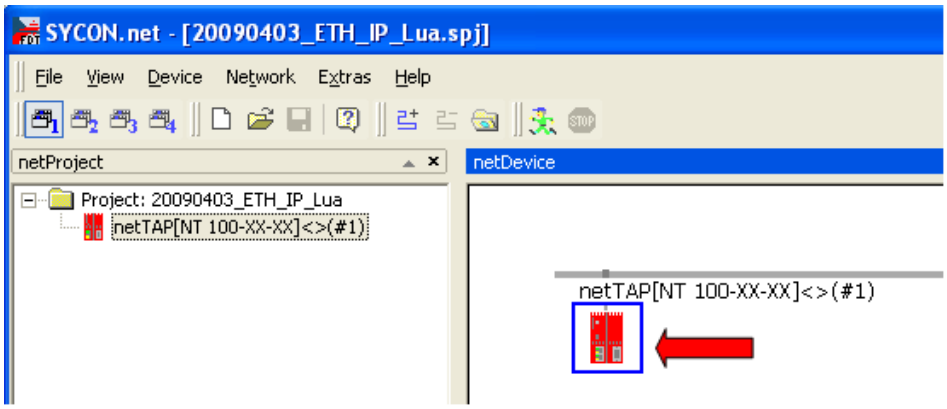
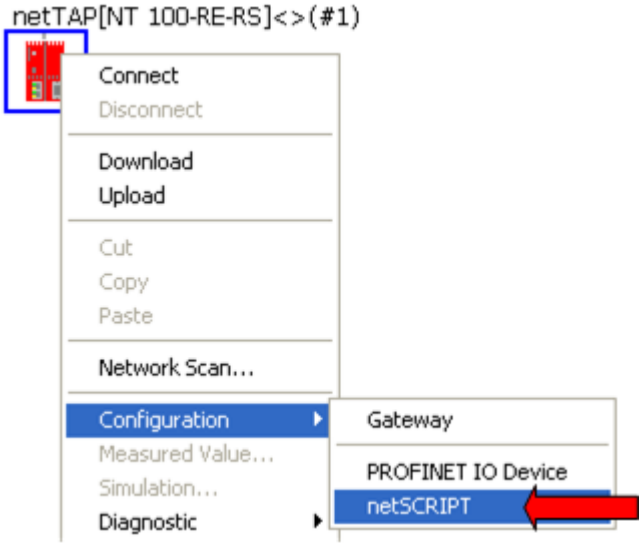


Figure 2: Select device which is script capable

- Open the context menu with a right mouse click on the device symbol.

The following window opens:



- Select **Configuration** > **netSCRIPT**.
- This opens the configuration window of the UART interface parameter:

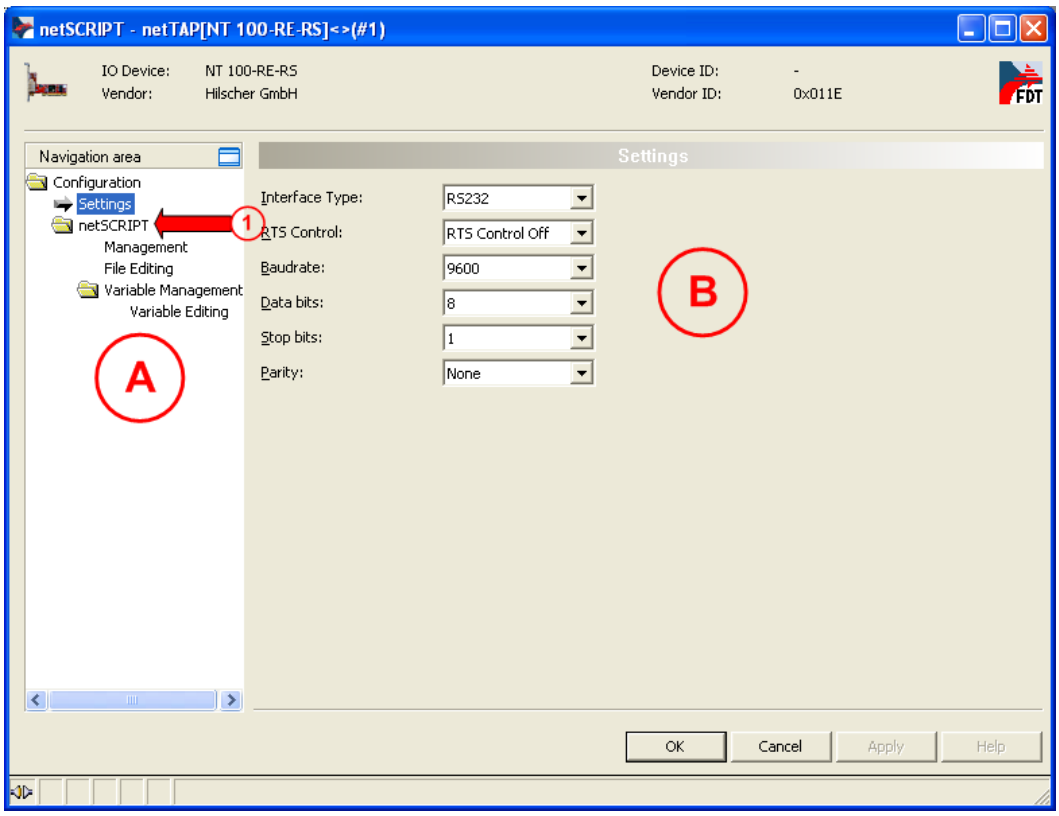


Figure 3: SYCON, UART Configuration

The settings made here in the window pane Ⓑ, become effective only for the UART interface, if in the script the function **PortReadConfigDb()** is called to read the settings (see section *PortReadConfigDb* on page 82). These settings have to be transfered when the interface is opened.

➢ Select in the navigation area Ⓐ netSCRIPT, to reach the script management.

3.2 Program Editor

3.2.1 Script File Management

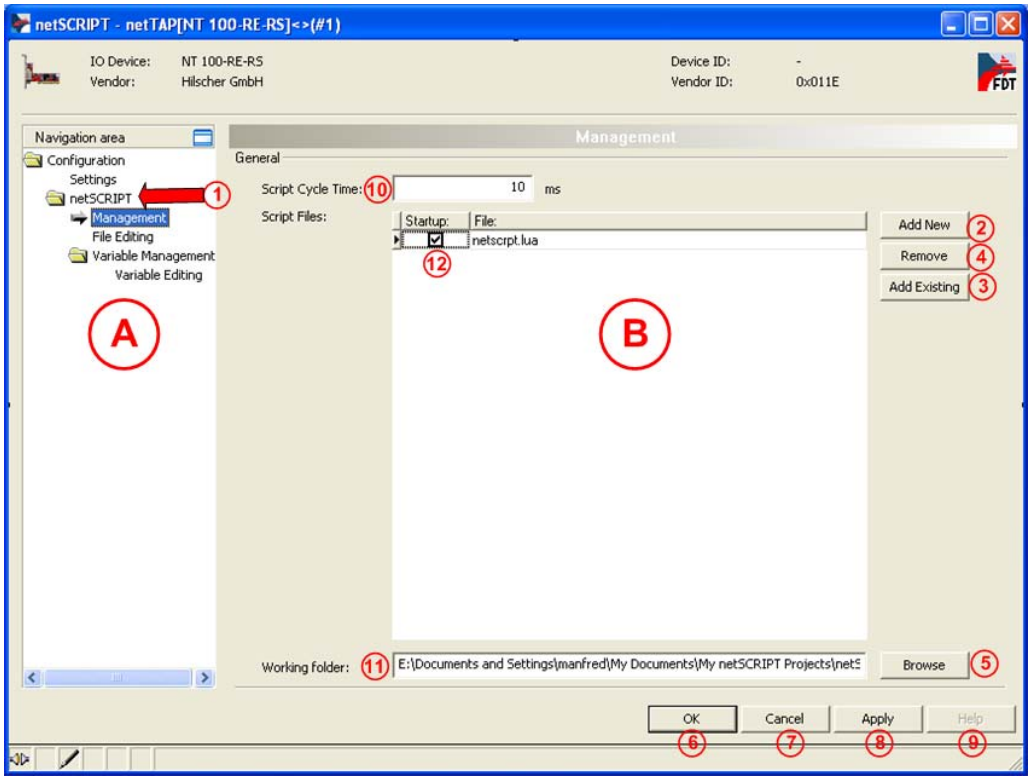


Figure 4: Script management

① Select in the window pane Ⓐ (navigation area) **netScript,** then **File Management**. This opens the configuration area shown.

Ⓑ In this window pane the script files available are listed.

② With the button "Add New" you can create a new script file which gets a standard name "netscript.lua" (as in the window pane Ⓑ displayed).

③ With this button, a script file can be imported in the SYCON project. It is also copied into the selected working directory ⑪.

④ With this button the selected file from window pane Ⓑ is deleted in the window pane Ⓑ, and is deleted from working folder which is selected at ⑪ .

⑤ With this button, the working directory ⑪ in which the script files are stored can be selected.

⑥ With this button, the entries of this window are saved. Afterwards the window is closed.

⑦ With this button you exit the window without saving.

⑧ With this button, the data are saved. The window remains open.

⑨ With this button, you reach the help topics pertaining to this window.

⑩ In this field, the cyclic time of the script is set for the device.

⑪ The directory in which the script file is saved is displayed.

⑫ With the checkbox "**Startup**", the script file is marked, which should be launched cyclically.

3.2.2 Script File Editing

In the following window, a script can be created or an existing script can be edited.

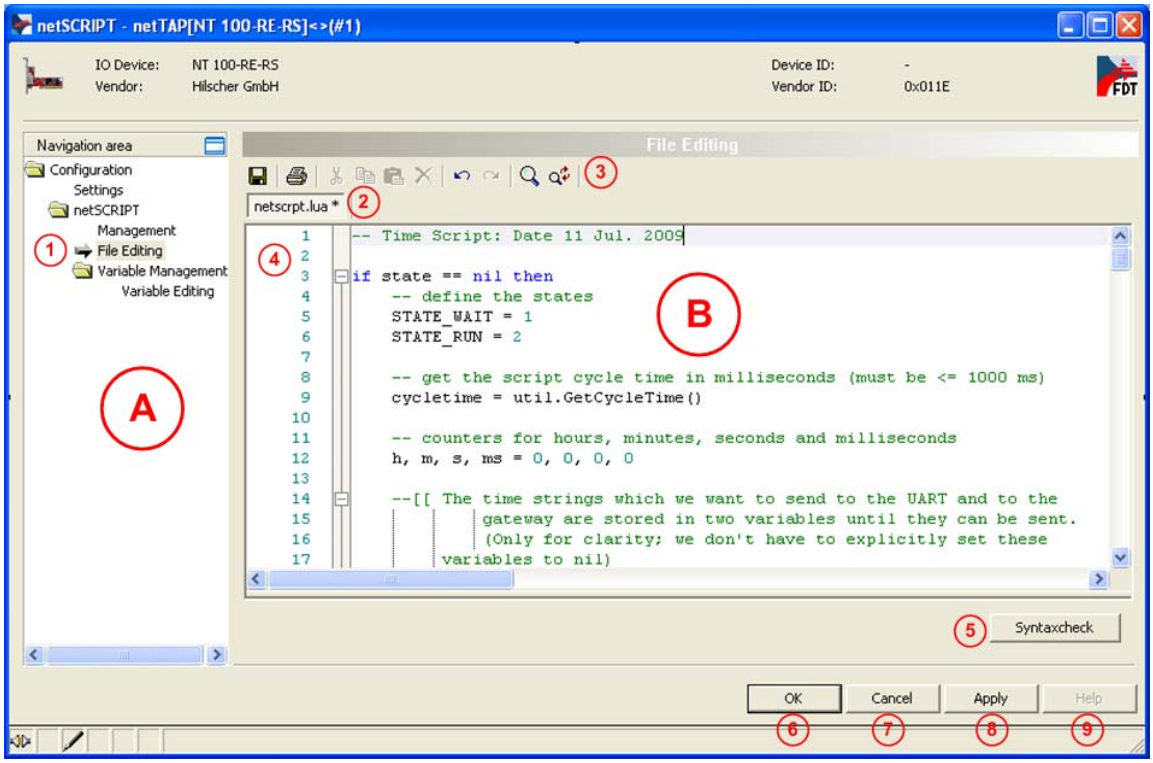


Figure 5: Editor Windows

① Select in the navigation area Ⓐ "**File Editing**". With it the editable script file is displayed from line ②.

Ⓑ In this window pane, the selected script file is displayed.

③ In this line, tools for script editing are offered.

④ In this column, the line number of the script is displayed.

⑤ With this button, the actual script file in the editor window can be submitted to a syntactic check.
The additional window pane Ⓒ opens (see Figure 6), where the result of the checking is displayed.

⑥ With this button, the entries of this window are saved. Afterwards the window is closed.

⑦ With this button, you exit the window without saving.

⑧ With this button, the data are saved. The window remains opened.

⑨ With this button, you reach the help topics pertaining to this window.

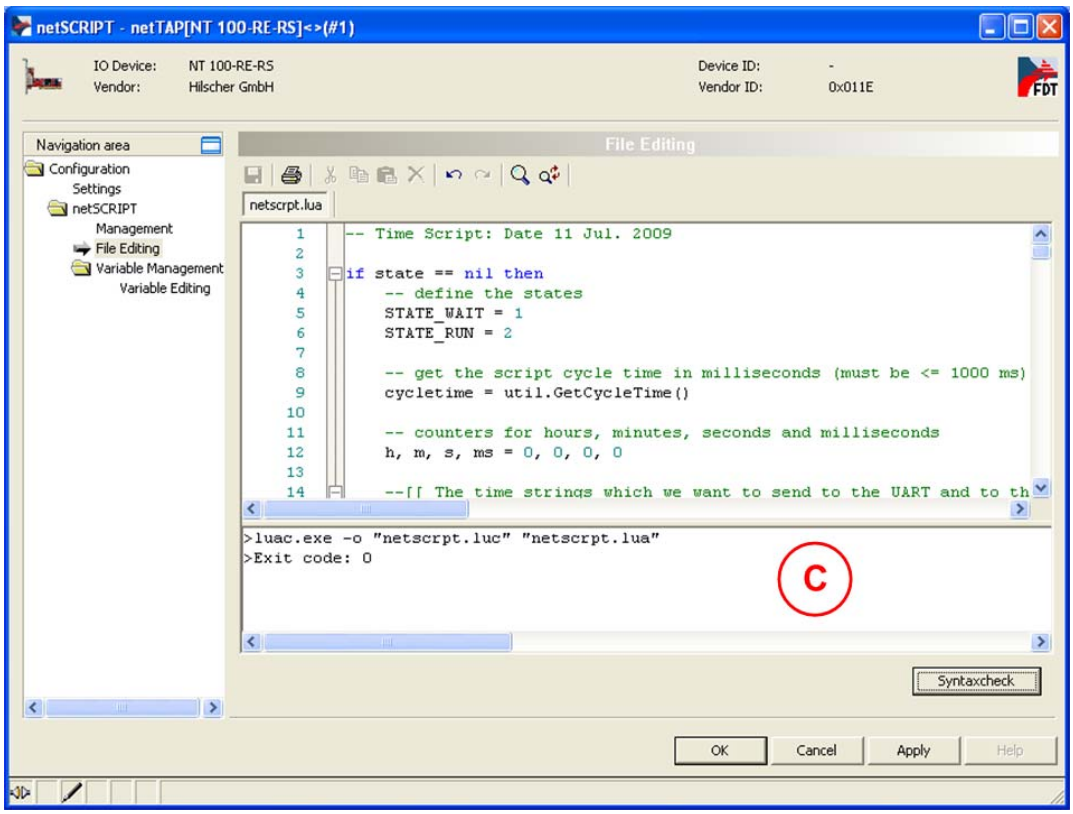


Figure 6: Editor Window Syntax check

In the figure above in window pane Ⓒ the result of the compilation is displayed. If the last entry in this window pane is not "**>Exit code: 0**", then the previous lines display where the error can be searched in the Script.

3.3 Configurable Variables (Parameters)

netSCRIPT variables are created basically in the script programs either locally or globally and have initial values. Additional configurable variables (parameters) can be defined outside of the script program with SYCON.net and values assigned to them. These variables (parameters) can be evaluated by the script.

With the download of the configuration and the script these variables are transferred into the target device and are stored in the netSCRIPT table VAR. These additional variables offer a flexible control of the program without changing the script code only by changing the value.

The usage of these variables within netSCRIPT is described in section 3.3.2.8 on page 28.

3.3.1 Definition of Configurable Variables (Variable Management)

➢ Choose in the navigation area Ⓐ under „**Configuration > netSCRIPT > Variable Management >Variable Editing**“ ①.

➢ This opens the following window.

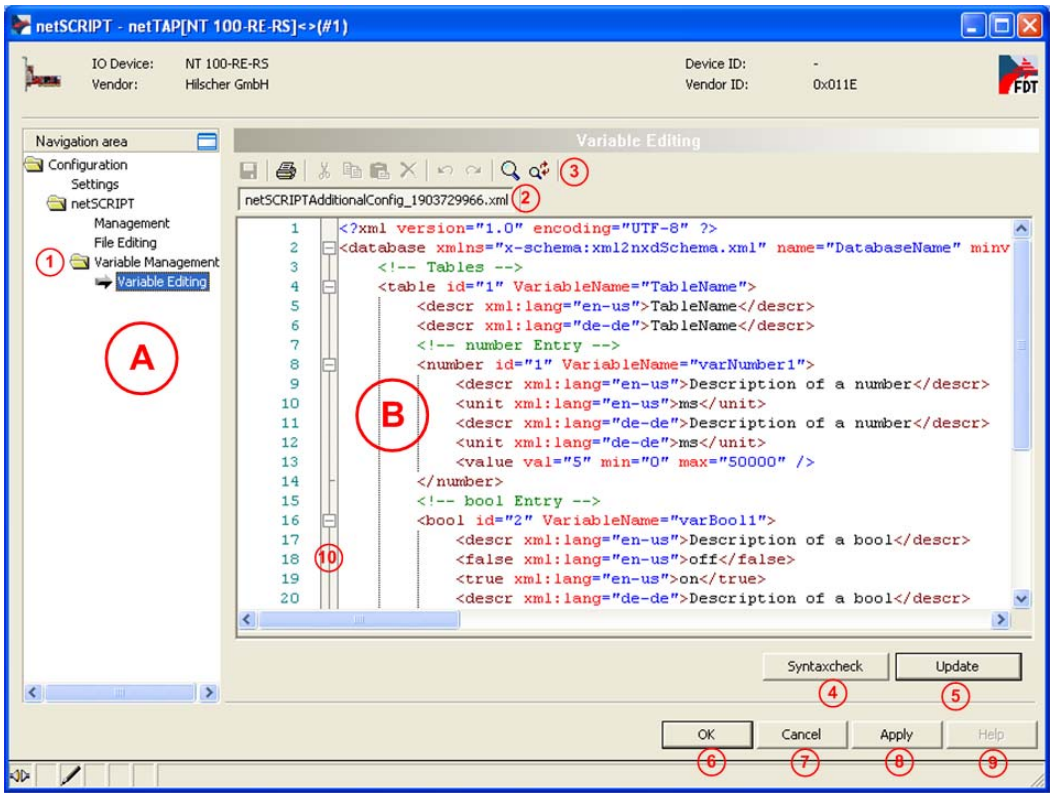


Figure 7: Configurable Variable Definition

Ⓑ In this window pane, an .xml file is displayed which is the base to create configurable variables for the netSCRIPT program. The syntax of the file is explained in section 3.2.2 on page 19.

② In this line, the file name of the variable file is displayed which can be edited in the window pane Ⓐ.

③ In this line, tools are offered to edit the XML file.

④ With this button, a syntax check of the XML file can be started. If the syntax check is not successful, then in window pane Ⓒ suitable error messages are displayed (see Figure 8).

⑤ With this button, the contents of the window pane Ⓑ are compiled and the contents in the window pane Ⓐ are updated. If the compilation is not successful, window pane Ⓒ opens with suitable error messages (see Figure 8).

⑥ With this button, the entries of this window are saved. Afterwards, the window is closed.

⑦ With this button, you exit the window without saving.

⑧ With this button, the data are saved. The window remains open.

⑨ With this button, you reach the help to this window.

⑩ In this column, coherent blocks can be minimized or expanded, by a click on the "**-**" or "**+**" signs.

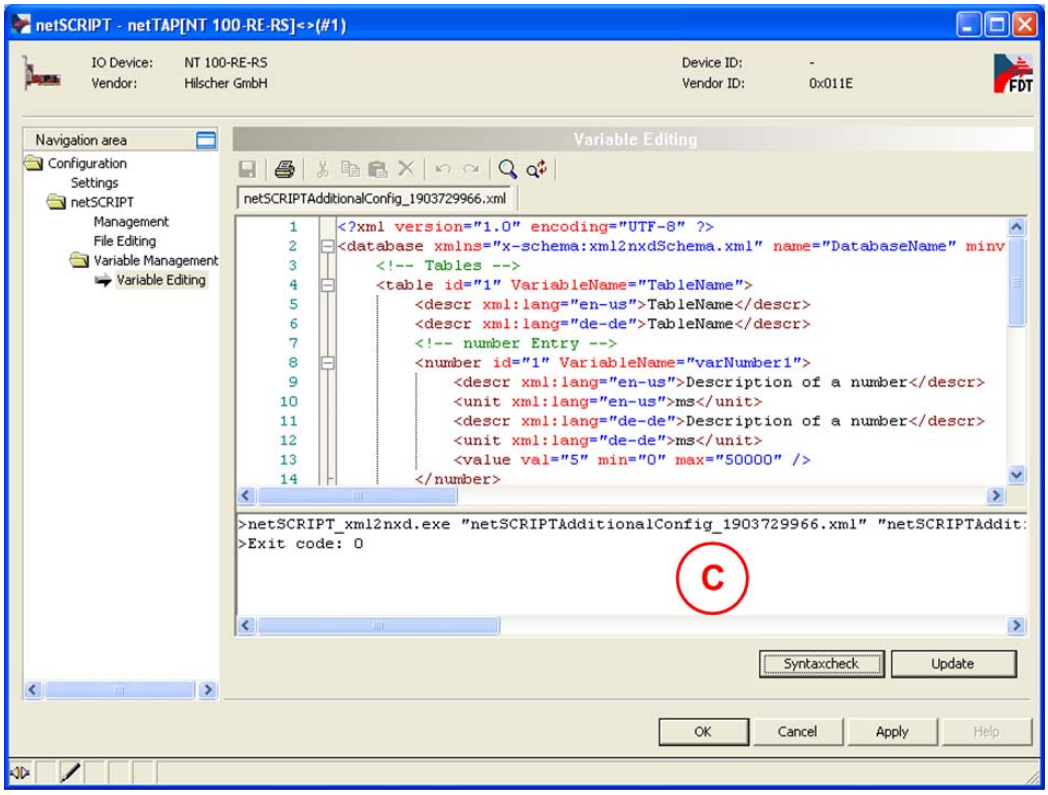


Figure 8: Definition Configurable Variables compiled

In the picture above, in window pane Ⓒ the result of the compilation is displayed. If the last entry in this window pane is not "**>Exit code: 0**", then the previous lines give information, where the error is to be searched in the variable definition.

3.3.2 Structure of the XML File for Configurable Variables

For one netTAP, device only one variable list can exist. The variable list is built up similar to HTML. Each start tag "**<aaa>**" has an end tag "**</aaa>**" respectively a comment „**<!--**“ start and "**-->**" end.

The XML file for the configurable netSCRIPT variable has basically 3 areas.

1. A file header.
2. The file body with the variables.
3. The file end.

It is possible to have several presentation languages (however, must not be defined) for a variable. These are independent from the netSCRIPT variable to which the value is assigned.

In the following three paragraphs, the details are described.

3.3.2.1 The File Header

In the file header, it is not permitted to change line 1...4.

In lines 5 and 6, it is fixed which language (Installation language of SYCON) the variable naming should be displayed. The languages German and English are currently possible.

Line-Nr.	xml-Code
1	`<?xml version="1.0" encoding="UTF-8" ?>`
2	`<database xmlns="x-schema:xml2nxdSchema.xml" name="DatabaseName" min-version="">`
3	`<!-- Tables -->`
4	`<table id="1" VariableName="TableName" >`
5	`<descr xml:lang="en-us">TableName</descr>`
6	`<descr xml:lang="de-de">TableName</descr>`

Table 8: XML-Code – File Header

Explanations to the lines:

Line 1:

Interpreter version number and character set information. This line may not be changed.

Line 2:

Contains processing-internal information. This line may not be changed.

Line 3:

This is a comment line and marks the beginning of the definition descriptions.

Line4

Marks the beginning of the variable table with the **id=“1“**. For a table beginning with **<table id = "1">** a suitable end tag **</table>** always belongs to it. Between these two tags is the description of the variables of table „**id="1"**“.

After the variable **VariableName,** and between the quotes which follow, a table name can be specified which can be used in netSCRIPT to address this table.

In a variable description file, there can be more than one variable table. These tables differ in their "**id**" numbers and the table name **VariableName**.

Line 5 and 6

In these lines the languages are specified into which the variable descriptions (in dependence of the selected SYCON language) should be displayed. **xml:lang="en-us"** for English and **xml:lang="de-de"** for German. With **>TableName<,** a naming text can be given between both carets for the table for the respective language.

3.3.2.2 File Body

In the file body, all variables with their display and data variables are described.

The single variables are marked with „**<number id = "n">**”. On this occasion, "**n**" defines the number of the position in which the variable should be shown later in the variable list. Hence, this number must be unique in the whole file.

In the following instructions, the possible single variables are listed and explained.

3.3.2.3 Numeric Variables

With a numerical variable it is not distinguished between an integer number and a decimal number.

Line-Nr.	xml-Code; Numeric Variable
7	`<!-- number Entry -->`
8	`<number id="1" VariableName="varNumber1">`
9	`<descr xml:lang="en-us">Description of a number</descr>`
10	`<unit xml:lang="en-us">ms</unit>`
11	`<descr xml:lang="de-de">Description of a number</descr>`
12	`<unit xml:lang="de-de">ms</unit>`
13	`<value val="5" min="0" max="50000" />`
14	`</number>`

Table 9: XML-Code - Numeric Variable

Explanation of the Line:

Line 7

Comment of the beginning of the definition.

Line 8

Naming of the variable type "<number>" and the position „**id = "1"**“ in the display list. Every position number may be assigned for the table only once!

Behind **`VariableName =`** is the variable name within quotation marks and is the name for the access within netSCRIPT.

Line 9

In this line, the describing naming of variable "**`Description of a number`**" is given for the display language **`"en-us"`**. The name is user defined.

Line 10

In this line, a unit "**`ms`**" is given for the display language "**`en-us`**".

Line 11

Here, according to the line number 9 the naming of the variables occurs for the German language **`"de-de"`**.

Line 12

Here, accordingly to line 10, the output of the unit is fixed into language German.

Line 13

In this line, the value range with a default value „**`val="5"`**” is set, and a minimum value „**`min="0"`**“ and a maximum value „**`max="50000"`**“ is defined.

Line 14

Contains the end tag "**`</number>`**" of the definition of the numerical variables.

3.3.2.4 Bool Variables

Example, the definition of a Boolean variable.

Line-Nr.	xml-Code; Bool-Variable
15	< <!-- bool Entry -->
16	<bool id="2" VariableName="varBool1">
17	<descr xml:lang="en-us">Description of a bool</descr>
18	<false xml:lang="en-us">off</false>
19	<true xml:lang="en-us">on</true>
20	<descr xml:lang="de-de">Beschreibung einer bool Variablen</descr>
21	<false xml:lang="de-de">aus</false>
22	<true xml:lang="de-de">ein</true>
23	<value val="1" min="0" max="1" />
24	</bool>

Table 10: XML-Code - Bool-Variable

Explanations to the lines:

Line 15

Comment at the beginning of the variable definition.

Line 16

Naming of the variable type "**<Bool>**" and the position „**id = "2"**“ in the display list. Every position number may be assigned for the table only once!

Behind **VariableName =** is the variable name within quotation marks and is the name for the access within netSCRIPT.

Line 17

In this line, the describing name of variable "**Description of a bool**" is given to the display language "**en-us**". The name is user defined.

Line 18

In this line, it is defined how the state "**false**" should be displayed in language "**en-us**". Here "**off**".

Line 19

In this line, it is defined how the state "**true**" should be displayed into language "**en-us**". Here "**on**".

Line 20

In this line, the describing name of the variables „**Beschreibung einer bool Variablen**“ is given for the display language "**de-de**". The name is user defined.

Line 21

In this line, it is defined how the state “false” in the language "**de-de**" should be displayed. Here "**aus**".

Line 22

In this line, it is defined how the state "**true**" should be displayed in language "**de-de**". Here „**ein**".

Line 23

In this line, with „**value val = "1"**“, the default value of the variable is given. With „**min = "0" max. = "1"**“, the possible minimum value or maximum value is defined. For a bool variable this is always 0/1.

Line 24

Contains the end tag "**</bool>**" of the definition of the bool variables includes.

3.3.2.5 String

A string value can have a maximum length of 64 bytes. It should be defined as follows:

Linen-Nr.	xml-Code;String-Variable
25	`<!-- string Entry -->`
26	`<string id="6" VariableName="varString1">`
27	`  <descr xml:lang="en-us">Description of a string</descr`
28	`  <descr xml:lang="de-de">Beschreibung der String-Variablen</descr`
29	`  <value val="Some string with max 64 length" min="0" max="64" />`
30	`</string>`

Table 11: XML-Code - String Variable

Explanation to the lines:

Line 25

Comment at the beginning of the definition of the string.

Line 26

Start tag of the definition with the line position specification **„id=“6“**“ in the variable list.

Behind **VariableName =** is the variable name within quotation marks and is the name for the access within netSCRIPT.

Line 27

Start tag **<descr xml:lang="en-us"**> of the description text for the string at the configurable variable list here for the language **"en-us".** Following the start tag are the indication text and then the end tag **</descr>** of the description.

Line 28

Start tag **<descr xml:lang="de-de"**> of the description text for the string at the operable variable list here for the language **"de-de".** Following of the start tag is the indication text, followed by the end tag **</descr>** of the description.

Line 29

The value tag with the text between the quotation marks after **val="...“** of the variable with the minimum length **min="0"** and maximum length of **max="64"** characters.

