IF TOOLS
Discover your connection

Manual

MSB-RS485-PLUS

Version 7.0.2

www.iftools.com

Inhaltsverzeichnis

1 Analyse von RS422/485 Bussystemen

Im Gegensatz zu anderen Bussen definieren RS422 bzw. RS485 nur die elektrischen Eigenschaften, alle weiteren Protokollebenen können frei spezifiziert werden. Eine Analyse muß deshalb neben den physikalischen Besonderheiten auch die unterschiedlichsten Protokolle berücksichtigen.

RS485 und RS422 (bzw. korrekter EIA-422 und EIA-485) werden aufgrund der großen Ähnlichkeit oft synonym verwendet, wobei der EIA-422 Standard als eine Art Untermenge von EIA-485 angesehen wird. Dies ist jedoch nur teilweise richtig.
Beide Standards benutzten ein verdrehtes Leitungspaar, um die invertierten und nichtinvertierten Pegel eines 1-Bit Datensignals zu übertragen. Am Empfänger wird aus der Differenz der beiden Spannungspegel das ursprüngliche Datensignal rekonstruiert. Gleichtaktstörungen wirken sich dadurch nicht auf die Übertragung aus, was zu einer erheblich besseren Störsicherheit beiträgt.
Als Konsequenz sind deshalb alle Daten- und Handshake Leitungen als Leitungspaare ausgeführt. Allerdings existiert weder für EIA-422 noch EIA-485 eine einheitliche Anschlußbelegung.

Eine EIA-422 Verbindung besteht i.a. aus zwei Leitungspaaren zum Senden und Empfangen und einer gemeinsamen Masse (was der klassischen 3-Draht RS232 Verbindung entspricht). Im Falle von RTS/CTS Handshake müssen 2 zusätzliche Leitungspaare vorhanden sein (EIA-422 wurde ursprünglich entwickelt um die Einschränkungen von EIA-232 Verbindungen zu überwinden).
Mit EIA-422 lassen sich Vollduplex Punkt-zu-Punkt Verbindungen und Multidrop Netzwerke realisieren. Letzteres erlaubt den unidirektionalen Anschluß von bis zu 10 Empfängern an einem Sender. Dabei erfolgt die Übertragung der Daten immer nur in einer Richtung (unidirektional), vom Sender zu den max. 10 Empfängern.

EIA-485 ist von vornherein als bidirektionales Bussystem mit bis zu 32 Teilnehmern (und mehr[1]) konzipiert. Die Daten können wahlweise über ein einzelnes Leitungspaar Halbduplex übertragen werden (die sogenannte 2-Draht-Technik oder kurz 2-wire), oder in einer Vollduplex fähigen Variante mit zwei getrennten

[1] abhängig vom sogenannten Unit Load können es bis zu 256 Teilnehmer sein

Sende- und Empfangsleitungspaaren (4-Draht-Technik, kurz 4-wire).
In einem 2-Draht Bussystem sind alle Sender und Empfänger gemeinsam mit einem Leitungspaar verbunden. Hauptvorteil der 2-Draht Technik ist ihre Multi-Master Fähigkeit, d.h. jeder Busteilnehmer kann prinzipiell mit anderen Teilnehmern Daten austauschen.
Die wohl bekannteste Anwendung auf Basis eines EIA-485 2-Draht Systems ist der PROFIBUS.

4-Draht Busse werden ausschließlich als Master-Slave Systeme verwendet (z.B. DIN-Meßbus). Dabei ist der Datenausgang des Masters mit allen Dateneingängen der Slaves über ein einzelnes Leitungspaar verbunden. Die Datenausgänge der Slaves wiederum sind über das zweite Leitungspaar mit dem Eingang des Masters verdrahtet.

In beiden Varianten kann immer nur ein Teilnehmer die Leitung 'treiben' (senden), die anderen Teilnehmer müssen ihren Sendebaustein deshalb in einen hochohmigen Zustand (tri-state) versetzen können.
Einige EIA-485 Geräte sorgen automatisch für eine korrekte Umsetzung des Tri-State Zustand, bei anderen muß dies explizit per Software erfolgen.

Physikalisch unterscheiden sich beide Schnittstellen nur geringfügig, so dass EIA-485 Geräte ohne Probleme in EIA-422 Systemen eingesetzt werden können, jedoch nicht umgekehrt da EIA-422 Geräte i.a. keinen Tri-State Zustand kennen.
Bei der Analyse einer EIA-422/485 Verbindung sind deshalb nicht nur die verschiedenen Anschlußvarianten (und die evtl. damit verbundenen Bus Signale) zu berücksichtigen. Da die EIA-422/485 Spezifikationen keine Aussagen über die Protokollebene macht, haben sich im Laufe der Zeit eine Reihe von unterschiedlichsten Übertragungsprotokollen etabliert; Protokolle mit asynchroner (UART basierender) als auch synchroner serieller Übertragung.
Protokolle mit synchroner Datenübertragung verwenden die unterschiedlichsten Arten der Bitübertragung, (mit und ohne separatem Synchrontakt) die jeweils eine spezielle Hardware erfordern.
Im Gegensatz dazu setzen asynchrone Übertragungstechniken auf den UART Baustein, seit Jahrzehnten Standard für serielle Schnittstellen bei PC's und Mikrocontrollern. Auf Grund ihrer einfachen Anbindung an PC's (über EIA-232 bzw USB auf EIA-422/485 Umsetzer) sind diese häufiger verbreitet, weshalb wir uns im folgenden auf UART basierende Protokolle konzentrieren. Dazu gehören:

1. **Din-Messbus**
2. **Modbus ASCII**
3. **Modbus RTU**
4. **Profibus**
5. **Anwenderspezifische Protokolle**

Die Art des Protokolles spielt dabei eine entscheidende Rolle nicht nur bei der Aufzeichnung und Auswertung der Kommunikation sondern insbesondere bei der Wahl des Analyse-Tools.

Nach diesem kurzen Exkurs in die Spezifikationen von EIA-422/485 ist der Rahmen gesteckt für unsere eigentliche Frage: Welche Möglichkeiten zur Analyse von EIA-485[2] Verbindungen gibt es und wofür sind sie geeignet?

Durch die einfache Anbindung eines auf einer asynchronen Datenübertragung basierenden EIA-485 Busses an einen herkömmlichen PC sind folgende Verfahren zur Auszeichnung und ggf. weiterer Auswertung mit entsprechender Software möglich:

1. **Aufzeichnung der Daten durch den seriellen Treiber des PC's**
2. **Bus-Abgriff durch EIA-485 Umsetzer (2-Draht Bus)**
3. **Bus-Abgriff durch 2 EIA-485 Umsetzer (4-Draht Bus)**
4. **Abtastung der Bus-Leitungen durch spezielle Hardware**

1.1 Spezieller serieller Treiber

Ist einer der Kommunikationspartner ein PC (i.a. der Master), kann durch Installation eines entsprechenden Treibers jedes empfangene bzw. gesendete Datenbyte mit protokolliert werden. Dieses Verfahren erlaubt allerdings nur das Protokollieren der durch den 'seriellen Treiber' des Betriebssystems verarbeiteten Bytes. Die Nachteile sind:

Durch Pufferüberläufe verloren gegangene Daten werden nicht erkannt.

Eine genaue Zeitangabe der übertragenen Bytes ist ebenfalls nicht möglich, da ein- und ausgehende Datenbytes zwar per 'Interrupt' signalisiert werden, die eigentliche Bearbeitung aber vom Betriebssystem zu einem nicht genau vorhersagbaren 'späteren' Zeitpunkt erfolgt.

Daher sind die Zeitangaben solcher 'Sniffer' Programme, oftmals sogar im Millisekunden Bereich, mit Vorsicht zu genießen. Sie entsprechen eher dem Zeitpunkt, wann die Bearbeitung des ein/ausgehenden Zeichens seitens des Betriebssystems erfolgte, und nicht dem Zeitpunkt, wann das Zeichen wirklich physikalisch auf der Leitung präsent war.
Aussagen über ein korrektes Zeitverhalten können deshalb nur bedingt gemacht werden. Bestimmte Busse wie z.B. Modbus RTU oder Profibus definieren eine Sendepause als Telegrammstart bzw. Ende. Die Daten innerhalb eines Telegrames (oder Frames) müssen 'schlupffrei' übertragen werden.

Informationen über den Tri-State Zustand gehen verloren, da der serielle Treiber nur die beiden logischen Zustände Buszustände verarbeitet. Fehler durch Datenkollision auf Grund mehrerer gleichzeitig sendender Busteilnehmer werden daher nicht erkannt.

1.2 Bus-Abgriff 2-Draht Bus

Der PC wird über einen entsprechenden EIA-485 Schnittstellen Umsetzer (EIA-232 auf EIA-485 bzw. USB auf EIA-485) quasi als zusätzlicher Bus-Teilnehmer

[2] EIA-485 Analysetools sind i.a. auch für EIA-422 Verbindungen geeignet. Im folgenden sprechen wir deshalb nur noch von EIA-485 wenn wir beide Standards meinen.

an den Bus angeschlossen.
Diese Variante erlaubt das Mitprotokollieren aller übertragenen Datenbytes (auch von zwei nicht PC Geräten) mit einem PC und entsprechender Software wie z.B. Hyperterm. Die Nachteile:

Da Sende- und Empfangsdaten über die gleiche Leitung gehen, kann physikalisch nicht zwischen gesendeten und empfangenen Daten unterschieden werden. Dazu ist die Auswertung der Telegramme nötig.

Bei Aussagen über das Zeitverhalten gilt das bereits unter 1.1 gesagte.

Die Anbindung eines EIA-485 Umsetzers erfolgt i.a. als serieller COM Port (entweder als virtueller COM Port im Falle eines USB auf EIA-485 Konverters oder direkt bei EIA-232 auf EIA-485 Schnittstellenwandler).
In beiden Fällen gehen Informationen über den Tri-State Zustand verloren, mit der Konsequenz, daß Busfehler verursacht durch mehrere aktive Sender nicht erkannt werden.

1.3 Doppelter Bus-Abgriff 4-Draht Bus

Die Sende- und Empfangsleitungen werden getrennt aufgezeichnet. Allerdings erfordert diese Variante zwei EIA-485 Umsetzer sowie spezielle serieller Treiber, da die getrennt empfangenen Datenbytes mit einer Zeitmarke oder eindeutigen Nummer versehen werden müssen, um sie miteinander synchronisieren zu können.

Die Datenrichtung wird hier zwar korrekt erkannt, allerdings kann aus oben genannten Gründen nicht eindeutig bestimmt werden, in welcher Reihenfolge Sende- und Empfangsdaten vorliegen. Letztendlich wurde diese Möglichkeit deshalb hauptsächlich unter MS-DOS verwendet, da dieses Betriebssystem (auf Grund seiner Einfachheit) eine direkte Echtzeitbearbeitung des UART erlaubt.

Für Aussagen über das Zeitverhalten sowie den Tri-State Buszustand gilt das gleiche wie unter 1.2.

1.4 Abtastung

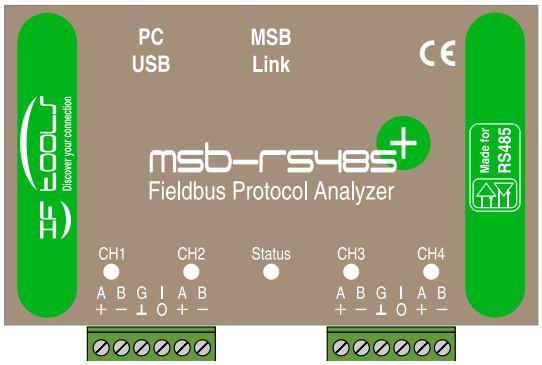
Dieses Verfahren erfordert eine zusätzliche Hardware, die alle Signalleitungen simultan abtastet und entsprechend aufarbeitet. Die Vorteile:

Da die Abtastung und Auswertung unabhängig von den angeschlossenen Geräten sowie des PC's erfolgt, werden ungültige Tri-State Zustände, falsche Baudraten bzw. UART Einstellungen der Teilnehmer eindeutig erkannt und protokolliert.

Darüber hinaus erlaubt nur die Abtastung aller Signalleitungen eine genaue zeitliche Erfassung der Datenleitungen (und eventuell verwendeter Handshake Leitungen wie RTS, CTS bei Punkt-zu-Punkt Verbindungen). Unabdingbar für Protokolle, die eine genaue Einhaltung von Sendezeiten/Pausen verlangen oder ein anderes genaues Zeitverhalten spezifizieren.

Mehr noch: Selbst 'jitternde' oder leicht abweichende Baudraten einzelner Teilnehmer sind damit erkennbar.

Abtastende Analysatoren vereinigen damit die Vorzüge eines Protokollanalysers mit den Eigenschaften eines Digitalscopes und bieten neben der Protokollierung des Datentransfers auch die physikalische Darstellung der Leitungspegel.

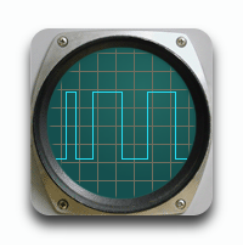


2

MSB-RS485-PLUS Analyser

Der MSB-RS485-PLUS Analyzer ist ein unverzichtbares Werkzeug zur Analyse und Optimierung von RS485/422 Verbindungen. Als eigenständiges Gerät liefert es Mikrosekunden genaue Daten über jede Leitungsänderung - unabhängig von PC und OS. Ausgestattet mit einer Vielzahl von Visualisierungs-Tools erlaubt es einen detailierten Blick in jede RS485/422 Kommunikation und erkennt Zustände, die nur einer echten 'Hardware Lösung' vorbehalten sind.

Der Analyser MSB-RS485-PLUS tastet alle 4 Feldbus Datenkanäle sowie 2 digitale IO Anschlüsse simultan mit einer Abtastrate von maximal 200 MHz ab. Dabei werden alle 'Ereignisse', sprich Pegelwechsel, mit einer Mikrosekunden genauen Zeitmarke versehen.
Als Ereignisse gelten alle Änderungen der Leitungspegel inklusive des Tri-State Zustandes, d.h. Wechsel von 0 (space) auf 1 (mark) oder von einem inaktiven (Tri-State) zu einem aktiven logischen Pegel.
Die aufgezeichneten Daten werden per USB 2.0 mit 480 MBit an den PC übertragen. Ein interner Cache dient dabei als zusätzlicher Zwischenpuffer bei sehr hohem Datenaufkommen.



Max. 200MHz Abtastung bereit für Feldbusse wie CAN, Profibus mit 10ns genauer Zeitauflösung

Der Analyser bietet Echtzeit Analyse mit zeitgleichen Zugriff und Anzeige beliebiger Abschnitte; auch während einer aktiven Aufnahme. Die max. Dateigröße für Aufzeichnungen ist 4 GByte und unbegrenzt (nur limitiert durch Festplatten Kapazität) bei Verwendung der speziellen Kommandozeilen Tools für Langzeitaufnahmen. Die Aufzeichnungszeit ist dabei nur abhängig von den ausgewählten Ereignissen und der Datenrate.

Zwei getrennt operierende USARTs unterstützen nicht nur die Analyse von Aufzeichnungen mit beliebigen Baudraten im Bereich 1 Bps bis 20 MBps sondern auch die Aufzeichnung zweier Bus Systeme mit unterschiedlichen Übertragungseigenschaften (Baudrate, Datenformat).
Und: Die MSB-RS485-PLUS Hardware ist bereits für Anwendungen mit synchronen Bussystemen vorbereitet. Die Freigabe erfolgt durch spätere Software Updates.

2.1 Vorteile einer Hardware Lösung

Der MSB-RS485-PLUS Analyzer bietet die Fähigkeiten eines Logik-Analysers verbunden mit einem sehr attraktiven Preis. Dabei werden die Nachteile reiner Software Lösungen durch die direkte Verarbeitung der Signaländerungen in einer unabhängigen Hardware vermieden.

Auf reiner Software basierende Lösungen sind abhängig von den nicht gleichmäßigen Reaktions- und Bearbeitungszeiten von Interrupt Anforderungen durch das Betriebssystems. Die Verwendung von USB auf EIA-422/485 Umsetzern fügt noch die Verzögerungen des USB Subsystems hinzu. Ist die Interruptbearbeitung zu langsam, können zusätzlich durch den Überlauf der nur sehr begrenzten Empfangspuffer Zeichen verloren gehen.
Die resultierenden Zeitmarken repräsentieren daher im besten Falle die Zeiten der Interrupt Bearbeitung des Betriebssystems, aber nicht den wirklichen Zeitpunkt des auftretenden Ereignisses.

Entsprechend ungenau ist die zeitliche Relation der Sende- und Empfangsdaten bei der Verwendung von zwei EIA-485 Umsetzern zur Aufzeichnung einer Vollduplex 4-Draht Verbindung (T+, T–, R+, R–).
Insbesondere bei Protokollen mit genau einzuhaltendem Zeitverhalten (max. Pause zwischen den Datenbytes eines Telegrammes bzw. Pause zwischen den Telegrammen wie sie z.B. von ModBus RTU und ProfiBus gefordert werden) können Sie nicht sicher sein, ob der gemessene zeitliche Ablauf auch der Wirklichkeit entspricht.

Im Gegensatz zu herkömmlichen Umsetzern unterstützt der MSB-RS485-PLUS Analyser auch Protokolle mit 9-Bit Datenlänge. 9-Bit Werte werden bei bestimmten binären Protokollen als Adressbyte sowie zur Kennzeichnung des Frame- bzw. Telegrammstart verwendet.

Durch die Anbindung des EIA-485 Busses per serieller Schnittstelle gehen alle Informationen über den Tri-State Zustand verloren. Mit der Konsequenz das Datenkollisionen durch mehrere gleichzeitig aktive Sender nicht erkannt werden. Die MSB-RS485-PLUS erfasst den Tri-State Zustand auch dann korrekt, wenn das Differenzsignal auf einen bestimmten Ruhepegel fixiert ist (Pullup-, Pulldown-Widerstände).

2.1.1 Vorteile der PLUS Serie

Der MSB-RS485-PLUS Analyzer im Vergleich zum Vorgängermodell:

Feature	MSB-RS485 PLUS	MSB-RS485
Real-Time Zeitmarken und Auflösung	10ns	1μs
Interne Uhr Genauigkeit (Langzeitaufnahme)	2.5ppm	50ppm
Maximale Baudrate	20 MBps	1 MBps
PC $\Leftrightarrow$ Analyzer USB Transfer Rate	480 MBit	12 MBit

Erkennt ungültige Pegel (Tri-State)	✓	✓
Synchronisierung mehrerer Analyser	✓	✓
120Ω Link/Sync Terminierung per Software	✓	—
Analyse synchroner Bus Systeme[1]	✓	—
Getrennte USART Einstellungen für Data A und B[1]	✓	—
Automatische Baudrate und Datenformat Erkennung	✓	✓
Erkennung von Baudrate Schwankungen und falschen Bitzeiten	✓	✓
120Ω Terminierung für alle 4 Feldbus Kanäle via Software	✓	—
Aktive Datenausgabe	✓	✓
Zusätzliche digitale IO Anschlüsse	2x 50mA	2x 10mA
Beliebige Baudraten von 1 bis max. MBps	✓	✓
Beliebige Baudrate von 1 Baud to max. Bps	✓	✓
Darstellung der Tri-State Datensignale	✓	✓
Support von 9-Bit Protokollen	✓	✓
Integrierter LED Tester für Signalpegel	✓	—
Korrekte Zeitbeziehungen zwischen Daten- und Kontrollleitungen	✓	—
Phoenix Stecker für Bus Anschluss	6 x 3.81mm[2]	6 x 3.5mm
Gehäuse für Phoenix Bus Anschluss Stecker	✓	—

1 Im Gerät bereits vorhanden, Funktion wird durch ein späteres Software Update aktiviert.

2 Erlaubt Anschlussleitungen mit dickerem Querschnitt und optional die Verwendung eines Gehäuses für die Anschlussstecker.

2.2 Innovatives Software Konzept

Die Analyser Software ist als Multi-Process Architektur und Abbild des OSI Schichten Modells konzipiert und läuft unter allen modernen Microsoft Windows OS (XP, Vista, Windows 8/8.1 und Windows 10) sowie allen modernen Linux 32 und 64 Bit Systemen.

Für Windows...
XP, Vista, 8, 8.1, 10

Bereits während der Aufzeichnung können beliebige Ausschnitte des Datentransfers untersucht werden. Dies schließt sowohl die physikalische Darstellung des Signals (Scopedarstellung) in unterschiedlichen zeitlichen Auflösungen also auch die Anzeige der übertragenen Datenbytes ein.
Die Software des MSB-RS485-PLUS ist modular als Multi-Process Architektur

...und alle Linux OS
32 und 64 Bit

aufgebaut. Während das Kontrollprogramm die Aufzeichnung kontrolliert, können die übertragenen Daten/Signale durch beliebig viele 'Analysefenster' zu unterschiedlichen Zeitpunkten untersucht und analysiert werden.
So kann z.B. das physikalische Signal eines 2-Draht Busses bzw. der beiden Datenleitungen eines 4-Draht Systems zum aktuellen Zeitpunkt verfolgt und gleichzeitig ein früherer Ausschnitt in einer höheren Auflösung betrachtet werden. Dies gilt im übrigen für alle Analysefenster und erlaubt damit auch den Vergleich übertragener Daten zu unterschiedlichen Zeitpunkten.

Umfangreiche Suchalgorithmen gestatten die gezielte Suche nach bestimmten Datensequenzen, wobei durch 'reguläre Ausdrücke' auch komplexe Suchanfragen wie:
Alle Sequenzen, die mit einem 'A' beginnen und mit einem 'Z' enden möglich sind. Darüber hinaus kann auch nach bestimmten Pegeln oder Pegelwechsel gesucht werden. Der MSB-RS485-PLUS Analyser wird über den per USB verbundenen PC versorgt und ist bei Verwendung eines Laptops oder Notebooks auch für den mobilen Einsatz geeignet.

Individual protocols
programmable in Lua

Ausgestattet mit einem leistungsfähigen Lua Interpreter ist der Analyser in der Lage alle denkbaren RS422/485 Feldbus Protokolle zu parsen und die Telegramme in vielfältiger Form darzustellen. Dies gilt auch für eigene oder proprietäre Protokolle. Die folgende Protokoll Vorlagen sind bereits enthalten:

- 3964(R)
- BACnet
- DF1
- DNP3
- Executive (Verkaufsautomaten)
- IEC60870-5-101
- IEC60870-5-103
- MDB/IPC
- Modbus ASCII & RTU
- MOVILINK
- NMEA
- P-Net
- Profibus
- SAE-J1587
- SAE-J1922
- SMA-NET
- SSI (synchron)
- USS

Weitere sind in Vorbereitung.

2.3 Anwendungsgebiete

Der Analysator MSB-RS485-PLUS findet seine Verwendung bei der Aufzeichnung und Auswertung von asynchronen Datenübertragung basierend auf der EIA-422/485 Spezifikation. Dies schließt 2-Draht, 4-Draht und EIA-422 Vollduplex Verbindungen inklusive Handshake mit ein.
Die hohe zeitliche Auflösung von 10 Nanosekunden erlaubt eine extrem genaue Timing Analyse der Kommunikation auch bei schnellen Bus Systemen (Profibus, CAN Bus), sowie detaillierte Aussagen über Reaktionszeiten in EIA-422/485 Protokollen.
Durch die aktive Abtastung aller Leitungspaare können auch Buskonflikte, hervorgerufen durch falsches Umsetzen des TriState Zustandes eindeutig erkannt werden. Dies gilt auch, wenn der Datenbus (Sende- und/oder Empfangsleitungen) per pull up/pull down auf ein Idle Pegel von 200mV eingestellt sind.

Typische Anwendungsgebiete sind:

- Industrielle Schnittstellen Applikationen
- Fabrik-Automation
- Industrie-Netzwerke
- Gebäudetechnik
- Maschinensteuerung/Automatisierungstechnik
- Embedded Devices

3

Features & Benefits

Der MSB-RS485-PLUS Analyzer bietet alle nötigen Eigenschaften für eine effektive Untersuchung von asynchronen EIA-422/485 Verbindungen. Insbesondere für Debugging, Tests und 'Reverse Engineering'.

- **Simultane Erfassung aller Leitungen durch externe Hardware :** Exakte Messung des Zeitverhaltens aller EIA-422/485 Signale mit einer Genauigkeit von 1μsec durch maximale Abtastung mit 200 MHz unabhängig vom Betriebssystem. Keine Verfälschung von Zeitpunkt und Reihenfolge durch verschleppte oder nicht bearbeitete System Interrupts (Software Lösungen).
- **Beliebige Baudraten mit FLEXUART:** Hochpräzises Setzen und Messen auch nicht standardisierter Baudraten im Bereich von 1Baud bis 20 MBaud mit 0.1% Genauigkeit. Aufzeichnung und Analyse mit beliebigen, auch unüblichen, Baudraten. Detektion von nicht synchronen oder driftenden Baudraten zwischen Sender und Empfänger.
- **Automatische Protokoll Erkennung :** Einfache Kontrolle oder Analyse der Datenverbindung bei unbekannten Verbindungsparametern.
- **Unterstützt 9 Bit Datenlänge :** Aufzeichnung und Analyse auch von Datenprotokollen mit 9 Bit Datenwortlänge.
- **Analyse von synchronen Übertragungen :** Bereits integriert ist die Unterstützung für das SSI (Synchronous Serial Interface) Protokoll. Weitere sind in Vorbereitung.
- **Scope ähnliche Darstellung der Datenleitungen:** Gleichzeitige Anzeige sowohl der logischen Signalpegel als auch der übertragenen Daten. Dadurch leichte Fehleranalyse bei Übertragungsfehlern, z.B. ungenauen Bitraten (Jittern) oder falschen Datenformate. Ausmessen der realen Signale mit integriertem Baudrate Lineal.
- **Segment-Analyse :** Richtungsabhängige Analyse einzelner Bus Segmente bzw. Bus Teilnehmer und damit Einkreisen fehlerhafter Sende-Teilnehmer durch transparente Bus-Auftrennung.
- **2 zusätzliche Digital-Ein-/Ausgänge :** Aufnahme zusätzlicher Steuerleitungen (Hilfssignale) oder Ausgabe der Bus-Richtung bzw. des Bus Status (aktiv/inaktiv) zur Triggerung eines externen Meßgerätes (Scope).
- **Protokoll Templates :** Definieren Sie eigene Regeln wie Ihre Daten dargestellt werden sollen und visualisieren Sie beliebige Protokolle.

- **Datenanalyse in Echtzeit :** Untersuchung der Verbindung bereits während der Aufzeichnung.
- **Detektion ungültiger Leitungspegel :** Erkennen von offenen Leitungen, Tri-State Zustände der Datentreiber und Buskonflikten.
- **Framing, Parity, Break Erkennung :** Direkte Analyse der Fehlerbedingungen und der Reaktionen der Endgeräte darauf.
- **Mustersuche mit regulären Ausdrücken :** Ermöglicht die Suche nach beliebigen Datensequenzen mit 'wild card' Zeichen (Platzhalter) sowie nach Zeitabständen und Pausen zwischen Zeichen(ketten).
- **Integrierter LevelFinder :** Findet beliebige statischen Pegel, Pegelwechsel sowie Fehlerzuständen. Kombiniert mit dem Auftreten bestimmter Datenbytes ist es ein wertvolles Werkzeug zur Analyze von Hardware Protokollen.
- **Skriptsprache Lua:** Zur Definition, Visualisierung, Verrechnung (Checksum Test) sowie Umwandlung der aufgenommenen Daten.
- **MultiViewKonzept :** Simultane Analyse der Aufzeichnung an verschiedenen Positionen und mit unterschiedlichen Ansichten. Z.B. übertragene Daten und ihr physikalischer Signalpegel oder um verschiedene Ausschnitte zu vergleichen.
- **Copy And Paste :** Einfaches Kopieren von aufgezeichneten Protokollen oder Datensequenzen in andere Anwendungen.
- **Datenexport als CSV:** Bearbeitung der aufgezeichneten Daten in Microsoft Excel oder anderen Tabellenkalkulations Programmen.
- **Direkte Anzeige des Datenstroms durch grüne LEDs :** Zusätzliche Kontrolle des Datenaufkommens, schnelle Kontrolle auf korrekte Datenverbindung.
- **Zukunftssicher durch moderne FPGA Technik :** Integrierte Gate Array Technologie des neuesten Stands ermöglicht eine permanente Weiterentwicklung und Anpassung des Analysers an die verschiedensten Applikationen. Die Aktualisierung der Firmware erfolgt dabei einfach beim Starten der Software.
- **Nanosekunden genaue Synchronisation mehrerer Analyser :** Durch die eingebaute Link Buchse ist die zeitgleiche Datenaufzeichnung zweier unterschiedlicher RS232/RS422/485 Verbindungen möglich. Die Genauigkeit beträgt 10 Nanosekunden.
- **Multiplattform Support :** Die MSB-RS485-PLUS Software wird als 'Native Binary' für Microsoft Windows®und Linux geliefert. Keine Emulation, keine zusätzlichen Bibliotheken oder Installationen wie .NET®oder Java®.
- **MultiLanguage Support :** Deutsche und englische Sprachunterstützung. Die Auswahl erfolgt automatisch entsprechend den OS Vorgaben, kann aber auch vorgegeben werden.
- **Multi-Process Architektur :** Aufteilung der verschiedenen Funktionen in unterschiedliche Programme/Prozesse garantiert größte Datensicherheit während der Aufzeichnung und bestmögliche Anpassung (Skalierung) an die System Resourcen.
- **Kompaktes Gehäuse mit USB Anschluß :** Kein zusätzliches Netzteil nötig. Mobiler Einsatz auch mit Laptop.

4 Spezifikationen

Allgemein

Protokoll Analysator zur Aufnahme und Analyse von asynchronen EIA-422/485 Verbindungen (2-Draht, 4-Draht, halb- und vollduplex) durch paralleles Abtasten mit maximal 200 MHz. Exakte Messung des Zeitverhaltens aller Daten- und Bussignale mit einer Genauigkeit von 10 Nanosekunden.
Dekodierung der Busleitungen T+, T–, R+, R– inklusive Tri-State Zustand mit beliebiger Baudrate im Bereich 1 Baud bis 20 MBaud.
Automatisches Erkennen der Baudrate und des Protokolls.
Ausgabe von Busstatus, Busrichtung als Digitalsignal.

RS485 Messung

- **Beliebige Baudraten mit FlexUart**
  Hochpräzises Setzen und Messen auch nicht standardisierter Baudraten im Bereich von 1 Baud bis 20 MBaud mit 0.1% Genauigkeit.
- **Datenformate**
  Parameter für serielle Datenübertragung: 5 bis 9 Datenbits; Parität aus, gerade, ungerade, fest 0, fest 1
- **Logischer Leitungsstatus**
  Logischer Pegel (A-B): 1 (V+), 0 (V-), ungültig ($-0.7V < In < +0.7V$)
- **Zeitauflösung**
  Alle Leitungsänderungen werden per Hardware exakt mit 10 Nanosekunden Genauigkeit erfasst - unabhängig vom Betriebssystem.

EIA-422/485 Anschlüsse

- **Signalpegel**
  Standard EIA-422/485 Pegel ± 0.2 V bis ± 12V, ESD geschützte Eingänge 12kOhm, Common Mode ± 7V. Erkennung Tri-State Pegel bei Differenzsignalen unter ca. ± 0.7V
- **Bus Anschluss**
  Anschlussstecker: 2* Phoenix MC 1,5/ 6ST-3,81 mit 2mm Schraubanschlüssen, je 6 Pins.
- **Intere Beschaltung**
  Alle Anschlüsse von Port 1 und Port 2 sind durch High Speed Treiber angeschlossen bzw. verbunden und werden automatisch entsprechend dem gewählten Anschlussmodus und der Datenrichtung geschaltet.

Zusätzliche Features

- **Hilfs-Ein/Ausgänge**
  Zwei Zusatzanschlüsse, wahlweise jeweils schaltbar zur Aufzeichnung von externen Signalen oder zur Ausgabe von Buszustandssignalen. Eingang: 0-5V, Triggerschwelle 1,65V, 10KOhm Pull up/down/off (wahlweise), Ausgang: 0/5V ca. 50mA
- **Cache**
  Interner Memory Cache zur Pufferung der Messdaten bei Aufzeichnung hoher Transferraten
- **Status LEDs**
  Mehrfarbige LEDs zur Anzeige von: Aufnahme- und Kanal/Bus Status (Pegel, Datenfluss)

Stromversorgung

Die Stromversorgung erfolgt direkt über das USB Kabel, Aufnahme ca. 250mA (USB GND entspricht EIA-485 GND).
Kein externes Netzteil nötig.

Unterstützte OS

- **Windows**
  Windows XP, Vista, Windows 7, Windows 8/8.1, Windows 10 (alle 32 und 64 Bit)
- **Linux**
  Alle Linuxe mit Kernel ab 2.4.18 und installierten Gtk2 Bibliotheken (sind Standard). (Im Zweifelsfalle können Sie einfach die Linux Version von unserer Download Seite testen). 32 und 64 Bit Systeme.

Abmessungen

- **Abmessungen**
  100mm x 60mm x 30mm (Länge, Breite, Höhe)
- **Gewicht**
  ca. 100g

Lieferumfang

- **Analyzer**
  MSB-RS485-PLUS Analyzer Gerät.
- **Anschluss-Set**
  bestehend aus:
  2* 6-polige Phoenix Schraubklemm-Anschlussstecker
  4* Abschlußwiderstände 120 falls Analyser End-Device ist
  4* Drahtbrücken für unterschiedliche Anschlussvarianten
  Schraubendreher für Phoenix Stecker
  USB Kabel zur Verbindung mit PC.
- **Software**
  CD für Windows und Linux, Anleitung als Online Hilfe und PDF Dokument in deutsch und englisch.

Voraussetzungen

- **Grafikdisplay**
  Grafikkarte und Monitor mit 1024x768 Auflösung und 16 Bit Farbtiefe oder besser.
- **Festplattenplatz**
  200 MByte freier Festplattenplatz für die Software Installation plus zusätzlicher Platz für die Aufnahmedateien.
- **Speicher (RAM)**
  256 MByte oder mehr.
- **USB Anschluss**
  Ein freier USB 2.0 Anschluss.

5 Programm Installation

Die MSB-Analyser Software steht sowohl für Microsoft Windows als auch für Linux zur Verfügung. Beide Versionen sind auf der Programm CD-ROM enthalten und werden Ihnen entsprechend Ihrem System automatisch zur Installation angeboten. Was Sie dabei im einzelnen beachten müssen, erfahren Sie in diesem Abschnitt.

Der MSB-Analyser wird per USB an den PC angeschlossen und kommuniziert mit diesem mittels eines virtuellen COM Port. Unter Microsoft Windows wird deshalb bei der Programm Installation automatisch ein entsprechender VCOM Treiber eingerichtet.
Linux Distributionen besitzen seit Kernel 2.4.18 von Haus aus ein auch für den MSB-Analyser funktionierendes Kernelmodul (`ftdi_sio`).
Die Software ist mehrsprachig (deutsch und englisch) und kann auch ohne Analysator installiert und verwendet werden, z.B. um bereits gemachte Aufnahmen auszuwerten (offline Modus). Oder wenn Sie die Fähigkeiten des Analysers am Beispiel mehrerer beigefügter Aufzeichnungen zunächst testen einmal möchten.

5.1 Installation unter Windows

Beenden Sie alle laufenden Anwendungen vor dem Einlegen der CD-ROM. Schliessen Sie den MSB-Analyser erst an, nachdem Sie die Programm Installation durchgeführt haben.

1 **Legen Sie die Installations CDROM ein.**
Der IFTOOLS Produkt Installer wird aufgerufen. Wenn der Installer nicht automatisch startet, doppelklicken Sie auf dem Desktop auf `Arbeitsplatz` bzw. öffnen diesen im Startmenü. Doppelklicken Sie das IFTOOLS-Setup-CD Icon um den IFTOOLS Produkt Installer zu starten.

2 **Produkt auswählen.**
Klicken Sie in der linken Produktauswahl auf den entsprechenden Analyser Typ (MSB-RS232 oder MSB-RS485). Starten Sie anschliessend die Software Installation inklusive des nötigen Treibers mit einem einzelnen Klick auf `Installation (Version` 7.0.2`)`. Eventuell dauert es einen Augenblick, bis der Installer auf dem Bildschirm erscheint.

3 **Software installieren.**
Setzen Sie die Installation gemäß den auf dem Bildschirm erscheinenden Hinweisen fort. Der für den Betrieb nötige Treiber wird dabei automatisch eingerichtet.

5.2 Installation unter Linux

Moderne Linux Distribution bieten i.a. den gleichen Komfort zur Programm Installation von CD-ROM. Wie bei Windows muß dieses Verhalten u.U. aber zunächst aktiviert werden. Ihr Linux System muß dazu die CD-ROM automatisch als ausführbar einhängen. Ist dies der Fall, läut die Installation im großen ganzen wie unter Windows ab.

1 **Legen Sie die Installations CDROM ein.**
Der IFTOOLS Produkt Installer wird aufgerufen. Je nach Distribution werden Sie zunächst gefragt, ob Sie das AUTORUN Feature aktivieren wollen. Beantworten Sie dies mit 'ja'. Wenn der Installer nicht automatisch startet, lesen Sie bitte den folgenden Anschnitt 'Manuelle Installation unter Linux'.

2 **Produkt auswählen.**
Klicken Sie in der linken Produktauswahl auf den entsprechenden Analyser Typ (MSB-RS232 oder MSB-RS485). Starten Sie anschliessend die Software Installation mit einem Klick auf `Installation (Version` 7.0.2`)`.

3 **Software installieren.**
Setzen Sie die Installation gemäß den auf dem Bildschirm erscheinenden Hinweisen fort. Das für den Betrieb nötige Kernelmodul ist Bestandteil aller Kernel seit Kernel Version 2.4.18 und muß deshalb nicht extra eingerichtet werden.

