Benutzerhandbuch

netSCRIPT

Programmiersprache für serielle Kommunikation

V1.3.x.x

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC090801UM06DE | Revision 6 | Deutsch | 2010-07 | Freigegeben | Öffentlich

Inhaltsverzeichnis

1 Einleitung

1.1 Über das Benutzerhandbuch

Dieses Benutzerhandbuch beschreibt die Programmiersprache netSCRIPT. Diese wird eingesetzt, um den Datentransfer zwischen unterschiedlichen Netzwerkprotokollen in Hilscher-Geräten zu realisieren.

netSCRIPT basiert auf der Skriptsprache Lua und ist um spezielle Kommunikationsfunktionen erweitert.

netSCRIPT Programme werden mit der Software SYCON.net erstellt und ins Zielgerät geladen. Dort werden sie über einen Interpreter zyklisch abgearbeitet.

Das Debuggen der Skriptdatei ist mit der Zusatzsoftware netSCRIPT_Debugger.exe auf jedem herkömmlichen PC möglich. Das Programm ist neben dem Konfigurationsprogramm SYCON.net separat zu installieren.

1.1.1 Änderungsübersicht

Index	Datum	Kapitel	Änderungen
4	19.02.10	13.1.9	Neuer Abschnitt „*Debugger beenden*“
			Beispiel Time-Skript an Realisierung angepasst.
5			netScript Version 1.2.x.x
		5.1.18	Fehlerkorrekturen (Skriptbeispiel jetzt lauffähig)
		6.2, 6.3, 6.5	Fehlerkorrekturen Fehlerkorrekturen
		7.1	Ergänzungen
		8.3	Ergänzungen
		9.1	Ergänzungen
		10	Korrekturen
		10.9.3.1 10.9.3.2	Abschnitte in der Reihenfolge getauscht
		8.3.5 8.3.6	port.STA_PATTERN_MATCHED → port.STA_PATTERN_MATCH
		11.1	Ergänzungen
		11.2	neu
		12.1.2	Ergänzungen
		14.7.7	Ergänzungen
6	13.07.10		netSCRIPT Version 1.3.x.x
		10	Kapitel Funktionen zur Kommunikation mit dem übergeordneten E/A-Netzwerk überarbeitet und erweitert mit neuer BusIO-Kommunikation ohne Handshake

Tabelle 1: Änderungsübersicht

1.1.2 Bezug auf netSCRIPT, Hardware, Software und Firmware

Hinweis: Die aufgeführten Hardware-Revisionen, Firmware- und Treiber-Versionen bzw. die Versionen für die Konfigurationssoftware gehören funktional zusammen

Bei vorhandener Hardware-Installation, die Firmware, den Treiber sowie die Konfigurationssoftware aktualisieren.

netSCRIPT läuft auf folgender Hardware

Gerät	Revision
NT 100-RE-RS	1
NT 100-DP-RS	1
NT 100-CO-RS	1
NT 100-DN-RS	1
NT 100-CC-RS	1

Tabelle 2: Bezug auf Hardware

netSCRIPT

netSCRIPT	Version
Basiert auf Lua Die aus Lua übernommenen Funktionen sind im Abschnitt 5 dieser Dokumentation beschrieben.	5.1
netSCRIPT	1.3.x.x

Tabelle 3: Bezug auf netSCRIPT

Software

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

Tabelle 4: Bezug auf Software

Firmware

Firmware Datei	Firmware Version
NTxxxNSC.NXF	1.3.x.x

Tabelle 5: Bezug auf Firmware

1.1.3 Konventionen in diesem Handbuch

Handlungsanweisungen, ein Ergebnis eines Handlungsschrittes bzw. Hinweise sind wie folgt gekennzeichnet:

Handlungsanweisungen:

➢ <Anweisung>

oder

1. <Anweisung>
2. <Anweisung>

Ergebnisse:

<Ergebnis>

Hinweise:

Wichtig: <Wichtiger Hinweis>

Hinweis: <Hinweis>

<Verweis auf weitere Informationen>

netSCRIPT Darstellungen in dieser Anleitung:

Syntax	Bedeutung
anweisungen	netSCRIPT Befehle werden in der Regel. klein geschrieben. Funktionen können auch Großbuchstaben enthalten oder ausschließlich aus Großbuchstaben bestehen. netSCRIPT unterscheidet zwischen Klein- und Großbuchstaben.
VAR1	Variablenname
_par	Parameter werden in Kleinbuchstaben dargestellt
_par$_i$	Parameter mit tief gestellten Indizes bedeuten, dass an dieser Stelle eine Liste von Operanden aufgeführt werden kann
Werte	Der Inhalt von Variablen / Parametern wird kursiv dargestellt
[ ]	Alternative Anweisungen stehen zwischen einfachen eckigen Klammern

Tabelle 6: Skript Beschreibungs-Syntax

Wird im Text ein Begriff in Anführungszeichen **fett** geschrieben „**Anweisung**“, so ist dieses ein direkter Bezug auf die beschriebene Funktion bzw. deren Parameter.

Steht ein „**:**“ am Beginn eines Funktionsnamens, so ist der Aufruf dieser Funktion vor dem „**:**“ mit dem Präfix des Rückgabewertes der entsprechenden „Open“-Funktion zu ergänzen.

1.1.3.1 Dokumentationen

Die nachfolgende Dokumentationsübersicht gibt Auskunft darüber, in welchem Handbuch Sie zu welchen Inhalten weitere Informationen finden können.

Alle in der Übersicht aufgeführten Handbücher sind auf der mitgelieferten CD unter dem Verzeichnis **Dokumentation** im Adobe-Acrobat® Reader-Format (PDF) zu finden.

Handbuch	Inhalt	Dokumentname
Benutzerhandbuch	Konfiguration von NT 100-Geräten	netTAP100_usermanual_de.pdf
Bediener-Manual	Installation der Software SYCON.net und Konfiguration des NT 100 mit dieser Software.	netGateway_de.pdf

Tabelle 7: Dokumentationen

Weitere Informationen zu der Script-Sprache Lua können Sie auf folgenden Internetseiten finden:

http://www.lua.org

und Dokumentation unter

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

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

1.2 Rechtliche Hinweise

1.2.1 Copyright

1.2.2 Wichtige Hinweise

Das Benutzerhandbuch, Begleittexte und die Dokumentation wurden mit größter Sorgfalt erarbeitet. Fehler können jedoch nicht ausgeschlossen werden. Eine Garantie, die juristische Verantwortung für fehlerhafte Angaben oder irgendeine Haftung kann daher nicht übernommen werden. Sie werden darauf hingewiesen, dass Beschreibungen in dem Benutzerhandbuch, den Begleittexte und der Dokumentation weder eine Garantie, noch eine Angabe über die nach dem Vertrag vorausgesetzte Verwendung oder eine zugesicherte Eigenschaft darstellen. Es kann nicht ausgeschlossen werden, dass das Benutzerhandbuch, die Begleittexte und die Dokumentation nicht vollständig mit den beschriebenen Eigenschaften, Normen oder sonstigen Daten der gelieferten Produkte übereinstimmen. Eine Gewähr oder Garantie bezüglich der Richtigkeit oder Genauigkeit der Informationen wird nicht übernommen.

Wir behalten uns das Recht vor, unsere Produkte und deren Spezifikation, sowie zugehörige Benutzerhandbücher, Begleittexte und Dokumentationen jederzeit und ohne Vorankündigung zu ändern, ohne zur Anzeige der Änderung verpflichtet zu sein. Änderungen werden in zukünftigen Manuals berücksichtigt und stellen keine Verpflichtung dar; insbesondere besteht kein Anspruch auf Überarbeitung gelieferter Dokumente. Es gilt jeweils das Manual, das mit dem Produkt ausgeliefert wird.

Die Hilscher Gesellschaft für Systemautomation mbH haftet unter keinen Umständen für direkte, indirekte, Neben- oder Folgeschäden oder Einkommensverluste, die aus der Verwendung der hier enthaltenen Informationen entstehen.

1.2.3 Haftungsausschluss

Die Software wurde von der Hilscher Gesellschaft für Systemautomation mbH sorgfältig erstellt und getestet und wird im reinen Ist-Zustand zur Verfügung gestellt. Es kann keine Gewährleistung für die Leistungsfähigkeit und Fehlerfreiheit der Software für alle Anwendungsbedingungen und -fälle und die erzielten Arbeitsergebnisse bei Verwendung der Software durch den Benutzer übernommen werden. Die Haftung für etwaige Schäden, die durch die Verwendung der Hard- und Software oder der zugehörigen Dokumente entstanden sein könnten, beschränkt sich auf den Fall des Vorsatzes oder der grob fahrlässigen Verletzung wesentlicher Vertragspflichten. Der Schadensersatzanspruch für die Verletzung wesentlicher Vertragspflichten ist jedoch auf den vertragstypischen vorhersehbaren Schaden begrenzt.

Es ist strikt untersagt, die Software in folgenden Bereichen zu verwenden:

- für militärische Zwecke oder in Waffensystemen;
- zum Entwurf, zur Konstruktion, Wartung oder zum Betrieb von Nuklearanlagen;
- in Flugsicherungssystemen, Flugverkehrs- oder Flugkommunikations-systemen;
- in Lebenserhaltungssystemen;
- in Systemen, in denen Fehlfunktionen der Software körperliche Schäden oder Verletzungen mit Todesfolge nach sich ziehen können.

Sie werden darauf hingewiesen, dass die Software nicht für die Verwendung in Gefahrumgebungen erstellt worden ist, die ausfallsichere Kontrollmechanismen erfordern. Die Benutzung der Software in einer solchen Umgebung geschieht auf eigene Gefahr; jede Haftung für Schäden oder Verluste aufgrund unerlaubter Benutzung ist ausgeschlossen.

1.2.4 Gewährleistung

Obwohl die Hard- und Software mit aller Sorgfalt entwickelt und intensiv getestet wurde, übernimmt die Hilscher Gesellschaft für Systemautomation mbH keine Garantie für die Eignung für irgendeinen Zweck, der nicht schriftlich bestätigt wurde. Es kann nicht gewährleistet werden, dass die Hard- und Software Ihren Anforderungen entspricht, die Verwendung der Software unterbrechungsfrei und die Software fehlerfrei ist. Eine Garantie auf Nichtübertretung, Nichtverletzung von Patenten, Eigentumsrecht oder Freiheit von Einwirkungen Dritter wird nicht gewährt. Weitere Garantien oder Zusicherungen hinsichtlich Marktgängigkeit, Rechtsmängelfreiheit, Integrierung oder Brauchbarkeit für bestimmte Zwecke werden nicht gewährt, es sei denn, diese sind nach geltendem Recht vorgeschrieben und können nicht eingeschränkt werden. Gewährleistungsansprüche beschränken sich auf das Recht, Nachbesserung zu verlangen.

1.2.5 Exportbestimmungen

Das gelieferte Produkt (einschließlich der technischen Daten) unterliegt den gesetzlichen Export- bzw. Importgesetzen sowie damit verbundenen Vorschriften verschiedener Länder, insbesondere denen von Deutschland und den USA. Die Software darf nicht in Länder exportiert werden, in denen dies durch das US-amerikanische Exportkontrollgesetz und dessen ergänzender Bestimmungen verboten ist. Sie verpflichten sich, die Vorschriften strikt zu befolgen und in eigener Verantwortung einzuhalten. Sie werden darauf hingewiesen, dass Sie zum Export, zur Wiederausfuhr oder zum Import des Produktes unter Umständen staatlicher Genehmigungen bedürfen.

1.2.6 Eingetragene Warenzeichen

Windows® 2000 / Windows® XP sind eingetragene Warenzeichen der Microsoft Corporation.

Adobe-Acrobat® ist ein eingetragenes Warenzeichen der Adobe Systems Incorporated.

Rocksoft® ist ein eingetragenes Warenzeichen der Rocksoft Pty Ltd, Australia.

1.3 Lizenzen

Lua ist Open Source Software, die von jedem genutzt werden kann.

http://www.lua.org

Lua ist eine Scriptsprache, die von der

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

entwickelt wurde.

2 Kurzbeschreibung und Voraussetzungen

2.1 Kurzbeschreibung

netSCRIPT ist eine auf Lua basierende Skript-Programmiersprache, die zur Erstellung und Laden der Programme in die Zielhardware das PC Konfigurations- und Diagnosewerkzeug SYCON.net voraussetzt.

Mit der Sprache netSCRIPT ist es möglich, Geräte die über eine serielle Kommunikationsschnittstelle (UART Port) wie RS232, RS422, RS485 verfügen, an eine übergeordnete Steuerung über ein weiteres Bussystem (Bus IO) zu koppeln.

netSCRIPT-Programme werden in SYCON.net verwaltet. Dort erfolgt das Editieren, die syntaktische Prüfung und die Speicherung. Zusammen mit der Gesamtnetzwerkkonfiguration des Zielgerätes wird das netSCRIPT Programm an die Zielhardware übertragen. Das serielle Protokoll und sein Ablauf kann dabei beliebig bestimmt und programmiert werden. Es stehen Funktionen für Senden und Empfangen über den UART zur Verfügung. Die Übergabe an das übergeordnete Bussystem erfolgt über Datenpuffer IN und OUT, für deren Zugriff weitere Funktionen zur Verfügung stehen. Darüber hinaus lassen sich Statusinformationen übermitteln.

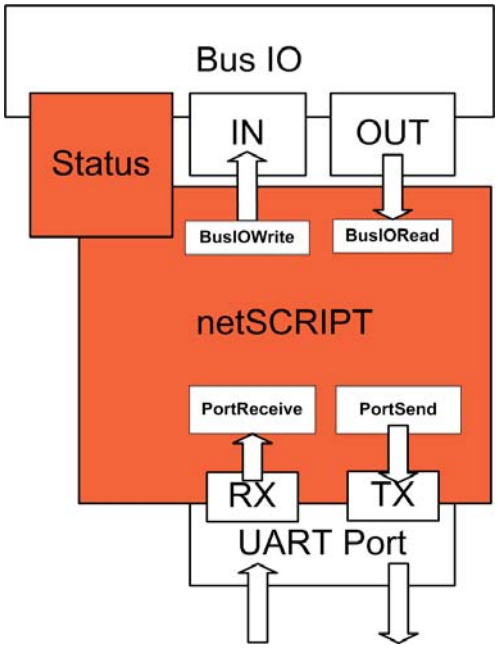


Abbildung 1: netSCRIPT Kommunikationskanäle

2.2 Systemvorraussetzungen

1. PC mit COM und USB 1.1 Schnittstellen und CD-ROM Laufwerk.
2. Betriebssystem Windows® 2000/Windows® XP.
3. SYCON.net und netSCIRPT-Debugger (Als Programmier und Debugging Werkzeuge).
4. Ein netSCRIPT fähiges Gerät als Zielhardware des netSCRIPT-Programmes, z.B. netTAP 100.

3 netSCRIPT-Editor

Der netSCRIPT-Editor ist Bestandteil der SYCON.net Software.

3.1 Aufruf des Editors

Die Scriptverwaltung und Editierfunktion befinden sich im Konfigurationsmenü eines netSCRIPT fähigen Gerätes. In diesem Abschnitt wird der Aufruf des Editors am Beispiel eines netTAP 100-Gerätes gezeigt.

3.1.1 Geräteauswahl

➢ Starten Sie die SYCON.net-Software auf Ihrem PC. Wählen Sie Ihr Projekt aus oder erstellen sie ein Projekt, in dem die Geräte enthalten sind, die Sie programmieren wollen.

➧ Es öffnet sich das folgende Fenster:

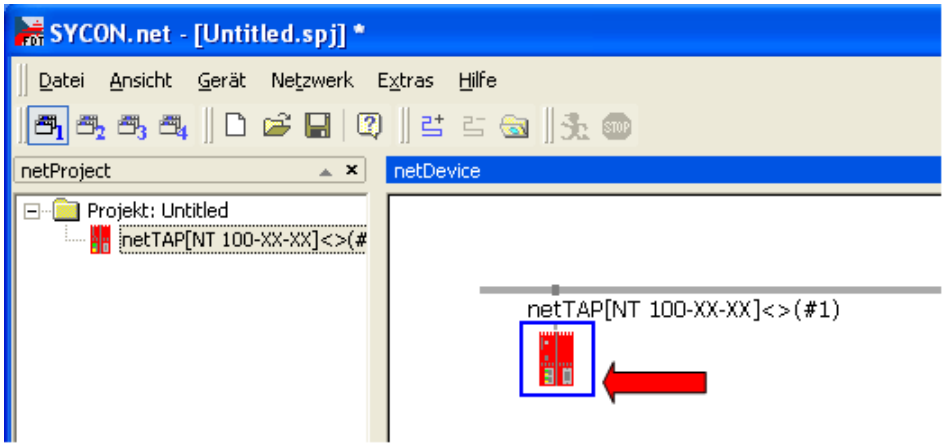
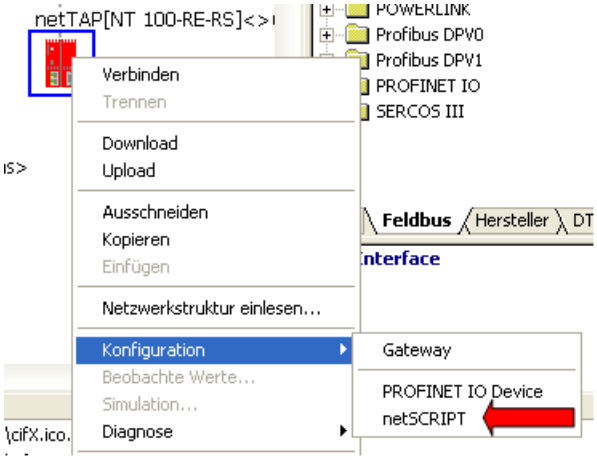


Abbildung 2: Auswahl skriptfähiges Gerät

➢ Machen Sie auf das Gerät, welches das Script erhalten soll einen Klick mit der rechten Maustaste.

➧ Es öffnet sich das folgende Fenster:



➢ Wählen Sie **Konfiguration > netSCRIPT** aus.

➧ Dies öffnet das Konfigurationsfenster der UART Schnittstellenparameter.

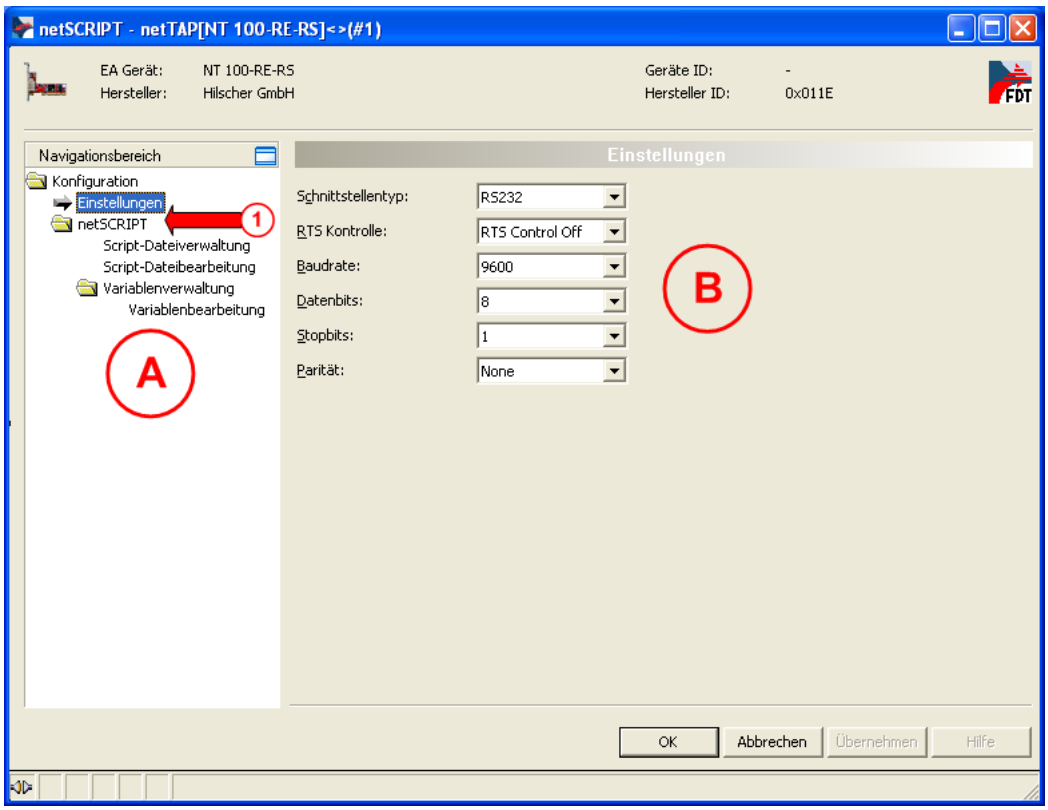


Abbildung 3: SYCON, UART-Einstellungen

Die hier im Fensterbereich Ⓑ gemachten Einstellungen werden nicht automatisch für die UART-Schnittstelle wirksam. Sie müssen im Skript mit der **Funktion PortReadConfigDb()** siehe Abschnitt 7.1.1.1 Seite 82 ausgelesen und beim Öffnen der Schnittstelle übergeben werden.

➢ Wählen Sie im Fensterbereich Ⓐ netSCRIPT aus, um in die Skriptdateiverwaltung zu gelangen.

3.2 Editor für Programme

3.2.1 Skriptdateiverwaltung

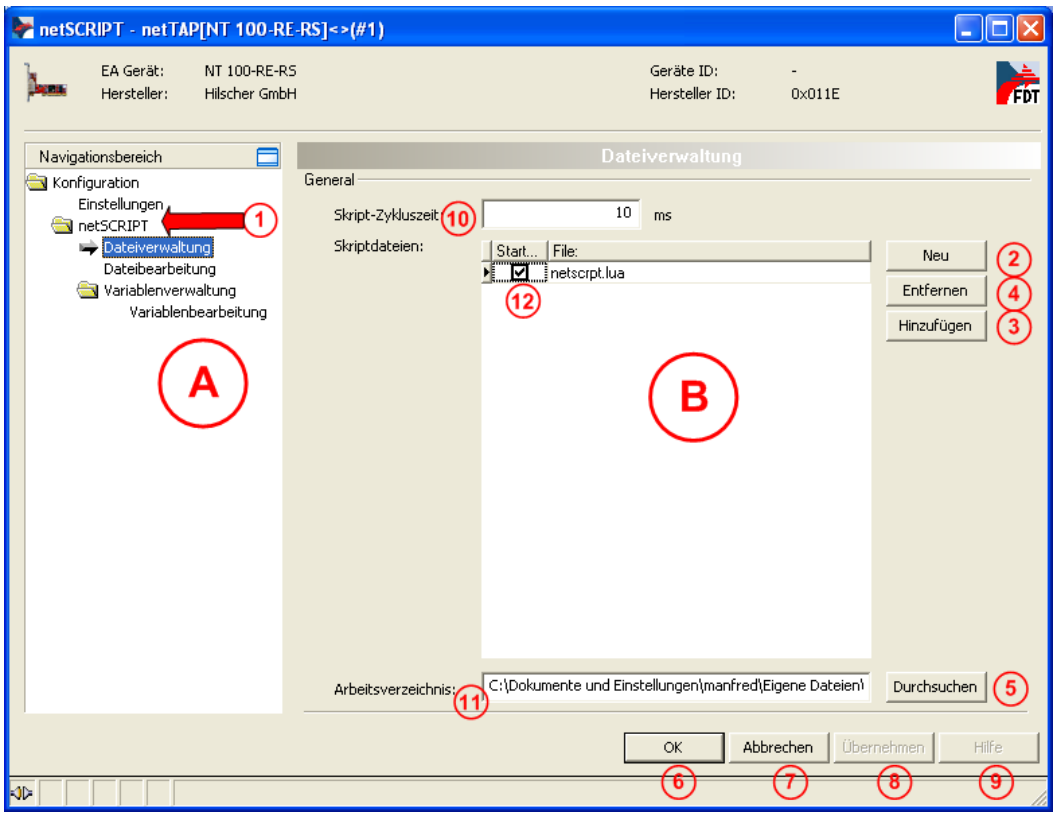


Abbildung 4: Scriptverwaltung

① Wählen Sie im Navigationsbereich Ⓐ die Zeile „netSCRIPT“ bzw. „Dateiverwaltung“ aus. Damit kommen Sie in den dargestellten Konfigurationsbereich.

Ⓑ In diesem Fensterbereich werden die zur Bearbeitung verfügbaren Skript-Dateien aufgelistet.

② Mit der Schaltfläche „Neu“ können Sie eine neue Skriptdatei anlegen, die einen Standardnamen „netscript.lua“ erhält (wie im Fensterbereich Ⓑ dargestellt).

③ Mit dieser Schaltfläche kann eine Skriptdatei in das SYCON-Projekt importiert werden. Dabei wird sie gleichzeitig in dem unter ⑪ ausgewähltem Arbeitsverzeichnis kopiert.

④ Mit dieser Schaltfläche wird die im Fensterbereich Ⓑ ausgewählte Datei aus Fensterbereich Ⓑ entfernt und aus dem ⑪ Arbeitsverzeichnis gelöscht.

⑤ Mit dieser Schaltfläche kann das Arbeitsverzeichnis ⑪ ausgewählt werden, in dem die Skript-Dateien abgelegt werden.

⑥ Mit dieser Schaltfläche werden die Einträge dieses Fensters gespeichert und anschließend in das aufrufende Fenster zurückgesprungen.

⑦ Mit dieser Schaltfläche brechen Sie diesen Bearbeitungsschritt ohne zu speichern ab und gelangen in das Hauptfenster des SYCON.net.

⑧ Mit dieser Schaltfläche werden die bisher erstellten Daten gespeichert. Das Fenster bleibt zur weiteren Bearbeitung geöffnet.

⑨ Mit dieser Schaltfläche gelangen Sie in die Hilfe für dieses Fenster.

⑩ In dieser Eingabefläche ist die zyklische Aufrufzeit des Scriptes im Gerät einzustellen.

⑪ Hier wird das Verzeichnis angezeigt, in dem die Skriptdatei gespeichert wird.

⑫ Mit der Schaltfläche „Start“ wird festgelegt, welche Skriptdatei zyklisch gestartet werden soll.

3.2.2 Skriptbearbeitung

Im folgenden Fenster kann ein Skript angelegt oder ein bestehendes editiert werden.

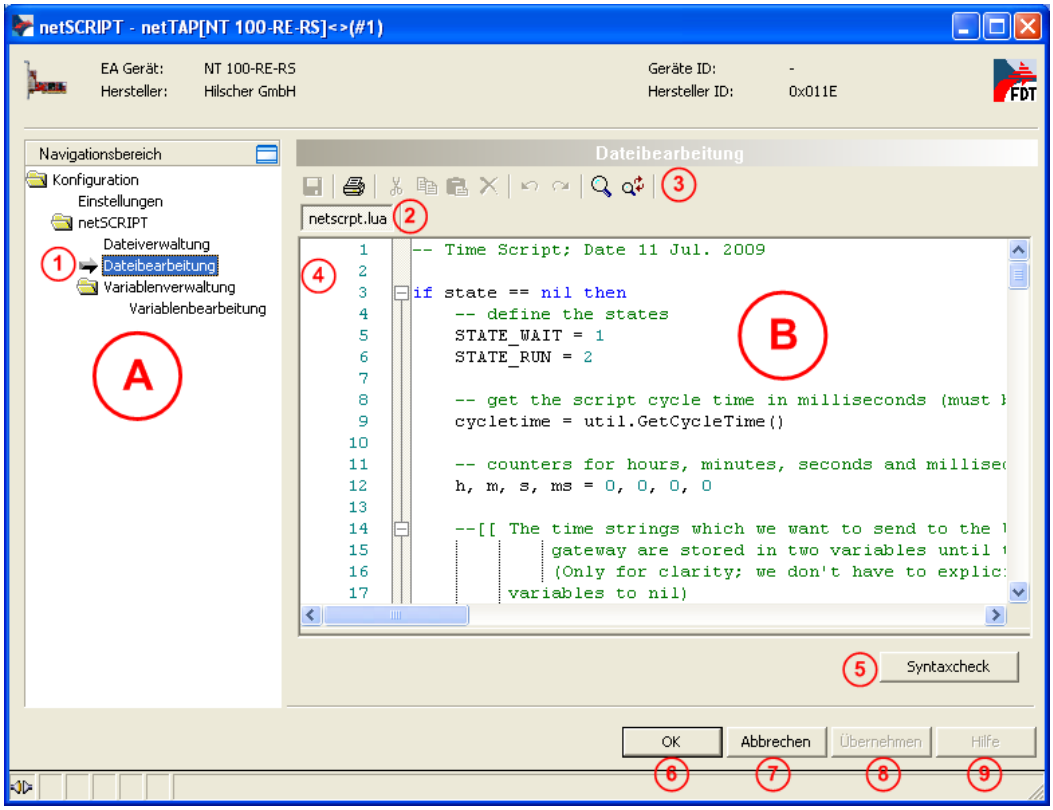


Abbildung 5: Editor Fenster

① Wählen Sie im Navigationsbereich Ⓐ den Eintrag „Dateibearbeitung“ aus. Damit werden Ihnen in der Zeile ② die bearbeitbaren Skriptdateien dargestellt

Ⓑ In diesem Fensterbereich wird Ihnen die ausgewählte Skriptdatei zur Bearbeitung präsentiert.

③ In dieser Zeile werden Ihnen Bearbeitungswerkzeuge für das Script angeboten.

④ In dieser Spalte werden Ihnen die Zeilennummern des Scriptcodes angezeigt.

⑤ Mit dieser Schaltfläche wird die aktuelle im Editorfenster angezeigte Skriptdatei einer syntaktischen Prüfung unterzogen.
Es öffnet sich der zusätzliche Fensterbereich Ⓒ, siehe Abbildung 6, in dem die Ergebnisse der Überprüfung angezeigt werden.

⑥ Mit dieser Schaltfläche werden die Einträge dieses Fensters gespeichert und in das aufrufende Fenster zurückgesprungen.

⑦ Mit dieser Schaltfläche brechen Sie diesen Bearbeitungsschritt ohne zu speichern ab und gelangen in das Hauptfenster des SYCON.net.

⑧ Mit dieser Schaltfläche speichern Sie den aktuellen Bearbeitungszustand, ohne das Fenster zu verlassen.

⑨ Mit dieser Schaltfläche können Sie eine Hilfe zu diesem Dialog aufrufen.

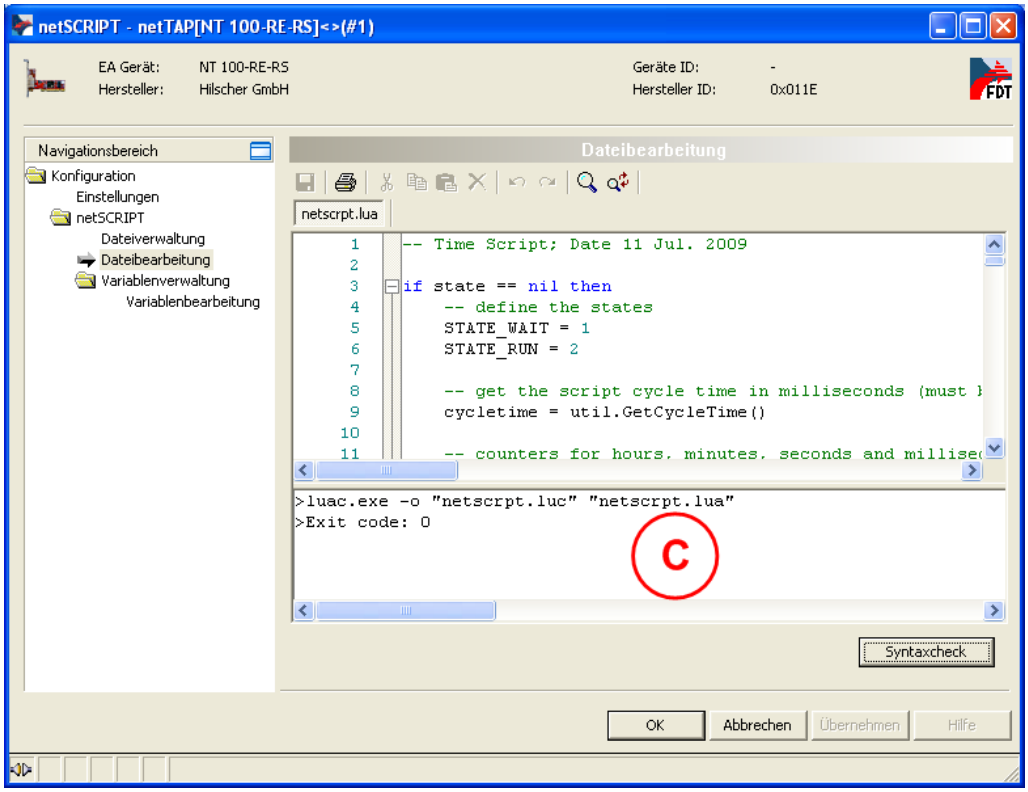


Abbildung 6: Editor Fenster kompiliert

In der obigen Abbildung, wird im Fensterbereich Ⓒ, das Resultat der Kompilierung angezeigt. Ist der letzte Eintrag in diesem Fensterteil ungleich „>Exit code: 0“, ist aus den vorangestellten Zeilen ersichtlich, wo der Fehler innerhalb der Skriptdatei zu suchen ist.

3.3 Bedienbare Variable (Parameter)

netSCRIPT-Variable werden grundsätzlich im Skriptprogramm selbst entweder lokal oder global angelegt und mit Werten vorbelegt. Über SYCON.net können außerhalb des Skriptprogramms zusätzliche bedienbare Variablen (Parameter) definiert und ihnen Werte zugewiesen werden, die dann vom Skript ausgewertet werden können.

Beim Download der Konfiguration und des Skriptes werden diese Variable mit ihren Werten an das Zielgerät übertragen und dort in der netSCRIPT-Tabelle VAR abgespeichert. Die Zusatzvariablen ermöglichen ohne den Skriptcode anfassen zu müssen eine flexible Steuerung des Programms alleine durch Veränderung der Variablenwerte.

Das Auslesen dieser Variablen in netSCRIPT ist in Abschnitt 3.3.2.8 auf Seite 28 beschrieben.

3.3.1 Definition bedienbarer Variable (Variablenverwaltung)

➢ Wählen Sie im Teilfenster Ⓐ (Navigationsbereich) unter „Konfiguration > netSCRIPT > Variablenbearbeitung“ ① aus.

➧ Es öffnet sich folgendes Fenster.

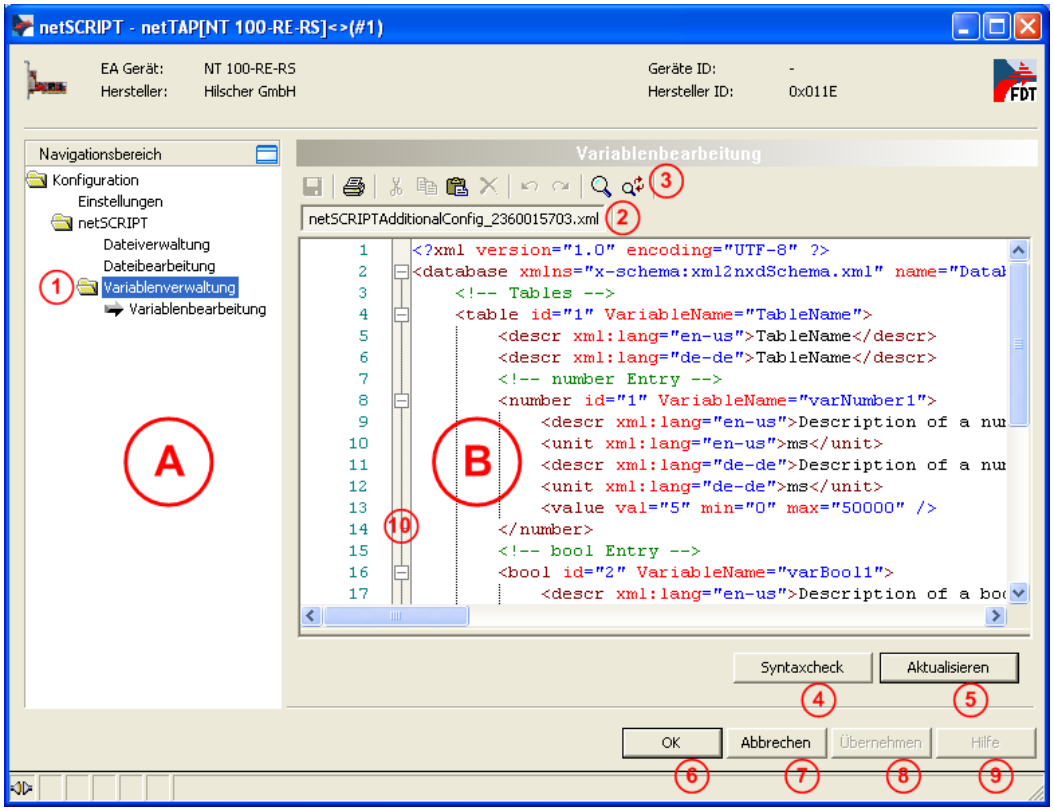


Abbildung 7: Bedienbare Variablen Definition

Ⓑ In diesem Fensterbereich sehen Sie eine .xml-Datei, mit der die bedienbaren Variablen des netSCRIPT-Programms erstellt werden. Die Syntax der Datei ist im Abschnitt 3.3.2 auf Seite 23 erklärt.

② In dieser Zeile sehen Sie den Dateinamen der Variablendatei, die in dem Fensterbereich Ⓐ editiert werden kann.

③ In dieser Zeile werden Ihnen Bearbeitungswerkzeuge für die XML-Datei angeboten.

④ Mit dieser Schaltfläche kann mit der XML-Datei ein Sytaxcheck durchgeführt. Ist der Syntaxcheck nicht erfolgreich, öffnet sich der Fensterbereich Ⓒ mit entsprechenden Fehlermeldungen (siehe Abbildung 8).

⑤ mit dieser Schaltfläche wird der Inhalt des Fensterbereichs Ⓑ kompiliert und der Inhalt im Fensterbereich Ⓐ aktualisiert. Ist die Kompilierung nicht erfolgreich, öffnet sich der Fensterbereich Ⓒ mit entsprechenden Fehlermeldungen(siehe Abbildung 8).

⑥ Mit dieser Schaltfläche wird das Script gespeichert und anschließend in das aufrufende Fenster zurückgesprungen.

⑦ Mit dieser Schaltfläche brechen Sie diesen Bearbeitungsschritt ohne zu speichern ab und gelangen in das Hauptfenster des SYCON.

⑧ Mit dieser Schaltfläche wird die Bearbeitung abgebrochen und ins nächst höhere SYCON-Fenster zurückgekehrt.

⑨ Mit dieser Schaltfläche können Sie eine Hilfe zur Bearbeitung aufrufen.

⑩ In dieser Spalte können mit der Mouse, durch ein Klick auf den Kästchen mit dem „-„ bzw. „+“ Zeichen, zusammenhängende Blöcke minimiert bzw. expandiert werden.

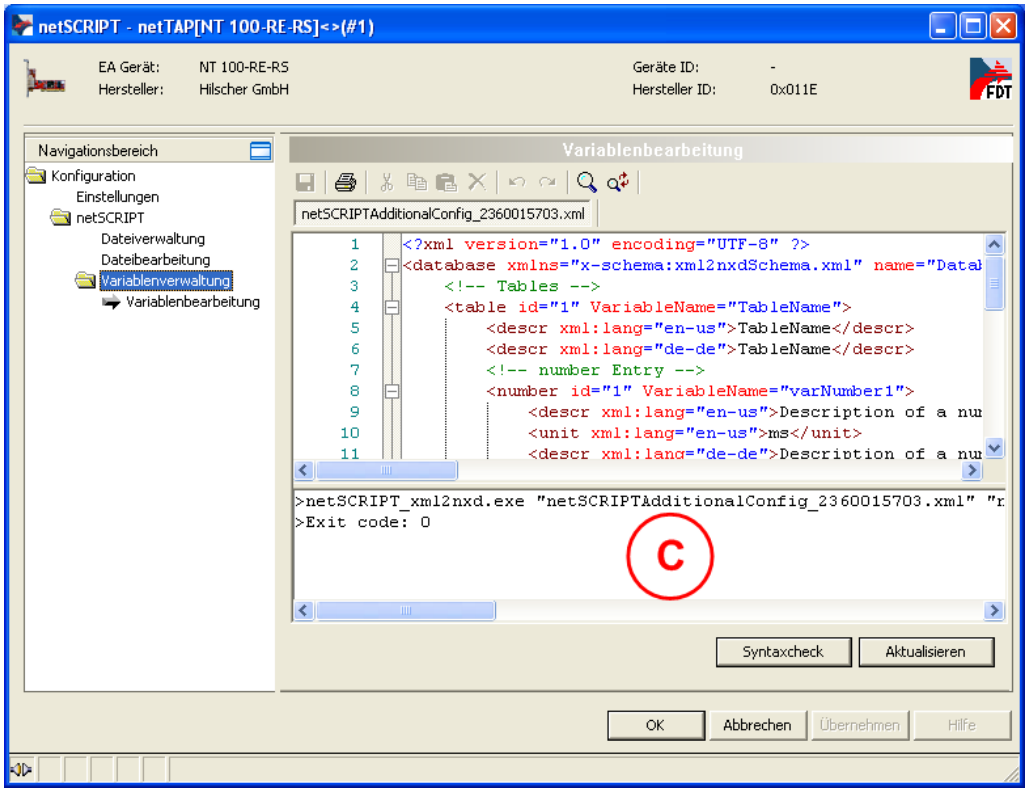


Abbildung 8: Bedienbare Variablen Definition kompiliert

Im obigen Bild wird im Fensterbereich Ⓒ das Ergebnis der Kompilierung angezeigt. Ist der letzte Eintrag in diesem Fensterteil ungleich „**>Exit code: 0**“ ist aus den vorherigen Zeilen ersichtlich, wo der Fehler in der Variablendefinition zu suchen ist.