Line 30

End tag of the string variable.

3.3.2.6 File End

Line-Nr.	xml-Code; File End
31	`</table>`
32	`</database>`

Table 12: XML-Code - File End

Explanation to the Lines:

Line 31

End tag of the table definition.

Line 32

End tag of the file definition.

3.3.2.7 Delete of a Table with Variable Definitions

Should a table with variable definitions be deleted, the whole XML file may not be deleted. Only the part between the tag

```
<database xmlns="x-schema:xml2nxdSchema.xml"
name="DatabaseName" minversion="">
```

and the tag

```
</database>
```

is allowed to be deleted.

3.3.2.8 Call of the Variables within netSCRIPT

The variables described in this section are automatically transferred with the download in netTAP and are callable there as follows:

1. **`VAR[“TableName“][“VariableName“]`**
2. If the table name and/or the variable name are only numerical, then the following method can be used:
   `VAR[1][2]`
   "1" is used for the table number and "2" for the variable number.
3. If a name is alphameric, the following method can be used:

   `VAR.TableName. VariableName`

3.3.3 Display of the Configurable netSCRIPT Variables

According to the definition of the configurable variables and their compilation and update in Figure 7: Configurable Variable Definition in section 3.3.1, the Navigation area (sub window Ⓐ) under the folder “Variable Management” gets a new line with the defined table name (see ①).

Here you can display the defined variable table ① in window pane Ⓑ and fill in values.

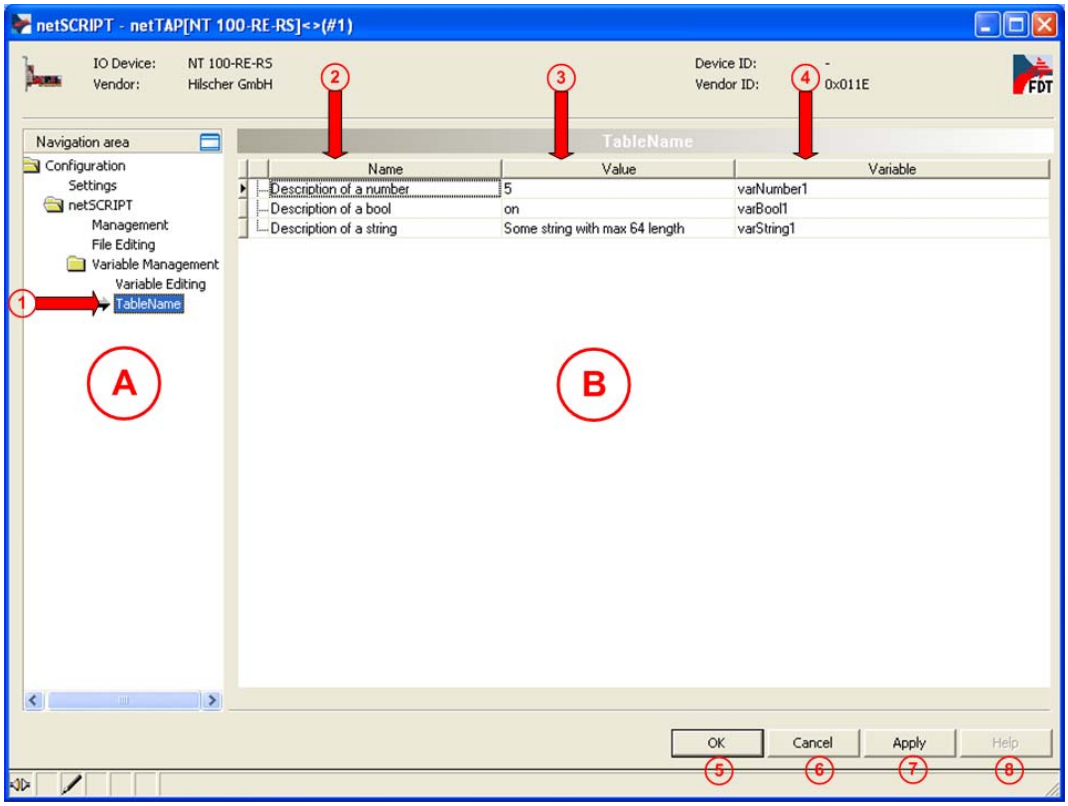


Figure 9: Configurable Variable Display

In the window pane Ⓑ you see the column:

② The variable name in the actual SYCON language,

③ The value set for the variables. This can be changed here after the selection with the curser,

④ The name of the variables in netSCRIPT.

With the button:

⑤ With this button the entries of this window are saved. Afterwards the window is closed.

⑥ With this button you exit the window without saving.

⑦ Save the actual variable values and return to the calling window.

⑧ With this button you reach the help topics pertaining to this window.

4 The netSCRIPT Language

The script language netSCRIPT is based on the script language Lua.

For the application within Hilscher communication devices and their special demands on the data transmission in networks and automation technology, the language was extended on a functional level.

The program written in SYCON.net is transferred to the target device with the download of the entire configuration. Usually, there are no real-time processing demands for the functions processed by the script program compared to the network protocol on higher level. Therefore, netSCRIPT always has a relatively low priority within the priority distribution in the pre-emptive operating system. Thus, interruptions of the running script program by the operating system may occur. Both the interruption and the return to program execution occur automatically and do not have any influence on the functional execution of the script program. Therefore, no special precautions are necessary within the program. As a general statement, please note that due to its low priority netSCRIPT cannot have any influence on the real-time properties of the other processes.

The firmware of the target device makes use the program and cyclically calls it within the given time frame, beginning with the first line of code and ending with the last one. The program is interpreted line by line at runtime and not executed as a compiled code like in several other programming languages. For this purpose, at Lua the source code is translated into a compact code for a virtual machine.

Usually, the provided functions do not have any suspending effect on the calling program code; instead, they always return to the calling program code immediately after execution. However, functions with a suspending behavior, for instance such functions which access special functions of the operation system, are specially marked with a corresponding note within the description. Please keep in mind that the time frame of cyclic calling cannot be maintained as soon as the script program does not return from a call within the cycle time.

Due to the principle of cyclic calling, it is a proven approach to design a script program via a so-called ‘status machine’ or ‘chain of steps’ and to process only small selected parts of the program within one cycle. A global running variable is used as an indicator which part of the program is to be processed just within the next cycle. A test of the contents of this variable, for instance, by an ‘if’ statement at the entry point into the script code is put in front of the chain of steps in order to branch into the various parts of the program using the values.

Of course, it is permitted to design extensively time consuming loops, even if they lead to violation of the cyclic time frame. As an extreme case, it is even possible that the program will not return from its call at all. This does not affect the system anyway, nor it will cause an error message or an abort of script processing. Instead of this, missed cycles are counted and caught up later on. For this purpose, the script program is called as often outside of the cyclic time frame without consuming cycle time, as it is necessary according to the number of missed cycles.

When using a script debugger, the script program loses its real time properties, and the cyclic execution is interrupted. The debugger is a so called soft debugger, i.e. when stopping the program it will neither stop the operating system, nor the CPU or the periphery. Only the script program and its

execution can be analyzed and controlled. During debugging, the rest of the system (such as the higher level communication network) continues working unaffectedly.

Note: The script program is cyclically executed with the cycle time to be adjusted by SYCON.net. Adjustment of the cycle time is described in detail within section 3.2.1 of this document.

Important: Within netSCRIPT, redefinition of all initially preset variables, functions and tables within the current script is possible. Therefore, the choice of names, especially variable and function names, has to be done with special care.

A line of code is finished by pressing the **ENTER** key. It is possible to put a semicolon in front of the **ENTER.** The semicolon, however, may also be used for the separation of two statements.

4.1 Syntax and Keywords

Note: netSCRIPT is a case-sensitive language: `and` is a reserved word, but `And` and `AND` are two different, valid names, therefore it is recommended always to write all variable names in minor letters.

4.1.1 Comments

A *comment* starts with a double hyphen (`--`) anywhere outside a string. `--`

Everything following the two minus signs is not read by the interpreter.

Comments extending over more than a single line are bound by two rectangular parentheses: `--[[` for the beginning and `]]` for the end.

Note: Recommendation for the editor:

If a part of the code is not to be executed for a limited time when testing the program, the following approach is recommended:

```
1  CODEBLOCK1        Code will be executed.
2  --[[              Start of a long comment.
3  CODEBLOCK2        Code will not be executed.
4  --]]              End of a long comment.
5  CODEBLOCK 3       Code will be executed.
```

If a blank is added into the line of the beginning of the comment between the two minus signs „`--` „ and the two open parentheses, both lines 2 and 4 are single comment lines on their own and **`CODEBLOCK2`** will be executed.

4.1.2 Keywords

Keywords are words which cannot be used for variable or functions names, because they will be interpreted as instruction.

These are the words:

```
and      break   do    else      elseif
end      false   for   function  if
in       local   nil   not       or
repeat   return  then  true      until     while
```

4.2 Variables

netSCRIPT is a dynamically typed language. This means that variables do not have types; only values do.

4.2.1 Variable Names

• The name of a variable has to begin with either a letter or an underscore.

• It is not allowed to use other characters than numbers, letters or underscores in the name.

• Use of large and small initial letters is distinguished in the name.

4.2.2 Assignments

An assignment always looks like:

```
Name = Expression
```

Values will be assigned to variables with the equal sign „**=**“. At the left side is the name of the variable and at the right side is the value to be assigned.

It is also possible to define a list of variables on the left side with a list of expressions on the right side according to the following:

```
Name1, Name2, Name3 =

Expression 1, Expression2, Expression3
```

Because global variables keep their values up to the next cycle program start, for the first start the following assignment is necessary:

```
var1 = var1 OR 1
```

In that case the value which the variable has had at the end of the last program cycle is assigned to variable **var1**. However, at the first cycle the value ***1*** is assigned to variable **var1.** If the value at the end of the previous cycle was **nil** then also the value ***1*** is assigned to variable **var1.**

For variables which have been defined outside of the cyclic program start, (for instance, if they have been predefined as shown above) assignment is not necessary.

4.2.3 Scope of Variables

The area of validity of variables can be global or local.

Global variables are valid for the whole script.

They keep their value at the end of processing the script up to the next cyclic start of the script.

Global variables are defined in the table _G.

Local variables are only valid in that block where they are defined. They do not exist outside this block behind the command „**end**“.

A local variable will be defined with the prefix „**local**„ in front of the name of the variable.

Access to local variables is more efficient than for global variables. Choosing a local variable is recommended for all variables which are required locally, but often.

You can also temporarily store global variables, which for instance are called n times, within a local variable. For instance:

```
local getDataChar = xc_uart.getDataChar.
```

Values of local variables can be written back to the global variable even in case of equal names, if you set the prefix of the environment table (usually „**_G.** “) in front of the name of the global variable. Then the variable within table _G is accessed. Also see section 4.3 Example for the validity of a variable, here for example for the variable „a“.

Code	Output:
`x = 5`	
`print(x)`	5
`do`	
`  local x = x`	
`  print x`	5
`  local a=1`	
`  print(a)`	1
`  do`	
`    local a=2`	
`    print(a)`	2
`  end`	
`  print(a)`	1
`  x = x + 3`	
`  print (x)`	8
`  _G.x = x`	
`end`	
`print(x)`	8

Without the assignment „**_G.x = x**“ at the end of the **do** loop in table 13 „**x**“ would have the value „5“, after execution of the „**end**“ statement.

4.2.4 Types

The following data types are available:

Type	Meaning / Usage
nil	Nothing, blank, not available.
numbers	Whole and broken figures, exponential figures.
String	Literal strings can be delimited by matching single or double quotes as follows:: “Text“ or ’Text’. In this context, the word „Text“ means the contents of the string. Long strings over more than one row start with --[[and end with --]]
Conditions	, true / false, yes/no, 1/0.
Functions	Are special cases of a block.
Table	The table which saves any value under an index (number, name, other object). e.g. t[1]=13 oder t[„surname“]=“Einstein“

Table 13: Overview Types

The type of an expression v can be interrogated with the function „type (v)“ (see section 5.1.21 on page 53).

4.2.4.1 Nil

The value (and the type) ***nil*** reserves space for currently not present values or unused variables. Additionally, in logical expressions ***nil*** is usually equivalent to **false**.

Accessing a variable to which no value has yet been assigned will cause the result to have the value ***nil***. In this case, no error message will be issued.

If you assign the value ***nil*** to a certain index of a table (***t[x]=nil***), the table entry will be erased if any has already existed.

If you assign the value ***nil*** to a global variable, this variable will no longer exist, i.e. the storage area occupied by the variable within the memory will be released by the garbage collector. To local variables, the value ***nil*** will be assigned.

```
var1 = nil
```

4.2.4.2 Numbers

Numbers are represented as 64 bit floating point value. Double Float values are represented with a 52 Bit mantissa.

Numbers may be entered as integer values, with or without sign, optionally with a decimal fraction, optionally with an exponential part, or as a whole number in hexadecimal representation.

Integers, fractions and negative numbers may be assigned to variables. As decimal sign the „.“ (Point) is used! Negative values are preceded by the minus sign „-„ immediately preceding the numeric part.

The following assignments are equivalent.

Assignment 1	Assignment 2
`_Zahl1 = 74`	`_Zahl1 = 0x4A`
`Zahl_2 = 105`	`Zahl_2 = 1.05e2`
`einHalbMalWurzel2 = 0.707107`	`einHalbMalWurzel2 = 0.5^0.5`

Table 14: Assingments

If a string contains only digits, it will be interpreted as a number by net-SCRIPT. But there is one exception: this is not valid within comparisons!

4.2.4.3 Strings

Strings are delimited by the characters `"Text"` or `'Text'` in the line. In this context, the word `"Text"` displays the contents of the string.

Strings may contain arbitrary sequences of bytes (also 0x0).

Strings extending over several lines will be enclosed with **[[** at the beginning and with **]]** at the end.

The following assignments are equivalent.

```
townname = " Hattersheim am Main "
street = ' Rheinstrasse 15'
varTyp1 = "function"
thestory = [[The history began sometime
                        and also ends once]]
```

4.2.4.4 Boolean Values

Boolean Values are a variable type which only can have only two states:

`true` and **`false`**.

Assignment:

```
sunshine = true
night = false
bright = true
```

The assignment is done using the reserved words **`true`** and **`false`**.

The value ***`nil`*** of a variable will be interpreted as **`false`**. All values besides ***`nil`*** and **`false`** will usually be interpreted as **`true`**.

4.2.4.5 Functions

Functions are encapsulated blocks of code which can be invoked more than once. Each call generates an own local parameter/variable block.

Functions can be recognized by **`()`**, for instance **`function()`**.

For the definition and the call of a function, please refer to *section* Functions at page 43 of this document.

4.2.5 Tables

In Lua, tables are hash tables, which map arbitrary indices (keys) to values. Both the indices and the mapped values may be arbitrary Lua objects such as numbers, strings or even tables themselves. The type is **`table`**.

Tables are defined at the allocation within curly brackets.

Definition:

```
myTable = {}
```

The assign statement to a table may look like:

```
myTable = { 1, 4, "Willi", true, function()
            dosomething end}
myTable2 = { X= 255 , Y =10, speed =88}
```

These values can be read as follows:

```
meinTable[1] –> 1
meinTable[2] –> 4
meinTable[3] –> "Willi
meinTable[4] –> true
meinTable[5] –> function()dosomething end
meinTable2.X –> 255
meinTable2.Y –> 10
meinTable2.speed –> 88
```

4.2.6 Garbage Collector

For the release of previously reserved memory areas, netSCRIPT provides a garbage collector. The garbage collector releases physical memory which has been reserved and occupied by variables no longer used. There is no reference any longer to the memory areas to be released by the garbage collector.

The functionality of the garbage collector can explicitly be invoked within the script if desired. The function to be invoked for this purpose is described in section collectgarbage (opt) on page 45 of this document.

4.3 Global System Variables

4.3.1 _G

_G is a global variable (table). All base functions are deposited within _G by Lua as are global variables with their corresponding memory pointers.

If another environment should be used for the script, another environment can be set with the function „setfenv (f, table)“ using the statement

```
FOR k,v IN pairs(_G) DO print(k,v) END
```

4.3.2 _VERSION

This is a global variable containing the version signification of Lua on which netSCRIPT is based.

4.3.3 _NETSCRIPT_VERSION

This is a global variable containing the version number of the netScript environment.

4.3.4 CYCLIC_FUNCTION

After loading the script, the compiled code of the script is bound to this variable. At every cyclic start, this variable is evaluated and the code engaged in it is executed. This function is executed beginning from the second cyclic start just by assigning a Lua function to this variable.

```
CYCLIC_FUNCTION = Funktionsname
```

Note:

If a function is assigned to the variable **CYCLIC_FUNCTION**, this function will be executed in the debugger on level 0. Without this assignment, however, the main program will be executed on level 0 within the debugger.

4.4 Operations

The following operations are integrated within netSCRIPT: The listing shows the operations with ascending priority:

```
or
and
<     >     <=    >=    ~=    ==
..
+     -
*     /     %
not   #     - (unary)
^
```

4.4.1 Mathematic Operations

Operator	Description	Example 1	Example 2
+	Serves for the addition of values.	c = a + b	1 + 3 = 4
-	Serves for the subtraction of values.	c = a - b	5 - 3 = 2
*	Serves for the multiplication of values.	c = a * b	2 * 3 = 6
/	Serves for the division of values.	c = a / b	8 / 2 = 4
^	Serves for the determination of the power of values (power of a with exponent b).	c = a ^ b	2 ^ 3 = 8 = (2 * 2 * 2)
-	Serves for the negation of values, i.e. plus will become minus and vice versa.	c = -a	if a == 2, then c == -2
%	Mathematical function that indicates the rest of a division of interger numbers		7 mod 2 = 1 because 7:2 = 3, rest 1

Table 15: Overview of mathematical operations

There is a specific feature in netScript. Within an expression such as `2 + "6"` the second value will not be interpreted as a text, but as a number and 8 will be displayed as result. However, there may be no characters between the single or double hyphens as this would inhibit the evaluation. For instance, the interpreter would neither understand nor numerically evaluate an expression such as `2 + "6s"`.

4.4.2 Logic Operations

There are three operands defined:

AND, OR and **NOT**

Operator	Description
and	Both condition 1 and condition 2 must be fulfilled: Result = Bedingung1 and Bedingung2.
or	Condition 1 or condition 2 must be fulfilled: Result = condition 1 or condition 2.
not	This is the negation of the former state

Table 16: Overview of logical operations

Examples of logical operations:

Variable definitions:

```
sunshine = true
dark = false
lightoff = true
```

Operator	Example	Result
and	**sunshine and dark**	***false***
or	**sunshine or lightof**	***true***
not	**NOT sunshine**	***false***

Table 17: Examples of Logical Operations

For other logical operations on bit level see section *6.1 Bit-Operation*.

4.4.3 Relational Operators

It is only permitted to compare identical types. If the types are different, then the result is ***false***.

The following relational operators are defined:

Operator	Description	Example	Result
==	Left side is equal the right side?	"Willi" == "willi"	false
~=	Left side is not equal the right side?	"Willi" ~= "willi"	true
<	Left side is smaller than the right side?	2 < 3	true
>	Left side is greater than the right side?	2 > 3	false
<=	Left side is less or equal the right side?	2 <= 3	true
>=	Left side is greater or equal the right side?	2 >= 3	false

Table 18: Overview of the Relation Operators

If the values at the left and the right side are not of the same type, an error will occur.

4.4.4 Control Structures

For, while, repeat.

4.4.4.1 FOR-Statement

Example:

```
1.  FOR var1 = start, stop, step DO
       Block
    END
```

The `Block` represents a list of instructions. Initially, the value **`start`** will be assigned to the variable **`var1`**.

During every cycle (between **`DO`** and **`END`**) the variable **`var1`** will be increased by the value of **`step`**. This will happen until **`var1`** will be equal to the value of **`stop`**. Finally, the next statement behind **`END`** will be executed then.

```
2.  FOR var1 IN table DO
      Block
   END
```

The `Block` represents a list of instructions. The loop (between **`DO`** and **`END`**) will be repeated as long as the value of the variable **`var1`** exists in the variable **`table`**. If the value of **`var1`** is not contained within the **`table`**, the statement behind **`END`** will be executed.

4.4.4.2 WHILE loop

```
WHILE CONDITION DO
  Block
END
```

The `Block` represents a list of instructions. As long as the `CONDITION` is evaluated as *true*, the statements between **`DO`** and **`END`** will be executed.

4.4.4.3 REPEAT loop

```
REPEAT
  BLOCK
UNTIL CONDITION
```

The `Block` represents a list of instructions. As long as `CONDITION` is evaluated as *true,* the statements contained in the `BLOCK` will be executed. If the `CONDITION` will become *false,* the statement following **`UNTIL`** will be executed.

Contrary to the for-loop, the `BLOCK` will at least be executed once.

4.4.4.4 BREAK

The **BREAK** statement makes enables leaving a loop independent of the step-in condition or the step-out condition.

The script will directly be continued following the **END** respectively **UNTIL** statement of the loop.

4.4.5 Branch Statement if ...then

```
blockA
IF CONDITION 1 THEN block1
 [ELSEIF CONDITION 2 THEN block2]
  [else block3]
END
blockB
```

If CONDITION1 is ***true***, the block of statements block1 following the first **THEN** will be executed.

If CONDITION1 is ***false,*** it will be checked whether CONDITION2 following the **ELSEIF** statement is ***true.*** In that case, the block of statements block2 will be executed. The line **{ELSEIF** CONDITION **THEN** block**}** can be repeated arbitrarily often between **IF** and **END** if necessary.

If neither CONDITION1 nor CONDITION2 is ***true***, the block of statements block3 will be executed. If the block of statements block3 does not exist, the program will be continued with blockB following the **END** statement.

After the **IF** statement, processing will be continued with blockB, independently from execution of the blocks block1, block2, or block3.

4.5 Functions

Functions are encapsulated blocks of instructions which provide their own environment for variables, which will be initialized by every call. The communication with other program statements will be accomplished by the use of transfer parameters.

A function needs to be defined before it can be called.

The variable environment of a function can be defined by the function „`setfenv`“ as described in chapter 5.1.17 setfenv (f, table) of this document.

4.5.1 Definition of a Function

```
FUNCTION name (var1, var2, var3)
  Block1
 [RETURN var2, var3]
END
```

Statement	Description
function	Starts a definition of a function.
name	The name of the function by which it may be called from an other statement.
(var1, var2, var_3)	Transfer parameter list. The parameters will each be divided by a comma. The parameters are defined locally and not known within the other program parts.
Block1	Processing block. All variables defined within this block are local variables, all others are global ones.
return var2, var_i	The keyword „return“ will cause returning from processing a function. Following a „return“ statement on or more comma separated values may be specified, whose values will be returned to the statement which originally has called the function.
END	Marks the end of the definition of the function.

Table 19: netSCRIPT - Function Definition

4.5.2 Function Calls

As an example, the function defined in chapter 4.5.1 will be called.

```
varD, varE = name(varA, varB, varC)
```

varD	The global variable **varD** will accept the value of the local return variable **var2** from chapter 4.5.1.
varE	The global variable **varE** will accept the value of the local return varable $\mathbf{var_3}$.
name	Name of the function to be called under which it has been defined.
(varA, varB, varC)	Between the (round) parentheses there are arbitrary expressions whose values will be transferred to the function. The assignments is as follows: **varA → var1** **varB → var2** **varC → var3**

Table 20: netSCRIPT - Function Call

Practical example of the definition of a function and its call.

Code	Result: print (x).

`function add(a,b)`
`  x=a+b`
`return x`
`end`
 	
`x = add(1,4)`
`print(x)`
`print(add(2,5))`
	
	5
	7

Table 21: netSCRIPT - Example of Function Definition and Call

5 Functions Library

Not all functions of the OS-Library defined in the Lua standard are usable, because they are not sensible in this environment.

5.1 Base Functions

Following base functions are integrated:

5.1.1 assert (v [, message])

Conditional error message transmission

This function issues an error if the value of its argument **v** is false (i.e., **nil** or **false**); otherwise, it returns all of its arguments. `message` is an error message; when absent, it defaults to `"assertion failed!"`

5.1.2 collectgarbage (opt)

This function provides an interface to the garbage collector. According to its first argument, (opt), different functions will be performed:

opt	Function
“stop”	Stops the garbage collector.
“restart”	Restarts the garbage collector.
“collect”	Performs a full garbage-collection cycle
"count"	Returns the total amount of memory in use by netSCRIPT (specified in kBytes).

Table 22: netSCRIPT – Garbage Collector

Example:

```
a = (collectgarbage ("count"))
```

returns the total amount of memory in use by netSCRIPT (in kBytes).

5.1.3 error (message [, level])

Absolute error message

(conditional error message see section `assert (v [, message]))`

This function aborts the execution of the script issuing an error message. Exception: If the call is located within a function having been called with „**pcall**“, execution will be continued after the invocation of „**pcall**“.

The last protected function call is finished (with „**pcall**“, see section pcall (f, arg1, ···)) and transfers the contents of variables „**message**“ and „**level**“ to the function „**pcall**“

A line number will precede the error message. The level parameter indicates which line is mentioned. The value 1 indicates the current position within the script. Within the execution of a function call, if a value of 2 is indicated for the level parameter, the position from where this function has been invoked is indicated. Within iterated function calls, even larger values are possible.

5.1.4 getfenv ([f])

This functions returns the environment (table) currently used by the function for storage of the variables of the function "**f**". Usually, this is the table **_G**. Also see section setfenv (f, table) on page 50 of this document.

f" can be a netSCRIPT function or a number of the stack level representing this function.

If the given function is not a netSCRIPT function, or if „**f**" is 0, **getfenv** returns the table of global environment (_G). The default value for "**f**" is 1

5.1.5 getmetatable (object)

This function returns the metatable of the object specified within parameter "**object".**

- If "**object"** does not have a metatable, this function will return "**nil**".
- Otherwise, if the object's metatable has a "**__metatable**" field, it will return the metatable of the given object.

5.1.6 ipairs (t)

This function returns an iterator function for the use in **for** loops, for instance. It returns the value pairs of a table of all entries with an integer key until the first alpha key or missing entry of the index row.

Code	Result: print (k,v).

`t = {100,200,300,400, x="22"}`
`t[10]=42`
`for k, v in ipairs (t) do`
`print (k,v)`
`end`
	`1 100`
	`2 200`
	`3 300`
	`4 400`

5.1.7 load (func [, blockname])

This function loads a block using function „**func**" until the value „**nil**" is returned. The respective result of „**func**" is again stored within a function.

Each call to „**func**" must return a string that is concatenated with previous results. Returning an empty string or the value „**nil**" indicates the end of the block.

If no errors are discovered during compilation of the block, the code will be stored within a function, otherwise the value „**nil**" will be returned and an according error message will be issued. „**blockname**" is used in this context for error messages and for debug information.

Code	Result

`n=0`
`stuecke = {"pri", "nt(", "42", ")"}`
`function gettext()`
`  n=n+1`
`  return stuecke[n]`
`end`
`a = (load(gettext, "test"))`
`pcall(a)`
	
	42

In the example above, as an end result, the statement „`print(42)`“ will be executed.

5.1.8 loadstring (string [, blockname])

Using this function, the contents of „**string**“ are transferred into a function call.

In order to load and start a string, the following syntax has to be used:

```
assert(loadstring(s))()
```

For indicating probable errors at run time, it is beneficial to provide a **blockname**, which will be output along with the error.

If absent, **blockname** defaults to the given string.

An example for building a function and the subsequent function call with parameter passing.

Code	Result

`fn = loadstring("return function (x)`
`print(x) end", "testprog")`
`a = fn()`
`a(50)`
	
	50

5.1.9 next (table, index)

This function returns the key and the value of the next entry of the table (incremented by 1 compared to the current index of the table) from the memory for this table.

table	Table name
index	Index of the table, it is not necessary that this is numeric.

Code	Result
`t = {100,200,300,400,500}` `print (next (t,3))`	 **4 400**

5.1.10 pairs (t)

This function returns an iterator function for **for** loops returning the value pairs of a table one by one:

Code	Result: print (k,v).

`t = {100,200,300,x = "nix", 500}`
`for k, v in pairs (t) do`
`print (k,v)`
`end`
	`1 100`
	`2 200`
	`3 300`
	`4 500`
	`x nix`

5.1.11 pcall (f, arg1, ···)

This function calls function **f** with the given parameters “PAR1, .”

The call should look like

`OK, x1, x2, x3 = pcall (f, PAR1, PAR2, ....)`

If the execution of function “**f**” completes without any errors, the value „true“ is assigned to the return parameter **`OK`** and the following parameters **`x1, x2, x3`** will contain the returned values of function “**f**”.

If an error occurs during the execution of function “**f**” the return parameter **`OK`** will contain the value **false** and parameter **`x1`** will contain such information as:

- Name of script
- Line number at which the error occurred
- A text describing the error

5.1.12 print (par1, par2, ···)

This function is a standard function of Lua which is designed as a standard output function but will be totally ineffective within a netSCRIPT enabled device due to the missing standard output facility.

In this context, the function is only mentioned to enable reproduction of the examples of the basic function within a standard Lua environment (on the PC).

The function puts out each of the listed parameters on the standard output channel. The function is not suited for formatted output.

For formatted output, the function **string.format()** has to be applied prior to **print()**.

5.1.13 rawequal (par1, par2)

This function checks `par1` for equality with `par2`without invoking any metamethod. The return value is **`true`** or **`false`**.

The comparison is done without invoking any meta functions.

5.1.14 rawget (table, index)

This function delivers the table entry **table[index]** without execution of any meta functions.

In the following example, a default value is set for the situation of an absent table entry to be returned at a normal table request. Function „rawget“ will return the actual entry.

Code	Result
t={} function default(tab, key) return 42 end mt={__index=default} setmetatable(t, mt) t.a=1 t.c=3 print(t.a, t.b, t.c) print(rawget(t, "a"), rawget(t, "b"), rawget(t, "c"))	1 42 3 1 nil 3

5.1.15 rawset (table, index, value)

This function executes the assignment **table[index]=value** without application of any meta functions.