5.2.1 Manuelle Installation unter Linux

Sollte der IFTOOLS Produkt Installer nach einlegen der CD-ROM nicht automatisch starten, gehen Sie wie folgt vor:

1 **Öffnen Sie eine Konsole**

2 **Kopieren Sie die Installationsdatei auf Ihren Desktop**
Geben Sie dazu folgendes Kommando ein:

```
cp CDROM_PFAD/programs/msb/msb-7.0.2-linux-installer.run ~/Schreibtisch
```

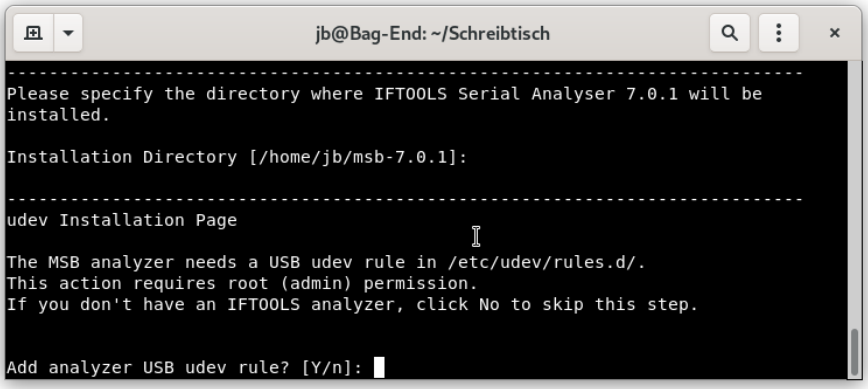
3 **Machen Sie die Installationsdatei ausführbar**
indem Sie das executable Flag setzen mit:

```
chmod +x ~/Desktop/msb-=\version\verb=-linux-installer.run
```

4 **Starten Sie die Installationsdatei**
Starten Sie die Installationsdatei per Mausklick (Doppelklick).
Alternativ können Sie den Installer auch im Textmodus starten. Dies bietet sich an, wenn der grafische Installer nicht starten sollte. Öffnen Sie dazu eine Textkonsole und geben folgendes Kommando ein:

```
~/Desktop/msb-7.0.2-linux-installer.run --mode text
```

Beachten Sie bitte!
Damit der Analyser von Ihrem Linux System korrekt erkannt wird, benötigen Sie eine sogenannte `udev` Regel. Diese wird beim Anschließen des Analysers automatisch ausgeführt und ermöglicht einen direkteren (deutlich schnelleren) Zugriff als über einen normalen (virtuellen) seriellen Port.
Diese `udev` Regel muss unter `/etc/udev/rules.d` eingetragen werden, was root Rechte voraussetzt. Der Installer frägt Sie deshalb nach Ihrem Benutzer Passwort um die `udev` Regel per sudoer einzutragen. (Bei der grafischen Installation passiert das gleiche). Im Text Mode sieht das wie folgt aus:



Es ist zwingend notwendig, diese Frage mit 'Y' zu beantworten. Andernfalls wird die `udev` Regel nicht installiert und der Analyser nicht erkannt!

5 Manuelles installieren der udev Regel

Falls bei der Installation der `udev` Regel etwas schief läuft, können Sie diese auch später auch selbst (per sudoer) installieren oder eine Person mit entsprechenden (root) Rechten fragen, dies für Sie zu tun. Wie gesagt, die Installation erfordert Schreibrechte in das `/etc/udev/rules.d` Verzeichnis.
Starten Sie dazu die Analyser Installation wie oben beschrieben und überspringen Sie Einrichtung der `udev` Regel durch Eingabe von 'n'. Nach Ende der Installation wechseln Sie in das neu eingerichtete Analyser Software Verzeichnis. I.a. finden Sie dieses als `msb-VERSION` in Ihrem Heimat/User Verzeichnis. In diesem Verzeichnis öffnen Sie ein Terminal und führen folgendes Kommando als sudoer oder alternativ als root aus:

```
sudo ./udev-install.sh
```

Sie können die Regel auch jederzeit wieder entfernen mit:

```
sudo ./udev-remove.sh
```

Denken Sie aber daran: Ohne diese Regel wird der Analyser dann nicht mehr von Ihrem System erkannt.

5.2.2 Installation für alle Benutzer

Eine System weite Installation ist genauso einfach wie die für einen einzelnen Benutzer. In diesem Fall müssen Sie lediglich den Installer als sudoer ausführen[1]:

```
sudo ~/Desktop/msb-7.0.2-linux-installer.run
```

Der Installer erkennt anhand des ausführenden Users (hier nun sudoer) dass eine System weite Installation erfolgen soll. Er schlägt Ihnen deshalb automatisch als Verzeichnis `/opt/msb-VERSION` vor.
Alternativ können Sie den Installer auch im text mode starten. Dies kann sinnvoll sein, wenn der grafische Installer nicht korrekt startet oder Sie nur Zugriff per Text Terminal haben:

```
sudo ~/Desktop/msb-7.0.2-linux-installer.run --mode text
```

5.3 Programm Updates

IFTOOLS veröffentlicht in unregelmäßigen Abständen Software Updates mit neuen Eigenschaften und Verbesserungen. Diese Updates sind für Sie kostenlos und können jederzeit unter folgender Adresse
https://iftools.com/download herunter geladen werden.
Bei den Updates handelt es sich um komplette Programminstallationen, die in der Windows Version auch den jeweils aktuellen Treiber enthalten. Updates können parallel zu Ihrer aktuellen MSB-Analyser Programm Version installiert werden. Windows User müssen dazu lediglich die Update (Installer) Datei ausführen.
Unter Linux ist es erforderlich, die Datei zunächst ausführbar zu machen und sie dann, wie oben beschrieben, zu starten.

[1] Sie können dies auch als root tun, aber wir empfehlen explizit die Installation per sudoer.

6

Anschluß des Analysers

Wie schließe ich den Analyser an meinen PC an? Wie binde ich ihn in die zu überwachende Verbindung ein? Was bedeuten die LED's? Diese und andere Fragen beantwortet der folgende Abschnitt.

Die EIA-422/485 Spezifikationen machen im Gegensatz z.B. zur EIA-232 keine Angaben über die Steckerbelegung. Anschlüsse und Steckverbinder von EIA-422/485 Verbindungen sind deshalb stets Applikations abhängig.
Um eine einfache Adaptierung des Analysers an die verschiedensten Bussysteme zu ermöglichen ist die MSB-RS485-PLUS mit zwei 6poligen Buchsen zur Aufnahme von jeweils einem Phoenix Schraubklemmstecker ausgestattet. Ein entsprechendes Anschluß-Kit inklusive Schraubenzieher ist beigelegt.
Über die Buchse PC USB wird der Analyser mit einem freien USB Anschluß Ihres PC's verbunden, auf dem die Analyser Software installiert ist. Die Spannungsversorgung erfolgt ebenfalls über das USB Kabel, so dass keine zusätzliche Versorgung benötigt wird und Sie den Analyser auch im mobilen Einsatz mit einem Laptop verwenden können. Die Stromaufnahme liegt bei ca. 250 mA.

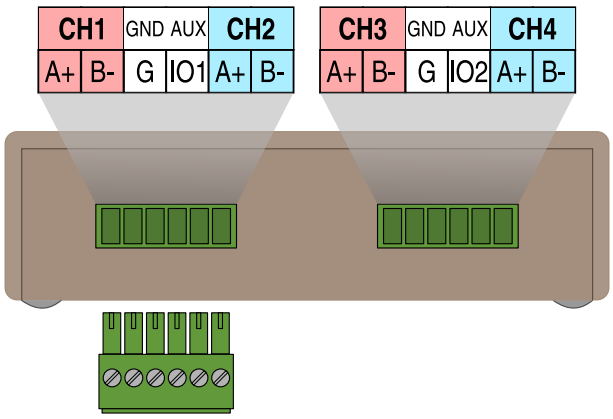
Der Analyzer wird an beliebiger Stelle an den vorhandenen Bus angeschlossen. Durch zwei variabel einsetzbare Daten Dekoder können sowohl Voll-Duplex Bus Systems als auch zwei unabhängige 2-Draht Busse (Systeme mit redundantem Design) analysiert werden.

Und der Analyser kann in den Bus eingeschleift werden. Er wird damit zum 'Router' zwischen zwei beliebigen Bus Segmenten. Die durch ihn laufenden Daten werden richtungsabhängig erfasst und dem jeweiligen Segment eindeutig zugeordnet. Unschätzbar bei Protokollen ohne Richtungsinformation!

Der Analyser besitzt zudem zwei zusätzliche digitale IO-Kanäle welche als weitere Hilfseingänge zur Aufzeichnung von Logiksignalen (wahlweise mit internem pull-up bzw. pull down) oder als Ausgang für Status Informationen wie Bus Richtung oder Bus Gültigkeit dienen können.

6.1 Busanschlüsse

Die MSB-RS485-PLUS bietet zwei steckbare 6-Pin Phoenix Schraubklemm Anschlüsse zur einfachen Verbindung des Analysers mit beliebigen Feldbus Systemen. Dies umfasst insgesamt vier RS485 Bus (Kanal) Eingänge sowie zwei digitale IO Ports.
Die Bedeutung der einzelnen Kanäle sowie IO Ein/Ausgänge hängt dabei vom gewählten Busanschluss ab und wird im folgenden genauer erläutert.



Phoenix MC 1,5 / 6ST-3,81

Beachten Sie das die neuen Anschlüsse eine andere Pin Belegung (verbesserte Signal-Rausch-Unterdrückung) und etwas größere Abmessungen besitzen als die des MSB-RS485 Analysers. Der neue Stecker erlaubt das Anklemmen von dickeren Kabeldurchmessern (2mm) wie sie bei Feldbus Installationen öfters zu finden sind. Zudem wird durch die unterschiedlichen Größen einer Verwechslung mit den alten Steckern und Beschädigungen durch die inkompatible Steckerbelegung vermieden.

Der neue Stecker kann zudem mit einem passenden Gehäuse versehen werden, z.B. um weitere Elektronik für externe Signalumsetzer hinzuzufügen.

6.2 Anschluss der Bus Leitungen

Der Bus wird entsprechend den Anschlussschaltbildern in den nächsten Kapiteln angeschlossen.
Grundsätzlich sollten die folgenden Hinweise beachtet werden.
Schließen Sie die positive Busleitung an $A+$ und die negative Busleitung an $B-$ des entsprechenden Analyser Kanals an. Manchmal haben die Busanschlüsse andere Bezeichnungen wie 'D', 'TX/RX' oder 'Data', aber die Polarität zwischen Analyser und Bus muss übereinstimmen.
Wir empfehlen eindringlich auch die Busmasse an den Analyzer anzuschließen, selbst wenn der Bus ohne Masseanschluss arbeitet. Das sollte generell für den gesamten Bus beherzigt werden. Der Grund dafür ist, dass durch externe Störungen induzierte Ausgleichsströme durch die Datenleitungen fließen anstelle der dafür vorgesehenen Masseleitung. Das führt zu Spannungsverschiebungen und Spitzen auf den Datenleitungen. Sporadische Übertragungsfehler und im schlimmsten Fall zerstörte Busgeräte können das Ergebnis sein.
Beachten Sie bitte außerdem, dass die Masse und Datenleitungen auf gleichem Potential sind wie die Bus Leitungen, es ist keine Potentialtrennung integriert. Wenn Sie eine elektrische Isolatione zwischen dem Bus (Analzer) und dem PC benötigen verwenden Sie entweder einen potentialfreien Laptop oder einen passenden USB Isolator.
Die nachfolgende Tabelle listet die geläufigsten Leitungsbezeichnungen auf:

EIA-485	MSB-RS485-PLUS	Alternative Bezeichnungen
A+	$A+$	TX+, TX+/RX+, D+, Data+, (Y)
B–	$B-$	TX–, TX–/RX–, D-, Data-, (G)
A+	$A+$	RX+[1]
B–	$B-$	RX-[2]

1,2 Nur bei vollduplex 4-Draht Systemen.

6.3 Interne Signalverarbeitung

Der MSB-RS485-PLUS Analyzer besitzt vier Differenzsignal-Eingänge CH1...CH4 die simultan abgetastet und aufgezeichnet werden. Für jeden Kanal steht das physikalische Signal in Form der Leitungs-/Pegeländerungen (Logikpegel Darstellung) zur Verfügung.
Zwei integrierte UARTs sorgen für die Dekodierung des seriellen Datenstromes in einzelne Datenbytes. Diese werden je nach Art des Bus-Anschlusses automatisch mit zwei der vier Differenzsignal-Eingänge CH1...CH4 verschaltet.

Durch dieses 'variable' Zuschalten der UARTs ist eine Vielzahl von Anschluss- und Analyse-Möglichkeiten gewährleistet.
So erlaubt der MSB-RS485-PLUS Analysator neben einem reinen 'Abgreifen' der Busleitungen auch den Bus 'durch' den Analyzer hindurch zu leiten, indem der Bus aufgetrennt und die 'Enden' mit CH1 und CH2 verbunden werden. Beide UARTs übernehmen dabei die getrennte und unabhängige Dekodierung der beiden Busabschnitte.

Ein kombiniertes Verfahren in dem ein Bus 'aufgetrennt', ein zweiter 'abgegriffen' wird, ermöglicht zudem das Herausfiltern einzelner Busteilnehmer auch bei Vollduplex 4-Draht Verbindungen.

Zwei zusätzlich generierte Tri-State Signale unterstützen diese Art der Analyse indem sie Informationen über die Gültigkeit und Richtung der Daten auf dem Bus sowie das 'gemeinsame' (da durch geschleifte) Datensignal liefern.
Die beiden Tri-State Signale sind in der folgenden Tabelle aufgelistet:

Bezeichnung	Mark (1)	Space (0)	ungültig
Bus-Dir (Datenrichtung)	CH2 → CH1	CH1 → CH2	Bus ungetrieben
Bus-Signal	durchgeleitetes Bus-(Daten)-Signal	durchgeleitetes Bus-(Daten)-Signal	keine Daten

6.4 Digitale Ein/Ausgänge

Der MSB-RS485-PLUS Analyser besitzt 2 zusätzliche Digitale IO Kanäle, die wahlweise als Hilfseingang (zur Aufzeichnung zusätzlicher Logiksignale) oder als Ausgang geschaltet werden können.
Letzteres erlaubt die Ausgabe der Busrichtung sowie Gültigkeit des Busses entweder einzelner Segmente oder durchgeschleifter Busleitungen.
Folgende Einstellungen sind getrennt für beide IO Kanäle möglich:

IO-Type	Beschreibung
Eingang	Eingang mit Pull down Widerstand
Eingang	Eingang mit Pull up Widerstand
Ausgang	Ausgang statisch 0
Ausgang	Ausgang statisch 1
Ausgang	Ausgabe der Bus-Richtung zwischen CH1 und CH2 (Segment Analyse): 0 : CH1 → CH2 1 : CH2 → CH1
Ausgang	Bus-Aktivität des Bus durch CH1 und CH2 (Segment Analyse): 0 : inaktiv (tri-state), 1 : aktiv
Ausgang	Bus-Aktivität CH1 1 : aktiv, 0 : inaktiv (tri-state)
Ausgang	Bus-Aktivität CH2 1 : aktiv, 0 : inaktiv (tri-state)
Ausgang	Bus-Aktivität CH3 1 : aktiv, 0 : inaktiv (tri-state)
Ausgang	Bus-Aktivität CH4 1 : aktiv, 0 : inaktiv (tri-state)
Ausgang	Fehler Ausgabe, 1-Bit breiter Pulse bei Frame, Parity oder Break
Ausgang	5V/50mA zur Versorgung externer Adapter

6.5 Bus Anschluss/Abgriff

EIA-422 Systeme sind i.a. als vollduplex Punkt-zu-Punkt Verbindungen ausgeführt, evtl. mit zusätzlichen Leitungspaaren für ein Hardware Handshake.
Wogegen eine EIA-485 Verbindung prinzipiell entweder als Multi-Master fähiges 2-Draht-System (halbduplex) oder als vollduplex 4-Draht-System auf Master-Slave Basis aufgebaut sein kann.

Allen seriellen Bussystemen ist die Tatsache zu eigen, daß die Signale auf unterster Ebene keine Rückschlüsse auf Richtung und Herkunft erlauben.
Der MSB-RS485-PLUS Analyser bietet deshalb zusätzlich die Möglichkeit zur Auftrennung der zu untersuchenden Verbindung in einzelne Segmente und/oder Teilnehmer.
Mit dieser sogenannten 'Segment-Analyse' können gezielt die gesendeten Daten eines einzelnen (oder mehrerer) Teilnehmer unabhängig vom restlichen Bus aufgenommen und direkt (ohne Einbeziehung des Protokolles) zugeordnet werden.

Ein EIA-485 Bus kann in zwei Abschnitte aufgetrennt und jeder einzeln unabhängig dekodiert werden, Sende- und Empfangsleitungen einer Vollduplex Verbindung (EIA-422/485) parallel aufgenommen oder im Mischbetrieb die Sendeleitung der Teilnehmer getrennt und die Sendeleitung des Master (Controllers) parallel aufgezeichnet werden.

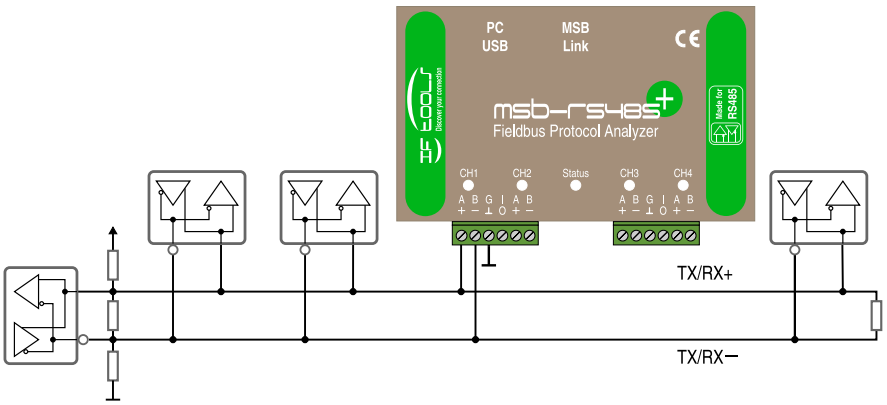
Um Anschluss und Zuordnung der einzelnen Signale so einfach wie möglich zu gestalten, bietet der Analyser eine Auswahl verschiedener Kombinationen aus Bussystem und Abgriff, (die sogenannte Anschlussart oder kurz Wiring), die mit den folgenden Anschaltungsvarianten übereinstimmen.

6.6 Abgriff 2-Draht System

Bei einem 2-Draht System wird der Bus bzw. das Leitungspaar einfach mit den entsprechenden Anschlüssen des Analysers an CH1 (Kanal 1) verbunden. Die übrigen Kanäle CH2...CH4 bleiben ungenutzt.
Dabei wird die invertierte Leitung des Busses an den $B-$ Eingang von CH1, die nicht invertierte mit den $A+$ Eingang von CH1 angeschlossen.

Die Bezeichnung 2-Draht System impliziert zwar nur die Verwendung des einen Leitungspaares, allerdings sollte ein korrekter Anschluß der Masse vorgesehen werden. Sollte der zu untersuchende Bus eine Signalmasse besitzen, verbinden Sie diese deshalb ebenfalls mit dem entsprechenden Anschluss G (Ground).
Bei einem einfachen Abgriff müssen Sie sich keine Gedanken über eine etwaige Terminierung machen.

Der Analyzer zeichnet bei dieser Anschlussart alle übertragenen Daten unabhängig von Quelle oder Richtung auf. Um etwaige Aussagen über den Sender der Daten zu bekommen, müssen Sie aber zusätzlich das entsprechende Protokoll kennen.

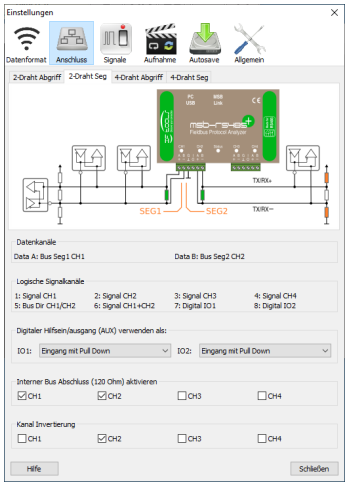


2-Draht Abgriff
für Halbduplex Bus
(Modbus, ProfiBus)

6.7 Segment Analyse 2-Draht System

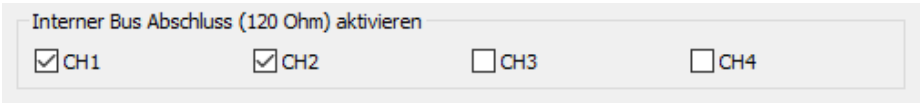
Im Gegensatz zum einfachen 'Abgriff' des Datensignals wird bei der Segment-Analyse die MSB-RS485-PLUS 'in' den Bus eingeschleift. Der Bus wird dadurch in einen Abschnitt (ein Segment) rechts und links dem Analyser aufgeteilt.
Obwohl diese Anschlussart etwas aufwendiger ist, hat sie doch einige Vorteile gegenüber dem einfachen Abgriff.

Der Analyser wird dadurch zur Schnittstelle zwischen zwei beliebigen Bussegmenten. Die durch diese Schnittstelle laufenden Daten werden richtungsabhängig erfasst und können dem jeweiligen Segment eindeutig zugeordnet werden. Besteht ein Segment nur aus einem einzelnen Busteilnehmer können damit gezielt die von diesem Teilnehmer gesendeten Daten von der restlichen Bus-Kommunikation unterschieden werden - und zwar auch ohne Kenntnis des verwendeten Protokolles.

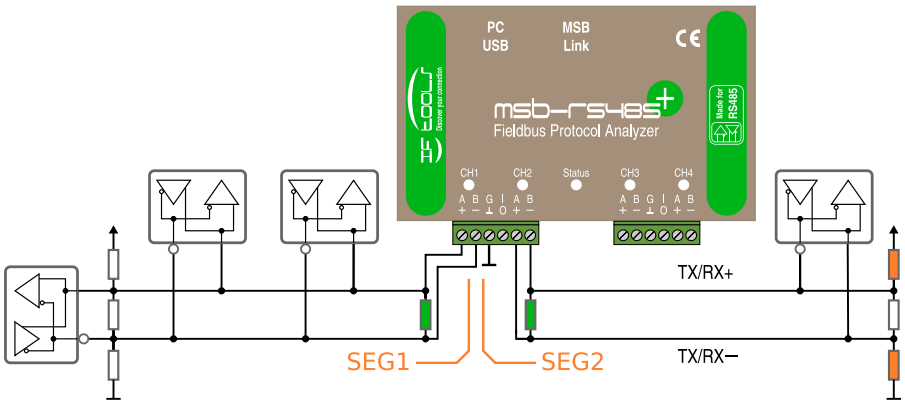


Bus Anschluss Dialog
Bus Anschluss und
Terminierung

Trennen Sie dazu den Bus am gewünschten Punkt auf und verbinden Sie das Leitungspaar des einen Segments mit den Anschlüssen des ersten Kanals CH1 des Analysers ($A+$, $B-$).
Das zweite Bussegment schließen Sie entsprechend an Kanal 2 an (CH2, $A+$ und $B-$).
Der Bus besteht jetzt aus zwei Segmenten. Beachten Sie deshalb, daß Sie evtl. die 'neuen' Enden des Busses entsprechend terminieren müssen. Der MSB-RS485-PLUS Analyser integriert bereits die passenden Abschlusswiderstände (im folgenden Bild grün dargestellt). Sie müssen diese lediglich im Bus Anschluss Dialog aktivieren.



Gleiches gilt für vorhandene Pullup-Widerstände. Pullups sind i.a. nur an einem Ende des Busses angebracht und im Wert System abhängig. Der Analyser kann diese deshalb nicht selbst zur Verfügung stellen und Sie müssen ggf. geeignete Widerstände abhängig von Ihrer Applikation nachrüsten (im Bild orange eingezeichnet).



2-Draht Segment Analyse
für Halbduplex Bus (Modbus, ProfiBus)

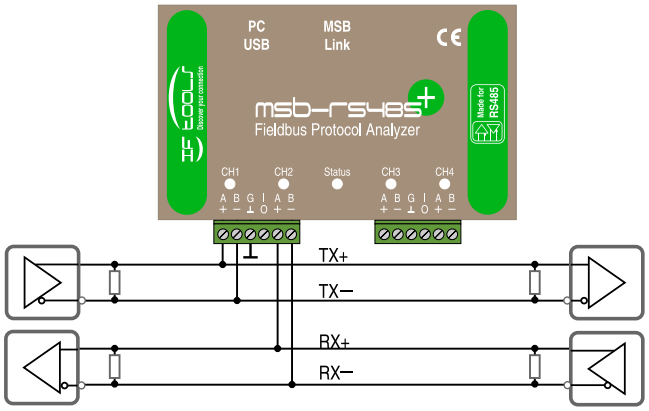
Der Analyser zeichnet die Daten beider Segmente unabhängig voneinander auf, wobei die Daten in beide Richtungen transparent durchgeleitet werden. Daten des einen Segments werden als von CH1 kommend und intern mit A, Daten des anderen Segments oder Teilnehmers an CH2 mit B gekennzeichnet (A und B entsprechen einfach zwei verschiedenen Datenquellen, die weder einem Leitungspaar noch einem speziellen Eingangskanal zugewiesen sind).

6.8 Abgriff 4-Draht System

Bei Punkt-zu-Punkt Verbindungen, wie sie z.B. EIA-422 Verbindungen zur Überbrückung größerer Entfernungen verwenden, sind nur zwei Teilnehmer vorhanden, die über zwei getrennte Sende- und Empfangskanäle kommunizieren. Ein einfacher Abgriff zur Aufnahme der Kommunikation ist daher völlig ausreichend und gewährleistet auch die Aufzeichnung der Datenrichtung.

Diese Anschlußart eignet sich ebenfalls zur Aufzeichnung/Analyse von Vollduplex EIA-485 Verbindungen (wie z.B. Din MessBus) wenn Sie keinen Busteilnehmer gesondert betrachten wollen und die Daten den Teilnehmern per Protokoll zuordnen können.

Verbinden Sie dazu einfach das Sendeleitungspaar $A+$, $B-$ mit den Anschlüssen von CH1 und das Empfangsleitungspaar mit den Anschlüssen $A+$, $B-$ von CH2.



Abgriff 4-Draht
Vollduplex EIA-422/485 (Din Messbus)

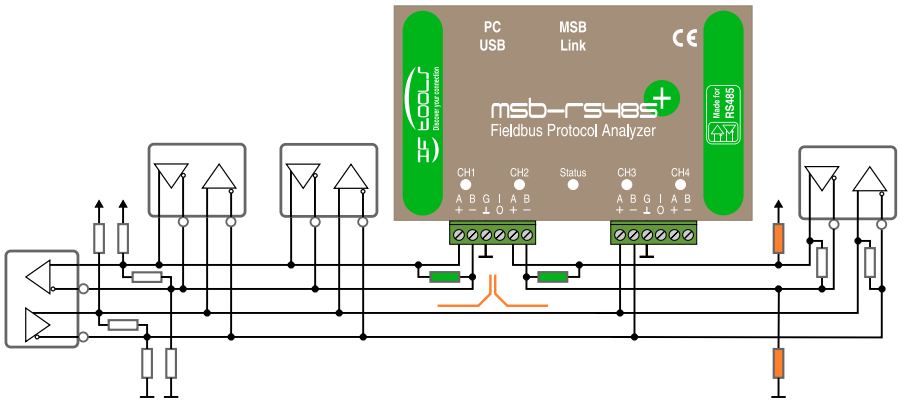
Alle gesendeten Daten werden dabei als von CH1 kommend mit A gekennzeichnet, die Daten aufgenommen an CH2 mit B.

6.9 Segment Analyse 4-Draht System

Bussysteme mit Vollduplex 4-Draht Verbindung (wie z.B. Din-Messbus) verwenden ebenfalls getrennte Sende- und Empfangskanäle.
Während der Master über einen Kanal (Masterbus) mit den Empfängern (Slaves) der Busteilnehmer verbunden ist, senden diese ihre Antwort über den anderen Kanal an den Master zurück. Um die gesendeten Daten des Master zu erfassen reicht deshalb ein einfacher Abgriff des Masterbus aus.
Dagegen 'teilen' sich die Slaves einen Kanal um ihre Antwort an den Master zurück zu senden. Mittels Segment-Analyse kann hier gezielt ein einzelner Busteilnehmer herausgetrennt und dessen Kommunikation mit dem Master aufgezeichnet und analysiert werden.

Analog zur 2-Draht Segmentanalyse müssen Sie dazu die Empfangsleitung an der gewünschten Stelle auftrennen.
Die beiden Bussegmente der Empfangsleitung werden dazu jeweils mit $A+$ und $B-$ von CH1 und CH2 verbunden. Auch hier gilt: Da die Auftrennung zwei unabhängige Bussegmente hinterlässt müssen Sie diese ggf. entsprechend terminieren. Der MSB-RS485-PLUS Analyser hat diese bereits 'an Bord' und Sie müssen diese nur noch im Bus Anschluss Dialog aktivieren. Zusätzlich sind evtl vorhandene Pullups zu berücksichtigen, siehe Abschnitt 6.7.
Der Anschluß des Masterbus erfolgt als einfacher Abgriff an CH3. Eine Terminierung ist hier unnötig.

4-Draht Segment Analyse
Vollduplex EIA-485 (Din Messbus)



In dieser Konfiguration werden alle von den Slaves gesendeten Daten an CH1 und CH2 aufgenommen und intern als A, die Daten des Masters an CH3 als B gekennzeichnet.
Die Zuordnung der Daten zu einem Segment bzw. bestimmten Busteilnehmer (CH1 oder CH2) erfolgt über ein zusätzliches Bus-Richtungs-Signal (bus direction). Dieses Signal kann bei der Darstellung explizit ausgewertet und damit der Sender entsprechend visualisiert werden.

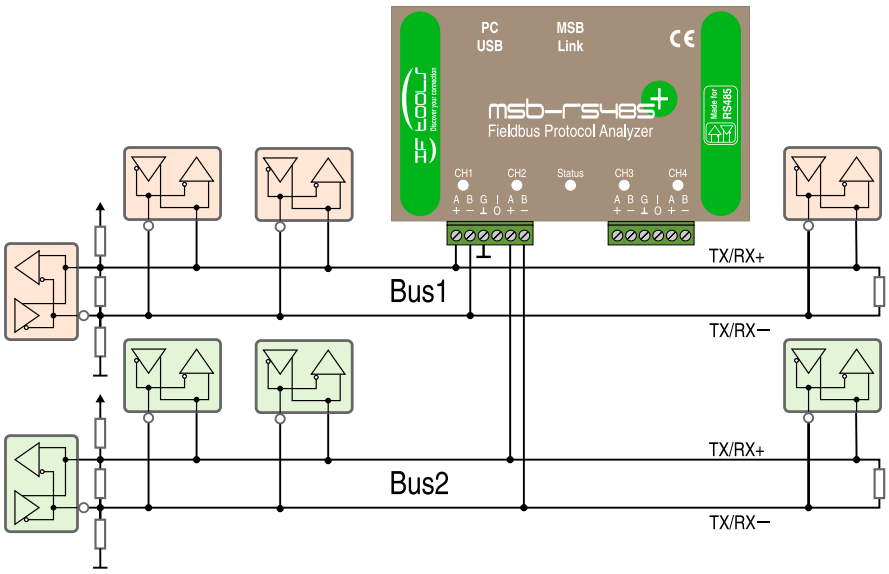
6.10 Analyse zweier unabhängiger Halb-Duplex Bus Systems

Dies ist ein spezieller Anwendungsfall des 4-Draht Abgriff wie in Abschnitt 6.8 beschrieben. Aus Sicht des MSB-RS485-PLUS Analysers spielt es dabei keine Rolle, ob Sie einen Voll-Duplex Bus an Kanal 1 und 2 angeschlossen haben, oder zwei unabhängige Halb-Duplex Bus Systeme. Die Anschluss Einstellungen sind identisch.

Die im MSB-RS485-PLUS Analyser integrierten beiden FLEXUARTs zur Umwandlung des Datenbits Stromes können komplett unterschiedlich parametrisiert werden [3]. Dies erlaubt nicht nur die Analyse von asynchronen Übertragungen mit verschiedenen Baudraten. Vielmehr können asynchrone und sogar synchrone Bussysteme (mit Manchester Kodierung, z.B. InterBus) kombiniert und gleichzeitig analysiert werden.

Die Aufnahme zweier unabhängiger Bus Systeme bietet sich u.a. bei Anwendungen mit redundanten Bus Systemen ab. Aber insbesondere auch zur Analyse bzw. Prüfung von Protokoll Umsetzern, Routern und ähnlichen Applikationen, bei denen Ein- und Ausgangsprotokoll bzw. Bus komplett unterschiedlich sind. Die Möglichkeit, zwei Halb-Duplex Busse mit verschiedenen Protokollen aufzunehmen, kann in mancher Applikation sogar den Einsatz zweier synchronisierter Analyser ersetzen.

Das folgende Bild zeigt den Anschluss des Analysers an zwei unabhängige Halb-Duplex Bussysteme. Beide sind für sich korrekt terminiert, so dass weitere Terminierungs- bzw. Pull-Up/Down Widerstände entfallen können.



Abgriff von zwei 2-Draht Bus Systemen
Analyse eines redundanten Busses oder von zwei unabhängigen Bus Systemen

[3] Dieses Feature wird in einer zukünftigen Version implementiert und ist aktuell noch nicht verfügbar

6.11 Signalzuordnung

Der MSB-RS485-PLUS Analysator besitzt 10 sogenannte Anzeigekanäle zur Visualisierung der aufgenommenen Informationen.
Zwei Datenkanäle dienen zur Anzeige der von den beiden UARTs generierten Bus-Daten (Daten A und B).
Weitere 8 Logikkanäle werden zur Anzeige des Tri-State Pegelverlauf der Differenzsignal Eingänge CH1...CH4 als auch zur Darstellung von Busaktivität und/oder Busrichtung sowie der beiden Digitaleingänge verwendet.
Die Zuordnung der einzelnen Anzeigekanäle variiert dabei leicht je nach gewählter Anschlussart.

Graue Einträge kennzeichnen Signale, die zwar prinzipiell vom Analyser aufgenommen werden, aber in der jeweiligen Anschlussart nicht verwendet werden. Nicht desto trotz können Sie diese zur Aufzeichnung zusätzlicher Busleitungen wie z.B. Handshake Signalen benutzen.

Die nach folgende Tabelle gibt Ihnen einen Überblick über die zur Verfügung stehenden Signalinformationen. Allerdings müssen Sie diese Tabelle bei der Auswertung nicht zu Rate ziehen. Die Analyser Software benennt alle Anzeigekanäle automatisch nach Anschlussart.

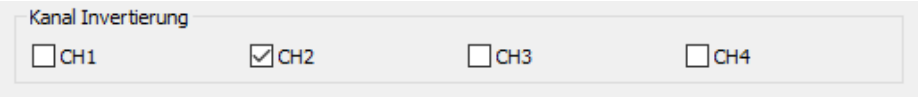
Anzeige-kanal[1]	2-Draht Abg	2-Draht Seg	4-Draht Abg	4-Draht Seg
Daten A (Data A)	Daten von Bus an CH1	Daten von Bussegment an CH1	Daten von Bus an CH1	Daten von Bus-segmenten an CH1 + CH2
Daten B (Data B)	Daten von Bus an CH2	Daten von Bussegment an CH2	Daten von Bus an CH2	Daten von Masterbus an CH3
Signal 1 (CH1)	Logiksignal an CH1	Logiksignal an CH1	Logiksignal an CH1	Logiksignal an CH1
Signal 2 (CH2)	Logiksignal an CH2	Logiksignal an CH2	Logiksignal an CH2	Logiksignal an CH2
Signal 3 (CH3)	Logiksignal an CH3	Logiksignal an CH3	Logiksignal an CH3	Logiksignal an CH3
Signal 4 (CH4)	Logiksignal an CH4	Logiksignal an CH4	Logiksignal an CH4	Logiksignal an CH4
Signal 5 (BDIR)	unbenutzt	Datenrichtung CH1 ⇔ CH2	unbenutzt	Datenrichtung CH1 ⇔ CH2
Signal 6 (BSIG)	unbenutzt	Logiksignal CH1 + CH2	unbenutzt	Logiksignal CH1 + CH2
Signal 7 (IO1)	IO1	IO1	IO1	IO1

Signal 8 (IO2)	IO2	IO2	IO2	IO2

[1] In Klammern die im Programm angezeigten Signalnamen

6.12 Kanal Invertierung

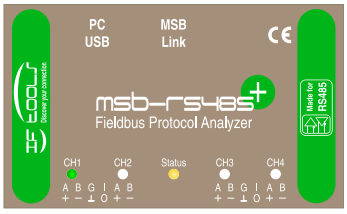
Manchmal stellen Sie vertauschte Datenleitungen erst dann fest, wenn Sie alle Vorbereitungen für eine Feldbus Aufzeichnung bereits abgeschlossen haben. Eine nachträgliche Korrektur ist - abhängig vom Anschluss - oftmals recht aufwendig.
Der Analyser erleichtert Ihnen in einem solchen Fall die Arbeit. Im Bus Anschluss Dialog können Sie jeden einzelnen Eingangskanal, falls nötig, individuell per Klick invertieren. Siehe nachfolgendes Bild.



Ein invertierter Kanal entspricht einem echtem physikalischen Tausch der Datenleitungen. Ein positiver Signalpegel wird negativ und umgekehrt. Dies gilt auch für logische Signalpegel. Sie können den Effekt im Signalmonitor betrachten.

6.13 Leuchtdioden

Der MSB-RS485-PLUS Analyser besitzt 5 mehrfarbige Leuchtdioden (LED) zur Anzeige seines Betriebszustandes und des Zustandes der Datenaufnahme. Die Leuchtdioden für die Datenaufnahme sind den jeweiligen Eingangskanälen der Phoenix Stecker zugeordnet. Sie sitzen über den beiden 6 poligen Phoenix Buchsen und sind mit CH1 und CH2 bzw. CH3 und CH4 bezeichnet.
Die Status LED ist in der Mitte und dient zur Signalisierung des Betriebs- und Aufnahmestatus.



Kontrol LEDs
für Status und Aufnahme Kontrolle

6.13.1 Bus Kanal LEDs

Die Kanal (Channel) LEDs geben Aufschluss über den aktuellen Zustand des angeschlossenen RS422/485 Bus.

LED Farbe	Bedeutung
● AUS	Kein angeschlossenes Signal oder Bus inaktiv
● GRÜN	Eingangspegel statisch 1 = logisch 0
● ROT	Eingangspegel statisch 0 = logisch 1
● GELB	Daten werden empfangen, wobei auch einzelne Bytes durch Streckung auf 100ms erkennbar sind
● BLAU	Kanal ist als Ausgang konfiguriert

6.13.2 Status LED

Die Status LED dient der Anzeige des Betriebszustandes des MSB-RS485-PLUS. Dies beinhaltet sowohl den Aufnahmestatus als auch die Anzeige von Fehlerzuständen.