3.3.2 Aufbau der XML-Datei für bedienbare Variable

Zu jedem netTAP-Gerät kann nur eine Variablenliste existieren. Die Variablenliste ist HTML-ähnlich aufgebaut. Zu jedem Beginntag „**<aaa>**“ gibt es eine Endetag „**</aaa>**“ bzw. einem Kommentar „**<!--**“ Beginn und „**-->**“ Ende.

Die XML-Datei für die bedienbaren netSCRIPT-Variablen besteht im Wesentlichen aus 3 Bereichen.

1. Einen Dateikopf.
2. Dem Dateikörper mit den Variablen.
3. Dem Dateiende.

Es können (müssen aber nicht), für eine Variable mehrere Ausgabesprachen definiert werden. Diese sind unabhängig von der netSCRIPT-Variablen, die der Wert zugewiesen wird.

Im Folgenden werden die Einzelheiten der drei Abschnitte behandelt.

3.3.2.1 Der Dateikopf

Im Dateikopf dürfen die Zeilen 1...4 nicht verändert werden.

In den Zeilen 5 und 6 wird festgelegt in welcher Sprache (Installationssprache des SYCON) die Variablenbenennung angezeigt werden soll. Es sind derzeit die Sprachen Deutsch und Englisch möglich. Auch weitere Sprachen wären möglich, wenn das Programm SYCON.net diese unterstützt.

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

Tabelle 8: XML-Code - Dateikopf

Erläuterungen zu den Zeilen:

Zeile 1:

Interpreterversionsnummer und Zeichensatzinformation. Diese Zeile darf nicht verändert werden.

Zeile 2:

Enthält verarbeitungsinterne Informationen. Diese Zeile darf nicht verändert werden.

Zeile 3:

Diese ist eine Kommentarzeile und kennzeichnet den Beginn der Definitionsbeschreibungen.

Zeile 4

Kennzeichnet den Beginn der Variablentabelle mit der id="1". Zu einem Tabellenbeginn **<table id="1">** gehört immer eine entsprechende Endekennung **</table>**. Zwischen diesen beiden Tags befindet sich die Beschreibung der Variablen der Tabelle „**id="1"**".

Hinter dem Variablenname **VariableName** kann zwischen den folgenden Anführungszeichen ein Tabellennahme angegeben werden, unter dem diese Tabelle in netSCRIPT angesprochen werden kann.

In einer Variablen-Beschreibungsdatei kann es mehr als eine Variablentabelle geben. Diese Tabellen unterscheiden sich in ihrer „**id**"-Nummer und dem Tabellennamen **VariableName**.

Zeile 5 und 6

In diesen Zeilen werden die Sprachen festgelegt, in denen die Variablenbeschreibungen (in Abhängigkeit der gewählten SYCON-Sprache) angezeigt werden sollen. **xml:lang="en-us"** für Englisch und **xml:lang="de-de"** für Deutsch. Mit **>TableName<** kann zwischen den beiden Winkelzeichen ein Benennungstext für die Tabelle zur jeweiligen Sprache angegeben werden.

3.3.2.2 Der Dateikörper

Im Dateikörper werden die einzelnen Variablen mit allen ihren Anzeige- und Datenvariablen beschrieben.

Die einzelnen Variablen sind mit einer „**<number id = „n">**" gekennzeichnet. Hierbei kennzeichnet „**n**" die Nummer der Position, an der die Variable später in der Variablenliste (in der Variablenverwaltung von SYCON.net) dargestellt werden soll. Daher muss diese Nummer in der gesamten Datei eindeutig sein.

Im Folgenden werden die einzelnen möglichen Variablen aufgelistet und erläutert.

3.3.2.3 Numerische Variable

Bei einer numerische Variable wird nicht zwischen einer Ganzzahl und einer Dezimalzahl unterschieden.

Zeilen-Nr.	xml-Code; numerische 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>`

Tabelle 9: XML-Code - Numerische Variable

Erläuterungen zu den Zeilen:

Zeile 7

Kommentar zum Beginn der Variablendefinition

Zeile 8

Benennung des Variablentyp „**<number**„und der Position „**id="1"**“in der Anzeigeliste. Jede Positionsnummer darf für die Tabelle nur einmal vergeben werden!

Hinter **VariableName=** steht in Anführungszeichen der Variablennamen, unter dem diese Variable in netSCRIPT ansprechbar ist.

Zeile 9

In dieser Zeile wird die beschreibende Benennung der Variablen „**Description of a number**“ zur Ausgabesprache **"en-us"** angegeben. Die Benennung ist beliebig wählbar.

Zeile 10

In dieser Zeile wird eine Maßeinheit „**ms**“ zur Ausgabesprache **"en-us"** angegeben.

Zeile 11

Entsprechend zur Zeilennummer 9 erfolgt hier die Benennung der Variablen in der Sprache Deutsch **"de-de"**.

Zeile 12

Entsprechend zur Zeile 10 wird hier die Ausgabe der Maßeinheit in der Sprache Deutsch festgelegt.

Zeile 13

In dieser Zeile wird der Wertebereich mit einem Defaultwert „**val="5"**“ sowie ein Minimalwert „**min="0"**“ und ein Maximalwert „**max="50000"**“ festgelegt.

Zeile 14

Beinhaltet den Endetag „**</number>**“ der Definition der numerischen Variablen.

3.3.2.4 Bool-Variable

Beispiel der Definition einer Bool-Variablen.

Zeilen-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>

Tabelle 10: xml-Code, Bool- Variable

Erläuterungen zu den Zeilen:

Zeile 15

Kommentar zum Beginn der Variablendefinition.

Zeile 16

Benennung des Variablentyp „**<bool>**“ und der Position „**id="2"**“ in der Anzeigeliste. Jede Positionsnummer darf für die Tabelle nur einmal vergeben werden!

Hinter **VariableName=** steht in Anführungszeichen der Variablennamen, unter dem diese Variable in netSCRIPT ansprechbar ist.

Zeile 17

In dieser Zeile wird die beschreibende Benennung der Variablen „**Description of a bool**“ zur Ausgabesprache **"en-us"** angegeben. Die Benennung ist beliebig wählbar.

Zeile 18

In dieser Zeile wird festgelegt, wie der Zustand „**false**“ In der Sprache „**en-us**“ dargestellt werden soll. Hier „**off**“.

Zeile 19

In dieser Zeile wird festgelegt, wie der Zustand „**true**“ In der Sprache „en-us“ dargestellt werden soll. Hier „**on**“.

Zeile 20

In dieser Zeile wird die beschreibende Benennung der Variablen „**Beschreibung einer bool Variablen**“ zur Ausgabesprache **"de-de"** angegeben. Die Benennung ist beliebig wählbar.

Zeile 21

In dieser Zeile wird festgelegt, wie der Zustand „**false**“ In der Sprache „**de-de**“ dargestellt werden soll. Hier „**aus**“.

Zeile 22

In dieser Zeile wird festgelegt, wie der Zustand „**true**“ In der Sprache „de-de“ dargestellt werden soll. Hier „“**ein**“.

Zeile 23

In dieser Zeile wird mit „**value val="1"**“ angegeben, welcher Defaultwert die Variable haben soll. Mit „**min="0" max="1"**“ wird der mögliche Minimalwert bzw. Maximalwert festgelegt. Bei einer Bool-Variablen ist dieses immer 0 / 1.

Zeile 24

Enthält der Endetag „**</bool>**” der Definition der Bool-Variablen.

3.3.2.5 String

Eine Stringvariable kann eine maximale Stringlänge von 64 Bytes aufnehmen. Sie wird wie folgt definiert:

Zeilen-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>`

Tabelle 11: XML-Code - String Variable

Erläuterungen zu den Zeilen:

Zeile 25

Kommentar zum Beginn der Definition der Stringvariablen.

Zeile 26

Beginntag der Stringdefinition mit der Positionsangabe **„id=“6“**“ in der Variablenliste.
Hinter **VariableName=** steht in Anführungszeichen der Variablennamen, unter dem diese Variable in netSCRIPT ansprechbar ist.

Zeile 27

Beginntag **<descr xml:lang="en-us"**> des Beschreibungstextes des Strings in der bedienbaren Variablenliste, hier für die Sprache „en-us“. Nach dem Beginntag folgt der Ausgabetext, gefolgt vom Endetag **</descr>** der Beschreibung.

Zeile 28

Beginntag **<descr xml:lang="de-de"**> des Beschreibungstextes des Strings in der bedienbaren Variablenliste, hier für die Sprache „de-de“. Nach dem Beginntag folgt der Ausgabetext, gefolgt vom Endetag **</descr>** der Beschreibung.

Zeile 29

Der Tag mit dem Textinhalt zwischen den Anführungszeichen von **val ="...“** der Variablen und der minimalen **min="0"** und maximalen **max="64**" Längenvorgabe.

Die Länge des Strings wird nur durch die Längenangabe **max=“n**“ begrenzt.

Zeile 30

Endetag der Stringvariablen.

3.3.2.6 Dateiende

Zeilen-Nr.	xml-Code; Dateiende
31	</table>
32	</database>

Tabelle 12: XML-Code - Dateiende

Erläuterungen zu den Zeilen:

Zeile 31

Endetag der Tabellendefinition.

Zeile 32

Endetag der Datendefinition.

3.3.2.7 Löschen einer Tabelle mit Variablendefinitionen

Soll eine Tabelle mit Variablendefinitionen gelöscht werden, darf nicht die gesamte XML-Datei gelöscht werden. Es darf nur der Teil zwischen dem Tag

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

und dem Tag

```
</database>
```

gelöscht werden.

3.3.2.8 Aufruf der Variablen unter netSCRIPT

Die in diesem Abschnitt beschriebenen Variablen werden automatisch beim Download ins netTAP transferiert und sind dort wie folgt aufrufbar:

1. **VAR[“TableName“][“VariableName“]**
2. Ist der Tabellenname und/oder der Variablenname nur numerisch kann auch folgende Schreibweise gewählt werden:
   VAR[1][2]
   wobei hier die die Ziffer „1“ für die Tabellennummer und die Ziffer 2 für die Variablennummer steht.
3. Ist ein Namensteil alphanumerisch, kann dafür auch folgende Schreibweise gewählt werden:

   VAR.TableName. VariableName

3.3.3 Darstellung der bedienbaren netSCRIPT Variablen

Nach der Definition der bedienbaren Variablen, deren Syntaxcheck und Aktualisierung in der Abbildung 7: Bedienbare Variablen Definition aus Kapitel 3.3.1, wird der Navigationsbereich (Teilfenster Ⓐ) unter dem Ordner Variablenverwaltung um den Eintrag des definierten Tabellennamens ① erweitert. Hier können Sie mit der Auswahl der definierten Variablen-Tabelle ① sich im Teilfenster Ⓑ die Variabeln anzeigen lassen und diese mit Werten versehen.

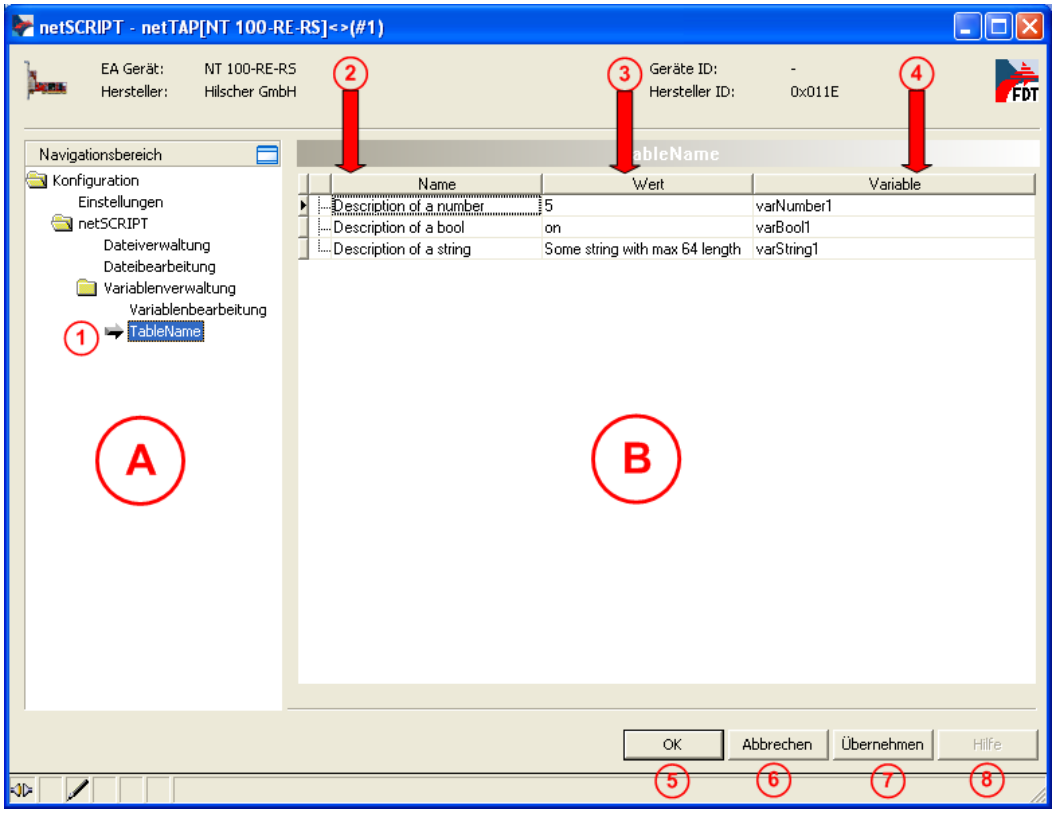


Abbildung 9: Bedienbare Variablen Darstellung

Im Teilfenster Ⓑ sehen Sie in der Spalte:

② den Variablennamen in der aktuellen SYCON Sprachauswahl,

③ den eingestellten Wert der Variablen. Dieser kann hier nach der Anwahl mit dem Courser verändert werden,

④ den Namen der Variablen im netSCRIPT.

Mit der Schaltfläche:

⑤ speichern Sie die aktuellen Einstellungen und gelangen in das aufrufende Fenster zurück,

⑥ verlassen Sie das Bearbeitungsfenster ohne die aktuellen Werte zu speichern,

⑦ speichern Sie die aktuellen Variablenwerte und gelangen in das aufrufende Fenster zurück,

⑧ Mit dieser Schaltfläche können Sie sich die Hilfe zu diesem Fenster aufrufen.

4 Die Sprache netSCRIPT

Die Scriptsprache netSCRIPT basiert auf der Scriptsprache Lua.

Für die Anwendung innerhalb Hilscher-Geräten, und deren spezielle Belange für die Datenübertragung in Netzwerken der Automatisierungstechnik, wurde sie funktionell erweitert.

Das in SYCON.net geschriebene Programm wird mit dem Download der Gesamtkonfiguration an das Zielgerät übertragen. In der Regel unterliegen die vom Skriptprogramm abgearbeiteten Funktionen im Vergleich zum übergeordneten Netzwerkprotokoll keiner Echtzeitanforderung. Aus diesem Grund hat netSCRIPT in der Prioritätenverteilung innerhalb des preemptiven Betriebssystems des Zielgerätes stets eine relativ niedrige Priorität. Es kommt daher vor, dass das Skriptprogramm während des Programmablaufes vom Betriebssystem kurzzeitig unterbrochen wird. Die Unterbrechung und Rückkehr geschieht automatisch und hat auf den funktionellen Ablauf des Skriptprogramms keinen Einfluss. Der Anwender muss hierbei keine speziellen Vorkehrungen im Programm treffen. Generell gilt, dass netSCRIPT aufgrund seiner niedrigen Priorität im System nie die Echtzeiteigenschaften der restlichen Abläufe beeinflussen kann.

Die Firmware des Zielgerätes bedient sich des Programms und ruft es zyklisch im vorgegebenen Zeitraster immer mit der ersten Codezeile auf und beendet es mit der letzten. Dabei wird das Programm zeilenweise zur Laufzeit interpretiert und nicht wie bei anderen Programmiersprachen üblich als vorkompilierter Code ausgeführt. Bei Lua wird der Quelltext hierzu beim Laden in einen kompakten Code für eine Virtual Maschine übersetzt.

In der Regel wirken die bereitgestellten Funktionen nicht suspendierend auf den aufrufenden Programmcode, sondern kehren nach ihrer Ausführung immer sofort wieder zurück. Funktionen mit suspendierendem Verhalten, die zum Beispiel spezielle Funktionen des Betriebssystems nutzen könnten, erhalten in der Beschreibung einen entsprechenden Hinweis. Bitte beachten Sie dabei, dass das zyklische Aufrufzeitraster nicht mehr gewahrt werden kann, sobald das Skriptprogramm nicht innerhalb der Zykluszeit aus einem Aufruf wieder zurückkehrt.

Aufgrund des zyklischen Aufrufprinzips hat sich bewährt ein Skriptprogramm über eine sogenannte Statusmaschine oder Schrittkette zu programmieren und pro Durchlauf immer nur kleine und ausgewählte Programmteile abzuarbeiten. Dabei dient eine globale Laufvariable als Indikator welcher Programmteil im nächsten Zyklus abgearbeitet werden soll. Eine Prüfung des Inhaltes der Variable z.b. mit „if“ bei Eintritt in den Skriptcode ist dabei üblicher Weise der Schrittkette vorangestellt, um über die Werte in die einzelnen Programmteile zu verzweigen.

Es ist natürlich erlaubt innerhalb des Programms auch zeitlich ausgedehnte Schleifen zu programmieren die dazu führen können, dass das Zyklusraster nicht eingehalten werden kann. Im Extremfall kann sogar das Programm gar nicht mehr aus einem Aufruf zurückkehren. Das führt zu keiner weiteren Beeinflussung des Systems oder gar einer Fehlermeldung oder Skriptabbruches. Verpasste Zyklen werden automatisch mitgezählt und später nachgeholt. Hierzu wird das Skriptprogramm außerhalb des zyklischen Rasters einfach um die verpasste Anzahl direkt hintereinander aufgerufen ohne die Zykluszeit dabei verstreichen zu lassen.

Bei der Verwendung des Skript-Debbugers verliert das Skriptprogramm seine Echtzeiteigenschaften, der zyklische Aufruf wird unterbrochen. Der Debugger ist ein sogenannter Soft-Debugger. Beim Anhalten des Programms stoppt er weder Betriebssystem, die CPU oder ihre Peripherie. Lediglich das Skriptprogramm und sein Ablauf kann mit dem Debugger analysiert und gesteuert werden. Während des Debuggens läuft der Rest des Systems wie zum Beispiel das übergeordnete Netzwerk ungestört weiter.

Hinweis: Das Skriptprogramm wird zyklisch mit der im SYCON.net vorgegebenen Zeit aufgerufen. Die Einstellung der Zykluszeit ist im Abschnitt 3.2.1 beschrieben.

Wichtig: In netSCRIPT ist es möglich im aktuellem Skript alle vorgegebenen Variablen, Funktionen und Tabellen über-/umzudefinieren. Daher ist die Auswahl, insbesondere von Variablen- und Funktionsnamen besonders sorgfältig vorzugehen!

Eine Programmzeile wird mit der **ENTER**-Taste abgeschlossen. Sie kann unmittelbar vor dem **ENTER** noch ein „;“ Semikolon enthalten. Das Semikolon dient auch zur Trennung zweier Anweisungen.

4.1 Syntax und Schlüsselworte

Hinweis: netSCRIPT unterscheidet zwischen Groß- und Kleinschreibung: and ist ein reserviertes Schlüsselwort, And und AND sind zwei unterschiedliche Variable.

4.1.1 Kommentare

Kommentare in der Zeile beginnen mit „--“ `--` alles was hinter den zwei Minuszeichen steht liest der Übersetzer nicht

Mehrzeilige Kommentare werden über zwei rechteckige Klammern hinter den zwei Minuszeichen: `--[[` für den Anfang und `--]]`für das Ende gebunden.

Hinweis: Empfehlung für den Editor:

Soll beim Programmtest zeitweise ein Codeabschnitt nicht durchlaufen werden, empfiehlt sich folgende Vorgehensweise:

1	`CODEBLOCK1`	Wird immer ausgeführt
2	`--[[`	Kommentar Beginn
3	`CODEBLOCK2`	Wird nicht ausgeführt
4	`--]]`	Kommentar Ende
5	`CODEBLOCK 3`	Wird immer ausgeführt

Wird in der Zeile des Kommentarbeginns zwischen den beiden Minuszeichen „`--` „ und den beiden geöffneten eckigen Klammern„`[[` „ ein Leerzeichen eingefügt, werden die Zeilen 2 und 4 zu eigenständigen Kommentarzeilen und der der **`CODEBLOCK2`** wird ausgeführt.

4.1.2 Schlüsselworte

Schlüsselworte sind Worte, die für Variable oder Funktionsnamen nicht verwendet werden dürfen, da sie als Befehle interpretiert werden.

Dieses sind:

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

4.2 Variable

Variable werden in netSCRIPT nicht deklariert und nicht typisiert. Die Typisierung wird durch die Wertzuweisung erreicht.

4.2.1 Variablennamen

• Eine Variable muss mit einem Buchstaben oder Unterstrich beginnen.

• Es dürfen keine anderen Zeichen als Buchstaben, Zahlen und Unterstriche verwendet werden.

• Groß und Kleinschreibung wird im Namen unterschieden.

4.2.2 Zuweisungen

Eine Zuweisung hat die Form:

```
Name = Ausdruck
```

Variablen werden mit dem „**=**“ Zeichen Werte zugewiesen. Dabei steht der Variablenname links vom Gleichheitszeichen und der zuzuweisende Wert rechts vom Gleichheitszeichen.

Es ist auch möglich einer Liste von Variablen eine Liste von Werten zuzuweisen, z.B. wie folgt:

```
Name1, Name2, Name3 = Ausdruck1, Ausdruck 2, Ausdruck 3
```

Da globale Variable auch nach dem Programmdurchlauf ihren Wert behalten, ist für den Erststart ggf. folgende Zuweisung sinnvoll.

```
var1 = var1 OR 1
```

Hierbei wird der Variablen **var1** der Wert zugewiesen, den die Variable zum Ende des vorherigen Programmdurchlaufs hatte. Sollte das Programm das erste Mal durchlaufen werden, oder die Variable hatte am Ende des vorherigen Zyklus den Wert ***nil***, so wird **var1** der Wert ***1*** zugewiesen.

Für Variable, die außerhalb des zyklischen Programmaufrufes definiert sind, z.B. wenn sie über SYCON.net vordefiniert wurden oder weil das Skript eine Zustandmaschine mit getrenntem Initialisierungsteil enthält, ist obiges Vorgehen nicht erforderlich.

4.2.3 Gültigkeitsbereiche

Variable können global oder lokal gültig sein.

Globale Variable sind im gesamten Script ansprechbar.

Sie behalten am Ende des Programms ihren Wert und stehen beim erneuten (zyklischen) Programmstart mit ihrem Wert aus dem vorherigen Zyklus erneut zur Verfügung.

Sie können im gesamten Script (auch innerhalb von Blöcken) definiert werden.

Globale Variable sind Einträge in der Tabelle _G.

Lokale Variable sind nur in dem Block gültig, in dem sie erzeugt werden und sie existieren nur so lange bis der Block mit „**end**“ verlassen wird.

Eine lokale Variable wird definiert, indem das Wort „**`local`**„ vor den Variablennamen gesetzt wird.

Der Zugriff auf lokale Variable ist effizienter als auf globale Variable. Lokale Variable empfehlen sich für alle Variable, die nur lokal, aber häufig benötigt werden.

Es ist auch möglich, globale Variable oder Funktionen, die z.B. in einer Schleife n-mal aufgerufen werden, vorübergehend in einer lokalen Variable zu speichern.

Werte lokaler Variablen können auch bei Namensgleichheit in die globale Variable zurückgeschrieben werden, wenn man vor dem globalen Variablennamen dem Präfix der Umgebungstabelle (im Regelfall „_G.“ setzt. Damit wird die Variable in der Tabelle _G angesprochen. Siehe auch Abschnitt 4.3 .

Beispiel für die Gültigkeit von lokalen Variablen, hier am Beispiel der Variablen „a“ und „x“.

Code	Ausgabe:

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

Ohne die Zuweisung „**_G.x = x**“ am Ende der **do**- Schleife in der obigen Tabelle 13 hätte „**x**“ nach dem „**end**“ der Schleife den Wert „5“.

4.2.4 Typen

Es gibt folgende Datentypen:

Typ	Bedeutung / Verwendung
Nil	Nichts, leer, nicht vorhanden.
Zahlen	Ganze und gebrochene Zahlen, Zahlen in Exponentialdarstellung
Zeichen String	Strings werden mit den Zeichen "Text" oder 'Text' in der Zeile begrenzt. Das Wort „Text" stellt hier den Inhalt des Strings dar. Strings, die über mehrere Zeilen gehen, werden beginnend mit [[und endend mit]] eingeschlossen.
Wahrheitswerte	true / false, **nil** wird als **false** interpretiert. alle anderen Werte werden als **true** interpretiert.
Funktionen	Sind Spezialfälle eines Blocks
Tabelle	Tabelle, die unter einem Index (Zahl, Name, sonstiges Objekt) einen beliebigen Wert speichert. z.B. t[1]=13 oder t[„surname"]="Einstein"

Tabelle 13: Übersicht der Typen

Der Typ eines Ausdrucks v kann mit der Funktion „type(v)" abgefragt werden (siehe Abschnitt 5.1.21 auf Seite 53).

4.2.4.1 Nil

Der Wert (und der Typ) ***nil*** ist ein Platzhalter für nicht vorhandene Werte bzw. nicht belegte Variablen. Außerdem ist er in Wahrheitsausdrücken der Regel gleichbedeutend mit **false**

Wird auf eine Variable zugegriffen, der noch kein Wert zugewiesen wurde, ist das Ergebnis der Wert ***nil***. Es wird in diesem Fall keine Fehlermeldung ausgegeben.

Trägt man in einer Tabelle unter einem bestimmten Index den Wert ***nil*** ein (***t[x]=nil***), wird der Tabelleneintrag unter x, falls einer existiert hat, gelöscht.

Weist man einer globalen Variablen (**var1**) den Wert ***nil*** zu, ist diese Variable danach nicht mehr existent, d.h. der von ihr belegte Speicherbereich wird vom Garbage Collector freigegeben. Lokalen Variablen wird der Wert ***nil*** zugewiesen.

```
var1 = nil
```

4.2.4.2 Zahlen

Zahlen werden im netSCRIPT als 64 Bit Fließkommazahl und Double Floats mit 52 Bit Mantisse abgebildet.

Zahlen können als Integer, mit oder ohne Vorzeichen, optional mit einem Dezimalanteil, optional mit einem expotential Teil, oder als ganze Zahl eingegeben werden.

Variablen können ganze, gebrochene, negative Zahlen zugewiesen werden. Als Dezimalzeichen wird der „**.**" (Punkt) verwendet! Negativen Zahlen wird das Minuszeichen „-„ unmittelbar der Ziffer vorangestellt.

Folgende Zuweisungen sind dabei Gleichwertig:

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

Tabelle 14: Zahlenzuweisungen

Enthält ein String nur Ziffern, wird er von netSCRIPT auch als Zahl interpretiert. (Ausnahme bei Vergleichen!)

4.2.4.3 String

Strings werden mit den Zeichen "Text" oder 'Text' in der Zeile begrenzt. Das Wort „Text" stellt hier den Inhalt des Strings dar.

Strings können beliebige Bytefolgen enthalten (auch 0x0).

Strings, die über mehrere Zeilen gehen, werden beginnend mit [[und endet mit]] eingeschlossen.

Zuweisung:

```
Ortsname = "Hattersheim am Main"
Strasse = 'Reinstrasse 15'
varTyp1 = "function"
dieGeschichte = [[ begann irgendwann
                        und wird auch einmal enden]]
```

4.2.4.4 Boolesche Werte

Sind ein Typ, der nur zwei Zustände kennt: ***true*** (wahr) und ***false*** (unwahr).

Zuweisung:

```
Sonnenschein = true
Nacht = false
hell = true
```

Die Zuweisung erfolgt mit den reservierten Wörtern **true** und **false**.

Der Wert ***nil*** einer Variablen wird als ***false*** interpretiert. Alle Werte außer ***nil*** und ***false*** werden in der Regel als ***true*** interpretiert.

4.2.4.5 Funktionen

Funktionen sind gekapselte Programmblöcke, die mehrfach aufgerufen werden können. Jeder Aufruf erzeugt einen eigenen lokalen Parameter / Variablenblock.

Sind durch **()** zu erkennen, z.B. function().

Definition und Aufruf einer Funktion siehe Abschnitt 4.5.

4.2.5 Tabellen

Lua-Tabellen sind Hashtabellen, die beliebige Indizes (Schlüssel, keys) auf Werte abbilden. Dabei können sowohl die Indizes als auch die Werte beliebige Lua-Objekte wie Zahlen, Strings oder z.B. Tabellen sein – Typ **table.**

Tabellen werden mit der Zuweisung von geschweiften Klammern definiert. Sie können beliebige Datentypen speichern, z.B. auch Tabellen.

Definition:

```
meinTable = {}
```

Definitionen mit Wertzuweisungen:

```
meinTable = { 1, 4, "Willi", true, function()
              machwas end}
meinTable2 = { X= 255 , Y =10, geschwindigkeit =88}
```

Diese Werte können auch einzeln wie folgt zugewiesen, bzw. gelesen werden:

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

4.2.6 Garbage Collector

Zur Freigabe von reserviertem Speicherplatz gibt es einen Garbage Collector. Dieser gibt den physikalischen Speicherplatz von nicht mehr verwendeten Variablen (von Speicherbereichen auf die keine Referenz mehr besteht) frei. Diese Funktionalität kann auch im Script explizit aufgerufen werden. Die hierzu aufzurufende Funktion ist im Abschnitt „collectgarbage (opt)“ Seite 45 beschrieben.

4.3 Globale Systemvariable

4.3.1 _G

_G ist eine globale Variable (Tabelle). In ihr sind alle Basisfunktionen von Lua und alle globalen Variable mit ihren Speicherzeigern hinterlegt.

Soll für das Script eine andere Umgebung genutzt werden, so kann mit der Funktion „setfenv“ eine andere Umgebung eingestellt werden.

```
for k,v in pairs(_G) do print(k,v) end
```

4.3.2 _VERSION

Eine globale Variable, die die Versionskennzeichnung von Lua enthält, auf dem netSCRIPT aufbaut.

4.3.3 _NETSCRIPT_VERSION

Eine globale Variable, die die Version der netSCRIPT- Umgebung enthält.

4.3.4 __CYCLIC_FUNCTION

Nach dem Laden des Skripts wird der kompilierte Skriptcode an diese Variable gebunden. Bei jedem zyklischen Start wird diese Variable ausgewertet und der daran gebundene Code ausgeführt. Durch Zuweisen einer Lua-Funktion an diese Variable wird ab dem 2. zyklischen Start diese Funktion ausgeführt.

```
__CYCLIC_FUNCTION = Funktionsname
```

Hinweis: Wird der Variablen __CYCLIC_FUNCTION eine Funktion zugewiesen, wird diese Funktion im Debugger als Level 0 ausgeführt. Ohne diese Zuweisung wird unter dem Level 0 das Hauptprogramm aufgeführt.

4.4 Operationen

Folgende Operationen sind in netSCRIPT integriert: Die Auflistung zeigt die Operationen mit aufsteigender Präzedenz.

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

4.4.1 Mathematische-Operationen

Operator	Beschreibung	Beispiel	Beispiel
+	Dient zur Addition von Werten.	c = a + b	1 + 3 = 4
-	Dient zur Subtraktion von Werten.	c = a - b	5 - 3 = 2
*	Dient zur Multiplikation von Werten.	c = a * b	2 * 3 = 6
/	Dient zur Division von Werten.	c = a / b	8 / 2 = 4
^	Dient zur Exponentiation von Werten. (c = Potenz b zur Basis a)	c = a ^ b	2 ^ 3 = 8 = (2 * 2 * 2)
-	Dient zur Negation von Werten. Aus plus wird minus und umgekehrt.	c = - a	wenn a == 2, dann c == -2
%	Mathematische Funktion, die den Rest aus der Division zweier ganzer Zahlen angibt		7 mod 2 = 1 Denn 7:2 = 3, Rest 1

Tabelle 15: Übersicht mathematischer Operationen

Eine Besonderheit gibt es in netSCRIPT. Schreibt man 2+"6" dann interpretiert das Programm den 2. Wert nicht als Text, sondern als Zahl und gibt 8 aus. Es dürfen aber keine anderen Zeichen als Zahlen in den Anführungsstrichen stehen. (2+"6s" versteht der Interpreter von netSCRIPT nicht).

4.4.2 Logische Operationen

Es gibt drei Operanden:

and, or und not

Operator	Beschreibung
and	Wenn Bedingung 1 und Bedingung 2 erfüllt sein muss dann: Ergebnis = Bedingung1 and Bedingung2.
or	Wenn Bedingung 1 oder Bedingung 2 erfüllt sein muss dann: Ergebnis = Bedingung1 or Bedingung2.
not	Ist die Negierung des Zustandes.

Tabelle 16: Übersicht logische Operationen.

Beispiele für logische Operationen:

Variablendefinitionen:

```
SONNESCHEINT = true
DUNKEL = false
LICHTAN = true
```

Operator	Beispiel	Ergebnis
and	**sonnescheint and dunkel**	***false***
or	**sonnescheint or lichtan**	***true***
not	**NOT sonnescheint**	***false***

Tabelle 17: Beispiele logischer Operationen.

Logische Operationen auf Bit-Ebene siehe unter dem Abschnitt 6.1 Bit-Operationen.

4.4.3 Vergleichs-Operationen

Es können nur identische Zeichentypen verglichen werden!

Operator	Beschreibung	Beispiel	Ergebnis
==	Ist etwas gleich dem Andern? (nicht auf Tabelleninhalte anwendbar)	"Willi" == "willi"	false
~=	Ist etwas ungleich dem Anderen?	"Willi" ~= "willi"	true
<	Ist kleiner als das Andere?	2 < 3	true
>	Ist etwas größer als das Andere?	2 > 3	false
<=	Ist etwas kleiner oder gleich dem Anderen?	2 <= 3	true
>=	Ist etwas größer oder gleich dem Anderen?	2 >= 3	false

Tabelle 18: Übersicht: Vergleichs- Operationen

Beim Vergleich von ungleichen Typen, z.B. Zahl mit String, String mit Tabelle kommt es zu einen Fehler.

4.4.4 Schleifen

for, while, repeat.

4.4.4.1 for-Schleife

Beispiel:

```
1.  for var1 = anfang, ende, sprungweite(optional) do
       Block
    end
```

`Block` steht hier stellvertretend für einen Liste von Anweisungen. Zunächst wird der Variable **`var1`** der Wert **`anfang`** zugewiesen.

Bei jedem Schleifendurchlauf (zwischen **`do`** und **`end`**) wird die Variable **`var1`** um den Wert der Variable **`SPRUNGWEITE`** verändert. Dieses erfolgt so lange, wie **`var1`** <= **`ende`** ist. Mit dem Erreichen des Wertes **`ende`** wird die nächste Anweisung nach der Schleife **`end`** ausgeführt.

```
2.  for var1 in table do
       Block
    end
```

`Block` steht hier stellvertretend für einen Liste von Anweisungen. Die Schleife (zwischen **`do`** und **`end`**) wird solange durchlaufen, wie der Inhalt der Variable **`var1`** in der Tabelle (Variable **`table`**) enthalten ist. Ist dieses nicht der Fall, wird die Anweisung nach **`end`** ausgeführt.

4.4.4.2 while-Schleife

```
while BEDINGUNG do
  Block
end
```

`Block` steht hier stellvertretend für einen Liste von Anweisungen. Solange `BEDINGUNG` = *true* ist, wird der Anweisungsblock ausgeführt. Danach werden die Anweisungen nach dem **`end`** ausgeführt.

4.4.4.3 repeat-Schleife

```
repeat
  Block
until Bedingung
```

`Block` steht hier stellvertretend für einen Liste von Anweisungen. Solange `BEDINGUNG` = *true* ist, wird der Anweisungsblock ausgeführt. Wird die `BEDINGUNG` = false, werden die Anweisungen nach **`until`** ausgeführt.

Im Gegensatz zur for-Schleife, wird hier der `BLOCK` mindestens einmal durchlaufen.

4.4.4.4 BREAK

Das **break** Kommando ermöglicht das verlassen von Schleifen, unabhängig davon, ob die Eingansbedingung oder Endbedingung der Schleife erfüllt, oder nicht erfüllt ist.

Das Programm wird unmittelbar nach dem **end** bzw. **until** der Schleifefortgesetzt.

4.4.5 Verzweigungen if ...then

```
BlockA
if Bedingung1 then Block1
 [elseif Bedingung2 then Block2]
  [else Block3]
end
BlockB
```

Ist die BEDINGUNG1 „***true***“ (wahr), wird der Anweisungsblock Block1 (nach dem 1. **then**) ausgeführt.

Ist die BEDINGUNG1 ***„false”*** (nicht wahr), wird geprüft, ob die BEDINGUNG2 (nach dem **elseif**) zutrifft. Ist dieses der Fall, wird der Anweisungsblock Block2 ausgeführt. Die Zeile **{elseif** BEDINGUNG **then** Block**}** kann beliebig häufig zwischen **if** und **end** wiederholt werden.

Ist weder die BEDINGUNG1 noch die BEDINGUNG2 „***true***“ wird der Anweisungsblock block3 ausgeführt. Ist der Anweisungsblock block3 nicht vorhanden, wird mit dem BlockB nach dem **end** fortgefahren.

Danach wird die Verarbeitung mit dem blockB fortgesetzt. Unabhängig ob eines der Blöcke Block1, Block2, oder Block3 durchlaufen wurde.

4.5 Funktionen

Funktionen sind gekapselte Programmblöcke, die ihre eigene Variablenumgebung (Tabelle) haben, der bei jedem Aufruf neu angelegt wird. Die Kommunikation mit dem übrigen Programm erfolgt mittels Übergabeparameter.

Eine Funktion muss vor dem Aufruf definiert sein.

Die Variablenumgebung kann für eine Funktion über die Funktion „setfenv“ siehe Abschnitt 5.1.17 speziell gesetzt werden.

4.5.1 Definition einer Funktion

```
function name (var1, var2, var3)
  Block1
 [return var2, var3, var4]
end
```

Funktion	Definitionsbeginn einer Funktion.
name	Name der Funktion unter dem sie später aufgerufen wird.
(var1, var2, var3)	Übergabeparameterliste. Die Parameter werden durch Komma getrennt. Diese Parameter sind lokal definiert und sind im übrigen Programm nicht bekannt.
Block1	Verarbeitungsblock. Alle in diesem Block definierten mit local definierten Variablen sind lokale Variable, alle anderen global.
return var2, vari	**return** kehrt aus einer Funktion zurück. Hinter der **return**-Anweisung können einer oder mehrere mit Kommata getrennte Ausdrücke angegeben werden, deren Werte an die Anweisung, die die Funktion auf gerufen hat, zurückgegeben werden
end	Beendet die Definition der Funktion.

Tabelle 19: netSCRIPT, Funktions-Definition

4.5.2 Aufruf einer Funktion

Als Beispiel wird die Funktion aufgerufen, die im Abschnitt 4.5.1 definiert wurde.

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

varD	Die Variable **varD** übernimmt den Wert der lokalen Rückgabevariablen **var2** aus Abschnitt 4.5.1 .
varE	Die globale Variable **varD** übernimmt den Wert der lokalen Rückgabevariablen $\mathbf{var_3}$.
name	Name der Funktion unter dem sie definiert wurde.
(varA, varB, varC)	Zwischen den runden Klammern stehen beliebige Ausdrücke, deren Werte an die Funktion übergeben werden. Die Zuordnung ist dabei wie folgt: **expr_1 → var1** **varB → var2** **varC → var3**

Tabelle 20: netSCRIPT, Funktionsaufruf

Praktisches Beispiel einer Funktionsdefinition und deren Aufruf.

Code	Ergebnis: print (x).

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

Tabelle 21: netSCRIPT, Beispiel Funktions- Definition und Aufruf

5 Funktions-Bibliothek-Grundfunktionen

Alle Funktionen der OS-Library des Lua-Standards sind nicht nutzbar, da in dieser Umgebung nicht sinnvoll.