In the following example, a normal insertion into the table is performed using the „set“ function, and read access to table „**t**“ is performed using function **„default“**

Code	Result

t={}
function default(tab, key) –[[Default value if index not available]]
  return 42
end
function set(table, key, value) –[[new entry in the table]]
  print(table, key, value)
  rawset(table, key, value)
end
 	
mt={__index=default, __newindex=set} --
setmetatable(t, mt)
t.a=1
t.c=3
rawset(t, "d", 4)
print("normal access: "
  , t.a, t.b, t.c, t.d)
print("with rawget: ", rawget(t, "a"), rawget(t, "b"), rawget(t, "c"), rawget(t, "d"))
	table: 01004A00 a
	table: 01004A00 c
	
	normal access:
	1 42 3 4
	with rawget:
	1 nil 3 4

5.1.16 select (index, par1, par2, par$_3$, ···)

If "**index"** is a number, the parameter of the index number and all other parameters are returned.

If "**index"** = „#", the number of entries will be returned.

Example:

Code	Result

`i = 2`
`a = select(i, "anton", "berta",`
`"cesar", "delta")`
`print ("Var1 =",a)`
`print (select(i, "anton",`
`"bater", "cesar",`
`"delta"))`
	`Var1 = berta`
	
	`bater cesar`
	`delta`

5.1.17 setfenv (f, table)

This function sets the environment to be used for the given function „**f**" to the table „**table**".

Instead of the also the level in which the function runs may be specified like:

„**f**" = n;

n=1: → the current level;

n= n+1 → the next level above the current level

n= 0 → indicates a change of the environment of the currently used level

Return values:

The function in which this function is called.

Note: This function may not be applied at level 0 as this would cause the loss of the references to all functions.

5.1.18 setmetatable (table, metatable)

Assigns a meta table „**metatable**" to a table „**table**". If „**metatable**" = „nil", an existing meta table will be removed from the table „**table**".

This function enables the definition of new operations for table values which would otherwise be forbidden. These functions need to be declared within the meta table accordingly.

The following functions may be redefined:

Function	Application for the Standard call
`__add`	Addition (+)
`__sub`	Subtraction (-)
`__mul`	Multiplication (*)
`__div`	Division (/)
`__pow`	Calculation of the power of the arguments (e)
`__unm`	The unary minus („-„) operation applied to a table.
`__concat`	Concatenation of two tables using.
`__eq`	Check of two tables for equality using the relation operator ==.
`__lt`	Comparison of two tables using the relation operator <.
`__le`	Comparison of two tables using the relation operator <=
`__index`	Call of a not existing table index.
`__newindex`	Insertion of a new entry (index) to a table.
`__call`	Access to any value within the table
`__tostring`	Application of the function **tostring** onto a table

Table 23: netSCRIPT - Function Replacements using Function `setmetatable`

Further information can be found under the topic Metatable Lua documentation available at http://www.lua.org/manual/5.1/ section 2.8.

The following example shows the generation and addition of new table entries:

Code	Result

`-- carrier table for vector functions`
`vector ={}`
 	
`-- constructor: generates a new vector object.`
`-- packs the arguments in a new`
`-- table and sets the meta table of this table.`
`-- The arguments have to be numeric.`
`function vector.new(...)`
`  local v = {...}`
`  setmetatable(v, vector.mt)`
`  return v`
`end`
 	
`-- generates a printable string from a`
`-- vector.`
`-- the function table.concat gives a string`
`-- separated by commas with the content of the vector.`
`function vector.toString(self)`
`  return ("(" .. table.concat(self, ",") .. ")")`
`end`
 	
`-- adds two vectors and gives the result`
`-- in a vector`
`function vector.add(v1, v2)`
`  local v = {}`
`  for i=1, math.min(#v1, #v2) do v[i]= v1[i]+v2[i] end`
`  setmetatable(v, vector.mt)`
`  return v`
`end`
 	
`-- meta table`
`-- the entry __add cause that the operator + applied to`
`-- two vector objects calls the add function`
`--`
`-- the entry __index cause that the defined functions`
`-- are directly callable to a vector object`
`-- v:toString()`
`vector.mt={__add=vector.add, __index=vector}`
 	
`-- Generates two vectors and adds them`
`vec1 = vector.new(1,2,3)`
`print("vec1 = ", vec1:toString())`
 	
`vec2 = vector.new(10, 10, 10)`
`print("vec2 = ", vec2:toString())`
 	
`vec3 = vec1 + vec2`
`print("VEC1 + VEC2 =", vec3:toString())`
	`vec1 =`
	`(1,2,3)`
	
	`vec2 =`
	`(10,10,10)`
	
	`VEC1 + VEC2`
	`= (11,12,13)`

5.1.19 tonumber (e [, base])

This function tries to convert its argument “**e**” to a number in a representation to the base „**base**“ if the parameter is a number or strings that can be evaluated to a number. Otherwise, it will return **nil**.

The optional parameter „**base**“ may range between 2 and 36 where the minor or major letter „A“ represents the number 10, whereas „Z“ represents the number 35.

The default value of base is 10.

Example:

Code	Result
`print(tonumber("55"))`	`55`
`print(tonumber(55))`	`55`
`print(tonumber("55a"))`	`Nil`
`print(tonumber(10101.1012e12))`	`1.01011012e+16`
`print(tonumber("100110", 2))`	`38`
`print(tonumber("100112", 2))`	`nil`
`print(tonumber(0xff))`	`255`
`print(tonumber("LUA",36))`	`28306`

Note: The maximum possible numerical value for hexadecimal numbers to be applied here is 0xffffffff. Larger values, like for instance 0x100000000 will be “rounded” to 0xffffffff without any error message being issued.

5.1.20 tostring (e)

Converts each data type into a string. Sometimes the function “string.format” may be suited better for this purpose.

Note: If this function is applied to tables or functions, for instance, the type and the address of the table or function will be returned.

5.1.21 type (v)

This function returns the type of parameter “**v**“. Possible results of this function are:

“nil”, "number", "string", "boolean", "table", "function", "thread", and "userdata".

5.1.22 unpack (list [, i [, j]])

This function returns the elements of table **list** including all numeric key values starting from key number **i** up to last key number **j**.

Example:

```
t={[1]="a", [2]="b", [4]="c"}
unpack(t) = „a", „b", nil, „c"
```

If **j** has been specified, the corresponding part of the list will be returned. If **j** has not been specified, only the values beginning from position **i** until the first index for which no according value has been stored, will be returned. Thus, in the last example only the values „a" und „b" would have been returned.

5.1.23 xpcall (f, err)

This function is quite similar to the function „pcall", but contrary to that function it allows the invocation of a special error handler using the „err" parameter.

„**xpcall**" calls function „f" in protected mode and uses „err" as error handler, which allows a reaction to the error message

Return values in case of success

1. „**true**"
2. All return parameters of function „**f**"

Return values in case of failure

1. "**false**"
2. The result of the evaluation of „**err**"

5.2 String Manipulation

Besides the library „string“ there is also the concatenation operator „**..**“ for the concatenation of strings. This allows the concatenation of two strings.

Code	Result
`A = 1; B = 3; C="a"` `print(A..B..C)`	`13a`

In the example above, it is mandatory that each variable to be concatenated must be formerly defined. This must be taken care of especially when something should be concatenated to an existing string, for instance, within the following statement:

„ **`STR = STR..string.char(ICHAR)`** „

The library „string“ contains further common functions for STRING manipulation. All functions concerning strings are contained within the table „**string**“.

If an index is set onto a string, take care of the first character really being at position one.

The library „string“ is based on a one byte character coding.

5.2.1 string.byte (s [, i [, j]])

This string manipulation function returns the internal numerical codes of the characters of string “**s**” beginning with position “**i**” up to position “**j**”.

The default value for „**i**“ is 1; the default value for „**j**“ is i.

Example:

Code	Result
`str = "abcdefghijklm"` `print(string.byte(str,3,7))`	`99 100 101` `102 103`

5.2.2 string.char (...)

This string manipulation function generates a string out of a list of byte values in the range of 0 up to 255. This is the reverse functionality of „string.byte“ described just above.

5.2.3 string.find (s, pattern [, init [, plain]])

This string manipulation function looks for the first match of “**pattern**” in the string “**s**” and returns the found range by the number of the start character and the number of the final character. If no match is found, „**nil**“ will be returned.

Init (numeric value) indicates the starting position from where to search.

If the pattern contains a string which also can be interpreted as a class of characters, but this should not be interpreted in such a way, the last parameter „**plain**“ has to be set to the value „**true**“. However, for this purpose the parameter „**init**“ is required.

5.2.3.1 Patterns

Patterns are sequences of pattern items. Commonly, a pattern comparison is successful if any part of the examined string matches the pattern.

The characters of the pattern are enclosed in anchors within substrings. These anchors can be represented by the characters „**$**“ (dollar sign) or „**^**“ (arrow up). Outside of substrings, however, these characters represent themselves.

The character „**^**“ at the beginning of the pattern causes the comparison to succeed only in case of a match between the pattern and the first part of the string to be examined. The same holds true for „**$**“ and the end of the string. The combination of both means the whole string must match the pattern.

Detection or exchange of patterns are relevant topics when performing string manipulations. Therefore, some special options are provided for pattern detection.

5.2.3.2 Captures

A sub-pattern (i.e. a partial pattern) can be marked by enclosure in parentheses for the definition of captures. A pattern comparison will then not only return the part of a string which is matched by the pattern itself, but also such parts, which are covered by the sub-patterns enclosed in brackets.

The sub-patterns are numbered according to the position of the opening bracket. For example in the pattern „(a*(.)%w(%s*))“:

The match with string „(a*(.)%w(%s*))“ receives number 1, the match with „.“ receives number 2 and the match with string „%s*“ receives number 3.

5.2.3.3 Character Classes

A *character class* (class of characters) is used to represent a specific set of characters. The following combinations allow the description of specific character classes:

- **.:** (a dot) represents all characters besides the characters ^$()%.[]*+-? which have a special meaning and which represent themselves.
- **%a:** represents all letters.
- **%c:** represents all control characters.
- **%d:** represents all digits.
- **%l:** represents all lowercase letters.
- **%p:** represents all punctuation characters.
- **%s:** represents all kinds of space characters.
- **%u:** represents all uppercase letters.
- **%w:** represents all alphanumeric characters.
- **%x:** represents all hexadecimal digits.
- **%z:** represents the character with representation 0.
- **%***: (where "***" is any non-alphanumeric character) represents the character *. This is the standard way to mark a character which is also used as control character within the script language as a character to be searched within the string. This is also valid for the character % (percent sign) itself.
- **[*set*]:** represents a group of characters in „***set*** ". A range of characters can be specified by the end sign „-„. All classes „%*" can be used within „***set***" as described above. All other characters represent themselves.

  Example:
  [*%w*_] or **[_*%w*]** represent all alphanumeric characters plus the underscore; **[*0-7*]** represents all numbers from 0 to 7 and **[*0-7%l%-*]** represents all numbers from 0 to 7, the minor letters and the sign „-„ (minus sign).
  Example:

Code	Result:
`s1 = "b1cad2aa bc"` `print(string.find(s1, "[2-6]"))`	`6  6`

- **[^*set*]:** represents the complement of ***set***, where ***set*** is interpreted as described above.
- For all classes represented by single letters (%a, %c, etc.), the corresponding uppercase letter represents the complement of the class. For instance, %S represents all non-space characters.
  E.g. **%W** represents all non-alphanumeric characters.

5.2.3.4 Pattern item

A *pattern item* can be:

- a single character class, which exactly matches a single character in the class;
- a single character class followed by „*****“ matches the longest possible sequence of 0,1 or multiple characters of the class.
- a single character class followed by „**+**“ matches the longest possible sequence of 1 or multiple characters of the class.
- Example:

Code	Result

`s1 = "bc1caaaad2aa bcade"`
`print(string.find(s1, "ca"))`
`print(string.find(s1, "ca*"))`
`print(string.find(s1, "ca+"))`
`print(string.find(s1, "a+"))`
	
	`4 5`
	`2 2`
	`4 8`
	`5 8`

- a single character class followed by '**-**', which also matches 0 or more than one repetitions of characters in the class.

Code	Result

`s1 = "bc1caaaad2aa bcade"`
`print(string.find(s1, "ca-",3))`
`print(string.find(s1, "a-",3))`
	
	`4 4`
	`3 2`

- a single character class followed by '?', which matches 0 or 1 occurrences of a character in the class;

Code	Result

`s1 = "bc1caaaad2aa bcade"`
`print(string.find(s1, "ca?"))`
`print(string.find(s1, "ca?",3))`
	
	`2 2`
	`4 5`

- %*n*, for *n* between 1 and 9; such item matches a substring equal to the *n*-th string having been captured (see below);
- %b*xy*, where *x* and *y* are two distinct characters; such item matches strings starting with *x* and ending with *y*, and where the *x* and *y* are *balanced*. This means that, if one reads the string from left to right, counting *+1* for an *x* and *-1* for a *y*, the ending *y* is the first *y* where the count reaches 0.

5.2.4 string.format (formatstring, ···)

This string manipulation function returns a formatted string consisting of a variable number of arguments according to the transformation description for formatting for the argument. A format instruction starts with a “%” character.

A transformation specification consists of the following:

% [F] [W] [G] U

F, **W** and **G** are optional, **U** is mandatory.

The meaning is:

F = Formatting sign.

W = Output distance „n“; n = Minimum number of the signs to be distributed.

G = Precision; „**.**“ or „**.***” „**.n**” (n = integer)

U = Transformation sign.

Formatting characters: **„[F]“**

Formatting characters	Meaning
„**-**„	Flush left adjustment.
„**+**“	With output of the algebraic sign „+“ or „-„
„ „ (space)	If the first character of the argument is no algebraic sign, a blank will be displayed.
„**0**“	On numerical output it is filling up with zeros will be done until the given width is reached.
„**#**“	The effect of „"**#**" depends on the change character: With "0" or „x, X“ the value with leading "0" or "0x" will be displayed. In "e, E, f" the value is displayed with decimal dot is displayed, even if no post comma places exist. With „g, G“ the value with decimal dot and post comma zeros is displayed.

Table 24: netSCRIPT - Function string.format - Formatting Characters

Distance: **„[W]“**

Output width	Meaning
„**n**„	At least n places are displayed. If necessary, places with leading zeros are filled in.

An unavailable or too small width will never cause characters not to be displayed. If the result of a transformation contains more characters than the width provides, nevertheless all characters will be displayed then.

Transformation characters: „U“

Transformation characters	Meaning
„**d**“, „**i**“	As a signed whole decimal number.
„**o**“	As an unsigned whole octal number
„**u**“	As an unsigned whole decimal number. Attention, unexpected string output can be produced when putting out negative values.
„**x**“, „**X**“	As an unsigned whole hexadecimal number With „x“ → a, b, c, d, e, f With „X“ → A,B,C,D,E,F
„**f**“	Floating decimal point number using the format [-]ddd.ddddd
„**e**“, „**E**“	As a number in exponential representation with the base 10. Thereby, the exponent includes at least 2 digits. With “e” [-]d.ddde±dd With “E” [-]d.dddE±dd
„**g**“, „**G**“	According to "e" or "E" if the exponent is less than -4, otherwise using the “f” format.
„**c**“	As Character.
„**s**“	As a character string.
„**%**“	The character "%" is displayed and no argument is evaluated.
„**q**“	Special option for the output of control character. The Interpretation of the control characters depends on the connected device.

Table 25: netSCRIPT - Function string.format - Transformation Characters

Precision: „[G]“

Output width	With change character	Meaning
„.**n**„ Where n is a whole number is.	d, i, o, u, x, X	Least number of characters to be displayed.
	e, E, f	Number of the post comma places to be displayed.
	g, G	Maximum number of characters to be displayed.
	others	Undefined behavior
„.*”		The next argument must be integer. If the value of this argument is negative, this accuracy specification will be ignored.

Table 26: netSCRIPT - Function string.format - Accuracy entries

Control characters within the string

Control character	Meaning on a terminal
„**\a**“	Ringing tone (also feasible with "\007").
„**\b**“	Backspace (positioning one character position backwards).
„**\f**“	Page feed.
„**\n**“	New line.
„**\r**“	Carriage return (to the beginning of the momentary line).
„**\v**“	Vertical tab character.
„**\'**“	Hyphen
„**\"**“	Quotation mark
„****“	Backslash

Table 27: netSCRIPT - Function string.format - Control Characters

The effect of the above control characters depends on the interpreting device in each case.

Examples of a formatting:

Code	Result
`a = 3.1415926535` `print(string.format("pi = %.4f", a))`	`pi = 3.1416`

`tag, title = "h1", "A title"`
`print(string.format("<%s>%s</%s>",`
`tag, title, tag))`
	`<h1>A Title</h1>`

`b = 56.3`
`print(string.format("%.4f", b))`
`print(string.format("%10.4f", b))`
`print(string.format("%12.4f", b))`
	`56.3000`
	`  56.3000`
	`    56.3000`

`c = -112345.12345`
`print(string.format("%d", c))`
`print(string.format("%e", c))`
`print(string.format("%g", c))`
	`-112345`
	`-1.123451e+005`
	`-112345`

`d = -123.4567e-3`
`print(string.format("%G", d))`
`print(string.format("%%", d))`
	`-0.123457`
	`%`

`print(string.format('%q', ' a string "test" and \n a new line '))`	"a string \"Test\" and \ a new line "
`print(string.format('%q', ' a string test and \n a \t new line '))`	"a string test and \ a new line"

5.2.5 string.gmatch (s, pattern)

This string manipulation function returns a function looking for **"pattern"** within **"s"** and then returns them. The returned function will notice up to where it has already searched and will return the next occurrence at the next invocation until the string has been processed completely.

Return parameter:

The search function.

Example:

"%a" looks for letters.

"%a+" looks for letters as long as there are letters available.

1. A little application.

Code	Result

`S = "hello wor1ld from Lua"`
`--[[note the number 1 in`
`S = "hello wor1ld from Lua"]]`
`MyFunc = string.gfind(S, "%a+")`
`print( MyFunc() )`
`print( MyFunc() )`
`print( MyFunc() )`
`print( MyFunc() )`
`print( MyFunc() )`
`print( MyFunc() )  --[[The function has no return!]]`
	Hello
	Wor
	ld
	from
	Lua

2. Application using the generic **for** loop:

Code	Result

`S = "hello world from Lua"`
`for wort in string.gfind(S, "%a+") do`
`  print(wort)`
 	
`end`
	hello
	world
	from
	Lua

3. Application using the generic **for** loop in conjunction with a table:

Code	Result

`MYTABLE = {}`
`S = "from=world, to=Lua"`
`for K, V in string.gfind(s,`
 	
`    "(%w+)=(%w+)") do`
`  print(k,v)`
 	
`    MYTABLE[K] = V`
`end`
	from world
	to Lua

Here a search for a word is performed, which is followed by an equal sign, then again followed by a word. Word 1 is stored within the first variable (k), and then word 2 is stored within variable v.

Then the table is allocated as follows

```
MYTABLE["from"] == "world"
MYTABLE ["to"]   == "Lua"
```

Accordingly:

```
MYTABLE.from == "world"
MYTABLE.to   == "Lua"
```

5.2.6 string.gsub (s, pattern, repl [, n])

This string manipulation function returns a copy of "**s**" in which all (or the first "**n**", if specified) occurrences of "**pattern**" have been replaced by a replacement string specified by „**repl**"

The replacement string may be a string, a table, or a function. "**gsub**" also returns, as its second value, the total number of matches having occurred.

If „**repl**" is a string, then its value is used for replacement for the found pattern. The character „**%**" works as an escape character: any sequence in „**repl**" of the form „**%n**" with "**n**" between 1 and 9 represents the value of the *n*-th captured substring (see below). The sequence `%0` represents the whole match. The sequence „**%%** represents a single „**%**" character.

Code	Result
`print(string.gsub("hello world", "(%w+)", "%1 %1"))`	`hello hello world world`
`print(string.gsub("hello world anton", "(%w+)", "%1 %1",1))`	`hello hello world anton`
`print(string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1"))`	`world hello Lua from`

If "**repl**" is a table, the value taken from the table will be replaced within „**s**" which is found within the table at the key, which belongs to the corresponding match of **„pattern".**

Code	Result
`local t = {name="lua", version="5.1"}` `print (string.gsub("$name-$version.tar.gz", "%$(%w+)", t))`	**lua-5.1.tar.gz**

If "**repl**" is a function, then this function is called every time a match occurs, as long as there are matches with "**pattern**" in „**s**".

Code	Result
`print(string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s) return loadstring(s)() end))`	`4+5 = 9 1`
To the explanation of the above call here a subdivision with her results:	
`print(string.find ("4+5 = $return 4+5$", "%$(.-)%$"))`	`7 18 return 4+5`
`s = "return 4+5" print(loadstring(s)())`	`9`

The character "$" in the above example is used as a delimiter.

If the value returned by the table query or by the function call is a string or a number, then the return value will be used as replacement string; otherwise, if it is **false** or **nil**, then no replacement will occur.

5.2.7 string.len (s)

This string manipulation function receives a string„**s**" and returns its length (number of characters). The character "****" within the string is not counted as character. If the „****" character is followed by a couple of zeroes (such as „**\000**") only the first zero after the „****" character will be counted in this context.

Code	Result
`print(string.len ("a\000")) print(string.len ("a\bbbbb")) print(string.len ("a/bbbbb"))`	`2 6 7`

5.2.8 string.lower (s)

This string manipulation function receives a string and returns a copy of this string "**s**" with all uppercase letters transformed to lowercase (and all other characters will remain unchanged)

5.2.9 string.match (s, pattern [, init])

This string manipulation function looks for the first match of "**pattern**" in the string "**s**". If a match occurs, the matching part of „**s**" will be returned, otherwise the value „**nil**" will be returned.

With the parameter „**init**", it can be specified at which position within the string to start the search

5.2.10 string.rep (s, n)

This string manipulation function reproduces the string „**s“** exactly **„n“** times.

Code	Result
`return = string.rep ("Lua ", 5)` `print( return )`	`Lua Lua Lua Lua Lua`

5.2.11 string.reverse (s)

This string manipulation function returns a string containing the characters of “**s**” in reversed order.

5.2.12 string.sub (s, i [, j])

This string manipulation returns the substring of string **s** beginning at “**i**” and continuing until “**j**” is reached. If **j** is not specified, it will be set to the value 1. I fit is negative, the until-position will be counted from the end of the string.

5.2.13 string.upper (s)

This string manipulation function receives a string and returns a copy of this string “**s**” with all lowercase letters transformed to uppercase (and all other characters will remain unchanged).

5.3 Table Manipulation

This library provides fundamental functions for table manipulation.

Note: The result of functions `concat`, `insert`, `remove` und `sort` is not defined, if there are „holes“ within the indices of the tables, such as {„a“, „b“, „c“, [5]=“d“}

5.3.1 table.concat (table [, sep [, i [, j]]]),

This table manipulation function produces one string from the contents of a table containing elements/entries only consisting of numbers and strings. The separator „**sep**“ (default „nil“) indicates how the single variables have to be separated. „**i**“ (default = 1) indicates from which line of the table output shall begin. „**j**“ (default = Table length) determines up to which line of the table the output shall be continued. In case of **i > j** an empty string will be returned.

Code	Result
`t1 = {"a", "b", "c", "d", "e"}` `print(table.concat (t1, "; ",2,4))`	`b; c; d`

5.3.2 table.insert (table, [pos,] value)

This table manipulation function inserts element „**value**“ at position „**pos**“ into the table. The default value for „**pos**“ is n+1, where n is the length of the table. If „**pos**“ > n+1, nothing at all will be inserted.

Code	Result

`t1 = {"a", "b", "c", "d", "e"}`
`table.insert(t1,3,"10")`
`print(table.concat (t1, "; "))`
	`a; b; 10;`
	`c; d; e`

If the table contains integer indices with holes, the input position is undefined.

5.3.3 table.maxn (table)

This table manipulation function returns the largest positive numerical index of the given table „**table**“, or zero if there is no numerical index in the table.

5.3.4 table.remove (table [, pos])

This table manipulation function removes the entry at position „**pos**“ from „**table**“, shifting down all other elements in order to close the gap, if necessary. The default value for „**pos**“ is n, so that a call “**table.remove(t)**” will remove the last element of table “**t**”. However, this is only valid if the table indices are a running sequence 1, …, n. commonly, an entry will be removed at position n, for which the following is valid:

`t[n]!=nil, t[n+1]==nil`

5.3.5 table.sort (table [, comp])

This table manipulation function sorts “**table**” elements in ascending order, i(table[n] < table[n+1] in ASCII-Code from table[1] to table[n]

If another sorting is desired, a parameter **comp** can be specified, which delivers two lines of the table and returns **true,** if the sorting of the lines corresponds to the desired sorting.

Without specification of „**comp**“ the table may contain only numbers or only strings.

5.4 Mathematical Functions

This library is an interface to the standard C math library. It provides all its functions inside the table “math”.

5.4.1 math.abs (x)

This mathematical function returns the absolute value of „**x**“

5.4.2 math.acos (x)

This mathematical function returns the arc cosine of „**x**“ (in radians).

5.4.3 math.asin (x)

This mathematical function returns the arc sine of „**x**“ (in radians)..

5.4.4 math.atan (x)

This mathematical function returns the arc tangent of „**x**“ (in radians)..

5.4.5 math.atan2 (y, x)

This mathematical function returns the arc tangent of „**y“*/*„x**“ (in radians), but uses the signs of both parameters to find the quadrant of the result. (It also handles correctly the case of „**x**“ being zero.).

5.4.6 math.ceil (x)

This mathematical function returns the smallest integer larger than or equal to „**x**“.

5.4.7 math.cos (x)

This mathematical function returns the cosine of „**x**“ (assumed to be in radians).

5.4.8 math.cosh (x)

This mathematical function returns the cosine of “**x**” (assumed to be in radians).

5.4.9 math.deg (x)

This mathematical function returns the angle “**x**” (given in radians) in degrees.

5.4.10 math.exp (x)

This mathematical function returns the value $\mathbf{e}^{\mathbf{x}}$ (**e^x**).

5.4.11 math.floor (x)

This mathematical function returns the largest integer smaller than or equal to „**x**“.

Important: When “**x**“ is already the result of former calculations, problems with rounding may occur and cause inaccuracies.

5.4.12 math.fmod (x, y)

This mathematical function returns the remainder of the division of “**x**” by “**y**”.

5.4.13 math.frexp (x)

This mathematical function returns **m** and **e** such that $x = m*2^e$, **e** is an integer and the absolute value of **m** is in the range [0.5, 1) (or zero when x is zero).

5.4.14 math.huge

The value HUGE_VAL is a constant representing positive infinity.

5.4.15 math.ldexp (m, e)

This mathematical function returns $m*2^e$ (where “**e**” is an integer).

5.4.16 math.log (x)

This mathematical function returns the natural logarithm of “**x**”.

5.4.17 math.log10 (x)

This mathematical function returns the base-10 logarithm of “**x**”.

5.4.18 math.max (x_1, x_2, ...,x_n)

This mathematical function returns the maximum value among its arguments.

5.4.19 math.modf (x)

This mathematical function returns two numbers, the integral part of “**x**” and the fractional part of “**x**”.

Code	Result

`Print(math.modf(5))`
`Print(math.modf(5.3))`
`Print(math.modf(-5.3))`
	`5  0`
	`5  0.3`
	`-5  -0.3`

5.4.20 math.pi

This mathematical function returns the value of *pi* (3.1415926535898).

5.4.21 math.pow (x, y)

This mathematical function returns $\mathbf{x}^{\mathbf{y}}$. (You can also use the expression **x^y** to compute this value.).

5.4.22 math.rad (x)

This mathematical function returns the angle “**x**” (given in degrees) in radians.

5.4.23 math.sin (x)

This mathematical function returns the sine of “**x**” (assumed to be in radians).

5.4.24 math.sinh (x)

This mathematical function returns the hyperbolic sine of “**x**”.

5.4.25 math.sqrt (x)

This mathematical function returns the square root of “**x**”. (You can also use the expression „**x^0.5**“ to compute this value.).

5.4.26 math.tan (x)

This mathematical function returns the tangent of “**x**” (assumed to be in radians).

5.4.27 math.tanh (x)

This mathematical function returns the hyperbolic tangent of “**x**”.

6 Special Functions for netTAP

Additional libraries for netTAP are: "**bit**" for bit operations and "**util**" for special utilities.

The interfaces are accessed via instances.

6.1 Bit-Operations

These operations are located in the library "**bit**". Their names start with the letter '**b**'.

This library enables performing bitwise relations between unsigned 32 bit integer variables.

Numbers can be entered in hexadecimal notation in the range 0 - ffffffff. If larger values are assigned to the variables, these are delimited to 0xffffffff without any error message.

6.1.1 bit.band

bit.band(a,b)	
Bit-wise AND relation of the unsigned integer values a and b within the range 0 - 0xFFFFFFFF.	
Argument a numeric	First unsigned 32 bit number.
Argument b numeric	Second unsigned 32 bit number.
Return value	Unsigned 32 bit number with bit-wise AND relation
numeric	
Status-/Error code	Lua error, for instance: a number was expected

State-/error numbers are saved in the variable lasterror.

6.1.2 bit.bor

bit.bor(a,b)	
Bit-wise OR relation of the unsigned integer values a and b within the range 0 - 0xFFFFFFFF.	
Argument a numeric	First unsigned 32 bit number.
Argument b numeric	Second unsigned 32 bit number.
Return value	Unsigned 32 bit number with bit-wise OR relation.
numeric	
Status-/Error code	Lua error, for instance: a number was expected

State-/error numbers are saved in the variable lasterror.

6.1.3 bit.bxor

bit.bxor(a,b)	
Bit-wise XOR relation (exclusive OR relation) of the unsigned integer values a and b within the range 0 - 0xFFFFFFFF.	
Argument a numeric	First unsigned 32 bit number.
Argument b numeric	Second unsigned 32 bit number.
Return value	Unsigned 32 bit number with bit-wise XOR relation.
numeric	
Status-/Error code	Lua error, for instance: a number was expected

State-/error numbers are saved in the variable lasterror.

6.1.4 bit.bnot

bit.bnot(a)	
Bit-wise inverting of argument a	
Argument a numeric	Unsigned 32 bit number to be negated.
Return value	Unsigned 32 bit number having been inverted bit-wise
numeric	
Status-/Error code	Lua error, for instance: a number was expected

State-/error numbers are saved in the variable lasterror.