LED Farbe	Bedeutung
● BLAU	Analyser ist per USB mit PC verbunden, Firmware nicht geladen und Programm nicht gestartet
●..● ORANGE blinkend	Firmware geladen und Gerät aufnahmebereit
● GRÜN	Aktive Aufnahme
● ROT	Fehler, USB Spannung zu niedrig
●..● ROT/WEISS blinkend	Überlastung eines AUX IO Ausganges
● WHITE	Datenüberlauf - Der Analyser produziert mehr Daten als vom PC verarbeitet werden kann[4]

[4]Die vom Analyser generierte Datenmenge kann in solchen Fällen oft durch Deaktivieren der Aufnahme der Leitungspegel reduziert werden. Siehe auch Abschnitt 8.2.3.

7

Analyse synchroner Bus Systeme

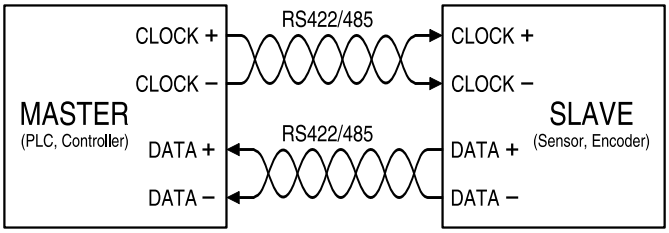
Wie schließe ich den Analyser an synchrone Bus Systeme wie z.B. SSI oder Manchester an und was muss ich dabei beachten? Wie kann ich die IO Ausgänge dazu verwenden, ein externes Digitalscope auf den Beginn der Datenrahmen bzw. auf Fehler in der Übertragung zu triggern? All das lesen Sie hier.

Das Hauptanwendung des MSB-RS485-PLUS ist sicherlich die Analyse von asynchronen Datenübertragungen, wobei jedes einzelne Datenbyte mit einem Startbit eingeleitet wird, gefolgt von 5-9 Datenbits, einem optionalen Parity Bit und einem Stopbit.
Das Design des MSB-RS485-PLUS Analyser bietet aber weitaus mehr Möglichkeiten zur Feld-Bus Analyse!

Durch die Integration zweier USARTs (Universal Synchronous And Asynchronous Receiver-Transmitter) ist der Analyser auch in der Lage, synchrone serielle Übertragungen aufzuzeichnen und zu analysieren.
Im Gegensatz zu asynchronen Transmissionen wird das Taktsignal nicht intern von jedem Bus Teilnehmer erzeugt (synchronisiert mit der fallenden Flanke des Startbits), sondern als separates Signal vom Bus Master bereitgestellt. Alternativ können die Daten auch als kombiniertes Daten/Takt-Signal (Phase Encoding) übertragen werden. Dies ist z.B. bei Manchester Bus Systemen der Fall.
Da die MSB-RS485-PLUS über zwei USARTs und vier unabhängige Eingangskanäle verfügt (CH1...CH4) können Sie bis zu zwei synchrone serielle Busse anschließen und analysieren.

7.1 Einen SSI Bus anbinden

Der SSI Bus (Synchronous Serial Interface) ist eine häufig eingesetzte Punkt-zu-Punkt Verbindung um in industriellen Anwendungen Sensordaten (i.a. Distanz Sensoren etc.) abzufragen.



Die physikalische Übertragung basiert auf RS422 und verwendet zwei RS485 Leitungspaare. Eine für den Datentakt (generiert durch den Master) und eine für die Daten (gesendet von dem Slave bzw. Sensor). Takt und Daten werden im allgemeinen über verdrehte Leitungspaare gemäß den RS422 Spezifikationen übermittelt. Das folgende Bild zeigt eine schematische Darstellung des Signalverlaufs und die spezifischen Zeitwerte. Die Datenübernahme erfolgt dabei immer mit der steigenden Flanke des Clock Signals (gekennzeichnet durch die Pfeilmarkierungen).

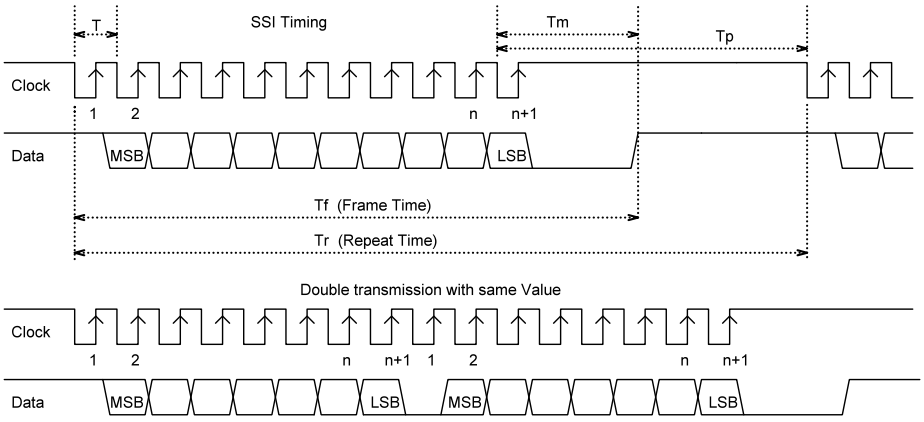
SSI Signalzeiten

T Taktbreite

Tm Device Monoflop Zeit

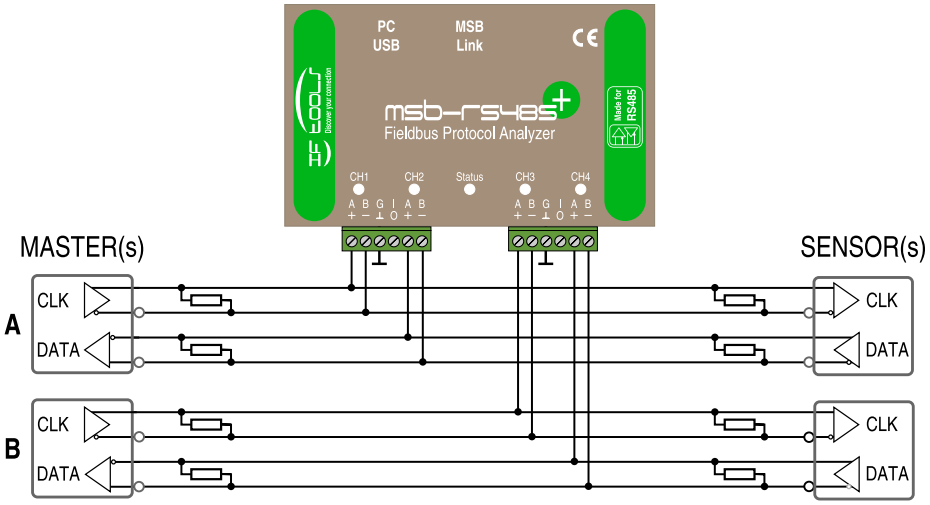
Tp Pause Zeit

n Taktnummer



Ein SSI Bus wird gemäß der nachstehenden Abbildung an den MSB-RS485-PLUS Analyser angeschlossen. Wie bereits im vorherigen Kapitel erwähnt: Es wird dringend empfohlen auch die Bus Masse mit dem Analyser zu verbinden! Auch dann, wenn der Bus ohne Masse arbeiten sollte! Dies sollte generell für alle Bus Teilnehmer berücksichtigt werden.

SSI Analyse
zweier SSI Bus Systeme



Bitte beachten Sie! Auch wenn der MSB-RS485-PLUS Analyser zwei unabhängige SSI Bus Systeme aufzeichnen kann, müssen beide aber die gleichen Übertragungsparameter vorweisen. Im Falle von SSI sind das die Anzahl der Bits in

einem Datenrahmen, eine optionales Taktverzögerung und doppeltes Senden des Datenrahmens.

7.1.1 Interne Signalverarbeitung

Um die übertragenen Daten korrekt erfassen zu können müssen die Bus Signale in ganz bestimmter Weise mit dem MSB-RS485-PLUS Analyser verbunden werden!
Der Takt der Bus Master muss immer an CH1 (SSI Bus A) bzw. an CH3 (SSI Bus B) angeschlossen werden. Der Anschluss der Datenleitungen erfolgt an CH2 (SSI Bus A) oder CH4 (SSI Bus B), siehe folgende Tabelle!
Ganz wichtig! Sowohl CH1/CH2 als auch CH3/CH4 arbeiten jeweils als Paar um die Daten aus den entsprechenden Takt/Daten-Signalen zu extrahieren. Die Bus Leitungen eines SSI Busses gehören deshalb immer auf einen Anschluss-Stecker!

SSI Bus Signal	Kanal	Beschreibung
BUS A, CLOCK+	CH1 A+	Positives CLOCK Signal SSI Bus A
BUS A, CLOCK−	CH1 B−	Negatives CLOCK Signal SSI Bus A
BUS A, DATA+	CH2 A+	Positives DATA Signal SSI Bus A
BUS A, DATA−	CH2 B−	Negatives DATA Signal SSI Bus A
BUS B, CLOCK+	CH3 A+	Positives CLOCK Signal SSI Bus B
BUS B, CLOCK−	CH3 B−	Negatives CLOCK Signal SSI Bus B
BUS B, DATA+	CH4 A+	Positives DATA Signal SSI Bus B
BUS B, DATA−	CH4 B−	Negatives DATA Signal SSI Bus B

Der Analyser zeigt alle per CH1/CH2 übertragene Signale als Daten A und alle Signale an CH3/CH4 als Daten B an.
Dies gilt auch für etwaige Rahmen- und Parity-Fehler während der SSI Übertragung. Beide Fehler werden den jeweiligen Daten A bzw. Daten B entsprechend dem Bus Anschluss zugeordnet.

7.1.2 Signal Zuordnung

Die Anzahl der vom MSB-RS485-PLUS Analyser zur Verfügung gestellten Datenkanäle (2) und Signalkanäle (8) ist durch die Hardware vorgegeben und unabhängig von der ausgewählten Übertragungsart (asynchrones oder synchrones Bus System).
Die beiden Datenkanäle dienen wie üblich zur Anzeige der übertragenen Daten (Bus A und Bus B).
Von den 8 Logik Kanälen werden 4 zur Anzeige der beiden SSI Daten und Takt Signale verwendet. Die übrigen 4 Kanäle sind keinem 'echten' Signal zugewiesen sondern 'virtuelle' Kanäle. Es handelt sich hierbei um zusätzliche Datenrahmen Signale, die die Analyser Hardware für alle Arten von synchrone Übertragungen generiert. Im Abschnitt 7.3 werden wir diese genauer erörtern.
Die folgende Tabelle listet die Details:

Anzeigekanal[a]	Beschreibung
Daten A (Data A)	Daten von SSI Bus A an CH1 und CH2
Daten B (Data B)	Daten von SSI Bus B an CH3 und CH4
Signal 1 (CH1)	SSI Bus A, Taktsignal an CH1
Signal 2 (CH2)	SSI Bus A, Datensignal an CH2
Signal 3 (CH3)	SSI Bus B, Taktsignal an CH3
Signal 4 (CH4)	SSI Bus B, Datensignal an CH4
Signal 5 (FRM:A)	Rahmen Gültigkeit-Signal für synchronen Bus A
Signal 6 (BITS:A)	Bitwechsel Signal für synchronen Bus A
Signal 7 (FRM:B)	Rahmen Gültigkeit-Signal für synchronen Bus B
Signal 8 (BITS:B)	Bitwechsel Signal für synchronen Bus B

[a] In Klammern die im Programm angezeigten Signalnamen

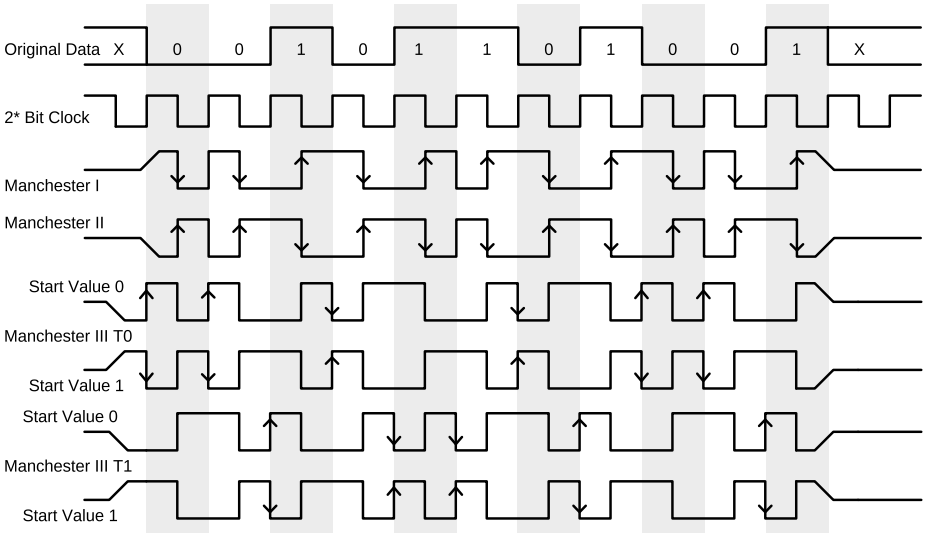
7.2 Analyse eines Manchester Bus

In einer Manchester Übertragung werden Daten und Takt (Clock) zu einem einzigen Signal kombiniert[1]. Das Signal ist damit wie asynchronen Übertragungen selbst-taktend. D.h. der Takt läßt sich aus dem Datensignal zurück gewinnen. Damit hören die Gemeinsamkeiten allerdings auch schon auf.
Während bei asynchronen Transmissionen einzelne Bit entweder als niedriger oder hoher Signalpegel spezifiziert sind, verwendet Manchester die Signaländerung zwischen den beiden Signalpegel als Bit Information[2]. Die Änderung selbst wird durch eine Verknüpfung des Datensignals mit einem Takt der halben Bitweite - bzw. der doppelten Bitrate - erzeugt. Das resultierende Signal enthält ZWEI Zustände (halbe Bits) für jedes übertragene Bit (high→low oder low→high). Die effektiven Bitrate wird damit auf die halben Übertragungsrate reduziert. Dies ist aber auch schon die einzige Einschränkung im Vergleich zu asynchronen Übertragungen.

Jedes Bit hat per Definition eine Flanke in der Bitmitte sowie abhängig vom Modus und Bitwert eine Flanke zu Beginn des Bits. Durch die hohe Zahl von Pegelwechseln ist der Mittelwert der Busspannung nahe Null, der Bus kann daher mit einfachen Transformatoren isoliert übertragen werden.
Voraussetzung dafür ist lediglich, dass der Bus im inaktiven Zustand einen Ruhepegel von Null Volt hat. Eine Erfordernis, die ein terminierter RS485 Bus perfekt erfüllt[3].

Das folgende Bild zeigt die Daten Codierung alle Manchester Varianten im Detail. Die entscheidenden Bitflanken sind durch Pfeile gekennzeichnet.

[1] Mit der zugrunde liegenden Annahme eines konstanten Taktsignals.
[2] Auch bekannt als Phase-Encoding oder PE.
[3] Aber nicht ein RS232 oder digitaler Bus.



Manchester transmissions
Data and clock linking

Die hohe Anzahl von Signalwechsel hat zudem noch weitere Vorteile. Denn dadurch ist es jedem Busteilnehmer leicht möglich, seine interne Abtastlogik mit jedem neuen Bit zu resynchronisieren.
Im Umkehrschluss bedeutet dies aber NICHT, dass Manchester eine Änderung der Bitrate während einer Übertragung erlaubt! Das Gegenteil ist der Fall. Bei einer variablen Bitrate kann die Abtastlogik nicht wissen, ob noch eine weitere Signalflanke kommt und ob diese Signalflanke dem Bitanfang oder der Bitmitte zuzuordnen ist. Beides setzt die genaue Kenntnis der Bitbreite/weite und damit der Bitrate voraus!
Der MSB-RS485-PLUS kann die Bitrate automatisch für Sie ermitteln - oder Sie geben diese manuell vor.

7.2.1 Manchester Implementierungen

Manchester kommt in zwei grundsätzlichen Varianten. In der ersten - bekannt als Manchester Encoding - hängt die Bit Information von der Polarität oder Richtung der Signaländerung in der Bitmitte ab.
Die zweite Variante verwendet die An- oder Abwesenheit einer Signaländerung beim Bitstart als Bit Information. Dieser Kodierungstyp wird auch als Differential Manchester Encoding oder kurz DME bezeichnet.
Beide Manchester Implementierungen unterscheiden zwei Unterarten, abhängig davon, wie die Polarität der Richtungsänderung bzw. die An/Abwesenheit eines Pegelwechsel einem logischen Wert 0 oder 1 zugewiesen wird.
Bei der ersten Variante - Manchester Encoding - werden diese beiden Typen als Manchester I (G.E. Thomas) und Manchester II (IEEE 802.3) bezeichnet. Sie sind leichter zu verstehen, deshalb beginnen wir mit diesen.

7.2.1.1 Manchester I und II

Manchester I und II verwenden - wie oben erwähnt - die Signaländerung/Flanke in der Mitte des Datenbits als Information. Der Unterschied zwischen beiden liegt lediglich in der Definition, welche Pegeländerung einem logischen Status 0 oder 1 entspricht.

- Manchester I (G.E. Thomas)
  Low zu High → 1
  High zu Low → 0
- Manchester II (IEEE 802.3)
  Low zu High → 0
  High zu Low → 1

Der initiale Signalpegel spielt dabei eine wichtige Rolle. Je nach Zustand des ersten zu übertragenden Bits (und der damit verbundenen Notwendigkeit eines Pegelwechsel von low auf high bzw. umgekehrt, muss die Übertragung mit einem entsprechenden Ausgangs- oder Startpegel beginnen. Der Signalpegel wechselt dabei vom Ruhepegel zu einem hohen oder niedrigen Pegel. Die Übertragung des ersten Bits endet dann mit Wechsel des Takt(Clock)Signals und dem dort stattfindenden Pegelwechsel.
Beachten Sie, dass die Datenübernahme bei Manchester I/II IMMER in der Bitmitte erfolgt.

7.2.1.2 Differential Manchester T0 und T1

Bei einer Differential Manchester Kodierung ist der Zustand eines übertragenden Bits durch die An- bzw. Abwesenheit eines Pegelwechsel beim Bit START definiert. Auch hier existieren zwei Untertypen, bezeichnet als T0 und T1.

- Manchester T0
  Anwesenheit eines Pegelwechsels → 0
  Abwesenheit eines Pegelwechsels → 1
- Manchester T1
  Anwesenheit eines Pegelwechsels → 1
  Abwesenheit eines Pegelwechsels → 0

Im Gegensatz zum vorherigen Manchester I/II erfolgt die Datenübernahme hier IMMER bei Bit Beginn! Der Signalwechsel in der Bitmitte ist allein dem Takt (Clock) geschuldet!

7.2.2 Interne Signalverarbeitung

Bei einer asynchronen Datenübertragung beginnt ein Datenrahmen (Bitfolge, die ein Datenbyte definiert) immer mit der ersten fallenden Flanke. Bei Manchester hingegen ist die erste Signalflanke nicht definiert und beliebig. Sie hängt, wie oben gesagt, vom Manchester Typ und dem ersten Bit ab. Um hier eine Zuordnung zu treffen, können zu Beginn einer Übertragung der Anwendung bekannte Bits gesendet werden über die sich die interne Biterkennung synchronisieren lässt.

Der MSB-RS485-PLUS erkennt die korrekten Bits auf eine andere Art:
Der Biterkennung wird gestartet, wenn der Bus von inaktiv (idle) zu aktiv (high oder low) wechselt. Dabei wird die dabei entstehende Signalflanke ignoriert, da sie nur der Einstellung des Ausgangspegels dient (im obigen Bild schräg dargestellt).
Es wird angenommen, dass die erste Flanke zu Bitmitte erfolgt, also bereits ein aktives Bit kennzeichnet. Im Fall Manchester I und II kann die Auswertung mit diesem Bit beginnen.

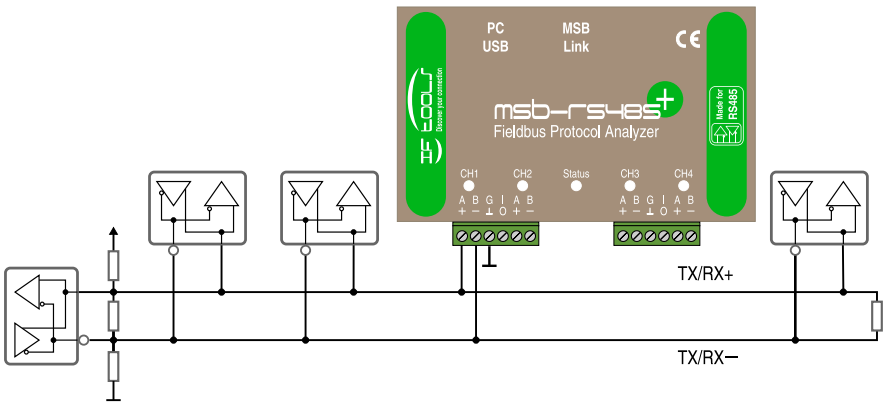
Im Falle der Differential Manchester Übertragungen T0 und T1 ist es ein wenig komplizierter. Ob es sich bei der ersten gültigen Flanke (high auf low bzw. umgekehrt) um eine signifikanten Pegeländerung oder lediglich um den Wechsel des Clock Signals in der Bitmitte handelt, wird erst bei der abschließenden Erkennung des Folgebits eindeutig klar. Das Wissen um die korrekte Bitrate (und damit Bitweite des Clock Signals) ist dabei essentiell!
Im Allgemeinen werden die Bits als Vielfache von 8 Bit gesendet. Der Analyser gibt bei dem letzten übertragenen Byte einen Framing Fehler aus, wenn dieses Byte nicht vollständig ist oder die Zahl der übertragenen Bits keine Vielfaches von 8 sind. Dies kann allerdings je nach Anwendung ignoriert werden.

7.2.3 Signal Zuordnung

Der MSB-RS485-PLUS Analyser verfügt über 2 Daten-Dekodier-Einheiten (Datenkanäle) und 8 Signalkanäle. Dies erlaubt die Aufnahme und Analyse von drei verschiedenen Manchester Bus Szenarien. Der Anschluss eines Manchester Busses an den Analysers ist dabei ähnlich dem bei Aufnahme asynchroner Bussysteme.
Sie können den Analyser mit einem halb-duplex 2-Draht Manchester Bus verbinden, ihn an einen voll-duplex 4-Draht Bus anschließen, zwei unabhängige Manchester Busse aufnehmen (allerdings beide mit identischer Parametrisierung) oder einen Manchester Bus durch den Analyser leiten.
Letzteres erlaubt die korrekte Zuordnung der Datenquelle (Sender) unabhängig vom verwendeten Protokoll.

Der Anschluss des Analysers an den Bus ist unabhängig vom Manchester Typ. Nicht desto trotz ist es für den Analyser wichtig zu wissen, welchen Manchester Typ Ihre Anwendung verwendet. Und abhängig von dem Typ kann es auch notwendig werden, die beiden Leitungen zu vertauschen. Allein um die korrekte Polarisation der Signalflanken und damit eine zuverlässige Datenerfassung zu gewährleisten. Zum Glück müssen Sie in einem solchen Fall die Anschlüsse nicht manuell um verdrahten. Der Analyser erledigt dies für Sie quasi per Klick.

Übliche Manchester Applikationen bestehen in der Regel nur aus einem RS485 Leitungspaar. Wir beginnen deshalb mit dieser Variante. Der Anschluss des Analysers erfolgt gemäß dem folgenden Bild.

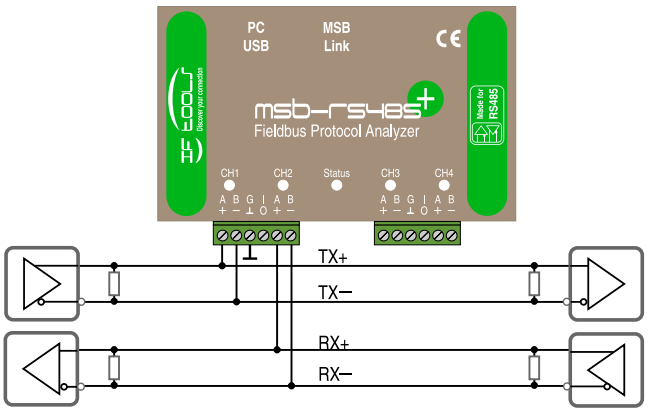


2-Draht Anschluss an einen einzelnen Manchester bus

In dem dortigen Manchester Einstelldialog können Sie die Signale individuell für alle Datenkanäle CH1...4 invertieren - falls nötig - siehe oben. Dies gilt im Übrigen auch für die nächsten beiden Anschlussarten.

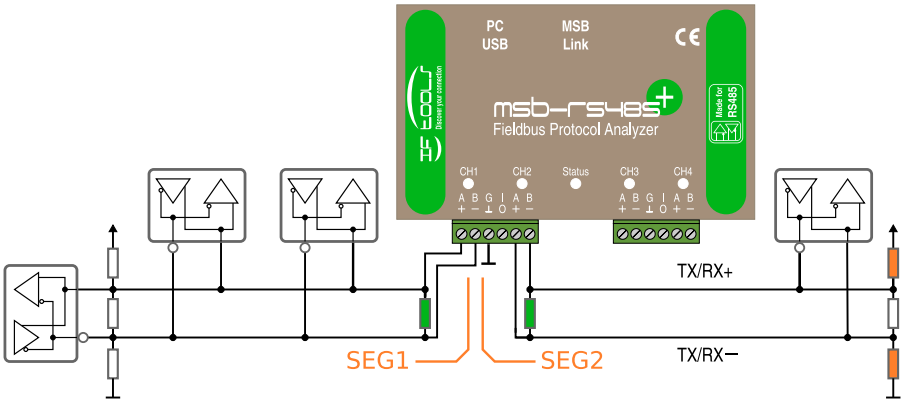
Da die MSB-RS485-PLUS 2 Datenkanäle (Dekodier-Einheiten) besitzt, spricht nichts dagegen, ihn mit zwei Manchester Bus Systemen zu verbinden. Dies kann - wie gesagt - ein voll-duplex System sein, eine Anwendung mit redundatem Bus oder zwei unabhängige Busse. Die einzige Voraussetzung ist, dass beide Busse mit den gleichen Manchester Einstellungen betrieben werden (Typ, Bitrate, Bit Order).
Wie Sie den Analyser mit zwei Manchester Busse verbinden zeigt das folgende Anschlussbild:

4-Draht Anschluss
für voll-duplex oder zwei Halb-duplex Manchester Bus Anwendungen



Die dritte Option ist den Analyser in einen Manchester Bus einzufügen und per Segment Mode zu betreiben. Die Vorteile sind dabei die gleichen, wie bereits im allgemeinen Anschluss-Kapitel beschrieben. Durch die Auftrennung des Busses in zwei Segmente arbeitet der Analyser als Router und kann jedes aufgenommene Datenbyte mit der exakten Herkunft (Segment/Richtung) versehen. Diese Information kann den Unterschied machen wenn Sie ansonsten keine Möglichkeit haben, die Herkunft der Daten bzw. Telegramme zu ermitteln. Z.B. weil das verwendete Protokoll eine solche Information nicht vorsieht.

2-Draht Segment Anschluss
für einen einzelnen Manchester Bus mit Richtungserkennung



Beachten Sie hier, dass die Auftrennung eines RS485 Busses eine zusätzliche Terminierung erforderlich macht (im Bild grün dargestellt). Dies können Sie

bequem im Anschluss Dialog vornehmen.

7.3 Spezielle Datenrahmen Signale

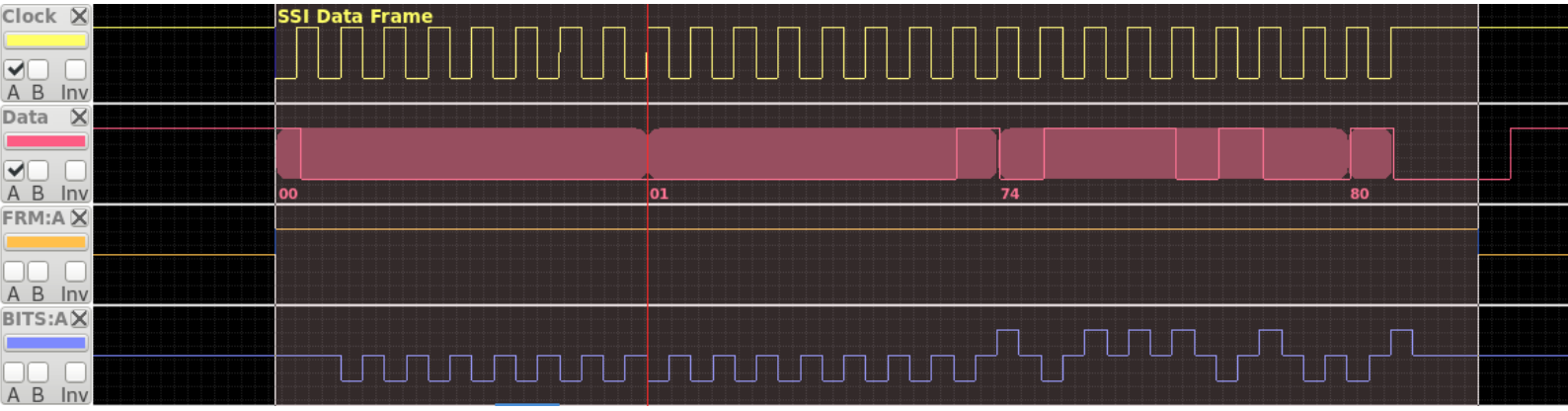
Bei der Analyse von seriellen Bus Transmissionen ist es entscheidend zu wissen wo in der Übertragung ein Datenrahmen beginnt und wo er endet. Vor allem wenn Sie die einzelnen Datenbytes in der Signal Darstellung korrekt eingeblendet sehen wollen oder die Daten per Lua Protokoll Template weiter verarbeiten möchten.

Bei asynchronen Übertragungen wird ein Datenrahmen (oder Data Frame) durch sein Datenformat wie z.B. '8N1' oder '7E1' (Anzahl Datenbits, optionales Parity, Anzahl Stopbits) definiert. Ein Datenrahmen beginnt immer mit einem Startbit und endet mit einem oder mehreren Stopbits.

Bei synchronen seriellen Transmissionen ist das nicht so klar.
Weder sind die einzelnen Datenbits immer gleich lang[4] noch werden die Daten in einzelnen Byte Paketen gesendet. Synchrone Übertragungen mit einer Manchester Codierung (Phase Encoded) kombinieren sogar Takt und Daten in einem einzigen Signal wobei dieses aus unterschiedlich langen Bitwechsel besteht.
Zum Glück kümmert sich die Analyser Hardware um solche Fälle und generiert zwei zusätzliche 'virtuelle' Signale für jeden angeschlossenen seriellen synchronen Bus. Das erste als FRM (Frame) bezeichnete Signal zeigt durch seinen Pegelwechsel die korrekte Länge des Datenrahmens an. Das zweite (genannt BITS) repräsentiert eine Kombination aus Daten und Taktsignal wodurch einzelne Bits auch bei Phasen kodierten Übertragungen wie Manchester eindeutig zugeordnet werden können.

7.3.1 SSI Datenrahmen und Bit Signale

Im nachfolgenden Bild (der Ausschnitt einer SSI Übertragung aufgenommen mit dem Signalmonitor) markiert das FRM:A Signal (orange) den zeitlichen Verlauf des Datenrahmens (Data A) mit einem High Pegel.

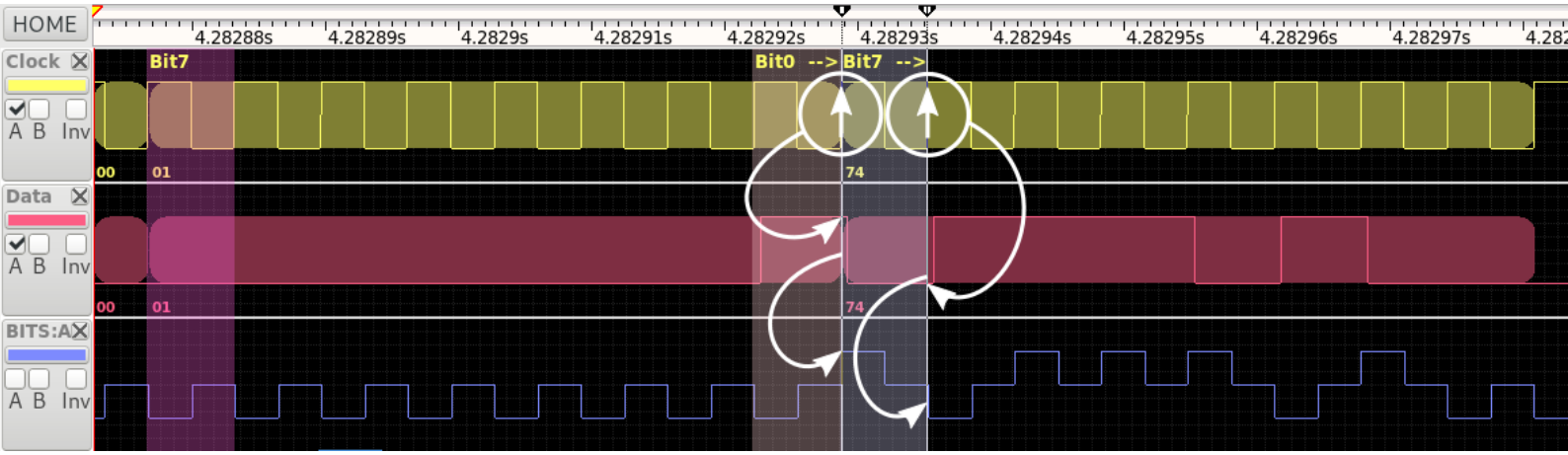


Datenrahmen Signale

[4] Bei I2C ist es erlaubt, dass der Master oder ein Slave den Takt zeitlich dehnen kann. Dies wird als 'Clock Stretching' bezeichnet.

Das BITS Signal hingegen repräsentiert Länge und Wert jedes einzelnen Bits im Datenrahmen. Ein gesetztes Bit (1) wird mit einem High Pegel angezeigt, ein nicht gesetztes Bit (0) mit einem Low Pegel. Der Default Pegel ist der RS485 Ruhepegel (Idle Pegel oder Bus Freigabe Pegel).
Insbesondere das erste Signal (FRM) spielt eine entscheidende Rolle um die einzelnen Telegramme (Folge von Datenbytes) aus dem Bitstrom zu extrahieren. Die interne Zuordnung der Signale wird im entsprechenden Abschnitt der jeweiligen synchronen Bus Verbindungen beschrieben.

Eine kleine Anmerkung!
Das BITS:A Signal im obigen Bild scheint einen Takt nach rechts verschoben. Dies ist der SSI Spezifikation geschuldet, die eine Datenübernahme mit der ansteigenden Flanke des Takt(Clock) Signals NICHT in der Mitte des Datenbits sondern gegen Ende vorschreibt. Der Grund: Bei der Datenübernahme am Ende kann die maximale Bandbreite der Leitung besser ausgeschöpft werden. Das Resultat ist eine robustere Datenübertragung auch bei schlechter Signalqualität. Der MSB-RS485-PLUS Analyser mit seiner hohen Zeitauflösung spiegelt diese Tatsache in großer Genauigkeit wieder. Das folgende Bild zeigt die entsprechend vergrößerten Pegelverläufe.

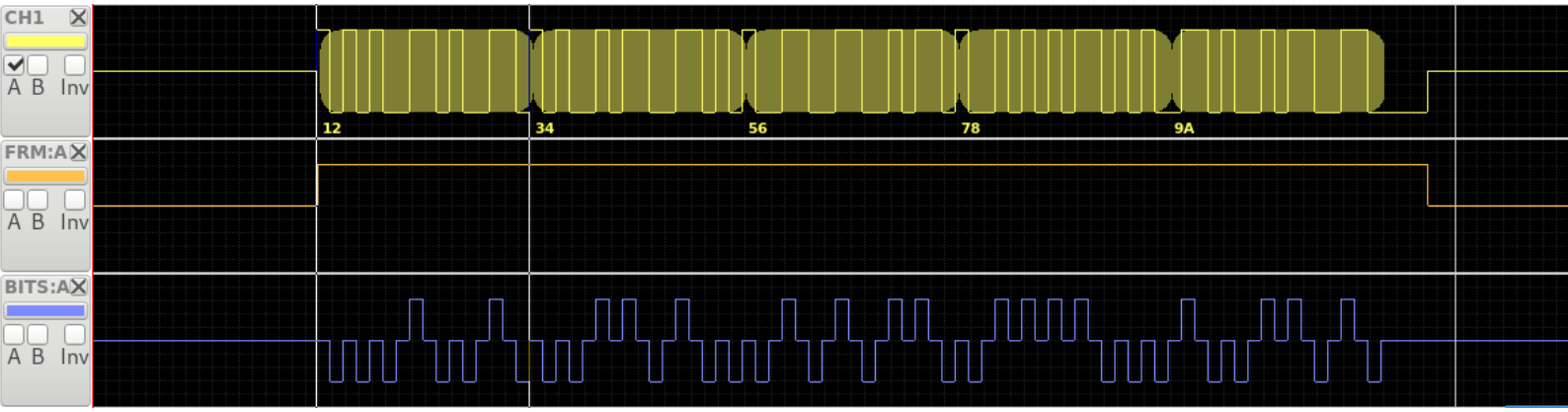


BITS:A Signal

Die beiden Cursor I und II (angedeutet durch die beiden Dreiecke in der Zeitskala oben) zeigen den jeweils entscheidenden Moment der Datenübernahme. Cursor I markiert dabei die steigende Flanke im Takt (Clock) Signal für Bit 0 (linker Kreis). Zu exakt diesem Zeitpunkt hat das Datensignal immer noch einen High Pegel - Bit 0 ist 1. Der Analyser zeigt dies durch einen ebenfalls kurzen (einen halben Takt breiten) positiven Pegel im BITS:A Signal an. Danach geht das BITS:A Signal wieder zurück auf den Idle (Ruhe) Pegel.
Bei der Abtastung des nächsten Bits (Bit 7 des nächsten Datenbytes. SSI überträgt die Daten mit den höchstwertigen Bit zuerst), ist der Bit Status noch low (markiert durch den rechten Kreis). Die Ausgabe im BITS:A Signal ist deshalb ein niedriger (negativer) Pegel ebenfalls für eine Taktlänge.