5.1 Basisfunktionen

Folgende Basis-Funktionen sind integriert:

5.1.1 assert (v [, message])

Bedingte Fehlermeldung.

Diese Funktion erzeugt einen Fehler, wenn der Parameter "**v**" den ‚Wert „nil" oder „false" hat. Andernfalls gibt die Funktion alle ihre Argumente zurück. „message" ist der Fehlertext. Wenn dieser nicht vorhanden ist, wird „assertion failed!" als Fehlermeldung ausgegeben.

5.1.2 collectgarbage (opt)

Diese Funktion stellt ein Interface zum Garbage Collektor dar. Sie erfüllt unterschiedliche Funktionen, abhängig vom Parameter (**opt**).

opt:	Funktion
"stop"	Stoppt den Garbage Collector
"restart"	Startet den Garbage Collector
"collect"	Führt einen kompletten Garbage Collector-Lauf durch.
"count"	Gibt den durch netSCRIPT belegten Speicherplatz in KBytes zurück.

Tabelle 22: netSCRIPT, Garbage Kollektor

Beispiel:

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

gibt im Debugger den momentan belegten Speicherplatz zurück.

5.1.3 error (message [, level])

Absolute Fehlermeldung, bedingte Fehlermeldung siehe Abschnitt 5.1.1

Bricht die Ausführung mit einer Fehlermeldung ab. Ausnahme: wenn der Aufruf sich in einer Funktion befindet, die mit pcall aufgerufen wurde. In diesem Fall wird die Ausführung hinter dem pcall fortgesetzt.

Beendet den letzten geschützten Funktionsaufruf (mit pcall, siehe Abschnitt 5.1.11) und übergibt an die Funktion „pcall" den Inhalt der Variablen „**message**" und „**level**".

Der Fehlermeldung wird eine Zeilennummer vorangestellt. Mit dem Level-Parameter wird bestimmt, welche Zeile das ist: bei 1 (Default) ist es die aktuelle Position im Skript. Wenn man sich in einer Funktion befindet und gibt Level 2 an, wird die Stelle ausgegeben, wo diese Funktion aufgerufen wird, usw., wenn man sich in einer Schachtelung von Funktionsaufrufen befindet, sind auch höhere Wertemöglich.

5.1.4 getfenv ([f])

Gibt die aktuelle Umgebungsvariable (Tabelle) zurück, in der die Variablen der Funktion „**f**“ abgelegt sind, im Normalfall ist dieses die Tabelle _G. Siehe hierzu auch setfenv() Abschnitt 5.1.17.

„**f**“ kann eine netSCRIPT Funktion oder die Nummer des Stack-Levels der Aufrufebene sein, die die Funktion repräsentiert. Ist „f“ keine Funktion oder „**f**“ ist 0, wird die Tabelle des Global-Variablenbereichs (_G) zurückgegeben. Der Defaultwert von „**f**“ ist 1.

5.1.5 getmetatable (object)

Diese Funktion gibt die Metatabelle des Ojektes zurück die im Parameter „**object**“ spezifiziert ist.

- Hat das „**objekt**“ keine Metatabelle, gibt die Funktion „**nil**“ zurück.
- Hat das „**objekt**“ eine Metatabelle, wird diese zurückgegeben.

5.1.6 ipairs (t)

Diese Funktion liefert eine Iterator- Funktion zurück, die z.B. in **for**-Schleifen verwendet werden kann.

Gibt bei jedem Aufruf ein Wertepaar einer Tabelle aller Einträge mit einem Integer-Schlüssel zurück, bis zum ersten Nicht-Integer-Schlüssel, bzw. fehlenden Eintrag in der Indexreihe.

Code	Ergebnis: 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])

Lädt einen Scriptcode mit der Funktion „**func**“ solange, bis die Funktion den Wert „**nil**“ zurück gibt. Dabei wird das jeweilige Ergebnis von “**func**” wiederum in einer Funktion gespeichert.

Jeder Aufruf von „**func**“ muss einen String zurückliefern, der mit dem zuvor ermittelten String verbunden wird. Die Rückgabe eines leeren Strings oder nil signalisiert das Ende des Blocks.

Werden bei der Compillierung des Blocks keine Fehler festgestellt, wird der Code in einer Funktion gespeichert, andernfalls wird „nil“ und eine Fehlermeldung zurückgegeben. Der „**blockname**“ wird für Fehlermeldungen und für Debug-Informationen verwendet.

Code	Ergebnis:

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

Im obigen Beispiel wird letztlich der Funktionsaufruf „print(42)“ ausgeführt.

5.1.8 loadstring (string [, blockname])

Mit dieser Funktion wird der Inhalt von „**string**“ zu einem Funktionsaufruf umgewandelt.

Um einen String zu laden und zu starten, ist folgende Syntax zu verwenden:

```
assert(loadstring(S))()
```

Um einen evt. Fehler später leichter identifizieren zu können, ist es sinnvoll einen „blockname“ mitzugeben, der in der Fehlermeldung dann mit ausgegeben wird.

Ist “blockname” nicht gegeben, wird der gegebene “string” verwendet.

Beispiel für einen Funktionsaufbau mit Parameterübergabe und anschließendem Aufruf der Funktion mit Wertübergabe.

Code	Ergebnis:.

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

5.1.9 next (table, index)

Gibt den Schlüssel und den Wert des nächsten Eintrags (um 1 größer als index) im Speicher für diese Tabelle zurück.

table	Tabellenname
index	Index der Tabelle, muss nicht numerisch sein.

Code	Ergebnis:.
`T = {100,200,300,400,500}` `print (next (T,3))`	`4  400`

5.1.10 pairs (t)

Liefert eine Iteratorfunktion für **for**, die nacheinander die Wertepaare einer Tabelle zurückliefert.

Code	Ergebnis: 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, par1, ···)

Ruft die Funktion "f" mit den Parametern **"par1,** ." auf.

Der Aufruf sollte wie folgt aussehen:

`OK, x1, x2, x3 = pcall (f, par1, par2, ....)`

Ist die Ausführung der Funktion "**f**" fehlerfrei, enthält der Rückgabeparameter **`OK`** den Wert „true" und die folgenden Parameter **`x1, x2, x3`** enthalten die Rückgabewerte der Funktion "**f**".

Tritt bei der Ausführung der Funktion "**f**" ein Fehler auf, enthält der Rückgabeparameter **`OK`** den Wert **false** und der Parameter **`x1`** die Informationen:

- Scriptname,
- Zeilennummer in der der Fehler auftrat,
- und einen Fehlertext.

5.1.12 print (par1, par2, ···)

Diese Funktion ist eine Lua- Standardfunktion, die wegen einer fehlenden Standardausgabe in einem netSCRIPT fähigen Gerät ins leere läuft.

Sie ist hier nur aufgeführt, damit die Beispiele der Basisfunktionen in einer Lua- Standardumgebung (auf dem PC) nachvollzogen werden können.

Gibt jede der aufgelisteten Parameter auf dem Standard-Ausgabekanal aus. Die Funktion ist nicht für eine formatierte Ausgabe geeignet.

Für formatierte Ausgaben ist die Funktion **string.format()** der Funktion **print()** vorzuschalten.

5.1.13 rawequal (par1, par2)

Prüft die Gleichheit zwischen **par1** und **par2** (ohne den Aufruf von Methafunktionen). Der Rückgabewert ist true oder false.

Der Vergleich erfolgt ohne den Aufruf von Metafunktionen.

5.1.14 rawget (table, index)

Liefert den Tabelleneintrag **`table[index]`**, ohne Metafunktionen auszuführen, zurück.

Im folgenden Beispiel wird ein Defaultwert für einen nicht vorhandenen Tabelleneintrag gesetzt, der bei einer normalen Tabellenabfrage zurückgegeben wird. Mit der Funktion „rawget“ wird der tatsächliche Eintrag zurückgegeben.

Code	Ergebnis:

`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)

Führt table[index]=value aus, ohne dabei Metafunktionen anzuwenden.

Im folgenden Beispiel wird der normale Tabelleneintrag über die Funktion „set“ abgewickelt, und der Lesezugriff auf die Tabelle „**t**“ über die Funktion „default“

Code	Ergebnis:

`T={}`
`function default(tab, key) –-[[Defaultwert wenn Index nicht vorhanden --]]`
`  return 42`
`end`
`function set(TABLE, KEY, VALUE) –[[neuer Eintrag in die Tabelle]]`
`  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("normaler Zugriff: ", T.A, T.B, T.C, T.D)`
`print("mit rawget: ", rawget(T, "A"), rawget(T, "B"), rawget(T, "C"), rawget(T, "D"))`
	`table: 01004A00  a`
	`table: 01004A00  c`
	
	`normaler Zugriff:`
	`  1  42  3  4`
	`mit rawget:`
	`  1  nil 3  4`

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

Ist **index** numerisch, wir der Parameter der Indexnummer und alle weiteren Parameter zurückgegeben.

Ist „**index**“ = „#“, wird die Anzahl der Einträge zurückgegeben.

Beispiel:

Code	Ergebnis:

`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)

Diese Funktion setzt für die Funktion „**f**“ den globalen Variablenbereich (von _G) auf die Tabelle „**table**“.

Statt der Funktion kann auch die Ebene angegeben werden in der die Funktion läuft.

„**f**“ = n;

n=1: → die aktuelle Ebene;

n= n+1 → die nächst höhere Ebene. Bei

n= 0 → wird die Umgebung der aktuell laufenden Ebene verändert.

Rückgabewert:

Die Funktion in der diese Funktion aufgerufen wird.

Hinweis: Diese Funktion darf nicht auf die Ebene 0 angewendet werden, da dadurch die Referenz auf alle Funktionen verloren gehen.

5.1.18 setmetatable (table, metatable)

Weist einer Tabelle „**table**“ eine Metatabelle „**metatable**” zu. Wenn „**metatable**” = „nil“ ist, wird eine vorhandene Metatabelle von der _Tabelle „**table**“ entfernt.

Mit dieser Funktion können neue Operationen für Tabellenwerte definiert werden, die sonst nicht erlaubt sind. Diese Funktionen müssen in der Metatabelle entsprechend definiert sein.

Folgende Funktionen können neu definiert werden:

Funktion	Anwendungsfall (wird aufgerufen für Standardaufruf)
__add	Addition (+)
__sub	Subtraktion (-)
__mul	Multiplikation (*)
__div	Division (/)
__pow	Potenzierung (e)
__unm	Wenn eine Tabelle mit dem unären Operator - versehen wird.
__concat	Wenn zwei Tabellen mit .. verknüpft werden.
__eq	Wenn zwei Tabellen mit == verglichen werden.
__it	Wenn zwei Tabellen mit < verglichen werden.
__le	Wenn zwei Tabelle mit <= verglichen werden
__index	Wenn ein Tabellenindex aufgerufen wird, der nicht existiert.
__newindex	Wenn eine Tabelle einen neuen Eintrag (Index) bekommt.
__call	Wenn irgendein Wert in der Tabelle angesprochen wird.
__tostring	Wenn die Funktion **tostring** auf eine Tabelle angewendet wird.

Tabelle 23: netSCRIPT, Funktionsersetzungen für die Funktion setmetatable

Weitergehende Informationen zum Thema Metatable sind unter Lua-Dokumentation unter und Dokumentation unter http://www.lua.org/manual/5.1/ Abschnitt 2.8 zu finden.

Folgendes Beispiel zeigt die Erzeugung von neuen Tabelleeinträgen und deren Addition.

Code	Ergebnis:

`-- Traegertabelle fuer die Vektorfunktionen`
`vector = {}`
 	
`-- Konstruktor: erzeugt ein neues Vektor-Objekt.`
`-- Verpackt die übergebenen Argumente in eine neue`
`-- Tabelle und setzt die Metatabelle dieser Tabelle.`
`-- Die Argumente müssen numerisch sein.`
`function vector.new(...)`
`  local V = {...}`
`  setmetatable(V, vector.mt)`
`  return V`
`end`
 	
`-- Erzeugt einen druckbaren String aus einem`
`-- Vektor.`
`-- Die table.concat-Funktion liefert einen mit Kommata`
`-- getrennten String mit den Inhalten des Vektors.`
`function vector.toString(self)`
`  return ("(" .. table.concat(self, ",") .. ")")`
`end`
 	
`-- Addiert zwei Vektoren und liefert das Ergebnis`
`-- in einem neuen Vektor zurück.`
`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`
 	
`-- Metatabelle`
`-- Der Eintrag __add bewirkt, dass der Operator + auf`
`-- zwei vector-Objekte angewandt die add-Funktion`
`-- aufruft.`
`-- Der Eintrag __index bewirkt, dass die definierten`
`-- Funktionen direkt auf einem vector-Objekt aufrufbar`
`-- sind: v:toString()`
 	
`vector.mt={__add=vector.add, __index=vector}`
`-- Erzeugt zwei Vektoren und addiert sie`
`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])

Wandelt die Variable „e“ in eine Zahl zur Basis „**base**“ um, wenn die Parameter Zahlen sind; oder Strings, die in eine Zahl umgewandelt werden können. Andernfalls wir „**nil**“ zurückgegeben.

Der optionale Parameter „**base**“ kann zwischen 2 und 36 liegen, dabei repräsentiert der Buchstabe „A“ (groß oder klein geschrieben) die 10 und der Buchstabe „Z“ die Zahl 35.

Der Defaultwert von „**base**“ ist = 10.

Beispiel:

Code	Ergebnis:
`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`

Hinweis: Der größte Zahlenwert von Hex-Zahlen ist 0xffffffff. Größere Werte wie z.B. 0x100000000 werden auf 0xffffffff „abgerundet“, ohne dass dabei ein Fehler ausgelöst wird.

5.1.20 tostring (e)

Wandelt jeden Datentyp in einen String um. Besser ist u.U. die Funktion “string.format” geeignet.

Hinweis: Wird diese Funktion z.B. auf Tabellen oder Funktionen angewendet, wird der Typ und die Adresse zurückgegeben.

5.1.21 type (v)

Gibt den Typ des Parameters “**v**“ zurück. mögliche Rückgabewerte sind:

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

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

Gibt die Elemente der Tabelle **list** mit allen numerischen Schlüsseln zurück. Mit **i** = von Schlüsselnummer bis **j** = bis Schlüsselnummer.

Beispiel:

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

Wenn **j** angegeben ist: wird der entsprechende Ausschnitt aus der Liste zurückgegeben. Ist **j** nicht angegeben, werden nur die Werte von Position **i** bis zum ersten Integer-Index, unter dem kein Wert eingetragen ist, zurückgeliefert. Im letzten Beispiel kommen also nur „a" und „b" zurück.

5.1.23 xpcall (f, err)

Diese Funktion ist ähnlich der Funktion „pcall", jedoch kann hier mit dem Parameter „err" ein spezieller ERROR-Händler aufgerufen werden.

„**xpcall**" ruft die Funktion „f" im geschütztem Mode auf, und benutzt „err" als Fehlerhändler, in dem auf die Fehlermeldung reagiert werden kann.

Returnwerte im Erfolgsfall:

1. „**true**"
2. Alle Rückgabeparameter der Funktion „**f**"

Returnwerte im Fehlerfall:

1. Im Fehlerfall "**false**"
2. Das Ergebnis von „**err**"

5.2 String Manipulation

Neben der Bibliothek „string“ gibt es noch für die Zusammenführung von Zeichenketten den Verkettungsoperator „**..**“.Damit können zwei Zeichenketten miteinander verknüpft werden.

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

Beim obigem Beispiel ist es zwingend, dass jede zu verkettende Variable zuvor definiert sein muss. Dieses ist insbesondere dann zu beachten, wenn an einem String etwas angefügt werden soll, z.B. bei der folgenden Anweisung: „ **`STR = STR..string.char(ICHAR)`** „

Die „string“ Bibliothek enthält weitere allgemeine Funktionen zur STRING-Manipulation. Alle String-Funktionen sind in der Tabelle „**string**“ enthalten.

Wird ein Index auf einen String gesetzt, ist zu beachten, dass das erste Zeichen an der Position eins steht

Die String-Library geht von einer ein Byte Zeichencodierung aus.

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

Gibt den numerischen Code der Zeichen aus dem String „s“ von der Position „i“ bis zur Position „j“ zurück.

Die Defaultwert für „**i**“ ist 1, der Defaultwert für „**j**“ ist „**i**“.

Beispiel:

Code	Ergebnis:
`STR = "abcdefghijklm"` `print(string.byte(STR,3,7))`	`99  100 101` `102 103`

5.2.2 string.char (...)

Erzeugt einen String aus einer Liste von Bytewerten (Bereich 0-255). Es ist die entgegen gesetzte Funktionalität von „string.byte“.

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

Sucht die erst Übereinstimmung im Sting “**s**” mit der Zeichnfolge “**muster**” und gibt den gefundenen Bereich mit der Nr. des Startzeichens und der Nummer des letzten Zeichens zurück. Wird keine Übereinstimmung gefunden wird „**nil**“ zurückgegeben.

init (numerisch) gibt die Startposition an, ab der gesucht werden soll.

Ist im Muster eine Zeichenkette enthalten, die auch als Zeichenklasse von Bedeutung ist, diese aber nicht als Zeichenklasse interpretiert werden soll, so ist der letzte Parameter „**plain**“ mit dem Wert „**true**“ zu belegen. Hierbei ist der Parameter „**init**“ allerdings erforderlich.

5.2.3.1 Muster

Muster sind eine Sequenz von Musterbausteinen. Im Allgemeinen ist ein Mustervergleich erfolgreich, wenn irgendein Teil des untersuchten Strings sich mit dem Muster deckt.

In Substrings sind die Zeichen des Musters mit Ankern eingeschlossen. Diese Anker werden durch das „**$**“ Zeichen oder „**^**“ repräsentiert. Außerhalb von Substrings repräsentieren die Zeichen „**$**“ und „**^**“ sich selbst.

Das Zeichen „**^**“ am Anfang des Musters bewirkt, dass der Vergleich nur gelingt, wenn das Muster den Anfang des durchsuchten Strings abdeckt. Das Gleiche gilt für „**$**“ und das Stringende. Beide kombiniert bedeuten, dass der gesamte String sich mit dem Muster decken muss

Bei String-Manipulationen geht es um Mustererkennungen, oder Musteraustausch. Deshalb gibt es zur Mustererkennung besondere Optionen.

5.2.3.2 Muster finden

In einem Muster kann ein Teilmuster mit Klammern () markiert werden. Ein Mustervergleich ergibt dann nicht nur den Teil eines Strings, der von dem Muster abgedeckt wird, sondern auch die Teile, die von den geklammerten Untermustern abgedeckt werden.

Die Teilmuster werden entsprechen der Position der öffnenden Klammer im Muster nummeriert. Zum Beispiel im Muster „(a*(.)%w(%s*))“: der Treffer der Zeichenkette „a*(.)%w(%s*)“ bekommt die Nummer 1, der Treffer der Zeichenkette „.“ Bekommt die Nummer 2 und der Treffer der Zeichenkette „%s*“ bekommt die Nummer 3.

5.2.3.3 Zeichenklassen

Eine Zeichenklasse repräsentiert einen bestimmte Zeichenart. Die folgenden Zeichenkombinationen ermöglichen die Beschreibung bestimmter zeichenklassen.

- **.:** (ein Punkt) repräsentiert alle Zeichen, außer ^$()%.[]*+-?, die eine spezielle Bedeutung haben, repräsentieren sich selbst.
- **%a:** repräsentiert alle Buchstaben.
- **%c:** repräsentiert alle Steuerzeichen.
- **%d:** repräsentiert alle Ziffern.
- **%l:** repräsentiert alle Kleinbuchstaben.
- **%p:** repräsentiert alle Interpunktions-Zeichen.
- **%s:** repräsentiert alle Arten Leerzeichen.
- **%u:** repräsentiert alle Großbuchstaben.
- **%w:** repräsentiert alle alphanumerischen Zeichen.
- **%x:** repräsentiert alle hexadezimal Ziffern.
- **%z:** repräsentiert das Zeichen mit dem Code 0.
- **%***: („*“ repräsentiert hier ein beliebiges Zeichen, welches kein Alpha-Numerischen-Zeichen ist.) Diese ist der Standardweg ein Zeichen, welches auch als Steuerzeichen in der Script-Sprache verwendet wird, als zu suchendes Zeichen eines Strings kenntlich zumachen. Dieses gilt auch für das % Zeichen selbst.
- **[*set*]:** repräsentiert eine Gruppe von Zeichen in „***set*** ”. Ein Bereich von Zeichen kann durch das Endezeichen „-„ spezifiziert werden. Alle Klassen „%*“ wie oben beschrieben können in „***set***“ benutzt werden. Alle anderen Zeichen in „***set***“ repräsentieren sich selbst.
  Beispiel:
  [*%w_*] oder **[*_%w*]** repräsentieren alle alphanumerischen Zeichen plus den Unterstrich; **[*0-7*]** repräsentieren alle Ziffern von 0 bis 7 und **[*0-7%l%-*]** repräsentiert alle Zahlen von 0 bis 7, die Kleinbuchstaben und das „-„ Zeichen.
  Beispiel:

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

- **[^*set*]:** repräsentiert das Komplement von ***set*** ,wobei ***set*** wie oben interpretiert wird.
- Für alle obigen Zeichenklassen gilt:
  Bei der Verwendung eines einzelnen Buchstabens (%a, %c usw.) bedeutet der korrespondiere Großbuchstabe das Komplement der Klasse. Z.B. **%W** repräsentiert alle NICHT alphanumerischen Zeichen.

5.2.3.4 Muster Positionen

Eine Zeichenposition kann sein:

- Eine Zeichenklasse, die genau ein Zeichen aus dieser Klasse abdeckt.
- Eine Zeichenklasse gefolgt von „*****“ deckt die längstmögliche Folge von 0, 1 oder mehreren Zeichen aus der Klasse ab.
- Eine Zeichenklasse gefolgt von + deckt die längstmögliche Folge von 1 oder mehreren Zeichen aus der Klasse ab..
- Beispiel:

Code	Ergebnis:

`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`

- Ein einzelnes Zeichen, gefolgt von einem '**-**', trifft zu, wenn kein oder mehrere Zeichen von dem Vorgängerzeichen vorhanden sind.

Code	Ergebnis:

`S1 = "bc1caaaad2aa bcade"`
`print(string.find(s1, "ca-",3))`
`print(string.find(s1, "a-",3))`
	
	`4 4`
	`3 2`

- Zeichenklasse gefolgt von „**?**“ deckt 0 oder 1 Zeichen aus der Klasse ab.

Code	Ergebnis:

`S1 = "bc1caaaad2aa bcade"`
`print(string.find(s1, "ca?"))`
`print(string.find(s1, "ca?",3))`
	
	`2 2`
	`4 5`

- %*n*, für *n* zwischen 1 und 9; entspricht einem Substring gleich dem n-ten gefundenen String (siehe unten).
- %b*xy*, wobei *x* und *y* zwei verschiedene Zeichen sind. Dieser Ausdruck führt zu Übereinstimmung mit Strings, die mit *x* anfangen und mit *y* aufhören sollen. Dieses bedeutet, dass, wenn man einen String von links nach rechts liest, +1 für ein *x* und -1 für ein *y* zählt. Der String gilt als gefunden, wenn der Zähler das erste Mal 0 wird.

5.2.4 string.format (formatstring, ···)

Diese Funktion gibt einen formatierten String einer variablen Anzahl von Argumenten, entsprechend der Umwandlungsvorgabe (Formatierung) für das Argument zurück. Die Formatierungsanweisung beginnt mit einem „%“ Zeichen.

Eine Umwandlungsvorgabe setzt sich wie folgt zusammen:

% [F] [W] [G] U

F, W und **G** können angegeben werden, U muss angegeben werden.

Dabei bedeuten:

F = Formatierungszeichen

W = Ausgabeweite „n“; n = Mindestzahl der auszugebenden Zeichen

G = Genauigkeit; „**.**“ oder „**.***” „**.n**” (n = Ganzzahl)

U = Umwandlungszeichen.

Formatierungszeichen: **„[F]“**

Formatierungszeichen	Bedeutung
„**-**„	Linksbündige Justierung.
„**+**“	Mit Ausgabe des Vorzeichens „+“ oder „-„
„ „ (Leerzeichen)	Falls das 1. Zeichen des Arguments kein Vorzeichen ist, wird ein Leerzeichen ausgegeben.
„**0**“	Bei numerischer Ausgabe wird mit Nullen bis zur angegebenen Weite aufgefüllt.
„**#**“	Die Auswirkung von „“**#**“ hängt vom Umwandlungszeichen ab: Bei „0“ bzw. „x, X“ wird der Wert mit vorangestelltem „0“ bzw. „0x“ ausgegeben. Bei „e, E , f“ wird der Wert mit Dezimalpunkt ausgegeben, auch wenn keine Nachkommastellen existieren. Bei „g, G“ wird der Wert mit Dezimalpunkt und Nachkommanullen ausgegeben.

Tabelle 24: netSCRIPT, Funktion string.format, Formatierungszeichen

Ausgabeweite: **„[W]“**

Ausgabeweite	Bedeutung
„**n**„	Mindestens **n** Stellen. werden ausgegeben. Ggf. werden die Stellen mit führenden Nullen aufgefüllt.

Niemals bewirkt eine nicht vorhandene oder zu kleine Weite, dass Zeichen nicht ausgegeben werden. Falls das Ergebnis einer Umwandlung mehr Zeichen enthält, als Weite vorgibt, dann werden trotzdem alle Zeichen ausgegeben.

Umwandlungszeichen: „U“

Umwandlungszeichen	Bedeutung
„**d**“, „**i**“	Als eine vorzeichenbehaftete ganze Dezimalzahl.
„**o**“	Als eine vorzeichenlose ganze Oktalzahl.
„**u**“	Als vorzeichnlose ganze Dezimalzahl. Achtung, bei negativen Zahlen können unerwartete Stringausgaben entstehen.
„**x**“, „**X**“	Als eine vorzeichenlose ganze Hexzahl. Bei „x“ → a, b, c, d, e, f Bei „X“ → A,B,C,D,E,F
„**f**“	Fließkommazahl, In Form von [-]ddd.ddddd
„**e**“, „**E**“	Als Expotentialzahl zur Basis 10. der Exponent enthält dabei mind. 2 Ziffern. Bei “e” [-]d.ddde±dd Bei “E” [-]d.dddE±dd
„**g**“, „**G**“	Wie bei “e” oder “E” wenn der Exponent <-4 ist, sonst im “f” Format.
„**c**“	Als Zeichen.
„**s**“	Als Zeichenkette.
„**%**“	Es wird das Zeichen „%“ ausgegeben und kein Argument ausgewertet.
„**q**“	Ermöglicht die Ausgabe von Steuerzeichen. Die Interpretation der Steuerzeichen ist vom angeschlossenem Gerät abhängig.

Tabelle 25: netSCRIPT, Funktion string.format, Umwandlungszeichen

Genauigkeitsangabe: „[G]“

Ausgabeweite	Bei Umwandlungszeichen	Bedeutung
„.**n**„ Wobei n eine Ganzzahl ist	d, i, o, u, x, X	Mindestzahl von auszugebenden Zeichen.
	e, E, f	Zahl der auszugebenden Nachkommastellen
	g, G	Maximale Anzahl von auszugebenden Zeichen.
	Sonstige	Undefiniertes Verhalten.
„.*****”		Das nächste Argument muss ganzzahlig sein. Ist der Wert dieses Arguments negativ, wird diese Genauigkeitsangabe ignoriert.

Tabelle 26: netSCRIPT, Funktion string.format, Genauigkeitsangaben

Steuerzeichen innerhalb des Strings

Steuerzeichen	Bedeutung auf einem Terminal
„**\a**“	Klingelton (auch mit \007 realisierbar).
„**\b**“	Backspace (ein Zeichen zurück positionieren).
„**\f**“	Seitenvorschub.
„**\n**“	Neue Zeile
„**\r**“	Wagenrücklauf (an den Anfang der momentanen Zeile positionieren).
„**\v**“	Vertikales Tabulatorzeichen.
„**\’**“	Hochkomma
„**\“**“	Anführungszeichen
„****“	Backslash

Tabelle 27: netSCRIPT, Funktion string.format, Steuerzeichen

Die Auswirkung obiger Steuerzeichen ist vom jeweils interpretierenden Gerät abhängig.

Beispiele für eine Formatierung:

Code	Ergebnis:
`A = 3.1415926535` `print(string.format("pi = %.4f", a))`	`pi = 3.1416`
`tag, TITLE = "h1", "Ein Titel"` `print(string.format("<%s>%s</%s>", tag, TITLE, tag))`	`<h1>Ein Titel</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', 'ein String "Test" und \n eine neue Zeile'))`	"ein String \"Test\" und \ eine neue Zeile"
`print(string.format('%q', 'ein String Test und \n eine \t neue Zeile'))`	"ein String Test und \ eine neue Zeile"

5.2.5 string.gmatch (s, muster)

Identisch mit dem Funktionsaufruf string.gmatch()

Diese Funktion liefert eine Funktion zurück, die in **„s“** nach **„muster“** sucht und gibt dieses dann zurück. Die zurückgegebene Funktion merkt sich, bis wo sie schon gesucht hat und gibt beim nächsten Aufruf auch das nächste Vorkommen zurück. Bis der String abgearbeitet ist.

Rückgabe:

Die Such-Funktion.

Beispiel:

"%a" sucht nach Buchstaben.

"%a+" sucht nach Buchstaben, solange welche folgen.

1. Eine einfache Anwendung:

Code	Ergebnis:

`S = "hello wor1ld from Lua"`
`--[[man beachte die 1 in`
`S = "hello wor1ld from Lua"]]`
`MyFunc = string.gfind(S, "%a+")`
`print( MyFunc() )`
`print( MyFunc() )`
`print( MyFunc() )`
`print( MyFunc() )`
`print( MyFunc() )`
`print( MyFunc() )  --[[Die Funktion liefert nichts mehr!]]`
	`Hello`
	`Wor`
	`ld`
	`from`
	`Lua`

2. Anwendung mit der generischen **for**-Schleife:

Code	Ergebnis:

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

3. Anwendung mit der generischen for-Schleife mit einer Tabelle:

Code	Ergebnis:

`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`

Hier wird nach einem Wort gesucht, gefolgt von einem Gleichheitszeichen dann gefolgt von einem Wort. Wort 1 kommt in die erste Variable k Wort 2 kommt dann in v

Dann ist myTable wie folgt besetzt:

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

bzw.

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

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

Gibt eine Kopie von "**s**" zurück, in der alle, oder die ersten "**n**" Treffer des "**muster**" durch "**repl**" ersetzt sind. „**repl**" kann ein String, eine Tabelle oder eine Funktion sein. Als zweiten Returnwert gibt „**gsub**" die Anzahl der in „**s**" gefundenen „**muster**" zurück.

Wenn „**repl**" ein String ist, dann wird dieser als Austausch-Wert für das gefundene Muster. Das „**%**"-Zeichen dient als Ausgabezeichen: jede Zeichenfolge in „**repl**" der Form „**%n**" mit „n" zwischen 1 und 9, steht für das n-ten gefundene Muster (siehe unten). Die Zeichenfolge „**%0**" steht für den gesamten gefundene „muster". Die Zeichenfolge „**%%**" steht für ein einzelnes „**%**" Zeichen.

Code	Ergebnis:
`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`

Wenn "**repl**" eine Tabelle ist, wird aus der Tabelle der Wert in „**s**" ersetzt, welcher unter dem Schlüssel der Tabelle gefunden wird, der zur entsprechenden Übereinstimmung von „**muster**" gehört.

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

Wenn "**repl**" eine Funktion ist, wird diese so oft aufgerufen, wie es Übereinstimmungen von „**muster**" in „**s**" gibt.

Code	Ergebnis:
`print(string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s) return loadstring(s)() end))`	`4+5 = 9 1`
Zur Erläuterung des obigen Aufrufs, hier eine Untergliederung mit ihren Ergebnissen:	
`print(string.find ("4+5 = $return 4+5$", "%$(.-)%$"))`	`7 18 return 4+5`
`s = "return 4+5" print(loadstring(s)())`	`9`

Das “$” Zeichen in obigem Beispiel wird als Begrenzungszeichen genutzt.

Ist der Rückgabewert durch eine Tabellenabfrage oder einer Funktion entstanden, der eine String oder eine Ziffer ist, wird der Rückgabewert als Ersetzungsstring verwendet. Ist der Rückgabewert **false** oder **nil** dann erfolgt keinen Ersetzung.

5.2.7 string.len (s)

Gibt die Länge (Anzahl Zeichen) des String „**s**“ zurück. Das Zeichen “****” in dem String wird nicht als Zeichen mitgezählt. Wird der „****“ von einer Anzahl Nullen gefolgt („**\000**“) wird nur die erste Null nach dem „****“ mitgezählt.

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

5.2.8 string-lower (s)

Gibt einen String zurück, in dem alle Großbuchstaben vom String “**s**” in Kleinbuchstaben umgewandelt sind.

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

Ermittelt die erste Übereinstimmung vom “**muster**” mit dem String „**s**“. Wird eine Übereinstimmung gefunden, wird der Teil von „**s**“ zurückgegeben, der auf das Muster passt, andernfalls der Wert „**nil**“.

Mit dem Parameter „**init**“ kann angegeben werden ab welcher Zeichenposition im String „s“ gesucht werden soll.

5.2.10 string.rep(s, n)

Diese Funktion vervielfältigt den String „**s“** mit der Anzahl **„n“** mal.

Code	Ergebnis:
`rueckGabe = string.rep ("Lua ", 5)` `print( rueckGabe )`	`Lua Lua Lua Lua Lua`

5.2.11 string.reverse (s)

Gibt einen String zurück, der die Zeichen von “**s**” in umgekehrter Reihenfolge enthält.

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

Gibt einen String zurück, der mit dem **i**-ten Zeichen des Strings **s** beginnt und mit den **j**-ten Zeichen endet. Ist **j** nicht gegeben, wird **j** auf 1 gesetzt. Ist **j** negativ, wird die bis- Position vom Stringende gezählt.

5.2.13 string.upper (s)

Gibt einen String zurück, in dem alle Kleinbuchstaben des String “s” in Großbuchstaben gewandelt sind. Alle anderen Zeichen bleiben unverändert.

5.3 Tabellenmanipulationen

In dieser Bibliothek sind alle Basisfunktionalitäten der Tabellenbearbeitung enthalten.

Hinweis: Das Ergebnis der Funktionen concat, insert, remove und sort ist nicht definiert, wenn die Tabelle „Löcher“ in ihren Indizes hat, wie {„a“, „b“, „c“, [5]=“d“}

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

Macht aus einer Tabelle deren Elemente/Einträge nur aus Ziffern und Strings besteht, einen Gesamt-String. Der Trenner „**sep**“ (default „nichts“) gibt an, wie die einzelnen Variable getrennt werden sollen. Mit „**i**“ (default = 1) kann bestimmt werden, ab welcher Tabellenzeile die Ausgabe beginnen sollt. Mit „**j**“ (default = Tabellenlänge) kann bestimmt werden, bis zu welcher Tabellenzeile die Ausgabe erfolgen soll. Ist **i > j** wird ein leerer String zurückgegeben.

Code	Ergebnis:
`t1 = {"a", "b", "c", "d", "e"}` `print(table.concat (t1, "; ",2,4))`	`b; c; d`
`print(table.concat({"a", "b", "c",` `[5]="d", ["x"]="y"})`	`abc`

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

Fügt den Wert „**value**“ in die Tabelle „**table**“ an der Position „**pos**“ ein. Der Defaultwert von „**pos**“ ist n+1. Ist „**pos**“ > n+1, wird nichts eingefügt.

Code	Ergebnis:

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

Wenn die Tabelle Integerindizes mit „Löchern“ enthält, ist die Einfügeposition nicht definiert.

5.3.3 table.maxn (table)

Gibt den größten numerischen Index der Tabelle „**table**“, oder 0 zurück, wenn es keinen numerischen Index in der Tabelle gibt.

5.3.4 table.remove (table [, pos])

Entfernt einen Eintrag aus der Tabelle **table** von der Position **pos**. Alle folgenden Elemente der Tabelle rücken eine Zeile nach, um die Lücke zu schließen. Der default für **pos** ist n, so dass beim Weglassen des Parameters **pos** der letzte Eintrag aus der Tabelle entfernt wird. Das gilt nur, wenn die Tabellenindizes eine fortlaufende Folge 1, …, n sind. Allgemein wird ein Eintrag an Position n entfernt, für den gilt

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

5.3.5 table.sort (table [, comp])

Sortiert die Elemente der Tabelle **table** in aufsteigender Reihenfolge (table[n] < table[n+1] nach ASCII-Code von table[1] bis table[n].

Wird eine andere Sortierung gewünscht, kann man mit dem Parameter **comp** eine Funktion angeben, welche 2 Tabellenzeilen zurückliefert und **true** zurückgibt, wenn die Sortierung der Tabellenzeilen der gewünschten Sortierung entspricht.

Ohne eine Angabe einer Funktion „**comp**“ darf die Tabelle entweder nur Zahlen, oder nur Strings enthalten.

5.4 Mathematische Funktionen

Diese Bibliothek hat eine Verknüpfung zur Standard-C-Mathematik-Bibliothek. Alle Funktionen sind in der Tabelle „**math**“ enthalten.

5.4.1 math.abs (x)

Diese Funktion gibt den Absolutwert des Parameters “**x**” zurück.

5.4.2 math.acos (x)

Diese Funktion gibt den arccosin von dem Parameter „**x**“ im Bogenmaß zurück.

5.4.3 math.asin (x)

Diese Funktion gibt den arcsin von dem Parameter „**x**“ im Bogenmaß zurück.

5.4.4 math.atan (x)

Diese Funktion gibt den arctan von dem Parameter „**x**“ im Bogenmaß zurück.

5.4.5 math.atan2 (y, x)

Diese Funktion gibt den arctan von „**y“/“x“** im Bogenmaß zurück.

5.4.6 math.ceil (x)

Diese Funktion gibt die kleinste Ganzzahl, die **>= „x“** ist zurück.

5.4.7 math.cos (x)

Diese mathematische Funktion gibt den Kosinus von “**x**” zurück..

5.4.8 math.cosh (x)

Diese Funktion gibt den Kosinus Hyperbolicus von “**x**” zurück.

5.4.9 math.deg (x)

Diese Funktion gibt den Winkel in Grad zurück von “**x**” zurück, wenn x im Bogenmaß angegeben ist.

5.4.10 math.exp (x)

Diese Funktion gibt den Wert $\mathbf{e}^{\mathbf{x}}$ (e^x) zurück

5.4.11 math.floor (x)

Diese mathematische Funktion gibt den größten Integerwert < oder = „**x**“ zurück.

Achtung! Bei einer Vorberechnung von “x“ kann es zu Rundungsproblemen und damit zu Ungenauigkeiten kommen.

5.4.12 math.fmod (x, y)

Diese Funktion gibt den Divisionsrest von “**x**” ***/*** “**y**” zurück. Z.B. math.fmod(13, 5) == 3

5.4.13 math.frexp (x)

Diese Funktion gibt die Werte **m** und **e** folgender Gleichung zurück: $x = m*2^e$. Dabei ist **e** ein integer Wert und **m** der Bereich von 0,5 ... <1, (oder m = 0 wenn x = 0 ist).

5.4.14 math.huge

math.huge ist eine Konstante und repräsentiert den Wert pos. Unendlich.

5.4.15 math.ldexp (m, e)

Diese Funktion gibt den Wert $m*2^e$ zurück, wobei e ein integer Wert sein muss.

5.4.16 math.log (x)

Diese mathematische Funktion gibt den natürlichen log. von “**x**” zurück.

5.4.17 math.log10 (x)

Diese Funktion gibt den Logarithmus von “**x**” zur Basis 10 zurück.

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

Diese Funktion gibt den größten Wert aus der Variablenliste x_1 ... x_n zurück.

5.4.19 math.modf (x)

Gibt den ganzzahligen Anteil und den Rest von „x“ zurück

Code	Ergebnis:

`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

Ist die Konstante π (3.1415926535898)

5.4.21 math.pow (x, y)

Diese Funktion gibt den Wert $\mathbf{x}^{\mathbf{y}}$ zurück. Kann auch wie folgt berechnet werden: **x^y**.

5.4.22 math.rad (x)

Diese Funktion gibt den Winkel "**x**" in grad im Bogenmaß zurück.

5.4.23 math.sin (x)

Diese Funktion gibt den Sinus von "**x**" im Bogemaß zurück.

5.4.24 math.sinh (x)

Diese Funktion gibt den Sinus Hyperbolicus von "**x**" zurück.

5.4.25 math.sqrt (x)

Gibt die Quadratwurzel von "**x**" zurück. Dieses kann auch mit dem Ausdruck „**x^0.5**" erreicht werden.

5.4.26 math.tan (x)

Gibt den Tangens von "**x**" im Bogenmaß zurück.

5.4.27 math.tanh (x)

Gibt den Tangens Hyperbolicus von "**x**" zurück.

6 Spezielle netSCRIPT Funktionen

Zusätzliche Bibliotheken für den netTAP sind: „**bit**“ für Bitoperationen und „**util**“ für spezielle Utilitys. Die Schnittstellen werden über Instanzen angesprochen.