6.1.5 bit.lshift

bit.lshift(a, n)	
Shift bits to the left.	
Argument a numeric	Unsigned 32 bit number to be shifted.
Argument n numeric	The number of bit positions how far the bits should be shifted to the left.
Return value	Unsigned 32 bit number bit-wise shifted to the left by n positions. The bit positions becoming vacant on the right will be filled in with zeros.
numeric	
Status-/Error code	Lua error, for instance: a number was expected

State-/error numbers are saved in the variable lasterror.

6.1.6 bit.rshift

bit.rshift(a, n)	
Shift bits to the right.	
Argument a numeric	Unsigned 32 bit number to be shifted.
Argument n numeric	The number of bit positions how far the bits should be shifted to the right.
Return value	Unsigned 32 bit number bit-wise shifted to the left by n positions. The bit positions becoming vacant on the left will be filled in with zeros.
numeric	
Status-/Error code	Lua error, for instance: a number was expected

State-/error numbers are saved in the variable lasterror.

6.2 Conversions of Numbers

The following conversions of numbers can be accomplished:

- From Lua number (stored in any format) to binary format (stored within a string).
- From binary format (stored within a string) to Lua number (stored in any format).

Important note about range of allowed values and accuracy: Internally, Lua works with double floating point numbers with 52 Bit mantissa. This allows the representation of all signed or unsigned into values from 8 to 32 bits width without any inaccuracies caused by truncation losses, however, this does not apply for 64-bit values.

6.2.1 util.NumToBin

util.numToBin(num,util.Type[,ENDIAN])	
This conversion function converts a number into a binary representation, e.g. **numToBin(0x12345678, util.UINT32)** results in a string with the bytes 0x78 0x56 0x34 0x12.	
Argument 1 num	Number to be converted in any arbitrary Lua format.
Argument 2 Type	The data type to be converted. The following types are possible: UNIT8, UNIT16, UNIT32, UNIT64; INT8, INT16, INT32, INT64; FLOAT, DOUBLE. The prefix "**util**." may never be omitted. It is used in conjunction with every data type.
Argument 3 ENDIAN	LITTLE_ENDIAN: The low byte comes first within the return string. BIG_ENDIAN: The high byte comes first within the return string. This argument is optional. The default value is LITTLE_ENDIAN.
Return value String	**String** with binary representation. **Nil**, on conversion error.
Status-/Error code	ERR_INVALID_PARAMETER (0xc0800301) Invalid value for destination type or argument ENDIAN. ERR_OUT_OF_RANGE (0xc0800302) The number violates the value range of the destination type.

State-/error numbers are saved in the variable lasterror.

6.2.2 util.BinToNum

util.binToNum(str,util.Type[,ENDIAN])	
This conversion function converts the binary representation of a number into a Lua-Number. **binToNum(str, util.UINT32)**, where **str** is a String containing the Bytes 0x78 0x56 0x34 0x12, would return the number 0x12345678.	
Argument 1 str	String containing the sequence of binary characters.
Argument 2 Type	The destination data type for the conversion. The following types are possible: UNIT8, UNIT16, UNIT32, UNIT64; INT8, INT16, INT32, INT64; FLOAT, DOUBLE. The prefix "**util**." may never be omitted. It is used in conjunction with every data type.
Argument 3 ENDIAN	LITTLE_ENDIAN: The low byte comes first within the return string. BIG_ENDIAN: The high byte comes first within the return string. This argument is optional. The default value is LITTLE_ENDIAN.
Return value String	**Number** in the Lua format. **Nil**, in case of conversion error.
Status-/Error code	ERR_INVALID_PARAMETER (0xc0800301) Invalid value for destination type or argument ENDIAN. ERR_WRONG_SIZE (0xc0800303) String has the wrong length for the given type.

State-/error numbers are saved in the variable lasterror.

6.3 LED – Control

This LED control function enables you to set or reset the Duo COM-LED status display in the secondary netTAP network module located at the right side of the device. Also see netTAP NT100 User Manual, section Serial Communication with netSCRIPT

If the netSCRIPT program is not running, for instance, at debugging, or in case no netSCRIPT program has been loaded at all, the device will take over control of the LED. In this case, the LED will cyclically blink in red color.

6.3.1 util.SetLed

util.setLed(Name, [Condition])	
Switches the Duo COM-LED on or off.	
Argument 1 Name	**run** → the green COM-LED will be switched on.
	error → the red COM-LED will be switched on.
Argument 2 Condition	**true** → LED on (default).
	false → LED off.
Return value	none
Status-/Error code	UTIL_INVALID_PARAMETER (0x C0800401) Unknown value for target type or **ENDIAN**.

6.4 Requesting the Cycle Time of the Script

For some applications, it can be important which cycle time has been given to the script by SYCON. The cycle time can be requested with the following function:

6.4.1 util.GetCycleTime

util.GetCycleTime()	
This function returns the configured cycle time of the script, specified in units of milliseconds. The time is adjusted within the configuration tool SYCON.net and will be transferred to the destination device together with the script program.	
Arguments	none
Return value	Script cycle time during milliseconds
Status-/Error code	none

6.5 CRC Checksum Functions

The check sum function has been implemented similarly to the Rocksoft CRC model. More information about the Rocksoft CRC model is available at http://www.ross.net/crc/download/crc_v3.txt.

To calculate any checksums, at first a hash object needs to be created. Using this hash object, the CRC calculation is performed from the provided data by additional function calls. Also a check sum of multiple single data packets can be created using this object.

6.5.1 Creation of Check Sum Object „HashCreate“

util.HashCreate (Type constant, [Parameter])	
Creates a hash object, performs some precalculations which are done only once (at CRC a table is precalculated).	
Arguments: Type constant	This item determines which kind of procedure is used for check sum calculation. For possible values see Table 28.
Parameter	Are only possible if type constant „util.HASHTYPE_CRC“ is used. For possible values see Table 29.
Return value	Hash object: Object, which is used to perform check sum calculations
Status-/ error code	**UTIL_INVALID_PARAMETER** (C0800401) Invalid parameter value (e.g. target type, endianness, LED identifier).

6.5.1.1 Variants of Check Sum Calculation

Check Sum	Type constant in function in util.HashCreate	Parameter
CRC	util.HASHTYPE_CRC	See below
Byte wise XOR	util.HASHTYPE_XOR	-
Byte wise sum mod 256	util.HASHTYPE_SUM	-

Table 28: netSCRIPT - Variants of check sum calculation

6.5.1.2 Parameter for Check Sum Calculation Variant CRC

For a CRC check sum the hash object is created as follows:

```
util.HashCreate(util.HASHTYPE_CRC, Width, Poly, Init,
RefIn, RefOut, XorOut)
```

In this context, the arguments have the following meaning

Argument	Meaning
Width	Degree of the polynomial, i.e. the largest exponent within the polynomial (8..32)
Poly	Polynomial as numerical value, e.g. the coefficients without the leading 1. For the polynomial $x^{16} + x^{15} + x^2 + 1$ the following is valid: Width = 16 and Poly = (binary) 1000000000000101 = 0x8005; For the polynomial $x^{15} + x^2 + 1$ the following is valid: Width = 15 and Poly = (binary) = 0x5
Init	Initial value of the CRC register (0.. 2^width – 1)
RefIn	Indicates the order of bit processing within the data bytes **false**: most significant bit first **true**: least significant bit first
RefOut	If set to **true**, the return value of the CRC calculation is reversed (mirrored) such as 10101111 → 11110101
XorOut	The return value of the CRC calculation is finally (eventually after bitwise reversal (RefOut = true)) XOR-related with the value specified here. Allowed range: 0 ... 2^width – 1

Table 29: netSCRIPT - CRC Parameters

6.5.2 Functions for Check Sum Calculation

The created hash object provides the following functions for check sum calculation:

6.5.2.1 Data Transfer to Hash Object

:Hash (string)	
„Adds“ the calculation of the data of „**string**“ to the checksum.	
Arguments: **string**	byte wise addition onto the check sum of the specified object
Parameter	none
Return value	none

If this function is called more than once (without having been reset meanwhile), the check sum will be collectively calculated over all strings having been transferred.

6.5.2.2 Check Sum Request

:HashResult()	
Delivers the current check sum as numerical value of the specified object.	
Arguments:	none
Parameter	current check sum of specified object
Return value	none

6.5.2.3 Reset of Hash Object

Prior to a repeated calculation of the check sum for one or multiple data packets, the hash object needs to be reset to the initial state.

:HashReset()	
Resets the hash object to the initial state.	
Arguments:	none
Parameter	none
Return value	none

6.5.2.4 Example Script for the Usage of CRC Functions

Creation of hash object „**h**“ (necessary only once):

```
h = util.HashCreate(util.HASHTYPE_CRC, 16, 0x8005, 0,
true, true, 0)
```

Calculation of CRC value:

```
h:Hash(„Halli“)
```

Read out of CRC value:

```
hcrc = h:HashResult()
```

Using the string functions, the CRC value can be inserted or appended at an arbitrary position of the string to be sent.

Prior to the next calculation, the hash object is required to be reset:

```
h:HashReset()
```

7 Serial Communication

The communication to the serial UART interface runs independently from the communication to the controller (also denominated as PLC or host).

The entire send and receive processing of the serial communication interface is automatically controlled by netSCRIPT functions. Only the levels 1 and 2 of the OSI communication model are administered outside of and independently from netSCRIPT. Therefore, the handling of start and stop bits and parity bits is no task to be handled within the netSCRIPT program. For correct function of the serial UART they only need to be configured once at the initialization of the serial port.

Other protocol requirements can be realized with netSCRIPT within the program parts for send and receive operations themselves, for instance CRC and check sum processing.

The communication functions for block mode provided by netSCRIPT allow serial communication over the UART only in Half-Duplex transmission.

The communication functions for character mode provided by netSCRIPT allow serial communication over the UART in Full-Duplex transmission.

Verify in your script if the serial line is free for a send command for RS-422 and RS-485 with multiple active participants.

7.1 Configuration Parameters for Data Transmission

The following configuration parameters have been defined for serial communication:

Parameter	Table key	Range	Default
Baud rate	**baudrate**	6 … 1000000	115200
Number of data bits	**databits**	1 … 8	8
Parity mode	**paritymode**	See below	None
Number of stop bits	**stopbits**	1 … 65535	1
Shift direction	**shiftdirection**	See below	LSB first
Interface type	**interfacetype**	See below	RtsCts
Handshake polarity	**handshakepolarity**	See below	0
Bus idle symbol	**busIdlesymbol**	0/1	1
Invert data mode	**invertdatamode**	true/false	false
Transfer Mode	**charmode**	true/false See below	false

Additional parameter for block mode:

Parameter	Table key	Range	Default
Acknowledge delay time: On reception, this timing parameter is the maximum time to wait until the first character is received (specified in units of 10ns)	**ackdelaytime**	0-2^32-1	0 representing ∞
Character delay time: On reception, this timing parameter is the maximum time between two characters until the end of the character sequence is detected (specified in units of 10ns)	**chardelaytime**	0-2^32-1	0 representing ∞
Character sequence for detection of end	**endpattern**	See below	empty
Mask for detection of end	**endmask**	See below	empty
Number of trailing bytes	**traillen**	0 … 255	0

Table 30: netSCRIPT - UART Parameters

The parameters are saved in a netSCRIPT table. They are accessible in the netSCRIPT table via the "table key" given in the table above and they are also modifiable.

The default value will become effective if no table is defined or no entry exists in the table for this key.

The parameters of the table above, for which the entry: "See below" can be found in column "Range", can accept the following predefined values:

paritymode:

port.PARITY_EVEN	0 in the even number of ones in the transfer data string, otherwise 1
port.PARITY_ODD	1 in an even number of ones in the transfer data string, therwise 0.
port.PARITY_NONE	(no Parity bit will be transmitted.)
port.PARITY_MARK	1 Parity bit with the fixed value 1 will be transmitted.)
port.PARITY_SPACE	(1 Parity bit with the fixed value 0 will be transmitted.)

shiftdirection:

port.SHIFT_DIRECTION_LSB_FIRST:
(The lowest order byte will be transmitted at first.)

port.SHIFT_DIRECTION_MSB_FIRST:
(The highest order byte will be transmitted at first.)

handshakemode:

port.HANDSHAKE_MODE_RS232	(without Hardware Handshake)
port.HANDSHAKE_MODE_RS232_RTS_CTS	(with Hardware Handshake)
port.INTERFACE_TYPE_RS422	RS422
port.HANDSHAKE_MODE_RS485	RS485

handshakepolarity:

This parameter determines for the RS232 interface type with hardware handshake the current status of RTS signal. This parameter does not have any importance for all other types of interfaces.

charmode:

true: the data transfer is done in character mode.

false: the data transfer is done in block mode.

endpattern:

This is a string of max. 8 bytes, and defines the character sequence by which the end of the telegram is recognized.

endmask:

This is a string of max. 8 bytes. If this parameter is specified, it defines the bits, which have to be taken into account when recognizing the end of the telegram from **endpattern**. The following rules apply:

0 → The bit is not used for comparison

1 → The bit is used for comparison

7.1.1 Functions for Initialization of the Serial Interface

7.1.1.1 PortReadConfigDb

PortReadConfigDb()	
This function reads configuration table of the UART interface saved by the SYCON.net (see section *Device Selection*) and transfers this table in a local table of the script in order to be able to modify the table there, if necessary.	
Arguments:	none
Return value	A **Table** is returned if the configuration file stored within the netTAP device by SYCON.net is present and can be interpreted.
	Nil if no table could be generated.
Status-/Error code within the variable „**lasterror**“	**ERR.PORT_NO_UARTDB** (0xC0800214) There is no stored configuration table present.
	ERR.PORT_PARSING_UARTDB (0xC0800215) The stored configuration table could not be read.

7.1.1.2 PortOpen

PortOpen([port number],[configuration table])	
Initializes and configures the UART interface	
Arguments: port number	Optionally, default value = 2, possible values: 0...3. netTAP100 only supports port 2
configuration table	Optionally, if not available, the default parameter is used. The entries of the table overwrite the default parameters.
Return value	Port instance if executed successfully
	Nil if the port could not be opened.
Status-/Error code within the variable „**lasterror**“	**err.PORT_NO_SUCH_PORT** (0x40800211) The transferred port number does not exist or is not supported.
	err.PORT_ALREADY_OPEN (0x40800212) The port has already been opened.
	err.PORT_INVALID_CONFIG (0xC0800201) The transferred configuration table contains an invalid value.
	err.PORT_XC_INIT_FAILED (0xC0800206) The serial interface could not be initialized.

7.1.1.3 :PortClose

:PortClose ()	
This function rejects all data not yet sent and closes the port. The instance which was returned by the function "PortOpen" is to be inserted as a prefix immediately preceding the colon.	
Arguments:	none
Return value:	none
Status-/Error code within the variable „lasterror“	**err.PORT_NOT_OPEN** (0xC0800213) The used instance of the port has not been opened.
	err.PORT_XC_INIT_FAILED (0xC0800206) The port could not be closed.

7.1.2 Example for Adjustment of Parameter Settings

The performed functions must be executed in the given order. The single function calls are optional. However, the last function call is mandatory.

7.1.2.1 Function Call including reading SYCON.net Settings

```
conf = PortReadConfigDb() --[[read in SYCON.net
                             configuration table]]--
conf.baudrate = 9600 – just modify baudrate
xuart = PortOpen(conf) – open the interface with
                         configuration table “conf”
```

See also the example script on the DVD
„Examples\netSCRIPT\Serial Port Blockmode\blkmode.lua

7.1.2.2 Function Call including without reading SYCON.net Settings

In this case the standard parameters described in section Configuration Parameters for Data Transmission apply. Then the function call

```
xuart = PortOpen()
```

is sufficient.

For example, if only the baud rate shall be modified compared to the standard parameter set, the following sequence of instructions can be applied:

```
conf = {baudrate = 9600}
xuart = PortOpen(conf)
```

See also the example script on the DVD
„Examples\netSCRIPT\Serial Port Blockmode\blkmode.lua

7.1.2.3 Closing the Ports

To close the port which was opened in the example in section 7.1.2.1 and 7.1.2.2 use:

```
xuart:PortClose()
```

8 Serial Communication in Block Mode

Several send and receive requests can be transferred from netSCRIPT to the UART synchronously. netSCRIPT reserves a special area in memory of 16 Kbytes for transfer of serial data **to** and **from** the UART. This memory area is dynamically allocated to the single requests according to their required amount of transmitted user data. Maximally, each send or transmit command can transfer 1024 Bytes user data. 64 Bytes of administration data are put in front of each command. In the limiting case, this results in 16 Kbytes / 1088 Bytes = 15 requests each with 1024 Bytes of user data to be processed synchronously. For example, if only 64 Bytes of user data are required to be transmitted, then 16 Kbytes / 128 Bytes = 128 requests may be activated.

Send and receive requests are activated as sequential requests in the serial UART via the netSCRIPT program. It is possible to transfer several requests to the UART synchronously. At each call, netSCRIPT dynamically reserves a sufficiently large transmission buffer per command, which is provided for sending and receiving. The number of requests to be activated in parallel synchronously depends on the amount of user data of the single requests. The parameters required for the calculation of this number can be found in the introduction of this section.

The serial UART provides two queues (FIFOs). Into the so called Request FIFO the single send and receive requests are entered according to the FIFO principle (**F**irst **I**n **F**irst **O**ut). netSCRIPT provides corresponding functions for this purpose. The UART sequentially processes the requests and acknowledges the requests within the so called Confirmation FIFO in a manner readable for netSCRIPT. Accessing this queue enables the netSCRIPT user to determine whether the requests have been processed successfully or a failure happened. Doing so, the acknowledgement of the request is removed from the Confirmation FIFO and the formerly allocated memory space is released and provided as free storage area for new requests. Thus a circulation of request blocks is set up. Requests cannot overtake one another, as the FIFO principle is always obeyed.

If two or more send requests are activated within the UART immediately following one after the other, the data are sent on the serial interface without any gap if the baud rate is not too high and the requests are processed with sufficient speed. In this manner it is possible to produce data streams of a length of more than 1024 Byte.

If the execution of the netSCRIPT program is faster than the requests can be processed by the UART due to a low baud rate, a lack of free request blocks may occur. If no free request block is available any more, a send or receive request will finally be denied.

The single requests can be supplied with an identification number to identify them uniquely after processing. Send and receive requests with or without identification number can arbitrarily be combined.

8.1 Block Processing without Identification Number

When using send request without identification number, the processed requests do not need to be removed from the Confirmation FIFO. For this kind of request netSCRIPT always takes over this task automatically performing a check for processed requests without ID at the invocation of each function of the serial interface and removing those requests. In this way, an arbitrary number of subsequent send requests can be activated without having to use a further netSCRIPT function. Thus the script program is simply, but there is no possibility to check for success or failure of the requests.

The general sequence of execution is illustrated in the following picture, and is described in the following table.

Point of time	Description
T1	The first sending block #1 is in the UART to be processed for sending. 5 further blocks wait in the queue of the Request FIFO for being processed in UART.
T2	The first block has been processed completely by the UART. It is forwarded to the Confirmation FIFO queue.
T3	The first receive block is currently in processing, two sent blocks still remain in the Confirmation FIFO queue. The receive block remains as long as it's being processed within the UART as long as the defined number of characters has been received.
T4	The receive request # 4 is filled with incoming data as they arrive at the UART until the expected amount of data has been reached. If now during T4 a request on received data occurs within the netSCRIPT program, all send requests without ID will be removed from the Confirmation FIFO queue. (Symbolically these requests are crossed). The Receive data are read out and this block is also released.
T5	The send request #5 is currently in the UART to be processed for sending. The receive data frame of request #4 can be read out by netSCRIPT.

Table 31: Sequence of Block Processing without Identification Number

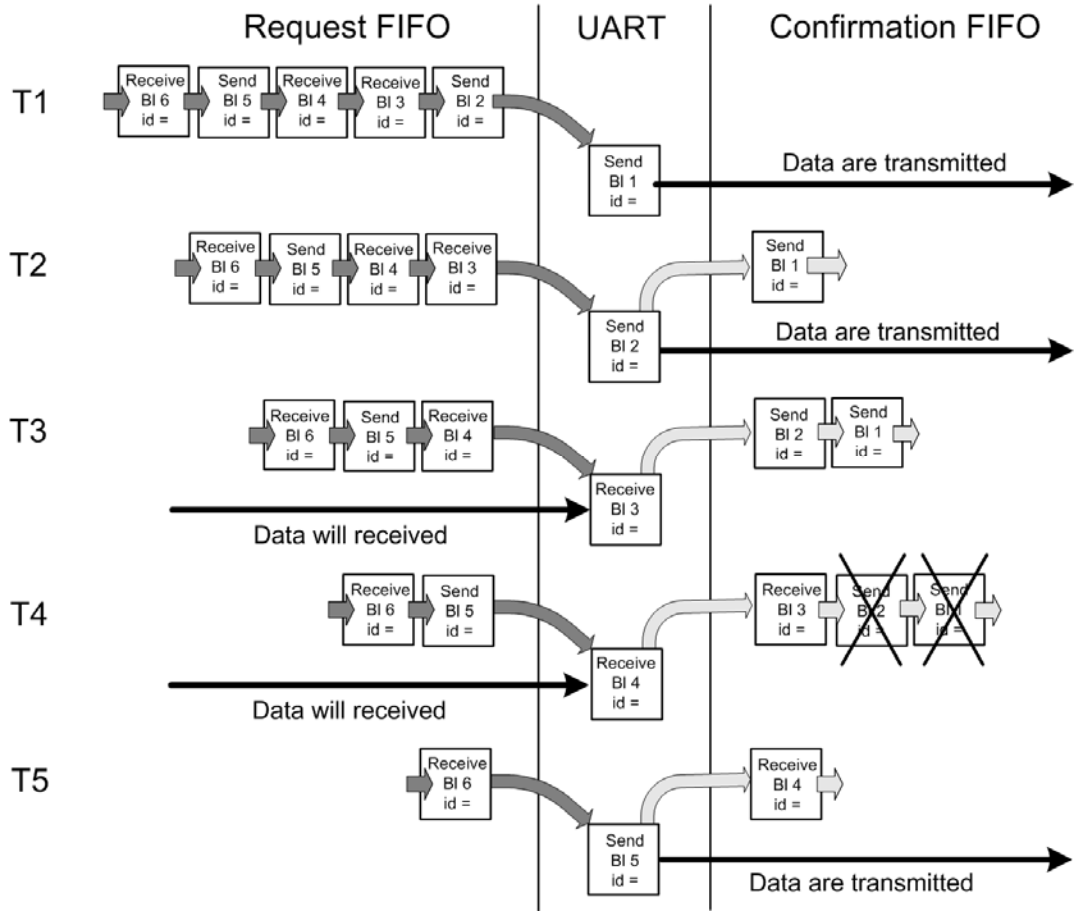


Figure 10: Send and Receive Data without Block-Id

8.2 Block Processing with Identification Number

Aim of using requests with identification number is to be able to uniquely identify these during request processing and to be aware of their status of transmission. This directly affects the program execution in netSCRIPT as already processed requests must explicitly be removed from the queue of the Confirmation FIFO by netSCRIPT function calls. Multiple requests with ID may be requested synchronously.

The general sequence of execution is illustrated in the following picture, and is described in the following table.

Point of time t	Description
T1...T3	The execution up to the time T3 exactly corresponds to the execution at block processing without identification number.
T4	If the data of the first receive data frame are to be read, the data frames for sending must be read / deleted from the Confirmation FIFO queue by 2 function calls before being able to process the receive data of request #3.
T5	Reading of the blocks 1, 2 and 3 has removed these from the Confirmation FIFO queue. Thus the request #4, which already has been filled with data at the time T4, can now be processed.

Table 32: Sequence of Block processing with Identification Number

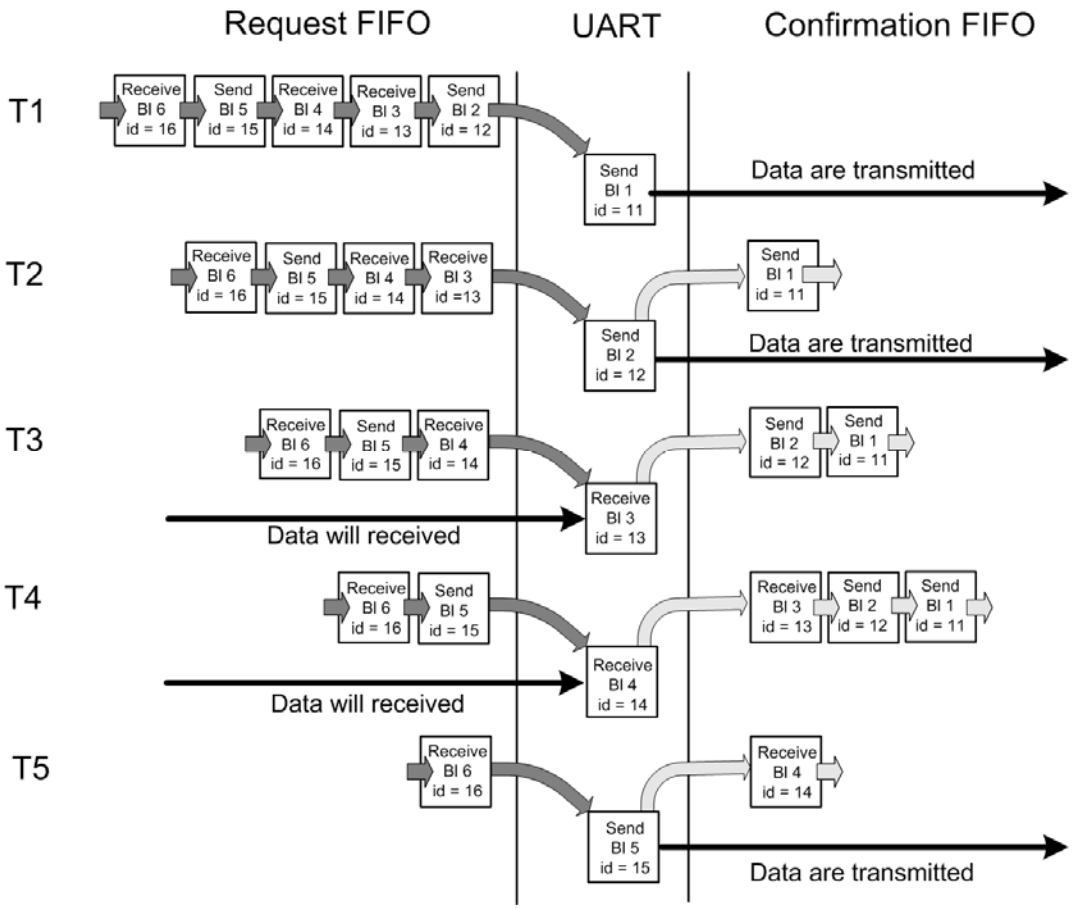


Figure 11: Send and Receive Data with Block-Id

8.3 Send / Receive Functions for the Block Mode

Before the functions described in this chapter can be used, the UART must be initialized by calling function „**PortOpen**“. Doing so, will create an instance, which has to be prepended to functions.

Example:

```
myuart = PortOpen()
```

By this call mentioned above a port instance has been generated to be prepend to all of the following port functions.

Example for passing the instance to the send function:

```
myuart:PortSend(.........)
```

8.3.1 :PortSend

:PortSend(string, [id])	
Generates a sending request in the Request FIFO queue and passes the data to be sent	
Arguments: **string**	Contains the data to be sent. Data length ≤ 1024 Byte.
id	Optionally, request identification number within the range of 0...2^32-1.
Return value:	**true**, if the send request could successfully be put into the Request FIFO queue.
	false, if the send request could not be generated.
Status-/Error code in „**lasterror**“	**err.PORT_STRING_TOO_LONG** (0x40800218) The string having been passed exceeds the limit of 1024 bytes.
	err.PORT_FIFO_FULL (0x40800205) The Request FIFO is completely filled; try to wait for the end of processing of running requests and to retry the request later on.
	err.PORT_NO_BUFFER (0x40800210) No send buffer of sufficient size could be provided. First data of completed requests must be fetched in order to be able to provide buffers of sufficient size.
	err.PORT_NOT_OPEN (0xC0800213) The specified port has not been opened.
	err.PORT_WRONG_MODE (0xC0800203) The port was opened in character mode and can’t be used requested with block mode.

8.3.2 :PortReceive

:PortReceive(len, [fMatch, [fTimeout]], [id])	
Creates a receive request within the Request FIFO Queue. The parameters **len, fMatch** and **fTimeout** are transferred to the UART for detection of end of reception. The UART finishes the reception of data if any of these three conditions is fulfilled. Contrary to a pure send request without ID processing of the request must in each case be accomplished using the function **:PortIsReceiveDone()** when issuing a receive request, independently, whether an **id** has been passed to the **:PortReceive()** function or not.	
Arguments: **len**	Number of bytes to be received. Possible values are: 1...1024, the default value is 1024.
fMatch	**true,** the parameters **endPattern** and **endMask** are used for detection of end of reception. The reception data length must be less than the argument **len.**
	false, no checks for detection of end of reception are done (default).
fTimeout	**true**, (default) the UART parameters **ackDelayTime** and **charDelayTime** are used for the monitoring by the UART.
	false, the UART parameters **ackDelayTime** and **charDelayTime** are not used for the reception monitoring by the UART.
id	Freely assignable request identification number within the range 0...2^32-1.
Return value:	**true**, if the request could successfully be put into the Request FIFO queue.
	false, if the request could not be generated.
Status-/Error code in „**lasterror**“	**err.PORT_INVALID_PARAMETER** (0x 40800216) An invalid parameter has been passed; please check the allowed range of values.
	err.PORT_FIFO_FULL (0x40800205) The Request FIFO is completely filled; try to wait for the end of processing of running requests and to retry the request later on.
	err.PORT_NO_BUFFER (0x40800210) No send buffer of sufficient size could be provided. First data of completed requests must be fetched in order to be able to provide buffers of sufficient size.
	err.PORT_NOT_OPEN (0xC0800213) The specified port has not been opened.
	err.PORT_WRONG_MODE (0xC0800203) The port was opened in character mode and can’t be used requested with block mode.