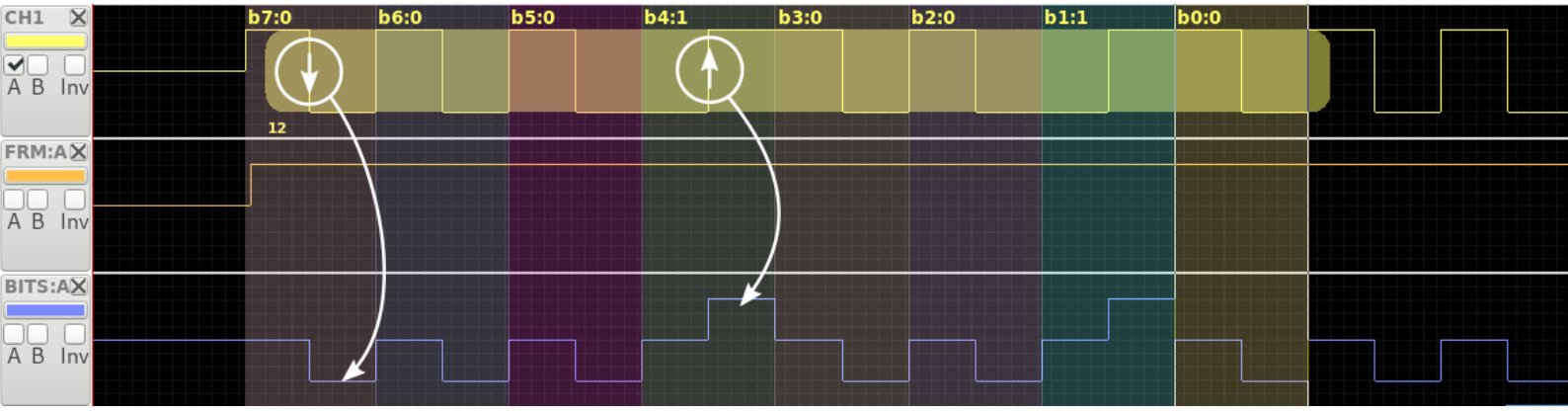
7.3.2 Manchester Datenrahmen und Bit Signale

Wie sinnvoll und wichtig ein Feature wie die virtuellen BITS Kanäle sein kann, erschließt sich insbesondere bei nicht einfach zu lesenden Datensignalen.
Manchester mit seiner Kombination von Daten und Takt ist ein gutes Beispiel hierfür. Sehen Sie sich dazu die folgende kurze (5 Byte lange) Manchester I Übertragung an.



Rahmensignal für Manchester I (G.E. Thomas)

Das allererste Byte (hexadezimal 12) ist markiert zwischen den beiden Cursor. Die Bitfolge hex 12 ist 00010010 (höchstwertiges Bit zuerst).
Kein Zweifel, die Dekodierung des reinen Datensignals mit bloßem Auge ist nur etwas für wirklich erfahrene Anwender. Mit dem virtuellen BITS Signal dagegen ist es ein Kinderspiel. Immer dann, wenn der Analyser ein Bit komplett detektiert hat, gibt er den Bitwert (0 oder 1) als ein halbes Bit breites Signal auf dem BITS Kanal aus.
Ein halbes Bit breit deshalb, um die Bitfolge im BITS Signal durch die trennenden Ruhe(Idle)Pegel leichter lesbar zu machen. Dies gilt vor allem bei unveränderten Bitfolgen wie hex FF oder hex 00, siehe das nächste Bild:

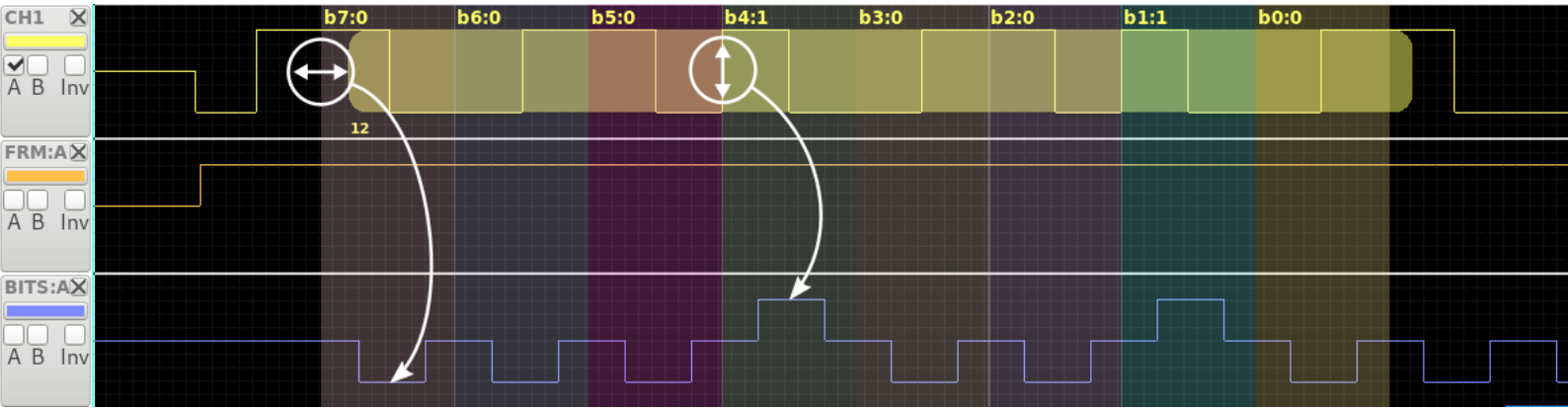


BITS:A Signal für Manchester I (G.E. Thomas)

Erinnern Sie sich?
Bei Manchester I Type G.E. Thomas bedeutet eine fallende Flanke in der Bit-

mitte eine 0, eine steigende Flanke eine 1. Im obigen Bild sind die einzelnen Bitwerte (Name und Wert) als unterschiedliche Regionen eingeblendet um das Ganze noch besser zu verdeutlichen.

Die Nützlichkeit des BITS Signal wird noch klarer, wenn Sie eine Manchester T0 oder T1 Übertragung analysieren wollen. Bei diesen Manchester Typen wird der Bitwert durch An- bzw. Abwesenheit eines Pegelwechsels im Datensignal definiert. Nehmen wir als Beispiel ein Manchester T1 Datensignal. Hier ist ein Bitwert von 1 als Anwesenheit eines Signalflanke beim Bitstart definiert (im Bild der rechte Kreis), während ein Bitwert von 0 durch keine Signaländerung oder Flanke zu Bit Beginn indiziert wird (im Bild der linke Kreis). Bei Manchester T0 ist es genau umgekehrt.



BITS:A Signal für Manchester T1

Da Manchester Übertragungen IMMER einen Pegelwechsel in Bitmitte aufweisen, ist es ein leichtes, entsprechende Regionen zur Veranschaulichung der einzelnen Bits einzufügen. Dies ist allerdings ziemlich zeitaufwendig und nicht wirklich eine Alternative, wenn Sie das logische Datensignal einfach mal kurz prüfen wollen.
Auch hier schafft das BITS Signal Abhilfe. Es zeigt Ihnen detailliert die einzelnen Bitwerte genau so, wie der Analyser diese aus dem Datensignal detektiert.

7.4 Digital IOs als Trigger-Signal

Bei der Aufzeichnung von synchronen seriellen Bussystemen sind alle internen Signal Kanäle bereits belegt. CH1...CH4 repräsentieren die Takt und Datenleitungen, CH5...C8 die virtuellen vom Analyser erzeugten Datenrahmen FRM und Datenbitwechsel BITS Signale.

Im Gegensatz zur asynchronen Betriebsart ist die Einstellung der digitalen IOs hier deshalb auf eine reine Signalausgabe beschränkt. Voreingestellt ist die Ausgabe eines kurzen Trigger Signals im Falle eines Fehlers während der Datenübertragung (im Datenrahmen). Z.B. ein von der Analyser Hardware erkannter Frame oder Parity Fehler. Alternativ können Sie den Ausgang aber auch dazu verwenden, auf den Beginn der Datenrahmen zu triggern. In diesem Fall liegt an dem Ausgang ein High Pegel an, solange Datenbits übertra-

gen werden.

Beide Einstellungen können sich als sehr hilfreich erweisen, wenn Sie die Datenpakete oder auftretende Übertragungsfehler z.B. mit einem externen Digitalscope genauer untersuchen wollen.
Die folgenden Einstellungen sind für beide IO Ausgänge möglich. Dabei ist IO1 dem ersten synchronen Bus zugeordnet, IO2 dem zweiten[5].
Die dritte Auswahl dient dabei zur Versorgung spezieller (individueller) Adapter. Sie ist unabhängig von der verwendeten Übertragungsart oder der gewählten Signaleinstellung.

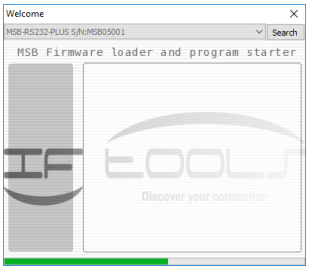
IO-Type	Description
Ausgang	Ausgabe bei Datenübertragungsfehler, $+5V$ (High) Pegel für ca $5\mu s$, sonst (Low) $0V$.
Ausgang	Ausgabe des Datenrahmen Signals, $+5V$ (High) solange die Datenübertragung aktiv ist, sonst $0V$ (Low).
Ausgang	5V/50mA zur Versorgung externer Adapter

[5]Dies gilt für alle vom Analyser unterstützten synchronen seriellen Bus Systeme.

8 Programm Start

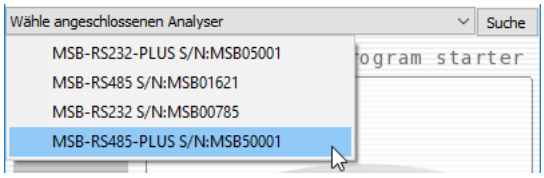
Das Kontrollprogramm ist das Cockpit Ihres Analysers und Dreh- und Angelpunkt für Ihre Feldbus Analyse. Hier geben Sie die Bus Spezifikationen ein, starten die Aufzeichnung, speichern oder laden Aufzeichnungsdateien oder Projekte und öffnen Analyse Tools um die Aufzeichnung auf verschiedenen OSI Ebenen in Echtzeit zu untersuchen.

Die Firmware des Analysers ist nicht fest im Gerät integriert sondern muß nach Einschalten (Anschluß) des Gerätes zunächst geladen werden. Dies findet allerdings nur einmal statt. Solange das Gerät eingeschaltet, d.h. mit Spannung versorgt wird, bleibt die Firmware erhalten.
Sobald Sie die Analyser Software durch Doppelklick auf Ihren Desktop Icon starten, erscheint deshalb zunächst der Firmware Loader.
Der Firmware Lader erkennt automatisch, ob ein Analyser angeschlossen ist und ob die Firmware bereits geladen ist oder noch übertragen werden muß. Wurde das Gerät korrekt erkannt und die Firmware geladen (erkennbar am Fortschrittsbalken im unteren Teil des Dialogfensters) startet automatisch das MSB-RS485-PLUS Kontrollprogramm.
Sollten Sie mehrere Analyser an Ihrem PC angeschlossen haben zeigt Ihnen der Firmware Loader eine Auswahlliste der vorhandenen Analyser.

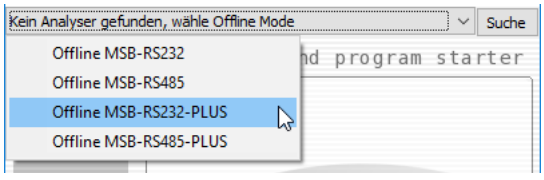


Der erste Start
lädt die Firmware
in den Analyzer

Wurde kein Analyser gefunden, obwohl Sie diese mit Ihrem PC verbunden haben, lesen Sie die Hinweise im Anhang zu Windows im Abschnitt D, bzw. für Linux unter E. Falls Sie einfach nur vergessen haben, den Analyser anzuschliessen, verbinden Sie diesen mit dem PC und klicken Sie auf den 'Suche' Knopf um die Auswahlliste zu aktualisieren.



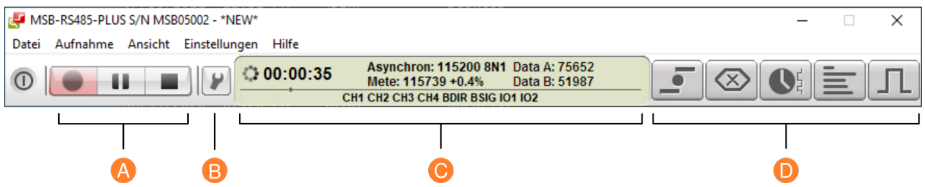
Sie können allerdings auch ohne MSB-RS485-PLUS mit dem Programm arbeiten, z.B. um nachträglich aufgezeichnete Daten auszuwerten oder das Tutorial durchgehen, um sich mit der Arbeitsweise des Programmes vertraut zu machen. Da das Programm in diesem Fall den Analyser Typ - MSB-RS232 (PLUS) oder MSB-RS485 (PLUS) - nicht automatisch ermitteln kann, müssen Sie diesen aus der Liste vorgeben.



Die MSB-RS485-PLUS Software verwendet eine MultiProzess Architektur. Das bedeutet, dass das Programm nicht ausschließlich in einem einzigen Fenster abläuft, sondern für die unterschiedlichsten Aufgaben ein speziell darauf abgestimmtes Tool startet.
Angesichts dessen mag Ihnen der Start eines kleinen Kontrollpanels kümmerlich erscheinen. Die Software soll Sie aber nicht mit unnötigen Fenstern, mehrzeiligen Werkzeugleisten und verschachtelten Menüs verwirren, sondern Ihnen einen Satz einfach zu bedienender Tools zur Verfügung stellen, die für Ihre Anwendung am geeignetsten sind.
Aus dem Kontrollprogramm heraus können Sie jederzeit - auch während einer aktiven Aufnahme - unterschiedlichste OSI Ebenen der Kommunikation betrachten und analysieren, indem Sie das entsprechende Tool (View) starten. Der Aufnahmeprozess ist davon völlig unabhängig.

8.1 Benutzer Interface

Das Kontrollprogramm erscheint sobald die Übertragung der Firmware abgeschlossen ist. Abhängig von Ihrer letzten Sitzung oder wenn Sie die Software per Doppelklick auf eine Projektdatei gestartet haben, erscheinen weitere Programm Fenster, die unterschiedliche Ansichten einer Aufnahme repräsentieren. Hier geht es allerdings nur um das Kontrollprogramm, und das sieht wie folgt aus:



A	**Aufnahmesteuerung:** Einfaches starten, pausieren oder stoppen einer Aufzeichnung.
B	**Einstellungen:** Alle nötigen Aufnahme Einstellungen sind mit einem Klick erreichbar. Kein umständliches Navigieren durch Menüeinträge.
C	**Aufnahmekontrolle :** Übersichtliche Darstellung der aufgezeichneten Daten/Ereignisse, Einstellung, Aufnahmedauer und Zustand. Wechsel der Anzeige per rechtem Mausklick.
D	**Analysetools:** Startet das gewünschte Analyzetool mit seinen letzten Einstellungen.

Das Kontrollprogramm ist in vier Abschnitte unterteilt. Auf der linken Seite befindet sich die Aufnahme Steuerung (A). Inspiriert von einem Audio oder Video Rekorder enthält es einen Aufnahme, Pause und Stopp Knopf. Die Bedeutung sollte klar sein. Ein einfacher Klick, und die Aufzeichnung beginnt...
Hinter dem Knopf mit dem Schraubenschlüssel Symbol (B) verbergen sich alle nötigen Aufnahme Einstellungen, siehe 8.2. Sie müssen den Analyser entsprechend Ihrer Bus Anwendung konfigurieren bevor Sie eine Aufzeichnung starten - allerdings nur einmalig solange Sie keine Änderungen an dem angeschlos-

senen Bus vornehmen.
Der zentrale Part des Kontrollprogrammes ist das Status Display (C). Hier erscheinen alle wesentlichen Informationen der aktuellen Aufnahme. Eine kurze Beschreibung der Übertragungsart, die Aufnahmezeit, gemessene Bitrate sowie die Anzahl der empfangenen Daten und Bus Ereignisse wie z.B. Pegelwechsel. Wir erklären dies im Detail im Abschnitt 8.4.
Rechts neben dem Status Display befinden sich die Starter für alle sogenannten Views. Dies sind eigenständige Programme oder Tools, die Ihnen unterschiedliche Einblicke in die verschiedensten Ebenen (OSI Schichten) der aktuellen Übertragung ermöglichen, siehe 8.5.

Wie bereits erwähnt: Die Analyser Software verwendet ein Multi-Prozess Design. Sie können beliebig viele Analysetools oder Views zur Untersuchung Ihrer Bus Kommunikation starten wie Sie möchten. Das Kontrollprogramm arbeitet völlig unabhängig von den Views und eine aktive Aufzeichnung wird auch dann nicht gestört, wenn eines der Views abstürzen oder unbedienbar werden sollte. Dies gibt Ihnen maximale Sicherheit für Ihre Aufzeichnung. Insbesondere wenn die äußeren Umstände nur eine einmalige Aufnahme erlauben oder Sie keine zweite Möglichkeit haben, einen sporadisch auftretenden Fehler zu erfassen.

Vor der eigentlichen Aufnahme müssen wir aber zunächst die Aufzeichnung konfigurieren.

8.2 Eine Aufzeichnung konfigurieren

Aufnahme Setup
Konfigurieren einer Aufzeichnung

Um bei unseren Bild eines Audio Rekorder zu bleiben müssen Sie zuerst die Aufnahmequelle (Eingang) wählen sowie den Aufnahmepegel justieren. Das gleiche gilt im übertragenen Sinne für den Analyser. Bevor Sie den Aufnahmeknopf betätigen müssen Sie zunächst den MSB-RS485-PLUS entsprechend konfigurieren! Die Software speichert alle Einstellungen automatisch, so dass Sie dies für einen bestimmten Bus nur einmal machen müssen.

Der MSB-RS485-PLUS Analyser kann verschiedenste Bus Systeme und Übertragungsarten aufnehmen (abtasten) und analysieren. Aber dazu müssen Sie dem Analyser zuerst mitteilen, welche Art von Bus System vorliegt und wie Sie diesen mit dem Analyser verbunden haben. Außerdem die entsprechenden Übertragungsparameter wie z.B. Bitrate, Datenformat etc.).
Das Programm führt Sie Schritt für Schritt durch alle nötigen Einstellungen. Klicken Sie einfach den Knopf mit dem Schraubenschlüssel Symbol  links neben dem Display um den Setup Dialog zu öffnen.

Der sich öffnenden Dialog ist in verschiedene Bereiche unterteilt, wobei die Wichtigkeit der Einstellungen von links nach rechts angeordnet ist.

1.Übertragung 2.Anschluss 3.Signale 4.Aufnahme 5.Autosave 6.Allgemein

Obligatorisch für eine Aufnahme sind die beiden ersten Einstellungen Übertragung und Anschluss. Hier spezifizieren Sie welche Art von Bus System der Analyser aufnehmen soll (in unserem Audio Metapher die Eingangsquelle) und wie die Daten aus dem übertragenen Bitstrom extrahiert werden sollen. Die anderen vier Dialog Sektionen (Signale, Aufnahme, Autosave und Generell) enthalten Vorgaben, die für die meisten Anwendungen passen sollten. Sie können diese daher erst mal ignorieren.

Einige Einstellungen beeinflussen direkt die Bus Aufzeichnung. Diese sind deshalb während einer aktiven Aufnahme gesperrt (ausgegraut). Das gleiche gilt, wenn Sie ohne verbundenen Analyser arbeiten (ohne angeschlossenes Gerät macht eine Einstellung desselben keinen Sinn).

8.2.1 Einstellung der Übertragungsart

Übertragung Setup

Die Einstellungen in diesem ersten Abschnitt (Tab) sagen dem Analyser alles was er über die physikalische Übertragungsart des angeschlossenen Feldbus wissen muss! Im OSI Modell umfasst dies die Bitübertragungsschicht und Sicherungsschicht.
Dies sind - vor allem - die Art der Übertragung und davon abhängig die entsprechenden Übertragungsparameter.
Der MSB-RS485-PLUS ermöglicht die Aufnahme von asynchronen und synchronen (SSI, Manchester) Bus Übertragungen[1]. Beide Arten sind völlig unterschiedlich und müssen separat spezifiziert werden.

Es ist hierbei absolut notwendig die richtigen Werte für die jeweiligen Übertragungsparameter einzustellen, da der Analyser ansonsten keine korrekten Daten aus dem Bitsignal extrahieren kann!

Beachten Sie dazu: Der Analyser kann die nötigen Parameter automatisch ermitteln, siehe Abschnitt Asynchrone Protokoll Ermittlung, Synchrone Protokoll Ermittlung und Manchester Protokoll Ermittlung. Allerdings müssen Sie diese explizit bestätigen, bevor Sie mit einer Aufnahme beginnen!
Der Grund hierfür: Eine automatische Ermittlung und Anpassung von Einstellungen wie z.B. der Bitrate während einer aktiven Aufnahme führt unweigerlich zu Datenverlusten, da die Hardware während des Umschaltvorgangs keine gültigen Daten erfassen kann.
Die bei weitem meisten Feldbusse basieren auf asynchrone Übertragungen. Wir starten deshalb mit dieser Art Übertragung.

8.2.1.1 Asynchrone (UART) Übertragung

Eine asynchrone Übertragung ist gekennzeichnet durch ein Bit- bzw. Baudrate und einem bestimmten Datenformat. Letzteres spezifiziert die Anzahl der verwendeten Datenbits, ein optionales Prüf- bzw. Paritätsbit und Anzahl von Stopbits. Diese Angaben werden zumeist in der Form 38400 8N1 oder 9600 7E2 angegeben. Wie bereits erwähnt: Der Analyser kann diese für Sie ermitteln oder Sie geben diese manuell in diesem Einstelldialog ein. Die nötigen Parameter sind:

[1] Weitere sind in Vorbereitung.

1 **Bitrate** - oder Baudrate rate, gibt die Übertragungsgeschwindigkeit an.
Der MSB-RS485-PLUS Analyser unterstützt zusätzlich zu den Standard Baudraten beliebige Baudraten im Bereich von 1Baud bis zu 20 MBaud.
Geben Sie einfach Ihre gewünschte Baudrate ein oder wählen Sie sie aus einer Liste der standardisierten Baudraten aus indem Sie auf den ▾ Knopf klicken.
Erlaubt sind alle Eingaben von 1 bis 20 MBaud.

2 **Datenbits** - die Anzahl der pro Byte übertragenen Bits.
Neben den üblichen Datenlängen 5...8 erlaubt der Analyser auch die Aufzeichnung von Verbindungen mit 9 Bit Daten wie sie gerne zur Adressdekodierung bzw. zur Kennzeichnung eines neuen Datentelegrammes verwendet werden.
Beachten Sie, das bei der Angabe von 9-Bit Daten die Parität ausgeschaltet bzw. fest auf 'Nein' gesetzt wird.

3 **Parity** - Prüfbit des gesendeten Datenbytes zur Fehler Erkennung.
Die Parity (oder Paritäts) Einstellung beeinflusst nicht die Ermittlung der Daten aus dem Bitstrom. Allerdings sollten Sie hier die gleiche Vorgabe (none, odd, even, mark oder space) wie in dem zu untersuchenden Bus verwenden um falsche Parity Fehler zu vermeiden.

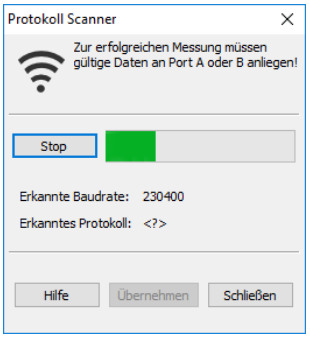
4 **Stop Bits** - wird vom Analyser automatisch behandelt.
Anwender fragen oft, wo die Anzahl der Stopbits (1 oder 2) angegeben wird. Die kurze Antwort ist: Die korrekte Anzahl der Stopbits ist nicht von Belang und Sie müssen sich nicht darum kümmern.

Und im Detail:
Während das Startbit den Beginn eines Datenrahmens einleitet, indem es vom Ruhepegel zum Aktivpegel wechselt (logisch 1), kennzeichnet das Stopbit das Ende des Datenrahmens durch die Rückkehr auf den Ruhepegel (logisch 0).
Zur richtigen Kodierung und Markierung des nächsten Datenrahmens ist ein Stopbit ausreichend. Dadurch ist sicher gestellt, das das nächste Startbit erneut aus einem Ruhepegel zum aktiven Pegel wechseln kann. Eine größere Anzahl Stopbits gibt dem Empfänger lediglich mehr Zeit zur Dekodierung (bevor das folgende Startbit einen weiteren Datenrahmen einleitet).
Für die korrekte Datenverabeitung durch den Analyser ist die Kenntnis der Anzahl von Stopbits jedoch unnötig. Stopbits sind für ihn lediglich ein Ruhepegel der ignoriert wird, bis der Empfang des nächsten Startbits die nächste Datenbyte Dekodierung einleitet.

Stop bits
are handled automatically

Automatische Asynchrone Protokoll Ermittlung

Der MSB-RS485-PLUS Analyzer enthält mit dem sogenannten FLEXUART Kern eine eigene Dekodiert-Hardware für die serielle Datenübertragung, die neben der Messung der Baudrate auch das automatische Erkennen des verwendeten Protokolls erlaubt.
Das einzige, was Sie tun müssen, ist den Knopf zu klicken um den Protokoll Scanner Dialog zu öffnen.
Eine korrekte Detektion der Verbindungsparameter setzt natürlich eine entsprechende Datenübertragung voraus. Dabei reicht es, die Daten an einem der Eingangskanäle CH1 oder CH2 anzulegen. Eine Aufzeichnung muss hierzu nicht extra gestartet werden.
Starten Sie die automatische Erkennung mit einem Klick auf den 'Start' Knopf des Scan Dialoges.



Protokoll Scanner
Automatische Erkennung von Baudrate, Datenlänge und Parität

Bei der Protokoll Erkennung misst der MSB-RS485-PLUS Analyser zunächst die Übertragungsrate der an CH1 oder CH2 angelegten Daten. Im weiteren Verlauf wird der Datenstrom analysiert und die korrekte Anzahl von Start, Datenbits und Parität ermittelt.
Der komplette Vorgang dauert nur wenige Sekunden und kann jederzeit durch erneutes Klicken des 'Start' Knopfes wiederholt werden. Sobald die Parameter korrekt ermittelt wurden, können Sie die gefundene Einstellung mit dem 'Übernehmen' Knopf übernehmen.

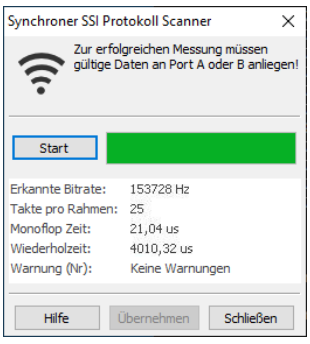
8.2.1.2 Synchrone SSI Übertragung

SSI ist eine serielle Punkt-zu-Punkt Übertragung und verwendet eine gemeinsame Taktleitung zur Synchronisierung der Daten zwischen den Bus Teilnehmern. Im Gegensatz zu asynchronen Bussen werden hier deshalb zwei RS485 Leitungspaare benötigt. Eine für den Takt (generiert durch den Bus Master) und eine für die Daten (erzeugt vom Slave/Sensor).
Die Daten werden innerhalb einer aktiven Takt-Phase (Rahmen) übertragen. Signifikant für die korrekte Daten Evaluierung durch die MSB-RS485-PLUS ist die Kenntnis der Anzahl von Takten innerhalb eines Datenrahmens (Datenbits). Die Einstellungen im Detail:

1 **Bit Anzahl** - oder Takte innerhalb eines Daten Rahmens
Die Anzahl der Daten Takte ist zwingend nötig um korrekte Daten zu erhalten. Wenn Sie sich über die Anzahl nicht sicher sind, können Sie den Analyser den richtigen Wert für Sie ermitteln lassen (siehe nächster Abschnitt).

2 **Abtastverschiebung** - verzögertes Abtasten der Datenbits
Wenn durch lange Leitungen Datenbits gerade an der Taktgrenze sind, kann es sinnvoll sein, das Bit um einen $\frac{1}{2}$ verzögert aufzunehmen. Sie können das im Signalmonitor leicht prüfen.

3 **Doppelte Übertragung** - zur Daten Verifizierung
Einige SSI Applikationen verwenden zur Datensicherheit eine doppelte Übertragung der Daten. In diesem Fall müssen Sie diesen Punkt aktivieren, damit der Analyzer zwei aufeinander folgende Datenübertragungen ohne Pause dazwischen als eine anzusehen ist.



Autodetect Dialog
zur Ermittlung von SSI Übertragungsparameters

Automatische SSI Protokoll Ermittlung

Wie bereits erwähnt ist der wichtigste SSI Parameter die Anzahl der Takte pro Datenrahmen. Der integrierte SSI Protokoll Scanner ermittelt diesen automatisch - er kann aber noch eine ganze Menge mehr.
Der Scanner führt eine detaillierte Timing-Analyse des angeschlossenen Busses durch und prüft die Übertragung auch auf spezielle Fehler Bedingungen. Klicken Sie einfach den Knopf zur automatischen Parameter Erkennung. Die folgenden SSI Parameter werden ermittelt:

1 **Bitrate** - Taktfrequenz oder Bitrate

2 **Takte pro Rahmen** - Anzahl der Takte in einem Datenrahmen

3 **Monoflop Zeit** - Reaktionszeit auf Rahmen Ende

4 **Wiederholrate** - Wiederholzeit der einzelnen Übertragungen

Mit Ausnahme der Bitrate (2) handelt es sich bei den anderen Parametern eher um informelle Werte. Diese können aber für die Beurteilung der Bus Qualität entscheidend sein.
Bei etwaige Unregelmäßigkeiten oder Fehler zeigt der Scanner eine entsprechende Warnung und - abhängig vom Fehler - einen Hinweis, wie dieser zu beheben ist.
Bevor Sie eine SSI Übertragung mit dem Analyser aufnehmen sollten Sie sich auf jeden Fall vergewissern, das der Scan fehlerlos verläuft!

8.2.1.3 Manchester Übertragung

Manchester verwendet zur Datenübertragung eine Kodierung, die neben der eigentlichen Dateninformation auch das Taktsignal enthält. Man spricht deshalb auch von einem selbst-taktenden Signal. Wir beschreiben Manchester detailliert in Kapitel **??**. Für eine erfolgreiche Dekodierung (und Analyse) müssen folgende Parameter vorgegeben werden:

1 **Manchester Typ**
Manchester spezifiziert insgesamt vier verschiedene Kodierungsarten. Bei den ersten beiden ist das Daten- und Taktsignal Phasen-Kodiert. Die beiden letzten sind bekannt als Differential Manchester Encoding. Für eine korrekte Datenabtastung ist es unabdingbar, den richtigen Typ anzugeben.

2 **Bitrate**
Die effektive Bitrate. Sie definiert letztendlich die Bitbreite und entspricht der halben Taktfrequenz.

3 **Bitfolge**
Gibt an, ob bei der Übertragung eines Datenbytes das höchstwertige Bit zuerst oder zuletzt gesendet wird.

Automatische Manchester Protokoll Ermittlung

Die Bitrate ist der einzige Parameter, der für alle Manchester Typen gilt!
Alle anderen Parameter (Typ und Bitfolge) können nicht automatisch detektiert werden. Sie werden durch die Anwendung/Applikation spezifiziert und können nicht alleine durch das Abtasten der Daten verifiziert werden.
Der Bitrate Scan wird durch Klick auf den Knopf zur automatischen Parameter Erkennung gestartet. Sobald ein gültiges Ergebnis vorliegt, können Sie dieses einfach übernehmen. Alle anderen Einstellungen (Typ und Bitfolge) müssen Sie entsprechend Ihrer Anwendung manuell vorgeben.

8.2.1.4 Messung der Bitrate

Das Display des Kontrollprogrammes zeigt die gemessene Bitrate beider Datenkanäle A und B. Dies ist die Voreinstellung. Dabei werden die Bitraten an CH1 und CH2 wechselseitig gemessen und das Ergebnis für die Anzeige verwendet. Die angezeigte Bitrate ist immer ein guter Indikator etwaiger Bus Aktivitäten. Sie wird kontinuierlich gemessen - auch ohne aktiver Aufnahme.
Sie können wahlweise einzelne Datenkanäle für die Messung deaktivieren. Z.B. wenn Sie definitiv nur eine Datenrichtung oder Bus messen wollen (Stichwort Segment Modus). Oder Sie deaktivieren die Messung komplett (einige synchrone Bus Systeme arbeiten ohne konstante Bitraten, hier ist die Anzeige ohne Information).

Bus Anschluss
Einstellungen

8.2.2 Bus Anschluss

Der MSB-RS485-PLUS Analyser verfügt über 4 Differenzsignal-Eingänge, die abhängig von der Anschlussart als Daten- bzw. Signalquellen verwendet werden. Maßgeblich für die interne Signalzuordnung ist deshalb, wie Sie den MSB-RS485-PLUS Analyser mit dem zu untersuchenden Bus verbunden. Da diese nicht automatisch ermittelt werden kann, müssen Sie die Anschlussart (Wiring) dem Analyzer mitteilen. Dies gilt im übrigen für alle Übertragungsarten!

Je nach Anschlusseinstellung wird nur ein Teil der Differenzsignal-Eingänge verwendet. Die unbenutzten Eingänge sowie die digitalen Hilfseingänge können Sie frei für Ihre Applikation verwenden. Nicht zur Verfügung stehende Signale werden mit 'deaktiviert' gekennzeichnet.
Das Einstellmenü zeigt für jede Anschlussart eine entsprechende Grafik sowie eine Tabelle der Signalzuordnungen. Die Signalnamen werden dabei automatisch zugewiesen. Im Einstellmenü der Signal Namen (siehe Seite 57) können Sie dieses Verhalten ändern und eigene Namen vergeben. Die Signalzuordnung ist unterteilt in:

- **2 Datenkanäle**
  Diese enthalten die dekodierten Daten der beiden USARTs. 9 Bit Daten werden ebenso unterstützt und angezeigt wie aufgetretene Übertragungsfehler (Parity, Frame). Datenrahmen mit größerer Bitanzahl wie z.B. bei synchronen Übertragungen werden in aufeinander folgende Datenbytes aufgeteilt.
  Abhängig von der Anschlussart kann ein Datenkanal auch die Daten mehrerer Differenzeingänge enthalten. In diesem Fall werden die entsprechenden Eingänge explizit mit einem '+' aufgelistet, beispielsweise CH1+CH2.
- **8 Logische Signalkanäle**
  Alle Differenzsignal-Eingänge CH1...CH4 werden unabhängig von Ihrer Datendekodierung als 'reines' Logiksignal bereit gestellt. Die vier Eingänge sind dabei fest auf die Signalkanäle 1 bis 4 abgebildet.
  Die Signalkanäle 5 und 6 besitzen einen Sonderstatus und stellen bei gewählter Segmentanalyse zusätzliche 'intern' erzeugte Logiksignale zur Verfügung. Es handelt sich dabei um die Bus-Richtung zwischen CH1 $\leftrightarrow$ CH2 (Signalkanal 5) sowie die Vereinigung der an beiden Eingängen CH1 und CH2 auftretenden Logiksignale (Signalkanal 6).
  Die Signalkanäle 7 und 8 sind den beiden digitalen Hilfseingängen zugeordnet.
  Bei eingestellter synchrone Bus Übertragung zeigen die Signale 5 bis 8 sowohl die Gültigkeit/Dauer der Datenrahmen (oder Dataframes) als auch eine Kombination der Daten- und Taktleitung an. Dabei sind Signale 5,6 dem ersten synchronen Bus und die Signale 7,8 dem zweiten Bus zugeordnet.
- **Digitale Hilfs-Ein/Ausgänge**
  Wie der Name bereits impliziert können beide Kanäle individuell als Ein- oder Ausgang betrieben werden. Damit ist sowohl die Aufzeichnung zusätzlicher digitaler Signale wie auch die Ausgabe des Bus-Status (Bus-Gültigkeit, Aktivität und Richtung) möglich. Sinnvoll, wenn Sie ein Signal zur Triggerung externer Messeinrichtungen benötigen.
  Per Voreinstellung sind beide als offene Eingänge mit Pulldown Widerstand geschaltet. Eine detailierte Beschreibung der verschiedenen Einstellungen finden im Abschnitt 6.4.

Digitale IOs bei synchroner Bus Analyse

Bei der Aufzeichnung/Analyse von synchronen Bussen kommt den beiden digitalen Hilfs-Ein/Ausgängen eine Sonderstellung zu. Im Gegensatz zur asynchronen Bus Analyse können diese nämlich nicht als Eingänge konfiguriert werden.
Der MSB-RS485-PLUS benutzt diese IO Kanäle deshalb als reine Signalausgabe um externe Messgeräte (z.B. ein Digitalscope) auf Übertragungsfehler oder den Beginn eines Datenrahmens zu triggern [2].

1 **Ausgabe Fehlersignal** - Positiver Pegel $(+5V)$ für mindestens $5\mu s$
2 **Ausgabe Rahmensignal** - Positiver Pegel $(+5V)$ während Datenrahmen

8.2.3 Signale

Signale auswählen und individuelle Namen vergeben

Alle vom MSB-RS485-PLUS Analyser abgetasteten Leitungen bzw. zur Verfügung gestellten Signale können einzeln für die Aufnahme aktiviert und mit einem individuellen Namen versehen werden. Letzteres ist im übrigen auch während einer laufenden Aufnahme möglich.
Die gewählte Bus-Anschlussart gibt eine sinnvolle Signalbezeichnung vor (Vorgabe Bus-Anschluss). Sie können dies aber einfach ändern indem Sie statt dessen 'Benutzer definiert' wählen. Als Platzhalter dienen die Bezeichnungen Sig1...8.
Jede Name kann maximal aus 7 Zeichen bestehen, erlaubt sind alle Zahlen und Buchstaben, der Unterstrich, Doppelpunkt und Punkt.
Der eingegebene Signalname wird sofort vom Program übernommen und von den offenen Views (Analyse Fenster) aktualisiert.