6.1 Bit-Operationen

Diese Operationen stehen in der Bibliothek „**bit**“ und beginnen mit einem b.

Es können hiermit vorzeichenlose 32 Bit integer Variable verknüpft werden.

Es können Zahlen in Hex-Notation in der Größe 0 - FFFFFFFF eingegeben werden. Werden den Variablen größere Werte zugewiesen, werden diese ohne Fehlermeldung auf 0xFFFFFFFF begrenzt.

6.1.1 bit.band

bit.band(a,b)	
Bitweise UND-Verknüpfung der Integer Zahlen a und b im Bereich 0 - 0xFFFFFFFF.	
Argument a numerisch	1. Vorzeichenlose 32-Bit Zahl.
Argument b numerisch	2. Vorzeichenlose 32-Bit Zahl.
Rückgabewert 1 numerisch	Vorzeichenlose 32-Bit Zahl mit bit-weiser UND-Verknüpfung.
Status-/Fehlernummer	Lua Fehler, z.B. Zahl erwartet

Status-/Fehlernummer werden in der Variablen „lasterror“ gespeichert.

6.1.2 bit.bor

bit.bor(a,b)	
Bitweise ODER-Verknüpfung der vorzeichenlosen Integer-Variablen a und b im Bereich 0 - 0xFFFFFFFF.	
Argument a numerisch	1. Vorzeichenlose 32-Bit Zahl.
Argument b numerisch	2. Vorzeichenlose 32-Bit Zahl.
Rückgabewert numerisch	Vorzeichenlose 32-Bit Zahl mit bit-weiser ODER-Verknüpfung.
Status-/Fehlernummer	Lua Fehler, z.B. Zahl erwartet

Status-/Fehlernummer werden in der Variablen „lasterror“ gespeichert.

6.1.3 bit.bxor

bit.bxor(a,b)	
Bitweise XODER-Verknüpfung (Exclusive-ODER-Verknüpfung) der vorzeichenlosen Integer-Variablen a und b im Bereich 0 - 0xFFFFFFFF.	
Argument a numerisch	1. Vorzeichenlose 32-Bit Zahl.
Argument b numerisch	2. Vorzeichenlose 32-Bit Zahl.
Rückgabewert numerisch	Vorzeichenlose 32-Bit Zahl mit bit-weiser XODER -Verknüpfung.
Status-/Fehlernummer	Lua Fehler, z.B. Zahl erwartet

Status-/Fehlernummer werden in der Variablen „lasterror“ gespeichert.

6.1.4 bit.bnot

bit.bnot(a)	
Bitweise Invertierung des Strings a	
Argument a numerisch	Vorzeichenlose 32-Bit Zahl.
Rückgabewert numerisch	Vorzeichenlose 32-Bit Zahl, die bitweise zu „**a**“ Invertiert ist
Status-/Fehlernummer	Lua Fehler, z.B. Zahl erwartet

Status-/Fehlernummer werden in der Variablen „lasterror“ gespeichert.

6.1.5 bit.lshift

bit.lshift(a, n)	
Bits nach links verschieben.	
Argument a numerisch	Vorzeichenlose 32-Bit Zahl.
Argument n numerisch	Die Shiftweite (0-31).
Rückgabewert numerisch	Vorzeichenlose 32-Bit Zahl, die Bitweise um „**n**“ Bits nach links verschoben ist. Die rechts frei werdenden Bits werden mit 0 aufgefüllt.
Status-/Fehlernummer	Lua Fehler, z.B. Zahl erwartet

Status-/Fehlernummer werden in der Variablen „lasterror“ gespeichert.

6.1.6 bit.rshift

bit.rshift(a, n)	
Bits nach rechts verschieben.	
Argument a numerisch	Vorzeichenlose 32-Bit Zahl.
Argument n numerisch	Die Shiftweite (0-31)..
Rückgabewert numerisch	Vorzeichenlose 32-Bit Zahl, die Bitweise um „**n**“ Bits nach recht verschoben ist. Die links frei werdenden Bits werden mit 0 aufgefüllt.
Status-/Fehlernummer	Lua Fehler, z.B. Zahl erwartet

Status-/Fehlernummer werden in der Variablen „lasterror“ gespeichert.

6.2 Zahlenumwandlungen

Folgende Zahlenumwandlungen können vorgenommen werden.

- Von einer Lua Zahl in ein Binärformat, gespeichert in einer String-Variable.
- Von einem Binärformat (gespeichert in einem String) in eine Lua Zahl.

Hinweis zu Wertebereich/Genauigkeit:

Lua arbeitet intern mit Double-Fließkommazahlen mit 52 Bit Mantisse und 11 Bit Exponent. Hiermit können alle Vorzeichen behafteten und vorzeichenlose int-Werte von 8-32 Bit ohne Rundungsverluste dargestellt werden, 64-Bit-Werte jedoch nicht

6.2.1 util.NumToBin

util.numToBin(Zahl, Type[,Endian])	
Wandelt die Zahl in eine Binär-Repräsentation um, z.B. **numToBin(0x12345678, util.UINT32, util.LITTLE_ENDIAN)** ergibt einen String mit den Bytes 0x78 0x56 0x34 0x12.	
Argument 1 **Zahl**	Zu wandelnde Zahl im beliebigen Lua-Format.
Argument 2 **Typ**	Der Datentyp in dem eine Umwandlung hinein erfolgen soll. Folgende Typen sind möglich: UINT8, UINT16, UINT32, UINT64; INT8, INT16, INT32, INT64; FLOAT, DOUBLE. Dem Datentyp ist immer der Präfix "**util.**" voranzustellen.
Argument 3 **Endian**	util.LITTLE_ENDIAN: Im Rückgabestring steht das Low-byte zuerst. util.BIG_ENDIAN: Im Rückgabestring steht das High Byte zuerst. Dieses Argument ist optional. Defaultwert ist util.LITTLE_ENDIAN.
Rückgabewert **String**	String mit Binär-Repräsentation.
	Nil, bei Fehler in der Umwandlung.
Status-/Fehlernummer	**err.UTIL_INVALID_PARAMETER** (0x C0800401) Unbekannter Wert bei Zieltyp oder ENDIAN.
	err.UTIL_OUT_OF_RANGE (0x40800402) Zahl liegt nicht im Wertebereich des Zieltyps.

Status-/Fehlernummer werden in der Variablen „lasterror" gespeichert.

6.2.2 util.BinToNum

util.binToNum(Str, Type[,Endian])	
Wandelt die Binär-Repräsentation einer Zahl in eine Lua-Zahl **binToNum(str, util.UINT32, uti.LITTLE_ENDIAN**), wobei str ein String mit den Bytes 0x78 0x56 0x34 0x12 ist, gibt die Zahl 0x12345678 zurück.	
Argument 1 **Str**	String mit der Binärzeichenfolge.
Argument 2 **Typ**	Der Datentyp aus dem eine Umwandlung heraus erfolgen soll. Folgende Typen sind möglich: UINT8, UINT16, UINT32, UINT64; INT8, INT16, INT32, INT64; FLOAT, DOUBLE. Bei jedem Datentyp ist immer der Präfix "**util.**" voranzustellen.
Argument 3 **Endian**	util.LITTLE_ENDIAN: in „str" wird das Low-byte als erstes Byte erwartet. util.BIG_ENDIAN: in „str" wird das High Byte als erstes Byte erwartet. Dieses Argument ist optional. Defaultwert ist util.LITTLE_ENDIAN.
Rückgabewert	Zahl im Lua-Format.
String	Nil, bei Fehler in der Umwandlung.
Status-/Fehlernummer	**err.UTIL_INVALID_PARAMETER** (0x C0800401) Unbekannter Wert bei Zieltyp oder **ENDIAN**.
	err.UTIL_WRONG_SIZE (0x40800403) String hat die falsche Länge für den angegebenen Typ.

Status-/Fehlernummer werden in der Variablen „lasterror" gespeichert.

6.3 LED –Ansteuerung

Es ist möglich die Duo COM-LED-Statusanzeige im sekundären netTAP Netzwerkmodul zu steuern. Siehe auch Benutzerhandbuch netTAP NT100 Abschnitt: Serielle Kommunikation mit netSCRIPT

Wenn das netSCRIPT-Programm nicht läuft, zum Beispiel beim Debuggen oder kein netSCRIPT-Programm geladen ist, übernimmt das Gerät die Steuerung der LED. In diesem Fall blinkt sie zyklisch rot.

6.3.1 util.SetLed

util.SetLed(Name, [Zustand])		
Schaltet die Duo COM-LED ein oder aus.		
Argument	**Name**	**„run“** es wird die grüne LED angesteuert.
		„error“ es wird die rote LED angesteuert.
	Zustand	**true** **macht die** LED an. Wenn kein Zustand angegeben wurde ist dies der Standardwert
		false **macht die LED aus.**
Rückgabewert		kein
Status-/Fehlernummer		**err.UTIL_INVALID_PARAMETER** (0x C0800401) Unbekannter Wert bei Zieltyp oder **ENDIAN**.

6.4 Abfrage der Zykluszeit des Skriptes

Für einige Anwendungen kann es bedeutend sein, welche Zykluszeit dem Skript vom SYCON vorgegeben wurde. Die Zykluszeit kann mit folgender Funktion abgefragt werden:

6.4.1 util.GetCycleTime

util.GetCycleTime()	
Gibt die konfigurierte Zykluszeit des Skripts in Millisekunden zurück. Die Zeit wird im Konfigurationswerkzeug SYCON.net eingestellt und mit dem Skriptprogramm an das Zielgerät übertragen.	
Argumente	keine
Rückgabewert	Skriptzykluszeit in Millisekunden
Status-/Fehlernummer	kein

6.5 Checksummenfunktionen CRC

Die Checksummenfunktion wurde in Anlehnung an das Rocksoft CRC-Modell realisiert. Nähere Informationen zum Rocksoft CRC-Model sind unter http://www.ross.net/crc/download/crc_v3.txt zu finden.

Um Checksummen zu berechnen, muss zunächst ein Objekt erzeugt werden. Mit diesem Objekt wird dann mit weiteren Funktionsaufrufen die CRC-Berechnung aus den übergebenen Daten ausführt. Zugriffe auf das Checksummen-Objekt werden akkumuliert. Daher ist es möglich einen Checksumme aus einzelnen Teilstrings aufzubauen. Zum Rücksetzen des Checksummenobjektes gibt es die Funktion „:HashReset().

6.5.1 Erzeugung des Prüfsummenobjektes „HashCreate“

util.HashCreate (Typkonstante, [Parameter])	
Erzeugt ein Hash-Objekt, macht einmalige Vorberechnungen (bei CRC wird eine Tabelle vorberechnet).	
Argumente: Typkonstante	Hiermit wird festgelegt nach welchem Verfahren die Checksumme berechnet wird. Mögliche Werte sieheTabelle 28.
Parameter	sind nur bei der Typkonstante „util.HASHTYPE_CRC“ möglich. Mögliche Werte siehe Tabelle 29
Rückgabewert	Hash-Objekt: Objekt mit dem die Prüfsummenberechnungen durchgeführt werden können.
Status-/Fehlernummer	**err.UTIL_INVALID_PARAMETER** (C0800401) unbekannter Checksummentyp bzw. ungültige Parameter

6.5.1.1 Prüfsummenberechnungsvarianten

Prüfsumme	**Typkonstante in Funktion in util.HashCreate**	**Parameter**
CRC	util.HASHTYPE_CRC	s.u.
Byteweises XOR	util.HASHTYPE_XOR	-
Byteweise Summe mod 256	util.HASHTYPE_SUM	-

Tabelle 28: netSCRIPT, Prüfsummenberechnungsvarianten

6.5.1.2 Parameter für die Prüfsummenberechnungsart CRC

Für eine CRC Prüfsumme wird das Prüfsummenobjekt wie folgt erzeugt:

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

Dabei haben die Argumente folgende Bedeutung:

Argument	Bedeutung
Width	Grad des Polynoms, d.h. der größte Exponent im Polynom (8..32)
Poly	Polynom als Zahlenwert, d.h. die Koeffizienten ohne die führende 1 Für das Polynom $x^{16} + x^{15} + x^2 + 1$ ist Width = 16 und Poly = binär 1000000000000101 = 0x8005; Für das Polynom $x^{15} + x^2 + 1$ ist Width = 15 und Poly = binär 101 = 0x5
Init	Initialwert des CRC-Registers (0.. 2^width – 1)
RefIn	Gibt die Reihenfolge an, mit der die Bits der Datenbytes verarbeitet werden. **false**: höchstwertigstes Bit zuerst **true**: niederwertigstes Bit zuerst
RefOut	Wenn **true**, wird der Ausgabewert der CRC-Berechnung gespiegelt z.B. 10101111 → 11110101
XorOut	Der Ausgabewert der CRC-Berechnung wird zuletzt (ggf. nach bitweiser Spiegelung (RefOut = true)) mit diesem Wert xor-verknüpft (0.. 2^width – 1)

Tabelle 29: netSCRIPT, CRC Parameter

6.5.2 Funktionen zur Berechnung der Prüfsummen

Das erzeugte Prüfsummenobjekt stellt folgende Funktionen zur Verfügung:

6.5.2.1 Datenübergabe an das Prüfsummenobjekt

:Hash (string)	
„Addiert“ die Berechnung der Daten von „**string**“ zur Prüfsumme.	
Argumente: **string**	wird byteweise auf die Prüfsumme des benannten Objektes aufaddiert.
Rückgabewert	keinen
Status-/Fehlernummer	keine

Wird obige Funktion mehr als einmal aufgerufen (ohne rückgesetzt worden zu sein) wird die Prüfsumme über alle übergebenen „Strings“ zusammen gebildet.

6.5.2.2 Checksummenabfrage

:HashResult()	
Liefert die aktuelle Prüfsumme als numerischen Wert des benannten Objektes.	
Argumente:	keine
Rückgabewert	Aktuelle Prüfsumme des Objektes
Status-/Fehlernummer	keine

6.5.2.3 Rücksetzen des Prüfsummenobjektes

Vor der erneuten Berechnung der Checksumme für ein oder mehrere Datenpackete, muss das Prüfsummenobjekt in den Ausgangszustand (Initwert des Create-Objektes) zurückgesetzt werden.

:HashReset()	
Setzt das Hash-Objekt in den Startzustand zurück.	
Argumente:	keine
Rückgabewert	keine
Status-/Fehlernummer	keine

6.5.2.4 Skriptbeispiel für den Einsatz der CRC-Funktionen:

Einmaliges Erzeugen des Hash-Objekts „**h**“:

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

Berechnen eines CRC-Wertes:

```
h:Hash(„Halli“)
```

Auslesen des CRC-Wertes als Zwischensumme:

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

Erneuter Aufruf der Berechnung:

```
h:Hash(„_Hallo“)
```

Im Checksummenobjekt steht jetzt der Wert, als wenn der Übergabestring bei ersten Aufruf

```
„Halli_Hallo“
```

gewesen wäre.

Ist die Berechnung beendet, ist das Checksummenobjekt zurückzusetzen.

```
h:HashReset()
```

Hiernach beginnt die nächste Berechnung erneut mit dem Initialwert.

7 Serielle Kommunikation

Die Kommunikation zur seriellen UART Schnittstelle läuft unabhängig von der Kommunikation zur übergeordneten Netzwerk und der Steuerung (auch SPS oder Host genannt).

Die gesamte Sende- und Empfangsverarbeitung der seriellen Schnittstelle wird automatisch durch netSCRIPT-Funktionen gesteuert. Lediglich die Ebene 1 und 2 des OSI-Models werden außerhalb und unabhängig von netSCRIPT verwaltet. Daher ist zum Beispiel die Start- und Stopbit-Handhabung sowie die Parity-Bits nicht Bestandteil der Daten die innerhalb des netSCRIPT-Programms behandelt werden müssen. Für die Funktionsweise des seriellen UARTs sind sie lediglich einmal beim Initialisieren des seriellen Ports zu konfigurieren.

Sonstige Protokollanforderungen sind mit netSCRIPT in den Programmteilen für Sende- und Empfangsbetrieb selbst zu realisieren, somit auch zum Beispiel CRC und Checksummen-Verarbeitungen.

Die Funktionen die netSCRIPT im Blockmodus bereitstellt, erlauben eine serielle Kommunikation über den UART im Halb-Duplex Verfahren.

Die Funktionen, die netSCRIPT im Zeichenmodus bereitstellt erlauben eine serielle Kommunikation über den UART im Voll-Duplex Verfahren. Bei RS485 ist auch im Zeichenmodus nur Halb-Duplex möglich.

Bei den RS422 und RS485 Schnittstellen, mit mehreren aktiven Teilnehmern, ist im Skript zu prüfen, ob die Datenleitung für einen Sendeauftrag frei ist.

7.1 Konfigurationsparameter der seriellen Datenübertragung

Zur Einstellung der seriellen Kommunikationsparameter sind folgende Parameter definiert:

Parameter	Tabellenschlüssel	Bereich	Standard
Übertragungsgeschwindigkeit	**baudrate**	6**...**1000000	115200
Anzahl der Datenbits	**databits**	1**...**8	8
Parität	**paritymode**	Siehe unten	None
Anzahl der Stoppbits	**stopbits**	1**...**65535	1
Schieberichtung	**shiftdirection**	Siehe unten	LSB first
Schnittstellentyp	**interfacetype**	Siehe unten	RS232 RtsCts
Übergabepolarität	**handshakepolarity**	Siehe unten	0
Ruhepegel	**busIdlesymbol**	0/1	1
Daten invertieren	**invertdatamode**	true/false	false
Übertragungsmodus	**charmode**	true/false Siehe unten	false

Für den Blockmodus gibt es folgende weitere Parameter:

Parameter	Tabellenschlüssel	Bereich	Standard
Bestätigungsverzugszeit: Beim Empfang maximale Wartezeit bis das erste Zeichen empfangen wird, in 10ns Schritten	**ackdelaytime**	0**...**2^32-1	0 entspricht ∞
Zeichenverzugszeit: Beim Empfang maximale Zeit zwischen 2 Zeichen bis das Ende der Zeichenfolge erkannt wird. In 10 ns Schritten (n * 10 ns)	**chardelaytime**	0**...**2^32-1	0 entspricht ∞
Endeerkennungs-Zeichenfolge	**endpattern**	Siehe unten	empty
Endeerkennungsmaske	**endmask**	Siehe unten	empty
Anzahl Nachlauf-Bytes	**traillen**	0**...**255	0

Tabelle 30: netSCRIPT, Parameter der seriellen Schnittstelle

Die Parameter werden in einer Tabelle gespeichert. Sie sind über den in der obigen Tabelle angegebenen „Tabellenschlüssel“ in netSCRIPT direkt ansprech- und veränderbar.

Der Standardwert wird wirksam, wenn keine Tabelle definiert ist, oder kein Eintrag in der Tabelle unter diesem Schlüssel vorhanden ist.

Die Parameter der obigen Tabelle, die unter „Bereich“ den Eintrag: „Siehe unten“ haben, können folgende vordefinierten Werte annehmen:

paritymode:

port.PARITY_EVEN	0 bei gerader Anzahl Einsen im Übertragungsstring, andernfalls 1.
port.PARITY_ODD	1 bei gerader Anzahl Einsen im Übertragungsstring, andernfalls 0
port.PARITY_NONE	(Es wird kein Parity-Bit übertragen)
port.PARITY_MARK	(Es wird ein Parity-Bit mit dem festen Wert 1 übertragen.)
port.PARITY_SPACE	(Es wird ein Parity-Bit mit dem festen Wert 0 übertragen.)

shiftdirection:

port.SHIFT_DIRECTION_LSB_FIRST:
(Das niederwertigste Bit wird zuerst übertragen.)

port.SHIFT_DIRECTION_MSB_FIRST:
Das höchtswertigste Bit wird zuerst übertragen.)

interfacetype:

port.INTERFACE_TYPE_RS232 (ohne HW Handshake)

port.INTERFACE_TYPE_RS232_RTS_CTS (mit Hardware Handshake)

port.INTERFACE_TYPE_RS422

port.INTERFACE_TYPE_RS485

handshakepolarity:

Dieser Parameter bestimmt beim Schnittstellentyp RS232 mit Hardware-Handshake den aktiven Zustand der RTS-Leitung. Bei allen anderen Schnittstellen ist dieser Parameter ohne Bedeutung.

charmode

true: Die Datenübertragung erfolgt zeichenweise.

false: Die Datenübertragung erfolgt im Blockmodus.

endpattern:

Ist ein String von max. 8 Bytes, und definiert die Zeichenfolge, an der das Telegrammende erkannt wird.

endmask:

Ist ein String von max. 8 Bytes. Ist dieser Parameter angegeben, definiert er die Bits, die zur Erkennung des Telegrammendes aus **endpattern** zu verwenden sind. Dabei gilt:

0 → Bit wird nicht zum Vergleich herangezogen,

1 → Bit wird zum Vergleich verwendet.

7.1.1 Funktionen zur Initialisierung der Seriellen-Schnittstelle

7.1.1.1 PortReadConfigDb

PortReadConfigDb()	
Liest die vom SYCON.net gespeicherte Konfigurationstabelle der UART-Schnittstelle (siehe Abschnitt 3.1.1) und überführt sie in eine lokale Tabelle des Skripts, um sie dort ggf. verändern zu können.	
Argumente:	Keine
Rückgabewert	**Tabelle**, wenn die vom SYCON.net im netTAP-Gerät abgelegte Konfigurationsdatei vorhanden ist und interpretiert werden kann, wird eine Tabelle zurückgegeben.
	nil, wenn keine Tabelle erzeugt werden konnte.
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_NO_UARTDB** (0x C0800214) Es existiert keine hinterlegte Konfigurationstabelle
	err.PORT_PARSING_UARTDB (0x C0800215) Die hinterlegte Kofnigurationstabelle konnte nicht gelesen werden

7.1.1.2 PortOpen

PortOpen([port number],[configuration table])	
Initialisiert und konfiguriert die serielle Schnittstelle	
Argument 1: **port number**	Optional, Standardwert = 2, mögliche Werte: 0...3. netTAP100 unterstützt nur Port 2
Argument „: **configuration table**	Optional, wenn nicht vorhanden, werden die Defaultparameter verwendet. Die Einträge der Tabelle überschreiben die Defaultparameter.
Rückgabewert	**Portinstanz** wenn erfolgreich
	nil, wenn der Port nicht geöffnet werden konnte.
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_NO_SUCH_PORT** (0x 40800211) Die übergebene Portnummer existiert nicht oder wird nicht unterstützt.
	err.PORT_ALREADY_OPEN (0x 40800212) Der Port ist bereits geöffnet.
	err.PORT_INVALID_CONFIG (0xC0800201) Die übergebene Konfigurationstabelle enthält einen ungültigen Wert.
	err.PORT_XC_INIT_FAILED (0x C0800206) Die serielle Schnittstelle konnte nicht initialisiert werden.

7.1.1.3 :PortClose

:PortClose ()	
Verwirft alle noch nicht gesendeten Daten und schließt den Port. Unmittelbar vor dem Doppelpunkt ist als Präfix die Instanz, die bei der Funktion „PortOpen“ zurückgegeben wurde, einzufügen.	
Argumente:	keine
Rückgabewerte:	keine
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_NOT_OPEN** (0x C0800213) Der Port der benutzte Portinstanz ist nicht geöffnet gewesen.
	err.PORT_XC_INIT_FAILED (0x C0800206) Der Port konnte nicht geschlossen werden.

7.1.2 Beispiele zur Parametereinstellung

Die aufgeführten Funktionen müssen in der angegebenen Reihenfolge ausgeführt werden. Die einzelnen Funktionsaufrufe sind optional. Der letzte Funktionsaufruf ist zwingend erforderlich.

7.1.2.1 Funktionsaufruf mit Einlesen der SYCON.net-Einstellungen

```
conf = PortReadConfigDb() --[[Einlesen der SYCON.net
                              Konfigurationstabelle]]--
conf.baudrate = 9600 -- nur baudrate abändern
xuart = PortOpen(conf) -- öffnen der Schnittstelle mit
                          der definierten Parameterta-
                          belle “conf” –
```

Siehe auch Beispiel-Skript auf der netTAP DVD unter:
„Examples\netSCRIPT\Serial Port Blockmode\blkmode.lua

7.1.2.2 Funktionsaufruf ohne Einlesen der SYCON.net-Einstellungen

In diesem Fall sind die im Abschnitt 0 beschriebenen Standard-Parameter gültig. Dann genügt der Funktionsaufruf:

```
xuart = PortOpen()
```

Soll hier zum Beispiel nur die Baudrate gegenüber den Standardparametern verändert werden, kann folgende Anweisungssequenz verwendet werden:

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

Siehe auch Beispiel-Skript auf der netTAP DVD unter:
„Examples\netSCRIPT\Serial Port Blockmode\blkmode.lua

7.1.2.3 Schließen des Ports

Der im Beispiel unter 7.1.2.1 und 7.1.2.2 zur Kommunikation geöffnete Port wird wie folgt geschlossen:

```
xuart:PortClose()
```

8 Serielle Kommunikation im Blockmodus

Mehrere Sende- oder Empfangsaufträge können aus netSCRIPT heraus gleichzeitig an den UART übergeben werden. Für die Übergabe der seriellen Daten an und von dem UART reserviert sich netSCRIPT einen Speicherbereich von insgesamt 16 KBytes. Dieser Bereich wird den einzelnen Aufträgen je nach ihrer zu übertragenen Nutzdatenanzahl dynamisch zugeteilt. Dabei kann jeweils ein Sende- oder Empfangsauftrag mit maximal 1024 Bytes Nutzdaten übertragen werden. Jedem Auftrag werden 64 Bytes Verwaltungsdaten vorangestellt. Im Grenzfall ergibt dies 16 KBytes / 1088 Bytes = 15 Aufträge mit jeweils 1024 Bytes Nutzdaten gleichzeitig. Sind zum Beispiel nur 64 Bytes Nutzdaten zu übertragen, lassen sich 16 KBytes / 128 Bytes = 128 Aufträge aktivieren.

Sende- und Empfangsanforderungen werden über das netSCRIPT-Programm als sequentielle Aufträge im seriellen UART aktiviert. Es ist möglich mehrere Aufträge gleichzeitig an den UART zu übergeben. Pro Auftrag reserviert netSCRIPT dynamisch bei jedem Aufruf einen ausreichend großen Übertragungspuffer der für das Senden und Empfangen bereitgestellt wird. Die Anzahl der möglichen gleichzeitig aktivierten Aufträge richtet sich nach der Nutzdatengröße der Einzelaufträge. Die Kenngrößen für die Berechnung der Anzahl finden Sie in der Einleitung dieses Abschnittes.

Der serielle UART besitzt zwei Warteschlangen (FIFOs). In die sogenannte Request FIFO werden die einzelnen Sende- und Empfangsaufträge nach dem FIFO-Prinzip (**F**irst **I**n **F**irst **O**ut) eingetragen. netSCRIPT stellt hierfür entsprechende Funktionen bereit. Der UART arbeitet nacheinander die Aufträge ab und quittiert für netSCRIPT lesbar die Aufträge in der sogenannten Confirmation FIFO. Ein Zugriff auf diese Warteschlange ermöglicht dem netSCRIPT Anwender den Erfolg oder Misserfolg der Aufträge zu erkennen. Dabei wird die Quittung des Auftrages aus der Confirmation FIFO entfernt und der Speicherbereich dieses Blocks steht als freier Speicher wieder für nächste Aufträge zur Verfügung. Es entsteht ein Kreislauf der Auftragsblöcke. Aufträge können sich nicht gegenseitig überholen, das FIFO Prinzip wird immer eingehalten.

Werden zwei oder mehrere Sendeaufträge unmittelbar hintereinander im UART aktiviert, so werden die Daten lückenlos, sofern die Baudrate nicht zu hoch ist und die Aufträge sehr schnell abgearbeitet werden, auf der seriellen Schnittstelle gesendet. So ist es möglich Datenströme > 1024Byte Länge zu erzeugen.

Ist der Ablauf des netSCRIPT Programms schneller als die Aufträge z.B. aufgrund einer niedrigen Baudrate vom UART bearbeitet werden können, entsteht ein Engpass in den freien Auftragsblöcken. Steht kein freier Auftragsblock mehr zur Verfügung, wird ein Sende- oder auch Empfangsauftrag schließlich abgelehnt.

Den einzelnen Aufträgen kann eine Identifikationsnummer übergeben werden, um sie nach Abarbeitung eindeutig zu identifizieren. Sende- und Empfangsaufträge mit und ohne Identifikationsnummer können beliebig kombiniert werden.

8.1 Sendeaufträge ohne Identifikationsnummer

Bei Verwendung von Sendeaufträgen ohne Identifikationsnummer müssen die erledigten Aufträge nicht aus der Confirmation FIFO entfernen werden. netSCRIPT übernimmt diese Aufgabe für diese Art Aufträge immer automatisch, indem es beim Aufruf einer jeden Funktion der seriellen Schnittstelle auf erledigte Aufträge ohne ID prüft und sie entfernt. Es lassen sich so Sendeaufträge in beliebiger Anzahl hintereinander aktivieren ohne eine weitere netSCRIPT-Funktion verwenden zu müssen. Das Skriptprogramm vereinfacht sich so. Es lässt sich aber nicht der Erfolg oder Misserfolg der Aufträge prüfen.

Der Grundsätzliche Ablauf ist in der nachfolgenden Abbildung 10 dargestellt, und wird in der folgenden Tabelle beschrieben.

Zeitpunkt	Beschreibung
T1	Der erste Auftrag Bl 1 ist im UART zur Versendung in der Bearbeitung. 5 Weiter Aufträge warten in der Request FIFO Warteschlange auf die Abarbeitung im UART.
T2	Der erste Auftrag Bl 1 ist vom UART verarbeitet. Er wird in die Confirmation FIFO Warteschlange weitergeleitet.
T3	Der erste Empfangsauftrag ist in der Verarbeitung, die zwei versendeten Aufträge befinden sich noch in der Confirmation FIFO Warteschlange. Der Empfangsblock bleibt so lange in der UART-Bearbeitung, bis die definierte Anzahl Zeichen eingegangen ist.
T4	Der Empfangsauftrag Bl 4 wird bei ankommenden Daten am UART mit Daten gefüllt, bis die erwartete Datenmenge erreicht ist. Erfolgt nun bei T4 eine Abfrage im netSCRIPT-Programm auf empfangene Daten, werden automatisch alle Sendeaufträge ohne ID aus der Confirmation FIFO Warteschlange entfernt (symbolisch sind die Aufträge durchgekreuzt). Die Empfangsdaten werden ausgelesen und dieser Block ebenfalls freigegeben.
T5	Der Sendeauftrag Bl 5 befindet sich in der UART-Verarbeitung. Die Empfangsdaten des Auftrages Bl 4 können von netSCRIPT ausgelesen werden.

Tabelle 31: netSCRIPT, Zeitpunkte UART Abarbeitung 1

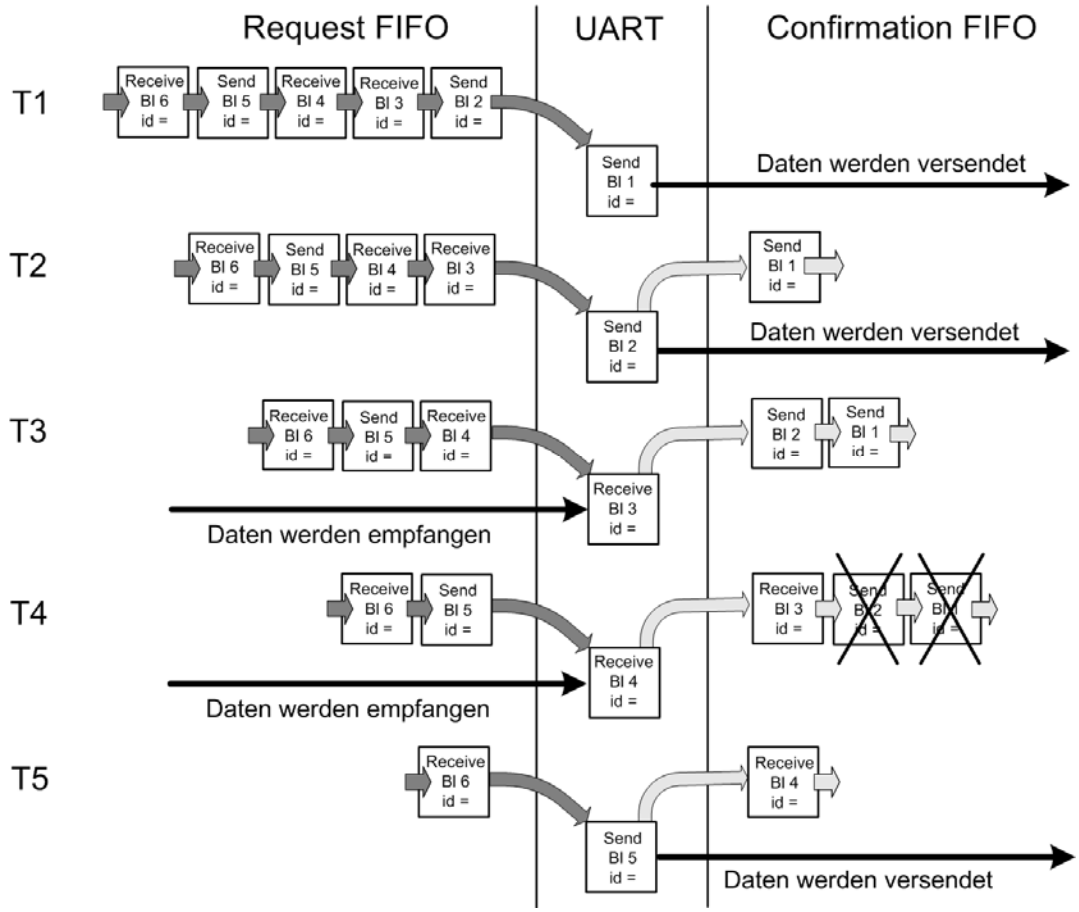


Abbildung 10: Sende- Empfangsverarbeitung ohne Auftrags-ID

8.2 Sendeaufträge mit Identifikationsnummer

Ziel von Aufträgen mit Identifikationsnummer ist es sie innerhalb der Auftragsabwicklung eindeutig zuordnen zu können und den Status ihrer Übermittlung zu erfahren. Das hat direkten Einfluss auf den Programmablauf in netSCRIPT. Bei Aufträgen mit ID müssen erledigte Aufträge per netSCRIPT-Funktion explizit aus der Warteschlange der Confirmation FIFO entfernt werden. Es dürfen mehrere Aufträge mit ID gleichzeitig beauftragt werden.

Der Grundsätzliche Ablauf ist im nachfolgenden Bild dargestellt, und wird in der folgenden Tabelle beschrieben.

Zeitpunkt	Beschreibung
T1...T3	Der Ablauf bis zum Zeitpunkt T3 entspricht dem Ablauf wie die Autragsabwicklung ohne ID.
T4	Sollen die Daten des ersten Empfangauftrages (Bl 3) gelesen werden, müssen mit 2 Funktionsaufrufen zuvor die Sendeaufträge aus der Confirmation FIFO gelesen/gelöscht werden, bevor der Empfang des Auftrages Bl 3 (id = 13) verarbeitet werden kann.
T5	Mit dem Lesen der Aufträge Bl 1, 2 und 3 sind diese aus der Confirmation FIFO entfernt worden. Damit kann der Auftrag Bl 4 (id = 14), der bereits zum Zeitpunkt T4 mit Daten gefüllt wurde, bearbeitet werden.

Tabelle 32: netSCRIPT, Zeitpunkte UART Abarbeitung 2

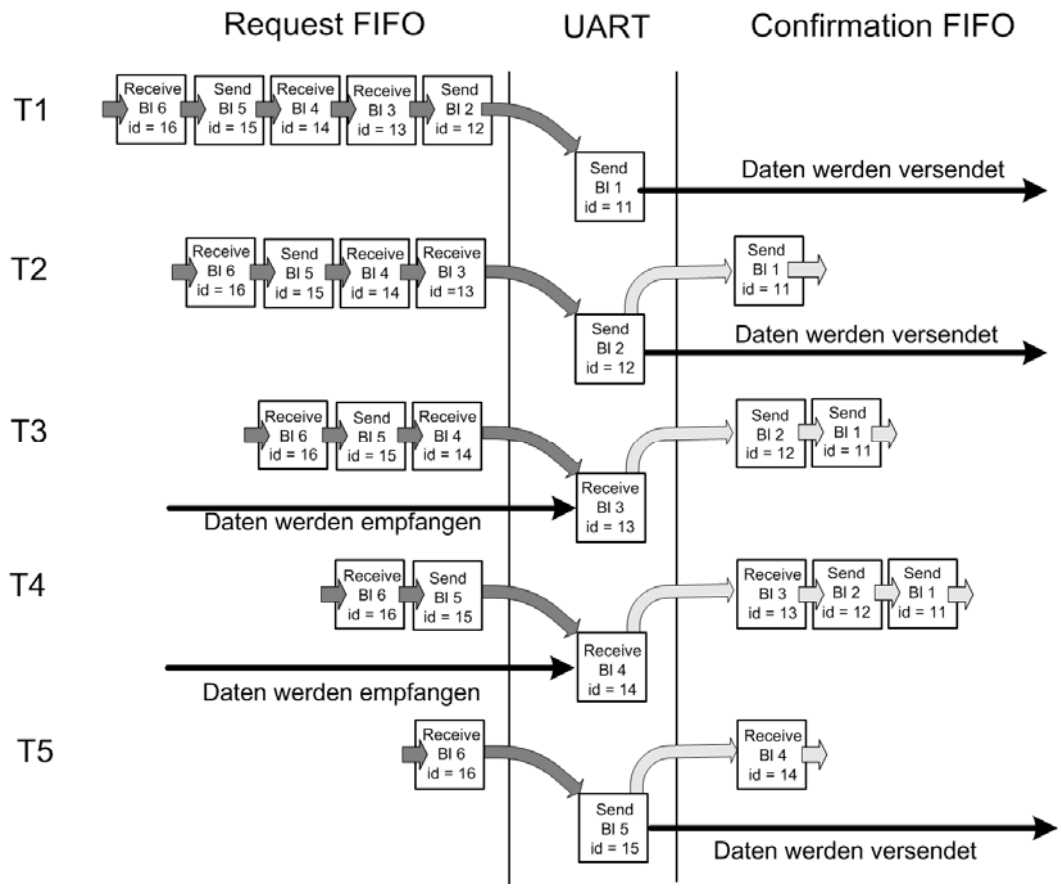


Abbildung 11: Sende- Empfangsverarbeitung mit Auftrags-ID

8.3 Sende- / Empfangsfunktionen für den Blockmodus

Bevor die in diesem Kapitel beschriebenen Funktionen verwendet werden können, ist mit der Funktion „**PortOpen**“ der UART zu initialisieren. Dabei wird eine Instanz erzeugt, die den Funktionen voranzustellen ist:

Beispiel:

```
myuart = PortOpen()
```

Mit dem obigem Aufruf ist die Portinstanz „myuart“ angelegt worden, die den nachfolgenden Portfunktionen vorangestellt werden muss.

Beispiel für die Übergabe der Instanz an die Sendefunktion:

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

8.3.1 :PortSend

:PortSend(string, [id])	
Erzeugt einen Sendeauftrag in der Request FIFO Queue und übergibt die zu sendenden Daten.	
Argumente: **string**	Enthält die zu sendenden Daten. Datenlänge ≤ 1024 Byte.
id	Optional, Auftrags-Identifikationsnummer im Bereich 0...2^32-1.
Rückgabewerte:	**true**, wenn der Sendeauftrag in die Request FIFO Queue eingestellt werden konnte.
	false, wenn der Sendeauftrag nicht erzeugt werden konnte.
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_STRING_TOO_LONG** (0x 40800218) Der übergebene String überschreitet die Anzahl von 1024Bytes.
	err.PORT_FIFO_FULL (0x 40800205) Die Request FIFO ist überfüllt, versuchen Sie laufende Aufträge abzuwarten und den Aufruf später noch einmal abzusetzen.
	err.PORT_NO_BUFFER (0x 40800210) Es konnte kein Sendepuffer in ausreichender Größe bereitgestellt werden. Erledigte Aufträge müssen erst abgeholt werden, um wieder Puffer bereit zu stellen.
	err.PORT_NOT_OPEN (0x C0800213) Der angegebene Port ist nicht geöffnet.
	err.PORT_WRONG_MODE (0xC0800203) Der Port wurde im Zeichen-Modus geöffnet und kann nicht im Blockmodus beauftragt werden.