8.3.3 :PortExchange

:PortExchange(string, len, [fMatch, [fTimeout]], [id])	
Generates a sending request in the Request FIFO Queue. Passes the data to be sent (Argument **string**). After the sending of the data by the UART this switches to receive operation and immediately waits for a receive signal. The parameters **len**, **fMatch** and **fTimeout** are passed to the UART to the detection of end of reception. The UART finishes the reception of data if any of these three conditions is fulfilled. Contrary to a pure send a request without ID processing of a combined send/receive request must be accomplished using function **:PortIsExchangeDone()**, independently, whether an **id** has been passed to the **: PortExchange ()** function or not.	
Arguments: **string**	Contains the length of the data to send. Data length must be less than 1024 bytes.
len	**Number** of bytes to be received. Possible values are: 1...1024, default is 1024. If this number of bytes have been received, receive operation will be finished.
fMatch	**true**, the parameters **endPattern** and **endMask** are used to the detection of end of reception. The actual receive data length of the character sequence to be expected must be less than the argument **len**.
	false, no checks for detection of end of reception are done (default).
fTimeout	**true**, (default) the UART parameters **ackDelayTime** and **charDelayTime** are used for the monitoring by the UART.
	false, the UART parameters **ackDelayTime** and **charDelayTime** are not used for the reception monitoring by the UART.
id	Freely assignable request identification number within the range 0...2^32-1.
Return value:	**true**, if the request could successfully be put into the Request queue.
	false, if the request could not be generated.
Status-/Error code in „**lasterror**“	**err.PORT_STRING_TOO_LONG** (0x 40800218) The string having been passed exceeds the limit of 1024 bytes.
	err.PORT_INVALID_PARAMETER (0x 40800216) An invalid parameter has been passed; please check the allowed range of values.
	err.PORT_FIFO_FULL (0x40800205) The Request FIFO is completely filled; try to wait for the end of processing of running requests and to retry the request later on.
	err.PORT_NO_BUFFER (0x 40800210) No send buffer of sufficient size could be provided. First data of completed requests must be fetched in order to be able to provide buffers of sufficient size.
	err.PORT_NOT_OPEN (0x 40800213) The specified port has not been opened.
	err.PORT_WRONG_MODE (0xC0800203) The port was opened in character mode and can’t be used requested with block mode.

8.3.4 :PortIsSendDone

:PortSendDone()	
This function checks for a processed request within the Confirmation FIFO Queue. If present, this request will be removed and its **id** will be passed as return value. Exactly one, namely the oldest send request will be removed. This function must be called only if an **id** has been passed during activation of the send request. Otherwise, send requests without any **id** will be removed automatically by netSCRIPT. Please note that the function will always remove the timely oldest request. If no processed send request is available, the function cannot be processed successfully.	
Arguments:	none
Return values:	**nil**, if no send request has been completely processed, or instead of a completed send request, another kind of completed request is ready to be fetched within the Confirmation FIFO Queue. This needs to be removed off the Confirmation FIFO Queue applying the according functions.
	id of the released sending request.
Status-/Error code in „**lasterror**“	**err.PORT_FIFO_EMPTY** (0x40800204) The Confirmation FIFO Queue currently does not contain any completely processed request
	err.PORT_NO_CONFIRMATION (0x40800217) A completely processed request is located at the Confirmation FIFO Queue, but this is no send request. Please always fetch the processed requests exactly in the order of their activation.
	err.PORT_NOT_OPEN (0xC0800213) The specified port has not been opened.
	err.PORT_WRONG_MODE (0xC0800203) The port was opened in character mode and can’t be used requested with block mode.

8.3.5 :PortIsReceiveDone

:PortReceiveDone()	
This function checks for a processed request within the Confirmation FIFO Queue. If present, this request will be removed and its **id** will be passed as return value. Exactly one, namely the oldest send request will be removed. Please take note that the function will always remove the timely oldest request. If no processed send request is available, the function cannot be processed successfully.	
Arguments:	none
Return value: **status**	**port.STA_ACK_TIMEOUT** (3) Within the configured time **ackdelaytime** no character could be received
	port.STA_CHAR_TIMEOUT (4) The reception has been finished due to the **chardelaytime** being elapsed.
	port.STA_SIZE_REACHED (2) The specified number of characters was received.
	port.STA_PATTERN_MATCH (1) The receive request has been processed successfully. The configured detection of end of reception by **endpattern** and **endmask** was successful and the reception has therefore been completed.
	nil, if no other receive request has been completed or instead of a completed receive request another kind of completed request is ready to be fetched at the Confirmation FIFO Queue. At first, this request must be removed from the Confirmation FIFO Queue with an according function. If **nil** is returned in this case, all other return values will also be set to the value **nil**.
data	The received string of data
	An empty string of length 0, if the status contains **port.STA_ACK_TIMEOUT** or **port.STA_CHAR_TIMEOUT**, or if **error ≠ nil** is true.
error	**nil**, with no error occurred.
	port.ERR_PARITY_ERROR (2) A parity error has been detected during reception.
	port.ERR_FRAMING_ERROR (4) A telegram frame error has been detected, such as a start –stop bit error.
	port.ERR_BREAK_DETECTED (1) A break has been received on the serial interface.
id	**nil**, if the receive request has not been supplied with an **id** at its generation
	id, which has been supplied to the receive request at generation with t **:PortReceive()**.
Status-/Error code in „**lasterror**“	**err.PORT_FIFO_EMPTY** (0x40800204) The Confirmation FIFO Queue currently does not contain any completely processed request
	err.PORT_NO_CONFIRMATION (0x40800217) A completely processed request is located at the Confirmation FIFO Queue, but this is no send request. Please always fetch the processed requests exactly in the order of their activation.
	err.PORT_NOT_OPEN (0xC0800213) The specified port has not been opened.
	err.PORT_WRONG_MODE (0xC0800203) The port was opened in character mode and can’t be used requested with block mode.

Example of the call of the function:

```
sta, data, e , ident = port:PortIsReceiveDone()
if sta == port.STA_PATTERN_MATCH then

end
```

8.3.6 :PortIsExchangeDone

:PortIsExchangeDone()	
This function checks for a processed combined send/receive request within the Confirmation FIFO Queue. If present, this request will be removed and its **id** and data will be passed as return values. Exactly one, namely the oldest send request will be removed. Please take note that the function will always remove the timely oldest request. If no processed combined send/receive request is available, the function cannot be processed successfully. The function returns the contents of the receive data of a send request having been put into the Request FIFO-Queue by the function **:PortExchange**.	
Arguments:	none
Return values : **status**	**port.STA_ACK_TIMEOUT** (3) Within the configured time **ackdelaytime** no character could be received.
	port.STA_CHAR_TIMEOUT (4) The reception has been finished due to the **chardelaytime** being elapsed.
	port.STA_SIZE_REACHED (2) The specified number of characters was received.
	port.STA_PATTERN_MATCH (1) The receive request has been processed successfully. The configured detection of end of reception by **endpattern** and **endmask** was successful, and the reception has been completed.
	nil, If the receipt block not concluded, or not the first block in the Confirmation FIFO cue. In this case all the other return values are also **nil**.
data	**string**, the received data string.
	An empty string of length 0, if the status contains s **port.STA_ACK_TIMEOUT** or **port.STA_CHAR_TIMEOUT** or if **error ≠ nil** is true.
error	**nil**, if no error has appeared.
	port.ERR_PARITY_ERROR (2) A parity error has been detected during reception.
	port.ERR_FRAMING_ERROR (4) A telegram frame error has been detected, such as a start – stop bit error.
	port.ERR_BREAK_DETECTED (1) A break has been received on the serial interface.
id	**nil**, if the receive request has not been supplied with an **id** at its generation
	id, which has been supplied to the receive request at generation within the Request FIFO.
Status-/Error code in „**lasterror**“	**err.PORT_FIFO_EMPTY** (0x40800204) The Confirmation FIFO Queue currently does not contain any completely processed request
	err.PORT_NO_CONFIRMATION (0x 40800217) A completely processed request is located at the Confirmation FIFO Queue, but this is no send request. Please always fetch the processed requests exactly in the order of their activation.
	err.PORT_NOT_OPEN (0xC0800213) The specified port has not been opened.
	err.PORT_WRONG_MODE (0xC0800203) The port was opened in character mode and can’t be used requested with block mode.

8.3.7 :PortAbort

:PortAbort()	
This function deletes all blocks from the Request and Confirmation FIFO Queue. Processing the currently executed block is immediately aborted.	
Arguments:	none
Return value:	none
Status-/Error code in „**lasterror**“	**err.PORT_NOT_OPEN** (0xC0800213) The specified port has not been opened.
	err.PORT_WRONG_MODE (0xC0800203) The port was opened in character mode and can’t be used requested with block mode.

9 Serial Communication in Character Mode

In character mode the UART operates in full duplex mode. This mode is able to send and receive data simultaneously (not for RS-485). Also for RS-485 the communication in character mod is possible for the script, but works on the serial line as in block mode.

The size of both reception and transmission FIFOs is 256 byte each.

The following graphics illustrates data processing:

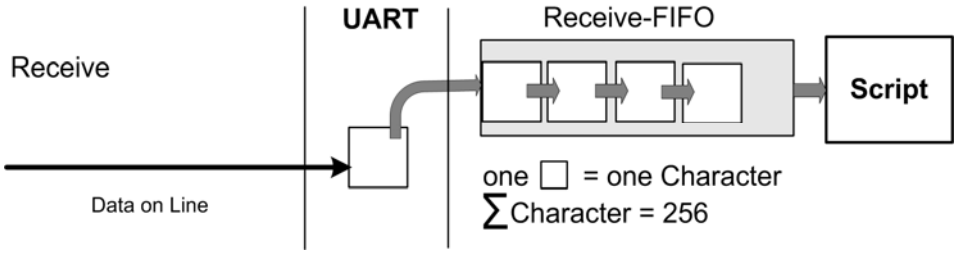


Figure 12: Processing on data reception in character mode.

In FIFO Mode up to 256 characters are recorded. To avoid an overflow of the data reception FIFO, the FIFO must be emptied within this data reception time period by netSCRIPT.

If there is an overflow at the data reception FIFO, all forthcoming characters will be ignored. In this case at the next read access with „PortGetChar“ the error message „ERR_RX_FIFO_OVERFLOW“ will be set.

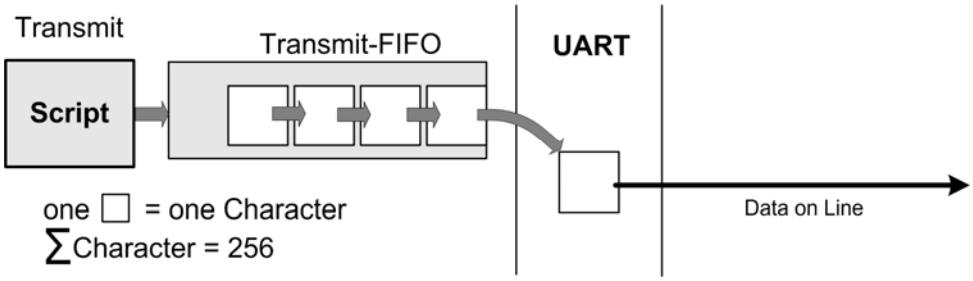


Figure 13: Processing on data transmission in character mode.

The setting of the transmission FIFO allows up to 256 characters to be stored there, which can sequentially be put out on the line. If a script function tries to write more characters into the FIFO as there is free space within the transmission FIFO, the write request will be denied from the FIFO.

Behavior at break of line:

If at the input line (RxD) there is level 0 for 11 bit periods, i.e. on reception of 11 zero-bits (which does not occur at usual data transmission), the break flag is set and one zero character is stored.

In character mode this character with the break flag is immediately visible at the FIFO.

At data transmission, data are sent which will never be received. If RTS/CTS handshake has been activated and the received level on CTS is not equal to the parameter Handshake Polarity nothing is sent.

9.1 Transmission- und Reception Functions

9.1.1 :PortGetChar

:PortGetChar([n])	
Reads **n** characters out of the FIFO data reception buffer (if available).	
Arguments: **string**	**n** is an optional argument indicating how many characters are to be read from the FIFO input memory. If **n** is not specified, one single character is read. If **n** is negative, all present characters are read. If no characters are available, then no error is generated.
Parameter **string**	If **n** characters are present in the data reception FIFO, these are returned string. If the data reception FIFO contains less than **n** characters, the value nil will be returned.
err	**port.ERR_PARITY_ERROR** (2) There is a parity error at least within one character. **port.ERR_FRAMING_ERROR** (4) A frame error has been detected during the reception of data. **port.ERR_BREAK_DETECTED** A break of the line has been detected. **port.ERR_RX_FIFO_OVERFLOW** (8) ...The FIFO had an overflow since the last call.
Status-/ error code in „**lasterror**“	**err.PORT_FIFO_EMPTY** (0x40800204) No input data available. Error only if **n**>0.
	err.PORT_INVALID_PARAMETER (0x40800216) Port: A function argument is of the wrong type, or its value is outside the allowed range (**n** exceeds 256)
	err.PORT_NOT_OPEN (0xC0800213) Port: Tried to call a function on a port which is not open.
	err.PORT_WRONG_MODE (0xC0800203) Port: Char mode function called in block mode, or vice versa.

Status-/ error code are store in variable „lasterror“.

9.1.2 :PortPutChar

:PortPutChar(str)	
Passes the string **str** to the transmission FIFO. If there is not sufficient space in the transmission FIFO, nothing at all will be passed. Only the number of data bits defined in the data table for the interface is passed.	
Arguments: **str**:	Accepts each number of arguments. Each argument can be either a character or a number. String arguments are sent byte wise. If less than eight data bits are configured, the upper bits of each byte within the string will be ignored. Numeric arguments will be converted into integer values, and the up to 8 least significant bits will be sent.
Return value:	**true**, if the request could be placed in the transmission FIFO.
	false, if the request could not be placed in the transmission FIFO.
Status-/Error code in „**lasterror**“	**err.PORT_NOT_OPEN** (0xC0800213) Port: Tried to call a function on a port which is not open.
	err.PORT_WRONG_MODE (0xC0800203) Port: Char mode function called in block mode, or vice versa.

Note: If all data within the reception or transmission FIFO shall be erased after a interruption of transmission, the port needs to be closed with function „:PortClose()“ and then reinitialized with function „:PortOpen()“.

10 Functions for the Communication with the superordinated I/O Network

A buffer of 1024 bytes for each direction is present in the device for the communication between netSCRIPT and the superordinated control unit. Both netScript and the superordinated control unit have to signal each other that new valid data are available and have to acknowledge each other that the data have been taken over. This process is named handshake. Two operating modes are possible and differ how this handshake is realized.

1. A read and write function is provided in **direct mode** to transfer any data. Here, no acknowledge for received data or new data is generated automatically. An acknowledge for received or a detection for new data has to be programmed, if necessary, in the script
2. A handshake procedure is implemented in **handshake mode** in the firmware and executed automatically from netSCRIPT. The script has not to take care for this handshake procedure. But this handshake procedure has to be programmed in the superordinated control unit.

   A read command is also available here, which however returns data only according to the handshake procedure. The take over of the data is acknowledged automatically.

   The write command only writes data when the last data written were acknowledged.

   A 24 byte large header is additionally available in both directions beside the data buffer to transfer the handshake information.

For both communication procedures the following is valid:

For the data transfer from the serial interface to the superordinated I/O network, which is usually the network of a controlling PLC and is named "Bus IO", is standardized. The same data transfer method is used for all networks like PROFIBUS, CANopen, DeviceNet or Real-Time Ethernet based systems like EtherCAT, EtherNet/IP or PROFINET IO.

For both transfer directions a header of 24 bytes and max. 1024 bytes for user data is reserved. Input is the direction from netSCRIPT to the superordinated control unit, because this is the input for the superordinated control unit. Output is the direction from the superordinated control unit to netSCRIPT, because this is the output for the superordinated control unit.

Byte	Output	Input	Meaning
0-23	Header for handshake protocol mode	Header for handshake protocol mode	Handshake flags, Length information, Error codes (for handshake mode only).
24-1047	Max. 1024 bytes data	Max. 1024 bytes data	User data

Table 33: netSCRIPT, Data Transfer Structure

The data header (byte 0 **...** 23) is also present in direct mode, but is not used.

The user has no direct access to this I/O data transfer memory. The data transfer is realized with functions for netSCRIPT instead, which handles the necessary access and synchronization. These netSCRIPT functions transfer only the user data.

The data amount of 1024 bytes is usually too much for I/O networks. It is not needed to transfer the complete user data to or from the superordinated control unit. The real data amount, which is transferred via the I/O network, is configured in the superordinated control unit.

Example:
A length of 64 bytes for output and a length of 70 bytes for input is set in the control unit for the netSCRIPT device. Reduced by the 24 bytes for the data header, netSCRIPT can receive 40 bytes user data per command and can send 46 bytes. The amount of user data is a result of the maximum data amount that should be transferred within one command via netSCRIPT to the IO interface.

Note: Please note that the structure of I/O data transfer memory is standardized within netSCRIPT, but the structure can be changed with the configuration software SYCON.net before the handover to the superordinated network. It is possible, for example, that the first register is mapped to the end of the transfer memory. Read more in the document User Manual, netTAP NT 100, Gateway Devices chapter 7.

10.1 Bus IO Communication – Start and End

To use the BusIO interface an instance must be created. Via this instance the additional functions of netSCRIPT can be used.

Even when the BusIO interface is set up correctly, user data only can be transferred, when the superordinated control unit has released the operation. This has to be done with the synchronization register and is checked by netSCRIPT automatically with the execution of the read and write functions.

10.1.1 BusIOOpen

The communication of the BusIO interface is initialized and resources are allocated.

BusIOOpen([Instanznr], [config])	
Initializes and opens the one BusIO instance.	
Arguments: **Instanznr**	If empty, the default instance (2) is provided. An instance can only be opened once.
config	Configuration table, see below.
Return values:	**instance,** when successful. (Has to be used for all additionally interface calls)
	nil, the instance could not be created.
Status/error code in the variable "lasterror"	**err.BUSIO_INVALID_CONFIG** (0xC0800303) One **config** parameter is invalid.
	err.BUSIO_NO_SUCH_INSTANCE (0x40800301) Instance could not be opened.
	err.BUSIO_ALREADY_OPEN (0x40800302) Instance is already opened.

Structure of the configuration table

Parameter	Key word	Range	Default
Direct mode	**directmode**	true/false (true = direct mode false = handshake mode)	false
Max. output length in direct mode (Gateway→ Lua)	**maxreadlen**	1..1024	1024
Max. input length in direct mode (Lua→ Gateway)	**maxwritelen**	1..1024	1024

Table 34: netSCRIPT, BusIO Configuration Table

The parameter of the BusIO configuration table are necessary for the direct mode only.

Example: open interface in handshake mode

```
Gw = BusIOOpen()
```

open interface on direct mode

```
Gw = BusIOOpen({directmode = true, maxreadlen = 100,
                maxwritelen = 50})
```

The amount of the internal copy process for the input and output buffer can be limited with the parameter maxreadlen / maxwritelen. Without this information always 1024 bytes are copied.

All read and write commands to this interface has to be done with the instance number returned from the **BusIOOpen** function as a prefix for the following functions.

Note:
By calling this function all input data (data to the superordinated control unit) are initialized with 0.

10.1.2 :BusIOClose

The connection to the BusIO interface is closed. All allocated resources are deallocated in the system. The communication on the superordinated network continues.

Use the function BusIOOpen() again to reopen the communication for netSCRIPT to the BusIO interface.

:BusIOClose ()	
Closes the BusIO instance.	
Arguments:	no
Return value:	no
Status/error code in the variable "lasterror"	**BUSIO_NOT_OPEN** (0x40800304) Instance could not be closed, because it was not opened before.

Example:

```
Gw:BusIOClose()
```

Note:
By calling this function all input data (data to the superordinated control unit) are initialized with 0.

10.2 Read / Write Functions for Direct Mode

The 24 byte large header in both directions are unconsidered in direct mode. No signaling is done automatically, if data are ready for transfer or data already have been taken over. This signaling has to be programmed in the script and in the superordinated control unit and within the 1024 data buffer.

10.2.1 :BusIOReadDirect()

:BusIOReadDirect([Offset[, Length]])	
In direct mode Reads a user data string from the superordinated control unit via the Bus IO interface.	
Argument **Offset**	Offset: Start position in the data buffer of the user data (0 .. maxreadlen-1, Default: 0) maxreadlen is the length of data to be transferred, which was defined with the function BusIOOpen.
Length	Length: 0.. maxreadlen-Offset, Default: maxreadlen-Offset maxreadlen is the length of data to be transferred, which was defined with the function BusIOOpen.
Return Values: **string**	**Outputdata**. If no data are transferred from the superordinated control unit after BusIOOpen, then a string with zero bytes is returned. **nil**: no error for offset/length
Status/error code in the variable "lasterror"	**BUSIO_BUFFER_LENGTH_EXCEEDED** (0x40800314) Invalid argument: Offset or lenght.
	BUSIO_NOT_OPEN (0x C0800304) The instance was not opened.
	BUSIO_WRONG_MODE (0xC0800306) This function was called for an interface, which was opened in handshake mode

Using this function the first 24 bytes of the interface are unconsidered. The sender gets **no** feedback, if the data were read and new data can be send.

10.2.2 :BusIOWriteDirect()

:BusIOWriteDirect(Offset, string, [fConfirm])	
In direct mode Writes the user data string into the send buffer to the superordninated control unit. This buffer is released only for data transfer, if **fConfirm** is NIL or true. As long as this argument is false the data of the send buffer is not transfered. This makes it possible to write the buffer in single steps.	
Arguments: **Offset**	Offset: Start position in the data buffer (0 .. maxwritelen-1) maxwritelen is the length of data to be transfered, which was defined with the function BusIOOpen.
string	Send data. Length 0..maxwritelen-Offset
fConfirm	**true** (Default): After execution of this command the data buffer is copied in the next IO cycle to the superordinated IO interface. **false:** Data buffer is not copied.
Return Value:	**true**, if data are stored in the buffer.
	false, if data were not stored in the buffer.
Status/error code in the variable "lasterror"	**BUSIO_BUFFER_LENGTH_EXCEEDED** (0x40800314) Invalid argument: Offset or lenght.
	BUSIO_NOT_OPEN (0xC0800304) The instance was not opened.
	BUSIO_WRONG_MODE (0xC0800306) This function was called for an interface, which was opened in handshale mode

10.3 Data Header for Handshake Mode

Each header in the transfer memory starts with a Synchronization register of 4 byte size. The superordinated control unit synchronizes the data flow via this register. Then a register of 4 byte size follows which contains the number of data for the data transfer. For the data direction 'Input' two more registers are present for error information to the superorninated control unit.

The data transfer via the BusIO interface is done via the following structure of the I/O data transfer memory.

Byte	Output (from PLC)	Input (to PLC)	Meaning
0-3	App_Handshake	Prot_Handshake	Synchronization register see section *Handshake and Initialization of the I/O Communication in Hand-shake* Mode on page 111
4-7	App_Tx_Bytecount	Prot_Rx_Bytecount	Number of valid bytes of user data .
8-11	Reserved	Prot_Rx_Error	Error number, program-mable, see function *:BusIOSetError()* on page 109.
12-15	Reserved	Prot_Tx_Error	Error number, program-mable, see function *:BusIOSetError()* on page 109.
16-19	Reserved	Reserved	-
20-23	Reserved	Reserved	-
24-1047	Max. 1024 bytes data	Max. 1024 bytes data	User data

Table 35: Communication Data structure

10.4 Read/Write Functions for Handshake Mode

10.4.1 :BusIORead

:BusIORead([fConfirm])	
In Handshake Mode: Reads the user data string (from the superordinated control unit) from the BusIO interface, when available and acknowledge it then.	
Arguments: **fConfirm**	States, if the read of the data of the control unit has to be acknowledged **true**, receipt of the data will be acknowledged. **false,** receipt of the data is not acknowledged. In this case the the 'receipt of the data' has to be used with the parameter value **true** with this function at a later time . The data transfer method does not allow the control unit to send new data, when old data are not acknowledged.
Return values:	**string**: read from the BusIO interface. If the length information in the header is more than the memory area (>1024 Byte), then **nil** is returned and lasterror is set to err.BUSIO_STRING_TOO_LONG. The receipt of the data is acknowledged nevertheless, except if fConfirm was set to false.
	nil, when error.
Status/error code in the variable "lasterror"	**err.BUSIO_RECEIVE_NO_DATA** (0x40800321) No data is available to be read.
	err.BUSIO_RECEIVE_DISABLED (0x40800322) The receipt of the data is not released from the control unit.
	err.BUSIO_STRING_TOO_LONG (0x 40800313) Error in the protocol header: invalid length information in the header from the superordinated control unit. Handshake is not acknowledged.
	err.BUSIO_NOT_OPEN (0x40800304) Instance could not be closed, because it was not opened before.
	BUSIO_WRONG_MODE (0xC0800306) This function was called for an interface, which was opened in direct mode

10.4.2 :BusIOWrite

:BusIOWrite(string)	
In Handshake Mode: Writes the user data string into the data transfer memory to the control unit, when an earlier data string was already acknowledged from the control unit.	
Arguments:	**string**, to be written to the BusIO interface
Return values:	**true**, when data could be transfered.
	false, when data could not be transfered.
Status/error code in the variable "lasterror"	**err.BUSIO_SEND_NOT_READY** (0x40800311) The receipt of the last send data was not acknowledged from the control unit yet.
	err.BUSIO_SEND_DISABLED (0x40800312) The send of the data is not released from the control unit.
	err.BUSIO_STRING_TOO_LONG (0x40800313) A buffer overflow happened. String to long. More than 1024 byte.
	err.BUSIO_NOT_OPEN (0x40800304) Instance could not be closed, because it was not opened before.
	BUSIO_WRONG_MODE (0xC0800306) This function was called for an interface, which was opened in direct mode

Example for an simple data transfer:

```
Gw = BusIOOpen()
Gw:BusIOWrite(„Hello“)
```

The string „Hello“ is written to the BusIO interface. The superordinated control unit gets the signal 'new data available’.

The function **BusIOOpen()** has to be called only once in the script. Because of that the interface is usable until function **Gw: :BusIOClose ()** is called.

10.5 Reset Command in Handshake Mode

The control unit is able to generate a reset command to netSCRIPT via the BusIO interface. If this function is intended to be used, then the netSCRIPT program has to check regulary to see if this command was activated by the control unit.

The netSCRIPT programmer is responsible for the program to be done for a reset. Of course the command can also be used for other purposes.

The position of the Reset bit in the head data is in section 10.9.1 described.

10.5.1 :BusIOIsReset

:BusIOIsReset()	
In Handshake Mode: Reads the **APP_HS_RESET_CMD** bit of the synchronization register. See section Superordinated Control Unit to netSCRIPT on page 112. Checks whether an unconfirmed reset command exists. The reaction to this has to be realized by the netSCRIPT programmer.	
Arguments:	no
Return value: **bool**	**true,** an unconfirmed reset command is available.
	false, no unconfirmed reset command is available.
Status/error code in the variable "lasterror"	**err.BUSIO_NOT_OPEN** (0x40800304) Instance could not be closed, because it was not opened before.
	BUSIO_WRONG_MODE (0xC0800306) This function was called for an interface, which was opened in direct mode

10.5.2 :BusIOResetDone

:BusIOResetDone()	
In Handshake Mode: Writes the **PROT_HS_RESET_ACK** bit of the synchronization register. See section netSCRIPT to Superordinated Control Unit on page 113. The receipt of the reset command is acknowledged. In general this function is called, when within the netSCRIPT program the reset was done.	
Arguments:	no
Return values:	no
Status/error code in the variable "lasterror"	**BUSIO_NOT_OPEN** (0x40800304) Instance could not be closed, because it was not opened before.
	BUSIO_WRONG_MODE (0xC0800306) This function was called for an interface, which was opened in direct mode

10.6 Ready Signal to the Control Unit in Handshake Mode

10.6.1 :BusIOSetRun

:BusIOSetRun(bool)	
In Handshake Mode: Signals the end of the initialization to the superordinated control unit. This function should be the first used within the netSCRIPT program after the initialization. The signal **PROT_HS_RUN_IND** is transfered to the superordinated control unit. See also section 10.9.1.2 on page 113.	
Arguments: **bool**	**true,** sets the status to 'ready'
	false, sets the status to 'not ready'
Return value:	no
Status/error code in the variable "lasterror"	**BUSIO_NOT_OPEN** (0x40800304) Instance could not be closed, because it was not opened before.
	BUSIO_WRONG_MODE (0xC0800306) This function was called for an interface, which was opened in direct mode

10.7 Report an Error to the superordinated Control Unit in Handshake Mode

It is possible to transfer error information to the control unit via the BusIO interface. Two error registers are available in the transfer memory for this, which can be written by netSCRIPT functions.

One register is defined for receive errors the other is defined for send errors.

10.7.1 :BusIOSetError()

:BusIOSetError(Direction, Errorflag, [Errorcode])		
In Handshake Mode: Sets or clears an error in the transfer memory to the control unit. See section Structure for Input - Data netSCRIPT to Control Unit on page 110 and bit **PROT_HS_TX_ERROR_IND** in the synchronization register respectively bit **PROT_HS_RX_ERROR_IND**. See section netSCRIPT to Superordinated Control Unit on page 113.		
Arguments:	**Direction**	**send**, sets or clears the error in the register (byte 4-7)
		receive, sets or clears the error in the register (byte 12-15)
	Errorflag	**nil,** leave error unchanged
		true, Set flag to 1 and signal it
		false, Set flag to 0
	Errorcode	**nil** or omit = leave error unchanged
		32 Bit unsigned error value. Can be set independently from the state of the error flag
Return values:		no
Status/error code in the variable "lasterror"		**BUSIO_INVALID_PARAMETER** (0x40800305) The values to be transfered are not reasonable.
		BUSIO_NOT_OPEN (0x40800304) Instance could not be closed, because it was not opened before.
		BUSIO_WRONG_MODE (0xC0800306) This function was called for an interface, which was opened in direct mode

10.8 I/O Data Structure for the Transfer to and from the Control Unit in Handshake Mode

10.8.1 Structure for Output - Data from the Control Unit to netSCRIPT

Byte	**Signal**	**Readable with**
0-3	Output synchronization register	No access, is read by the netSCRIPT functions automatically
4-7	Number of user data to be transferred	No access, is read by the netSCRIPT functions automatically
8...23	Not used	Not used
24-1047	Output User data	Function **:BusIORead()**

Table 36: netSCRIPT – Structure for Output – Data from the Control Unit

See section Superordinated Control Unit to netSCRIPT on page 112 for the structure of the synchronization register.