Die Aufzeichnung der Pegelveränderungen können für jede Leitung einzeln ein- oder ausgeschaltet werden indem Sie das 'Häckchen' Aufzeichnen neben dem Signalnamen setzen bzw. entfernen. Als Vorgabe sind alle Pegeländerungen an den Differenzsignal-Eingängen CH1 bis CH4, den beiden Digitaleingängen sowie die Dekodierung der seriellen Datenströme durch die beiden USARTS aktiviert.
Beachten Sie, dass hier die von Ihnen vergebenen Signalnamen stehen. Sollten Sie 'Benutzer definiert' gewählt aber keine Namen eingetragen haben, erscheinen hier - keine Namen!

Die Dekodierung der Daten durch die UARTs erfolgt unabhängig von der Aufzeichnung der Pegelwechsel. Wenn Sie nur die Daten aber nicht das logische Signal benötigen, können Sie die Aufzeichnung der entsprechenden Kanäle CH1...CH4 auch deaktivieren, da jeder Pegelwechsel als zusätzliches Ereignis gespeichert wird und dadurch die Datenmenge deutlich erhöht.

Generell gilt: Je mehr Ereignisse Sie aktivieren desto mehr Daten fallen für die Aufzeichnung an.

8.2.4 Aufnahme

Bei der Fehlersuche in seriellen Verbindungen stellt sich oft das Problem, das Sie nicht wissen, wann der Fehler auftritt, sie aber zum Zeitpunkt des Fehlers

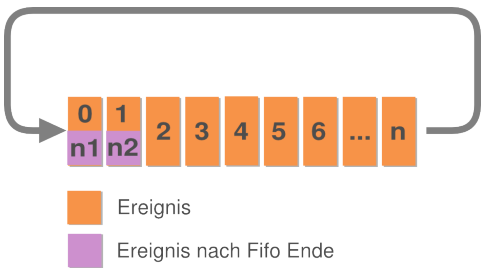
[2] Siehe Kapitel 7.4 für weitere Details

Aufnahmeart wählen
Synchrones Aufzeichnen, kontinulierliches Aufnehmen oder Aufnahmeschleife

eine entsprechend große Datenmenge benötigen, um Aussagen über eventuelle Ursachen machen zu können.
Sie können natürlich die Aufzeichnung einfach solange laufen lassen, bis der Fehler auftritt, was allerdings je nach Aufzeichnungskriterien eine schnell wachsende Datenmenge verursacht (bei der Aufzeichnung aller Pegelwechsel in einer 115200 schnellen Verbindung sind das über 2 MByte Daten pro Sekunde).
Der Analyser unterstützt deshalb zwei Aufzeichnungsarten:

1 **Kontinuierliche Aufzeichnung:**
Im kontinuierlichen Modus werden alle auftretenden Daten gespeichert, bis die Aufzeichnung beendet wird. Dieser Modus ist immer dann sinnvoll, wenn Sie bereits während der Aufzeichnung den Datenstrom beobachten oder analysieren wollen.

2 **Aufnahmeschleife mit Fifo Modus:**
Im Fifo Modus wird eine bestimmte Anzahl der zuletzt (vor Ende der Aufzeichnung) aufgetretenen Ereignisse gespeichert. Sie wird vorgegen durch eine Maximalanzahl (1000...1000000 Ereignisse) oder als maximale Zeit im Bereich von 10...600 Sekunden.
Dies entspricht quasi dem Verhalten eines analogen Endlosbandes, (wie Sie z.B. bei analogen Überwachungskameras üblich sind), bei dem immer die durch die Bandlänge definierte zuletzt verstrichene Zeit aufgenommen wird. Allerdings können Sie hier die Bandlänge in einem vorgegebenen Bereich frei wählen.



Beachten Sie, dass im Fifo Modus während der Aufzeichnung keine Analysetools verwendet werden können. Der Grund hier liegt darin, dass die Analysetools einen wahlfreien Zugang zu den aufgenommenen Daten erfordern, was im Fifomodus nicht gegeben ist. Da der Fifomoduls allerdings hauptsächlich zur Aufzeichnung mit 'späterer' Auswertung verwendet wird, ist das nicht unbedingt von Nachteil.
Sobald Sie die Aufnahme stoppen werden die aufgezeichneten Daten automatisch normalisert, d.h. in ihre richtige zeitliche Abfolge gebracht und können dann wie üblich analysiert werden.

Zwei Analyser synchronisieren

Per Voreinstellung arbeitet jeder MSB-Analyser autonom solange er nicht über die MSB-Link Buchse Synchron-Impulse eines verbundenen 'Masters' empfängt. Sobald Sie zwei unabhängige Verbindungen gleichzeitig aufnehmen wollen, beispielsweise den RS232 und RS485 Anschluß eines Pegelwandlers, müssen Sie einen der beteiligten MSB-Analyser als Aufnahme 'Master' definieren.
Ob das Gerät im Master oder Slave Betrieb arbeitet erkennen Sie an der entsprechenden Kennung in der Kontrollanzeige. Da zwei Geräte nicht gleichzeitig als Master arbeiten können, wird die Einstellung beim Slave mit Empfang des ersten Synchron-Impulses über die MSB-Link Buchse deaktiviert.

Benutzen Sie den 'Zeige verbundenen Analyser' Knopf, wenn Sie sich nicht sicher sind, welcher Analyser von den Einstellungen betroffen (d.h. mit dem Programm verbunden) ist.

Aufzeichnungsdatum anpassen

Aufnahmedatum und Zeit richten sich immer nach der aktuellen Zeit und lokalen Zeitzone. Manchmal ist es deshalb nötig, Zeit und Datum nachträglich anzupassen, z.B. wenn Sie zwei Aufzeichnungen zeitlich anpassen wollen oder unterschiedliche Zeitzonen berücksichtigen müssen.

Klicken Sie den Knopf 'Jetzt anpassen' und wählen Sie das gewünschte Datum sowie die gewünschte Uhrzeit in den Stunden, Minuten und Sekunden Feldern.

Bestätigen Sie die Änderungen mit Ok. Die neue Datumzeit wird sofort auf jedes offene View angewendet. Sie können das ursprüngliche Aufnahmedatum jederzeit in dem Eingabedialog wieder herstellen.
Bei der Anpassung handelt es sich lediglich um einen internen Offset, die Aufnahme selbst wird dadurch nicht verändert!

Speichern des neuen Aufzeichnungsdatum

Die Änderung des Augfzeichnungsdatums bzw. Zeit wirkt sich nicht auf die Aufzeichnungsdatei aus und ist lediglich temporär für die aktuelle Sitzung. Um die Änderung permanent zu sichern müssen Sie die Aufzeichnung speichern.

8.2.5 Autospeichern

Die vom Analyser aufgenommenen Datenmengen können schnell mehrere hundert Megabyte erreichen. Die MSB-Analyser Software speichert die Daten deshalb nur explizit wenn Sie das wünschen.
Allerdings gibt es Situationen die eine Speicherung ausdrücklich erfordern. Z.B. wenn Sie mehrere aufeinander folgende Aufnahmen zur späteren Auswertung machen wollen oder ein im Slave Modus arbeitender Analyser nicht direkt zugänglich ist.
In beiden Fällen können Sie eine automatische Speicherung der Aufnahme voreinstellen. Die Speicherung erfolgt wahlweise bei:

Aufnahme speichern
automatisch nach
Aufnahmestop

1 Nach Stop einer Synchronaufnahme
2 Nach jedem Stop

Ort und Verzeichnis sind frei wählbar. Der Dateiname wird vom Programm automatisch aus Seriennummer des Analysers und Startdatum der Aufnahme gebildet um Überschreibungsfehler zu vermeiden. Sie können allerdings einen beliebigen Prefix zur besseren Kennzeichnung der Aufnahmedateien voranstellen.

8.2.6 Allgemein

Generelle Einstellungen
Warnungen, Taskleiste, externe Synchronisation

Diese Seite ist für allgemeine Einstellungen vorgesehen. U.a. können Sie hier die Sicherheitsabfrage bei ungesicherten Daten ein/ausschalten (Vorgabe ist ein) sowie die automatische Speicherung der aktuellen Sitzung bei Programmende (de)aktivieren (Vorgabe ist ebenfalls ein, sodass die aktuelle Sitzung beim nächsten Programmstart wiederhergestellt wird.

Die MSB-Analyser Software erlaubt die 'externe' Synchronisation der Views durch eine andere, parallel laufende Analyser Applikation. Dies ist z.B. sinnvoll, wenn Sie zwei synchron aufgenommene Aufzeichnungsdateien in zwei Programmen parallel untersuchen und die Views zwischen beiden sychronisieren wollen wie Sie es bereits innerhalb einer Aufnahme gewohnt sind.
In diesem Fall müssen Sie die externe Synchronisation explizit erlauben.

8.3 Aufzeichnung starten

Record Pause Stop
Aufnahmekontrolle

Sobald Sie Anschlussart und Übertragungsparameter festgelegt haben reicht ein Klick auf die Aufnahmetaste (Record) um die Aufzeichnung zu starten. Der Aufnahmeknopf beginnt rot zu leuchten und das Aktivsymbol im Display sich zu drehen.
Mit der Pause Taste können Sie die Aufnahmejederzeit anhalten und zu einem späteren Zeitpunkt durch erneutes Klicken des Aufnahmetaste fortsetzen.
Um die Aufzeichnung endgültig anzuhalten, genügt ein Mausklick auf die Stop Taste.
Die MSB-RS485-PLUS Analyser Software erlaubt Ihnen bereits während der Aufzeichnung die Untersuchung der Daten. Insofern werden Sie eine Aufzeichnung in der Regel nur dann stoppen, wenn Sie eine neue Aufnahme starten wollen, oder um sie zur späteren Auswertung zu speichern.
Alle weiteren Fenster/Tools zur Anzeige der übertragenen Daten oder deren physikalischer Ebene können wahlweise geöffnet oder geschlossen werden, ohne die Aufzeichnung zu beeinflussen.

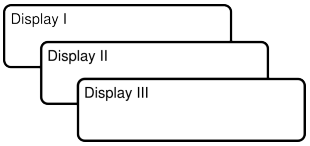
MSB-RS485-PLUS Software ohne Analyser testen

Sie können das Programm auch ohne angeschlossenen Analyser ausprobieren. Laden Sie dazu einfach per Strg+O eine Bespielaufzeichnung aus dem Beispiel (examples) Ordner des Installationsverzeichnis.

8.4 Display Anzeige

Zentraler Bestandteil des Kontrollprogrammes ist die Anzeige der aktuellen Aufzeichnung (Display).

Um alle Informationen übersichtlich darstellen zu können, lässt sich die Anzeige in drei verschiedenen Darstellungen betreiben. Die Umschaltung erfolgt einfach per rechtem Mausklick auf das Display.
Zusätzlich zu den Verbindungsdaten wird auch die verfügbare Aufzeichnungskapazität dargestellt. Die dazu benötigte Festplattenkapazität hängt sehr stark vom Datenaufkommen und den ausgewählten Ereignissen, die aufgezeichnet werden sollen, ab. Eine Abschätzung des Verbrauchs liefert der Marker (Punkt) auf der horizontalen Trennlinie. Er zeigt den noch freien Platz (rechts von der Markierung) im Verhältnis zur insgesamt verfügbaren Kapazität an.
Voreingestellt ist das unter Windows bzw. Linux übliche temporäre Verzeichnis. Sie können dies allerdings auch per Programmaufruf (siehe Abschnitt 23.5.4, zusätzliche Programm Argumente) ändern.
Die Geschwindigkeit, mit der die Markierung nach rechts wandert ist abhängig von der Anzahl der auftretenden (und ausgewählten) Ereignissen sowie der Größe des im verwendeten temporären Verzeichnisses noch freien Platzes.



Anzeige umschalten
mit rechtem Mausklick

8.4.1 Anzeige I

Die Default Ansicht, wenn Sie das Programm gestartet haben. Sie informiert Sie über die verstrichene Aufnahmezeit, die gewählte Übertragungsart und Parameter (hier Asynchron, Datenformat, Bitrate), sowie zum Vergleich die gemessene Bitrate (Mete) mit prozentualer Abweichung zur eingestellten Rate.
Unter der Linie des Indikators für den noch freien Aufnahmeplatz sind die aktiven Eingangskanäle bzw. Signale zu sehen, die Sie für die Aufnahme aktiviert haben.
Die Anzahl der übertragenen Datenbytes abhängig von Richtung, Bus oder Bus-Segment wird auf der rechten Seite angezeigt.

8.4.2 Anzeige II

Die zweite Anzeige informiert Sie über die Gesamtzahl der übertragenen Datenbytes sowie der Anzahl der übrigen Ereignisse wie z.B. Pegelveränderungen, aber auch vom Analyser generierte Ereignisse wie Änderung der Bus-Richtung, Datenrahmen Gültigkeit etc.
Beachten Sie: In diesem Anzeige Modus wird als Zeitangabe das zuletzt aufgetretene Ereignis verwendet, nicht die laufende Aufzeichnungsdauer.

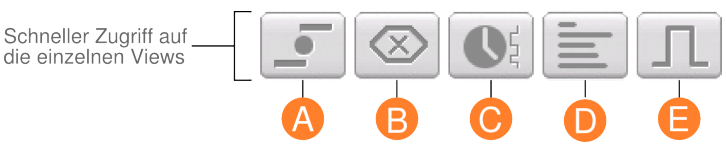
8.4.3 Anzeige III

Die dritte Anzeigevariante dient zur Kontrolle der Verbindung zum Analyser. Der Analyser MSB-RS485-PLUS wird über eine USB Verbindung versorgt und gesteuert. Die Kommunikation erfolgt über einen USB Direkt Link mit High Speed 480 MBit.

Die Felder *Gaps* und *Fifo* zeigen Ihnen an, ob der Analyser mehr Daten aufgezeichnet hat, als der PC (auf Grund eines zu langsamen Anschlusses) verarbeiten konnte. Normalerweise sollten beide Felder den Wert 0 enthalten. Abweichende Werte bedeuten, dass entweder der interne Puffer des Analysers (Gaps) bzw. die Hardware oder der Treiber das Datenaufkommen nicht verarbeiten konnte. In diesem Fall sollten Sie die Anzahl der aufzuzeichnenden Ereignisse reduzieren.
Alle Signal- und Datenleitungen können individuell zur Aufzeichnung (de)aktiviert werden. Nicht eingeschaltete Leitungen werden durch Striche dargestellt. Im obigen Beispielen werden die Pegelwechsel der Eingangskanäle CH3 und CH4 nicht aufgenommen.

8.5 Die Analysetools

Das Kontrollprogramm stellt lediglich die vom Analyser gelieferten Verbindungsdaten zur Verfügung. Die eigentliche Darstellung und Analyse der Daten erfolgt durch Analysetools - separate Programme, die die Daten an unterschiedlichen Stellen und in verschiedenen Darstellungsarten visualisieren.
Sie können beliebig viele Analysefenster öffnen indem Sie entweder die unten aufgeführten Kurzbefehle verwenden, oder auf einen der Schnellstart Knöpfe rechts neben dem Displayfeld des Kontrollprogrammes klicken.



A	(S.83)	**Virtueller Ledtester:** Das virtuelle Gegenstück zu einem Ledtester.
B	(S.85)	**Datenmonitor:** Datendump der übertragenen Daten mit umfangreichen Suchmöglichkeiten.
C	(S.111)	**Ereignismonitor:** Alle Leitungsveränderungen übersichtlich dargestellt, Suche nach Pegelveränderungen.
D	(S.125)	**Protokollmonitor:** Darstellung von Datensequenzen mit eigenen Protokolldefinitionen.
E	(S.203)	**Signalmonitor:** Digital Scope ähnliche Visualisierung aller Leitungen.

8.6 Ein Aufzeichnung speichern

Unabhängig von Status der Aufzeichnung (aktiv, pausierend oder gestoppt) können die bis dato angefallenen Daten jederzeit in einer Datei gespeichert werden.

Drücken Sie die Tasten Strg+S oder wählen Sie im Menü Datei den Menüpunkt Speichern→Speichere Aufzeichnung. In dem sich öffnenden Dialog können Sie entweder einen neuen Dateinamen eingeben (als Endung wird automatisch .msblog hinzugefügt) oder eine bereits vorhandene Aufzeichnung überschreiben.

Diese Datei enthält dann alle Informationen über die von Ihnen ausgewählten und vom Analyser aufgezeichneten Ereignisse und Datenbytes sowie die nötigen Baudrate/Protokoll Informationen. Einstellungen des Kontrollprogrammes oder evtl. geöffneter Analysefenster werden separat als Projektdatei gespeichert.

Jedesmal, wenn Sie eine Aufzeichnung sichern, wird die von Ihnen ausgewählte Datei in einer Liste zuletzt geöffneter Aufzeichnungsdateien gespeichert und kann von dort jederzeit wieder geladen werden. Nähere Informationen dazu finden Sie im Abschnitt zuletzt geöffnete Aufzeichnungen und Projekte.

Speicherung beliebiger Ausschnitte

Um einen beliebigen Ausschnitt aus den aufgenommenen Daten zu speichern verwenden Sie den Ereignismonitor und markieren dort die gewünschten Bereiche. Um nur die übertragenen Daten oder einen Ausschnitt aus ihnen zu speichern (z.B. um Sie zu vergleichen) öffnen Sie den Datenmonitor und selektieren dort den entsprechenden Bereich.

8.7 Ein Sitzung als Projekt speichern

Eine Sitzung bezeichnet den aktuellen Stand des laufenden Analyser Programms und enthält alle momentanen Einstellungen und Ansichten in denen

sich das Programm auf dem Bildschirm präsentiert. D.h. neben den Verbindungseinstellungen des Kontrollprogrammes auch Position, Größe und Inhalt aller geöffneten Analysetools. Darüber hinaus alle markierten Regionen.
Sie können die aktuelle Sitzung jederzeit sichern indem Sie die Tasten Strg+Umschalt+S drücken oder im Dateimenü dem Menüpunkt Speichern→Speichere Projekt ... auswählen.
Beim Speichern einer Sitzung werden immer auch die bis dahin aufgenommenen Daten in einer separaten Aufzeichnungsdatei mit gleichen (Projekt) Namen, aber anderer Endung gespeichert.

Getrennte Projekt- und Aufzeichnungsdateien

Projektdateien haben immer die Endung $\star$.msbprj, die Aufzeichnungsdateien die Endung $\star$.msblog.

Entsprechend wird beim Öffnen einer zuvor gespeicherten Projektdatei eine (falls vorhandene) Aufzeichnungsdatei mit gleichem Namen geladen.
Damit haben Sie alle Informationen, die Sie benötigen, um eine Untersuchung der aufgezeichneten Daten exakt an der Stelle wieder aufnehmen zu können, an der Sie diese beendet bzw. unterbrochen haben.
Gespeicherte Projekte werden ebenfalls in einer Liste zuletzt geöffneter Projektdateien verwaltet, siehe hierzu zuletzt geöffnete Aufzeichnungen und Projekte. Jede Sitzung kann als eigenständige Projektvorlagen gespeichert werden.
Löschen Sie dazu die aktuell aufgenommenen Daten indem Sie im Dateimenü den Punkt Neu→Neue Aufzeichnung auswählen oder einfach die Tasten Strg+N drücken. Anschließend speichern Sie die Sitzung unter einem Dateinamen Ihrer Wahl.

8.8 Eine frühere Aufzeichnung öffnen

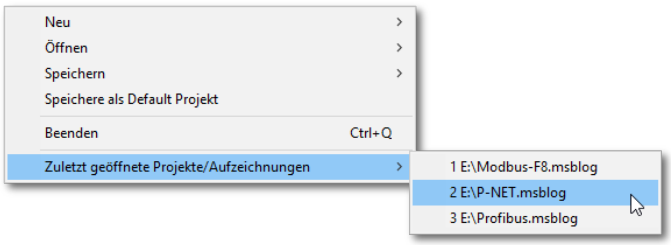
Die Unterteilung zwischen Projektdateien und Aufzeichnungen ist bewusst gewählt worden. Dadurch können Sie jederzeit eine frühere Aufzeichnung in ein aktuelles Projekt laden ohne dabei ihre aktuellen Einstellungen zu verlieren.
Drücken Sie die Taste Strg+O oder klicken Sie im Dateimenü den Punkt Öffnen→Öffne Datenaufzeichnung um eine Aufzeichnung in Ihre aktuelle Sitzung zu laden. Beachten Sie, dass dies nur möglich ist, wenn Sie keine aktive Aufzeichnung läuft und das dabei ihre dato aufgenommenen Daten überschrieben werden.

8.9 Eine frühere Sitzung (Projekt) öffnen

Drücken Sie die Tasten Strg+Umschalt+O oder klicken Sie im Dateimenü den Eintrag Öffnen→Öffne Projekt um eine gespeicherte Sitzung erneut zu öffnen. Das Kontrollprogramm lädt die zugehörige Aufzeichnung, platziert die zum Zeitpunkt der Speicherung aktiven Analysetools und übernimmt die entsprechenden Einstellungen - kurzum restauriert den Programmstatus, wie er beim Speichern der Sitzung vorherrschte.

8.10 Zuletzt geöffnete Aufzeichnungen und Projekte

Alle von Ihnen gespeicherte Aufzeichnungen und Sitzungen (Projekte) werden in zwei separaten Listen vermerkt. Sie haben damit einen sehr schnellen Zugang gerade auf die Dateien, die Sie zuletzt verwendet haben.
Die Aufzeichnung- und Projektlisten enthalten die einzelnen Einträge in der Reihenfolge ihrer Verwendung, wobei die Dateien jüngsten Datums an oberster Stelle stehen. Alle Dateien werden mit vollem Pfad gelistet um deren Unterscheidung zu erleichtern.



Klicken Sie im Dateimenü den Punkt 'Zuletzt geöffnete Aufzeichnungen' und wählen Sie die von Ihnen gewünschte Aufzeichnung aus. Dies entspricht dem Öffnen eines Aufzeichnung mit dem Aufzeichnung Öffnen Dialog, ist allerdings um einiges schneller, da Sie sich hier nicht durch verschiedene Verzeichnisbäume bewegen müssen.
Sollte die ausgewählte Datei nicht mehr verfügbar sein, z.B. weil Sie sie gelöscht haben, werden Sie vom Programm gefragt, ob die fragliche Datei aus der Liste entfernt werden soll. Wenn die Datei wirklich nicht mehr existiert, können Sie diese Frage ruhig mit 'Ja' beantworten. Befindet sich die Datei allerdings auf einem z.Z. nicht verfügbaren Datenträger (d.h. sie ist nur momentan nicht verfügbar), verneinen Sie die Frage und der Eintrag bleibt erhalten.

8.11 Drag und Drop

Sie können eine beliebige Aufnahme oder Projektdatei auch einfach per Drag und Drop in die Anwendung laden. Ziehen Sie dazu die gewünschte Rekord- oder Projektdatei aus Ihrem Dateibrowser oder von Ihrem Desktop in das Kontrollprogramm.
Die aktuelle Sitzung wird dadurch beendet und durch die Daten der ausgewählten Datei ersetzt. Handelt es sich um eine Projektdatei werden zusätzlich die entsprechenden Session Einstellungen wieder hergestellt.
Beachten Sie das während einer laufenden Aufzeichnung Drag und Drop nicht möglich ist.

8.12 Anschluß mehrerer Analyser

Sie können gleichzeitig mehrere Analyser an Ihrem PC betreiben. Außerdem ist es damit möglich, während einer Aufzeichnung die Daten bzw. Ereignisse mit denen eines anderen Analysers oder einer früheren Aufzeichnung direkt zu vergleichen. Jedes Kontrollprogramm inkl. seiner Analysetools agiert dabei völlig unabhängig voneinander.
Um ein Kontrollprogramm explizit mit einem ganz bestimmten MSB-RS485-PLUS Analyser zu verbinden, müssen Sie das Programm mit Angabe der Seriennummer des MSB-RS485-PLUS Analysers starten. Die Seriennummer befindet sich auf der Unterseite des Analysers und wird in jedem Fensterrahmen der Software eingeblendet. Sie hat die Form `MSB#####`.
Geben Sie keine Seriennummer ein, verbindet sich das Programm mit dem

ersten Gerät, welches es am USB Bus findet. Dies ist gleichzeitig das Standardverhalten.
Allerdings ist damit bei mehreren Geräten nicht gewährleistet, dass es sich dabei immer um ein- und denselben Analyser handelt.
Um per Doppelklick auf das MSB-RS485-PLUS Starticon einen ganz bestimmten Analyser auszuwählen, gehen Sie wie folgt vor:

1. Rechtsklicken Sie das MSB-RS485-PLUS Icon und wählen Sie den Eintrag *Kopieren*.
2. Rechtsklicken Sie eine freie Stelle Ihres Desktops und wählen Sie Einfügen um eine Kopie des Starticons anzulegen.
3. Nennen Sie die Kopie um in z.B. `MSB#####` (`#####` entspricht dabei der Seriennummer des zu verbindenden Analysers).
4. Rechtsklicken Sie das umbenannte Icon und wählen Sie den Punkt *Eigenschaften*.
5. Ergänzen Sie im Feld *Ziel*: den Aufruf des Kontrollprogrammes um den Programmparameter `-nMSB#####`. Also in der Art:
   `C:\Program Files (x86)\msb-7.0.2\msb_serv.exe -nMSB#####`
   resp. für Linux:
   `~/msb-7.0.2/msb_serv -nMSB#####`
6. Klicken Sie OK um Ihre Ergänzung zu übernehmen.

Beachten Sie, dass die Seriennummer exakt mit der Ihres Analysers übereinstimmt. Ansonsten wird das Gerät beim Start der Software nicht gefunden und Sie erhalten eine Fehlermeldung. Auf diese Weise können Sie für jeden MSB-RS485-PLUS Analyser ein eigenes Starticon definieren.

8.13 Automatisches Starten nach Rechner Reboot

Der Analyser kann automatisch nach Hochfahren (booten) des Rechners gestartet und in den Aufnahmemodus versetzt werden. Damit der Analyzer allerdings automatisch nach Start und Laden der Firmware in den Aufnahmemodus schaltet, muß dem Programm ein zusätzlicher Parameter übergeben werden. Gleichzeitig bewirkt die Angabe dieses Parameters, daß bei System Shutdown die Aufzeichnung korrekt abgeschlossen und gespeichert wird.
Im einzelnen bedeutet dies:

1. Sobald der Anmeldeprozess (Login) abgeschlossen ist, wird nach einem angeschlossenen MSB-RS485-PLUS Analyser gesucht und die Firmware übertragen.
2. Der Analyser schaltet anschliessend in den Aufnahmemodus und beginnt mit der Aufzeichnung der Verbindung. Dabei werden automatisch die zuletzt verwendeten Einstellungen berücksichtigt.
3. Jede neue Aufzeichnung wird in einer eigenen Logdatei abgelegt, die sich aus der Seriennummer des Analysers und dem aktuellen Datum und Uhrzeit des Beginns der Aufzeichnung zusammensetzt. Eine Aufnahme, gestartet am 2. Juli 2012 um 09:31:07 wird z.B. als `MSB01060-20120702093107.msblog` gespeichert.
4. Die Analyser Software schließt die Aufnahme automatisch, wenn der Rechner heruntergefahren wird.

Beachten Sie, daß bei nicht ordnungsgemäßem Abschalten des Rechners (ohne ihn herunter zu fahren) die letzten Ereignisse der Aufzeichnung u.U. nicht gespeichert werden können.

Autostart bei sehr großem Datenaufkommen

Bei sehr großen Datenmengen kann das Abspeichern durchaus einige Minuten dauern und den Shutdown Vorgang entsprechend verzögert. Alternativ können Sie auch die Kommandozeilen Tools aus Kapitel 23 verwenden, indem Sie im Autostart Ordner ein entsprechendes Skript bzw. eine Batchdatei ablegen.

8.13.1 Autostart aktivieren unter Windows

Windows startet selbständig alle Programme, die sich bei Anmeldung im Startup Ordner des angemeldeten Benutzers befinden (ab Windows 7). Dies gilt auch für die MSB-Analyser Software.

1. Drücken Sie die Tastenkombination ⊞ + R und geben Sie folgendes Kommando ein: `shell:startup`
   Dies öffnet das Dateifenster mit dem Startup Verzeichnis.
2. Kopieren Sie das Analyser Desktop Icon in das Startup Verzeichnis.
3. Rechtsklicken Sie das KOPIERTE Icon im Startup Verzeichnis und wählen Sie ≫ Eigenschaften ≪.
4. Ergänzen Sie den Zieleintrag (Ziel) mit dem autostart Parameter -a, also:
   `C:\Program Files (x86)\msb-7.0.2\msb_serv.exe -a`

   Klicken Sie Übernehmen und anschließend OK um Ihre Änderung zu speichern. Beim nächsten Hochfahren des Systems wird nun automatisch der Analyser gestartet und mit einer Aufnahme begonnen.

8.13.1.1 Autostart aktivieren unter Linux

Unter Linux wird ein ähnlicher Ansatz (Verzeichnis für zu startende Programme nach grafischem Login) wie bei Windows verwendet. Allerdings unterscheidet Linux zwischen System weiten und Benutzer abhängigen Programmen. Das Verzeichnis für Programme, die unabhängig vom angemeldeten Benutzer gestartet werden sollen, befinden sich unter:

```
/etc/xdg/autostart
```

Allerdings benötigen Sie für dieses Verzeichnis root Rechte. Wir empfehlen deshalb, die Analyzer Software aus dem Startverzeichnis des jeweiligen Benutzers zu starten. Insbesondere, wenn Sie die Software als normaler User installiert haben. Das Startverzeichnis (und u.U. auch der Autostart Mechanismus) können abhängig von der verwendeten Desktop Umgebung (Ubuntu, Gnome, Kde) variieren. Für die meisten Linux Systeme befindet er sich aber hier:

```
~/.config/autostart
```

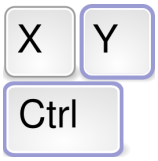
Um das Analyser Programm automatisch zu starten öffnen Sie zunächst dieses Verzeichnis in Ihrem Datei Browser und kopieren das Analyser Start Icon auf dem Desktop in dieses Verzeichnis. Anschließend rechts-klicken Sie das dort neu angelegte und wählen ≫Eigenschaften≪. Im Feld Befehl ergänzen Sie das Kommando um den Autostart Parameter `-a` wie bereits im vorherigen Abschnitt beschrieben. Das Kommando sollte dann wie folgt aussehen:

```
~/msb-7.0.2/msb_serv -a
```

Danach schließen Sie den Dialog und melden Sie erneut am System an um den Autostart zu prüfen. Ein Reboot ist nicht nötig!

8.14 Kurzbefehle

Die Kurzbefehle oder Tastenkürzel im Überblick:



Tastenkommandos der wichtigsten Funktionen

Aktion	Kurzbefehl
Online Hilfe zum Kontroll Programm	F1
Neue Aufzeichnung	Strg+N
Neues Projekt	Strg+Umschalt+N
Öffne Aufzeichnung	Strg+O
Öffne Projekt	Strg+Umschalt+O
Aufzeichnung speichern unter...	Strg+S
Projekt speichern unter...	Strg+Umschalt+S
Aufzeichnung (Record) starten	R
Aufzeichnung pausieren	P
Aufzeichnung stoppen	S
Virtueller Ledtester öffnen	Strg+Alt+L
Datenansicht öffnen	Strg+Alt+D
Ereignisansicht öffnen	Strg+Alt+E
Protokollansicht öffnen	Strg+Alt+P
Signalansicht öffnen	Strg+Alt+S
Lesezeichen (Regionen) öffnen	Strg+Alt+R
Einstellungen speichern und Programm beenden	Alt+F4

8.15 Allgemeine Programm Parameter

Das MSB-RS485-PLUS Kontrollprogramm kann mit einer Reihe von zusätzlichen Argumenten (Programm Parametern) aufgerufen werden, um z.B. eine explizite Spracheinstellung auszuwählen, den Offline Modus zu wählen, oder den angeschlossenen Analyser Typ vorgeben.
In den meisten Fällen ist die Voreinstellung (automatische Suche und Initiali-

sierung des Analysers) völlig ausreichend. Sollte der Analyser trotzdem einmal nicht korrekt erkannt werden, (dies kann z.B. bei Verwendung von Bluetooth Adaptern passieren, da diese eine Reihe von virtuellen COM Ports für sich reservieren), oder Sie möchten ein anderes Verzeichnis zur Speicherung der temporären Logdateien vorgeben, können Sie dieses Verhalten mit den folgenden Programmparametern explizit vorgeben.

Die Parameter könne Sie wie unter Anschluss mehrerer Analyser beschrieben auch Ihrem Starticon zuordnen.

Argument	Beschreibung
-a	Startet den Analyser im Autostart Modus. D.h. nach Laden der Firmware wird sofort in den Aufnahmemodus geschaltet und alle Aufzeichnungen mit einer durchlaufenden Nummer versehen.
-D *Verzeichnis*	Vorgabe des Arbeitsverzeichnis.
-e	Startet das Kontrollprogramm mit den Vorgabewerten. Alle gespeicherten Programm- und Sitzungseinstellungen werden ignoriert.
-f *firmware*	Laden einer alternative Firmware (Firmware Datei). ACHTUNG! Eine falsche/ungültige Firmware kann das Gerät beschädigen!
-i	Erzwingt das Laden der Firmware beim Start des MSB-RS485-PLUS Kontrollprogrammes (auch wenn sie bereits geladen sein sollte).
-j	Erzwingt die Öffnen der Programmfenster auf dem aktuellen Bildschirm. Verwenden Sie diesen Parameter, wenn Sie ein Projekt laden, welches auf einem System mit mehreren Bildschirmen abgespeichert wurde und die Programmfenster evtl. nicht sichtbar sind.
-l *Sprache*	Gibt die Spracheinstellung vor. Werte für *Sprache* sind: 0: Systemvorgabe, 1: Englisch, 2: Deutsch Syntax: `msb_serv -l1`
-n *Serno*	Vorgabe eines Analysers mit der Seriennummer *Serno*. Dadurch kann bei gleichzeitigem Betrieb mehrerer MSB-RS485-PLUS gezielt ein Analyser ausgewählt werden. Syntax: `msb_serv -n MSB12345`

-o *Typ*	Startet das Kontrollprogramm im 'offline' Modus für den angegebenen Analyser Typ (und unterdrückt den Auswahldialog). Ein angeschlossener Analyser wird nicht gesucht. Aufzeichnungen sind nicht möglich, allerdings können gespeicherte Projekte oder Aufzeichnungen bearbeitet bzw. untersucht oder mit einer laufenden Aufzeichnung verglichen werden. Syntax: `msb_serv -o typ`
	Erlaubte Typen sind:
	`MSB-RS232`
	`MSB-RS232-PLUS`
	`MSB-RS485`
	`MSB-RS485-PLUS`
	z.B. `msb_serv -o MSB-RS232`

-r *Zahl*	Verlangsamt den Transfer der Firmware um die angegebenen Zahl (Vorgabe ist 0, d.h. keine Verzögerung, bis maximal 100.
-T *Verzeichnis*	Vorgabe des Verzeichnisses, in dem die temporären Logdateien gespeichert werden. Per default ist das: `C:\Dokumente und Einstellungen\Benutzer\Lokale Einstellungen\Temp` (Windows) bzw. `/tmp` (Linux).
--verbose	Legt eine Reportdatei (AnalyzerScan.txt) mit Details zur Analyser Erkennung auf dem Desktop ab. Senden Sie diese Datei an support@iftools.com wenn der Analyzer nicht korrekt von der Software erkannt wird.

8.16 Spezielle Programm Parameter

Neben den 'normalen' Programm Parametern unterstützt das Kontrollprogramm auch eine Reihe von Einstellungen, die den Programmablauf im Speziellen beeinflussen.
Die entsprechenden Parameter sind im nach folgenden aufgelistet und werden vom Programm nicht gespeichert. D.h. sie müssen explizit bei jedem Programmstart übergeben werden.

Argument	Beschreibung
--exit-without-saving	Beendet das Programm ohne über ungesicherte Daten oder Einstellungen zu warnen.
--fifo-size	Gibt die Größe der Daten Fifo zwischen Analyser und Programm vor. Voreinstellung ist 10MB. Bei extrem hohen Bitraten und hoher Bus Auslastung kann die Größe erhöht werden. Allerdings ist hier der frei verfügbare Speicher zu berücksichtigen. Das folgende Beispiel erhöht die Fifo auf 700MB: `msb_serv --fifo-size=700000000`

--ignore-unsaved-data	Deaktiviert die Abfrage bei ungesicherten (d.h. noch nicht gespeicherten) Daten. Dies ist z.B. sinnvoll bei Tests die keine Speicherung der Daten erfordern.
--max-used-memory=*size*	Führt das Programm mit einem reduzierten Shared Memory Bereich aus. Die Vorgabe ist 4 GByte. Zum Beispiel: `msb_serv --max-used-memory=1000000000`
--socket=*portnumber*	Zur Kommunikation mit dem Schalteditor wird ein Socket Port im Bereich 50000...50100 verwendet. Sollte dieser Bereich bereits durch eine andere Anwendung belegt sein, können Sie einen Port explizit vorgeben. Gültige Portnummern beginnen bei 1024 und enden bei 65535. Die Portnummer 0 deaktiviert den Socket komplett, eine Verwendung des Schalteditors ist dann nicht möglich.

9

Das MultiView Konzept

Bereits während der Aufzeichnung können Sie die Daten an zeitlich unterschiedlichen Stellen in unterschiedlichen Formaten und mit verschiedener zeitlicher Auflösung untersuchen. Wir nennen das Konzept MultiView, die Aktoren Views oder AnalyseTools.

Die MSB-RS485-PLUS Analyzer Software verwendet eine Multi-Process Architektur um ein Höchstmaß an Stabilität und Skalierbarkeit zu garantieren. Aufzeichnung der Daten des per USB angeschlossenen Analysers sowie Darstellung und Auswertung erfolgen durch getrennte und unabhängige Programme/Prozesse die miteinander kommunizieren (verteilte Anwendung). Dies hat eine Reihe entscheidender Vorteile:

- Eine Aufzeichnung kann gleichzeitig an verschiedenen Stellen und in unterschiedlichsten Darstellungen untersucht werden.
- Visualisierung in Echtzeit bereits während der Aufzeichnung.
- Anzahl der Ansichten nur abhängig von Rechen-/Systemleistung (Skalierbarkeit).
- Applikationsfehler in den darstellenden Programmen wirken sich nicht auf die Aufzeichnung aus.