8.3.2 :PortReceive

:PortReceive(len, [fMatch, [fTimeout]], [id])	
Erzeugt einen Empfangsauftrag in der Request FIFO Queue. Die Parameter **len, fMatch** und **fTimeout** werden an den UART zur Empfangs-Endeerkennung übergeben. Der UART beendet den Datenempfang wenn eine dieser drei Bedingungen zutrifft. Bei einem Empfangsauftrag muss im Gegensatz zu einem reinen Sendeauftrag ohne ID in jedem Fall die Abarbeitung des Auftrages mit der Funktion **:PortIsReceiveDone()** erfolgen, unabhängig davon ob der **:PortReceive()**-Funktion eine **id** mitgegeben wurde oder nicht.	
Argumente: **len**	Datenanzahl in Bytes die Empfangen werden sollen. Mögliche Werte: 1...1024, Ist die Zahl erreicht beendet der UART den Empfangsbetrieb.
fMatch	**true**, es werden die Parameter **endpattern** und **endmask** zur Empfangsendekennung verwendet. Dabei muss die Empfangsdatenlänge ≤ dem Argument **len** sein.
	false, Es wird auf keine Endekennung getestet (default).
fTimeout	**true**, (default) es werden die UART-Parameter **ackdelaytime** und **chardelaytime** zur Überwachung vom UART verwendet.
	false, die UART-Parameter **ackdelaytime** und **chardelaytime** werden nicht zur Empfangsüberwachung verwendet.
id	Frei vorgebbare Auftrags-Identifikationsnummer 0...2^32-1.
Rückgabewerte:	**true**, wenn der Auftrag in die Request FIFO Queue eingestellt werden konnte.
	false, wenn der Auftrag nicht erzeugt werden konnte.
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_INVALID_PARAMETER** (0x 40800216) Ungültiger Parameter wurde übergeben, überprüfen Sie den erlaubten Wertebereich.
	err.PORT_FIFO_FULL (0x 40800205) Die Request FIFO ist überfüllt, versuchen Sie laufende Aufträge abzuwarten und den Aufruf später noch einmal abzusetzen.
	err.PORT_NO_BUFFER (0x 40800210) Es konnte kein Empfangspuffer in ausreichender Größe bereitgestellt werden. Erledigte Aufträge müssen erst abgeholt werden, um wieder Puffer bereit zu stellen.
	err.PORT_NOT_OPEN (0x C0800213) Der angegebene Port ist nicht geöffnet.
	err.PORT_WRONG_MODE (0xC0800203) Der Port wurde im Zeichen-Modus geöffnet und kann nicht im Blockmodus beauftragt werden.

8.3.3 :PortExchange

:PortExchange(string, len, [fMatch, [fTimeout]], [id])	
Erzeugt einen Sende-Empfangsauftrag in der Request FIFO Queue. Übergibt zunächst die sendenden Daten (Argument **string**). Nach der Versendung durch den UART geht er in Empfangsbetrieb und wartet auf ein Empfangszeichen unmittelbar. Die Parameter **len, fMatch** und **fTimeout** werden an den UART zur Empfangs-Endeerkennung übergeben. Der UART beendet das Füllen dieses Datenblocks wenn eines dieser drei Argumente zutrifft. Bei einem Send-Empfangsauftrag muss im Gegensatz zu einem reinen Sendeauftrag ohne ID in jedem Fall die Abarbeitung des Auftrages mit der Funktion **:PortIsExchangeDone()** erfolgen, unabhängig davon ob der **:PortExchange ()**-Funktion eine **id** mitgegeben wurde oder nicht.	
Argumente: **string**	Enthält die zu sendenden Daten. Datenlänge ≤ 1024 Byte
len	Datenanzahl in Bytes die Empfangen werden sollen. Mögliche Werte: 1...1024. Ist die Zahl erreicht beendet der UART den Empfangsbetrieb.
fMatch	**true**, es werden die Parameter **endpattern** und **endmask** zur Empfangsenderkennung verwendet. Dabei muss die reelle Empfangsdatenlänge der zu erwartenden Zeichenfolge ≤ dem Argument **len** sein.
	false, Es wird auf keine Endekennung geprüft.
fTimeout	**true**, (default) es werden die UART-Parameter **ackdelaytime** und **chardelaytime** zur Überwachung vom UART verwendet.
	false, die UART-Parameter **ackdelaytime** und **chardelaytime** werden nicht zur Empfangsüberwachung verwendet.
id	Frei vorgebbare Auftrags-Identifikationsnummer 0...2^32-1.
Rückgabewerte:	**true**, wenn der Auftrag in die Request Queue eingestellt werden konnte.
	false, wenn der Auftrag nicht erzeugt werden konnte.
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_STRING_TOO_LONG** (0x 40800218) Der übergebene String überschreitet die Anzahl von 1024Bytes.
	err.PORT_INVALID_PARAMETER (0x 40800216) Ungültiger Parameter wurde übergeben, überprüfen Sie den erlaubten Wertebereich.
	err.PORT_FIFO_FULL (0x40800205) Die Request FIFO ist überfüllt, versuchen Sie laufende Aufträge abzuwarten und den Aufruf später noch einmal abzusetzen.
	err.PORT_NO_BUFFER (0x 40800210) Es konnte kein Send- oder Empfangspuffer in ausreichender Größe bereitgestellt werden. Erledigte Aufträge müssen erst abgeholt werden, um wieder Puffer bereit zu stellen.
	err.PORT_NOT_OPEN (0x C0800213) Der angegebene Port ist nicht geöffnet.
	err.PORT_WRONG_MODE (0xC0800203) Der Port wurde im Zeichen-Modus geöffnet und kann nicht im Blockmodus beauftragt werden.

8.3.4 :PortIsSendDone

:PortIsSendDone()	
Prüft auf einen abgearbeiteten Sendeauftrag in der Confirmation FIFO Warteschlange. Wenn vorhanden wird dieser entfernt und dessen **id** als Rückgabewert übergeben. Es wird immer einer und der zeitlich älteste Sendeauftrag entfernt. Diese Funktion muss nur aufgerufen werden, wenn bei der Aktivierung des Sendeauftrages dem Auftrag eine **id** mitgegeben wurde. Andernfalls werden Sendeaufträge ohne ID automatisch von netSCRIPT entfernt. Bitte beachten Sie, dass die Funktion immer nur den zeitlich ältesten Auftrag in der Confirmation FIFO entfernt. Liegt dabei kein erledigter Sendeauftrag vor, kann die Funktion nicht erfolgreich durchgeführt werden.	
Argumente:	Keine
Rückgabewerte:	**nil**, wenn kein Sendeauftrag abgeschlossen oder statt eines erledigten Sendeauftrages ein anderer erledigter Auftrag in der Confirmation FIFO zu Abholung vorliegt. Dieser ist zunächst aus der Confirmation FIFO mit seiner entsprechenden Funktion aus der Warteschlange zu entfernen.
	id des freigegebenen Sendeblocks.
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_FIFO_EMPTY** (0x40800204) Die Confirmation FIFO enthält gar keine abgearbeiteten Aufträge
	err.PORT_NO_CONFIRMATION (0x 40800217) Es liegt ein erledigter Auftrag in der Confirmation FIFO, dies ist aber kein Sendeauftrag. Bitte holen Sie die erledigten Aufträge immer in der Reihenfolge ab in der Sie die Aufträge aktiviert hatten.
	err.PORT_NOT_OPEN (0x C0800213) Der angegebene Port ist nicht geöffnet.
	err.PORT_WRONG_MODE (0xC0800203) Der Port wurde im Zeichen-Modus geöffnet und kann nicht im Blockmodus beauftragt werden.

8.3.5 :PortIsReceiveDone

:PortIsReceiveDone()	
Prüft auf einen abgearbeiteten Empfangsauftrag in der Confirmation FIFO Warteschlange. Wenn vorhanden wird dieser entfernt und dessen **id** und Daten als Rückgabewert übergeben. Es wird immer einer und der zeitlich älteste Empfangsauftrag entfernt. Bitte beachten Sie, dass die Funktion immer nur den zeitlich ältesten Auftrag in der Confirmation FIFO entfernt. Liegt dabei kein erledigter Empfangsauftrag vor, kann die Funktion nicht erfolgreich durchgeführt werden.	
Argumente:	Keine
Rückgabewerte: **status**	**port.STA_ACK_TIMEOUT** (3) Innerhalb der konfigurierten **ackdelaytime** konnte kein Zeichen empfangen werden
	port.STA_CHAR_TIMEOUT (4) Der Empfang wurde bedingt durch Ablauf der **chardelaytime** beendet,
	port.STA_SIZE_REACHED (2) Die im Auftrag festgelegte Zeichenzahl wurde empfangen.
	port.STA_PATTERN_MATCH (1) Der Empfangsauftrag wurde erfolgreich ausgeführt. Die konfigurierte Endekennung in **endpattern** und **endmask** wurde erkannt und der Empfang dadurch beendet.
	nil, wenn kein Empfangsauftrag abgeschlossen oder statt eines erledigten Empfangsauftrages ein anderer erledigter Auftrag in der Confirmation FIFO zu Abholung vorliegt. Dieser ist zunächst aus der Confirmation FIFO mit seiner entsprechenden Funktion aus der Warteschlange zu entfernen. Bei einem Rückgabewert von **nil** sind auch alle anderen Rückgabewerte **nil**.
data	Der empfangene Datenstring.
	leerer String der Datenlänge 0**,** wenn der Status **port.STA_ACK_TIMEOUT** oder **port.STA_CHAR_TIMEOUT** enthält, oder **error ≠ nil** ist..
error	**nil**, wenn kein Fehler aufgetreten ist.
	port.ERR_PARITY_ERROR (2) Paritätsfehler während des Empfanges festgestellt
	port.ERR_FRAMING_ERROR (4) Telegrammrahmenfehler, wie Start-Stoppbitfehler festgestellt.
	port.ERR_BREAK_DETECTED (1) Ein break auf der seriellen Schnittstelle wurde empfangen.
id	**nil**, wenn dem Empfangsauftrag bei seiner Erzeugung mit **:PortReceive()**keine **id** mitgegeben wurde.
	id, die dem Empfangsauftrag bei der Erzeugung mit **:PortReceive()** mitgegeben wurde.
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_FIFO_EMPTY** (0x40800204) Die Confirmation FIFO enthält gar keine abgearbeiteten Aufträge
	err.PORT_NO_CONFIRMATION (0x 40800217) Es liegt ein erledigter Auftrag in der Confirmation FIFO, dies ist aber kein Empfangsauftrag. Bitte holen Sie die erledigten Aufträge immer in der Reihenfolge ab in der Sie die Aufträge aktiviert hatten.
	err.PORT_NOT_OPEN (0x C0800213) Der angegebene Port ist nicht geöffnet.
	err.PORT_WRONG_MODE (0xC0800203) Der Port wurde im Zeichen-Modus geöffnet und kann nicht im Blockmodus beauftragt werden.

Beispiel für den Aufruf der Funktion:

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

end
```

8.3.6 :PortIsExchangeDone

:PortIsExchangeDone()	
Prüft auf einen abgearbeiteten Sende-Empfangsauftrag in der Confirmation FIFO Warteschlange. Wenn vorhanden wird dieser entfernt und dessen **id** und Daten als Rückgabewert übergeben. Es wird immer einer und der zeitlich älteste Sende-Empfangsauftrag entfernt. Bitte beachten Sie, dass die Funktion immer nur den zeitlich ältesten Auftrag in der Confirmation FIFO entfernt. Liegt dabei kein erledigter Send-Empfangsauftrag vor, kann die Funktion nicht erfolgreich durchgeführt werden. Gibt den Inhalt der Empfangsdaten eines mit der Funktion **:PortExchange** in die Request- FIFO-Queue eingestellten Sendeblocks zurück.	
Argumente:	keine
Rückgabewerte: **status**	**port.STA_ACK_TIMEOUT** (3) Innerhalb der konfigurierten **ackdelaytime** konnte kein Zeichen empfangen werden
	port.STA_CHAR_TIMEOUT (4) Der Empfang wurde bedingt durch Ablauf der **chardelaytime** beendet,
	port.STA_SIZE_REACHED (2) Die im Auftrag festgelegte Zeichenzahl wurde empfangen.
	port.STA_PATTERN_MATCH (1) Der Empfangsauftrag wurde erfolgreich ausgeführt. Die konfigurierte Endekennung in **endpattern** und **endmask** wurde erkannt und der Empfang dadurch beendet.
	nil, Wenn der Empfangsblock nicht abgeschlossen, oder nicht erster Block in der Confirmation FIFO Queue ist. In diesem Fall sind auch alle anderen Rückgabewerte **nil**.
data	**string**, der empfangene Datenstring.
	Leerer String der Datenlänge 0**,** wenn der Status **port.STA_ACK_TIMEOUT** oder **port.STA_CHAR_TIMEOUT** enthält, oder **error ≠ nil** ist..
error	**nil**, wenn kein Fehler aufgetreten ist.
	port.ERR_PARITY_ERROR (2) Paritätsfehler während des Empfanges festgestellt
	port.ERR_FRAMING_ERROR (4) Telegrammrahmenfehler, wie Start-Stoppbitfehler festgestellt.
	port.ERR_BREAK_DETECTED (1) Ein break auf der seriellen Schnittstelle wurde empfangen.
id	**nil**, wenn dem Auftrag bei der Erzeugung im Request FIFO keine **id** mitgegeben wurde.
	id, die dem Empfangsblock bei der Erzeugung im Request FIFO mitgegeben wurde.
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_FIFO_EMPTY** (0x40800204) Die Confirmation FIFO enthält gar keine abgearbeiteten Aufträge
	err.PORT_NO_CONFIRMATION (0x 40800217) Es liegt ein erledigter Auftrag in der Confirmation FIFO, dies ist aber kein Sende-Empfangsauftrag. Bitte holen Sie die erledigten Aufträge immer in der Reihenfolge ab in der Sie die Aufträge aktiviert hatten.
	err.PORT_NOT_OPEN (0x C0800213) Der angegebene Port ist nicht geöffnet.
	err.PORT_WRONG_MODE (0xC0800203) Der Port wurde im Zeichen-Modus geöffnet und kann nicht im Blockmodus beauftragt werden.

8.3.7 :PortAbort

:PortAbort ()	
Löscht alle Aufträge aus der Request und Confirmation FIFO Warteschlange Die Verarbeitung des aktuell bearbeiteten Blocks wird sofort abgebrochen.	
Argumente:	keine
Rückgabewerte:	keine
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_NOT_OPEN** (0x C0800213) Der angegebene Port ist nicht geöffnet.
	err.PORT_WRONG_MODE (0xC0800203) Der Port wurde im Zeichenmodus geöffnet und kann nicht im Blockmodus beauftragt werden.

9 Serielle Kommunikation im Zeichenmodus

Im Zeichenmodus arbeitet der UART im Voll-Duplex betrieb. Er kann gleichzeitig Daten Empfangen und Senden (nicht bei RS485). Auch bei RS485 ist die Kommunikation des Skriptes im Zeichenmodus möglich, arbeitet jedoch auf der Leitung wie im Blockmodus.

Die Größe der Empfangs- und Sende-FiFOs ist je 256 Byte.

Die Verarbeitung kann den folgenden Grafiken entnommen werden:

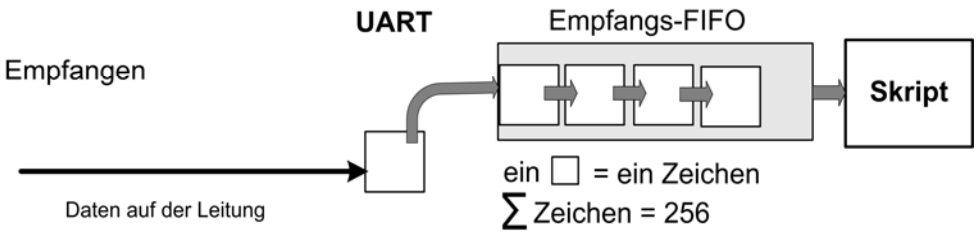


Abbildung 12: Empfangsverarbeitung im Zeichen-Modus

Im FIFO-Mode werden vom UART bis zu 256 Zeichen aufgezeichnet. Um einen Überlauf des Empfangs-FIFO's zu vermeiden, muss in dieser Empfangszeit der FIFO von netSCRIPT mind. einmal ausgelesen und damit geleert werden.

Wenn der Empfangs-FIFO überläuft, werden weitere Zeichen ignoriert. In diesem Fall wird beim nächsten Lese-Aufruf mit „PortGetChar„die Fehlermeldung „ERR_RX_FIFO_OVERFLOW“ gesetzt.

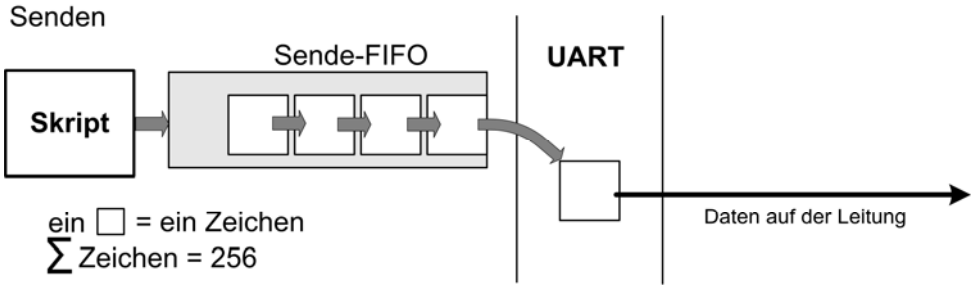


Abbildung 13: Sendeverarbeitung im Zeichen-Modus

Von netSCRIPT aus können max. 256 Zeichen in den Sende-FIFO eingestellt werden, die vom UART nacheinander auf der Leitung ausgegeben werden. Sollen mit einer Skript-Funktion mehr Zeichen in den FIFO eingestellt werden, als freier Platz im Sende-FIFO ist, wird der Schreibauftrag vom FIFO abgewiesen.

Verhalten bei Leitungsbruch:

Wenn am Eingang (RxD) für 11 Bit-Zeiten Pegel 0 anliegt, d.h. 11 Nullbits empfangen werden (kommt bei normaler Übertragung nicht vor) wird das Break-Flag gesetzt und ein Null-Zeichen gespeichert. Im Zeichen-Modus ist dieses Zeichen mit dem Break-Flag sofort in der FIFO sichtbar.

Beim Senden wird ins leere gesendet. Wenn RTS/CTS-Handshake aktiv ist (durch den Initialisierungsparameter eingeschaltet ist) und der empfangene Pegel auf CTS ungleich dem Parameter Handshake Polarity ist, wird nicht gesendet

9.1 Sende- und Empfangsfunktionen

9.1.1 :PortGetChar

:PortGetChar([n])	
Liest **n** Zeichen aus dem FIFO-Empfangspuffer (wenn verfügbar).	
Argumente **n**:	**n** ist optional, gibt an wie viele Zeichen aus dem FIFO Eingangsspeicher gelesen werden sollen. Ist **n** nicht angeben, wird ein Zeichen gelesen. Ist **n** negativ, werden alle vorhandenen Zeichen gelesen. Sind keine Zeichen vorhanden, wird keine Fehlermeldung generiert.
Rückgabewerte: **string**	Wenn **n** Zeichen im Empfangs-FIFO vorhanden sind, werden diese als String zurückgegeben. Enthält der Empfangs-FIFO weniger als **n** Zeichen, wird nil zurückgegeben.
err	**port.ERR_PARITY_ERROR (2)** mind. ein Zeichen hat ein Paritätsfehler. **port.ERR_FRAMING_ERROR** (4) Ein Rahmenfehler wurde während des Datenempfangs festgestellt. **port.ERR_BREAK_DETECTED** (1) Ein Leitungsbruch wurde erkannt. **port.ERR_RX_FIFO_OVERFLOW** (8) ...Der FIFO hatte seit des letzten Aufrufs ein Überlauf.
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_FIFO_EMPTY** (0x40800204) Keine Eingangsdaten vorhanden. Fehlermeldung, nur wenn **n**>0 ist.
	err.PORT_INVALID_PARAMETER (0x40800216) Ein Funktionsargument hat einen falschen Typ oder **n** ist > 256
	err.PORT_NOT_OPEN (0xC0800213) Es wurde versucht auf einen nicht geöffneten Port zuzugreifen.
	err.PORT_WRONG_MODE (0xC0800203) Der Port wurde im Blockmodus geöffnet.

9.1.2 :PortPutChar

:PortPutChar(str)	
Übergibt den String **str** an den Sende-FIFO. Ist nicht genügend Platz im Sende-FIFO, wird nichts übergeben. Es wird nur die Anzahl Datenbits übergeben, die in der Datentabelle für die Schnittstelle definiert sind.	
Argumente **str**:	Akzeptiert jede Anzahl von Argumenten. Jedes Argument kann ein Zeichen oder eine Zahl sein. String-Argumente werden byteweise gesandt. Wenn weniger als acht Datenbits konfiguriert werden, werden die oberen Bits jedes Bytes in dem String ignoriert. Numerische Argumente werden zu ganzen Zahlen umgewandelt, und die niedrigsten bis zu acht Bit werden gesandt.
Rückgabewerte:	**true**, wenn der Auftrag in den Sende-FIFO eingestellt werden konnte.
	false, wenn der Auftrag nicht in den Sende-FIFO eingestellt werden konnte.
Status-/Fehlercode in der Variablen „lasterror“	**err.PORT_NOT_OPEN** (0xC0800213) Es wurde versucht auf einen nicht geöffneten Port zuzugreifen.
	err.PORT_WRONG_MODE (0xC0800203) Der Port wurde im Blockmodus geöffnet.

Hinweis: Sollen nach einer Übertragungsstörung alle im Empfangs- bzw. Sende-FIFO befindlichen Daten gelöscht werden, ist der Port mit der Funktion „:PortClose()“ zu schließen und mit „:PortOpen()“ neu zu initialisieren.

10 Funktionen zur Kommunikation mit dem übergeordneten E/A-Netzwerk

Zur Kommunikation zwischen netSCRIPT und der übergeordneten Steuerung steht in jeder Richtung ein 1024 Bytes langer Datenpuffer zur Verfügung. Beide Seiten müssen sich gegenseitig signalisieren, wenn gültige Daten für die andere Seite bereitstehen, und quittieren wenn Daten übernommen wurden. Diesen Vorgang bezeichnet man als Handshake. Es gibt zwei Betriebsarten, die sich darin unterscheiden, wie der Handshake realisiert wird:

1. Im **Direktmodus** gibt es eine Lese- und eine Schreibfunktion, mit denen beliebige Daten übertragen werden können. Hierbei wird keine Empfangsquittung / Neudateninformation automatisch generiert. Hier muss ggf. im Skript eine entsprechende Empfangs-Bestätigung / -Erkennung programmiert werden.
2. Im **Handshakemodus** ist ein bestimmtes Handshakeprotokolls vorgegeben, das von netSCRIPT automatisch durchgeführt wird. Das Skript braucht sich also nicht darum zu kümmern. Allerdings muss dieses Protokoll auch in der übergeordneten Steuerung programmiert werden.
   Auch hier gibt es einen Lesebefehl, der jedoch nur dann Daten zurückliefert, wenn gemäß des Protokolls gültige Daten vorliegen. Die Übernahme dieser Daten wird dann automatisch quittiert.

   Der Schreibbefehl schreibt Daten nur dann, wenn die Steuerung den Empfang der zuletzt gesendeten Daten bereits quittiert hat. In beiden Richtungen ist zusätzlich zum Datenpuffer ein 24 Bytes großer Header definiert, über den die Handshake-Informationen ausgetauscht werden.

Für beide Kommunikationsverfahren gilt:

Die Übergabe der Daten, an das der seriellen Schnittstelle übergeordnete IO-Netzwerk, in der Regel das Netzwerk der steuernden SPS wird „BUS IO“ genannt, ist standardisiert. Für alle Netzwerke ob z.B. PROFIBUS, CANopen, DeviceNet oder Real-Time Ethernet basierten Systeme wie EtherCAT, EtherNet/IP, PROFINET IO gilt das gleiche Datenübergabeverfahren.

Innerhalb von netSCRIPT werden ein 24Byte großer Datenkopf und maximal 1024 Bytes Nutzdaten in beide Übertragungsrichtungen als Übergabespeicher reserviert. Als Eingabe wird die Richtung von netSCRIPT zur Steuerung bezeichnet weil die Steuerung die Eingabe (Eingänge) liest, als Ausgabe die Richtung von der Steuerung zu netSCRIPT weil die Steuerung die Ausgabe (Ausgänge) schreibt.

Byte	Ausgabe	Eingabe	Bedeutung
0-23	Header für Handshakemodus	Header für Handshakemodus	Handshakeflags, Längenangabe, Fehlercodes (nur für Hanshakemodus).
24-1047	Max. 1024 Bytes Daten	Max. 1024 Bytes Daten	Nutzdaten

Tabelle 33: netSCRIPT, Transferdatenstruktur

Im Direktmodus ist der Datenkopf (Byte 0 **...** 23) ebenfalls vorhanden, wird jedoch nicht genutzt.

Auf die beiden E/A-Übergabespeicher hat der Anwender keinen unmittelbaren Zugriff. Stattdessen wird die Datenübergabe innerhalb netSCRIPT auf Funktionen abgebildet, die die nötigen Zugriffe und Synchronisationsmechanismen abhandeln. Dabei werden mit den Funktionen nur die Nutzdaten transferiert.

Für IO-Netzwerke ist in der Regel die Übertragung einer Datenmenge von 1024 Bytes pro Slave-Gerät zu viel. Es muss nicht die gesamte Datenmenge an die übergeordnete Steuerung übertragen und von ihr Empfangen werden. Die reelle Datenmenge die über das IO-Netzwerk tatsächlich übertragen wird, wird in der Steuerung konfiguriert. Beispiel:
In der Steuerung ist für das netSCRIPT-Gerät eine Ausgangslänge von 64 Bytes und eine Eingangslänge von 70 Bytes eingestellt. Abzüglich des 24 Byte Datenkopfes ergibt dies effektiv 40 Nutzdatenbytes die netSCRIPT von dort pro Auftrag empfangen und 46 Bytes die von netSCRIPT gesendet werden können. Diese Nutzdatenmenge sollte mit der maximalen Datenzahl die über die serielle Schnittstelle des netSCRIPT-Gerätes übertragen wird abgestimmt werden, um die Daten komplett in einem Auftrag über die E/A-Schnittstelle übergeben zu können.

Hinweis:

Es ist zu beachten, dass die E/A-Transferdatenstruktur von netSCRIPT zwar standardisiert ist, einzelne Elemente der Struktur aber über das Konfigurationswerkzeug SYCON.net vor der Übergabe an das überlagerte Netzwerk verschoben werden können. Zum Beispiel ist es möglich das erste Übergaberegister ans Ende des Übergabespeichers zu mappen. Lesen Sie dazu das Benutzerhandbuch netGateway_DTM_de.pdf.

10.1 Bus IO-Kommunikation Einrichten und Beenden

Um die Kommunikation zur Bus IO-Schnittstelle aufnehmen zu können, muss ein Objekt eingerichtet werden, über die in netSCRIPT die weiteren Funktionen der Schnittstelle angesprochen werden können.

Selbst wenn die Bus IO-Schnittstelle richtig eingerichtet ist, können Nutzdaten nur gelesen und geschrieben werden, wenn die übergeordnete Steuerung den Betrieb auch freigegeben hat. Dies hat im Synchronisationsregister **AppHandshake** zu erfolgen und wird automatisch von netSCRIPT bei der Ausführung der Lese- und Schreibfunktionen geprüft.

10.1.1 BusIOOpen

Es wird die Kommunikation zur Bus IO-Schnittstelle initialisiert, und ein neues Objekt erzeugt.

BusIOOpen([Instanznr], [config])	
Initialisiert und erzeugt ein Bus IO-Objekt.	
Argumente: **Instanznr**	wenn leer, wird die Default-Instanz (2) geliefert. Eine Instanz kann nur einmal geöffnet werden.
config	Konfigurationstabelle, siehe unten.
Rückgabewerte:	**instanz,** wenn erfolgreich. (Ist bei allen weiteren Schnittstellenaufrufen zu verwenden.)

	nil, die Instanz konnte nicht eingerichtet werden.
Status-/Fehlercode in der Variablen „lasterror“	**err.BUSIO_INVALID_CONFIG** (0xC0800303) Ein **config** Parameter ist ungültig.
	err.BUSIO_NO_SUCH_INSTANCE (0x 40800301) Instanz kann nicht geöffnet werden.
	err.BUSIO_ALREADY_OPEN (0x40800302) Instanz ist bereits geöffnet.

Aufbau der Konfigurationstabelle

Parameter	Schlüsselwort	Wertebereich	Default
Direktmodus	**directmode**	true/false (true = Direktmodus false = Handshakemodus)	false
Max. Output-Länge im Direktmodus (Gateway→ Lua)	**maxreadlen**	1..1024	1024
Max. Input-Länge im Direktmodus (Lua→ Gateway)	**maxwritelen**	1..1024	1024

Tabelle 34: netSCRIPT, BusIO Konfigurationstabelle

Die Parameter der BusIO Konfigurationstabelle sind nur für den Direktmodus erforderlich.

Beispiel: öffnen der Schnittstelle im Handshakemodus:

```
Gw = BusIOOpen()
```

öffnen der Schnittstelle im Direktmodus:

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

Mit den Parametern maxreadlen / maxwritelen läßt sich im Direktmodus der Umfang von internen Kopiervorgängen der Input/Output-Puffer beschränken. Ohne diese Angaben werden jeweils 1024 Bytes kopiert.

Alle Lese- und Schreibaufträge an diese Schnittstelle müssen mit der zurück gelieferten Instanz der **BusIOOpen**-Funktion als Präfix erfolgen. Dieser Präfix muss vor dem „**:**“ der Funktion stehen.

Hinweis:
Mit dem Aufruf dieser Funktion werden alle Inputdaten (Daten an die übergeordnete Steuerung) mit 0 initialisiert.

10.1.2 :BusIOClose

Die Verbindung zum Bus IO-Schnittstelle wird geschlossen, alle bereitgestellten Ressourcen werden im System wieder freigegeben. Die Kommunikation am übergeordneten Netzwerk findet weiterhin statt. Um die Kommunikation per netSCRIPT über die Schnittstelle erneut zu starten muss der Aufruf der BusIOOpen()-Funktion erfolgen.

:BusIOClose ()	
Schließt die Bus-IO-Instanz.	
Argumente	keine
Rückgabewerte:	keine
Status-/Fehlercode in der Variablen „lasterror“	**err.BUSIO_NOT_OPEN** (0x 40800304) Instanz kann nicht geschlossen werden, da nicht geöffnet.

Beispiel:

```
Gw:BusIOClose()
```

Hinweis:
Mit dem Aufruf dieser Funktion werden alle Inputdaten (Daten an die übergeordnete Steuerung) mit 0 initialisiert.

10.2 Read- / Write-Funktionen für den Direktmodus

Im Direktmodus bleiben die 24 Byte langen Header in beiden Richtungen unberücksichtigt. Es erfolgt also keine automatische Signalisierung, wenn Daten bereit stehen bzw. übernommen wurden. Eine solche Signalisierung muss im Skript und in der übergeordneten Steuerung realisiert und über die 1024-Byte-Datenbereiche abgewickelt werden.

10.2.1 :BusIOReadDirect()

:BusIOReadDirect([Offset[, Länge]])	
Im Direktmodus: Liest einen Nutzdatenstring von der übergeordneten Steuerung über die Bus IO-Schnittstelle.	
Argumente **Offset**	Offset: die Startposition im Datenpuffer der Nutzdaten (0..maxreadlen-1, Default: 0) maxreadlen ist die, mit der Funktion BusIOOpen definierte, zu übertragende Datenlänge.
Länge	Länge: 0.. maxreadlen-Offset, Default: maxreadlen-Offset maxreadlen ist die, mit der Funktion BusIOOpen definierte, zu übertragende Datenlänge.
Rückgabewerte: **string**	**Outputdaten**. Wenn nach BusIOOpen noch keine Daten von der übergeordneten Schnittstelle übermittelt wurden, wird ein String aus Nullbytes zurückgegeben. **nil**: wenn ein Fehler bei Offset/Länge vorliegt.
Status-/Fehlercode in der Variablen „lasterror“	**err.BUSIO_BUFFER_LENGTH_EXCEEDED** (0x40800314) Ungültige Argumente: Offset oder Länge.
	err.BUSIO_NOT_OPEN (0x C0800304) Die Instanz ist nicht geöffnet.
	err.BUSIO_WRONG_MODE (0xC0800306) Die Funktion wurde für eine im Handshakemodus geöffnete Schnittstelle aufgerufen

Bei dieser Funktion bleiben die ersten 24 Bytes der Schnittstelle unberücksichtigt und unbehandelt. Der Sender erhält darüber **keine** Rückmeldung, dass die Daten gelesen wurden, und somit neue Daten gesendet werden können.

10.2.2 :BusIOWriteDirect()

:BusIOWriteDirect(Offset, string, [fConfirm])		
Im Direktmodus: Schreibt einen Nutzdatenstring in den Sendepuffer zur überlagerten Steuerung. Dieser Puffer wird jedoch nur zur Übertragung freigegeben, wenn in **fConfirm** nichts, oder true steht. Solange dieses Argument auf false steht, wird das in den Sendepuffer geschriebene nicht übertragen. Damit ist es möglich den Puffer in Einzelschritten zu beschreiben.		
Argumente:	**Offset**	Die Startposition im Datenpuffer (0..maxwritelen-1) maxwritelen ist die, mit der Funktion BusIOOpen definierte, zu übertragende Datenlänge.
	string	Sendedaten. Länge 0..maxwritelen-Offset
	fConfirm	**true**: (Default): Nach Ausführen des Befehls wird der Datenpuffer im nächsten E/A-Zyklus an die übergeordnete EA-Schnittstelle kopiert. **false:** Der Datenpuffer wird nicht kopiert.
Rückgabewerte:		**true**, wenn die Daten im Puffer abgelegt wurden.
		false, wenn die Daten nicht im Puffer abgelegt wurden.
Status-/Fehlercode in der Variablen „lasterror“		**err.BUSIO_BUFFER_LENGTH_EXCEEDED** (0x40800314) Ungültige Argumente: Offset oder Länge.
		err.BUSIO_NOT_OPEN (0x C0800304) Die Instanz ist nicht geöffnet.
		err.BUSIO_WRONG_MODE (0xC0800306) Die Funktion wurde für eine im Handshakemodus geöffnete Schnittstelle aufgerufen

10.3 Datenkopf für den Handshakemodus

Jeder Datenkopf im Übergabespeicher beginnt mit einem 4 Byte großen Synchronisationsregister. Über dieses Register wird die Übergabe der über die IO Schnittstelle geschriebenen und gelesenen Daten (im Handshakemodus) mit der übergeordneten Steuerung synchronisiert. Es folgt ein 4Byte großes Register das die Datenanzahl der übergebenen Daten beinhaltet. In der Datenrichtung Eingabe stehen zwei weitere Register zur Verfügung, um Fehlernachrichten an die Steuerung abzusetzen.

Der Datentransfer über die Bus IO-Schnittstelle mit Handshakemodus erfolgt über folgenden strukturellen Aufbau der beiden E/A-Transferdatenstrukturen.

Byte	Ausgabe(von der Steuerung)	Eingabe(zur Steuerung)	Bedeutung
0-3	App_Handshake	Prot_Handshake	Synchronisationsregister siehe Abschnitt „Handshake und Initialisierung der E/A-Kommunikation“ Seite 112
4-7	App_Tx_Bytecount	Prot_Rx_Bytecount	Anzahl gültiger Bytes im Nutzdatenteil.
8-11	Reserviert	Prot_Rx_Error	Fehlernummer, programmierbar, siehe Funktion :BusIOSetError() Seite 110.
12-15	Reserviert	Prot_Tx_Error	Fehlernummer, programmierbar, siehe Funktion :BusIOSetError() Seite 110.
16-19	Reserviert	Reserviert	-
20-23	Reserviert	Reserviert	-
24-1047	Max. 1024 Bytes Daten	Max. 1024 Bytes Daten	Nutzdaten

Tabelle 35: netSCRIPT, Transferdatenstruktur Handshakemodus

10.4 Read- / Write-Funktionen für den Handshakemodus

10.4.1 :BusIORead

:BusIORead([fConfirm])	
Im Handshakemodus: Liest einen Nutzdatenstring von der Steuerung über die Bus IO-Schnittstelle soweit verfügbar und bestätigt dies ggf.	
Argumente: **fConfirm**	Gibt an, ob das Lesen der Nutzdaten der Steuerung bestätigt werden soll. **true** (default), der Datenempfang wird bestätigt. **false,** der Datenempfang wird nicht bestätigt. In diesem Fall muss zu einem späteren Zeitpunkt das Lesen der Daten mit der gleichen Funktion und dem Parameterwert „True“ erfolgen. Gemäß Übergabeprozedur kann die Steuerung keine weiteren Daten senden, wenn ältere nicht bestätigt wurden.
Rückgabewerte:	**string**: von der Bus IO-Schnittstelle gelesener Datenstring. Falls die Längenangabe im Datenheader größer ist, als der Speicherbereich (>1024 Byte), wird **nil** zurückgegeben und lasterror = err.BUSIO_STRING_TOO_LONG gesetzt. Der Datenempfang wird trotzdem bestätigt, außer wenn fConfirm = false übergeben wurde.
	nil, bei Fehler, oder keine gültigen Daten.
Status-/Fehlercode in der Variablen „lasterror“	**err.BUSIO_RECEIVE_NO_DATA** (0x 40800321) Es sind keine neuen Daten zum Einlesen vorhanden.
	err.BUSIO_RECEIVE_DISABLED (0x 40800322) Der Datenempfang ist von der Steuerung nicht freigegeben. Das entsprechende Bit im App_Handshakeregister ist nicht gesetzt. Siehe Bediener-Manual netGateway.
	err.BUSIO_STRING_TOO_LONG (0x 40800313) Fehler im Protokollkopf, ungültige Längenangabe im Header von der übergeordneten Steuerung erhalten. Handshake wird nicht bestätigt.
	err.BUSIO_NOT_OPEN (0x 40800304) Die an die Instanz gekoppelte Bus IO-Schnittstelle wurde nicht geöffnet.
	err.BUSIO_WRONG_MODE (0xC0800306) Die Funktion wurde für eine im Direktmodus geöffnete Schnittstelle aufgerufen.

10.4.2 :BusIOWrite

:BusIOWrite(string)	
Im Handshakemodus: Schreibt einen Nutzdatenstring in den Sendeübergabepuffer zur Steuerung, sofern ein zeitlich vorher geschriebener Datenstring von der Steuerung bereits quittiert wurde.	
Argumente:	**string**, zu sendende Nutzdaten als String.
Rückgabewerte:	**true**, wenn die Daten an übergeben werden konnten.
	false, wenn die Daten nicht übergeben werden konnten.
Status-/Fehlercode in der Variablen „lasterror“	**err.BUSIO_SEND_NOT_READY** (0x40800311) Der Empfang der zuletzt gesendeten Daten wurde von der Steuerung noch nicht bestätigt.
	err.BUSIO_SEND_DISABLED (0x 40800312) Das Schreiben der Daten ist von der Steuerung nicht freigegeben. Das entsprechende Bit im App_Handshakeregister ist nicht gesetzt. Siehe Bediener-Manual netGateway.
	err.BUSIO_STRING_TOO_LONG (0x 40800313) Es gab einen Bufferüberlauf, der zu übergebene Datenstring ist zu lang (>512 Byte).
	err.BUSIO_NOT_OPEN (0x 40800304) Die an die Instanz gekoppelte Bus IO-Schnittstelle wurde nicht geöffnet.
	err.BUSIO_WRONG_MODE (0xC0800306) Die Funktion wurde für eine im Direktmodus geöffnete Schnittstelle aufgerufen**.**

Beispiel für eine einfache Datenübergabe:

```
Gw = BusIOOpen()