10.8.2 Structure for Input - Data netSCRIPT to Control Unit

Byte	**Signal**	**Writable with**
0-3	Input synchronization register	No access, is written by the netSCRIPT functions
4-7	Number of user data to be transferred	No access, is written by the netSCRIPT functions
8-11	Error register (receive error) to transfer error information	Writable with the netSCRIPT function **:BusIOSetError()**.
12-15	Error register (transmit error) to transfer error information	Writable with the netSCRIPT function **:BusIOSetError()** .
16..23	Not used	Not used
24-1047	Input User data	Function **:BusIOWrite()**

Table 37: netSCRIPT – Structure for Input – Data to the Control Unit

See section netSCRIPT to Superordinated Control Unit on page 113 for the structure of the synchronization register.

10.9 Handshake and Initialization of the I/O Communication in Handshake Mode

The data transfer is between the control unit and netSCRIPT is organized by a transfer method within the I/O data transfer memory.

The basic idea of this method is that for each action a pair of bits is used in both synchronization registers. One bit is used to request an action the other is used to acknowledge the action. One is located in the input synchronization register the other in the output synchronization register.

One action is requested by setting the command bit unequal to the acknowledge bit. The other side acknowledges this request by setting the acknowledge bit equal to the command bit.

10.9.1 Structure of the Synchronization Register in the I/O Data

10.9.1.1 Superordinated Control Unit to netSCRIPT

Structure of the synchronization register of the control unit to netSCRIPT:

Bit-No.	Name	Read by function
0	**APP_HS_TX_CMD**	:BusIORead
1	**APP_HS_RX_ACK**	:BusIOWrite
2 ... 5	Not used	
6	**APP_HS_TX_ENABLE_CMD**	automatisch
7	**APP_HS_RX_ENABLE_CMD**	automatisch
8 ... 14	Not used	
15	**APP_HS_RESET_CMD**	:BusIOIsReset
16 ... 31	Not used	

Table 38: netSCRIPT – Synchronization Register to netSCRIPT

Where:

APP_HS_TX_CMD

Command from the control unit to send output data to netSCRIPT. Is checked by the function **:BusIORead()** automatically.

APP_HS_RX_ACK

Acknowledge received input data in the control unit. The bit is checked automatically within netSCRIPT by the function **:BusIOWrite()**.

APP_HS_TX_ENABLE_CMD

Enable the output user data transfer from the control unit to net-SCRIPT. If this bit is not set, then netSCRIPT will not evaluate requested commands via the bit **APP_HS_TX_CMD.**

APP_HS_RX_ENABLE_CMD

Enable the input user data transfer from netSCRIPT to the control unit. If this bit is not set, then netSCRIPT can not requested commands via the bit **APP_HS_RX_CMD.**

APP_HS_RESET_CMD

Is queried by the function **:BusIOIsReset()**.

	RESET_CMD	RESET_ACK
Initial state	0	0
Superordinated control unit requests a reset. This is queried by **:BusIOIsReset()**	1	0
In the script, function **:BusIOResetDone()** is called	1	1
Superordinated control unit has taken back the reset command RESTE_CMD.	0	1
netSCRIPT takes back the reset acknowledge RESET_ACK automatically.	0	0

The communication to and from the serial interface within netSCRIPT can used independently from the I/O synchronization mechanism. There is no causal coherence between both interfaces.

10.9.1.2 netSCRIPT to Superordinated Control Unit

Structure of the synchronization register from netSCRIPT to the control unit:

Bit-Nr.	Name	Written by function
0	**PROT_HS_TX_ACK**	:BusIORead
1	**PROT_HS_RX_CMD**	:BusIOWrite
2	Not used	
3	**PROT_HS_RUN_IND**	:BusIOSetRun
4	**PROT_HS_TX_ERROR_IND**	:BusIOSetError
5	**PROT_HS_RX_ERROR_IND**	:BusIOSetError
6	**PROT_HS_TX_ENABLE_ACK**	automatically
7	**PROT_HS_RX_ENABLE_ACK**	automatically
8 ... 14	Not used	
15	**PROT_HS_RESET_ACK**	:BusIOResetDone

Table 39: netSCRIPT – Synchronization Register to the Superordinated Control Unit

Where:

PROT_HS_TX_ACK

Acknowledge bit from netSCRIPT to the superordinated control unit for sent output data. Is used by the function **:BusIORead()** automatically.

PROT_HS_RX_CMD

Command for input user data from netSCRIPT to the superordinated control unit. The bit is checked automatically within netSCRIPT by the function **:BusIOWrite()**.

PROT_HS_RUN_IND

The netSCRIPT program signals 'Ready' and the end of initialization. if used by the function **:BusIOSetRun()**.

PROT_HS_TX_ERROR_IND

0: ok, 1: Error (Error number see error register in bytes 12-15). Is written by the function **:BusIOSetError()**.

This bit is set from netSCRIPT independently from **APP_HS_RX_ENABLE_CMD**.

PROT_HS_RX_ERROR_IND

0: ok, 1: Error (Error number see error register in bytes 8-11). Is written by the function **:BusIOSetError()**.

PROT_HS_RESET_ACK

The reset command of the control unit is acknowledged. Is written by the function **:BusIOResetDone()**.

PROT_HS_TX_ENABLE_ACK

Acknowledges the enabling of output data transfer from the control unit to netSCRIPT. This bit is controlled from netSCRIPT automatically.

PROT_HS_RX_ENABLE_ACK

Acknowledges the enabling of input data transfer from netSCRIPT to the control unit. This bit is controlled from netSCRIPT automatically.

10.9.2 Initializing of the Communication

Start of the communication

Step	Action: **Start of the communication, initialization is done by the superordinated control unit.**	**Handshake-send byte of the superordinated control unit** 7 6 5 4 3 2 1 0	**Handshake-receive byte of the superordinated control unit** 7 6 5 4 3 2 1 0
1	Memory after power on. netSCRIPT signals 'ready'.	0 0 x x x x 0 0	0 0 0 0 X 0 0 0
2	The superordinated control unit starts the communication with netSCRIPT. By setting bit 6 and 7 allows netSCRIPT to communicate with the superordinated control unit.	1 1 x x x x 0 0	0 0 0 0 X 0 0 0
3	netSCRIPT receives the handshake flags from the superordinated control unit. These release the following actions: The send direction to the superordinated control unit is enabled based on bit 7. The receive direction for netSCRIPT via BusIO is enabled based on bit 6.	1 1 x x x x 0 0	
4	The receipt is acknowledges by netSCRIPT. The data transfer to the superordinated control unit can start.		1 1 0 0 X 0 0 0
	After the superordinated control unit has received the acknowledgment from netSCRIPT for 'send and receive ready' it can send data to netSCRIPT.		1 1 0 0 X 0 0 0

Table 40: netSCRIPT – Initialising of the Communication

➪ After enabling the communication the control unit or netSCRIPT can now start the communication

Note: The value of bit 3 of the handshake receive byte of the superordinated control unit (in the table above marked with "X") depends from the function call **:BusIOSetRun**. If this function called within netSCRIPT this bit has value 1, otherwise 0.

10.9.3 Acknowledgment of the Processing between the Superordinated Control and netSCRIPT

An acknowledgment of the receipt is expected from the receiver for each data transfer from netSCRIPT to the superordinated control unit and visa versa. As long as this receive acknowledgment is not available, no further data can be send to the receiver.

This handshake procedure is described in the following section for both directions.

Note: In the following tables an "x" marks an undefined bit position and an "X" marks a defined but not relevant bit position.

10.9.3.1 Superordinated Control Unit to netSCRIPT

The value of a bit marked with "x" don't cares.

Step	Action: **The superordinated control unit sends data to netSCRIPT**	**Handshake-send byte of the superordinated control unit / receive byte of the gateway** 7 6 5 4 3 2 1 0	**Handshake-receive byte of the superordinated control unit / send byte of the gateway** 7 6 5 4 3 2 1 0
0	State before beginning to send (and after initialization). Handshake bit 0 in the receive and send buffer has the same value. → It is possible to send data to netSCRIPT.	1 1 x x x x X 0	1 1 x x x x X 0
1	The superordinated control unit has provided user data in its send buffer, sets the length information in the header and sets handshake bit 0.	1 1 x x x x X 1	
2	The data is transfered to netSCRIPT	1 1 x x x x X 1	
3	As long as bit 0 of the handshake in the send and receive buffer is unequal, it is not allowed for the superordinated control unit to send new user data to the gateway.	1 1 x x x x X 1	1 1 x x x x X 0
4	netSCRIPT recognizes that bit 0 of the handshake in the send and receive buffer is unequal, that new user data is available from the superordinated control unit.	1 1 x x x x X 1	1 1 x x x x X 0
5	When netSCRIPT unit has taken the user data from the receive buffer, netSCRIPT sets bit 0 in the handshake send byte and acknowledges		1 1 x x x x X 1
6	The handshake send byte of the gateway is transfered to the superordinated control unit.		1 1 x x x x X 1
7	The superordinated control unit recognizes that handshake bit 0 in the send and receive buffer is equal → netSCRIPT has received the data and is ready to receive new user data	1 1 x x x x X 1	1 1 x x x x X 1
8	The superordinated control unit provides new user send data for netSCRIPT in its send buffer, sets the length information in th header and sets the handshake bit 0 to zero.	1 1 x x x x X 0	
9	The data is transfered to netSCRIPT	1 1 x x x x X 0	
10	As long as bit 0 of the handshake in the send and receive buffer is unequal, it is not allowed for the superordinated control unit to send new user data to the gateway.	1 1 x x x x X 0	1 1 x x x x X 1
11	netSCRIPT recognizes that bit 0 of the handshake in the send and receive buffer is unequal, that new user data is available.	1 1 x x x x X 0	1 1 x x x x X 1
12	When netSCRIPT has taken the user data from the receive buffer, the superordinated control unit sets bit 0 to zero in the handshake send byte and acknowledges		1 1 x x x x X 0
13	The handshake send byte of netSCRIPT is transfered to the superordinated control unit.		1 1 x x x x X 0
14	The superordinated control unit recognizes that handshake bit 0 in the send and receive buffer is equal → netSCRIPT has received the data and is ready to receive new user data		1 1 x x x x X 0
15	Handshake bit 0 in the receive and send buffer has the same value. This is the same state as in step 0. Now this procedure can start from the beginning.	1 1 x x x x X 0	1 1 x x x x X 0

10.9.3.2 netSCRIPT to the superordinated Control Unit

The value of a bit marked with x does not matter.

Step	Action: **netSCRIPT sends data to the superordinated control unit**	**Handshake-send byte of the superordinated control unit / receive byte of netSCRIPT** 7 6 5 4 3 2 1 0	**Handshake-receive byte of the superordinated control unit / send byte of netSCRIPT** 7 6 5 4 3 2 1 0
0	State before beginning to send. Handshake bit 1 in the receive and send buffer has the same value. → It is possible to send data to the superordinated control unit.	1 1 x x x x 0 X	1 1 x x x x 0 X
1	netSCRIPT has provided user data to be sent, sets the length information in the header and sets handshake bit 1.		1 1 x x x x 1 X
2	The data is transfered to the superordinated control unit		1 1 x x x x 1 X
3	As long as bit 1 of the handshake in the send and receive buffer is unequal, netSCRIPT is not allowed to send new user data to the superordinated control unit.	1 1 x x x x 0 X	1 1 x x x x 1 X
4	The superordinated control unit recognizes that bit 1 of the handshake in the send and receive buffer is unequal, that new user data from netSCRIPT is available.	1 1 0 0 0 0 0 X	1 1 x x x x 1 X
5	When the superordinated control unit has taken the user data from the receive buffer, the superordinated control unit sets bit 1 in the handshake send byte and acknowledges	1 1 0 0 0 0 1 X	
6	The handshake send byte of the superordinated control unit is transfered to netSCRIPT.	1 1 0 0 0 0 1 X	
7	netSCRIPT recognizes that handshake bit 1 in the send and receive buffer is equal → The superordinated control unit has received the data and is ready to receive new user data	1 1 0 0 0 0 1 X	1 1 x x x x 1 X
8	netSCRIPT provides new user send data, set the length information in the header and sets the handshake bit 1 to zero.		1 1 x x x x 0 X
9	The data is transfered to the superordinated control unit		1 1 x x x x 0 X
10	As long as bit 1 of the handshake in the send and receive buffer is unequal, netSCRIPT is not allowed to send new user data to the superordinated control unit.	1 1 0 0 0 0 1 X	1 1 x x x x 0 X
11	The superordinated control unit recognizes that bit 1 of the handshake in the send and receive buffer is unequal, that new user data from netSCRIPT is available.	1 1 0 0 0 0 1 X	1 1 x x x x 0 X
12	When the superordinated control unit has taken the user data from the receive buffer, the superordinated control unit sets bit 1 to zero in the handshake send byte and acknowledges	1 1 0 0 0 0 0 X	1 1 x x x x 0 X
13	The handshake send byte of the superordinated control unit is transfered to netSCRIPT.	1 1 0 0 0 0 0 X	1 1 x x x x 0 X
14	netSCRIPT recognizes that handshake bit 1 in the send and receive buffer is equal → The superordinated control unit has received the data and is ready to receive new user data		
15	Handshake bit 1 in the receive and send buffer has the same value. This is the same state as in step 0. Now this procedure can start from the beginning.	1 1 x x x x 0 X	1 1 x x x x 0 X

11 Error-Handling

11.1 About “lasterror”

Error information of the interface communication (port and Bus-IO) is deposited in the variables **”lasterror”**.

Within netSCRIPT, the treatment of the errors occurs purely symbolically like the following example points. To use the error codes in this script is preceded by a prefix "err." (without quotes).

```
if lasterror == err.BUSIO_SEND_NOT_READY then

…

end
```

or

```
If uart:PortSend(“hello”) then

elseif lasterror == err.PORT_FIFO_FULL then

elseif lasterror == err.PORT_NO_BUFFER then

end
```

If the error values of other purposes are used, for example, outside from netSCRIPT, the values reflect 32 bits of value.

As 8-figure hex numbers, their meaning is described in the following table. In it containing error numbers without entry in the column error code, are abnormal termination errors (Errors one start of the script processing prevent), which are indicated in the diagnostic ofSYCON.net as an error message.

If the hex number begins with 0xC, the Script processing is stopped (with the exception of the case where the call occurred with the function **pcall**). These messages are marked in the following table in the column “stop” with ▼.

If the error code begins with 0x4, the script processing is not stopped. The error can be treated in the Script further.

11.1.1 Error Codes in “lasterror”

In the variable **lasterror.**

Error number in hex	Error code	Stop	Meaning
C0800002		▼	Storage allocation in the netScript task has missed.
C0800080		▼	Lua start failed.
C0800081		▼	No script file available. Script file could not be loaded. Script file as "startup" does not select.
C0800082		▼	The value of __CYCLIC_FUNCTION is not a function.
C0800083		▼	An error has appeared in an Error Handler.
C0800084		▼	A storage allocation in the Lua interpreter has missed.
C0800085		▼	A Lua panic has appeared. (The error has probably appeared beyond the script).
C0800101	LUA_ERROR	▼	Lua error (from Lua even released error).
C0800201	err.PORT_INVALID_CONFIG	▼	Faulty configuration parameter.
C0800202	err.PORT_INVAL_PORT	▼	Wrong port number, port does not exist.
C0800203	err.PORT_WRONG_MODE	▼	It was tried to use a port in a mode, but the port was not initialized for the requested mode.
40800204	err.PORT_FIFO_EMPTY		No dates of receipt available.
40800205	err.PORT_FIFO_FULL		Not of enough free FIFO memory available.
C0800206	err.PORT_XC_INIT_FAILED	▼	The initialization has missed.
40800210	err.PORT_NO_BUFFER		No free memory block availably (perch mode).
40800211	err.PORT_NO_SUCH_PORT		Call of a non-existent port authority.
C0800213	err.PORT_ NOT_OPEN	▼	It was tried to access a port which was not open.
C0800214	err.PORT_NO_UARTDB	▼	No UART Configurations data bank was found.
C0800215	err.PORT_PARSING_UARTDB	▼	Errors with interpret to the UART Configurations data bank.
40800216	err.PORT_INVALID_PARAMETER		Faulty parameters by sending call and receipt call.
40800217	err.PORT_NO_CONFIRMATION		End password of the telegram not found.
40800218	err.PORT_STRING_TOO_LONG		The string to be transmitted is too long (for sending buffer).
40800212	err.PORT_ALREADY_OPEN		It was tried to open a port which is already opened.
40800302	err.BUSIO_ALREADY_OPEN		Instance is already used.
40800301	err.BUSIO_NO_SUCH_INSTANCE		Instance does not exist.
C0800303	err.BUSIO_INVALID_CONFIG	▼	Faulty configuration data (does not appear currently).
C0800304	err.BUSIO_NOT_OPEN	▼	Instance not opened.
C0800305	err.BUSIO_INVALID_PARAMETER	▼	Wrong parameter.
C0800306	err.BUSIO_WRONG_MODE		A function for direct mode was called in handshake mode or visa versa.
40800311	err.BUSIO_SEND_NOT_READY		Write: Input buffer not free.
40800312	err.BUSIO_SEND_DISABLED		Write: RxEnableCmd is not put.
40800313	err.BUSIO_STRING_TOO_LONG		Write: String too long for send buffer. Read: Invalid length in header.
40800314	err.BUSIO_BUFFER_LENGTH_EXCEEDED		The function :BusIOReadDirect or :BusIOWriteDirect are used with invalid length or offset values for the arguments.

Error number in hex	Error code	Stop	Meaning
40800321	err.BUSIO_RECEIVE_NO_DATA		Read: no new data.
40800322	err.BUSIO_RECEIVE_DISABLED		Read: TxEnableCmd is not put.
C0800401	err.UTIL_INVALID_PARAMETER	▼	Invalid parameter with purpose type, ENDIAN, LED or Identifier.
40800402	err.UTIL_OUT_OF_RANGE		Number does not lie in the worth area of the purpose type.
C0800411	err.UTIL_STRING_TOO_LONG	▼	String from the variable list SYCON is too long.
C0800410	err.UTIL_UNKNOWN_TYPE	▼	It is an unknown variable type in the variable list SYCON.
40800403	err.UTIL_WRONG_SIZE		The value does not have the correct size for the given type.

Table 41:netSCRIPT - „lasterror“ Error Codes

The errors which are marked with the character ▼ can be indicated only in the SYCON.net. See section 12.1.2 page 124. The processing of the script was stopped.

The execution of the script continues for all other error cases.

11.2 Return values for Status and Error of the Port Functions

For the functions: **PortIsReceiveDone** and **PortIsExchangeDone**

11.2.1 Possible values of confirmation status:

"Status Code"	Return-value	Meaning
port.STA_ACK_TIMEOUT	3	Reception ended after the acknowledge delaytime, no characters have been received (an empty string is returned)
port.STA_SIZE_REACHED	2	Reception has ended because the maximum number of characters given in the request has been received.
port.STA_PATTERN_MATCH	1	Reception has ended because the the end pattern has been found in the input.
port.STA_CHAR_TIMEOUT	4	Reception has ended because the pause after the last received character was longer than chardelaytime

Example for an error handling:

```
sta, data, rxerr = port:PortIsReceiveDone()
if sta == port.STA_PATTERN_MATCH then
…
end
```

11.2.2 Possible values for receive error:

"Error Code"	Return-value	Meaning
port.ERR_PARITY_ERROR	2	At least one character has a parity error
port.ERR_FRAMING_ERROR	4	A framing error has been detected during reception
port.ERR_BREAK_DETECTED	1	A break condition has been detected during reception
port.ERR_RX_FIFO_OVERFLOW	8	Char mode only: The FIFO has overflowed since the last call to PortGetChar

Only the most severe receive error is reported:
if a Framing error is reported, characters may also have had parity errors;
if a break was detected, framing or parity errors may also have occurred,
if a FIFO overflow was detected, all other errors may also have occurred.

12 Troubleshooting

Depending on the type of error analysis, there are two possibilities.

- For Device errors or communication errors on the bus, there is the diagnostic in Sycon.
- For script errors, there is the netSCRIPT Debugger.

12.1 Diagnostics in SYCON.net

As a rule, the USB diagnostic interface of the target device is used to communicate with the configuration tool SYCON.net.

If the SYCON software is connected to the diagnostic interface of the target device, the state of the netSCRIPT function in the device can be determined.

This error diagnostic is only possible if the logical connection to the device is built up by the SYCON.net.

12.1.1 Diagnostic

➢ Click with the right mouse button on the icon of the device which should be analyzed.

➪ The following context menu opens.

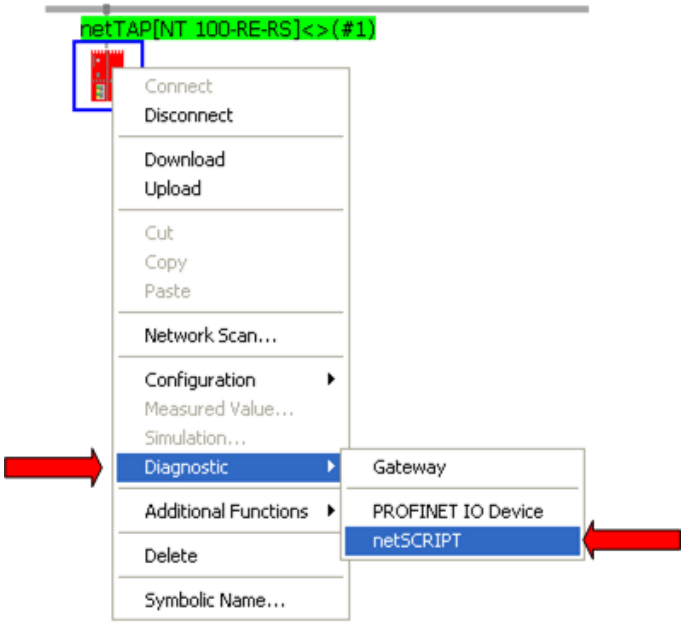


Figure 14: SYCON - Diagnostic Menu

➢ Select “Diagnostic > netSCRIPT”.

➪ The window opens with general diagnostic.

12.1.2 General Diagnostic - Stop Error in SYCON.net

Because a stop error stops the processing of a script, no error analysis with the debugger is possible. These errors are displayed in SYCON and can be used for error analysis.

➢ Under the menu of the netSCRIPT device “Diagnosis > General Diagnosis” you get the following information:

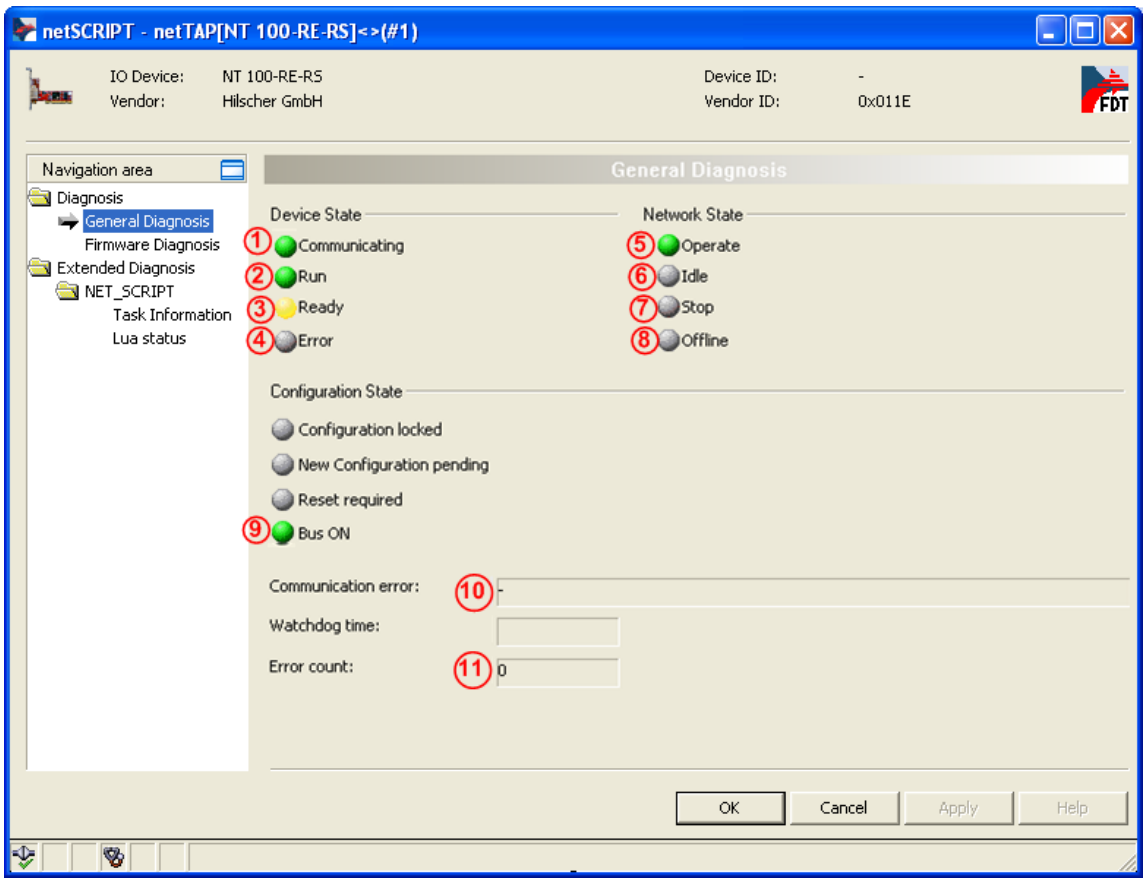


Figure 15: SYCON - Diagnostic General Diagnosis

The displays have the following meaning

Indication	Color	Meaning
Device State		
Communication ①	green	Script is running: Displays that the script is executed.
	grey	Script is not running /stopped
Run ②	green	Script loaded.
	gray	No script loaded
Ready ③	yellow	netSCRIPT task initialized successfully
	grey	Initialization error
Error ④	red	Script stops: A error has occured which prevents the script processing. Under communication error, an error number and an error text is entered.
	grey	Script is running: No error since last restart of the script.
Network State		
Operate ⑤	green	Operates: Indicates that the script is processed cyclically.
	gray	Script is not processed cyclically.
Idle ⑥	yellow	Script stops: netSCRIPT waits for commands from the Debugger Reasons: After sings step; break point reached; Script re-loaded; Error.
	gray	netSCRIPT NOT running in debug mode.
Stop ⑦	red	Stop: Script was stopped because of an error.
	gray	Script is not stopped.
Offline ⑧	yellow	Offline: no script loaded, or netSCRIPT- Initialization error
	gray	Script NOT offline.
Configuration State		
Configuration locked	yellow	NOT used.
	gray	
New Configu-ration pending	yellow	NOT used.
	gray	
Reset required	yellow	NOT used.
	gray	
Bus ON ⑨	green	Bus ON: Shows that netSCRIPT can communicate to the Mapping-Task (BUSIO Interface).
	gray	Communication to the Mapping-Task is not possible.

Table 42: Display General Diagnostic

Parameter	Meaning
Communication Error ⑩	Communication Error: Shows the last error message of the communication error since last power on. The error message is deleted when the script is reloaded or loaded from the debugger.
Watchdog time	Watchdog time: NOT used.
Error Count ⑪	Error Count: This field holds the total number of errors detected since power-up, respectively after reset. The protocol stack counts all sorts of errors in this field no matter if they were network related or caused internally. This error counter is not deleted when the script is loaded from the debugger.

Table 43: Parameter General Diagnostic

12.1.3 Firmware Diagnosis

In the dialog **Firmware Diagnosis** the actual task information of the firmware is displayed.

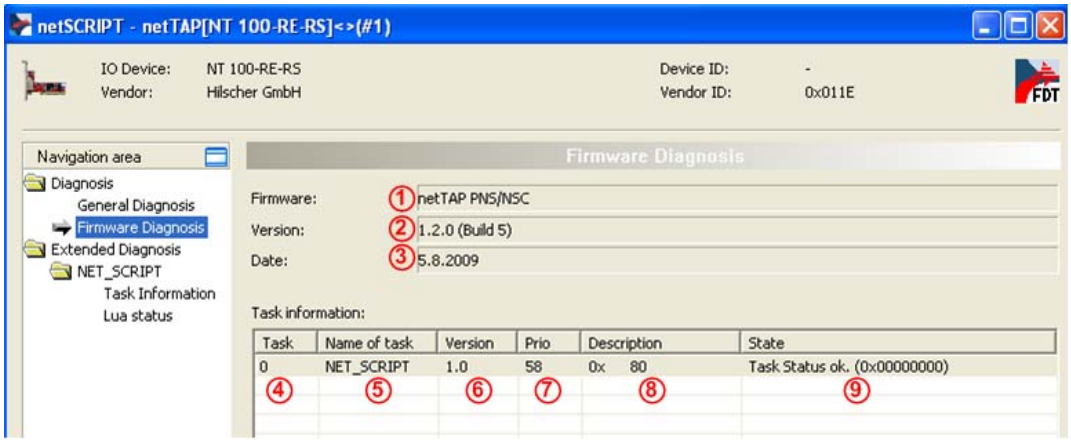


Figure 16: SYCON - Diagnostic Firmware Diagnostic

① The name of the loaded firmware.

② The version number of the loaded firmware.

③ The date of the firmware creation

Task Information:

The table **Task Information** is listing the task information of the single firmware tasks.