Durch die Fähigkeit der einzelne Programme (Views) miteinander zu kommunizieren ergeben sich zudem eine Reihe von Möglichkeiten, die die Analyse einer EIA-422/485 Datenübertragung deutlich vereinfachen.
So können die verschiedensten Ansichten der Aufzeichnung miteinander *verschränkt* werden. Was bedeutet das?
Jedes Anzeigeprogramm kann als *Master* ausgewählt werden, worauf alle anderen Datenansichten automatisch diesem folgen und entsprechend ihre Anzeige aktualisieren. Zum Beispiel:
Der Graph des physikalischen Datensignals (Logikpegel Ansicht) folgt dem Cursor des Datenmonitors oder umgekehrt.
Die Suche nach einem bestimmten Pegelwechsel oder einer Datenpause blendet die zugehörigen Datensequenzen ein. Der Klick auf ein aufgezeichneten Parity Fehler zeigt das entsprechende Signal, etc...

9.1 Synchronisierung

Die Art, miteinander zu kommunizieren, wird als Synchronisierung bezeichnet, die Bedienung ist für alle Views identisch.
Jedes Anzeigeprogramm kann dabei wahlweise der aktuellen Aufzeichnung folgen und damit immer das zuletzt aufgetretene Ereignis (Datenbyte, Pegelwechsel) darstellen. Oder den momentanen Ausschnitt sperren, z.B. um diesen mit einem anderen Ausschnitt der Aufnahme zu vergleichen.
Ist das Anzeigeprogramm auf verschränkt geschaltet, reagiert es auf alle Synchron Anforderung, die andere Views auslösen und blendet den entsprechenden Ausschnitt der Aufzeichnung in der eigenen Darstellung ein. Dabei ist das Programm, mit welchem der Anwender gerade interagiert automatisch der *Master*.
Mit diesem einfachen Konzept lassen sich beliebige Ansichten miteinander koppeln (*verschränken*). Und dies völlig unabhängig von der laufenden Aufzeichnung.

Synchron Ansicht
individuell für jedes View

Symbc	Bedeutung	Beschreibung
	Folgen (Autoscroll)	die Anzeige folgt der Aufzeichnung und blendet immer die zuletzt aufgenommenen Daten ein.
	Gesperrt	der Inhalt des Views wird *eingefroren*, z.B. um es mit anderen Stellen der Aufzeichnung zu vergleichen.
	Verschränkt	die Anzeige wird mit dem Inhalt des *Master* Views synchronisiert.

9.1.1 Folgen (Autoscroll)

Liegt Ihr Augenmerk auf den letzten Ereignissen der von Ihnen untersuchten Datenverbindung, beispielsweise wenn Sie die aktuell gesendeten Datenbytes verfolgen oder den aktuellen Buszustand (Gültigkeit, Richtung) kontrollieren wollen dann aktivieren Sie den *Folgen* Knopf in der Werkzeugleiste.
Das Analysefenster schaltet dadurch in den Autoscroll Mode und verschiebt seinen sichtbaren Datenausschnitt immer so, dass das letzte Ereignis sichtbar ist.
Beachten Sie, dass im Autoscroll Modus keine Synchronisierung mit anderen Anzeigetools stattfindet. Ein aktives Autoscrolling beschränkt sich einzig und allein auf das betreffende Fenster und hat keine Auswirkungen auf andere Analysefenster.

9.1.2 Gesperrt

Für den Fall, dass die geöffneten Fenster unterschiedliche Ausschnitte der Daten repräsentieren sollen, ist eine Synchronisation oder ein Folgen der Ansicht unerwünscht. Schließlich wollen Sie ja gerade verschiedene Ausschnitte betrachten, die eine Aktualisierung des Fensterinhalts zunichte machen würde. In diesem Fall stellen Sie den Anzeigemodus auf gesperrt.

9.1.3 Verschränkt

Sobald Sie diesen Knopf betätigen folgt der Inhalt des Fensters den Cursorbewegungen des aktiven Fensters - synchronisiert sich mit dem Analysefenster,

welches gerade den Eingabefokus hat, d.h. von Ihnen bedient wird (Master Window).
Haben Sie mehrere Analysefenster geöffnet, ist automatisch das Fenster der 'Master', der den Eingabefokus hat, d.h. das aktive Fenster ist. Alle Cursor Bewegungen oder Verschiebungen mit der Maus werden automatisch auf die offenen Fenster übertragen, bei denen der Anzeigemodus auf *verschränkt* eingestellt ist.

9.2 Views (Ansichten)

Views sind eigenständige Programme, die sich in eine aktuell laufende Aufnahme oder gespeicherte Aufzeichnung *einklinken* und die Daten in einer bestimmten Form visualisieren. Die MSB-RS485-PLUS Analyser Software verfolgt dabei das Konzept, für jede Art der Betrachtung der Daten ein darauf optimiertes Anzeigetool zur Verfügung zu stellen. Jedes View stellt die Funktionen zur Verfügung, die seiner Interpretation der Daten entsprechen. Die Bedienung bleibt dadurch einfach und überschaubar, mehrzeilige Werkzeugleisten und überladene Menüs außen vor.
So suchen Sie z.B. in einer Datendarstellung nach Datensequenzen, während Sie bei einer Ereignisdarstellung nach bestimmten Pegelveränderungen Ausschau halten. Jedes View liefert Ihnen den Suchdialog, den Sie auf Grund der Darstellung erwarten.
Datenansichten, die Sie nicht benötigen, schliessen Sie einfach - oder öffnen Sie erst gar nicht. Da es sich um eigenständige Programme handelt, können Sie diese in Größe und Position beliebig plazieren oder auf verschiedenen virtuellen Desktops verteilen.
Das Session Management speichert dabei alle Einstellungen. Views werden automatisch mit den letzten Einstellungen angezeigt und können per einfachem Klick kopiert werden.
Folgende Views sind verfügbar:

9.2.1 Virtueller Ledtester

Den aktuellen Leitungspegel anzeigende LED Leitungstester gehören zum Standardreportaire bei der Kontrolle serieller EIA-232 Verbindungen.
Wir haben sein virtuelles Gegenstück für den Einsatz an EIA-422/485 Verbindungen modifiziert. Damit ist eine schnelle Kontrolle des Buszustandes (inaktiv/aktiv aber auch der Busrichtung, evtl. Handshake Leitungen, und der digitalen Hilfseingänge möglich.

9.2.2 DataView - Datenmonitor

Der Datenmonitor repräsentiert die übertragenen Daten als eine Folge von Datenbytes in verschiedenen Formaten (ASCII, dezimal oder hex). Als besondere Eigenschaft ermöglicht der Datenmonitor die Suche nach bestimmten Mustern mittels regulärer Ausdrücke, die weit über das Suchen nach Worten oder Buchstabenfolgen hinaus geht. Zusätzlich kann gezielt nach Pausen zwischen Sendung/Antwort bzw. generell zwischen beliebigen Daten gesucht werden.

Mit der integrierten Scriptsprache Lua können die angezeigten Daten beliebig verrechnet und eingefärbt werden. Protokolle können visualisiert, Prüfsummen in Echtzeit überprüft und Daten in andere Formate umgewandelt werden.

9.2.3 EventView - Ereignismonitor

Jedes Leitungsänderung ist ein Ereignis und wird protokolliert. Sei es der Wechsel einer Steuerleitung oder das einzelne Bit eines übertragenen Datenbytes. Der Ereignis Monitor erlaubt das einfache Navigieren zwischen allen oder bestimmten Ereignistypen, das Messen der Zeiten zwischen den Ereignissen und das Suchen nach ganz bestimmten Zuständen bzw. Zustandsänderungen, z.B. Wechsel des Buszustandes (tri-state) oder eines Handshake Signals während einer Datensequenz etc.

9.2.4 ProtocolView - Protokollmonitor

Das Protokoll View versetzt Sie in die Lage, die aufgezeichneten Datensequenzen nach bestimmten Regeln darzustellen.
Definieren Sie Ihr eigenes Protokoll, so das jede Datensequenz in einer eigenen Zeile dargestellt wird. Darüber hinaus sind beliebige Bereiche der Sequenz farbig markierbar, um sie noch leichter lesbar zu machen.

9.2.5 SignalView - Signalmonitor

Der MSB-RS485-PLUS Analyser tastet alle Signale mit bis zu 200 Mhz ab. Das Ergebnis liefert der Signalmonitor. Analog zu einem Digitalscope können Sie beliebige Ausschnitte anfahren und in unterschiedlicher Vergrößerung untersuchen.
Durch die Synchronisierung beliebiger Views sehen Sie sofort, welches Signalverhalten jedem einzelnem Datenbyte zu Grunde liegt und damit die reale Welt Ihrer EIA-422/485 Verbindung.

9.2.6 Regionen

Regionen sind beliebige Abschnitte der Aufzeichnung. Sie sind am besten mit Bookmarks (Lesezeichen) zu vergleichen und definieren besondere zeitliche Bereiche in der Aufzeichnung. Regionen können einen Namen zugewiesen bekommen, interessant sind sie aber insbesondere durch die Fähigkeit, *Synchron Anforderungen* aussenden zu können.
Damit genügt ein Klick auf den Beginn oder das Ende einer Region, um diese in den Anzeigeprogrammen einblenden zu lassen und auf diese Weise zwischen verschiedenen Bereichen der Aufzeichnung hin- und herzuschalten.

Fenster kopieren
um Inhalt mit anderem Bereich zu vergleichen

9.3 Views kopieren

Das *Clone* Symbol in der Werkzeugleiste startet eine exakte Kopie des aktuellen Analysefensters mit all seinen Eigenschaften und Position innerhalb der aufgenommenen Daten.
Damit können Sie eine aktuelle Ansicht festhalten, während Sie mit der Kopie (oder dem Original) weiter arbeiten. Sinnvoll vor allem dann, wenn Sie unterschiedliche Datenbereiche vergleichen wollen.

Default Setup
für diesen View Typ

9.4 Default View Einstellungen

Position, Größe und individuelle Einstellung jedes offenen Views (Datenmonitor, Signalmonitor, etc.) wird bei Programmende per Voreinstellung in der Default Projektdatei[1] gespeichert.

[1] Projektdateien enthalten eine komplette Beschreibung der aktuellen Sitzung. Sie sind Thema des nächsten Kapitels.

Unabhängig hiervon sind die Einstellungen, mit denen ein View aus dem Kontrollprogramm heraus gestartet werden. Sie können diese View spezifischen Standardeinstellung aber jederzeit ändern, indem Sie das 'Stecknadel' Symbol in der Werkzeugleiste des gewünschten Views klicken. Ein Beispiel:

Sie möchten den Signalmonitor immer mit dem dunklen Thema (schwarzer Signalhintergrund) und mit nur 4 angezeigten Signalen starten. Sagen wir nur die Signal e RxD, TxD, RTS and CTS bzw. CH1...CH4 bei einem MSB-RS485 (PLUS) Analyser.

In diesem Fall konfigurieren Sie die Einstellungen des Signalmonitors nach Ihren speziellen Vorstellungen und klicken danach das 'Stecknadel' Icon um diese Einstellungen als Vorgabe fest zu 'pinnen'. Das Kontrollprogramm bestätigt die neuen Einstellungen (die sich nur auf den Signalmonitor auswirken) mit einer kurzen Meldung.
Sobald Sie einen neuen Signalmonitor aus dem Kontrollprogramm heraus starten werden automatisch diese neuen Einstellungen verwendet. Diese werden zudem beim Schließen des Kontrollprogrammes automatisch im Default Projekt gespeichert und sind auch in der nächsten Sitzung weiterhin gültig.

10 Session Management

Eine Programm Sitzung beinhaltet oftmals eine Vielzahl geöffneter Fenster unterschiedlichster Darstellung. Das Session Management sorgt dafür, dass Sie bei erneutem Programmstart alles so wiederfinden, wie Sie es verlassen haben.

Der Session Manager der MSB-RS485-PLUS Software gewährleistet, daß alle für eine Programmsitzung relevanten Einstellungen wie Aufzeichnungsparameter, Fenstereigenschaften der offenen Views (Position, Größe), Inhalt (d.h. Farben, Schriftarten, Darstellungsformate) beim Schliessen des Kontrollprogrammes gespeichert und beim nächsten Start wieder hergestellt werden.
Die Speicherung der aktuellen Programmeinstellungen erfolgt völlig transparent und ohne das Sie etwas dazu tun müssen.
Sie haben allerdings auch die Möglichkeit, die aktuelle Sitzung inklusive der aufgezeichneten Daten als Projekt abzuspeichern. In diesem Fall können Sie zu einem späteren Zeitpunkt mit der Untersuchung der aufgenommenen Daten fortfahren, indem Sie einfach die Projektdatei erneut öffnen.

10.1 Projekte

Projekte dienen zur Speicherung Ihrer aktuellen Arbeit (Analyse) mit der MSB-RS485-PLUS Software, d.h. inklusive der aufgezeichneten Daten. Ein Projekt besteht deshalb immer aus zwei Dateien:

1. **Projekt Datei:** Beschreibt den Zustand und die Eigenschaften aller offenen Views. Projektdateien tragen immer die Endung `*.msbprj`.
2. **Record Datei:** Enthält die eigentlichen Daten und alle für die Datenaufzeichnung relevanten Informationen wie: Datenrate, Protokoll, definierte Regionen, aufgezeichnete Ereignisse und Zeitpunkt der Aufzeichnung. Record Dateien werden immer mit der Endung `*.msblog` abgelegt.

Warum diese Aufteilung in zwei Dateien?

Gespeicherte Sitzungen (Projekte) entsprechen dem Wunsch des Anwenders, das Programm individuell für seine Bedürfnisse konfigurieren zu wollen. Diese sind oftmals unabhängig von bestimmten Aufzeichnungen. Sei es, dass er Platzierung und Darstellung der einzelnen Fenster seiner Bildschirmauflösung anpasst, oder einfach nur seine Farb/Schriftarten umsetzen möchte.
Aufzeichnungsdateien wiederum enthalten Information, die unabhängig von

den Sitzungseinstellungen sind. D.h. genaue Angaben zum Protokoll, Zeitmarken, Regionen, verwendete Signalnamen sowie welche Ereignisarten/typen aufgezeichnet wurden. Darüber hinaus sollten Aufzeichnungen von unterschiedlichen Personen (mit unterschiedlichen Vorstellungen, was die Konfigurierung der Software anbelangt) analysiert werden können.

Durch die Unterteilung in Projekt und Record Dateien bieten sich zudem eine Reihe von Vorteilen:

- Die Speicherung der Aufzeichnung erfolgt unabhängig von der aktuellen Sitzung.
- Eine Aufzeichnung kann in eine bestehende Sitzung geladen werden ohne diese zu verändern.
- Andere Anwender können eine Aufzeichnung mit Ihrer individuellen Konfiguration untersuchen
- Projektdateien können gezielt für bestimmte Analysen/Projekte definiert und weiter gegeben werden.

Durch die klare Trennung zwischen Projekt- und Aufzeichnungsdateien können Sie jederzeit eine Aufzeichnung mit Ihrer individuellen Programmeinstellung untersuchen, oder aber eine bestimmte Programm Konfiguration (Projekt) für die Analyse einer Aufzeichnung verwenden.
Projekt- und Recorddateien haben jeweils ihr eigenes Icon um sie leichter zu unterscheiden. Sie werden bei der Installation mit der MSB-RS485-PLUS Software verknüpft und lassen sich dadurch einfach per Doppelklick (Windows) öffnen.

Projekt Dateien
speichern alle Sitzungs relevanten Eigenschaften und besitzen die Endung *.msbprj

Record Dateien
enthalten die eigentlichen Daten sowie alle zur Übertragung gehörenden Parameter. Sie haben die Endung *.msblog

10.2 Projekte speichern/laden

Speichern und Laden von Projekten erfolgt immer aus dem Kontrollprogramm heraus. Die Aufteilung in eine Projekt- und eine Record Datei geschieht dabei automatisch, wobei beide Dateien bis auf die Endung den gleichen, von Ihnen vorgegebenen Namen besitzen.
Entsprechend wird beim Öffnen eines Projektes eine (falls existierende) Aufzeichnungsdatei mit dem gleichen Namen geladen.
Gleiches gilt, wenn Sie die MSB-RS485-PLUS Software per Doppelklick auf eine Projektdatei `*.msbprj` aus dem Windows Dateiexplorer starten. Jedes Öffnen einer Projektdatei lädt automatisch die zugehörige Aufzeichnungsdatei. Beachten Sie, dass bestimmte Einstellungen, z.B. die Baudrate, sowohl als Vorgabe (in einer Projektdatei) wie auch als zwingender Bestandteil einer Aufzeichnung gespeichert werden.

Sobald eine Aufzeichnung von der Software geladen wird, werden deshalb die dazu nötigen Informationen aus der Aufzeichnungsdatei übernommen und die Sitzungsparameter in diesem Fall überschrieben. Dies gilt neben den Protokoll Einstellungen (Baudrate, Parity, Stopbit) auch für die Definition der Signalnamen sowie Einstellung der aufzuzeichnenden Ereignisse. Alle diese Einstellungen sind untrennbar mit den aufgezeichneten Daten verknüpft.

Reine Projektdatei ohne Datenaufzeichnung erzeugen

Um eine aktuelle Sitzung als Konfiguration für spätere Untersuchungen zu sichern, speichern Sie diese als Projekt ohne Daten aufzuzeichnen oder löschen die Record Datei anschließend.

10.3 Automatisches Speichern einer Sitzung

Dieser Vorgang findet völlig transparent im Hintergrund statt, wenn Sie die aktuelle Sitzung durch Schließen des Kontrollprogrammes beenden. Die MSB-RS485-PLUS Software speichert dazu alle nötigen Einstellungen in einer Konfigurationsdatei mit folgender Namensgebung:

```
ANALYZERTYPE-SERIALNUMBER.msbprj
```

Typical names are:

```
MSB-RS232-MSB00000.msbprj
MSB-RS232-PLUS-MSB05001.msbprj
MSB-RS485-MSB00000.msbprj
MSB-RS485-MSB04080.msbprj
MSB-RS485-PLUS-MSB00000.msbprj
MSB-RS485-PLUS-MSB05002.msbprj
```

Sollten Sie keinen Analyser angeschlossen haben, werden die Einstellungen in der Datei `ANALYZERTYPE-MSB00000.msbprj` gesichert.
Das Verzeichnis, unter welchem die Default Projektdateien abgelegt werden, hängt von Ihrem OS ab. Windows Anwender finden sie unter:

C:\User\USER\AppData\Roaming\IFTOOLS\SerialAnalyzer\VERSION\Defaults

mit Ausnahme von Windows XP. Dort werden sie gespeichert unter:

C:\Documents and Settings\USER\Application Data\IFTOOLS\SerialAnalyzer\VERSION\Defaults

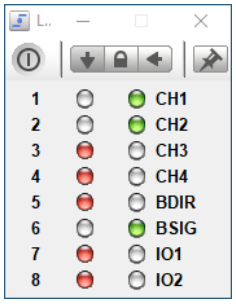
Linux User finden Sie wie üblich in Ihrem Home Verzeichnis unter:

/home/USER/.IFTOOLS/SerialAnalyzer/VERSION/Defaults

11 Ein virtueller Ledtester

LED Leitungstester gehören zum Standard Repertoire bei der Kontrolle serieller RS232 Verbindungen. Das für EIA-422/485 Verbindungen adaptierte virtuelle Gegenstück erlaubt einen schnellen Überblick über den Bus-Status, Bus-Datenrichtung und Bus-Aktivität.

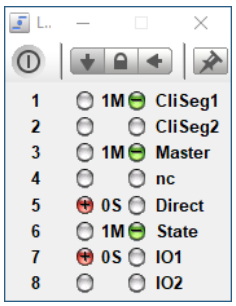
Der virtuelle Ledtester ist einem einfachen, handelsüblichen seriellen Leitungstester nachempfunden und zeigt den Status aller Differenzsignal-Eingänge CH1 bis CH4 sowie beider digitaler Hilfseingänge an. Zusätzlich wird das Bus-Signal sowie die Bus-Richtung zwischen CH1 und CH2 (Segement Analyse) visualisiert.
Der Ledtester (oder Leitungsmonitor) besitzt zur besseren Übersichtlichkeit für beide aktive Leitungszustände eine separate Diodenreihe aus jeweils roten und grünen LEDs.
Die roten LEDs auf der rechten Seite signalisieren einen positiven Leitungspegel, die grünen auf der linken Seite einen negativen Pegel.



Virtueller Ledtester
mit Standard Namen...

In Bereich ±0.7V sind beide LEDs aus, dies entspricht dem inaktiven Bus/Leitungszustand bei EIA-485 Verbindungen. Der Schwellwert der EIA-485 Empfänger liegt zwar bei ±200mV. Durch die höheren Pegel des MSB-RS485-PLUS Analysers werden allerdings auch durch Pullups bzw. Pulldowns voreingestellte Bus-Ruhepegel noch zuverlässig als ungültige Buszustände erkannt, wenn sie den ±200mV Bereich überschreiten.

Die Anzeige erfolgt unabhängig davon ob Sie eine Aufnahme gestartet haben oder nicht. Sie können den Leitungsstatus Monitor allerdings auch dazu verwenden, den Status zu einem bestimmten Zeitpunkt innerhalb der Aufzeichnung anzuzeigen.
Wählen Sie dazu den 'Sync' Modus in der Werkzeugleiste. Damit wird der Leitungsstatus mit dem aktiven Anzeigefenster, z.B. einem Datenmonitor synchronisiert. Oder Sie 'frieren' den aktuellen Status der Leitungen ein, indem Sie die aktuelle Ansicht des Ledtesters verriegeln.



Eingeblendete Pegel
und individuelle Namen...

Aktive Pegel einer EIA-422/485 Verbindung werden wahlweise mit logisch 0/1, als Space/Mark oder aber als physikalische positive oder negative Spannung beschrieben. Dies ist oftmals eher verwirrend statt hilfreich. Um Ihnen das Leben ein wenig einfacher zu machen blendet der virtuelle Ledtester deshalb

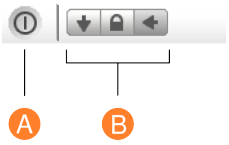
zusätzliche Informationen zu Zustand der Leitungen ein. Bewegen Sie einfach den Mauszeiger über den Ledtester um diese Informationen sichtbar zu machen.

Pegel	Bedeutung
1M	logisch 1, Mark, entspricht einer negativen Spannung von –0.7V...-7V am Differenzsignal-Eingang (grüne LED mit Minus-Symbol)
0S	logisch 0, Space, entspricht einer positiven Spannung von +0.7V...+12V am Differenzsignal-Eingang (rote LED mit Plus-Symbol)

Wie alle anderen Views aktualisiert auch der virtuelle Ledtester die Signalnamen, sobald diese im Kontrollprogramm geändert wurden. Das gilt im übrigen auch, wenn Sie die Bus Anschlussart wechseln.

11.1 Werkzeugleiste

Die Werkzeugleiste dient zum schnellen Zugriff der am meisten benötigten Funktionen. Einige davon sind bei allen Views identisch, andere spezifisch für den virtuellen LedTester.



A **Ende:** Speichert alle Einstellungen (Fensterposition) und schließt das Fenster

B **Anzeigemodus:** Je nach Anzeigemodus zeigt der Ledtester immer den aktuellen Zustand der Leitungen, ist verriegelt oder aktualisiert den Inhalt mit anderen Fenstern. Die Voreinstellung ist der altuelle Leitungszustand.

12 Der Datenmonitor

Sie suchen nach bestimmten Datenfolgen? Nach Übertragungspausen bestimmter Länge? Der Datenmonitor zeigt Ihnen die Daten in ihrem zeitlichen, realen Verlauf. Die Darstellung erfolgt wahlweise in dezimal, hex oder ASCII und enthält zusätzliche Parity, Framing oder Break Informationen. Reguläre Ausdrücke erlauben die Suche nach beliebigen Datenmustern und vieles mehr...

Der Datenmonitor zeigt alle übertragenen und vom Analyser aufgezeichneten Datenbytes in der Reihenfolge ihres Auftretens. Änderungen der Steuerleitungen werden ausgeblendet, so daß nur die reinen Nutzdaten, die über die serielle Verbindung transportiert wurden, dargestellt werden.
Die Daten können dabei getrennt nach Datenkanal A/B (die Zuordnung Eingangssignal/Datenkanal ist abhängig von der Anschluss-Einstellung) oder zusammen angezeigt werden (siehe Signalauswahl). Letzteres ist sinnvoll, wenn die Reaktion auf gesendete Daten untersucht werden soll.
Wollen Sie beide Datenkänale A und B gleichzeitig ansehen, allerdings ohne diese zu mischen, starten Sie einfach zwei Daten Monitore.
Sie können auch verschiedene Ausschnitte des aufgezeichneten Datenstroms betrachten. Genauer gesagt: Sie können soviele Daten Monitore öffnen wie Sie wollen. Die Resourcen Ihres Rechners sind die einzige Einschränkung, der Sie unterliegen.

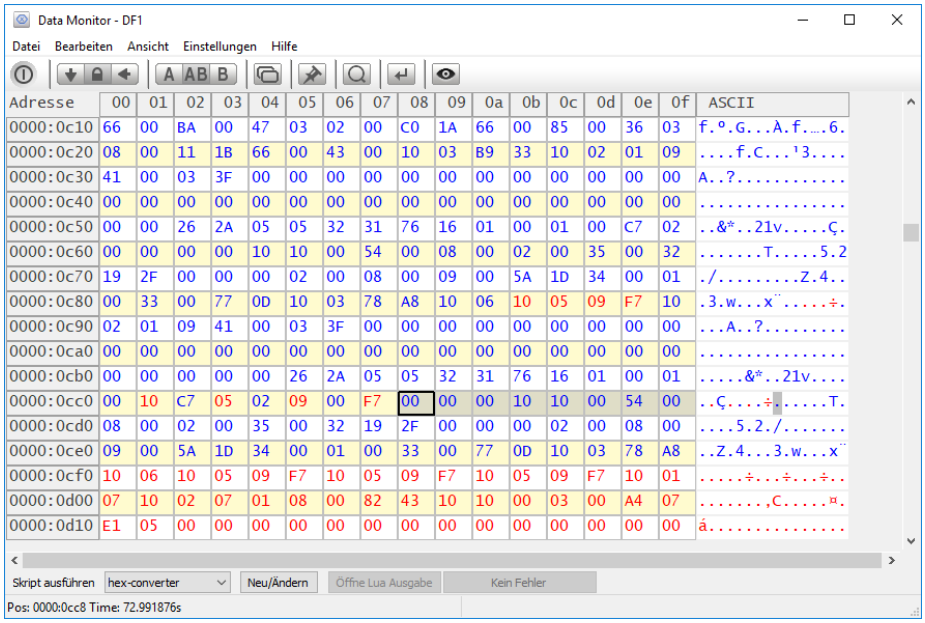
12.1 User Interface

Der Datenmonitor stellt die übertragenen Datenbytes in Form eines Hexeditors dar. Die Datenansicht ist auf 8 Zeichen oder Bytes[1] pro Zeile voreingestellt, die in hexadezimaler Schreibweise sowie ihrem ASCII Gegenstück angezeigt werden. Sie können dies allerdings jederzeit ändern. Jeder Datenzeile ist eine laufende Adresse oder Position vorgestellt, die die genaue Position innerhalb des aufgezeichneten Datenstroms wieder gibt. Nicht druckbare Zeichen wie z.B. das Zeilenvorschub Zeichen werden in der ASCII Anzeige als Punkt dargestellt.
Mit den Pfeil- und Bildtast können Sie den Cursor beliebig über die Daten steuern, wobei die Statuszeile zusätzliche Informationen zum genauen Zeitpunkt

[1] Genau genommen 9 Bit Werte, da der MSB-RS485-PLUS Analyser Übertragungen mit 9 Bit Datenlänge unterstützt.

des Auftretens, der Position und der Anzahl im Vergleich zur Gesamtzahl anzeigt.

Im Falle eines Übertragungsfehlers (Frame oder Parity) blendet der Datenmonitor automatisch den Fehler in den zugehörigen Datenbytes ein. Dies gilt auch für ein Break, welches im Datenstrom sonst leicht als Null Byte fehlinterpretiert werden könnte, siehe auch folgender Abschnitt 12.1.1.

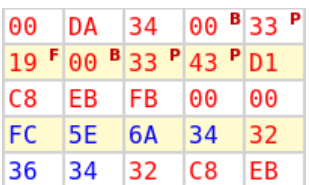


Integrierte Skriptsprache Lua

Mit der integrierten Lua SkriptInterpreter können Sie die angezeigten Daten beliebig verrechnen, in andere Datenformate umwandeln und das Ergebnis in dem Lua Ausgabe Fenster (siehe Abschnitt 12.6 ausgeben lassen.

Mehr noch - Sie können die Daten Skript-gesteuert farblich markieren und damit Protokolle oder besonders interessante Abschnitte hervorheben. Und wenn Sie zusätzlich eine Prüfsumme validieren wollen - auch das ist kein Problem. Ein kleines Lua Skript erledigt dies für Sie und hebt korrekte und/oder falsche Prüfsummenwerte automatisch hervor.
Alle Lua relevanten Funktionen sind direkt unter der Datenanzeige gruppiert und jederzeit leicht aufzurufen wenn Sie diese benötigen. Mehr Informationen dazu finden Sie im Abschnitt 12.6.

Die Statuszeile liefert Ihnen zusätzliche Informationen über die aktuelle Cursorposition. Der linke Feld Pos: zeigt die aktuelle Position oder Adresse des Cursors innerhalb der ausgewählten Daten, das rechte Feld Time: den genauen Zeitpunkt, wann das Datenbyte aufgetreten ist.

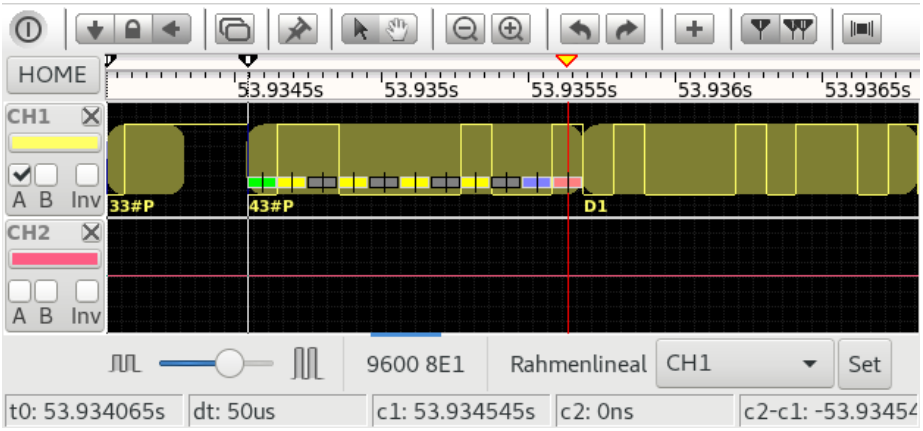


Anzeige Datenfehler
Frame, Parity, Break

12.1.1 Anzeige von Daten Fehler

In einer asynchronen seriellen Datenverbindung werden Datenbytes als Abfolge von einzelnen Bits übertragen. Jedes Byte wird mit einem Startbit einge-

leitet, gefolgt von 5 bis 9 Datenbits, einem optionalen Paritätsbit und einem endenden Stopbit. Die ganze Bit Sequenz wird als Datenrahmen oder Frame bezeichnet.
Das Startbit (oder besser die fallende Flanke des Startbits) ist insofern wichtig, weil es nicht nur den Beginn des Datenrahmens kennzeichnet sondern zusätzlich die Triggerung sowie Resynchronierung der Signalabtastung seitens des Empfängers einleitet.
Falls der Analyser einen falschen Datenrahmen erkennt, z.B. wenn die Anzahl der gesendeten Datenbits nicht mit der Einstellung der Verbindung in der Aufzeichnung übereinstimmt, werden diese Datenbytes mit einem 'F' (Frame Fehler) gekennzeichnet.
Ein Parity Fehler 'P' wird angezeigt, wenn das empfangene Parity Bit nicht den eingestellten Parametern entspricht. Dies passiert z.B. wenn Sie eine Aufzeichnung mit 38400 8E1 (E = even/gerade Parität) konfigurieren, die Übertragung aber 38400 8O1 (O = odd/ungerade Parität) verwendet.
Beide Fehler (frame oder parity) können auch durch eine falsche oder schwankende Baudrate verursacht werden. Sie sind immer ein Zeichen, das mit der Aufzeichnung oder Übertragung etwas nicht in Ordnung ist.
Eingeblendete 'B' indizieren einen Break Zustand. Ein Break ist definiert als eine spezifische Zeit (deutlich länger als die Übertragungszeit eines einzelnen Zeichens) in der das Signal einen Low-Pegel aufweist. Das zugehörige Datenbyte wird deshalb als Null Wert angezeigt darf aber nicht mit einem normalen 0 Datenbyte verwechselt werden. Der Analyser unterscheidet zwischen normalen, binären Null Bytes und einem Break. Breaks werden zuweilen als Telegramm Trennzeichen, zur Synchronisation oder Reset eines Busteilnehmers verwendet.
Zur genaueren Analyse öffnen Sie einfach einen Signalmonitor und stellen dessen Synchronisierung auf 'anderen Views folgen', siehe auch nächster Abschnitt 12.1.2. Sobald Sie dann ein fehlerhaftes Datenbyte im Datenmonitor anklicken, wird dieses als Tri-State Signal im Signalmonitor angezeigt und Sie können den Signalverlauf dort genauer untersuchen.



Signal im Sync Mode
zeigt angeklickte Daten

Im obigen Bild Beispiel zeigt der Signalmonitor den Grund für den Paritätsfehler. Die Aufnahme wurde mit 8E1 gemacht (gerade Parität), die Daten aber mit 8O1 (ungerade Parität) übertragen. Der Analyser markiert deshalb dieses

Datenbyte weil er eine gerade Anzahl von Bits im Datenrahmen (inklusive Paritätsbit) erwartet und das Paritätsbit deshalb statt einem Low Pegel einen High Pegel haben muss (im Rahmenlineal blau dargestellt).
Manche Aufzeichnungen weisen nur sehr wenige Datenfehler auf. Wenn Sie sicher sein wollen, das Ihre Aufzeichnung in keine Fehler enthält, können Sie gezielt nach solchen suchen. Wie das geht wird im Abschnitt 12.5.3 detailliert erklärt.

12.1.2 Synchronisierung der Ansicht

Scroll Sperren Update
durch andere Views

Allen MultiView Programm ist gemeinsam, daß Sie die Ansicht synchronisieren, sperren oder per Autoscroll die jeweils zuletzt aufgenommenen Daten anzeigen lassen können. Dies gilt auch für den Datenmonitor. Ist der Datenmonitor das aktive Fenster, d.h. das Fenster, welches Ihre Eingaben entgegen nimmt, dann wird mit jeder Bewegung oder Plazierung des Cursors ein entsprechendes Sync.-Signal an alle anderen geöffneten Analysefenster gesendet.

Dies schließt auch die Bewegung des Cursors als Ergebnis einer Suche oder Positionierung mit ein. Auf diese Weise können Sie sich z.B. im Signalmonitor den Signalverlauf bei Auftreten bestimmter Datensequenzen betrachten.
Links klicken Sie einfach das gewünschte Datenbyte, um dessen Ansicht bei anderen Views einzublenden.

Entsprechend reagiert der Datenmonitor auf eine Synchronisierung durch andere Views und blendet den zugehörigen Datenausschnitt ein, wobei der Cursor auf das dem Ausgangsereignis folgende Datenbyte gesetzt wird.
Wie sich der Datenmonitor beim Empfang eines Sync.-Signals (durch ein anderes, die Eingabe besitzendes Analysefenster) verhält, bestimmen die bei jedem Analysefenster identischen Sync. Knöpfe in der Werkzeugleiste.
Per default ist die Ansicht des Datenmonitors verriegelt, d.h. es reagiert auf keine Änderungen durch andere Analysetools. Beachten Sie, daß der Datenmonitor unabhängig von dieser Einstellung bei Cursorbewegungen immer ein Sync.-Signal generiert! Fenster, die nicht synchronisiert werden sollen, müssen explizit verriegelt werden.

12.1.3 Auswahl des Datenkanals

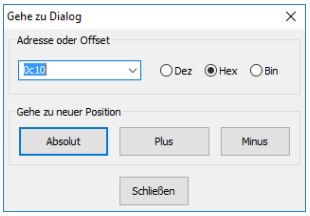
Signalauswahl
getrennte oder
gemeinsame Darstellung

Im Datenmonitor können wahlweise beide Datenkanäle A und B oder jeweils ein einzelner dargestellt werden. Je nach Anschlussart (Abgriff oder Segmentanalyse macht die Anzeige von beiden Kanälen mehr oder weniger Sinn. Sie können jederzeit zwischen den einzelnen Richtungen wechseln, indem Sie die Signalauswahl in der Werkzeugleiste verwenden.
Die Signalauswahl entscheidet gleichzeitig darüber, welche Daten abgespeichert werden können. Wenn Sie beide Datenkanäle A+B ausgewählt haben, werden generell auch beide abgespeichert, ansonsten nur die Daten des ausgewählten Datenkanals. Auf diese Weise ist es möglich, gezielt die Daten abhängig von einer Quelle in einer Datei abzulegen.

12.1.4 Fensterinhalt positionieren

Neben der Navigation mit dem Cursor oder der linken Scrolleiste bietet der Datenmonitor auch ein absolutes Positionieren sowie eine Verschiebung der momentanen Ansicht relativ zur aktuellen Cursor Position.

Klicken Sie das  Symbol in der Werkzeugleiste oder im Menü Ansicht→Gehe zu Dialog um den Dialog zur absoluten/relativen Positionierung zu öffnen. Alternativ können Sie auch einfach Strg+G drücken.
Geben Sie einfach die absolute Adresse oder den gewünschten Offset zur aktuellen Position ein und klicken Sie auf einen der folgenden Knöpfe.



Absolutes/relatives Positionieren mit Strg+G

- **Absolut:** Bewegt den Datenausschnitt zur angegebenen Adresse, Kurzbefehl Alt+A.
- **Plus:** Addiert den angegebenen Wert zur aktuellen Position und verschiebt den Datenausschnitt Richtung Datenende, Kurzbefehl Alt+P
- **Minus:** Subtrahiert den angegeben Wert von der aktuellen Position und verschiebt das Datenfenster entsprechend Richtung Datenanfang, Kurzbefehl Alt+N.