Gw:BusIOWrite(„Hallo“)
```

–- Es wird der String „Hallo“ an die Bus IO-Schnittstelle geschrieben. Der Steuerung wird signalisiert das neue Daten vorliegen. —

Die Funktion **BusIOOpen()** muss im gesamten Skript nur einmal bei der Initialisierung aufgerufen werden. Damit ist die Schnittstelle bis zum Aufruf der Funktion **Gw: :BusIOClose ()** verwendbar.

10.5 Rücksetzen-Auftrag im Handshakemodus

Die Steuerung ist in der Lage über die BUS IO-Schnittstelle ein Rücksetz-Kommando an netSCRIPT zu senden. Dieser Rücksetzauftrag wird nicht automatisch von der netSCRIPT-Steuerung abgehandelt, sondern muss in wiederkehrenden Abständen im netSCRIPT Programm geprüft werden, ob ein solches Kommando von der Steuerung abgesetzt wurde.

Für den Programmablauf, der beim „Rücksetzen“ abläuft ist alleine der netSCRIPT-Anwender verantwortlich. Selbstverständlich kann das Kommando auch für andere eigene Zwecke verwendet werden.

Die Position des Reset-Bits in den Kopfdaten ist in Abschnitt 10.9.1 beschrieben.

10.5.1 :BusIOIsReset

:BusIOIsReset()	
Im Handshakemodus: Liest das **APP_HS_RESET_CMD** Bit des Synchronisationsregisters. Siehe Abschnitt „Datenrichtung von Übergeordneter Steuerung nach netSCRIPT“ Seite 113. Prüft, ob ein unbestätigtes Reset-Kommando vorhanden ist. Die Reaktion auf das Vorhandensein dieses Kommandos ist im netSCRIPT zu programmieren.	
Argumente	keine
Rückgabewert **bool**	**true**, es liegt ein unbestätigtes Rücksetz-Kommando vor.
	false, es liegt kein unbestätigtes Rücksetz-Kommando vor.
Status-/Fehlercode in der Variablen „lasterror“	**err.BUSIO_NOT_OPEN** (0x 40800304) Die an die Instanz gekoppelte Bus IO-Schnittstelle wurde nicht geöffnet.
	err.BUSIO_WRONG_MODE (0xC0800306) Die Funktion wurde für eine im Direktmodus geöffnete Schnittstelle aufgerufen.

10.5.2 :BusIOResetDone

:BusIOResetDone()	
Im Handshakemodus: Schreibt das **PROT_HS_RESET_ACK** Bit des Synchronisationsregisters. Siehe Abschnitt „Datenrichtung von netSCRIPT zur übergeordneten Steuerung“ Seite 114. Der Empfang des von der Steuerung abgesetzten Rücksetz-Kommandos wird bestätigt. In der Regel wird diese Funktion aufgerufen, wenn in netSCRIPT das Rücksetzen durchgeführt wurde.	
Argumente	keine
Rückgabewerte	keine
Status-/Fehlercode in der Variablen „lasterror“	**err.BUSIO_NOT_OPEN** (0x 40800304) Die an die Instanz gekoppelte Bus IO-Schnittstelle wurde nicht geöffnet.
	err.BUSIO_WRONG_MODE (0xC0800306) Die Funktion wurde für eine im Direktmodus geöffnete Schnittstelle aufgerufen.

10.6 Bereitmeldung an die Steuerung im Handshakemodus

10.6.1 :BusIOSetRun

:BusIOSetRun(Wert)	
Im Handshakemodus: Meldet das Ende der Initialisierung und signalisiert diesen Status der übergeordneten Steuerung. Diese Funktion sollte innerhalb von netSCRIPT die erste Funktion sein die nach der Initialisierung aufgerufen wird. Es wird das Signal **PROT_HS_RUN_IND** an die übergeordnete Steuerung übergeben. Siehe auch Abschnitt 10.9.1.2 Seite 114.	
Argumente: **Wert**	**true,** Run-Bit auf 1 setzen **(**setzt den Status auf bereit).
	false, Run-Bit auf 0 setzen (setzt den Status auf nicht bereit).
Rückgabewert	kein
Status-/Fehlercode in der Variablen „lasterror“	**err.BUSIO_NOT_OPEN** (0x 40800304) Die an die Instanz gekoppelte Bus IO-Schnittstelle wurde nicht geöffnet.
	err.BUSIO_WRONG_MODE (0xC0800306) Die Funktion wurde für eine im Direktmodus geöffnete Schnittstelle aufgerufen.

10.7 Fehler Status übertragen im Handshakemodus

Es ist möglich der Steuerung über die Bus IO-Schnittstelle per netSCRIPT Fehlermeldungen zu übergeben. Dazu stehen in dem Eingangs-Übergabespeicher an die Steuerung zwei Fehlerregister zur Verfügung die per netSCRIPT-Funktion beschrieben werden können.

Per Definition ist ein Register für Empfangsfehler und ein Register für Sendefehler vorgesehen. Selbstverständlich können die Fehlerregister auch anderweitig verwendet werden.

10.7.1 :BusIOSetError()

:BusIOSetError(Direction, Errorflag, [Errorcode])	
Im Handshakemodus: Setzt oder löscht eine Fehlernummer im Eingangs-Übergabespeicher zur Steuerung: Datenstruktur siehe Abschnitt „Eingabedaten-Struktur, netSCRIPT → Steuerung“ Seite 111 und im Synchronisationsregister das Bit **PROT_HS_TX_ERROR_IND** bzw. das Bit **PROT_HS_RX_ERROR_IND**. Siehe Abschnitt „Datenrichtung von netSCRIPT zur übergeordneten Steuerung“ Seite 114.	
Argumente: **Direction**	**send**, Errorcode in das Empfangs-Fehlerregister eintragen oder dort löschen (Byte 4-7).
	receive, Errorcode in das Sende-Fehlerregister eintragen oder dort löschen (Byte12-15).
Errorflag	**nil,** Fehler unverändert lassen.
	true, Flag auf 1 setzen und signalisieren.
	false, Flag auf 0 setzen.
Errorcode	**nil** oder weglassen = Fehlerwert unverändert lassen.
	Der 32 Bit unsigned-Fehlerwert kann unabhängig vom Zustand des Errorflags gesetzt werden.
Rückgabewerte:	keine
Status-/Fehlercode in der Variablen „lasterror“	**err.BUSIO_INVALID_PARAMETER** (0x 40800305) Die Übergabewerte sind nicht plausibel.
	err.BUSIO_NOT_OPEN (0x 40800304) Die Kommunikation ist nicht instanziiert.
	err.BUSIO_WRONG_MODE (0xC0800306) Die Funktion wurde für eine im Direktmodus geöffnete Schnittstelle aufgerufen.

10.8 E/A-Datenstruktur zur Datenübergabe von und zur Steuerung/Master im Handshakemodus

10.8.1 Ausgabedaten-Struktur, Steuerung → netSCRIPT

Byte	**Signal**	**Auslesbar mit**
0-3	Ausgabe-Synchronisationsregister	Kein Zugriff, wird automatisch von netSCRIPT gelesen.
4-7	Anzahl der übertragenen Nutzdaten	Kein Zugriff, wird automatisch von netSCRIPT gelesen.
8...23	Nicht genutzt	Nicht genutzt.
24-1047	Ausgabe-Nutzdaten	Funktion **:BusIORead()**

Tabelle 36: netSCRIPT, Transferdatenstruktur von der Steuerung

Aufbau des Synchronisationsregisters siehe Abschnitt „Datenrichtung von Übergeordneter Steuerung nach netSCRIPT“ Seite 113.

10.8.2 Eingabedaten-Struktur, netSCRIPT → Steuerung

Byte	**Signal**	**über netSCRIPT schreibbar mit**
0-3	Eingabe-Synchronisationsregister.	Kein Zugriff, wird von den netSCRIPT-Funktionen automatisch beschrieben.
4-7	Anzahl der übertragenen Nutzdaten.	Kein Zugriff, wird von den netSCRIPT-Funktionen beschrieben.
8-11	Fehlerregister (receive Error) zur Übergabe von Fehlerzuständen.	Beschreibbar von netSCRIPT mit der Funktion **:BusIOSetError()**.
12-15	Fehlerregister (transmit Error) zur Übergabe von Fehlerzuständen.	Beschreibbar von netSCRIPT mit der Funktion **:BusIOSetError().** .
16..23	Nicht genutzt.	Nicht genutzt.
24-1047	Eingabe-Nutzdaten.	Funktion **:BusIOWrite().**

Tabelle 37: netSCRIPT, Transferdatenstruktur zur Steuerung

Aufbau des Synchronisationsregisters siehe Abschnitt „Datenrichtung von netSCRIPT zur übergeordneten Steuerung“ Seite 114.

10.9 Handshake und Initialisierung der E/A-Kommunikation im Handshakemodus

Zwischen dem Bus-Master (übergeordnete Steuerung) und dem netSCRIPT fähigen Gerät wird der Datenaustausch in einem Übergabenverfahren im Header der E/A-Daten organisiert.

Die Grundidee der Übergabe ist, dass es für jede Aktion in den beiden Synchronisationsregistern ein Bitpaar gibt, dass zur Synchronisation dient. Ein Bit dient zur Anforderung der Aktion (CMD), das andere zur Bestätigung (ACK). Eines der beiden liegt im Ausgabe-Synchronisationsregister, eines im Eingabe-Synchronisationsregister. Es wird jeweils von jedem Kommunikationspartner auf das Eingabe-Synchronisationsregister nur lesend auf das Ausgabe-Synchronisationsregister nur schreibend zugegriffen (aus Sicht des jeweiligen Kommunikationspartners).

Eine Aktion wird angefordert, indem ein Kommando-Bit ungleich dem Quittungs-Bit gesetzt wird. Die andere Seite bestätigt die Aktion, indem sie das entsprechende Quittungs-Bit wieder gleich dem Kommando-Bit setzt.

10.9.1 Aufbau der Synchronisationsregister in den E/A-Daten

10.9.1.1 Datenrichtung von Übergeordneter Steuerung nach netSCRIPT

Aufbau des Synchronisationsregisters der Steuerung in Richtung netSCRIPT:

Bit-Nr.	Name	Gelesen von Funktion
0	**APP_HS_TX_CMD**	:BusIORead
1	**APP_HS_RX_ACK**	:BusIOWrite
2 ... 5	Nicht verwendet	
6	**APP_HS_TX_ENABLE_CMD**	automatisch
7	**APP_HS_RX_ENABLE_CMD**	automatisch
8 ... 14	Nicht verwendet	
15	**APP_HS_RESET_CMD**	:BusIOIsReset
16 ... 31	Nicht verwendet	

Tabelle 38: netSCRIPT, Synchronisationsregister zu netSCRIPT

Dabei bedeutet:

APP_HS_TX_CMD

Kommando Ausgabe-Nutzdaten von der Steuerung an netSCRIPT senden aktivieren. Wird von der Funktion **:BusIORead()** automatisch geprüft, ob neue geschriebene Daten vorliegen.

APP_HS_RX_ACK

In der Steuerung von netSCRIPT empfangene Eingabe-Nutzdaten bestätigen. Das Bit wird aus netSCRIPT über die Funktion **:BusIOWrite()** automatisch abgefragt, ob der vorherige Schreibauftrag quittiert wurde.

APP_HS_TX_ENABLE_CMD

Freigabe der Ausgabe-Nutzdatenübertragung von der Steuerung an netSCRIPT. Ist dieses Bit nicht gesetzt, wird netSCRIPT ein über das Bit **APP_HS_TX_CMD** angeforderte Kommando nicht auswerten.

APP_HS_RX_ENABLE_CMD

Freigabe der Eingabe-Nutzdatenübertragung von netSCRIPT an die Steuerung. Ist dieses Bit nicht gesetzt, wird netSCRIPT kein Kommando über das Bit **APP_HS_RX_CMD** absetzen.

APP_HS_RESET_CMD

Wird von der Funktion **:BusIOIsReset()** abgefragt.

	RESET_CMD	RESET_ACK
Ausgangszustand	0	0
Übergeordnete Steuerung fordert Reset an und wird über **:BusIOIsReset()** abgefragt	1	0
Im Skript wird die Funktion **:BusIOResetDone()** aufgerufen.	1	1
Übergeordnete Steuerung hat RESET_CMD zurückzunehmen.	0	1
netSCRIPT nimmt automatisch RESET_ACK zurück.	0	0

Die Kommunikation in netSCRIPT von und zur seriellen Schnittstelle kann von dem E/A Synchronisationsmechanismus unabhängig betrieben werden. Es gibt also keinen Kausalzusammenhang zwischen den beiden Übergabeschnittstellen.

10.9.1.2 Datenrichtung von netSCRIPT zur übergeordneten Steuerung

Aufbau des Synchronisationsregisters von netSCRIPT in Richtung Steuerung:

Bit-Nr.	Name	Wird geschrieben von Funktion
0	**PROT_HS_TX_ACK**	:BusIORead
1	**PROT_HS_RX_CMD**	:BusIOWrite
2	nicht verwendet	
3	**PROT_HS_RUN_IND**	:BusIOSetRun
4	**PROT_HS_TX_ERROR_IND**	:BusIOSetError
5	**PROT_HS_RX_ERROR_IND**	:BusIOSetError
6	**PROTHS_TX_ENABLE_ACK**	automatisch
7	**PROT_HS_RX_ENABLE_ACK**	automatisch
8 ... 14	nicht verwendet	
15	**PROT_HS_RESET_ACK**	:BusIOResetDone

Tabelle 39: netSCRIPT,Synchronisationsregister zur Steuerung

Dabei bedeutet:

PROT_HS_TX_ACK

Ein von der übergeordneten Steuerung aktiviertes Kommando Ausgabe-Nutzdaten wird mit diesem Bit quittiert. Es wird von der Funktion **:BusIORead()** automatisch bedient.

PROT_HS_RX_CMD

Kommando Eingabe-Nutzdaten von netSCRIPT an die übergeordnete Steuerung senden aktivieren. Es wird von der Funktion **:BusIOWrite()** automatisch geprüft.

PROT_HS_RUN_IND

Das netSCRIPT Programm meldet Bereitschaft und signalisiert das Ende seiner Initialisierung. Es wird von der Funkti**on :BusIOSetRun()** bedient.

Dieses Bit wird von netSCRIPT unabhängig vom Steuerbit **APP_HS_RX_ENABLE_CMD** gesetzt.

PROT_HS_TX_ERROR_IND

0: ok, 1: Fehler (Fehlernummer siehe Fehlerregister in den Bytes 12-15). Wird von der Funktion **:BusIOSetError()** geschrieben.

PROT_HS_RX_ERROR_IND

0: ok, 1: Fehler (Fehlernummer siehe Fehlerregister in den Bytes 8-11).Wird von der Funktion **:BusIOSetError()** beschrieben.

PROT_HS_RESET_ACK

Das Reset-Kommando der übergeordneten Steuerung wird bestätigt. Es wird von der netSCRIPT-Funktion **:BusIOResetDone()** beschrieben.

PROT_HS_TX_ENABLE_ACK

Die Freigabe des Ausgabe-Nutzdatenübertragung von der Steuerung in Richtung netSCRIPT wird quittiert. Das Bit wird von netSCRIPT automatisch bedient.

PROT_HS_RX_ENABLE_ACK

Die Freigabe des Eingabe-Nutzdatenübertragung von netSCRIPT in Richtung der Steuerung wird quittiert. Das Bit wird von netSCRIPT automatisch bedient.

10.9.2 Initialisierung der Kommunikation

Start der Kommunikation

Schritt	**Aktion:** **Start der Kommunikation, Initialisierung erfolgt von der übergeordneten Steuerung aus.**	**Handshake-Sendebyte der übergeordneten Steuerung** 7 6 5 4 3 2 1 0	**Handshake-Empfangsbyte der übergeordneten Steuerung** 7 6 5 4 3 2 1 0
1	Speichersituation nach Power on. netSCRIPT meldet sich bereit.	0 0 x x x x 0 0	0 0 0 0 x 0 0 0
2	Die übergeordnete Steuerung eröffnet die Kommunikation mit netSCRIPT. Durch setzen des Bits 6 und 7 wird netSCRIPT erlaubt die Kommunikation mit der übergeordneten Steuerung aufzunehmen.	1 1 x x x x 0 0	0 0 0 0 x 0 0 0
3	netSCRIPT empfängt die Handshakeflags der übergeordneten Steuerung. Diese lösen folgende Aktionen aus: Auf Grund von Bit 7 wird die Senderichtung zur übergeordneten Steuerung freigeschaltet. Auf Grund von Bit 6 wird die Empfangsrichtung von auf der Bus IO-Ebene für netSCRIPT frei geschaltet.	1 1 x x x x 0 0	
4	Der Empfang des Handshakebytes wird von netSCRIPT wie folgt bestätigt. Damit kann der Datenversand an die übergeordnete Steuerung beginnen.		1 1 0 0 x 0 0 0
	Nachdem die übergeordnete Steuerung die Bestätigung von netSCRIPT über die Sende- und Empfangsbereitschaft erhalten hat, kann auch Sie Daten an netSCRIPT senden.		1 1 0 0 x 0 0 0

Tabelle 40: netSCRIPT, Kommunikationsinitialisierung

➪ Nach der Freigabe der Kommunikation kann jeder Kommunikationspartner die Datenkommunikation zu jedem beliebigen Zeitpunkt eigenständig starten.

Hinweis: Der Wert des Bit 3 des **Handshake-Empfangsbyte der übergeordneten Steuerung** (in der obigen Tabelle mit „X“ gekennzeichnet) ist abhängig von Aufruf der Funktion „:BusIOSetRun“. Ist diese Funktion im Skript aufgerufen, hat dieses Bit den Wert 1, andernfalls 0.

10.9.3 Verarbeitungsbestätigung zwischen übergeordneter Steuerung und netSCRIPT

Für jede gesendete Date, von netSCRIPT an die übergeordnete Steuerung und umgekehrt, wird vom Empfänger eine Empfangsbestätigung erwartet. Solange diese Empfangsbestätigung nicht vorliegt, werden keine weiteren Daten an den Empfänger versendet.

Dieser Handshake-Ablauf wird im Folgenden für beide Richtungen beschrieben.

Hinweis: In den folgenden Tabellen wird ein „**x**“ für eine nicht definierte Bitposition und ein „**X**“ für eine definierte, hier aber unbedeutende Bitposition verwendet.

10.9.3.1 Übergeordnete Steuerung in Richtung netSCRIPT

Der Wert des mit X gekennzeichnete Bits ist für diesen Vorgang ohne Bedeutung.

Schritt	**Aktion:** **Die übergeordnete Steuerung sendet Daten an netSCRIPT**	**Handshake-Sendebyte der übergeordneten Steuerung / Empfangsbyte von netSCRIPT** 7 6 5 4 3 2 1 0	**Handshake-Empfangsbyte der übergeordneten Steuerung / Sendebyte von netSCRIPT** 7 6 5 4 3 2 1 0
0	Der Datenzustand in den Handshake- Sende und Empfangspuffern sei wie nach der Kommunikationsinitialisierung. Bit 0 des Handshakebytes hat sowohl im Sende als auch im Empfangspuffer den Wert 0. → Ein Datenversand an netSCRIPT kann eingeleitet werden.	1 1 x x x x X **0**	1 1 x x x x X **0**
1	Die übergeordnete Steuerung stellt die Sendedaten in den Sendebuffer, schreibt die Länge der Nutzdaten in den Header und setzt im Handshakebyte das Bit 0 im Sendebuffer auf 1	1 1 x x x x X **1**	
2	Die Daten werden an netSCRIPT übertragen	1 1 x x x x X **1**	
3	Solange eine Ungleichheit des Bits 0 des Handshakebytes im Sende und Empfangsbuffer der übergeordnete Steuerung bestehet, dürfen keinen neuen Daten zur Versendung bereitgestellt werden.	1 1 x x x x X **1**	1 1 x x x x X **0**
4	netSCRIPT erkennt an der Ungleichheit des Handshake-Bit 0 im Sende und Empfangbuffers, das neue Daten von der übergeordnete Steuerung eingetroffen sind.	1 1 x x x x X **1**	1 1 x x x x X **0**
5	netSCRIPT entnimmt die von der übergeordnete Steuerung gesendeten Daten aus seinem Empfangsbuffer und bestätigt diesen Vorgang durch setzen des Bit 0 des Handshakebytes in seinem Sendebuffer auf 1.		1 1 x x x x X **1**
6	Der Zustand des Handshakebytes von netSCRIPT wird an die übergeordnete Steuerung übertragen.		1 1 x x x x X **1**
7	Die übergeordnete Steuerung erkennt an der Identität des Bits 0 des Handshakebytes im Sende und Empfangsbuffer, dass das netSCRIPT die gesendeten Daten empfangen hat. → Es können neue Daten an netSCRIPT übertragen werden.	1 1 x x x x X **1**	1 1 x x x x X **1**
8	Die übergeordnete Steuerung stellt neue Daten für netSCRIPT in seinen Sendebuffer, schreibt die Länge der Nutzdaten in den Header und setzt dabei das Bit 0 des Handshakebytes auf den Wert 0.	1 1 x x x x X **0**	
9	Die Daten werden an netSCRIPT übertragen.	1 1 x x x x X **0**	
10	Solange eine Ungleichheit im Bit 1 des Handshakebytes zwischen dem Sende und Empfangsbuffer besteht, dürfen keine weiteren Daten zur Versendung bereit gestellt werden.	1 1 x x x x X **0**	1 1 x x x x X **1**
11	netSCRIPT erkennt an der Ungleichheit des Handshake-Bit 0 im Sende- und Empfangbuffers, das neue Daten von der übergeordnete Steuerung eingetroffen sind.	1 1 x x x x X **0**	1 1 x x x x X **1**
12	netSCRIPT entnimmt die Daten aus seinem Empfangsbuffer und setzt das Handshakebit 0 in seinem Sendebuffer auf den Wert des Handshake-Bits 0 in seinem Empfangsbuffer.		1 1 x x x x X **0**
13	Die Daten des Sende-Handshakebuffers von netSCRIPT werden an die übergeordnete Steuerung übertragen.		1 1 x x x x X **0**
14	Die übergeordnete Steuerung erkennt an dem Wechsel des Handshake-Bits 0 in seinem Empfangsbuffers auf 0, dass netSCRIPT die Daten Empfangen hat.		1 1 x x x x X **0**
15	Die Handshake-Bits 0 im Sende und Empfangbuffers haben den selben Wert. Damit ist der Ausgangszustand wie unter Schritt 0 erreicht und der Vorgang kann erneut ablaufen.	1 1 x x x x X **0**	1 1 x x x x X **0**

10.9.3.2 netSCRIPT in Richtung übergeordneter Steuerung

Der Wert eines mit x gekennzeichneten Bits ist für diesen Vorgang ohne Bedeutung.

Schritt	Aktion: netSCRIPT Sendet Daten an die übergeordnete Steuerung	Handshake-Sendebyte der übergeordneten Steuerung / Empfangsbyte von netSCRIPT 7 6 5 4 3 2 1 0	Handshake-Empfangsbyte der übergeordneten Steuerung / Sendebyte von netSCRIPT 7 6 5 4 3 2 1 0
0	Datenzustand vor Beginn des Sendevorgangs. Das Handshake-Bit 1 im Empfangs und im Sendebuffer hat denselben Zustand → Ein Datenversand an die übergeordnete Steuerung kann eingeleitet werden.	1 1 x x x x **0** X	1 1 x x x x **0** X
1	netSCRIPT hat Daten zum Versand bereitgestellt, schreibt die Länge der Nutzdaten in den Header und setzt im Handshake-Byte das Bit 1.		1 1 x x x x **1** X
2	Die Daten werden an die übergeordnete Steuerung übertragen.		1 1 x x x x **1** X
3	Solange die Ungleichheit im Bit 1 des Handshakebytes im Sende und Empfangpuffer von netSCRIPT besteht darf netSCRIPT keine weiteren Daten zur Versendung an die übergeordnete Steuerung bereitstellen.	1 1 x x x x **0** X	1 1 x x x x **1** X
4	Die übergeordnete Steuerung erkennt an der Ungleichheit im Bit 1 des Handshake-Bytes zwischen Sende und Empfangsbuffers, dass neue Daten von netSCRIPT eingetroffen sind.	1 1 x x x x **0** X	1 1 x x x x **1** X
5	Sind die Daten in der übergeordnete Steuerung angekommen und aus dem Empfangspuffer abgeholt worden, wird dieses von der übergeordnete Steuerung durch setzen des Byte 1 in Ihrem Handshake-Sendebyte bestätigt.	1 1 x x x x **1** X	
6	Das Handshake-Sendebyte der übergeordneten Steuerung wird an netSCRIPT übertragen.	1 1 x x x x **1** X	
7	netSCRIPT erkennt, dass das Handshake-Bit 1 im Sendepuffer und Empfangspuffer den selben Wert hat. → Die übergeordnete Steuerung hat die zuvor gesendeten Daten in Empfang genommen und ist zum Empfang weiterer Daten bereit.	1 1 x x x x **1** X	1 1 x x x x **1** X
8	netSCRIPT setzt bei der Einstellung neuer Sendedaten, die Länge der Nutzdaten in den Header und das Handshake-Bit 1 in seinem Sendebuffer auf 0.		1 1 x x x x **0** X
9	Die Daten werden an die übergeordnete Steuerung übertragen.		1 1 x x x x **0** X
10	Solange eine Ungleichheit im Bit 1 des Handshakebytes zwischen dem Sende und Empfangsbuffer besteht, dürfen keine weiteren Daten zur Versendung bereit gestellt werden.	1 1 x x x x **1** X	1 1 x x x x **0** X
11	Die übergeordnete Steuerung erkennt an der Ungleichheit des Bit 1 in seinem Handshake-Sende-und Empfangs-Buffer, dass neue Daten von netSCRIPT eingetroffen sind.	1 1 x x x x **1** X	1 1 x x x x **0** X
12	Die übergeordnete Steuerung holt die neuen Daten aus dem Empfangspuffer und setzt zur Empfangsbestätigung das Bit 1 im Handshake-Byte seines Sendepuffers auf 0 (den selben Wert wie im Empfangspuffer)	1 1 x x x x **0** X	1 1 x x x x **0** X
13	Die Änderung im Handshake-Byte der übergeordnete Steuerung wird an netSCRIPT übertragen.	1 1 x x x x **0** X	1 1 x x x x **0** X
14	netSCRIPT erkennt am Wechsel des Bit 1 in Handshake-Byte in seinem Empfangsbuffer, dass die Daten von der übergeordnete Steuerung entgegengenommen wurden.		
15	Die Handshake-Bits 1 haben bei netSCRIPT im Sende und Empfangbuffer den selben Wert. Damit ist für netSCRIPT der Zustand wie im Schritt 0 gegeben und der Vorgang kann von vorne beginnen.	1 1 x x x x **0** X	1 1 x x x x **0** X

11 Error-Handling

11.1 über „lasterror“

Fehlerinformationen der Schnittstellenkommunikation (Port und Bus IO) sowie aus der Funktionsbibliothek „util“ werden in der Variablen **„lasterror“** hinterlegt.

Innerhalb netSCRIPTs erfolgt die Behandlung der Fehler rein symbolisch wie das folgende Beispiel zeigt. Zur Verwendung der Fehlercodes im Script ist diesen als Präfix ein „**err.**“ (ohne Anführungszeichen) voranzustellen.