No.	Column	Meaning
④	Task	Task number
⑤	Name of task	Name of the task
⑥	Version	Version of the task
⑦	Prio	Priority of the task
⑧	Description	Description of the task
⑨	Status	Status of the task

Table 44: Description Table Task Information

12.1.4 Task-Information

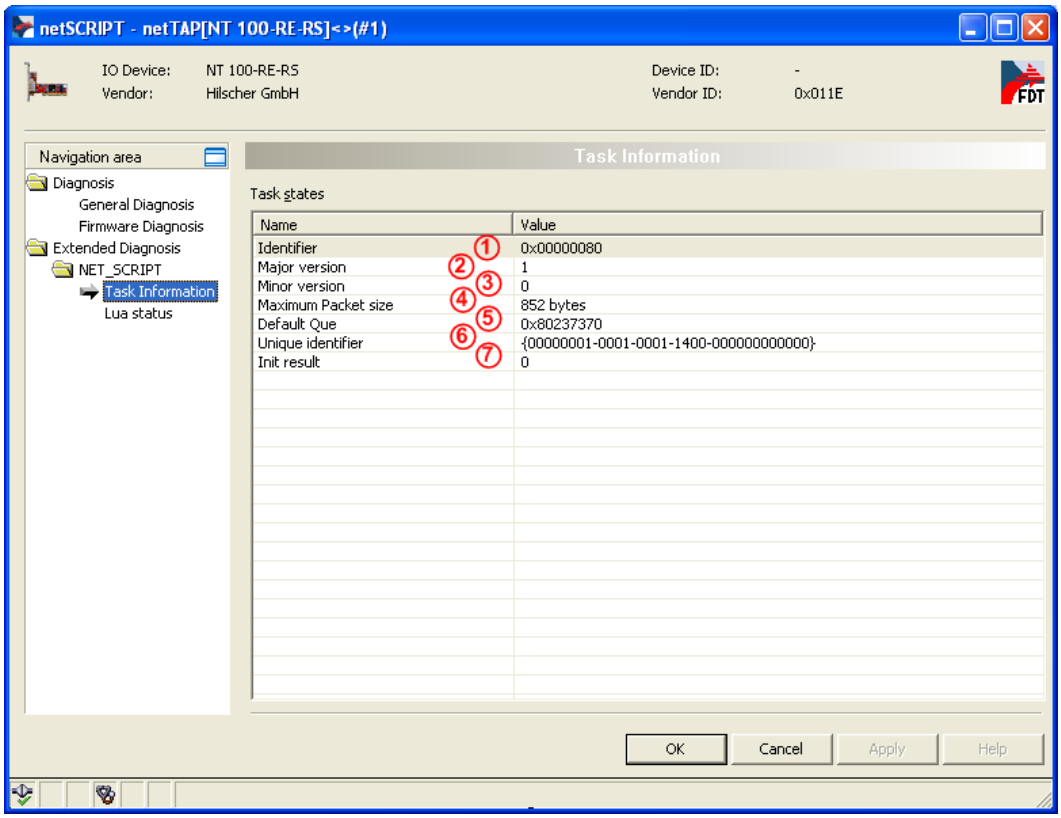


Figure 17: SYCON - Diagnostic Task- Information

In this window following information is displayed:

① Identifier: The task number of the netSCRIPT task.

② Identifier: The task number of the netSCRIPT task.

③ Internal version number.

④ Internal packet size to the external communication.

⑤ Address of the diagnosis package queue.

⑥ Internal version number.

⑦ Information about the initialization state of the netSCRIPT function. (should always be 0).

12.1.5 Lua-Status

Under "Lua status" the following information is displayed:

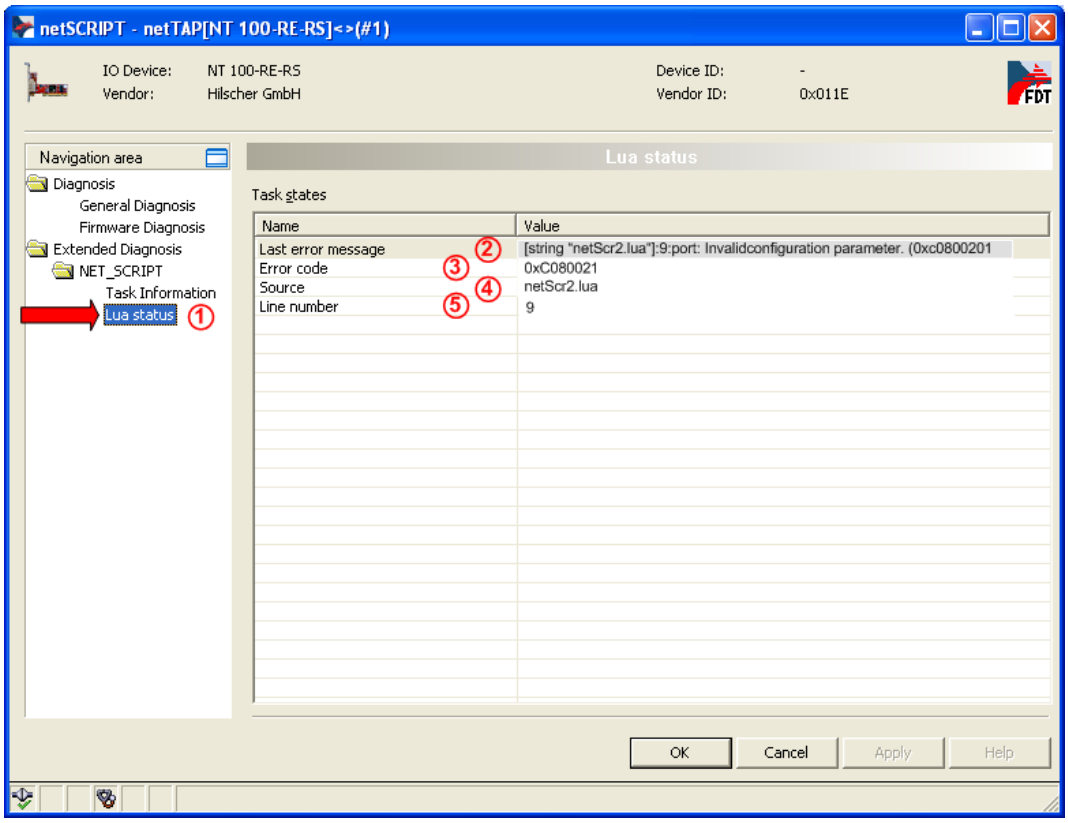


Figure 18: SYCON - Diagnostic Lua Status

① The diagnosis call path.

② The error text and the error number is displayed.

③ In this line you find the error code.

④ Here it is displayed in which script file the error has occured.

⑤ Here you find the line number of the code in the script file.

13 Troubleshooting netSCRIPT

A netSCRIPT debugger is available for error location in the execution of script programs. The debugger is independent and regardless of SYCON.net software under Windows and a so-called software debugger.

An additional driver in the device firmware of the netSCRIPT capable device communicates with the debugger. There, the driver checks in the cyclic netSCRIPT-processing loop automatically for commands of the debugger. The commands for example "stop" or "start" are integrated by the driver into the program execution and enable it to control the execution of the program.

Note: The proper function of the debugger requires that the netSCRIPT program be debugged always performs its process cyclically, otherwise the debugger is not able to communicate to the device driver software and debug the executed program.

If the script execution was stopped due to a serious error, only an error analysis may be possible with SYCON.net.

13.1 netSCRIPT Debugger

The Debugger serves to test the script in the netTAP device.

It enables debugging of the code line by line and shows the contents of the variables in every program step. It is also possible to change the source file and to transfer this into the device.

The debugger is able to open the current script running in the netTAP without a project, to load it from the target device and to display it. However, to change the Script it is necessary to have a project opened.

There is the possibility to set breakpoints.

Note: For a better clarity the areas of the debugger window are marked with letters and the buttons with numbers in the following figures.

13.1.1 Installation

To use the debugger, it is essential that the SYCON.net software communication drivers (for example USB or TCP/IP) for the target device are installed on the Windows platform.

How to install the debugger is described in the document “Software Installation - Gateway Solutions UM xx EN.pdf” which is on the DVD in the directory Documentation.

13.1.2 Start the Debugger

Start the debugger with „Start > All Programs > Hilscher GmbH > net-SCRIPT_Debugger > netSCRIPT_Debugger“ or via the desktop icon

The following screen appears.

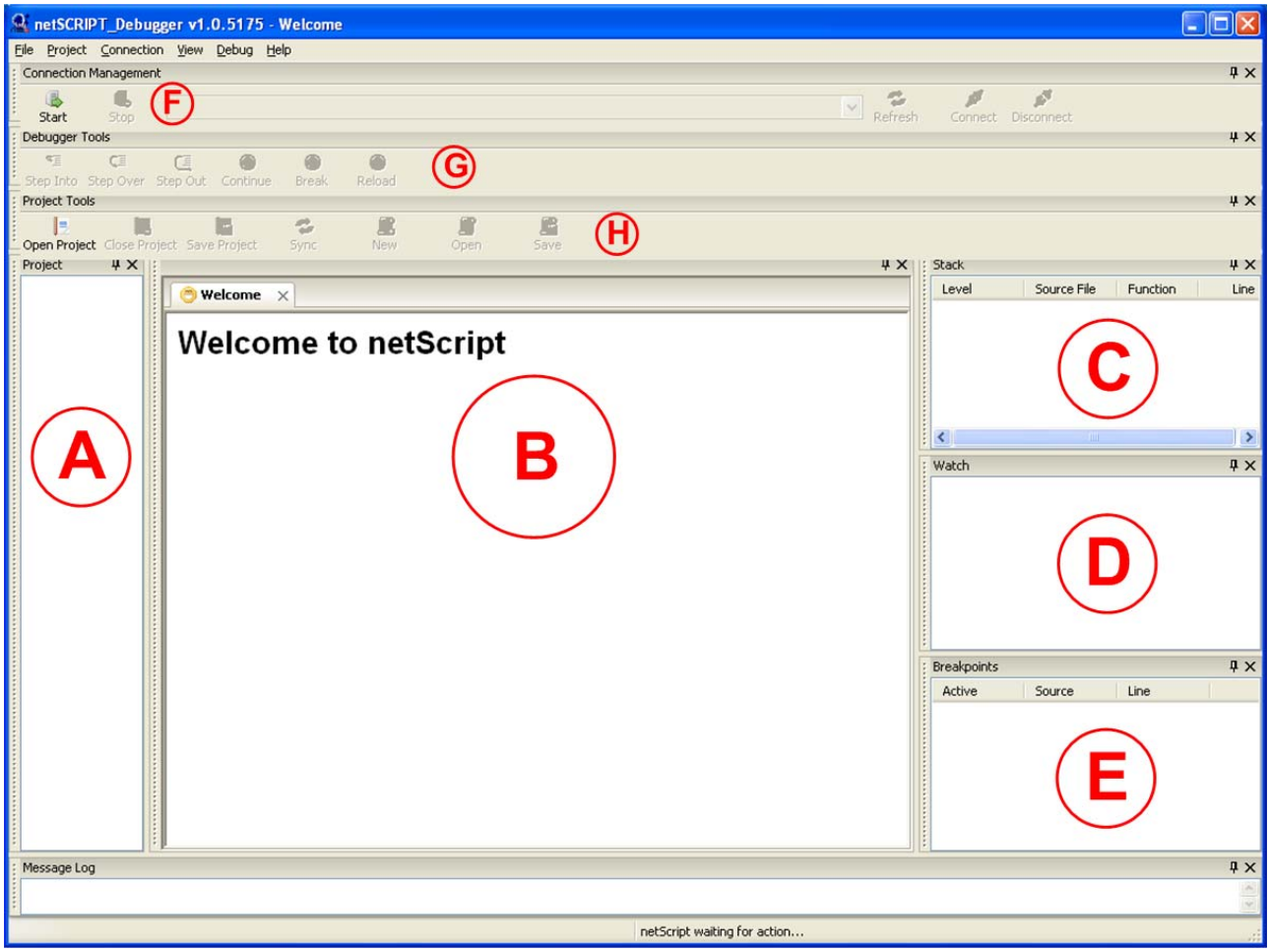


Figure 19: netSCRIPT Debugger Window

The debugger has basically 8 areas:

Ⓐ The area of project selection.

Ⓑ The script display and editing area.

Ⓒ The window in which the nesting of the script is displayed in which the script is actually running. Level 0 is always the main program.

Ⓓ The window in which the debugging mode, all variables and their current contents are displayed.

Ⓔ The window in which the position of the set breakpoints appear.

Ⓕ The bar with the tools to connect to the device netSCRIPT.

Ⓖ The bar of the debugging tools.

Ⓗ The bar of the project tools

13.1.3 Connection to the netTAP Device

First the hardware connection to the netTAP device has to be established. In general, these will be a USB connection. USB connections are displayed as serial COM interfaces of the PC, e.g., "COM1_Ch1"... "COM4_Ch1" in connection select menu.

➢ To connect the debugger to the netTAP unit click on button ① in area Ⓕ which is the area of connection management.

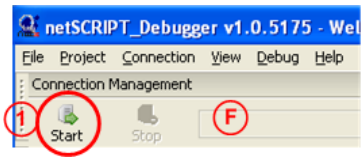


Figure 20: netSCRIPT Debugger Start Connection

- As a result a Scan for the possible connections is done from PC side and netSCRIPT capable devices found.
- After successful Scan the following becomes visible in the connection administration:

Figure 21: netSCRIPT Debugger Connection Management

- The operating buttons ③, ④ and ⑤ will become selectable. In the line Ⓕ the found connection (with device type, device version, communication channel, device type number and devices-standard number) is displayed

  With the button ③ the connection channel can be closed again.

  With the button ④ a new connection scan can be started.

  With the button ⑤ the logical connection to the netSCRIPT task can be established to the device. Also, the button ⑪ in the line of the debugger tools becomes selectable.

13.1.4 Load Current Script from netTAP

➢ After the connection setup to the target, the button (11) (break) has to be selected

Note: Also, the Script is interrupted in the processing!

Figure 22: netSCRIPT Debugger Debugger Tools

➢ The active netSCRIPT program loaded in the debugger is interrupted and is displayed.

Now the whole Debug functionality can be used. See section 13.1.7 , page 135.

Note: Nevertheless, the script loaded from the device into the debugger can only be changed when at the same time the corresponding project is opened.

13.1.5 Open a Project

➢ Open in window pane (H) the button (13).

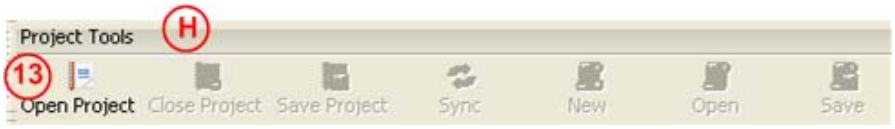


Figure 23: netSCRIPT Debugger - Project Tools

➢ This opens the file management window of the operating system. Here the corresponding project with the file ending .nxspr has to be selected.

➢ After the selection of the project it appears in window pane Ⓐ

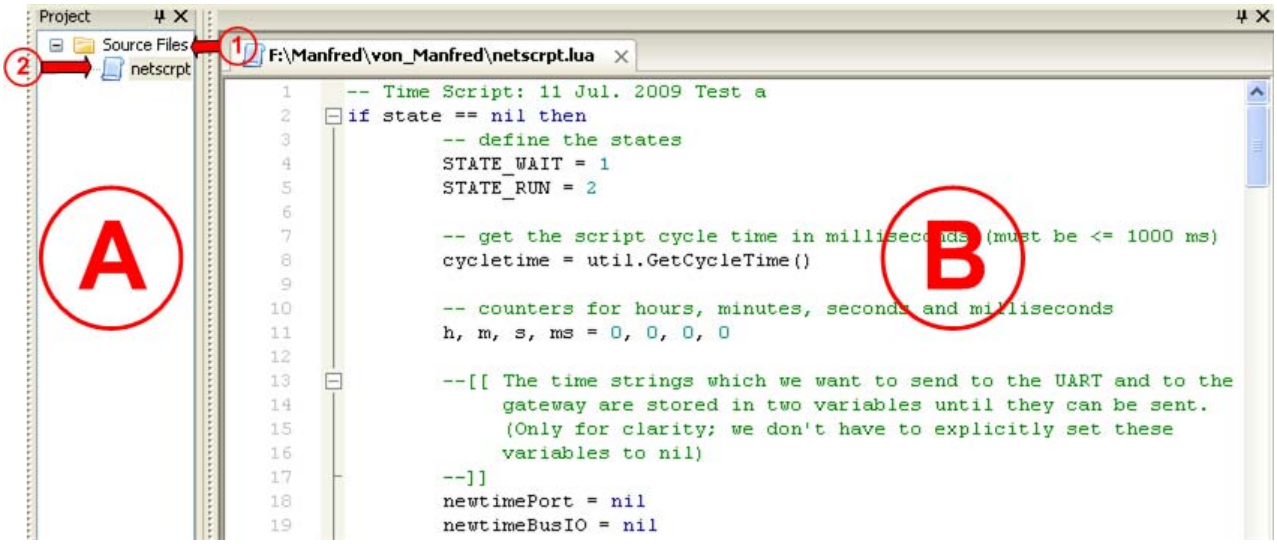


Figure 24: netSCRIPT Debugger Window pane A and B

➢ A double click on the entry ① opens the file tree.

➢ After a double click on the file to be opened its contents appear in window pane Ⓑ.

If the project contains only one script file, this is displayed immediately in window pane Ⓑ.

If the file is edited, the icon in front of the file path in the head of the window Ⓑ becomes red which indicated that there is a difference between the displayed script and the saved project version.

Figure 25: netSCRIPT Debugger - Changed Script File

Changed script files can be only loaded into the netSCRIPT device if the project was saved before.

➢ Save the project with the button ⑲ of the project tools.

Figure 26: netSCRIPT Debugger - Project Tools - save - load.

13.1.6 Load a Script into the netTAP Device

- With a mouse click on button 16 of the project tools bar, the currently netSCRIPT file displayed in the window B is loaded into the device.
- If the current script in the device is still executed, then this has to be interrupted with button 11 of the debugger tools.

Figure 27: netSCRIPT Debugger - Project Tools - Break

Thus, the following buttons are enabled:

Figure 28: netSCRIPT Debugger – Tools - Debug mode

- A mouse-click on button 12 is necessary in window pane H to start the newly downloaded program, whereby the new script is compiled for execution.
- To start the new script click button 10.

13.1.7 Script Debug

A script file loaded from the target device into the debugger as well as a script file loaded from a project and loaded into the target device can be debugged. If it is needed to edit the code in the debugger, then a project file is necessary.

The elements of the debugger tool bar in window pane H are:

Figure 29: netSCRIPT Debugger Tools - Debug Mode

7 Step Into:

The script line is executed where the courser  is displayed. If this is a function call of a netSCRIPT function, the debugger goes to the first line of this function.

8 Step Over:

The upcoming function block is automatically executed. The debugger stops in the next following line after the function block.

⑨ Step Out:

If a function block was entered with "Step Into", then further execution of the code is done automatically. The debugger stops in the first line after the return from the function call.

⑩ Continue:

The debug mode will be left and the cyclic processing of the script will continue.

⑪ Break:

The cyclic processing of the script will be stopped at the beginning of the next cycle and the debug mode is started.

Note: If the script program is running in an endless loop, the script execution can not be interrupted with this command. In such a case, a changed script file has to be loaded into the target device which contains no endless loop. Then the power supply has to be disconnected and reconnected to the device once, so that the device starts with the new script file.

⑫ Reload:

The script file loaded with button ⑯ into the target device is started for cyclic execution.

13.1.7.1 Watch-Window

In this window pane Ⓓ in the Debug mode (Step Into) all netSCRIPT defined variables, functions and tables with the actual values are displayed. These entries have a significant icon for a better identification.

 Function call with name and address.

 String variable with name and value.

 Numerical variable with name and value.

 Boolean variable with name and state

 Table, with a click on prehired "+" character the table with her contents can be opened.

 Instance with name

13.1.7.2 Set Breakpoints

Max. 32 breakpoints can be set.

Pay attention to the fact that the breakpoints set are in the range the script runs through during script processing.

➢ To set a breakpoint click with the left mouse button right beside the line number. See position ① in the following figure.

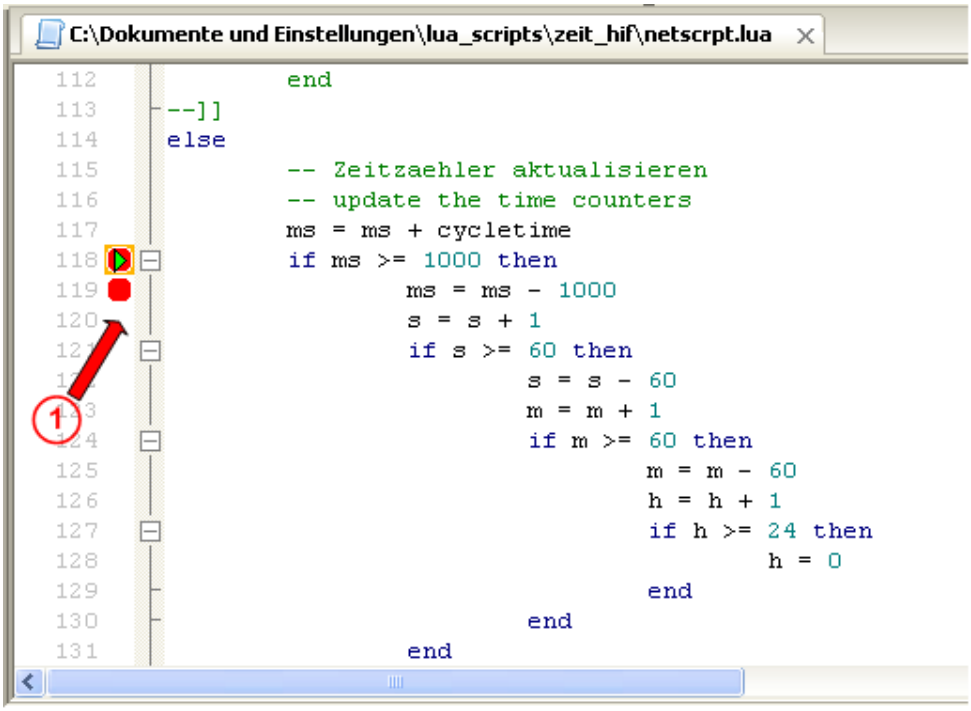


Figure 30: netSCRIPT Debugger Set Breakpoint

↬ A red rectangle appears at this position. At the same time this breakpoint is listed in the breakpoint window Ⓔ.

↬ If the script execution is continued, the processing is stopped at the set breakpoint (before execution of the line). This position is marked with a green arrow .

Set breakpoints can be removed by clicking the set breakpoint again with the left mouse button again.

13.1.8 Script Edit

In order to edit a script it is necessary:

- to have a project loaded in the debugger,

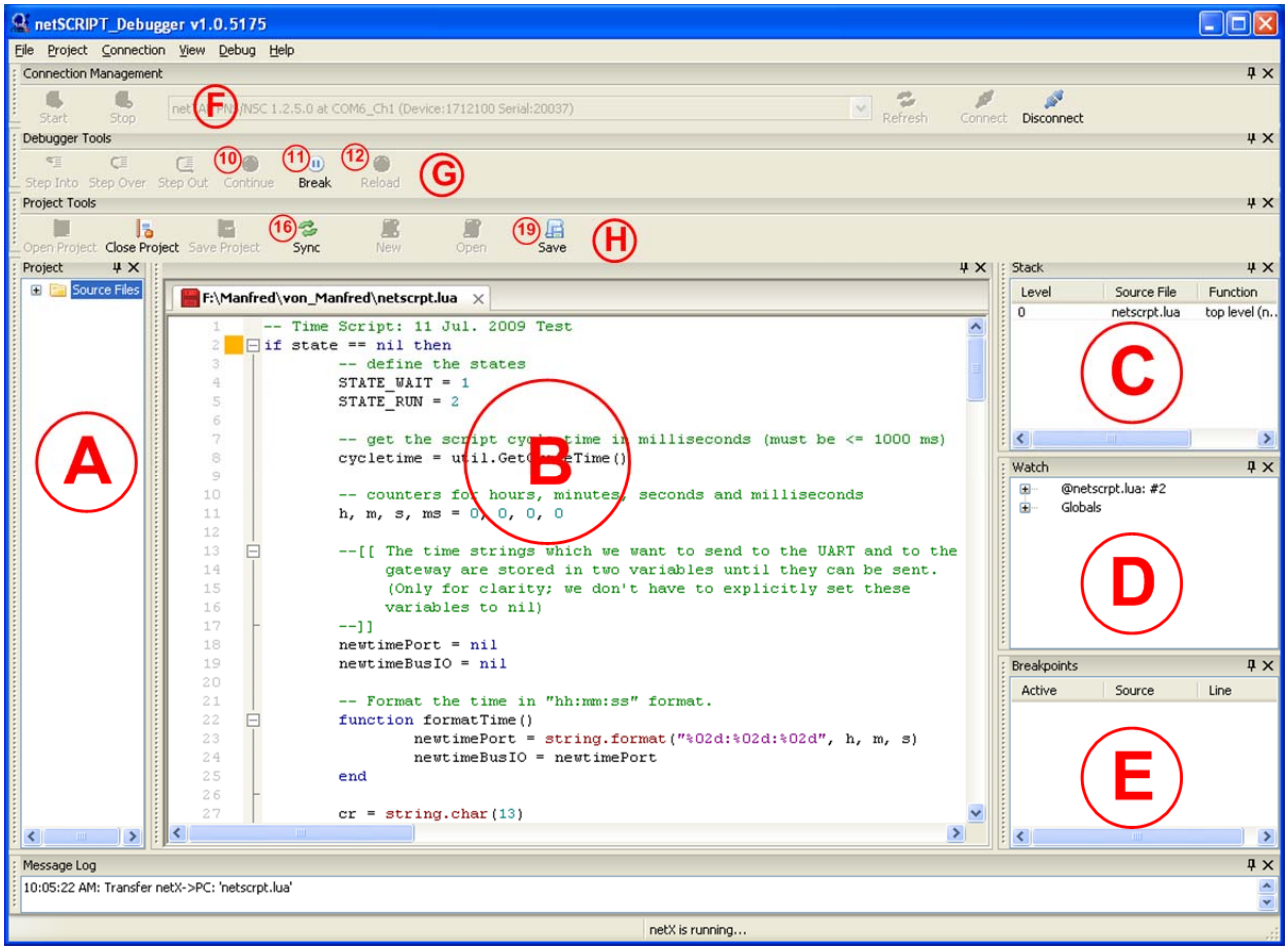


Figure 31: netSCRIPT Debugger - Script Editing

- Now the script is editable in the window pane (B).
- After editing the script, the script must be saved with the button (19) before it can be loaded into the target device. The script from the editor window is not loaded, but the saved script file will be loaded into the target device.
- After saving the script the button (16) Sync has to be clicked to transmit the script into the target device. Nevertheless, in this state the script is not executed yet in the target device.

➢ Stop the execution of the script in the target device by a click on button (11). By clicking this, the remaining buttons of the debugger tool bar (G) become enabled.

Figure 32: netSCRIPT Debugger Tools

➢ With a click on button (12) the new script file is loaded for the execution. Afterwards the new script can be started by a click on button (10).

Note: The last download of a netSCRIPT file into the netTAP device should be done with the editor of SYCON.net. With this you make sure that no inconsistency within the project managements happens.

13.1.9 Exit the Debugger

Please pay attention in whcih state the debug mod is, when you want to exit the debugger.

13.1.9.1 Exit the Debugger – With suspended Script Processing

→ The script remains stopped after the debugger was exited and the USB connection was disconnected

→ The script is executed after the device was repowered

13.1.9.2 Exit the Debugger – with running Script Processing

→ The script is executed after the debugger was exited and the USB connection was disconnected

13.1.9.3 Disconnect Debug Connection

➢ Verify, if the script is executed. You can see this in the debugger tool bar: The button „Break“ is selectable as shown in the following figure.

Figure 33: Debugger Tools. Break

And in the bottom line of the debugger: The text script runs is displayed.

- Click in the Connection Management button ⑥ „Disconnect“.

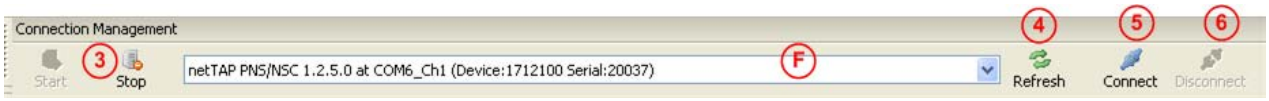


Figure 34: Debug – Connection Management

- Click in the Connection Management the button ③ „Stop“, because the debugger can end the connection to USB.
- Close/Exit the debugger window.

14 Simple netSCRIPT Sample Application

For execution of the scripts as presented here a netSCRIPT capable device with a serial interface and a PC for configuration is necessary. On the PC a hyperterminal emulation is required.

14.1 Example Program: ECHO

This script echos a character on the RS-232 interface in block mode.

This script is on the DVD:
“Examples\netSCRIPT\Echo\netscrpt.lua“.

14.2 Example Program: Blockmode

In this script the initializing of a RS interface in block mode is shown. The script echos the received character and counts the cycles.

This script is on the DVD:
“Examples\netSCRIPT\Serial Port Blockmode\blkmode.lua”

You can load this script with the debugger into the device for execution.

14.3 Example Program: Eliza

This is a script in FIFO mode (character mode). It realizes a communication with words via the hyperterminal (Joseph Weizenbaum's classic Eliza)

This script is on the DVD:
“Examples\netSCRIPT\Eliza\eliza.lua” zu finden.

You can load this script with the debugger into the device for execution.

14.4 Example Program: BusIOCount

With this script the error code in the state byte 8-11 of the BusIO interface is incremented cyclically.

This script is on the CD in the folder
“Examples\netSCRIPT\BusIOCount\busiocnt.lua “.

For execution of the script a netSCRIPT capable device with a serial interface and a PC for the configuration is required. To read the counter a superordinated control unit with the bus interface and the used network protocol is required.

The script can be loaded into the device with SYCON.net only (because of the needed signal mapping.) The signal mapping has tobe done by the user.

14.5 Example Program: hello_World

Opens the serial interface with specific parameter settings, sends a data string via this interface and closes afterwards the interface.

This script is on the CD in the folder
„Examples\netSCRIPT\Hello World\hello.lua “.

14.6 Example Program: LedFlash

The "Run Led" (depending on the cycle time) is switched on and off cyclically.

A counter is incremented by 1 at every cyclic start of the script. For descreasing the cycle time, the function Modular is used.

This script is on the CD in the folder
“Examples\netSCRIPT\LED Flash\ledflash.lua”.

14.7 Example Program: Time

This program realizes a clock. Their start time can be set via the UART interface and / or via the bus IO interface.

This script is on the CD in the folder
“Examples\netSCRIPT\Time\time.lua“.

For execution of the script, a netSCRIPT capable device with a serial interface and a superordinated bus interface is required. Exemplarily this was done with a netTAP NT 100 device from Hilscher. For configuration of the device, a PC is required. On the PC a hyperterminal emulation is required.