Die Eingabe kann wahlweise in dezimaler, hexadezimaler oder binärer Form erfolgen. Klicken Sie einfach auf das gewünschte Zahlenformat. Wie die meisten anderen Dialoge können Sie auch den Gehe zu Dialog solange geöffnet lassen, solange Sie ihn benötigen.

12.2 Bereich auswählen

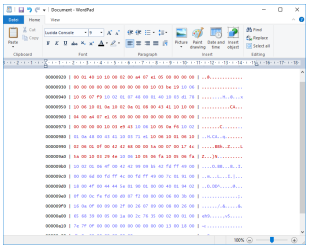
Wenn Sie innerhalb des Datenbereichs, d.h. auf einem der angezeigten Daten, die rechte Maustaste drücken, öffnet sich ein Kontextmenü, mit welchem ein beliebiger Ausschnitt der aufgezeichneten Daten ausgewählt oder selektiert werden kann. Dabei ist es egal, ob Sie zunächst den Anfang des gewünschten Bereiches markieren oder das Ende.

Oder Sie Klicken auf das erste Byte des zu kopierenden Bereichs bei gleichzeitig gedrückter Strg Taste. Anschließend verschieben Sie den sichtbaren Ausschnitt bis zum Ende des zu Bereichs und markieren das Ende der Auswahl mit einem Linksklick bei gleichzeitig gedrückter Umschalt Taste. Der Bereich wird daraufhin hellblau hinterlegt.
Alle Daten markiert die Tastenkombination Strg+A.

Die damit verbundene Datenauswahl kann separat gespeichert oder mit der Taste F4 einer Region zugewiesen werden. Durch das gezielte Abspeichern bestimmter Datensequenzen in einer Datei können Sie diese Daten z.B. auf Übertragungsfehler untersuchen, indem Sie sie mit den Originaldaten vergleichen.
Mit der Export oder Copy And Paste Funktion können Sie einen beliebigen markierten Ausschnitt der Daten in anderen Anwendungen weiter verarbeiten.

12.2.1 Copy and Paste

Copy And Paste kopiert den ausgewählten Bereich in die Zwischenablage und fügt ihn in einer anderen Anwendung wieder ein. Handelt es sich dabei um eine Textverarbeitung mit RTF Unterstützung wie z.B. WordPad®, Microsoft Word®oder OpenOffice Writer®erfolgt die Darstellung mit farbiger Anzeige der Datenrichtung[2].
Da reine Texteditioren (wie NotePad) keinen formatierten Text verarbeiten und



Copy and Paste in Textverarbeitung

[2]Farbiges Copy and Paste unterstützt z.Z. nur die Analyser Software für Microsoft Windows.

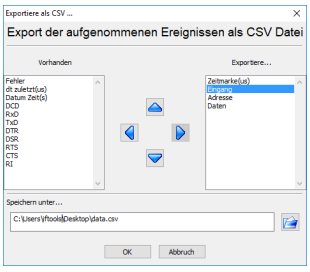
deshalb keine Farbinformationen besitzen, muß die Datenrichtung der einzelnen Daten anders visualisiert werden. Den Daten des ersten Datenkanals (A) wird dazu ein einzelner Punkt, den Daten des zweiten Datenkanals (B) ein Doppelpunkt vorangestellt. Die Textdarstellung der Daten erfolgt generell hexadezimal, um Probleme mit unterscheidlichen Zeichensätzen zu vermeiden.

```
00000000 | :73 :65 :6e :64 :20 :64 :61 :74 :61 :20  | send data
00000010 | :77 :69 :74 :68 :6f :75 :74 :20 :65 :72  | without er
00000020 | :72 :6f :72 :0a .73 .6f .6d .65 .20 .72  | ror.some r
00000030 | .65 .73 .70 .6f .6e .73 .65 .0a :66 :72  | esponse.fr
00000040 | :61 :6d :65 :21 :0a .66 .72 .61 .6d .65  | ame!.frame
00000050 | .20 .72 .65 .73 .70 .6f .6e .73 .65 .0a  |  response.
00000060 | :00 .00 :70 :61 :72 :69 :74 :79 :0a .70  | ..parity.p
00000070 | .61 .72 .69 .74 .79 .20 .61 .6e .73 .77  | arity answ
00000080 | .65 .72 .0a .00 .6e .6f .20 .6f .6f .70  | er..no oop
00000090 | .73 .00 :31 :32 :33 :00                  | s.123.
```

12.2.2 Datenauschnitt speichern

Der Datenmonitor bietet Ihnen die Möglichkeit beliebige Bereiche als Binärdaten in einer Datei zu speichern. Diese Datei enthalt damit eine exakte Folge der Daten, die Sie markiert haben. Sie werden dies zu schätzen wissen, wenn Sie die vom Analyser aufgezeichneten Datensequenzen mit anderen, als Datei vorliegenden Daten vergleichen wollen.
Beispielsweise, wenn Sie das Ergebnis der von einem Busteilnehmer gesendeten Daten oder die gesendeten Originaldaten kennen, und einfach wissen möchten, ob diese auch so übertragen wurden.
Selektieren Sie dazu einfach den gewünschten Bereich oder mit Strg+A alle Daten und klicken im Menü Datei den Eintrag Speichere Auswahl unter....
Generell gilt: Sie müssen zuerst einen Bereich selektieren (auswählen), bevor Sie ihn abspeichern können. Wenn Sie beide Datenkanäle zur Anzeige ausgewählt haben (A+B) werden auch beide gespeichert. Für einen Vergleich bietet sich deshalb an, nur einen Kanal auszuwählen.

12.2.3 Exportieren der Daten



Datenexport
als Komma CSV Datei

Um einen Ausschnitt oder alle aufgezeichneten Daten gesondert zu analysieren, z.B. mit einem Tabellenkalkulations Programm, können Sie diese als sogenannte CSV (Comma Separated Values) Datei exportieren. Tabellenkalkulations Programme bieten umfangreiche statistische Möglichkeiten um die Daten zu bearbeiten. Sei es eine Häufigkeitsverteilung der einzelnen Daten, minimale und maximale Zeiten zwischen einzelnen Bytes etc. Die hier besprochenen Exportmöglichkeiten beziehen sich vor allem auf die Daten. Wenn Sie an einer Analyse aller anderen Ereignisse interessiert sind, werfen Sie einen Blick auf das Kapitel Auswahl exportieren im Ereignismonitor.
Selektieren Sie den gewünschten Bereich und Klicken Sie anschliessend im Menü Datei den Eintrag ’ Exportiere als CVS’.
In dem sich öffnenden Exportdialog können Sie aus der linken Liste der verfügbaren Werte einen beliebigen Wert auswählen indem Sie ihn einfach anklicken und anschliessend mit dem ’Pfeil nach rechts’ Knopf in die Liste der zu exportierenden Werte verschieben. Wiederholen Sie dies einfach mit allen von Ihnen benötigten Werten.

Um die Reihenfolge der exportierenden Werte zu ändern klicken Sie auf den zu verschiebenden Wert (in der Liste der exportierenden Werte) und bewegen ihn, mit dem 'Pfeil nach oben' bzw. 'Pfeil nach unten' Knopf in die gewünschte Richtung.
Entsprechend können Sie einen in der Exportliste markierten Eintrag mit dem 'Pfeil nach links' Knopf wieder entfernen, d.h. er wird wieder in die Auswahlliste zurück gelegt.
Abschliessend geben Sie einen Dateinamen ein, unter welchem die Exportdatei gespeichert werden soll und klicken den 'OK' Knopf um den Export zu starten.
Beim Export der Daten wird die aktuelle Ansicht des Datenmonitors berücksichtigt. D.h. die Daten werden in hexadezimaler (mit voran gestelltem '0x', dezimaler Schreibweise oder als ASCII Zeichen in eingeschlossenen Hochkommas in die Datei geschrieben. Entsprechendes gilt für das Format der Adresse. (Bei der Adresse handelt es sich um die Position des Datenbytes im Datenstrom). Ein Beispiel mit hexadezimalem Adress- und Datenformat:

```
"Zeitmarke(us)","Adresse","Eingang","Daten"
3547,0x000050,A,0x20
3547,0x000051,B,0x20
3634,0x000052,A,0x21
3634,0x000053,B,0x21
3720,0x000054,A,0x22
3720,0x000055,B,0x22
...
```

Der gleiche Ausschnitt, diesmal mit dezimaler Adressdarstellung und Anzeige der Datenbytes in ASCII.

```
"Zeitmarke(us)","Adresse","Eingang","Daten"
3547,00000080,A,' '
3547,00000081,B,' '
3634,00000082,A,'!'
3634,00000083,B,'!'
3720,00000084,A,'"'
3720,00000085,B,'"'
...
```

Beachten Sie, dass die Zeitmarken in Mikrosekunden (us) protokolliert werden. Durch die Verwendung eines Loopback Steckers treten in unserem Beispiel jeweils immer zwei Ereignisse an beiden Datenkanälen A und B Zeit gleich auf.

12.3 Programm Einstellungen

Die Darstellung der Daten läßt sich in weiten Bereichen an Ihre eigenen Anforderungen anpassen. Öffnen Sie dazu den Einstelldialog im Menü:
`Einstellungen→Datenmonitor einrichten`.
Jedes View bietet die für es relevanten Einstellmöglichkeiten. Im Falle des Datenmonitors sind es:

- **Anzeige:** Anzahl und Anzeigeformat der Spalten und Daten.
- **Farben:** Farbregeln zur Darstellung bzw. Markierung bestimmter Daten.

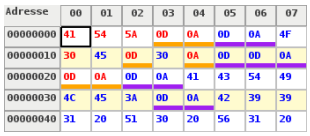
- **Schrift:** Schriftart und Schriftgröße.

Alle Einstellungen können zunächst per *Anwenden* Knopf ausprobiert werden, bevor Sie sie mit *OK* endgültig übernehmen.

12.3.1 Spaltenanzahl und Datenformat

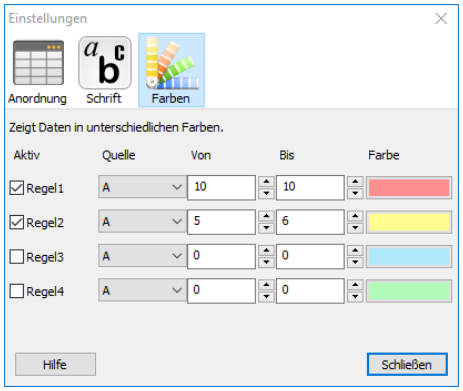
Sie können sowohl die Anzahl der Spalten als auch die Darstellungsart (hexadezimal, dezimal und ASCII) getrennt nach Datenbytes und Adressanzeige individuell vorgeben. Die Anzahl der Zeilen verändern Sie durch vergrößern bzw. verkleinern des Programmfensters.
Zusätzlich können Sie für die ersten 32 Zeichen des ASCII Zeichensatzes (Kontrollzeichen) die allgemein definierten Namen einblenden. Z.B. statt dem Hexwert 0A des Linefeed ein 'LF'.
Nicht druckbare Zeichen können in der ASCII Spalte wahlweise als Punkt oder als Originalzeichen entsprechend dem gewählten Zeichensatz angezeigt werden.

12.3.2 Daten einfärben



Daten färben
mit Farbregeln...

Der Datenmonitor erlaubt Ihnen beliebige Daten abhängig von der Datenquelle einzufärben. Dies macht vor allem dann Sinn, wenn Sie bestimmte Datenbytes oder Sequenzen besonders hervorheben möchten. Beispielsweise EOS Zeichen wie Carridge Return und/oder Linefeed. Oder Zeichen mit gesetztem 8 oder 9ten Bit wie sie oft bei Bus-Protokollen zur Kennzeichnung von Adress-Kommandos verwendet werden.
Im Einstellmenü Farben können Sie dazu bis zu 4 sogenannten *Farbregeln* definieren, die auf die Anzeige der übertragenen Daten angewendet werden.



Jede einzelne Regel beinhaltet die Datenquelle oder Datenkanal (A bzw. B), einen Bereich für den Datenwert und die Farbe, mit der diese Datenbytes eingefärbt werden sollen. Sie können jede Regel einzeln ein- oder ausschalten, indem Sie diese explizit aktivieren bzw. deaktivieren.
Die Eingabe der Datenwerte von/bis erfolgt dezimal, wobei der Zahlenbereich 0...511 umfaßt. Werte über 255 machen allerdings nur Sinn, wenn Sie eine Übertragung mit 9 Datenbits analysieren.
Die Regeln werden in Ihrer Reihenfolge von 1 bis 4 (oder von oben nach unten) abgearbeitet. Regeln können sich überschneiden. In diesem Fall kommt die zuletzt gemachte Regel zur Anwendung. Damit ist es möglich einzelne Regeln in Teilen zu überschreiben, z.B. einzelne Bytes einer zuvor definierten Regel nochmals gezielt anders zu färben.
Alle von Ihnen eingegebenen Regeln werden automatisch gespeichert und sind gleichzeitig für alle später geöffneten Datenmonitore vorhanden.

Farbregeln sind nur für einfache Anwendungsfälle gedacht. Wenn Sie die Daten nach komplexeren Regeln markieren wollen verwenden Sie den integrierten Lua Skripteditor, siehe Abschnitt 12.6.

12.3.3 Schriftart ändern

Schriftart
wählen im Einstelldialog

Neben der Spaltenanzahl und Darstellung der Datenbytes können Sie auch das Schriftbild ändern. Z.B. um statt der voreingestellten Proportionalschrift

eine Schriftart mit gleicher Buchstabenbreite zu wählen, oder um einfach die Schriftgröße bzw. Schriftstill anzupassen.
Klicken Sie dazu `Einstellungen→Datenmonitor einrichten` um den Einstelldialog zu öffnen.
Die eingestellte Schrift wird automatisch gespeichert.

12.4 Der Dateninspektor

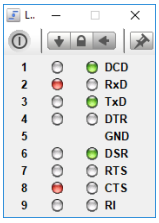
Die über die serielle Leitung transportierten Daten sind die eine Sache. Um Kommunikationsproblem auf die Schliche zu kommen, ist es aber oftmals erforderlich, das genaue Zeitverhalten der transportierten Daten zu kennen. Oder mit anderen Worten: Welche Zeitdifferenz besteht zwischen zwei Datenbytes? Wie lange dauert es, bis auf einen gesendeten String die Antwort eintrifft? Klicken Sie auf das Augen Symbol in der Werkzeugleiste oder drücken Sie Strg+I um den Dateninspektor zu öffen. Der Dateninspektor gibt Ihnen eine Reihe von Informationen:

Dateninspektor
zeigt Abstände und konvertiert Datenbytes

- **Position:** Nummer des Bytes sowie seine Herkunft.
- **Status:** Fehlerstatus des Bytes, z.B. Parity, Framing oder Break.
- **Absolute Zeit:** Liefert die absolute Zeit im lokalen Zeitformat, an der das Datenbyte gesendet wurde.
- **Zeitabstand:** Zeigt den Abstand zum vorherigen und folgenden Byte an.
- **Konvertierung:** Konvertiert den Wert unter dem Cursor in andere Zahlenformate.

Anzeige des Leitungsstatus

Um gleichzeitig den aktuellen Leitungsstatus zu dem Datenbyte zu sehen, öffnen Sie einfach den 'Virtuellen Ledtester' (Leitungsstatus) im Kontrollprogramm und schalten diesen auf synchronisierenden Betrieb.



Virtueller Ledtester
zeigt Pegelzustände

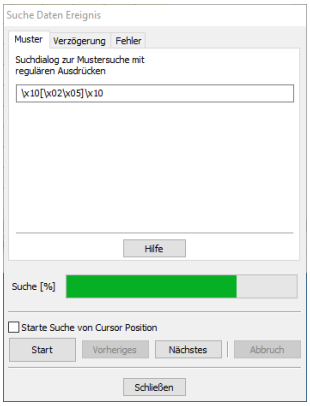
12.5 Aufzeichnung durchsuchen

Der Datenmonitor enthält eine Reihe von optimal auf die Datensuche abgestimmte Suchmechanismen. Sei es die Suche nach einer ganz bestimmten Folge von Datenbytes, eine zeitliche Unter- oder Überschreitung zwischen Anforderung und Antwort oder einfach nach Übertragungsfehlern wie Paritäts-, Rahmenfehler oder Breaks.
Da jede Suche von Beginn oder von der aktuellen Cursorposition gestartet werden kann, können alle Suchmechanismen beliebig kombiniert werden.

12.5.1 Mustersuche

Eine der herausragenden Eigenschaften des Datenmonitors ist die Suche nach ganz bestimmten Datensequenzen. Die Sucheingabe beschränkt sich hierbei nicht nur auf das einfache Vergleichen von Datenfolgen. Vielmehr erlaubt der Suchdialog auch die Eingabe sogenannter regulärer Ausdrücke.

Reguläre Ausdrücke entsprechen einer stark erweiterten Form der Platzhalter oder 'wild card' Zeichen, die Sie vom MSDOS DIR Kommando kennen. So listet z.B. das Kommand DIR *.HTM alle Dateien auf, die die Endung HTM tragen.



Beliebige Sequenzen suchen mit regulären Ausdrücken

DIR DATEI?.TXT liefert die (falls vorhandenen) Dateien DATEI1.TXT, DATEI2.TXT etc. Ähnliche Mechanismen zur Suche bestimmter Datensequenzen bietet der Suchdialog des Datenmonitors. Geöffnet wird er wahlweise mit dem Suchsymbol in der Werkzeugleiste oder einfach mit Strg+F.

Die Suche startet per Voreinstellung mit dem Beginn der Datenaufzeichnung. Sie können eine Mustersuche aber auch ab einem beliebigen Zeit/Startpunkt durchführen. Positionieren Sie dazu einfach den Cursor des Datenmonitors an der gewünschten Startposition und aktivieren Sie das Feld Starte Suche von Cursor Position.

Um eine ganz bestimmte Datensequenz innerhalb der Aufzeichnung zu finden, müssen Sie diese Sequenz zunächst in dem Texteingabefeld beschreiben. Das kann eine ganz einfache Zeichenfolge sein, z.B. LOGIN in einer Modemverbindung. Klicken Sie den Suche Knopf um den bis dahin aufgenommenen Datenstrom nach dieser Zeichenfolge durchsuchen zu lassen.
Die Suche beschränkt sich dabei immer auf den angezeigten Datenkanal. Haben Sie Kanal A ausgewählt werden nur alle diesem Datenkanal zugeordneten Bytes durchsucht, entsprechendes gilt für B.
Bei der Anzeige beider Datenkanäle werden alle aufgenommenen Daten berücksichtigt. Nun ist es aber oftmals so, daß das, was Sie suchen, nicht durch eine einfache Zeichenfolge beschrieben werden kann. Beispielsweise enthält die Datenaufzeichnung das Wort 'LOGIN' in folgenden Kombinationen: LOGIN, Login oder login. Die beiden letzten könnten Sie analog zu MSDOS mit der Eingabe ?ogin suchen. Um alle drei Varianten zu finden, müssen Sie das Suchmuster beschreiben als: Aneinanderreihung der Buchstaben L,O,G,I,N, wobei jedes einzelne Zeichen wahlweise groß oder klein geschrieben werden darf.
Die Umsetzung erfolgt mit einem regulären Ausdruck, dessen Ausdrucksmöglichkeiten in der unten aufgeführten Tabelle aufgelistet sind.

```
[Ll][Oo][Gg][Ii][Nn]
```

Jedes einzelne Zeichen wird durch eine Menge beschrieben, die exakt dem jeweiligen Klein- und Großbuchstaben entspricht.
Angenommen, Sie untersuchen eine Datenverbindung, bei der hin und wieder statt einem CRLF (Carridge Return, Linefeed) das CR unterdrückt wurde. D.h. Sie suchen ein einzelnes LF, welches KEIN vorangehendes CR enthält. Der dazu passende reguläre Ausdruck lautet

```
[!\x0D]\x0A
```

und liest sich als: Alle Zeichen außer Hex 0D (entspricht dem CR), gefolgt von Hex 0A (LF).
Ein regulärer Ausdruck ist eine Folge beliebiger Zeichen, wobei bestimmte Zeichen eine Sonderfunktion inne haben. Sie sind in der folgenden Tabelle aufgeführt. Wollen Sie eines dieser Sonderzeichen als normales Zeichen verwenden, z.B. weil Sie die Sequenz Passwort? suchen, und '?' hier NICHT für ein beliebiges Zeichen sondern wirklich für das Fragezeichen steht, müssen Sie es 'quoten'. Dies geschieht durch Voranstellen des \ Zeichens. Also

```
Passwort$\backslash$?
```

Mit dem '*' Zeichen in einem Suchmuster werden beliebige Datensequenzen bezeichnet. Es macht nur Sinn, wenn es von anderen 'Suchvorgaben' eingerahmt wird. Für sich alleine trifft es einfach auf alles zu, was sicherlich nicht dem Gedanken an eine Suche entspricht.
Der folgende Ausdruck findet alle Namenseingaben, die zwischen einem 'LOGIN' und 'PASSWORT' stehen, nicht aber allein stehende 'LOGIN' Sequenzen ohne folgende 'PASSWORT' Sequenz:

LOGIN*PASSWORT

Suche nach 9-Bit Sequenzen

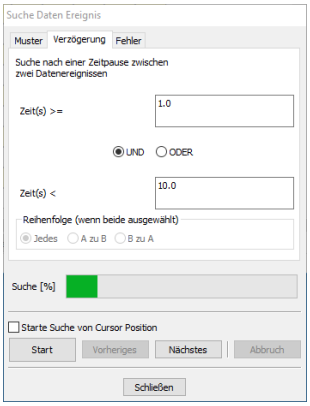
Als besonderen Fall unterstützt der Suchmechanismus des Datenmonitors auch 9-Bit Werte. Ein 9-Bit Wert kann nicht als normales Zeichen eingegeben werden. Der Datenmonitor erweitert deshalb die oben beschriebene Angabe von beliebigen Hexzahlen um eine 'spezielle' 3-stellige Hex Eingabe. Diese wird immer mit einem großen \X eingeleitet, gefolgt von dem 3-stelligen Hexwert.
Das nachstehende Beispiel sucht nach einer Sequenz, die entweder mit einem Hexwert 10B oder 133 startet, gefolgt von einem Byte Hex 33:

[\X10B\X133]\X033

Die folgende Tabelle gibt Ihnen einen Überblick über die im Suchdialog implementierten Ausdrücke.

Ausdruck	Bedeutung
?	ein beliebiges Zeichen
*	eine beliebige Zeichenkette
[abc]	ein Zeichen aus der angegebenen Menge *abc*
[!abc]	ein Zeichen das nicht der Menge *abc* angehört
\xHL	ein Zeichen in hexadezimaler Schreibweise, H entspricht den oberen 4 Bit, L den unteren 4 Bit
\X1HL	das gleiche wie zuvor, allerdings für 9-Bit Werte. Das erste Zeichen muss immer entweder 0 oder 1 sein. Der gültige Bereich ist von hex 000 bis 1FF. Beachten Sie das ein 'großes X' immer einen 9-Bit Wert verlangt!
\?	das Zeichen ?
*	das Zeichen *
\[	das Zeichen [
\]	das Zeichen]
\d	eine beliebige Ziffer 0...9
\n	das Zeichen für einen Zeilenumbruch (Linefeed hex 0x0A)
\s	ein beliebiges Whitespace Zeichen (Leerzeichen, Zeilenumbruch, Zeilenrücklauf, Tabulator)
\\	das Zeichen \

Ungültige Eingaben werden zur einfachen Korrektur im Eingabefeld markiert.



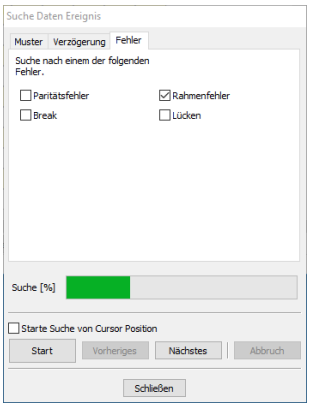
Zeitabstände finden und Sendepausen ermitteln

12.5.2 Zeitabstände suchen

Neben der Suche nach beliebigen Zeichenketten bietet der Datenmonitor auch die Suche nach bestimmten Zeitabständen zwischen zwei Datenereignissen. Klicken Sie dazu auf das Suchsymbol in der Werkzeugleiste oder drücken Sie Strg+F und wählen Sie den Reiter Verzögerung aus.
Die Zeitangaben erfolgen immer in Sekunden, beispielsweise 0.0015 für 1.5ms. Die kleinste Zeiteinheit entspricht dabei der Auflösung des Analysers, d.h. Zeitabstände von 0.000001 oder 1μs (MSB-PLUS Serie 10ns).
können als Über- oder Unterschreitung eines bestimmten Wertes oder aber als Bereich definiert werden. Der Knopf mit dem symbolisierten Verketten Symbol bestimmt, ob beide Zeitangaben für das Suchergebnis gültig sein müssen (UND Verknüpfung), ODER nur eines von beiden (was der Voreinstellung entspricht).
Betrachten Sie hierzu die folgende Tabelle:

Zeit(s) >=	Logik	Zeit(s) <	Ergebnis
1.000000s	ODER	0	Findet alle Abstände die größer als 1s ODER kleiner als 0s sind. Negative Zeiten gibt es nicht, so dass hier nach einem Bereich größer als der angegebenen Zeit von 1s gesucht wird.
1.000000s	UND	2.000000s	Findet alle Abstände die größer als 1s UND kleiner als 2s sind.
10000000s	ODER	0.001s	Findet alle Abstände die größer als 1000000s ODER kleiner als 1ms sind. Da solch große Abstände wie 1000000s vermutlich nicht vorkommen, werden nur alle Zeiten kleiner als 1ms gefunden.

Neben den Zeitvorgaben spielt auch die Datenfolge eine wichtige Rolle. D.h. ob der zeitliche Abstand zwischen zwei Datenbytes einer Quelle, z.B. Datenkanal A gemessen wird. Oder aber zwischen einem Datenbyte, aufgezeichnet durch den Datenkanal A gefolgt von einem Datenbyte (einer möglichen Antwort) von Datenkanal B. Je nach Wiring können Sie so gezielt nach Antwortzeiten eines bestimmten Busteilnehmers suchen bzw. das zeitliche Antwortverhalten prüfen. Die Reihenfolge, die beim Suchen berücksichtigt werden soll, können Sie explizit vorgeben. Voreingestellt ist Jedes, d.h. die Reihenfolge ist unerheblich.
Beachten Sie, dass eine Reihenfolge nur dann vorgegeben werden kann, wenn Sie im Datenmonitor auch beide, d.h. A und B ausgewählt haben. Andernfalls ist diese Einstellung deaktiviert.



Übertragungsfehler suchen und finden

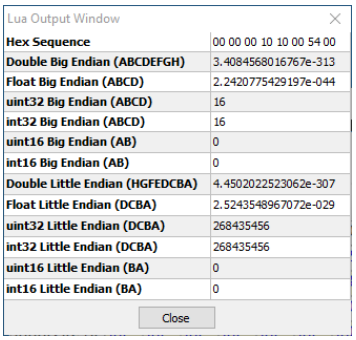
12.5.3 Übertragungsfehler suchen

Die Fehlersuche des Datenmonitors bezieht sich auf Fehler in der Datenübertragung. Dabei handelt es sich um Rahmenfehler und Paritätsfehler. Breaks

sind im eigentlichen Sinne kein Fehler. Da sie aber nicht mit einem binären Nullbyte verwechselt werden dürfen und in der seriellen Kommunikation oftmals zur Initialisierung bzw. zum Rücksetzen eines Teilnehmers verwendet werden, sind sie hier ebenfalls integriert.
Die Suche nach Fehlern ist leicht. Markieren Sie einfach einen oder mehrere Fehlerzustände und starten Sie die Suche mit Klick auf den Startknopf.

12.6 Integrierte Skriptsprache Lua

Der Datenmonitor war das erste Analysefenster mit integrierter Lua Unterstützung. In Version 5.0 wurde diese nochmals komplett überarbeitet und mit einer Menge neuer Features ausgestattet, die die Verarbeitung der Daten noch einfacher und effizienter gestaltet.



Zahlen Konverter
geschrieben in Lua

Zuallererst: Das bisherige Design von Zeilen orientierten 'Watch' Ausdrücken die unter der Datenanzeige ein/aus geklappt werden konnten, wurde durch ein allgemeines und frei positionierbares Ausgabefenster ersetzt. Das neue Ausgabefenster erlaubt nicht nur die Anzeige einzeiliger Informationen sondern auch komplexe Tabellen wie im Bild links zu sehen.

Und: Sie sind nicht länger auf den doch recht limitierten Editor früherer Versionen angewiesen. Der Datenmonitor verwendet nun den eigenständigen, komplett ausgestatteten Editor, der ab Version 5.0 der Analyser Software das Editieren von Lua Skripten übernimmt.
Der neue Editor hilft nicht nur mit vordefinierten Code Gerüsten, er erlaubt Ihnen zudem das Testen/Ausführen ausgewählter Code Zeilen und kompletter Skriptdateien.
Die Fähigkeiten des Editors sind so umfangreich, dass wir sie in einem eigenen Kapitel 17 beschreiben.

Der Lua Interpreter des Datenmonitor hilft Ihnen Fragen wie diese zu beantworten:

- Was ist die CRC16 Prüfsumme einer gegebenen Datenfolge?
- Welchem vorzeichenlosen 32 bit Wert entsprechen die ausgewählten 4 Bytes?
- Wie lange dauerte es, eine bestimmte Bytefolge zu übertragen?

Es gibt kaum etwas, was Sie nicht mit dem integrierten Lua Interpreter tun können, solange Ihr Code sich an die Begebenheiten des Datenmonitors hält.

12.6.1 Wie funktioniert es?

Der Austausch von Daten zwischen dem Datenmonitor und Lua wird durch einen recht simplen Mechanismus gewährleistet. Der Datenmonitor übergibt alle nötigen Informationen wie Cursor Position und optionale Auswahl von Datenfolgen an Lua. In Lua können Sie wahlfrei auf alle aufgenommenen Datenbytes zugreifen und sie nach belieben verarbeiten. Das Ergebnis geben Sie entweder als einfache Lua Wert (Zahl, String) oder als Tabelle von Schlüssel/Werte Paare zurück. Im Detail:

Der Datenmonitor führt bei jeder Cursor Bewegung oder Update des Skripts eine ganz bestimmte Lua Funktion aus. Diese Funktion `onchange` wird mit

der aktuellen Cursor Position und optionalem Start/Ende einer im Datenmonitor ausgewählten Bytefolge aufgerufen. Die Funktion ist definiert als:

```
1  function onchange( cursor, selbeg, selend )
2      -- input your code here
3  end
```

Im Funktionsrumpf können Sie durch das Datenmonitor eigene `data` Modul auf jedes aufgenommene Byte zugreifen - absolut oder relative zur aktuellen Cursor Position. Ein Beispiel:

```
1  function onchange( cursor, selbeg, selend )
2      -- input your code here
3      return data.at( cursor ):val()
4  end
```

Die Funktion `onchange` liefert hier einfach den Datenwert an der aktuellen Cursor Position zurück. Mittels der Modul Funktion `data.at` können Sie Informationen zu jedem Byte durch Angabe der Byte Position - wie in der Adressleiste zu sehen - abfragen.
Das Resultat von `data.at(cursor)` ist ein Datenobjekt welche alle wichtigen Eigenschaften des Datenbytes enthält. Diese sind neben dem eigentlichen Wert (auch 9-Bit), die Quelle (oder Richtung) sowie wann das Datenbyte empfangen wurde. Hier fragen wir allerdings nur den Wert ab und geben diesen als einfach Lua Zahl zurück.

Der Rückgabewert von `onchange` wird immer als Lua Tabelle mit Schlüssel/Werte Paaren interpretiert. Ist es wie hier nur ein einfacher Wert, wird dieser als Schlüssel/Name angezeigt und der Wert bleibt blank. Das Ergebnis sieht dann so aus:

A8	

Das von `data.at()` gelieferte Datenobjekt enthält - wie gesagt - noch weitere Informationen, die wir ebenfalls abfragen und anzeigen können. Erweitern wir unser Beispiel deshalb zu:

```
1  function onchange( cursor, selbeg, selend )
2      -- input your code here
3      local d = data.at( cursor )
4      return {
5          ["Value"] = string.format("%02X", d:val() ),
6          ["Time"] = d:time().."s",
7          ["Source"] = d:dir()
8      }
9  end
```

In Zeile 3 speichern wir zunächst das Ergebnis von `data.at()` als lokale Variable. `d` enthält damit alle von uns benötigten Informationen. In Zeile 4 erzeugen wir eine Lua Tabelle, fügen diese als Schlüssel/Werte Paare (siehe 18.2.5) ein und geben sie zurück. Das Ergebnis im Lua Ausgabefenster sieht so aus:

Source	1
Time	354.84122s
Value	3F

Da das Ergebnis bereits als Tabelle (oder Array) von Schlüssel/Werte Paare vorliegt, ist es einfach, dieses als zweispaltige Tabelle darzustellen. Die linke Spalte enthält dabei die Schlüssel (oder Namen), die rechte die zugehörigen Werte.
Sie werden bemerken, das die Ausgabe in einer anderen Reihenfolge erfolgt. Der Datenmonitor sortiert die von `onchange` gelieferte Tabelle immer in der alphabetischer Reihenfolge der Schlüssel/Namen. Daher Source, Time, Value. Wir kommen später noch einmal darauf zurück.

Im nächsten Beispiel wollen wir die Prüfsumme einer im Datenfenster markierten (selektierten) Bytefolge ausrechnen und ausgeben.

```
1  function Mod256Sum( from, to )
2    local chksum = 0
3    for i=from,to do
4      chksum = chksum + data.at( i ):val()
5    end
6    return chksum % 256
7  end
8
9  function onchange( cursor, selbeg, selend )
10     if selbeg and selend then
11         if selbeg > selend then
12             selbeg, selend = selend, selbeg
13         end
14         return { ["Result:"] = Mod256Sum( selbeg, selend ) }
15     else
16         return { ["Result:"] = "No selection!" }
17     end
18 end
```

Die Zeilen 1...7 zeigen die Funktion zur Berechnung der Prüfsumme. Hier das einfache Aufaddieren aller Datenbytes von Position `from` bis `to` mit abschließendem Modulo 256 (Lua `%` Operator) um nur die untersten 8 Bits zurückzugeben.
Der Datenmonitor übergibt die ausgewählte Datenfolge als Start `selbeg` und Ende Position `selend`. Bei keiner Auswahl sind beide Parameter nil und das Skript liefert 'No selection! in Zeile 16'.

Die Auswahl einer Datensequenz kann vom Anfang zum Ende oder umgekehrt erfolgen. Bei letzterem führt dies zu einem Startposition kleiner als die Endposition. Das Skript prüft dies in Zeile 11 und verwendet eine kleine Lua Spezialität um beide Werte mit einer einzigen Anweisung zu vertauschen[3].
Es folgt der Aufruf der eigentlichen checksum Funktion und die Rückgabe der berechneten Prüfsumme als Dezimalwert.

In unserem letzten Beispiel wollen wir das Markieren (Färben) von Datenbytes relativ zur Cursor Position demonstrieren.
Dies ist insbesondere interessant wenn Sie den Datenmonitor mit dem Protokollmonitor synchronisieren. Sobald Sie ein Telegramm im Protokollmonitor anklicken, sendet dieser Anfang und Ende des Telegramms als Auswahl an den Datenmonitor. Sie können diese Information in `onchange` nutzen, um gezielt

[3]Lua erlaubt mehrfache Zuweisungen. In diesem Fall wertet Lua zuerst alle Werte aus und führt dann die Zuordnungen aus. Ideal zum Austauschen zweier Werte

Datenbytes dieses Telegramms in der Anzeige zu markieren oder anderweitig zu verarbeiten.
Ein entsprechendes Skript für Modbus RTU Übertragungen verwendet dieses Feature. Sie finden es unter den DataView Vorlagen.
Wir beschränken uns hier auf die Basics und zeigen, wie Sie einfach das vorherige, aktuelle und auf den Cursor folgende Byte rot, grün und blau hervorheben können.

```
function onchange( cursor, selbeg, selend )
    -- input your code here
    data.cursorcolours{
        [-1] = 0xFF0000,  --> red
        [0] = 0x00FF00,  --> green
        [1] = 0x0000FF    --> blue
    }
end
```

Die genauen Details erklären wir in Abschnitt 12.7.1 aber die Funktion sollte klar genug sein. Die Tabelle `cursorcolours` ist eine typische Lua Tabelle mit Schlüssel/Werte Paaren. Dabei entspricht der Schlüssel der relativen Cursor Position (0 bedeutet hier die aktuelle Cursorposition), der Wert der Farbe in RGB Hex Notation[4].

Jedes mal, wenn Sie den Cursor bewegen (und damit `onchange` aufrufen) wird der Inhalt der Tabelle `cursorcolours` vom Datenmonitor ausgewertet und die Anzeige entsprechend aktualisiert. Die Rückgabe eines bestimmten Wertes ist nicht erforderlich, es sei denn Sie benötigen zusätzliche Informationen im Lua Ausgabe Fenster.

12.6.2 Sortierte Resultate

Zurück zu unserem ersten Beispiel. Wir erwähnten, das Lua Tabellen Schlüssel/Werte Paare in keiner spezifischen Reihenfolge enthält[5]. Der Datenmonitor ordnet deshalb die Schlüssel/Werte Paare in der alphabetischen Reihenfolge der Schlüssel an. Dies ist aber nicht immer die Reihenfolge in der Sie es codiert haben:

Source	3
Time	2
Value	1

```
return {
    ["Value"] = 1,
    ["Time"] = 2,
    ["Source"] = 3
}
```

Sie können die Reihenfolge durch Voranstellen eines Text Präfix mit endendem Tabulator Zeichen beeinflussen um das gewünschte Resultat zu erzielen:

Value	1
Time	2
Source	3

```
return {
    ["1\tValue"] = 1,
    ["2\tTime"] = 2,
    ["3\tSource"] = 3
}
```

Der Datenmonitor ignoriert alle mit einem Tab vorangestellten Zeichen (inklusive des Tabulator Zeichens) bevor es den Schlüsselnamen ausgibt. Nicht desto

[4] 0xRRGGBB Notation, wobei RR der 8-Bit Rotanteil Hexwert, GG der 8-Bit Grünanteil Hexwert und BB der 8-Bit Blauanteil Hexwert ist

[5] Es gibt eine Reihenfolge! Diese basiert aber auf der Berechnung des internen Hash Wertes und ist für weder relevant noch nützlich.

trotz ist dies ziemlich umständlich. Vor allem dann, wenn Ihre Tabelle mehr als 3 Zeilen enthält und Sie die Reihenfolge beim Codieren öfters anpassen müssen.