```
if lasterror == err.BUSIO_SEND_NOT_READY then

…

end
```

oder

```
If uart:PortSend(“hello”) then

elseif lasterror == err.PORT_FIFO_FULL then

elseif lasterror == err.PORT_NO_BUFFER then

end
```

Werden die Fehlerwerte für andere Zwecke zum Beispiel außerhalb von netSCRIPT verwendet, so reflektieren die Werte einen 32Bit Wert.

Als 8 stellige Hexzahl ist deren Bedeutung in der folgenden Tabelle beschrieben. Darin enthaltende Fehlernummern ohne einen Eintrag in der Spalte Fehlercode, sind Abbruchfehler (Fehler die einen Start der Skriptverarbeitung verhindern), die in der SYCON.net Umgebung in der Diagnose als Fehlermeldung angezeigt werden.

Beginnt die Hexzahl mit 0xC, wird das Script in der Abarbeitung gestoppt (Ausnahme, der Aufruf erfolgte mit der Funktion **pcall**). Diese Meldungen sind in der folgenden Tabelle in der Spalte Stopp mit ▼ gekennzeichnet.

Beginnt der Fehlercode mit 0x4, wird die Scriptabarbeitung nicht angehalten. Der Fehler kann im Script weiter behandelt werden.

11.1.1 Fehlercodes in „lasterror“

in der Variablen **lasterror.**

Fehler-Nummer in Hex	Fehlercode	Stopp	Bedeutung
C0800002		▼	Speicherzuweisung in der netScript-Task ist fehlgeschlagen.
C0800080		▼	Lua-Start fehlgeschlagen.
C0800081		▼	Kein Skriptfile vorhanden. Skriptfile konnte nicht geladen werden. Skriptfile nicht als “Startup” markiert.
C0800082		▼	Wert von __CYCLIC_FUNCTION ist keine Funktion.
C0800083		▼	Es ist ein Fehler in einem Error Handler aufgetreten.
C0800084		▼	Eine Speicherzuweisung im Lua-Interpreter ist fehlgeschlagen.
C0800085		▼	Eine Lua-Panik ist aufgetreten. (Der Fehler ist wahrscheinlich außerhalb des Skriptes aufgetreten).
C0800101	err.LUA_ERROR	▼	Lua-Fehler (von Lua selbst ausgelöster Fehler).
C0800201	err.PORT_INVALID_CONFIG	▼	Fehlerhafte Konfigurations- Parameter.
C0800202	err.PORT_INVAL_PORT	▼	Falsche Port-Nummer, Port existiert nicht.
C0800203	err.PORT_WRONG_MODE	▼	Es wurde versucht einen Port in einem Modus anzusprechen, für den der Port nicht initialisiert wurde.
40800204	err.PORT_FIFO_EMPTY		Keine Eingangsdaten vorhanden.
40800205	err.PORT_FIFO_FULL		Nicht ausreichend freier FIFO-Speicher vorhanden.
C0800206	err.PORT_XC_INIT_FAILED	▼	Die Initialisierung ist fehlgeschlagen.
40800210	err.PORT_NO_BUFFER		Kein freier Speicherblock verfügbar (block mode).
40800211	err.PORT_NO_SUCH_PORT		Aufruf einer nicht existierenden Port-Instanz.
C0800213	err.PORT_NOT_OPEN	▼	Es wurde versucht auf einen nicht geöffneten Port zuzugreifen.
C0800214	err.PORT_NO_UARTDB	▼	Es wurde keine UART-Konfigurations-DB gefunden.
C0800215	err.PORT_PARSING_UARTDB	▼	Fehler beim interpretieren der UART-Konfigurations-DB.
40800216	err.PORT_INVALID_PARAMETER		Fehlerhafte Parameter bei Sende-/Empfangsaufruf.
40800217	err.PORT_NO_CONFIRMATION		Endekennung des Telegramms nicht gefunden.
40800218	err.PORT_STRING_TOO_LONG		Der zu übermittelnde String ist zu lang (für Sendepuffer).
40800212	err.PORT_ALREADY_OPEN		Es wurde versucht ein Port zu öffnen, der bereits geöffnet ist.
40800302	err.BUSIO_ALREADY_OPEN		Instanz wird schon benutzt.

Fehler-Nummer in Hex	Fehlercode	Stopp	Bedeutung
40800301	err.BUSIO_NO_SUCH_INSTANCE		Instanz existiert nicht.
C0800303	err.BUSIO_INVALID_CONFIG	▼	Fehlerhafte Konfigurationsdaten (tritt aktuell nicht auf).
C0800304	err.BUSIO_NOT_OPEN	▼	Instanz nicht geöffnet.
C0800305	err.BUSIO_INVALID_PARAMETER	▼	Falscher Parameter.
C0800306	err.BUSIO_WRONG_MODE		Eine DirektModus-Funktion wurde für eine im Handshakemode geöffnete Schnittstelle aufgerufen oder umgekehrt.
40800311	err.BUSIO_SEND_NOT_READY		Write: Eingabepuffer nicht frei.
40800312	err.BUSIO_SEND_DISABLED		Write: RxEnableCmd ist nicht gesetzt.
40800313	err.BUSIO_STRING_TOO_LONG		Write: String zu lang für Sendepuffer. Read: Ungültige Datenlänge im Header.
40800314	err.BUSIO_BUFFER_LENGTH_EXCEEDED		Die Funktionen :BusIOReadDirect oder :BusIOWriteDirect haben ungültige Lengen- oder Offset- Übergabe-Argumente.
40800321	err.BUSIO_RECEIVE_NO_DATA		Read: keine neuen Daten.
40800322	err.BUSIO_RECEIVE_DISABLED		Read: TxEnableCmd ist nicht gesetzt.
C0800401	err.UTIL_INVALID_PARAMETER	▼	Ungültiger Parameter bei Zieltyp, ENDIAN, LED oder Identifier.
40800402	err.UTIL_OUT_OF_RANGE		Zahl liegt nicht im Wertebereich des Zieltyps.
C0800411	err.UTIL_STRING_TOO_LONG	▼	String aus der SYCON- Variablenliste ist zu lang.
C0800410	err.UTIL_UNKNOWN_TYPE	▼	Es ist ein unbekannter Variablentyp in der SYCON- Variablenliste.
40800403	err.UTIL_WRONG_SIZE		Der Wert hat nicht die korrekte Größe für den angegebenen Typ.

Tabelle 41: netSCRIPT, „lasterror“ Fehlercodes

Die Fehler, die mit dem Zeichen ▼ gekennzeichnet sind (Fehlernummer C...), können nur im SYCON.net angezeigt werden. Siehe Abschnitt 12.1.2 Seite 125 . Die Abarbeitung des Skript wird angehalten.

Bei allen anderen Fehlerkennungen läuft die Abarbeitung des Skriptes weiter.

11.2 Rückgabewerte für Status und Error der Port-Funktionen

11.2.1 Rückgabewerte des Parameters “Status“:

Für die Funktionen: **PortIsReceiveDone** und **PortIsExchangeDone**

Status-Kode	Rückgabe wert	Bedeutung
port.STA_PATTERN_MATCH	1	Der Empfang wurde nach erreichen der der definierten Endekennung im Protokoll beendet.
port.STA_SIZE_REACHED	2	Der Empfang wurde nach erreichen der definierten Anzahl Empfangszeichen beendet.
port.STA_ACK_TIMEOUT	3	Der Empfang wurde nach erreichen der eingestellten ackdelaytime beendet. Es wurde kein Zeichen empfangen. Ein leerer String wurde übergeben.
port.STA_CHAR_TIMEOUT	4	Der Empfang wurde beendet, nachdem die definierte chardelaytime erreicht wurde.

Beispiel für eine Fehlerbehandlung:

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

11.2.2 Rückgabewerte des Parameters “Error”:

der Empfangsfunktionen

Error-Kode	Rückgabe wert	Bedeutung
port.ERR_BREAK_DETECTED	1	Ein Leitungsbruch wurde erkannt.
port.ERR_PARITY_ERROR	2	Ein Zeichen des Empfangsblocks hat einen Parity-Fehler.
port.ERR_FRAMING_ERROR	4	Ein Rahmenfehler (Fehlerhafte Start / Stopp- Kennung) ist aufgetreten.
port.ERR_RX_FIFO_OVERFLOW	8	Nur im Zeichenmodus. Der FIFO hatte ein Überlauf seit dem letzten Funktionsaufruf PortGetChar.

Nur der schwerwiegendste Fehler wird gemeldet:

Wird z.B. ein Rahmenfehler gemeldet, ist es möglich, dass auch einzelne Zeichen einen Parity-Fehler haben.
Wird ein Leitungsbruch gemeldet, ist es möglich, dass auch Parity-Fehler aufgetreten sind.
Wird ein FIFO- Überlauf gemeldet, können auch alle anderen Fehler aufgetreten sein.

12 Fehlersuche

Abhängig von der Art des Fehlers gibt es zwei Analysemöglichkeiten.

- Für Gerätefehler oder Kommunikationsfehler auf der Bus-Seite gibt es die Diagnose im Sycon.net.
- Für Skriptfehler gibt es des netSCRIPT Debugger.

12.1 Diagnose im SYCON.net

In der Regel wird mit der USB Diagnoseschnittstelle des Zielgerätes mit dem Windows Konfigurationswerkzeuges SYCON.net kommuniziert.

Ist die SYCON-Software über die Serviceschnittstelle an das Zielgerät angeschlossen, kann darüber ermittelt werden, in welchem Status sich die netSCRIPT-Funktion im Gerät befindet.

Diese Fehlerdiagnose ist nur möglich, wenn vom SYCON.net die logische Verbindung zum Gerät aufgebaut ist.

12.1.1 Aufruf der Diagnose

➢ Mit der rechten Maustaste auf das Symbol des Gerätes klicken, welches analysiert werden soll.

➢ Es öffnet sich folgendes Kontextmenü.

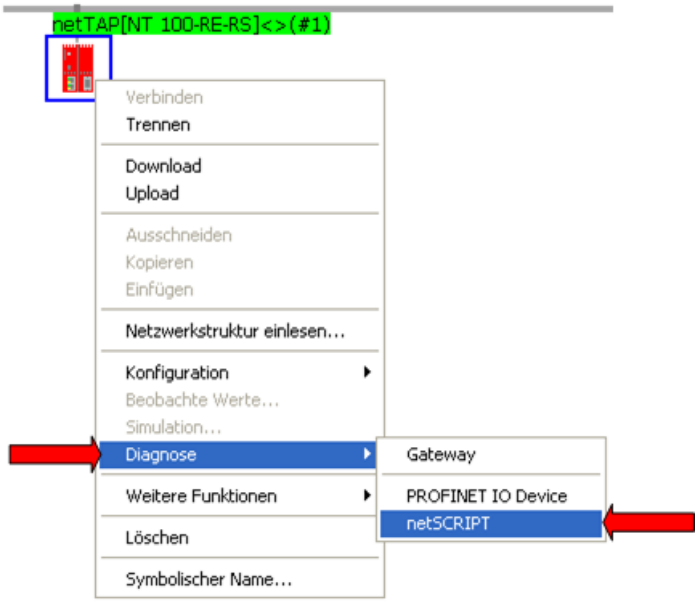


Abbildung 14: SYCON-Diagnose Auswahlpfad

➢ Wählen Sie “Diagnose > netSCRIPT” aus.

➢ Es öffnet sich das Fenster der Allgemeindiagnose.

12.1.2 Allgemeindiagnose - Stopp-Fehler im SYCON.net

Da bei einem Stopp-Fehler die Verarbeitung des Skriptes angehalten ist, ist keine Fehleranalyse über den Debugger möglich. Daher werden diese Fehler dann im SYCON angezeigt und können zur Fehleranalyse herangezogen werden.

- Unter dem Menüpfad des netSCRIPT-Gerätes (rechte Maustaste) “Diagnose > netSCRIPT > Allgemeindiagnose” erhalten Sie folgende Informationen:

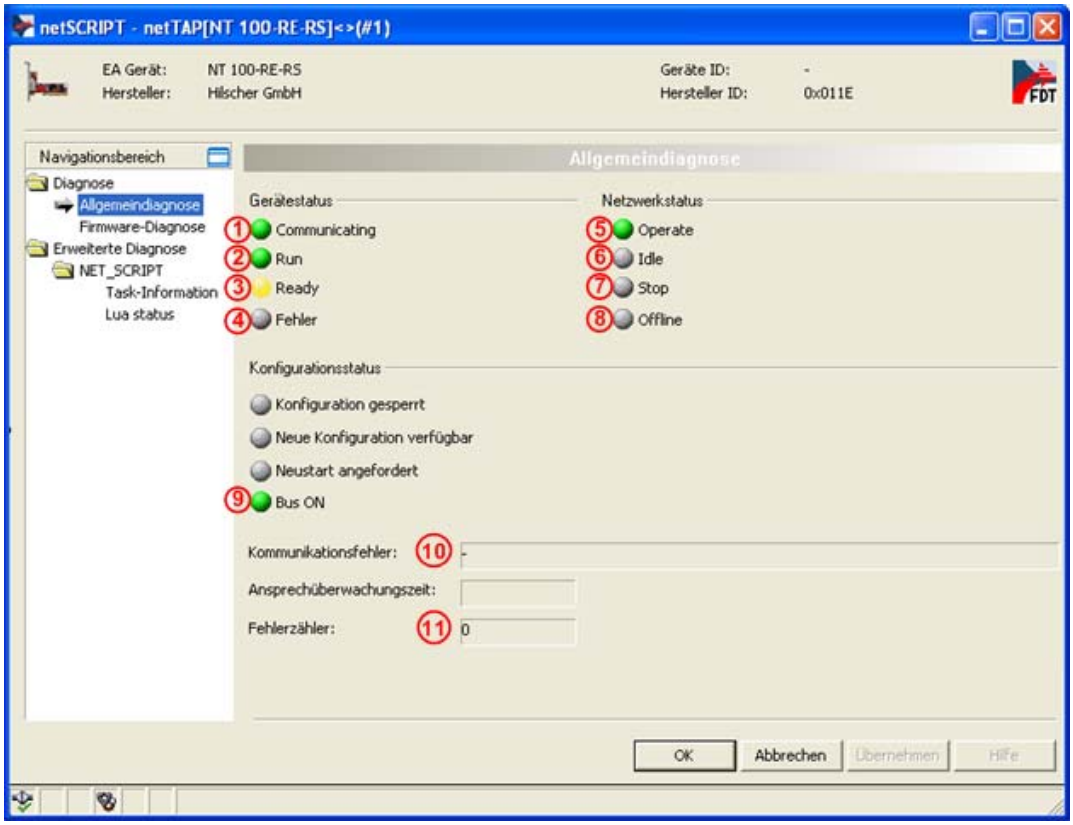


Abbildung 15: SYCON.net-Allgemeindiagnose

Dabei haben die Anzeigen folgende Bedeutung:

Anzeige	Farbe	Bedeutung
Gerätestatus		
Communication ①	grün	Skript läuft: Zeigt an, dass das Skript momentan ausführt wird.
	grau	Skript läuft nicht/angehalten
Run ②	grün	Skript geladen.
	grau	kein Skript geladen
Ready ③	gelb	netSCRIPT-Task erfolgreich initialisiert.
	grau	Initialisierungsfehler
Fehler ④	rot	Skript steht: Es ist ein Fehler aufgetreten, der die Skriptabarbeitung verhindert. Bei Kommunikationsfehler ist eine Fehlernummer und ein Fehlertext eingetragen.
	grau	Skript läuft: Kein Fehler seit dem letzten Skript-Neustart
Netzwerkstatus		
Operate ⑤	grün	In Betrieb: Zeigt an, dass das Skript zyklisch abgearbeitet wird.
	grau	Skript wird NICHT zyklisch abgearbeitet.
Idle ⑥	gelb	Skript angehalten: netSCRIPT wartet auf Befehle vom Debugger Ursachen: Nach Einzelschritt; Breakpoint erreicht; Skript neu geladen; Fehler.
	grau	netSCRIPT läuft NICHT im Debugmode.
Stopp ⑦	rot	Stopp : Skript wurde wegen eines Fehlers angehalten.
	grau	Skript ist nicht gestoppt.
Offline ⑧	gelb	Offline: kein Skript geladen, oder netSCRIPT-Initialisierungsfehler.
	grau	Skript NICHT offline.
Konfigurationsstatus		
Konfiguration gesperrt	gelb	Konfiguration gesperrt: NICHT genutzt
	grau	
Neue Konfigu-ration verfügbar	gelb	Neue Konfiguration verfügbar. NICHT genutzt
	grau	
Neustart angefordert	gelb	Neustart angefordert:. NICHT genutzt
	grau	
Bus ON ⑨	grün	Bus ON: Zeigt an, dass netSCRIPT mit der Mapping-Task (BUSIO-Schnittstelle) kommunizieren kann..
	grau	Kommunikation zur Mapping-Task ist nicht möglich.

Tabelle 42: Anzeigen Allgemeindiagnose

Parameter	Bedeutung
Kommunikationsfehler ⑩	Kommunikationsfehler: Letzter Fehler bei der Skriptausführung. Die Fehlermeldung wird beim Neuladen des Skripts (auch über den Debugger) gelöscht..
Ansprechüber-wachungszeit	Ansprechüberwachungszeit: NICHT verwendet
Fehlerzähler ⑪	Fehlerzähler: Zeigt die Gesamtzahl der Fehler an, die seit dem Gerätestart bzw. nach einem Geräte-Reset aufgetreten sind. Darin sind alle Fehler enthalten, egal ob es sich um Netzwerkfehler oder um geräteinterne Fehler handelt. Dieser Fehlerzähler wird nicht beim Neuladen des Skriptes (über den Debugger) gelöscht.

Tabelle 43: Parameter Allgemeindiagnose

12.1.3 Firmware-Diagnose

Im Dialog **Firmware-Diagnose** werden die aktuellen Task-Information der Firmware angezeigt.

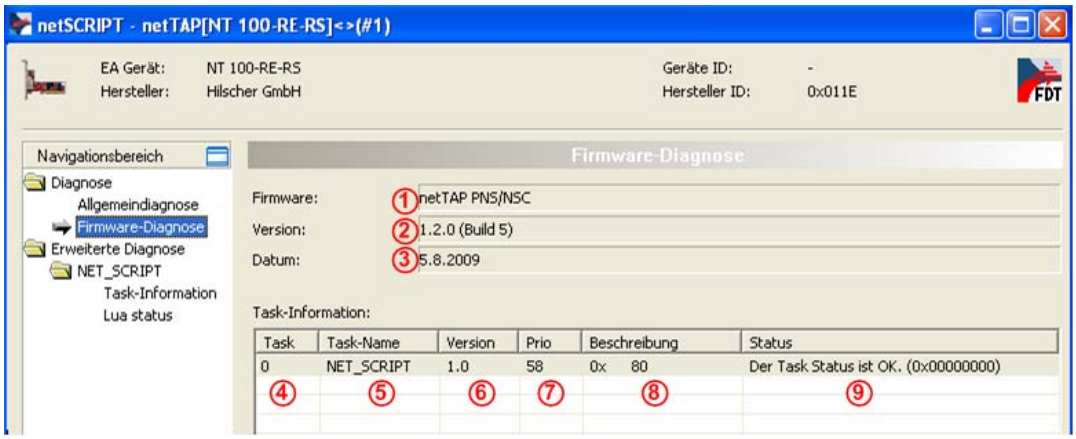


Abbildung 16: SYCON- Diagnosefenster „Firmeware-Diagnose“

① Der Name der geladenen Firmware.

② Die Versionsnummer der geladenen Firmware.

③ Das Datum der Firmwareerstellung

Task Information:

Die Tabelle **Task Information** listet die Task-Information der einzelnen Firmware-Tasks auf.

Nr.	Spalte	Bedeutung
④	Task	Nummer der Task
⑤	Task-Name	Name der Task
⑥	Version	Version der Task
⑦	Priorität	Priorität der Task
⑧	Beschreibung	Beschreibung der Task
⑨	Status	Status der Task

Tabelle 44: Beschreibung Tabelle Task Information

12.1.4 Task-Informationen

In diesem Fenster werden Informationen zur netSCRIPT-Task im Gerät angezeigt.

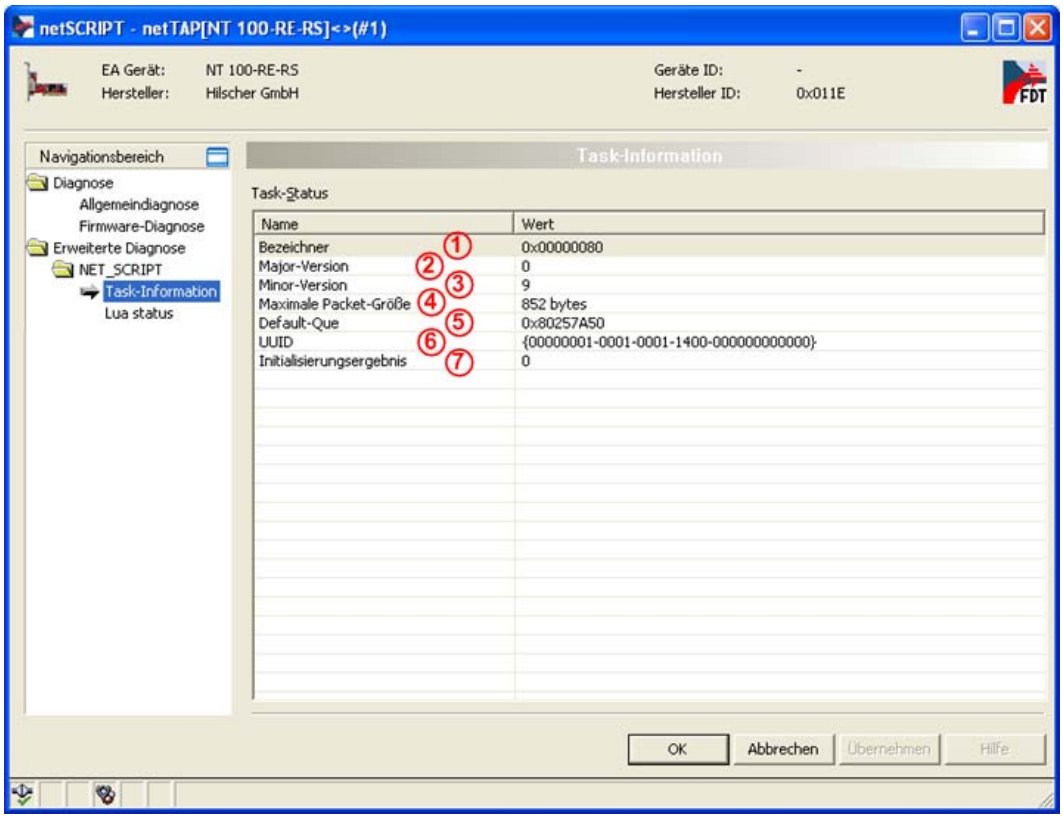


Abbildung 17: SYCON- Diagnosefenster „Task- Information“

In diesem Fenster werden folgende Daten angezeigt:

① Bezeichner: Die Tasknummer der netSCRIPT Task.

② Interne Versionsnummer.

③ Interne Versionsnummer.

④ Interne Paketgröße zur externen Kommunikation.

⑤ Adresse der Diagnose Paketschlange

⑥ Interne Versionsnummer.

⑦ Information über den Initialisierungszustand der netSCRIPT-Funktion. (sollte immer 0 sein).

12.1.5 Lua-Status

Unter „Lua status“ sind folgende Informationen zu finden:

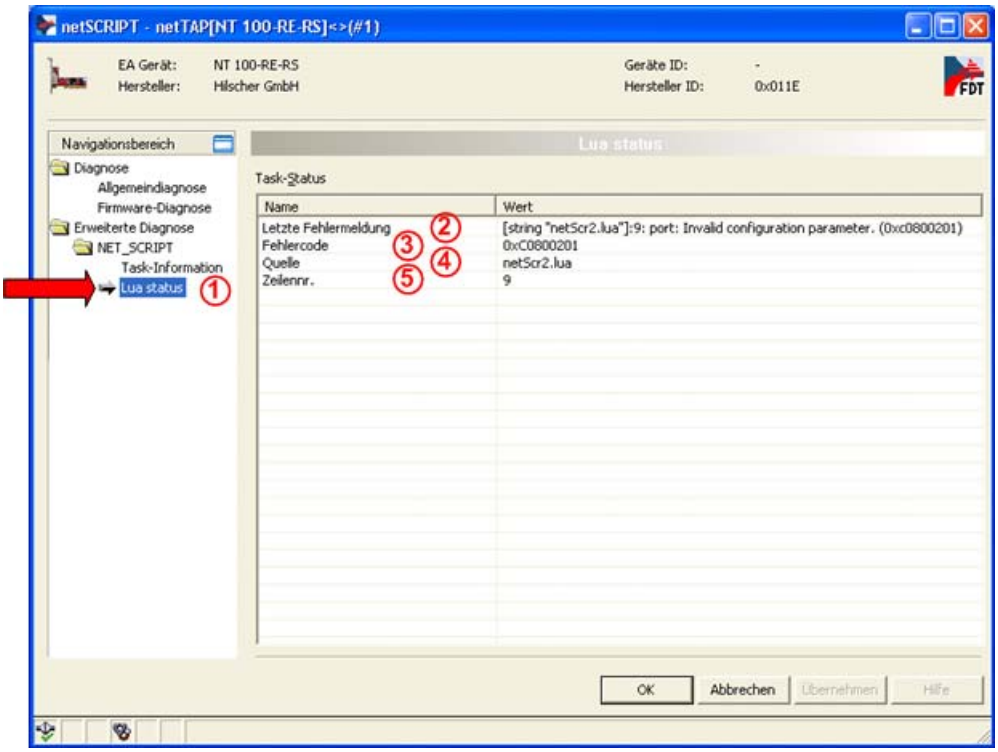


Abbildung 18: SYCON- Diagnosefenster , Lua status

① Der Diagnose Aufrufpfad.

② Hier finden Sie den Fehlertext und die Fehlernummer angezeigt.

③ In dieser Zeile finden Sie den Fehlercode.

④ Hier wird angezeigt, in welcher Skript-Datei der Fehler entstanden ist.

⑤ Hier finden Sie die Zeilennummer des Codes in der Skriptdatei.

13 Fehlersuche netSCRIPT

Für die Fehlersuche in Abläufen eines Scriptprogramms gibt es einen netSCRIPT-Debugger. Der Debugger ist ein eigenständig und unabhängig von SYCON.net lauffähiges Programm unter Windows und ein sogenannter Software-Debugger.

Ein Zusatztreiber in der geladenen Gerätesoftware des netSCRIPT fähigen Gerätes kommuniziert mit dem Debugger. Dort prüft der Treiber in der zyklischen netSCRIPT-Abarbeitungsschleife automatisch auf Kommandos vom Debugger. Die Aufträge wie zum Beispiel „Unterbrechen“ oder „Start“ werden dann vom Treiber in den Programmfluss eingearbeitet und ermöglichen den Ablauf des Programms zu kontrollieren.

Hinweis: Die einwandfreie Funktionsweise des Debuggers hat zur Vorraussetzung, dass das zu debuggende netSCRIPT-Programm stets seine zyklische Abarbeitung durchführt, ansonsten ist der Debugger nicht in der Lage mit dem Debug-Treiber der Gerätesoftware zu kommunizieren und das ausgeführte Programm zu debuggen.

Sollte die Script-Abarbeitung wegen eines schweren Fehlers gestoppt worden sein, ist u. U. nur noch eine Fehleranalyse über die SYCON.net-Kofnigurationssoftware möglich.

13.1 netSCRIPT Debugger

Der Debugger dient zur Überprüfung der richtigen Funktionsweise der geladenen Scriptdatei im Zielgerät.

Er ermöglicht es den Code zeilenweise zu Debuggen und den Inhalt der Variablen in jedem Programmschritt anzuzeigen. Hierzu ermöglicht er die Quelldatei zu verändern und auch wieder an das Zielgerät zurück zu übertragen.

Der Debugger ist in der Lage dass aktuell laufende Script ohne ein Projekt zu öffnen oder zu erstellen aus dem Zielgerät hoch zu laden und anzuzeigen. Zum Ändern des Scriptes ist es jedoch erforderlich ein Projekt geöffnet zu haben.

Es besteht die Möglichkeit Breakpoints zu setzen.

Hinweis: Zur besseren Übersichtlichkeit sind in den folgenden Grafiken die einzelnen Bereiche des Debuggerfensters mit Buchstaben und die Schaltflächen mit Ziffern gekennzeichnet.

13.1.1 Installation

Für die Nutzung des Debuggers ist Voraussetzung, dass auch die Software SYCON.net und ein Kommunikationstreiber z.B. USB oder TCP/IP für das Zielgerät auf der Windows-Plattform installiert ist.

Wie der Debugger zu installieren ist, ist in dem Dokument „SYCON.net Installation - Gateway Solution UM 01 DE.doc“ unter dem Verzeichnis „Documentation“ auf der Produkt DVD nachzulesen.

13.1.2 Start des Debuggers

Der Debugger wird auf dem PC über den Dialogpfad „Start → alle Programme → Hilscher GmbH → netSCRIPT_Debugger → netSCRIPT_Debugger“ oder über das Desktop-Icon

gestartet.

➪ Er Startet mit dem folgendem Bild:

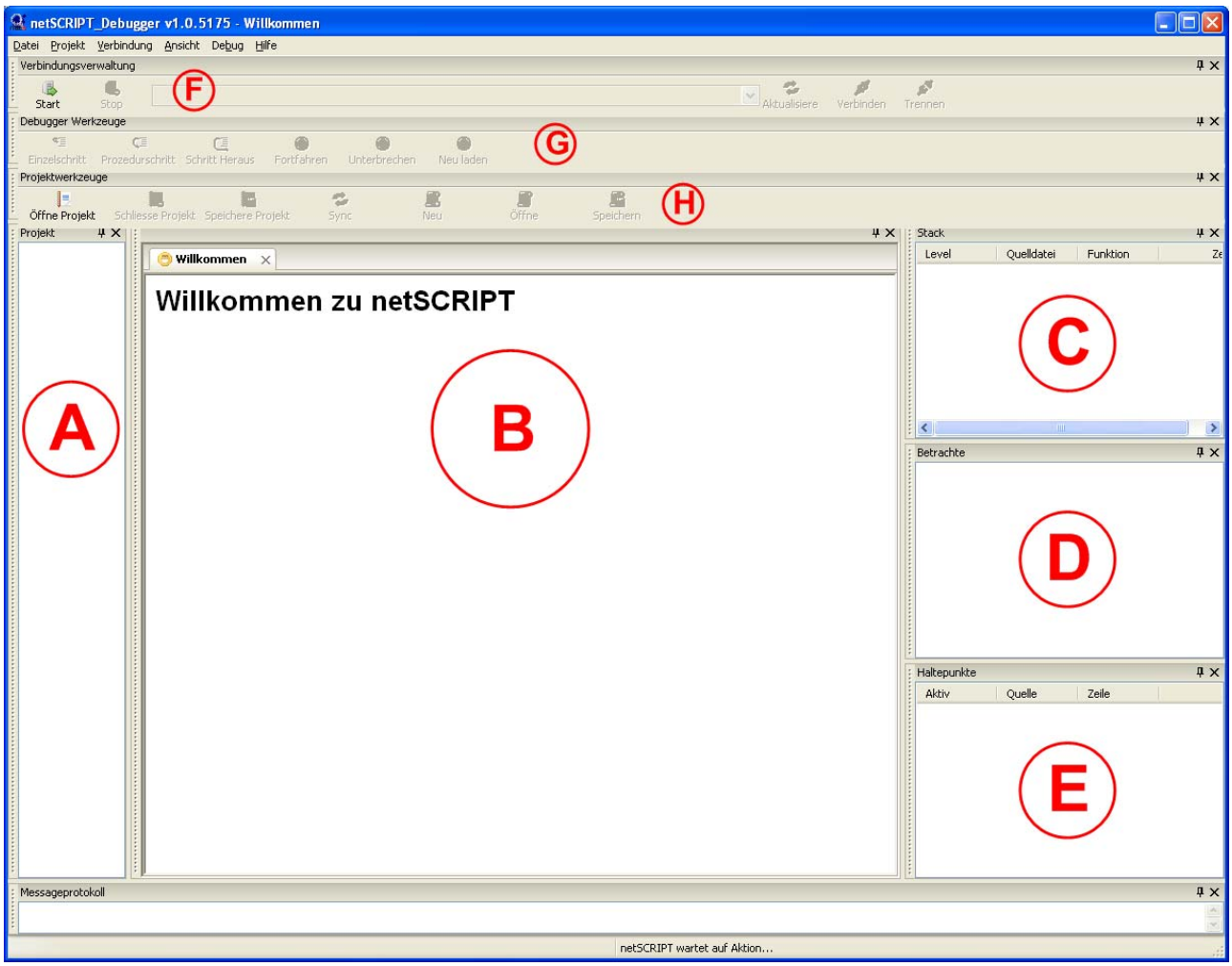


Abbildung 19: netSCRIPT Debugger Fensterbereiche

Der Debugger besteht im Wesentlichen aus 8 Bereichen:

Ⓐ Dem Bereich der Projektauswahl.

Ⓑ Dem netSCRIPT Anzeige- und Editierbereich.

Ⓒ Dem Fenster, in dem die Verschachtelungstiefe des Scriptes angezeigt wird, in dem sich das Script aktuell befindet. Level 0 ist dabei immer das Hauptprogramm.

Ⓓ Dem Fenster, in dem im Debugg-Modus alle Variablen und deren aktuellen Inhalte angezeigt werden.

Ⓔ Dem Fenster in dem die Position der gesetzten Haltepunkte angezeigt wird.

Ⓕ Die Leiste mit den Werkzeugen zum Verbindungsaufbau zum netSCRIPT-Gerät.

Ⓖ Die Leiste der Debug-Werkzeuge.

Ⓗ Die Leiste der Projekt-Werkzeuge.

13.1.3 Verbindungsaufbau zum netTAP-Gerät

Zunächst ist die Hardware-Verbindung zum netTAP-Gerät herzustellen. In der Regel wird diese eine USB-Verbindung sein. USB Verbindungen werden auf serielle COM-Schnittstellen des PCs abgebildet und erscheinen z.B. als „COM1_Ch1“,... „COM4_Ch1“ in Verbindungsauswahlmenü.

➢ Um die Verbindung vom Debugger zum netTAP-Gerät aufzubauen, ist im Fensterbereich Ⓕ, im Bereich der Verbindungsverwaltung, die Schaltfläche ① zu betätigen.

Abbildung 20: netSCRIPT Debugger Start Verbindungsaufbau

- Daraufhin wird ein Scan über die möglichen Verbindungen seitens des PCs und der gefundenen netSCRIPT fähigen Geräte durchgeführt.
- Nach erfolgreichem Scan wird in der Verbindungsverwaltung folgendes sichtbar:

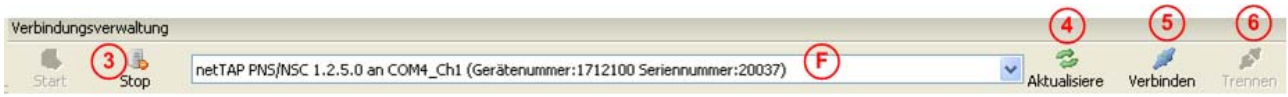


Abbildung 21: netSCRIPT Debugger Verbnindungsverwaltung

- Es werden die Bedienflächen ③, ④ und ⑤ anwählbar. In der Zeile Ⓕ wird die gefundene Verbindung (mit Gerätetyp, Geräteversion, Kommunikationskanal, Gerätetypnummer und Geräte-Seriennummer) angezeigt.

  Mit der Schaltfläche ③ kann der Verbindungskanal wieder geschlossen werden.

  Mit der Schaltfläche ④ kann eine neue Verbindungssuche gestartet werden.

  Mit der Schaltfläche ⑤ kann die logische Verbindung zur netSCRIPT-Task im Gerät hergestellt werden. Dabei wird in der Zeile der Debugger Werkzeuge die Schaltfläche ⑪ anwählbar.

13.1.4 Aktuelles Script aus netTAP laden

➢ Nach dem Verbindungsaufbau zum Zielgerät ist die Schaltfläche (11) (Unterbrechen) anzuwählen.

Hinweis: Das Script wird dabei in der Abarbeitung unterbrochen!

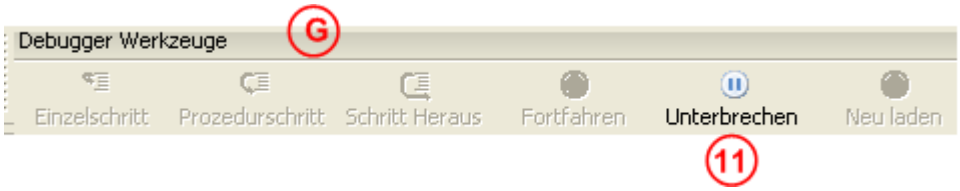


Abbildung 22: netSCRIPT Debugger Werkzeuge, Fensterbereich H

➢ Es wird das aktuell im Zielgerät geladene netSCRIPT-Programm in den Debugger geladen und angezeigt.

Es kann jetzt die gesamte Debug-Funktionalität genutzt werden. Siehe Abschnitt 13.1.7, Seite 136.

Hinweis: Das aus dem Gerät in den Debugger geladene Skript kann jedoch nur geändert werden, wenn gleichzeitig das dazugehörige Projekt geöffnet ist.

13.1.5 Projekt öffnen

➢ Im Fensterbereich (H) ist die Schaltfläche (13) anzuwählen.

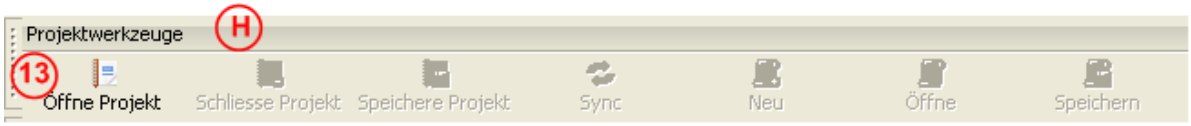


Abbildung 23: netSCRIPT Debugger, Projektwerkzeuge, Fensterbereich H

➢ Es öffnet sich das Dateiverwaltungsfenster des Betriebssystems. Hier ist das zugehörige Projekt mit der Dateiendung .nxspr auszuwählen.

Nach der Auswahl des Projektes erscheint dieses im Fensterbereich (A)

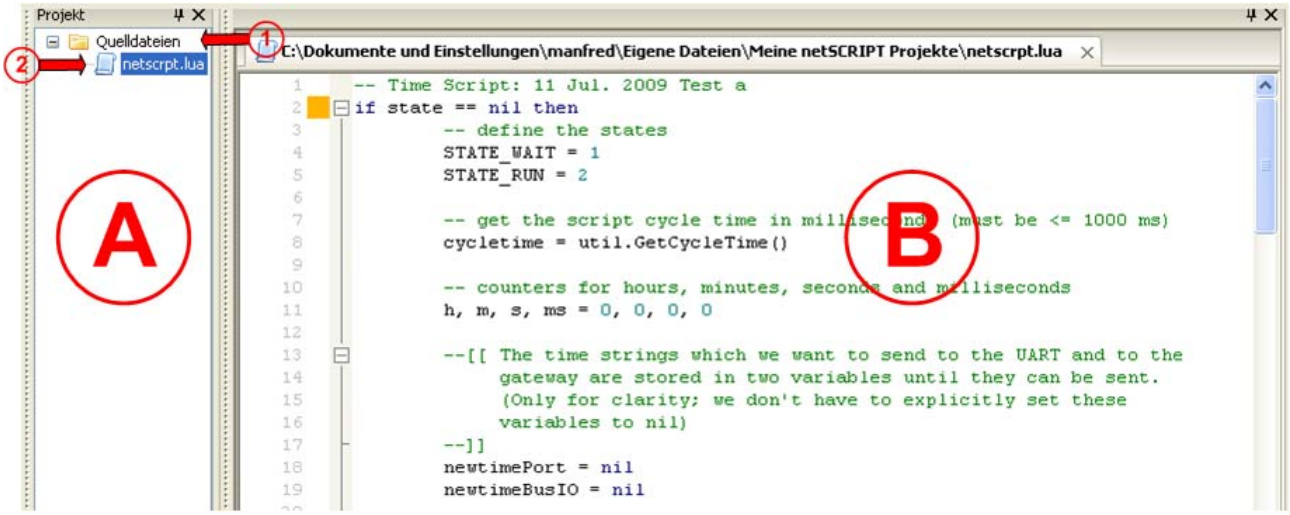


Abbildung 24: netSCRIPT Debugger Fensterbereich A und B

- Es ist ein Doppelklick auf die Zeile (1) zu machen, wodurch sich der darunterliegende Dateibaum öffnet.
- Nach einem Doppelklick auf die zu öffnende Datei erscheint deren Inhalt im Fensterbereich (B).

Enthält das Projekt nur eine Skriptdatei, wird diese unmittelbar im Fensterbereich (B) angezeigt.

Wird die Datei editiert, wird das Symbol vor dem Dateipfad im Kopfbereich des Fensters (B) rot, was auf die Diskrepanz zwischen der angezeigten Skriptversion und der gespeicherten Projektversion hinweist.

Abbildung 25: netSCRIPT Debugger - geänderte Skriptdatei.

Geänderte Skriptdateien können nur ins netSCRIPT-Gerät geladen werden, wenn das Projekt zuvor gespeichert wurde.

- Speichern Sie das Projekt mit der Schaltfläche (19) der Projektwerkzeuge.

Abbildung 26: netSCRIPT Debugger, Projektwerkzeuge, Speichern, ´Laden.

13.1.6 Script ins netTAP laden und starten

- Mit einem Mauseklick auf die Schaltfläche ⑯ der Projektwerkzeuge wird die aktuell im Fenster Ⓑ angezeigte netSCRIPT-Datei ins Gerät geladen.
- Sollte das aktuell im Gerät befindliche Skript noch in der Ausführung sein, ist dieses mit der Schaltfläche ⑪ der Debugger Werkzeuge zu unterbrechen.

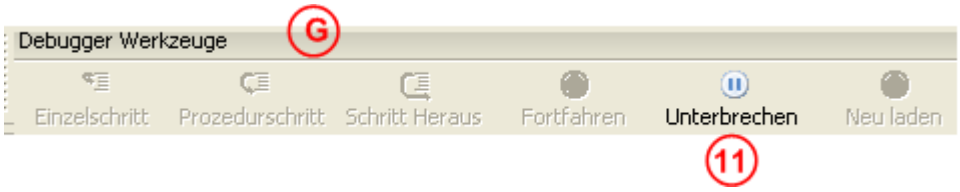


Abbildung 27: netSCRIPT Debugger Werkzeuge, Fensterbereich H

Dadurch werden die folgenden Schaltflächen zur Bedienung freigegeben:

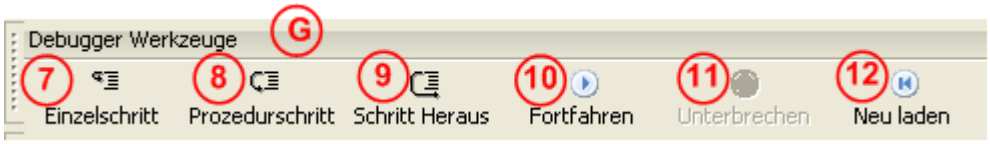


Abbildung 28: netSCRIPT Debugger Fensterbereich H

- Zum Start des neu herunter geladenen Skriptprogramms ist im Fensterbereich Ⓗ ein Mausklick auf die Schaltfläche ⑫ erforderlich, wodurch das neue Skript zu Ausführung kompiliert wird.
- Zum Start der Ausführung des neuen Skriptes ist dann die Schaltfläche ⑩ zu betätigen.

13.1.7 Skript debuggen

Es kann eine vom Zielgerät zurückgeladene Skriptdatei als auch aus einem Projekt geladene und ins Gerät übertragene Skript gedebugt werden. Soll der Code im Debugger editiert werden können, ist das Vorhandensein einer Projektdatei erforderlich.

Die Bedienflächen des Debugger Tools (im Fensterbereich Ⓖ)sind:

Abbildung 29: netSCRIPT Debugger Fensterbereich H

⑦ Einzelschritt:

Es wird die Scriptzeile ausgeführt, vor der der Courser  steht. Ist dieses ein Funktionsaufruf einer in netSCRIPT definierten Funktion, geht der Debugger in die erste Zeile dieser Funktion.

⑧ Prozedurschritt:

Der anstehende Funktionsblock wird automatisch ausgeführt. der Debugger bleibt in der nächsten folgenden Zeile nach dem Funktionsblock stehen.

⑨ Schritt Heraus:

Wurde in einem Funktionsblock mit „Einzelschritt“ hineingegangen, wird die weitere Abarbeitung des Codes automatisch durchlaufen. Der Debugger bleibt in der ersten Zeile nach der Rückkehr aus dem Funktionsaufruf stehen.

⑩ Fortfahren:

Der Debug-Modus wird verlassen, die zyklische Verarbeitung des Scriptes wird fortgeführt.

⑪ Unterbrechen:

Die zyklische Verarbeitung des Scriptes wird zum nächsten zyklischen Start gestoppt und in den Debug-Modus gewechselt.

Hinweis: Befindet sich das Skriptprogramm in einer Endlosschleife, kann über dieses Kommando die Scriptverarbeitung nicht unterbrochen werden. In solch einem Fall muss eine veränderte Skriptdatei ins Zielgerät geladen werden, welche keine Endlosschleife mehr enthält. Danach ist die Spannungsversorgung des Gerätes einmal zu unterbrechen, damit es bei Spannungswiederkehr mit der neuen Skriptdatei startet.

⑫ Neu laden:

Die zuvor mit der Schaltfläche ⑯ ins Zielgerät geladene Skriptdatei wird zur zyklischen Ausführung gestartet.

13.1.7.1 Beobachtungs-Fenster

In diesem Fensterbereich Ⓓ werden im Debug-Modus (Einzelschritt) alle in netSCRIPT und Lua definierten Variablen, Funktionen und Tabellen mit den aktuellen Inhalten angezeigt. Zur besseren Identifikation ist den Einträgen ein signifikantes Icon vorangestellt.

Funktionsaufruf mit Name und Adresse.

String-Variable mit Namen und Inhalt.

Numerische Variable mit Namen und Wert.

Boolsche Variable mit Namen und Zustand

Tabelle, mit einem Klick auf das vorangestellte „+“ Zeichen kann die Tabelle mit ihrem Inhalt aufgeblättert werden.

Instanz mit Namen

13.1.7.2 Haltepunkte setzen

Es können max. 32 Haltepunkte gesetzt werden.

Achten Sie darauf, dass die Haltepunkte in Bereiche gesetzt werden, die während der Skriptabarbeitung aktuell auch durchlaufen werden.

➢ Zum Setzen eines Haltepunktes klicken Sie mit der linken Maustaste rechts neben die Zeilenzahl. Siehe Position ① im folgendem Bild.

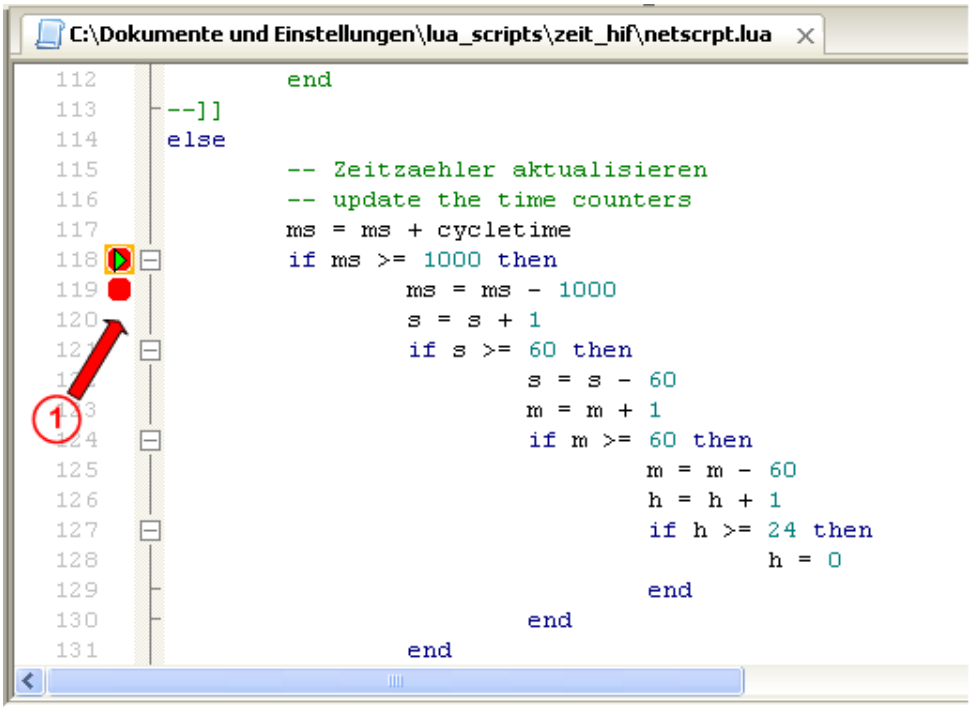


Abbildung 30: netSCRIPT Debugger Breakpoint setzen.

- Es erscheint ein rotes Rechteck an dieser Position. Gleichzeitig wird dieser Breakpoint in einer Liste im Haltepunktfenster Ⓔ aufgenommen.
- Wird die Skriptverarbeitung wieder aufgenommen, wird die Abarbeitung am gesetzten Breakpoint (vor Ausführung der Zeile) angehalten. Die Position wird mit einem grünen Pfeil gekennzeichnet.

Gesetzte Haltepunkte werden durch erneutes Anklicken mit der linken Maustaste wieder gelöscht.

13.1.8 Skript editieren

Zum Editieren eines Skriptes ist es erforderlich:

- ein Projekt im Debugger geladen zu haben,

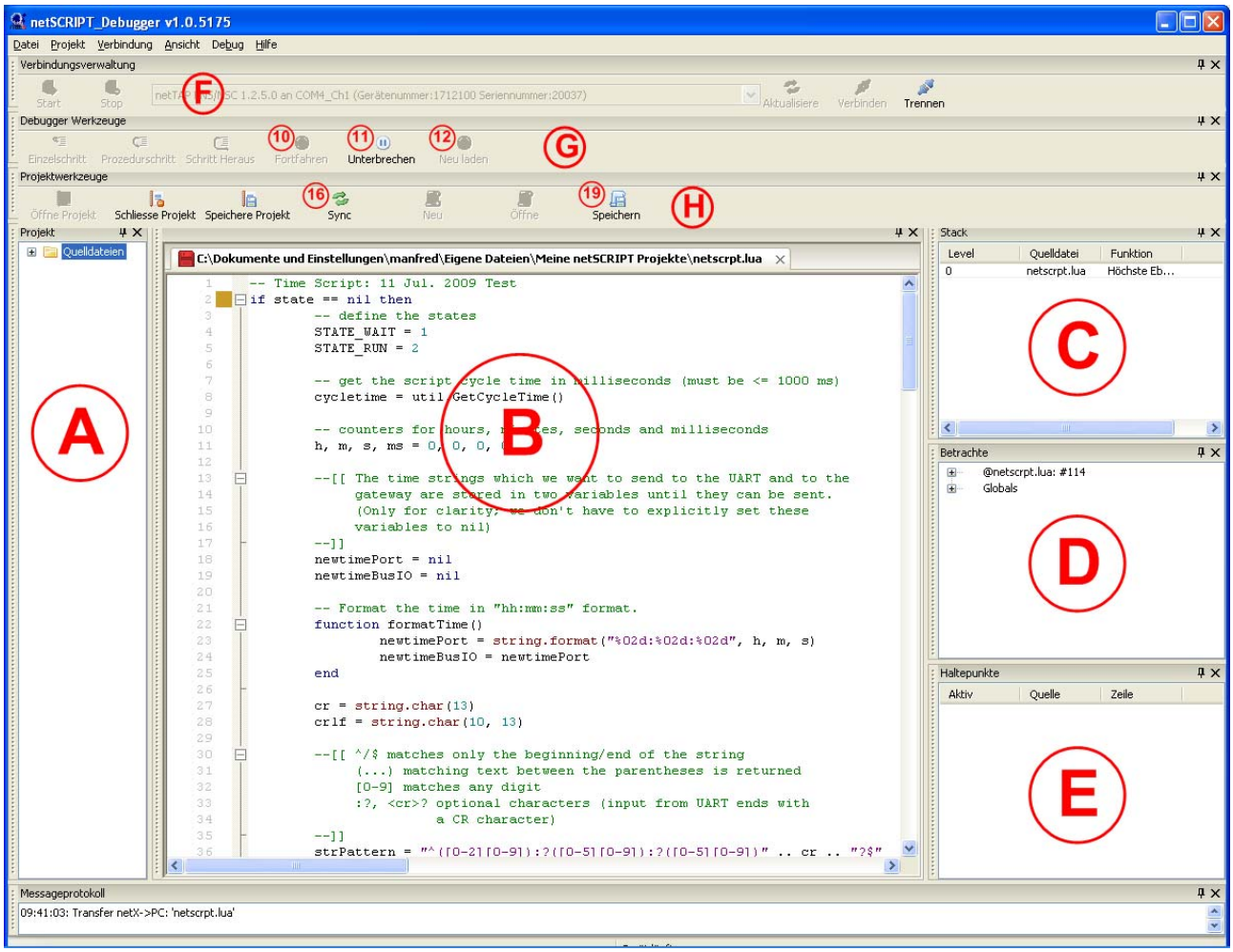


Abbildung 31: netSCRIPT Debugger.Skript editieren

- Jetzt ist im Fensterbereich (B) das Skript editierbar.
- Nach dem Editieren muss das Skript mit der Schaltfläche (19) gespeichert werden, bevor es ins Zielgerät geladen werden kann. Es wird nicht das Script aus dem Editor-Fenster ins Zielgerät geladen, sondern die gespeicherte Skriptdatei.
- Nach dem Speichern ist die Schaltfläche (16) Sync anzuklicken, wodurch das Skript in das Zielgerät übertragen wird. In diesem Zustand wird es jedoch noch nicht im Zielgerät ausgeführt.

- Es ist die Abarbeitung des im Zielgerät derzeit ausgeführten Skriptes durch anklicken der Schaltfläche (11) Unterbrechen zu beenden. Hierdurch werden die übrigen Schaltflächen der Bedienzeile (G) Debugger Werkzeuge anwählbar.

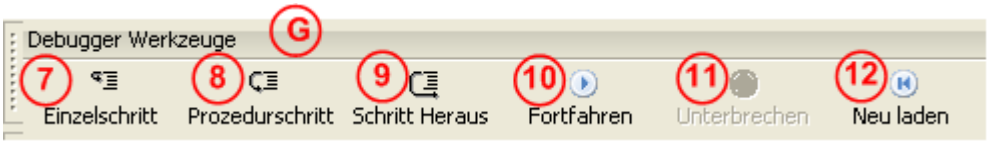


Abbildung 32: netSCRIPT Debugger Werkzeuge

- Mit betätigen der Schaltfläche (12) wird die neue Skriptdatei zur Ausführung geladen. Anschließend kann das neue Skript durch betätigen der Schaltfläche (10) Fortfahren gestartet werden.

Hinweis: Der letzte Download eines netSCRIPT-Files in das Zielgerät sollte immer aus dem Editor im SYCON.net erfolgen, damit vermeiden Sie Inkonsistenzen in der Projektverwaltung.

13.1.9 Debugger beenden

Beim beenden des Debuggens achten Sie bitte darauf, in welchem zustand des Debug-Mode Sie den Debugger verlassen.

13.1.9.1 Debugger beenden, mit unterbrochener Skriptabarbeitung

→ Das Skript bleibt auch nach Beendigung des Debuggers und Trennen der USB-Verbindung im netTAP-Gerät gestoppt.

→ Das Skript läuft nach einer Betriebsspannungsunterbrechung im netTAP-Gerät wieder an.

13.1.9.2 Debugger beenden bei laufender Skriptabarbeitung

→ Das Skript läuft auch nach beenden des Debuggers und unterbrechen der USB-Verbindung im netTAP Gerät weiter.

13.1.9.3 Debug-Verbindung trennen

- Kontrollieren Sie, ob das Skript läuft. Dieses erkennen Sie an der Anwählbarkeit der Schaltflächen in der Debugger Werkzeugleiste. Hier muss, wie im nachfolgenden Bild dargestellt, die Schaltfläche „Unterbrechen“ anwählbar sein.

Abbildung 33: Werkzeugleiste Unterbrechen

Und in der Fußzeile des Debuggers. Hier muss mittig der Text „Skript läuft“ angezeigt werden.

- Betätigen Sie in der Bedienerzeile „Verbindungsverwaltung“ die Schaltfläche ⑥ „Trennen“.

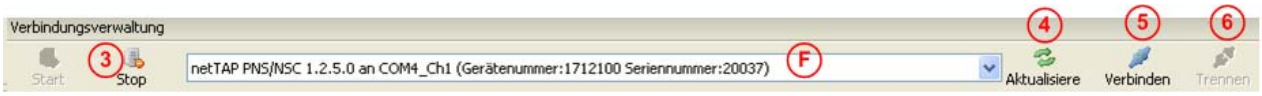


Abbildung 34: Debug-Verbindungsverwaltung

- Betätigen Sie in der Bedienerzeile “Verbindungsverwaltung” die Schaltfläche ③ „Stop“, damit auch der Debugger die Verbindung zur USB-Schnittstellenverwaltung beendet.
- Schließen ‚Sie das Debugger-Programmfenster.

14 Einfache netSCRIPT Beispielanwendung

Für alle hier vorgestellten Skripts, wird zur Ausführung ein netSCRIPT fähiges Gerät mit einer seriellen Datenschnittstelle und ein PC zur Konfiguration benötigt. Auf dem PC muss eine Hyperterminalemulation vorhanden sein.

14.1 Beispielprogramm: Echo

Dieses Skript echot ein Zeichen auf der RS 232 Schnittstelle im Blockmodus.

Diese Skript ist auf den CD unter
“Examples\netSCRIPT\Echo\netscrpt.lua“ zu finden.

Sie können das Skripts direkt mit dem netSCRIPT Debugger in das Gerät zur Ausführung laden.

14.2 Beispielprogramm: Blockmode

In diesem Skript wird die Initialisierung der RS-Schnittstelle im Blockmodus gezeigt. Das Skript echot das über die RS-Schnittstelle eingehende Zeichen und zählt die Zyklen.

Diese Skript ist auf den CD unter
“Examples\netSCRIPT\Serial Port Blockmode\blkmode.lua” zu finden.

Sie können das Skripts direkt mit dem netSCRIPT Debugger in das Gerät zur Ausführung laden.

14.3 Beispielprogramm: Eliza

Ist ein Script im FIFO-Mode (Charmode) der RS-Schnittstelle. Es realisiert über das Hyperterminal eine Schlüsselwort-Kommunikation nach Joseph Weizenbaum's classic Eliza in englischer Sprache.

Diese Skript ist auf den CD unter
“Examples\netSCRIPT\Eliza\eliza.lua” zu finden.

Sie können das Skript direkt mit dem netSCRIPT Debugger in das Gerät zur Ausführung laden.

14.4 Beispielprogramm: BusIOCount

Mit diesem Skript wird zyklisch der Errorcode im Statusbyte 8-11 der BusIO-Schnittstelle inkrementiert.

Diese Skript ist auf den CD unter
“Examples\netSCRIPT\BusIOCount\busiocnt.lua“ zu finden.

Zum Auslesen des Zählers wird eine übergeordnete Steuerung mit der Busschnittstelle und dem verwendeten Netzwerkprotokoll benötigt.

Das Skript kann wegen der erforderlichen Signalzuordnung für die Bus-Kommunikation nur mit SYCON.net ins Gerät geladen werden. Die Signalzuordnung im Gateway ist vom Anwender vorzunehmen.

14.5 Beispielprogramm: hello_World

Öffnet die Serielle Schnittstelle mit spezifischen Parametereinstellungen, sendet einen Datenstring über diese Schnittstelle und schließt anschließend die Schnittstelle.

Dieses Skript ist auf den CD unter
„Examples\netSCRIPT\Hello World\hello.lua“ zu finden.

14.6 Beispielprogramm: LedFlash

Es wird die „Run Led“ (in Abhängigkeit der Zykluszeit) zyklisch angesteuert.

Ein Zähler wird mit jedem zyklischen Start des Skriptes um 1 erhöht. Zur Untersetzung der Zykluszeit wird die Funktion Modular verwendet.

Diese Skript ist auf den CD unter
“Examples\netSCRIPT\ LED Flash\ledflash.lua“ zu finden.

Es wird kein Hyperterminal benötigt.

14.7 Beispielprogramm: Time

Dieses Programm realisiert eine Uhr. Deren Startzeit über die UART-Schnittstelle und / oder über die Bus IO-Schnittstelle eingestellt werden kann.

Dieses Skript ist auf den CD unter dem Verzeichnis
“Examples\netSCRIPT\Time\time.lua“ zu finden.

Diese Anwendung ist für den CHAR-Mode der seriellen Schnittstelle prädestiniert, da gleichzeitig Zeichen versendet und empfangen werden können. Der Empfang der Zeichen für das stellen der Uhr kann gleichzeitig zur Urzeitausgabe erfolgen.

Zur Ausführung wird ein netSCRIPT fähiges Gerät mit einer seriellen Datenschnittstelle und einer übergeordneten Bus-Schnittstelle benötigt. Beispielhaft wurde das netTAP100 der Firma Hilscher eingesetzt. Zur Konfiguration des Gerätes wird ein PC benötigt. Auf dem PC muss eine Hyperterminalemulation vorhanden sein.

Das Skript kann alleine über die serielle Schnittstelle bedient werden und ist ohne übergeordnete Steuerung lauffähig. Es bedient jedoch parallel zur seriellen Schnittstelle auch die Schnittstelle zur übergeordneten Steuerung. Um alle Möglichkeiten dieses Beispiels nutzen zu können, kann der folgende Hardwareaufbau realisiert werden:

14.7.1 Aufbau

14.7.1.1 Hardwareaufbau

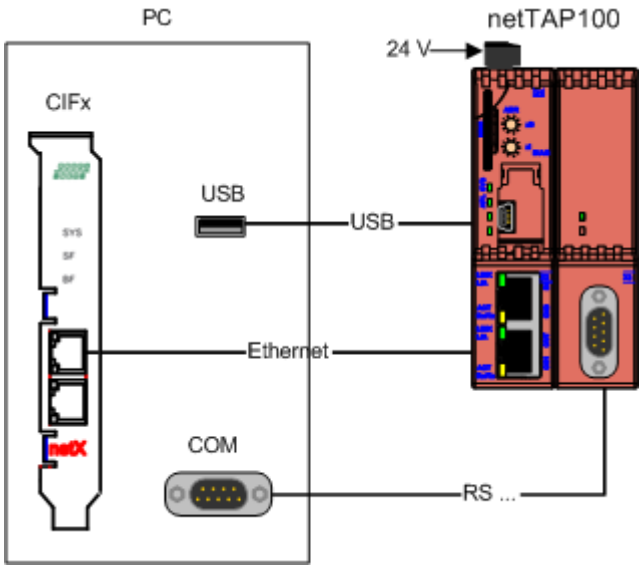


Abbildung 35: Beispiel Skript Time - Hardwareaufbau

Um nur die Kommunikation der seriellen Schnittstelle und das Laden des netSCRIPT Programms in das Zielgerät netTAP 100 über die USB-Schnittstelle zu testen, ist kein übergeordneter Bus-Master (cifX-Karte in einem PC) erforderlich. Als Bus-Master kann jede beliebige Steuerung verwendet werden, die über PROFINET kommunizieren kann.

Mit einem USB Kabel ist zwischen der USB-Schnittstelle des netTAP 100 Gerätes und einer USB-Schnittstelle des PC's eine Verbindung herzustellen.

Mit der Software SYCON.net sind eine Firmware und das Skript in das Zielgerät zu laden.

Der Test des Skriptes kann über die serielle Schnittstelle des netTAP's mit einem RS232 Kabel und einer Verbindung zum COM-Anschluss des PCs erfolgen. Als Kommunikationspartner dient die in Windows mitgelieferte Hyperterminal Applikation.

Mit dem Debugger ist eine Kommunikation über USB aufzubauen, wenn der Ablauf des Programms verfolgt werden soll.

14.7.1.2 Terminal-Einstellungen

Auf dem PC ist das Hyperterminal-Programm mit folgenden Einstellungen zu öffnen.

1. Es ist die Kommunikationsschnittstelle einzutragen, an der das RS232-Kabel angeschlossen wurde (in der Regel COM1).
2. Unter Konfigurieren sind folgende Einstellungen vorzunehmen:
   Übertragungsrate (Bits pro Sekunde): =115200
   Datenbits: 8
   Parität: keine
   Stoppbits: 1
   Flussteuerung: Hardware
3. ASCII-Konfiguration
   Eingegebene Zeichen lokal ausgeben.

14.7.1.3 Übergeordnete Steuerung

Für den durchgängigen Funktionstest des Beispielprogramms muss die übergeordnete Steuerung die Funktion eines Bus-Masters übernehmen und den E/A-Datensaustausch mit dem netTAP Gerät durchführen. Dazu ist das netTAP-Gerät mit dem entsprechenden Buskabel mit dem Bus-Master zu verbinden.

14.7.2 Erläuterungen zum Beispiel-Skriptprogramm

Anhand des Beispiels werden die Grundlagen eines sinnvollen Skript-Programmaufbaus erläutert.

14.7.2.1 Verbale Programmbeschreibung

Das Programm „Time“ realisiert eine Uhr, die zyklisch die Uhrzeit auf dem Hyperterminal als ASCII-Text über die serielle Schnittstelle, und über das BUS IO Interface an die Steuerung ausgibt. Nach dem Einschalten läuft die Zeit mit dem Wert 00:00:00 los. Per Tastatureingabe über das Hyperterminal ist es möglich die Zeit zu verändern. Das Programm erlaubt ferner die Zeit auch über die BUS IO-Schnittstelle über die übergeordnete Steuerung zu verändern.

Hinweis: Die Übertragung der Uhrzeit über die Bus IO Schnittstelle gelingt nur, wenn die Steuerung auch entsprechend die Freigabe im Synchronisationsregister gesetzt hat.

Es ist ein 4 stelliger Zähler (h,m,s,ms) realisiert, der Stunden, Minuten, Sekunden und die Zykluszeit aufsummiert. Die Ausgabe erfolgt im Format hh:mm:ss.

Die Zeit kann sowohl über die UART-Schnittstelle, als auch über die BUS IO-Schnittstelle eingestellt werden. Die Eingabe über die serielle Schnittstelle erfolgt im gleichen Format hh:mm:ss und wird mit der Enter-Taste abgeschlossen. Empfängt das Beispielprogramm ein Tastaturzeichen über die serielle Schnittstelle, unterbricht es die Zeitausgabe.

14.7.2.2 netSCRIPT-Programm Aufbau

Hinweis: Bei allen netSCRIPT-Programmen ist zu beachten, dass das gesamte Programm bei jedem zyklischen Start (nach der eingestellten Zykluszeit) neu durchlaufen wird. Daher ist der Definitionsteil des Skriptes vom zyklisch abzuarbeitenden Skriptteil über einen Skriptbefehl zu trennen.

Grundsätzlicher Skriptprogramm-Aufbau:

```
-- 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
```

Abbildung 36: Script Time, Grundstruktur“

Ⓐ Im Kopfbereich eines jeden Scriptes sollten einige Kommentarzeilen stehen, die den Einsatzbereich des Skriptes und Scriptversion dokumentieren.

① In dieser Zeile wird auf die Existenz der Variablen **`state`** abgefragt. Ist diese nicht definiert (beim Erststart) wird der folgende Codeblock abgearbeitet, andernfalls wird der Block nach „**else**“ weitergearbeitet.

Ⓑ In diesem Bereich steht der Definitionsteil des Scriptes, der nur beim erstmaligen Script-Start durchlaufen werden soll.

② In dieser Zeile wird die Variable **`state`** definiert, um einen nochmaligen Durchlauf (bis zum nächsten Spannungsausfall) des Codeblocks Ⓑ zu vermeiden.

Ⓒ Dieser Block ist zunächst auskommentiert. In ihm ist eine Möglichkeit gezeigt, wie eine Ausgabe auf die UART-Schnittstelle solange unterbunden werden kann, bis ein Telgramm von der übergeordneten Funktionseinheit über die BUS IO-Schnittstelle erfolgt ist.

Ⓓ In diesem Bereich steht der Scriptcode, der bei jedem Zyklus durchlaufen wird.

③ Diese Zeile beendet den gesamten Code.

Die Script-Teile Ⓐ, Ⓑ und Ⓒ sind Teile einer „Zustandsmaschine“, die prinzipiell wie folgt aufgebaut ist:

```
if state == nil then
    -- Initialisierung
    ...

    -- Definieren der Zustaende
    STATE_1 = 1
    STATE_2 = 2

    -- Setzen des nächsten Zustands
    state = STATE_1

elseif state == STATE_1 then
    -- Aktionen in Zustand 1
    ...
    -- ggf. Zustand wechseln
    state = STATE_2

elseif state == STATE_2 then
    -- Aktionen in Zustand 2
    ...
    -- ggf. Zustand wechseln
    state = STATE_1

end
```

Bei einem zyklischen Durchlauf dieses Skriptabschnittes (wie es bei der Abarbeitung im Gerät der Fall ist), wird immer nur der aktuell gültige Zustandsteil abgearbeitet, da die Zustands-Variablen zwischen Skript-Ende und dem Skript-Neustart unverändert bleiben.

14.7.2.3 Details des Blocks B

In diesem Block werden:

1. die Funktionen für die Ausgabeformatierung und Zeiteingabeerkennung definiert,
2. unter „--UART Initialisation“ die serielle Schnittstelle im CHAR-Mode initialisiert, mit „**PortOpen()**“
3. unter „-- BusIO Initialisation“ die Schnittstelle zur übergeordneten Steuerung initialisiert. „**BusIOOpen()**“
4. optional die Abfrage der Kommunikationsfreigabe der seriellen Schnittstelle über die BusIO Schnittstelle (übergeordnete Steuerung) realisiert siehe hierzu „Details des Blocks C“

14.7.2.4 Details des Blocks C

Um diesen Block zu aktivieren sind die Kommentarzeichen unter ① und ② des unten dargestellten Programmabschnittes zu entfernen.

```
      -- 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)
```

Abbildung 37: Script Time, Block C

Solange „**state**“ auf „**STATE_WAIT**“ gesetzt ist, ist die „**elseif**“-Bedingung **true** und die folgende „**else**“-Bedingung unter ④ wird nicht ausgeführt. Erst wenn bei ③ „**state**“ auf „**STATE_RUN**“ gesetzt wird, wird im nächsten zyklischen Aufruf der Code nach der „**else**“-Bedingung unter ④ ausgeführt, und damit die UART-Schnittstelle bedient.

14.7.2.5 Details des Blocks D

Der Block (D) teilt sich in drei Abschnitte auf:

1. Zeitzählung
2. UART- Kommunikation
3. BusIO- Kommunikation

Abschnitt 1 - Zeitzählung

```
-- 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

      -- set the new send strings
      formatTime()

      -- toggle the "run" LED every second
      util.SetLed("run", s % 2 == 1)
```

(1) (2) (3)

Abbildung 38: Script Time - Block D Abschnitt 1

Im Abschnitt (1) der obigen Abbildung werden die Zähler für die Zeitanzeige versorgt.

Bei (2) wird eine Zeitformatierungsfunktion aus dem Initialisierungsbereich (B) aufgerufen.

Bei (3) wird die LED auf dem UART-Modul zum blinken angesteuert.

Abschnitt 2 - UART Kommunikation.

```
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
```

Abbildung 39: Script Time, Block D Abschnitt 2

① Es wird abgefragt, ob über die UART-Schnittstelle etwas empfangen wurde.

② Es wird geprüft ob der Empfang ok. war. Bei fehlerhaftem Zeichenempfang wird eine Fehlermeldung ausgegeben.

③ Die empfangenen Zeichen werden angereiht.

④ Ist das letzte empfangene Zeichen ein „CR“, werden die angereihten Zeichen der Funktion „parseTime“, die im Bereich Ⓑ definiert wurde, übergeben.

⑤ In der Funktion „parseTime“ wird geprüft, ob die eingegebenen Zeichen als Uhrzeit interpretiert werden können. Wenn ja, wird die neue Zeit weiter verarbeitet, andernfalls wird **nill** zurückgegeben und eine Fehlermeldung auf dem Port ausgegeben.

⑥ Es wird die aktuelle Uhrzeit ausgegeben, unabhängig davon, ob Zeichen über die serielle Schnittstelle empfangen wurden. Der Speicher für die Ausgabezeit auf der seriellen Schnittstelle wird initialisiert.

Abschnitt 3 - BusIO-Kommunikation

```
(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
```

Abbildung 40: Script Time - Block D Abschnitt 3

(1) Es wird die Bus IO-Schnittstelle auf neue Daten geprüft.

(2) War ein Telegramm in der Schnittstelle, wird zunächst die Funktion „parseTime“ aufgerufen. In Abhängigkeit des Ergebnisses dieser Funktion wird ggf. ein Fehler zur BUS IO Schnittstelle übergeben. War die Interpretation des Telegramms erfolgreich, werden durch die Funktion „parseTime“ auch die Zeitzähler neu gesetzt.

(3) Ist eine neue Ausgabezeit vorhanden, wird diese der BusIO-Schnittstelle übergeben.

(4) Hier ist eine optionale Möglichkeit gezeigt, wie die Ausgabe auf der UART-Schnittstelle beendet werden kann, wenn die Telegrammübergabe an die BusIO-Schnittstelle fehlgeschlagen ist.

14.7.3 Bedienung des Programms

14.7.3.1 Bedienung über die serielle Kommunikationsschnittstelle RS232

Das Programm startet eigenständig nach dem Herunterladen mit der SYCON.net Software mit der seriellen Übertragung, ohne das eine Freigabe der übergeordneten Steuerung über die BUS IO Schnittstelle nötig ist.

- Stellen Sie über die COM-Schnittstelle Ihres PC's eine Verbindung zur seriellen Schnittstelle des Zielgerätes her.
- Starten Sie auf dem PC ein Terminalprogramm mit den im Abschnitt 14.1 angegebenen Parametern.

Das Fenster des Terminals zeigt dann folgende Daten:

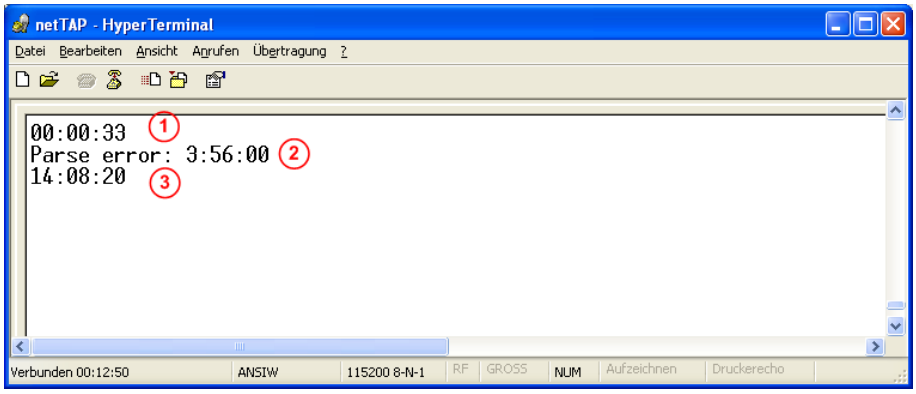


Abbildung 41: Script Time, Anzeige auf seriellem Terminal

(1) In dieser Zeile sehen Sie die Ausgabe ohne eine vorgegebene Startzeit. Das Skript läuft seit 33 Sekunden.

Für eine Uhrzeitvorgabe können Sie im Terminalfenster direkt eine Zeit im Format hh:mm:ss eingeben. Die Eingabe ist mit der „Enter-Taste" abzuschließen.

(2) In der Zeile sehen Sie die Reaktion des Skriptes auf eine fehlerhafte Eingabe mit einer Ausgabe der empfangenen Zeichen. Es fehlte eine Stelle in der Stunden-Eingabe.

(3) In dieser Zeile sehen Sie die Zeitausgabe nach einer korrekten Uhrzeitvorgabe. Die aktuelle Zeitzählung wird immer in der selben Zeile ausgegeben.

14.7.3.2 Bedienung über die übergeordnete Netzwerkschnittstelle

Für die Kommunikation über die übergeordnete Netzwerkschnittstelle des netTAP-Gerätes benötigen Sie einen Bus-Master.

Bitte beachten Sie, dass die Ein- und Ausgabelänge der zu übertragenden E/A-Daten von und zum netTAP in der Steuerung/Master auf mindestens 24Byte konfiguriert werden, damit der Datenkopf für die Synchronisation zwischen netTAP und Steuerung über die E/A Daten erfolgen kann. Die reinen Nutzdaten beginnen ab dem 25'ten Byte. Die Uhrzeit wird vom Programm über die BUS IO-Schnittstelle im Format hh:mm:ss übertragen. Es sind also 8 Nutzdatenbytes zusätzlich an E/A Daten zu übertragen. Insgesamt benötigt das Beispielprogramm also 32Byte E und A Daten.

Im folgenden Beispiel wurde eine CifX-Karte von Hilscher als PROFINET IO Controller genutzt.

➢ In dem Übergabeverfahren zwischen Master und netTAP ist zunächst vom Master aus eine Freigabe für Sende- und Empfang zu erfolgen. Dazu muss im Synchronisationsregister Byte 0…3 der Ausgabedaten der Wert 0x000000C0 (LSB first) gesendet werden.

Das folgende Bild zeigt oben die Eingangsdaten die vom netTAP über den Bus-Master gelesen und unten die Ausgabedaten die von Bus-Master geschrieben wurden.

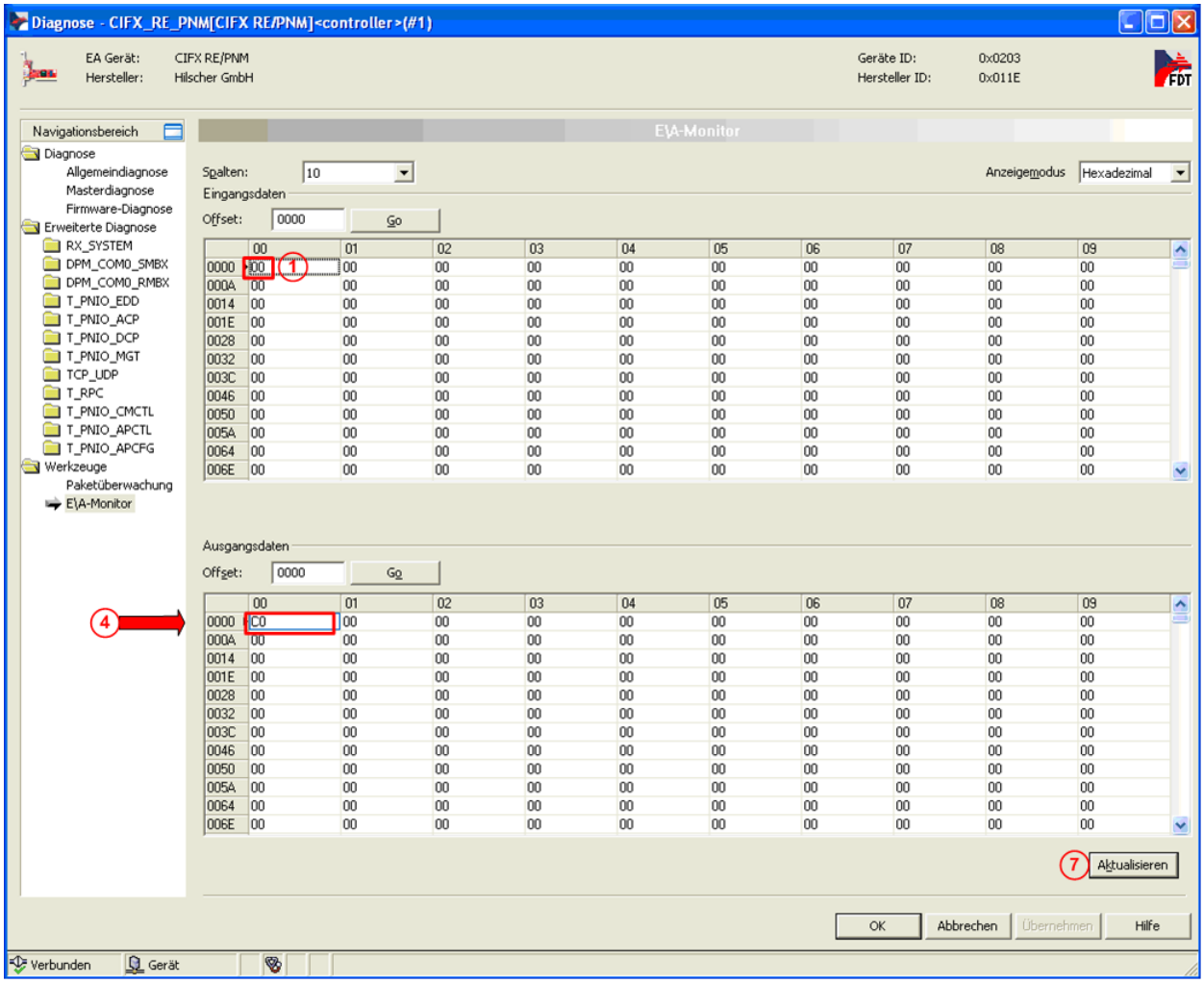


Abbildung 42: Script Time, Feldbusinitialisierung Datenaustausch

① Handshakebyte 0: Der Wert ist 0x00, wenn im Skript die Funktion „:BusIOSetRun“ nicht aufgerufen wird. Wurde diese Funktion bereits aufgerufen, steht hier der Wert „08h“.

➢ Nach dem Setzen der Freigabebits 6 und 7 (0xC0) (④) und anschließender Betätigung der Schaltfläche ⑦ (wodurch die Daten an das Script versendet werden) antwortet das Beispielprogramm unmittelbar mit der Sendung der aktuellen Zählerwerte (Uhrzeit). Im Bus-Master ist dieser Wert in den Eingabedaten zu lesen.

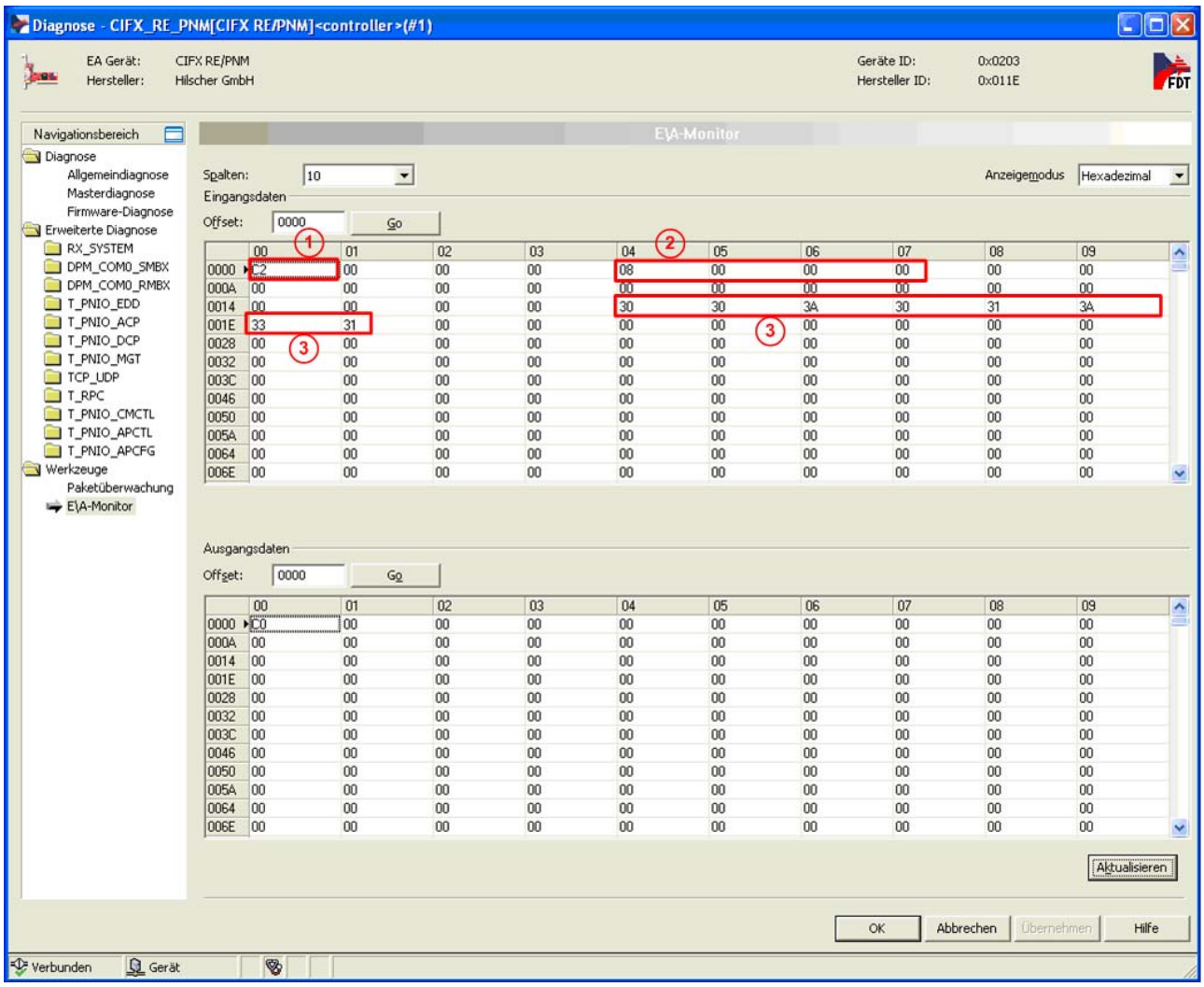


Abbildung 43: Script Time, Sendung der aktuellen Zeit

① Im Synchronisationsregister Byte 0 wird mit 0xC2 die Übertragung der Nutzdaten bestätigt. Mit der Funktion „:BusIOSetRun“ im Skript wird der Wert 0xCA zurückgegeben.

② Byte 4..7 enthält die empfangene Nutzdatenlänge.

③ Ab Byte 24 bis Byte 31 wird die Uhrzeit zurückgegeben. Hier 0x30 0x30 0x3A 0x30 0x31 0x3A 0x33 0x31 → 00:01:31. Die Zeit im Skript wurde bislang noch nicht gestellt.

Soll ein weiteres Mal die Zeit vom Beispielprogamm übertragen werden, muss ihm mitgeteilt werden, dass die erste Datenübertragung erfolgreich war. Dieses erfolgt durch Angleichen des Zustandes des Bits 1 im Ausgabe-Synchronisationsregister Byte 0 an den Zustand des Bits 1 im entsprechenden Eingabe-Synchronisationsregisters Bild Pos. ①.

In den Ausgabedaten ist hierzu der Wert in Byte 0 C0h auf 0xC2 zu verändern.

Soll die Uhrzeit über den Feldbus gesetzt werden, ist wie folgt vorzugehen:

1. Stellen Sie fest welche Handshakesituation zuletzt vom Beispielprogramm an den Master gesendet wurde. 0xC0 oder 0xC2, siehe ① im folgenden Bild.

2. Setzen Sie das Ausgabe-Synchronisationsregister Byte 0 an (4) des folgenden Bildes auf den Zustand der folgenden Tabelle und invertieren Sie damit den Zustand des Bits 0.

Letzter vom Beispielprogramm erhaltene Synchronisationsregister-Wert.	An das Beispielprogramm zu sendender Synchronisationsregister-Wert.
C0 (C8)	C1, Daten an Skript senden
C2 (CA)	C3, Daten vom Skript lesen

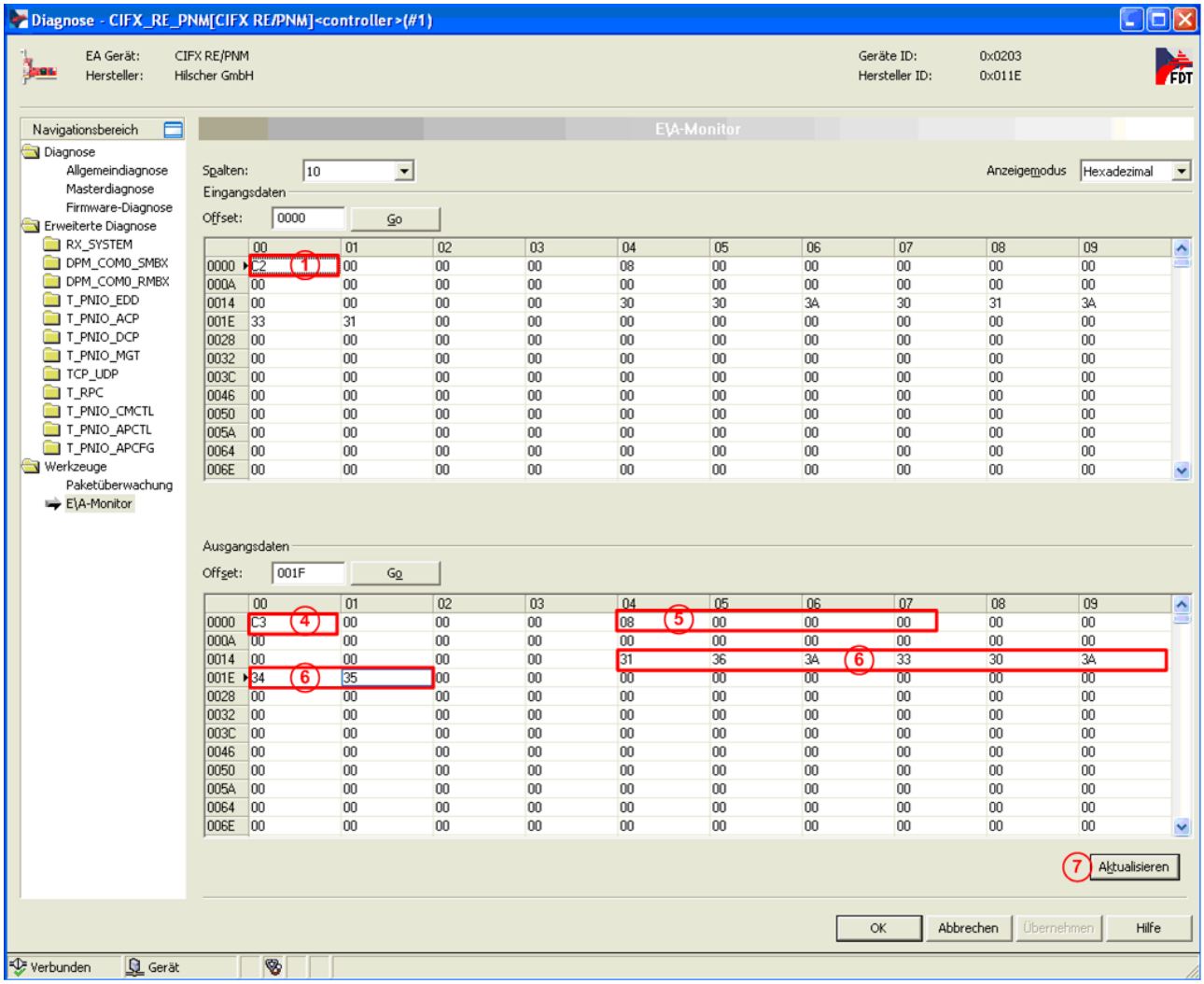


Abbildung 44: Script Time, Stellen der Uhrzeit über Feldbus

3. Geben Sie im Handshakebyte ((4)) die notwendige Date ein, damit ein stellen der Zeit im Skript möglich wird. Geben Sie im Byte 4 ((5)) die zu sendende Datenlänge (hier 0x08) und im Byte 24 bis Byte 31 ((6)) die Zeit ein, mit der das Skript nach Absenden des Telegramms weiter arbeiten soll. Im obigem Bild wurden die Zeit auf den Wert 0x31 0x36 0x3A 0x33 0x30 0x3A 0x34 0x35 → als Uhrzeit: 16:30:45“ gesetzt. Mit betätigen der Schaltfläche (7) werden die Daten versendet.

   Wollen sie nur erneut Daten lesen, geben Sie im Byte 4 ((5)) die Datenlänge 0 ein.

① Der Datenerhalt wurde vom Beispielprogramm durch Angleichen des Bits 0 im Eingabe-Synchronisationsregister mit 0xC3 bestätigt. Es könnten nun neue Daten gesendet werden.

③ Die aktuelle Zeit entspricht nun den gesendeten Daten und ist direkt auf der seriellen Schnittstelle im Terminalprogramm sichtbar.

14.7.3.3 Möglicher Handshakeablauf aus Sicht der übergeordneten Steuerung

Beim Handshake Eingang in Klammern stehende Werte bedeuten den Wert des Bytes nach Aufruf der Funktion „:BusIOSetRun“ im Skript.

Handshake Eingang Byte 0	Handshake Ausgang Byte 0	Beschreibung
00 (08)	00	Initialzustand.
	C0	Freigabe der Kommunikation,→ das Skript sendet die ersten Daten.
C2 (CA)		Daten wurden vom Skript versand.
	C2	Der Datenempfang wird bestätigt, → das Skript sendet neue Daten.
C0 (C8)		Daten wurden vom Skript versand.
	C0	Datenempfang wird bestätigt →das Skript sendet neue Daten.
C2 (CA)		Daten wurden vom Skript versand.
	C1	Es werden Daten ans Skript versendet.
C3 (CB)		Daten wurden vom Skript empfangen.
	C3	Der Datenempfang wird bestätigt → das Skript sendet neue Daten.
C1 (C9)		Daten wurden vom Skript versand.
	C1	Der Datenempfang wird bestätigt, → das Skript sendet neue Daten.
C3 (CB)		Daten wurden vom Skript versand.
	C2	Der Datenempfang wird bestätigt. → Es werden neue Daten ans Skript versendet.
C0 (C8)		Daten wurden vom Skript versand.
	C0	Der Datenempfang wird bestätigt. → Es werden neue Daten ans Skript versendet.
C2 (CA)		Daten wurden vom Skript versand.
	C2	Der Datenempfang wird bestätigt. → Es werden neue Daten ans Skript versendet.
C0 (C8)		Daten wurden vom Skript versand.
	C1	Der Datenempfang wird bestätigt. → Es werden neue Daten ans Skript versendet.
C3 (CB)		Daten wurden vom Skript versand.
	C2	Der Datenempfang wird bestätigt. → Es werden neue Daten ans Skript versendet.
C0 (C8)		Daten wurden vom Skript versand.

15 Verzeichnisse

15.1 Abbildungsverzeichnis

15.2 Tabellenverzeichnis

16 Glossar

Lua

Lua ist eine frei nutzbare Scriptsprache, die von der "**Pontifícia Universidade Católica do Rio de Janeiro**- ©" entwickelt wurde.

netSCRIPT

Ist eine skriptbasierte Programmiersprache für Kommunikationsgeräte der Hilscher GmbH, die es Anwendern ermöglicht selbst Programmabläufe / Protokollumsetzungen in Zielgeräte zu programmieren.

netTAP

Ist ein Gerät zur Protollumsetzung zweier unterschiedlicher Netzwerksysteme der Hilscher GmbH, auf dem netSCRIPT eingesetzt werden kann.

SYCON.net

PC-Konfigurationswerkzeug der Hilscher GmbH zur Konfiguration von Feldbusgeräten.

UART

Eine UART-Schnittstelle dient zum Senden und Empfangen von Daten über eine Datenleitung und bildet den Standard der seriellen Schnittstellen an PCs und Mikrocontrollern (z. B. RS-232 oder RS-485).

BUS IO

Ist die Bezeichnung der Ein- und Ausgabeschnittstelle des übergeordneten E/A-Netzwerkes innerhalb netSCRIPT. Es werden an die BUS IO-Schnittstelle Daten zur Steuerung (Master,Host,SPS) übergeben oder von dort Daten von der Steuerung empfangen.

17 Technische Daten

Speicherplatz für Programme und Variable	ca. 1 Mbyte
Zykluszeit	min. 1 ms max. 1000 ms
Schnittstellenunterstützung	RS-232, RS-422, RS-485 im Zeichen- und Blockmode
	IO-Datenkommunikation zum Gateway
Max. Nutzdatenlänge	1024 Byte

18 Kontakte

Hauptsitz

Deutschland
Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Telefon: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

Support
Telefon: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Niederlassungen

China
Hilscher Systemautomation (Shanghai) Co. Ltd.
200010 Shanghai
Telefon: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn

Support
Telefon: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

Frankreich
Hilscher France S.a.r.l.
69500 Bron
Telefon: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr

Support
Telefon: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

Indien
Hilscher India Pvt. Ltd.
New Delhi - 110 025
Telefon: +91 11 40515640
E-Mail: info@hilscher.in

Italien
Hilscher Italia srl
20090 Vimodrone (MI)
Telefon: +39 02 25007068
E-Mail: info@hilscher.it

Support
Telefon: +39 02 25007068
E-Mail: it.support@hilscher.com

Japan
Hilscher Japan KK
Tokyo, 160-0022
Telefon: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp

Support
Telefon: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

Korea
Hilscher Korea Inc.
Suwon, 443-810
Telefon: +82-31-204-6190
E-Mail: info@hilscher.kr

Schweiz
Hilscher Swiss GmbH
4500 Solothurn
Telefon: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch

Support
Telefon: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

USA
Hilscher North America, Inc.
Lisle, IL 60532
Telefon: +1 630-505-5301
E-Mail: info@hilscher.us

Support
Telefon: +1 630-505-5301
E-Mail: us.support@hilscher.com