The script can be used only via the serial interface and is operational without a superordinated control unit.

To be able to use all possibilities of this example, the following hardware installation is necessary.

14.7.1 Installation

14.7.1.1 Hardware Installation

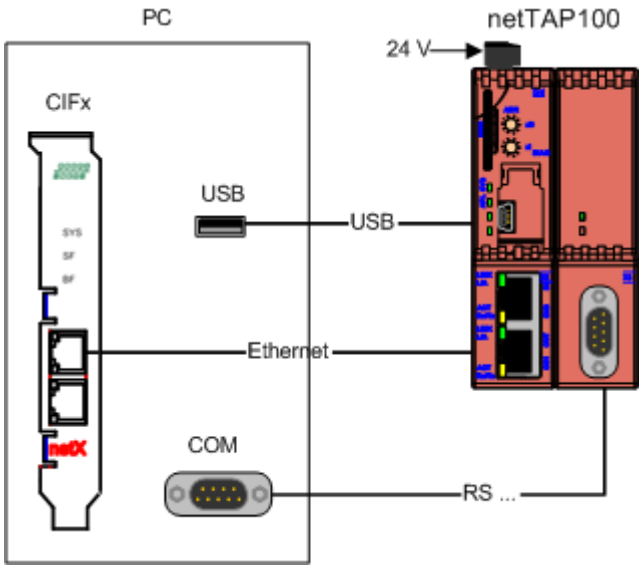


Figure 35: Script Time Example - Hardware Installation

To test only the communication of the serial interface and the transfer of the netSCRIPT program into the target device via the USB interface, no superordinated bus master (cifX card in a PC) is necessary. As a bus master any control can be used which can communicate via PROFINET.

Connect a USB cable to the USB interface of the netTAP NT 100 device and a USB interface of the PC.

A firmware and the script need to be loaded with SYCON.net software into the target device.

The test of the script can be done via the serial interface of the netTAP NT 100 with a RS232 cable and a connection to the COM connection of the PC. Use the hyperterminal software of Windows as communication partner.

From the debugger establish a communication via USB if the execution of the script should be monitored.

14.7.1.2 Terminal Settings

Open on the PC the hyperterminal program with the following settings:

1. Enter the communication interface to which the RS232 cable is connected (typically COM1).
2. Enter the following settings for configuration:
   Bits per second: =115200
   Data bits: 8
   Parity: none
   Stop bits: 1
   Flow control: Hardware
3. File > Properties > ASCII-Setup
   Echo typed characters locally.

14.7.1.3 Superordinated Control

For the complete function test of the example script the superordinated control has to do the function of a bus master and do the I/O data exchange with the netTAP device. In addition the netTAP device has to be connected with a suitable bus cable with the bus master.

14.7.2 Explanations - Script Example Programs

With help of the example the base of a meaningful script program structure is explained.

14.7.2.1 Verbal Program Description

The program "Time" realizes a clock, which cyclically displays the time on the HyperTerminal as ASCII text via the serial interface, and over the Bus IO interface to the superordinated control. After switching on the power, the time starts with the value 00:00:00. With a keyboard input via the HyperTerminal, it is possible to set the time. The program also allows to set the time via the Bus IO interface on the control system.

Note: The transmission of the time via the Bus IO interface is only successfully if the control has also released the corresponding synchronization register.

This is a 4 digit counter (h,m,s,ms), which adds up hour, minute, seconds and the cycle time. The display is done with the format hh:mm:ss.

The time can be set via the UART interface, as well as via the Bus IO interface. Also the input via the serial interface is done in the same format hh:mm:ss and is finished with the Enter key. If the example script receives a keyboard character via the serial interface, it interrupts the output of the time.

14.7.2.2 netSCRIPT Program Structure

Note: Please note that with each cyclic start each script is executed completely again (after the cycle time configured). Hence, the definition part of the script has to be separated from the cyclic script part with a script command.

Basic script program structure:

```
-- Time
-- This script demonstrates the use of the Port and BusIO interfaces
-- and shows how to realize a simple state machine.
-- It prints the time to the serial port and the bus and receives
-- a new time to set from the port and the bus.
--
-- Requirements:                                              A
-- netTap with netSCRIPT firmware,
-- serial console connected via RS232, 115200 Baud 8N1
-- bus master, e.g. cif
--
-- Author     Date          Change
-- ================================================
-- S. Lesch   Jan 15, 2010  bugfix: stop printing time during input
--                                  clear input string after cr
-- S. Lesch   Sep 16, 2009  removed German comments
-- S. Lesch   Aug 28, 2009  added comments, adapted for char mode
-- S. Lesch   Jul 9, 2009
if state == nil then  1
        B
    state = STATE_RUN
-- elseif              2
        C
else
   D
end  3
```

Figure 36: Script Time - Basic Structure

Ⓐ In the head of every script, some comment lines should be present which document the application of the script and script version used.

① In this line, the existence of the variables **`state`** is verified. If this variable is not defined (with the first start), the following code block is executed, otherwise the block after „**else**“ is executed.

Ⓑ This is the definition part of the script which should be executed only the first time the script was started.

② In this line variable **`state`** is defined to avoid that this code block Ⓑ is executed another time (until the next voltage failure).

Ⓒ This block is initially commented out. This block shows a possibility of how an output on the UART interface can be prevented, as long as a telegram from the superordinated control via the Bus IO interface has occurred.

Ⓓ This part of the script is executed every cycle.

③ This line quits the whole code.

The script parts Ⓐ, Ⓑ and Ⓒ are parts of a state machine, which has the following structure:

```
if state == nil then
    -- Initialization
    ...

    -- Definition of the states
    STATE_1 = 1
    STATE_2 = 2

    -- Set of the next state
    state = STATE_1

elseif state == STATE_1 then
    -- Action in state 1
    ...
    -- if necessary change state
    state = STATE_2

elseif state == STATE_2 then
    -- Action in state 2
    ...
    -- if necessary change state
    state = STATE_1

end
```

With a cyclic pass of this script section (as it is the case for usual processing) only the currently valid part of the state machine is processed, because the state variables stay unchanged between the end of the script and the restart of the script.

14.7.2.3 Details of Block B

In this block

- the functions for output formatting and time input recognition are defined.
- under “—UART initialization” the serial interface is initialized in CHAR mode using “PortOpen()”
- under “— BusIO initialization” the interface to the superordinated control is initialized using “BusIOOpen()”
- optionally the query of the communication enabling of the serial interface over the BusIO interface (superordinated control) is implemented

14.7.2.4 Details of Block C

To activate this block the comment characters have to be deleted at ① and ② of the program section displayed below.

```
      -- update state
      state = STATE_RUN
--  (1) state = STATE_WAIT

-----------------------------------------------------------------------
-- optional state: wait until a string has been received from BusIO.
-----------------------------------------------------------------------

--[[ (2)
elseif state == STATE_WAIT then
      if b:BusIORead() then
            state = STATE_RUN (3)
      end
--]]
else (4)
```

Figure 37: Script Time - Block C

As long as „**state**“ is „**STATE_WAIT**“ the „**elseif**“ condition is **true** and the following „**else**“ condition at ④ is not executed. When at ③ „**state**“ is „**STATE_RUN**“ then the code after the „**else**“ condition at ④ is executed in the next cycle and then the UART interface is used.

14.7.2.5 Details of Block D

The block Ⓓ has three sections:

1. Time counting
2. UART communication
3. BusIO communication

Section 1, Time counting

```
-- Zeitzaehler aktualisieren
-- update the time counters
ms = ms + cycletime
if ms >= 1000 then
      ms = ms - 1000
      s = s + 1
      if s >= 60 then
            s = s - 60
            m = m + 1
            if m >= 60 then
                  m = m - 60
                  h = h + 1
                  if h >= 24 then
                        h = 0
                  end
            end
      end

      -- neue Sendestrings erstellen
      -- set the new send strings
      formatTime()

      -- run-LED im Sekundentakt blinken lassen
      -- toggle the "run" LED every second
      util.SetLed("run", s % 2 == 1)
end
```

(1) (2) (3)

Figure 38: Script Time - Block D Section 1

In section ① of the picture above the counters are used for the time display.

With ② the time formatting function from the initialization area Ⓑ is called.

With ③ the LED will blink on the UART module.

Section 2, UART communications

```
local strChar, rxErr = p:PortGetChar()
if strChar then
      -- If case of a receive error, send a message and clear the input string.
      if rxErr then
            p:PortPutChar(crlf, "Receive error", crlf)
            strInput = ""
      -- If the character was received correctly, append it to the input string.
      else
            strInput = strInput..strChar
            -- If the character is a CR, parse the input and set new time.
            if strChar:byte(1) == 13 then
                  if not parseTime(strInput) then
                        -- if unsuccessful, print an error message.
                        p:PortPutChar(crlf, "Parse error: ", strInput, crlf)
                  end
                  strInput = ""
            end
      end
end

-- send current time string to UART
if strInput == "" and newtimePort and p:PortPutChar(cr, newtimePort, cr) then
      newtimePort = nil
end
```

Figure 39: Script Time - Block D Section 2

① It is verified, whether something was received via the UART interface.

② It is checked, whether the reception was ok. At erroneous character reception an error message is issued.

③ The received characters are serialized.

④ If the last received character is a “CR”, the serialized characters are passed to the function "parseTime" which was defined in the area Ⓑ.

⑤ Within function "parseTime" it is checked, whether the characters having been put in can be interpreted as a time value. If yes, processing of the new time will be continued, otherwise **nil** will be returned and an error message will be put out on the port.

⑥ The current time is put out independently whether characters have been received via the serial interface. The memory for the output time on the serial interface is initialized.

Section 3 – BusIO communications

```
1  str = b:BusIORead()
   if str then
2        b:BusIOSetError("receive", not parseTime(str))
   end

   if newtimeBusIO then
3        if b:BusIOWrite(newtimeBusIO) then
               newtimeBusIO = nil
         --[[
4        elseif lasterror == err.BUSIO_SEND_DISABLED then
               state = STATE_WAIT
         --]]
         end
   end
```

Figure 40: Script Time - Block D Section 3

1 The bus IO interface is checked for new data.

2 If a telegram was in the interface, the function "parseTime" is called. Depending on the result of this function an error to the Bus IO interface will be passed if necessary. If the interpretation of the telegram was successful, the time counters are set again by the function "parseTime".

3 If a new output time exists, this will be transferred via the Bus IO interface.

4 This is an optional possibility how the output on the UART interface can be quit if the telegram handing via the Bus IO interface has failed.

14.7.3 Use of the Program

14.7.3.1 Usage via the Serial Communication Interface RS232

The program starts automatically after the download with SYCON.net software with the serial transmission without a release of the superordinated control via the Bus interface IO necessary.

- Build up a connection from the COM port of your PC to the serial interface of the target device.
- Start a terminal program on the PC with the parameters listed in section 14.1.

Then the window of the terminal shows the following data:

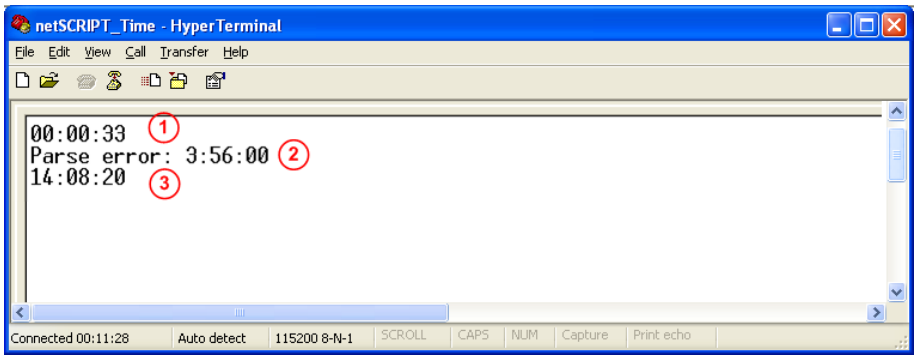


Figure 41: Script Time - Display on a Serial Terminal

(1) In this line you see the output without a start time was set. The script runs for 33 seconds.

For a default time, enter a time in the format hh:mm:ss in the terminal window. The input has to be finished with the "Enter" key.

(2) In this line, the reaction of the script to a faulty input is displayed with an output of the received characters. There was a number in the input for the hour missing.

(3) In this line, the time output after a correct default time is displayed. The actual time is always displayed in the same line.

14.7.3.2 Usage via the Superordinated Network Interface

For the communication via the superordinated network interface of the netTAP device a bus master is needed.

Please note that the input and output length of the I/O data to be transmitted from and to netTAP needs to be configured in the control unit / master with at least 24 bytes, so that the data header can occur for the synchronization between netTAP and control via the I/O data. The pure user data begin from the 25'th byte. The time will be transmitted from the program via the Bus IO interface in the format hh:mm:ss. 8 user data bytes are to be transmitted, in addition, in I/O data. All together the example program needs 32 bytes input and output data.

In the following example a CifX card was used as a PROFINET IO controller.

➢ In the handshake procedure between the master/controller and the netTAP a release is for send and receive data has to be done first from the master. For this value 000000C0h (LSB first) in the synchronization register byte 0 to 3 of the output data has to be sent.

The following figure shows at the top the data received from netTAP via the bus master and below the output data from the bus master.

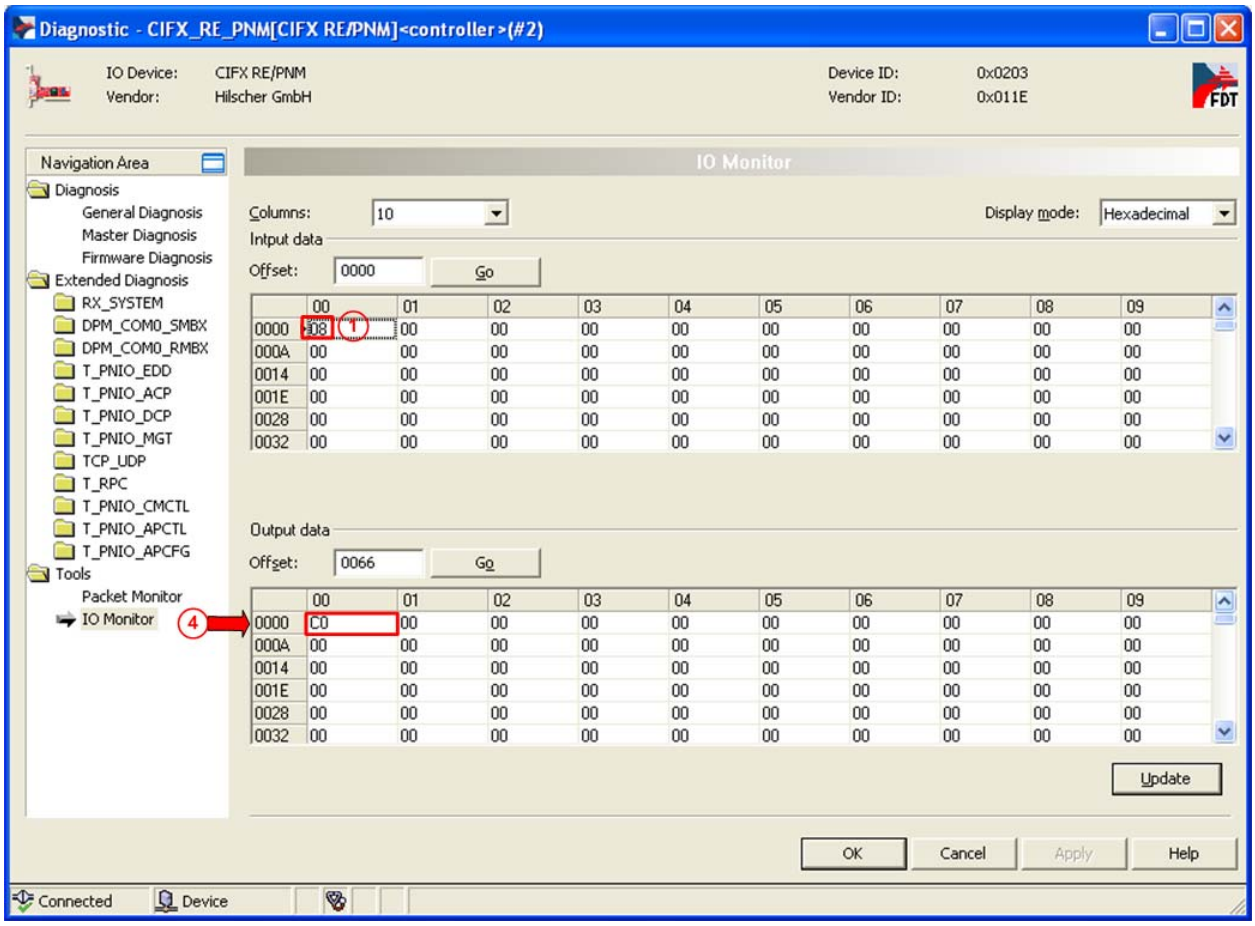


Figure 42: Script Time – Fieldbus Initialization Data Exchange

① Handshake byte 0: The value is 0x00 if in the script the function ":BusIOSetRun" is not called. If this function was already called, the value "0x08" is here.

➢ After setting the release bits 6 and 7 (0xC0) (④) and acknowledging with button ⑦ (by which the data are send to the script) the example program answers immediately with the actual counter values (time). In the bus master this value can be read in the input data.

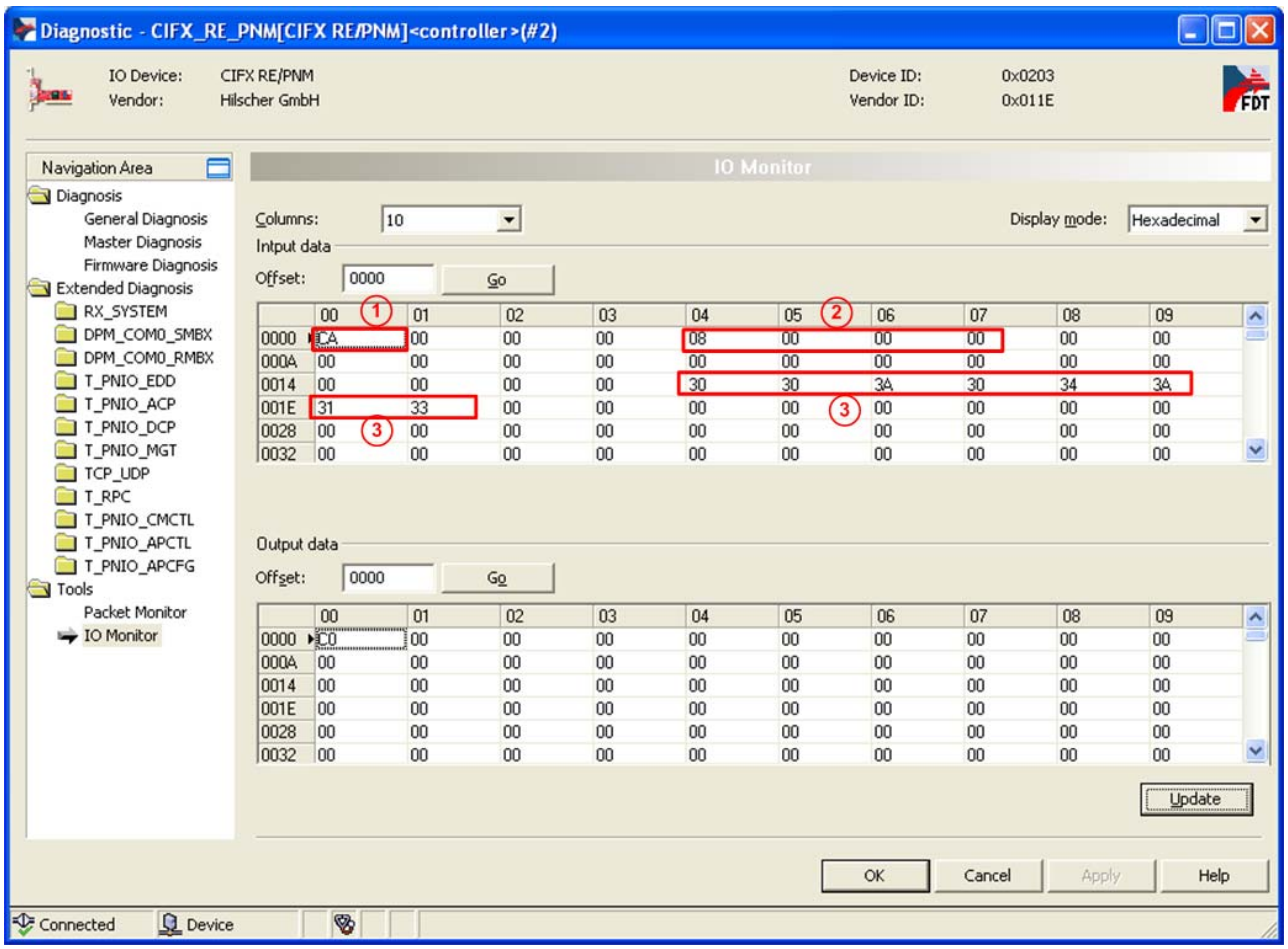


Figure 43: Script Time - Set the Actual Time

① In the synchronization register byte 0 - 3 with 0xC2000000 the transmission of the user data is confirmed. With the function ":BusIOSetRun" in the script the value 0xCA is returned.

② Byte 4-7 contains the received user data length.

③ From byte 24 to byte 31 the time is returned. Here, 0x30 0x30 0x3A, 0x30 0x31 0x3A, 0x33 0x31 → 00:01:31. The time in the script was not set till now.

Should the time be transmitted the next time by the example script, must be informed (acknowledged) that the first data transmission was successful. This is done by adapting the state of the bit 1 in the output-synchronization register byte 0-3 to the state of bit 1 in the corresponding input-synchronization register figure pos. ①.

Moreover in the output data the value in byte 0 changes from 0xC0 to 0xC2.

Should the time be sent out on the fieldbus, the following needs to be done:

1. First find out the handshake situation send to the master from the example program. 0xC0 or 0xC2, see ① in the following figure.

2. Set the output synchronization register byte 0-3 in ④ of the following figure to the state of the following table and invert it with the state of the bit 0.

Latest from the example program preserved synchronization register value	To the example program synchronization register value to be sent
000000C0	000000C1
000000C2	000000C3

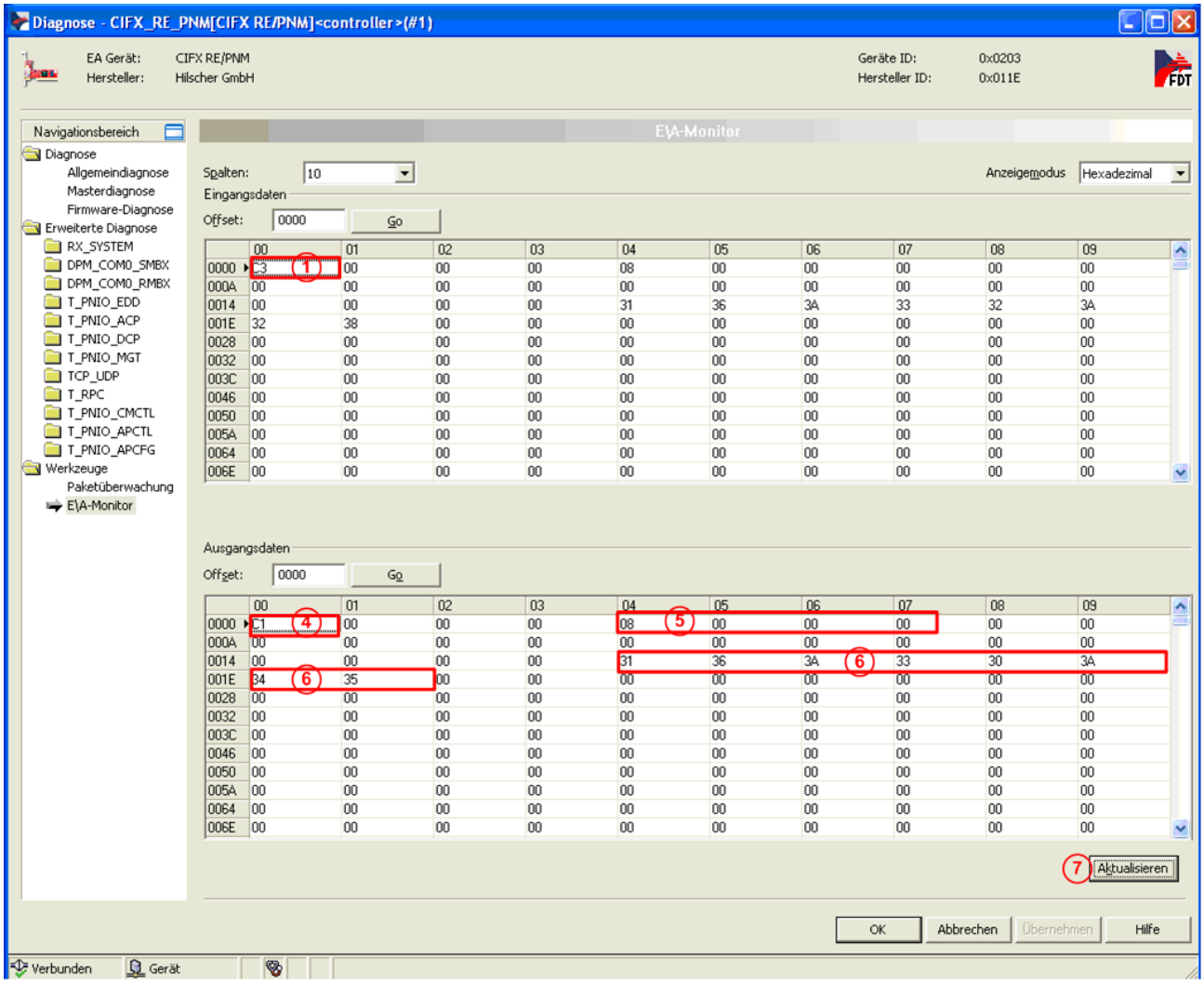


Figure 44: Script Time - Time set via Fieldbus

3. Enter in the handshake byte (④) the necessary value, with this the time in the script is editable. Enter in byte 4 (⑤) the data length to be sent (here 0x08) and in the byte 24 to byte 31 (⑥) the time which the script should use for sending the telegram. In the above figure the time has value 0x31, 0x36 0x3A, 0x33 0x30 0x3A, 0x34 0x35 as a time: 16:30:45“. With a click on button ⑦ the data are sent.

   If you want to read only data again then enter for byte 4 (⑤) data length 0.

① The data receipt was confirmed by the example program by adapting the bit 0 in the input synchronization register 0xC3. Now new data can be sent.

② Now the actual time corresponds to the sent data and is visible directly on the serial interface in the terminal program.

14.7.3.3 Possible Handshake Sequence from the Perspective of the Superordinated Control

With the handshake input in clips standing values signify the value of the byte after call of the function ":BusIOSetRun" in the script.

Handshake Input Byte 0	Handshake Output Byte 0	Description
00 (08)	00	Initial state.
	C0	Release of the communication,→ the script sends the first data.
C2 (CA)		Data are transmitted from the script.
	C2	The data receipt is confirmed → the script sends new data.
C0 (C8)		Data are transmitted from the script.
	C0	The data receipt is confirmed → the script sends new data.
C2 (CA)		Data are transmitted from the script.
	C1	Data are dispatched to the script.
C3 (CB)		Data are transmitted from the script.
	C3	The data receipt is confirmed → the script sends new data.
C1 (C9)		Data are transmitted from the script.
	C1	The data receipt is confirmed → the script sends new data.
C3 (CB)		Data are transmitted from the script.
	C2	The data receipt is confirmed → the script sends new data.
C0 (C8)		Data are transmitted from the script.
	C0	The data receipt is confirmed → the script sends new data.
C2 (CA)		Data are transmitted from the script.
	C2	The data receipt is confirmed. → New data are dispatched to the script.
C0 (C8)		Data are transmitted from the script.
	C1	The data receipt is confirmed → the script sends new data.
C3 (CB)		Data are transmitted from the script.
	C2	The data receipt is confirmed. → New data are dispatched to the script.
C0 (C8)		Data are transmitted from the script.

15 Lists

15.1 List of Figures

15.2 List of Tables

16 Glossar

Lua

Lua is a freely usable Script language, witch is developed by the "**Pontifícia Universidade Católica do Rio de Janeiro**- ©".

netSCRIPT

Is a script-based programming language for communication devices of Hilscher GmbH, witch allows users themselves to program flows to program / protocol conversions into sights.

netTAP

Is a device which can be used in protocol conversions between two different network systems Hilscher GmbH, on the netSCRIPT.

SYCON.net

PC-configuration tools of Hilscher GmbH used for configuration of fieldbus devices.

UART

A UART interface serves for the sending and receiving of data about a data line and illustrates the standard of the serial interfaces in PCs and microcontrollers (e.g., RS-232 or RS-485).

BUS IO

Is the name of the in and output interface of the superior I/O network within netSCRIPT. Data to the control (master, host, PLC) will pass to the BUS it IO interface or receive from there data of the superior control.

17 Technical Data

Storage space for programs and variable approx. 1 Mbyte	
Cycle time	min 1 ms max 1000 ms
Serial Interface support	RS-232, RS-422, RS-485
Maximum usable data	1024 Byte

18 Contacts

Headquarters

Germany
Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

Support
Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

China
Hilscher Systemautomation (Shanghai) Co. Ltd.
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn

Support
Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

France
Hilscher France S.a.r.l.
69500 Bron
Phone: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr

Support
Phone: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

India
Hilscher India Pvt. Ltd.
New Delhi - 110 025
Phone: +91 11 40515640
E-Mail: info@hilscher.in

Italy
Hilscher Italia srl
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it

Support
Phone: +39 02 25007068
E-Mail: it.support@hilscher.com

Japan
Hilscher Japan KK
Tokyo, 160-0022
Phone: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp

Support
Phone: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

Korea
Hilscher Korea Inc.
Suwon, 443-810
Phone: +82-31-204-6190
E-Mail: info@hilscher.kr

Switzerland
Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch

Support
Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

USA
Hilscher North America, Inc.
Lisle, IL 60532
Phone: +1 630-505-5301
E-Mail: info@hilscher.us

Support
Phone: +1 630-505-5301
E-Mail: us.support@hilscher.com