Es wäre schön, den Präfix automatisch und unabhängig zu vergeben, so dass dieser beim Editieren nicht weiter berücksichtigt werden muss. Mit einer simplen Sortierfunktion können wir dies auf einfache Art und Weise realisieren:

```
1  function onchange( cursor, selbeg, selend )
2      local d = data.at( cursor )
3      local n = 0
4
5      function sort()
6          n = n + 1
7          return string.format("%02i\t", n)
8      end
9
10     return {
11         [sort().."Value"] = string.format("%02X", d:val() ),
12         [sort().."Time"] = d:time(),
13         [sort().."Source"] = d:dir(),
14     }
15 end
```

Zeile 3 legt zunächst eine lokale Zählvariable (zum Sortieren) an. Zeile 5...8 ist die eigentliche Sortierfunktion und liefert bei jedem Aufruf den nächsten Präfix inklusive Tab Zeichen. Wir verwenden hier ausdrücklich ein 2-stelliges Zahlenformat `"01\t"..."99\t"`, da ansonsten z.B. der Wert 11 zwischen 1 und 2 einsortiert werden würde.
Zu guter Letzt müssen wir lediglich die `sort()` Funktion unseren Schlüsselnamen voranstellen, wie in den Zeilen 11...13 gezeigt.

12.6.3 Lua Skript auswählen und ausführen

Die Analyser Software enthält eine Reihe von Beispiel Skripten für den Datenmonitor. Eines davon wandelt die Datenbytes an Cursor Position in verschiedenste Zahlenformate um und ist sehr hilfreich, wenn Sie die übertragenen Zahlenwerte eines Protokolls prüfen wollen.
Ein anderes Skript berechnet mehrere Standard Prüfsummen einer ausgewählten Datensequenz.
Der Datenmonitor fasst alle Lua relevanten Bedienelemente unter dem Hauptfenster zusammen. Hier können Sie ein Lua Skript auswählen, im Editor öffnen oder ein neues anlegen.

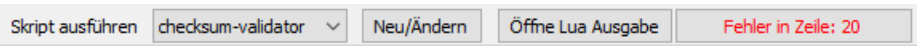
Der Knopf 'Öffne Lua Ausgabe' öffnet das Skript Ausgabefenster zur Anzeige der vom ausgewählten Skript zurückgegebenen Resultate. Das Ausgabefenster bleibt geöffnet bis Sie es wieder schließen.

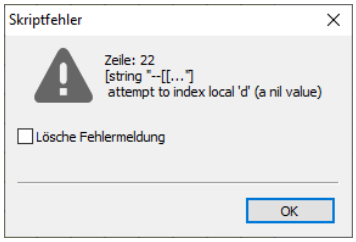
Beachten Sie!
Der Datenmonitor besitzt keinen expliziten Lua ein/aus Schalter. Der integrierte Lua Interpreter benötigt nur wenige Resourcen und kann deshalb aktiv im Hintergrund bleiben - auch wenn gerade kein Lua Ausgabefenster offen ist.

Wenn Sie definitiv keine Interaktion mit Lua wollen (z.B. keine Markierung von Daten, etc.) wählen Sie einfach das **idle** Skript aus. Es macht das, was der Name sagt, nämlich untätig sein.

12.6.4 Skript Fehler

Niemand ist perfekt. Das gilt natürlich auch beim Schreiben selbst kleiner Lua Programme. Schnell schleicht sich hier und da ein Syntax oder Schreibfehler ein. Der Editor selbst kann Ihre Eingaben nur begrenzt prüfen, da der eigentliche Code im Datenmonitor ausgeführt wird[6].
Etwaige Fehler werden deshalb im Datenmonitor selbst angezeigt. In diesem Fall wird der Lua Fehler Knopf aktiv:





Fehler Dialog

Solange kein Fehler auftritt ist der Knopf inaktiv. Im Fehlerfall zeigt der Knopf die Zeile des Fehlers in roter Schrift an und Sie können den Knopf anklicken um weitere Informationen zu dem Fehler aufzurufen.
Der sich öffnende Fehlerdialog beschreibt die Art des Fehler sowie die Zeile in Ihrem Skript, an der er aufgetreten ist (und Sie ihn im Editor korrigieren können).
Die rote Fehlermeldung verschwindet automatisch, sobald Sie den Fehler beseitigt und die Änderungen gespeichert haben. Temporär können Sie die Fehlermeldung auch löschen, indem Sie 'Lösche Fehlermeldung' im Dialog anklicken bevor Sie diesen schließen.

12.6.5 Skript Debuggen

Einen Fehler im Skript zu erkennen ist die eine Sache, ihn zu beheben eine ganz andere. Ein Mittel der Wahl bei interpretierenden Skript Sprachen (aufgrund der kurzen Edit/Test Zyklen) ist das Platzieren von Debug Ausgaben an entsprechenden Stellen. Damit können Sie das Ausführung bestimmter Code Abschnitte überprüfen und/oder sich den Inhalt ausgewählter Variablen anzeigen lassen.
Der Datenmonitor bietet hierzu sein eigenes Debug Ausgabefenster. Sie können es jederzeit im Menü 'Ansichten' oder per Tastenkürzel Strg + Alt + O öffnen.

Das `debug` Modul wird weiter unten noch detailliert beschrieben. Hier nur in Kürze: Sie können eine beliebige Anzahl von Lua Variablen, Text oder die Kombination von beiden mit folgender Funktion ausgeben:

```
1 function onchange( cursor )
2     debug.print( "Cursor Position", cursor )
3 end
```

Sie können die Debug Ausgabe zudem an eine spezielle Funktion delegieren und die Ausgabe mit einem zentralen Flag ein bzw. ausschalten.

```
1 DEBUG=true
2
```

[6] Der Editor erlaubt Ihnen, ausgewählte Code Zeilen oder ganze Buffer Inhalte auszuführen. Er kennt allerdings keine Datenmonitor relevanten Funktion wie z.B. onchange und behandelt diese deshalb als Fehler!

```
3  function print(...)
4      if DEBUG then
5          local s = ""
6          for i,v in ipairs( arg ) do
7              s = s..v.."\t"
8          end
9          debug.print( s )
10     end
11 end
12
13 function onchange( cursor, selbeg, selend )
14     local d = data.at( cursor )
15     if d then
16         print( "Cursor:", cursor )
17     end
18 end
```

Die Funktion `debug.print(...)` akzeptiert eine variable Anzahl von Argumenten. Deshalb iteriert die `print` Funktion in unserem Beispiel über alle Parameter (Zeile 6). In Zeile 7 werden diese zu einem per Tab separierten String zusammen gesetzt und in Zeile 9 dann endgültig ausgegeben. Vorausgesetzt die globale Variable `DEBUG` in Zeile 1 ist true.

12.6.6 Speicherort der Protokoll Templates

Der Speicherort der Template Vorlagen für den Datenmonitor (sowie aller anderer Lua Skripte) hat sich mit Version 5.0 geändert.

Linux Anwender finden Sie nun unter:

```
~/.IFTOOLS/SerialAnalyzer/7.0.2/Templates/DataView
```

Für Windows Anwender ist der Speicherort:

```
C:\Users\USERNAME\AppData\Roaming\IFTOOLS\SerialAnalyzer\7.0.2\Templates\DataView
```

Dies ist aber nur von informeller Bedeutung. Wenn Sie ein neues Datenmonitor Template/Skript anlegen oder ein vorhandenes unter einem anderen Namen speichern sorgt der Editor dafür, das diese immer im richtigen Verzeichnis abgelegt werden. Und das mit gutem Grund:
Der Datenmonitor prüft dieses Verzeichnis nach neuen Skripten sobald Sie den Skript/Template Auswahlknopf klicken. Alle gefundenen Templates werden dabei alphabetisch in der Auswahlliste angezeigt.
Sie können - natürlich - Templates auch unter einem anderen Ort speichern. Z.B. um Sie auf einen anderen Datenträger zu kopieren oder anderweitig zu verwenden.
Die Analyser Software allerdings berücksichtigt nur Skripte in dem vorgegebenen Ort!

12.6.7 Ein Template importieren

Manchmal möchten Sie vielleicht ein Skript aus einer anderen Quelle verwenden. Z.B. ein Template von einem Kollegen oder aus dem Internet (der IFTOOLS Download Seite). Wie bereits erwähnt, müssen Sie das Template im richtigen Verzeichnis speichern, damit der Datenmonitor es findet.

Importieren oder Hinzufügen eines neuen Templates ist einfach. Ziehen Sie dazu einfach das gewünschte Template (Dateiendung msbtml) in das offene Telegramm Fenster des Datenmonitors (drag and drop). Das ist schon alles!
Das Programm speichert automatisch das Template im entsprechenden Verzeichnis und wendet es auf die aktuelle Aufzeichnung an.
Im falle eines Fehlers bekommen Sie eine entsprechende Meldung im Fehlerknopf der Statuszeile. Das Programm warnt Sie zudem, wenn ein Template gleichen Namens bereits existiert.

12.6.8 Wie kann ich überflüssige Skripte löschen

Alle Datei Operationen wie Laden, Editieren und Speichern obliegen dem Verantwortungsbereich des Editors. Der Datenmonitor selbst bietet keinerlei Datei Funktionalität außer ein ausgewähltes Skript auszuführen.
Sie können allerdings jederzeit den Editor starten und dort das 'Öffne Datei' Icon in der Werkzeugleiste klicken um den Dateidialog Ihres OS zu öffnen. Dort können Sie nicht länger benötigte Dateien löschen oder in den Papierkorb verschieben.

12.6.9 Einschränkungen

Sie können beliebige Operationen in Lua ausführen, komplexe Funktionen schreiben und aufwendige Verrechnungen durchführen. Allerdings erlaubt der Datenmonitor jedem Lua Script nur eine bestimmte Anzahl von Rechenoperationen bzw. Zeitdauer zur Ausführung. Sobald Ihr eingebenenes Script diese überschreitet bekommen Sie eine entsprechende Fehlermeldung zu sehen. Und das hat seinen guten Grund:
Sollten Sie aus welchem Grund auch immer eine Endlosschleife programmieren wird Sie der Datenmonitor freundlich darauf hinweisen anstatt wortlos jegliche weitere Zusammenarbeit einzustellen.

12.7 Datenmonitor spezifische Lua Erweiterungen

Der Datenmonitor erweitert Lua mit mehreren Modulen, die auch in anderen Views mit Lua Interpreter verwendet werden. All diese werden im Detail in Kapitel 19 beschrieben. Ein exklusiv im Datenmonitor verfügbare Modul ist das `data` Modul.
Das `debug` Modul kennen Sie evtl. bereits aus dem Protokollmonitor. Auch wenn es sich bei diesem um keine spezielle Datenmonitor Erweiterung handelt, beschreiben wir dennoch hier, um seine Arbeitsweise im Kontext des Datenmonitors zu zeigen.

12.7.1 Das data Modul

Das `data` Modul bietet den nötigen wahlfreien Zugriff auf alle empfangenen Datenbytes per Adress/Position Index.
Beachten Sie das die aufgenommenen Daten von 0 gezählt werden und das die Adresse/Position eines Datenbytes von der ausgewählten Anzeige A, B bzw. A+B abhängt.
Das `data` Modul erlaubt Ihnen zudem, Datenbytes relativ zur aktuellen Cursor Position zu markieren. Das Modul im Detail:

Funktion	Beschreibung
at	Liefert Informationen zu dem Datenbyte an der gegebenen Adresse/Position. Dies beinhaltet den Datenbytewert (inkl. 9-Bit), die Richtung (Quelle A oder B) sowie den Zeitpunkt des Auftretens.
cursorcolours	Eine interne Tabelle, die Farbangaben zu relativen Cursor Positionen enthält.

data.at

Liefert ein Datenobjekt welches alle wichtigen Informationen zu dem Datenbyte an der angegebenen Adresse/Position enthält. Das Datenobjekt erlaubt die Abfrage des Byte Wertes mit `val()`, die Richtung bzw. Quelle mit `dir()` sowie mit `time` den Zeitpunkt (Zeitstempel) an dem der Analyser das Datenbyte aufgezeichnet hat.

data.at(*index*)

- **index:** Die Adresse bzw. Position des Datenbytes im Bereich der zum Zeitpunkt der Abfrage vorhandenen Datenbytes. Ein ungültiger Index (außerhalb des Aufnahmebereichs oder negativ) resultiert in einem **nil** Rückgabewert.

Sie können die gewünschten Information jedes mal durch direkten Aufruf von `data.at()` abfragen:

Example

```
1  function onchange( cursor, selbeg, selend )
2      -- input your code here
3      return {
4          ["Value"] = string.format("%02X", data.at( cursor ):val() ),
5          ["Time"] = data.at( cursor ):time()..."s",
6          ["Source"] = data.at( cursor ):dir()
7      }
8  end
```

Dies bedeutet allerdings eine erneute `data.at` Abfrage für jede Datenobjekt Eigenschaft. Auch wenn die Zugriffszeit recht schnell ist, ein besserer Ansatz ist das Datenobjekt lokal zu speichern und dieses dann zur Abfrage der jeweiligen Informationen zu benutzen. Es ist immer eine gute Strategie zuvor zu prüfen, ob der Aufruf von `data.at(...)` ein gültiges Datenobjekt liefert. Dies ist z.B. nicht der Fall, wenn Sie auf ein außerhalb des Aufnahmebereichs liegendes Datenbyte klicken (im Datenmonitor als graues XX gekennzeichnet). Das folgende Beispiel erledigt dies in Zeile 4.

Example

```
1  function onchange( cursor, selbeg, selend )
2      -- input your code here
3      local d = data.at( cursor )
```

```
4       if d then
5           return {
6               ["Value"] = string.format("%02X", d:val() ),
7               ["Time"] = d:time().."s",
8               ["Source"] = d:dir()
9           }
10      end
11  end
```

data.cursorcolours

Dies ist eine interne Tabelle von Schlüssel/Werte Paaren. Die Schlüssel entsprechen dabei einer relativen Cursor Position (z.B. 0, 1, 2 aber auch -1, -2), die Werte einer hexadezimalen RGB Farbnotation (z.B. 0xFF0000 für rot). Beachten Sie, dass im Gegensatz zur normalen Lua Indizierung die Cursor Positionen von 0 gezählt werden! 0 bedeutet die aktuelle Cursor Position, ein positiver Wert eine Position nach dem Cursor, ein negativer Werte eine Position vor dem Cursor.

Das folgende Beispiel markiert das Byte vor dem Cursor rot, das Byte an der aktuellen Cursor Position grün, und das auf den Cursor folgende Byte blau.

Examples

```
1   function onchange( cursor, selbeg, selend )
2       -- input your code here
3       data.cursorcolours{
4           [-1] = 0xFF0000,   --> red
5           [0] = 0x00FF00,   --> green
6           [1] = 0x0000FF     --> blue
7       }
8   end
```

12.7.2 Das debug Modul

Der Datenmonitor besitzt ein eingebautes Debug Fenster um spezielle Zustände in Ihrem Skript anzuzeigen. Dies kann der Inhalt einer bestimmten Variable sein, oder das Resultat einer Operation. Das Debug Modul enthält hierzu alle nötigen Funktionen um beliebige Werte oder Texte auszugeben. Zusätzlich können Sie die Debug-Ausgabe per Skript anhalten, zu einem geeigneten Zeitpunkt wieder aufnehmen und Ausgaben zusammenfassen. Z.B. bei wiederholenden Debug Meldungen. Um das Ausgabefenster für Debug Meldungen zu öffnen drücken Sie einfach:
Strg + Alt + O

Function	Description
clear	Löscht den aktuellen Inhalt im Debug Fenster.
print	Gibt die übergebenen Argumente im Debug Fenster aus. Sie können beliebig viele Argumente (Text oder Wert) per Komma getrennt im Funktionsaufruf angeben.
resume	Setzt eine zuvor unterbrochene Debug Ausgabe wieder fort.

summarize	Wenn aktiviert sammelt die Debug Ausgabe alle identischen Meldungen und gibt diese mit Angabe einer entsprechenden Wiederholungszahl aus.
suspend	Stoppt (suspendiert) die Debug Ausgabe. Alle weiteren Ausgaben werden unterdrückt bis die Ausgabe per `debug.resume` fortgesetzt wird.
timeprompt	Zeigt die aktuelle Zeit (hh:mm:ss) am Anfang jeder Debug Ausgabe. Sie können dies ein- bzw. ausschalten, indem Sie dieser Funktion true oder false übergeben.

debug.clear

Löscht den aktuellen Inhalt des Debug Ausgabefensters.

debug.clear()

Examples

```
1  function onchange( cursor )
2      -- print the current cursor position (address)
3      debug.clear()
4      debug.print( cursor )
5  end
```

debug.print

Gibt die per Komma separierten Argumente in dem Protokollmonitor internen Debugfenster aus.

debug.print(*param1,param2,...*)

- **param:** Durch Komma getrennte Liste der Parameter.

Examples

```
1  function onchange( cursor )
2      debug.clear()
3      debug.print( "Pos:"..cursor, "Value:"..data.at( cursor ):val()
4  end
```

Das Beispiel oben gibt die Cursor Position sowie den Wert des Datenbytes an der Cursor Position im Debug Ausgabefenster aus.

debug.resume

Setzt eine zuvor ausgesetzte Debugausgabe wieder fort. Es ist eher unwahrscheinlich, dass Sie diese Funktion benötigen. Sie ist aber Teil des `debug` Moduls und verdient eine kurze Beschreibung.

debug.resume()

Examples

```
1  function onchange( cursor)
2      local d = data.at( cursor )
3      if not d then
4          debug.suspend()
5      else
6          debug.resume()
7      end
8      debug.print( cursor )
9  end
```

In diesem Beispiel stoppen wir alle Debug Ausgaben wenn der Cursor auf ein Datenbyte außerhalb des bislang aufgenommenen Bereiches gesetzt wird. Sobald der Cursor wieder über ein gültiges Datenbyte platziert wird, schalten wir die Debug Ausgabe (resume) wieder ein. Zugegeben handelt es sich um ein etwas konstruiertes Beispiel, aber die Bedeutung sollte klar sein.

debug.summarize

Fasst identische Debug Meldungen zusammen und gibt sie erst aus, wenn eine davon unterschiedliche Meldung auftritt. Die wiederholten Meldungen werden wie folgt dargestellt:

```
THE DEBUG MESSAGE
The previous message repeated n times.
```

`n` bezeichnet die Anzahl der aufgetretenen Wiederholungen.
Gewöhnlich reicht es eine Anweisung wie `debug.summarize(true)` am Anfang des Templates zu setzen. D.h. außerhalb von `onchange()`, da diese Funktion nur einmal aufgerufen werden muss (siehe Zeile 1).

debug.summarize()

Examples

```
1  debug.summarize( true )
2
3  function onchange( cursor )
4      debug.print( data.at( cursor ):val()
5  end
```

Hier sammelt der `summarize` Mechanismus alle Debug Meldungen solange Sie den Cursor über Datenbytes des gleichen Wertes bewegen. Erst wenn der Cursor einen davon abweichender Wert trifft, wird die Anzahl der identischen Datenbytes sowie das neue, davon abweichende im Debug Fenster ausgegeben.

debug.suspend

Unterdrückt alle weiteren Debugausgaben via `debug.print` bis ein Aufruf von `debug.resume` diese wieder freigibt. Siehe das vorherige Beispiel (resume).

debug.timeprompt

Aktiviert oder deaktiviert die zusätzliche Zeitangabe wann die Debugausgabe erfolgte. Voreingestellt ist eine Ausgabe ohne Zeit.

Wenn eingeschaltet wird jeder Ausgabe die aktuelle Zeit im Format hh:mm:ss vorangestellt. Ein Beispiel:

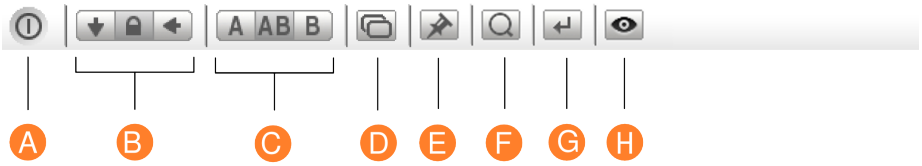
```
12:24:48: My debug message
```

Examples

```
1  debug.timeprompt( true )
2
3  function onchange( cursor )
4      debug.clear()
5      debug.print( "Pos:"..cursor, "Value:"..data.at( cursor ):val()
6  end
```

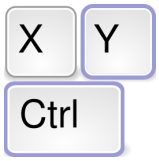
12.8 Die Werkzeugleiste

Die Werkzeugleiste dient zum schnellen Zugriff der am meisten benötigten Funktionen. Einige davon sind bei allen Monitoren indentisch, andere spezifisch für den Datenmonitor.



A	**Ende:** Speichert alle Einstellungen und schließt das Fenster
B	**Anzeigemodus:** Je nach Anzeigemodus zeigt der Fensterinhalt immer das aktuelle (zuletzt aufgenommene) Ereignis, ist verriegelt oder aktualisiert den Inhalt mit anderen Fenstern.
C	**Datenrichtung:** Der Datenmonitor kann beide Datenrichtungen (Port A und Port B) gemeinsam oder einzeln anzeigen, z.B. um diese getrennt in zwei einzelnen Fenstern darzustellen.
D	**Neue Ansicht:** Öffnet ein neues Fenster mit dem gleichen Ausschnitt und identischen Einstellungen.
E	**Default Einstellung:** Speichert die aktuellen Datenmonitor Einstellungen als Vorgabe wenn ein neues Datenmonitor Fenster geöffnet wird.
F	**Suchdialog:** Öffnet den Suchdialog zur Mustersuche oder Suche nach Übertragungspausen.
G	**Gehe zu...:** Öffnet den Gehe zu Dialog zur Eingabe einer absoluten Position oder relativem Offset.
H	**Dateninspektor:** Startet den Dateninspektor.

12.9 Kurzbefehle



Tastenkommandos der wichtigsten Funktionen

Aktion	Kurzbefehl
Online Hilfe zum Datenmonitor	F1
Auswahl als Region speichern	F4
Auswahl als Datenrahmen in Region speichern	Umschalt+F4
Markiere Startadresse der Auswahl	Strg+Linke Maustaste
Markiere Endadresse der Auswahl	Umschalt+Linke Maustaste
Alles auswählen	Strg+A
Auswahl aufheben	Umschalt+Strg+A
Ausgewählter Datenbereich in Zwischenablage kopieren	Strg+C
Ausgewählter Datenbereich exportieren	Strg+E
Suchdialog öffnen	Strg+F
Gehe zu Adresse... Dialog öffnen	Strg+G
Dateninspektor öffnen	Strg+I
Öffne Ansicht in einem neuen Fenster	Umschalt+Strg+N
Öffnet das interne Debug Ausgabefenster	Strg + Alt + O
Fenster schliessen	Strg+Q
Ausgewählte Daten als Binärdatei speichern	Strg+S

13
Der Ereignismonitor

Wann ist welches Ereignis aufgetreten? Hat während der Datenübertragung ein bestimmter Pegelwechsel stattgefunden oder wurde ein Fehlerzustand (Break, Paritäts- bzw. Rahmenfehler) erkannt. Wie war der Zustand der Busleitungen zu einem bestimmten Zeitpunkt.
Der Ereignismonitor listet Sie alle auf, sucht nach Ereignissequenzen oder Pegelzuständen und exportiert die Ereignisse als CSV Datei.

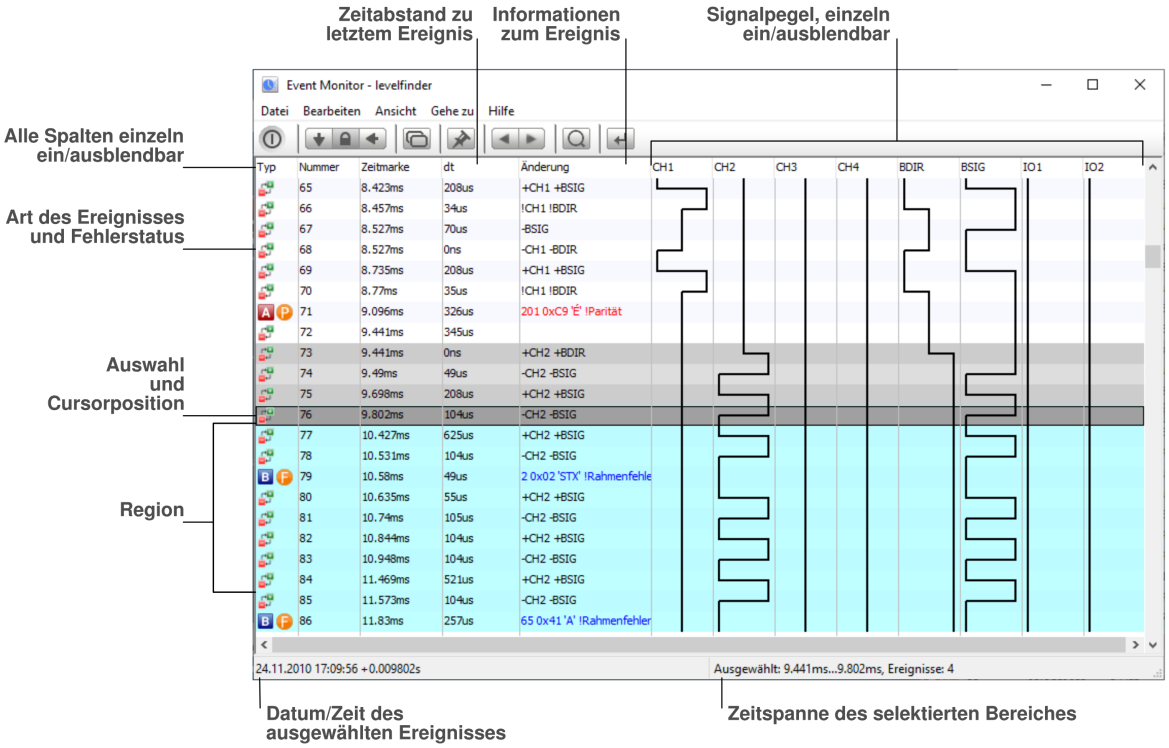
Im Gegensatz zum Datenmonitor stellt der Ereignismonitor alle aufgetretenen Ereignisse (Daten und Pegelwechsel) in ihrem zeitlichen Ablauf dar. Während der Datenmonitor umfangreiche Mechanismen zur Untersuchung von Datensequenzen bietet und die Datenansicht der Aufzeichnung repräsentiert, ist der Ereignismonitor auf die Ansicht und Suche von Leitungsänderungen optimiert. Dies betrifft insbesondere Änderungen im Pegel der Steuersignale, aber auch asynchrone Ereignisse wie framing, parity Fehler oder breaks.
Jedes Ereignis ist mit einer Zeitmarke versehen, die den genauen Zeitpunkt seines Auftretens mit einer Auflösung von 10ns wiedergibt. Der zeitliche Abstand zwischen den Ereignissen hat allerdings keine Einfluß auf die Darstellung. Leitungszustände bzw. Wechsel vor und nach einem bestimmten Zeitpunkt der Aufzeichnung sind deshalb leicht zu erkennen.

Die Suche nach bestimmten Datensequenzen ist Aufgabe des Datenmonitors. Sobald eine Suche allerdings einen Leitungszustand, einen Fehler oder eine Änderung der Pegel beinhaltet, ist der Ereignismonitor das richtige Werkzeug. Zudem können Sie den Ereignismonitor auch dazu verwenden, bestimmte Bereiche einer Aufzeichnung als neue Aufzeichnung abzuspeichern.

Mit dem neuen Levelfinder sind nicht nur beliebige statische Pegelzustände als Suchmuster definierbar sondern auch Abfolgen von Ereignissen und Pegelwechsel.
Die einzelnen Suchparameter können wahlweise UND oder ODER verknüpft und zusätzlich mit einer Zeitspanne versehen werden um gezielt nach Ereignissen innerhalb eines definierten Zeitrahmens zu suchen oder auszuschliessen.
Im Abschnitt Ereignissuche werden die Suchoptionen ausführlich beschrieben.

13.1 User Interface

Das Fenster des Ereignismonitors liefert Ihnen zu jedem Zeitpunkt der Aufzeichnung einen schnellen Überblick der Pegelzustände. Von hier starten Sie die Suche nach bestimmten Ereignissequenzen, den Export beliebiger Ausschnitte oder vergleichen unterschiedliche Abschnitte miteinander.



13.1.1 Jede Zeile ein Ereignis

Der Ereignismonitor stellt alle aufgenommenen Ereignisse in Form einer Liste dar, wobei jede Zeile einem einzelnen Ereignis entspricht und die Änderung zum vorherigen Leitungszustand repräsentiert. Die Listenansicht der Ereignisse ist frei konfigurierbar. Bis auf den ersten Eintrag (der Ereignistyp) können sie jede Spalte ausblenden, in dem Sie Ihre Breite einfach per Maus auf Null setzen. (Im Ansicht Menü können sind die ausgeblendete Spalten jederzeit wieder aktivierbar).

Spalten ausblenden
Einfach Spaltenbreite zusammenziehen

Die Bezeichnung der Signalspalten passt sich automatisch den von Ihnen vergebenen Signalnamen an. Signalnamen ändern Sie global im Einstelldialog des Kontrollprogrammes.

13.1.1.1 Alle Ereignistypen auf einen Blick

Der Ereignismonitor unterscheidet zwischen folgenden Ereignissen:

Symbc	Ereignistyp	Bedeutung
A	Datenbyte	Datenbyte empfangen von Datenkanal A

B	Datenbyte	Datenbyte empfangen von Datenkanal B
	Pegelwechsel	Eine Änderung des Pegels an einem beliebigen Differenzsignaleingang. Dies schließt Pegelwechsel durch übertragene Datenbytes mit ein.
F	Framing	Empfangenes Datenbyte mit Framing Fehler
P	Parity	Empfangenes Datenbyte mit Parity Fehler
B	Break	Break erkannt

Die eingerückten Fehlersymbole treten nie einzeln sondern immer mit einem Datenbyte auf. Schließlich handelt es sich hierbei um Fehler bei der Datenübertragung.
Änderungen der Signalpegel durch die übertragenen Daten lösen ebenfalls Pegelereignisse aus, gefolgt von einem Datenereignis, sobald alle Datenbits vollständig übertragen wurden.

13.1.2 Signaländerungen

Ein Pegelwechsel Ereignis wird immer dann aufgezeichnet, wenn einer der 8 Signalpegel (Eingänge am Analyser) seinen Zustand verändert hat. Der MSB-RS485-PLUS Analyser unterscheidet 3 (tri-state) Pegelzustände: High, low und invalid (idle). In der Spalte 'Änderung' wird für jedes geänderte Signal der neue Signalpegel/zustand eingetragen. Dabei werden High Pegel mit einem Plus +, Low Pegel mit einem Minus − und ungültige oder Idle Pegel mit einem Ausrufezeichen ! gekennzeichnet. So bedeutet:

```
-CH1 +CH2 !CH3
```

Das CH1 Signal geht auf einen Low Pegel zurück, während gleichzeitig (im Rahmen der Zeitauflösung) das Signal an CH2 auf einen High Pegel und CH3 auf einen Idle/Invalid (oder auch ungetriebenen) Zustand wechselt.

```
-CH1 -CH2
```

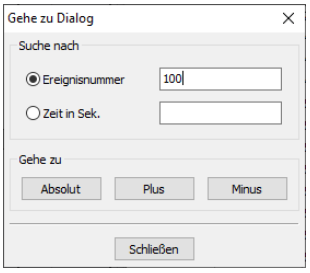
In diesem Beispiel wechseln beide Kanäle CH1 und CH2 zeitgleich auf einen Low Pegel.

Pegelwechsel der Datenleitungen

Sollten Sie keine Pegelwechsel auf den Differenzsignaleingängen CH1...CH4 sehen, dann müssen Sie diese im Kontrollprogramm noch aktivieren.

13.2 Durch Ereignisliste navigieren

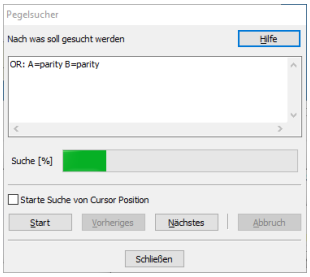
Der Ereignismonitor bietet neben den üblichen Scroll Möglichkeiten per Maus, Mausrad, Scrollbalken oder den Pfeil und 'Bild auf/ab' Tasten das gezielte Springen von einem Ereignis zum nächsten bzw. vorherigen Ereignis gleichen Typs.



Gehe zu Ereignis
absolut, schrittweise oder mit Zeitangabe

Klicken Sie dazu auf das gewünschte Ereignis und drücken Sie dann die Strg-Taste in Verbindung mit den Pfeil nach unten bzw. Pfeil nach oben Tasten. Auf diese Weise navigieren Sie z.B. einfach von Break zu Break oder von Datenbyte zu Datenbyte.
Bei längeren Aufzeichnungen können Sie auch gezielt Bereiche ab einem bestimmten Ereignis oder ab einer bestimmten Zeitmarke einblenden. Letztere erwartet die Eingabe eines Zeitoffsets in Sekunden seit Beginn der Aufzeichnung.
Die Angabe einer Ereignisnummer erlaubt Ihnen dagegen entweder absolut ein ganz bestimmtes Ereignis anzuspringen, oder aber in definierten Schritten zum aktuellen Ereignis durch die Aufzeichnung zu navigieren. Beispielsweise alle 1000 Ereignisse vor oder zurück.

13.3 Ereignissuche mit dem Levelfinder



Levelfinder
sucht Pegelwechsel, Fehler und Zeitabstände zwischen Ereignissen

Das Finden von ganz bestimmten Ereignissequenzen ist eine der herausragenden Eigenschaften des Ereignismonitors. Im Gegensatz zum Datenmonitor ist die Suche hier nicht auf bestimmte Datensequenzen beschränkt (für die der Datenmonitor die bessere Alternative ist), sondern vielmehr gezielt auf die physikalische Änderungen der einzelnen Leitungen abgestimmt.
Was heißt das genau?
Mit dem Ereignismonitor können Sie die Änderung und/oder den Zustand beliebiger Leitungen vorgeben, nach welchem Sie die bis dahin aufgenommene Ereignisaufzeichnung durchsuchen wollen. Dabei können Sie die Suche mit dem Auftreffen eines bestimmten Datenbytes bzw. einer Reihe gesetzter Bits in einem Datenbyte, oder einem asynchronen Ereignis (break, framing oder parity Fehler), verknüpfen. Klicken Sie auf das 'Lupen Symbol' in der Werkzeugleiste oder drücken Sie Strg-F um den Suchdialog zu öffnen.

13.3.1 Suchmuster eingeben

Das Suchmuster kann beliebig kompliziert sein. Der integrierte LevelFinder nimmt deshalb die Sucheingabe in Form logischer Ausdrücke entgegen, wobei jeder Ausdruck aus einer oder mehreren Bedingungen besteht, die AND (und) oder OR (oder) verknüpft sein können.

Verknüpfung: Bedingung1 Bedingung2 ... BedingungN

Bespielsweise:

AND: CH1=high BDIR=high

Die Formulierung der einzelnen Bedingungen entspricht dabei der intuitiven Fragestellung bei der Suche nach bestimmten Zuständen. Im vorangegangenen Beispiel: Suche Stelle der Aufzeichnung, an der der Differenzpegel an CH1 `high` UND das Bus-Richtungs-Signal BDIR ebenfalls `high` ist [1].
Jede Bedingung besteht aus einem Ziel (auf welches die Bedingung angewandt werden soll) und einer Zustandsbeschreibung. Also:

Ziel=Zustand

[1] Dieser Fall beschreibt einen Buskonflikt, da ein `high` Pegel des BDIR Signals eine aktive Datenübertragung an CH2 bedeutet und Busteilnehmer an CH1 nicht zur gleichen Zeit aktiv sein dürfen

Ziel ist entweder eine einzelne Signalleitung, angegeben durch ihren jeweiligen (auch Benutzer definierten) Namen, oder eine Datenquelle (Datenkanal) A bzw. B.
Zustand definiert den Zustand, den das Ziel bei der Suche einnehmen muß. Im Falle einer Signalleitung sind das die drei möglichen Pegelzustände on, off und invalid (oder die alternativen Bezeichnungen mark, space, high, low,). Ist das Ziel eine Datenquelle, sind mögliche Zustände ein exakter Datenwert, ein Bitmuster oder ein Fehler.

Korrektes Formulieren einer Bedingung

Bedingungen dürfen keine Leerzeichen enthalten. Beachten Sie außerdem, daß die Bezeichnungen 'A' und 'B' reserviert sind und deshalb nicht als Signalnamen verwendet werden dürfen.

13.3.1.1 Formulierung eines Pegelzustandes

Pegelzustände können für jede der vier Differenzsignaleingänge CH1...CH4, die beiden digitalen Hilfseingänge IO1, IO2 sowie den intern vom Analyser generierten Bussignalen BDIR und BSIG definiert werden. Nicht angegebene Signale werden bei der Suche einfach ignoriert.

Eingabe	Beschreibung
1, on, high, mark, -V	Signalpegel ist logisch eins, dies entspricht einer negativen Spannung von –0.7V...–7V am Differenzsignal-Eingang. Alle aufgeführten Zustandsbeschreibungen sind dabei absolut gleichwertig, d.h. CH1=on ist gleichbedeutend mit CH1=1 oder CH1=high.
0, off, low, space, +V	Signalpegel ist logisch null, was einem physikalischen Spannungspegel von +0.7V...+12V am Differenzsignal-Eingang entspricht. Alle aufgeführten Zustandsbeschreibungen sind dabei absolut gleichwertig, d.h. CH1=off ist gleichbedeutend mit CH1=0 oder CH1=low.
none, invalid, inactive, 0V	Signalpegel ist ungültig. Dies entspricht einem physikalischen Spannungspegel von -0.7V...+0.7V (gemäß der EIA-422/485 Definition liegt der Schwellwert der Empfängerbausteine bei $\pm$200mV, durch den höheren Pegel des MSB-RS485-PLUS Analysers werden aber auch durch Pullup bzw. Pulldown voreingesetllte Ruhepegel noch sicher als solche erkannt). Die Formulierung eines ungültigen Pegels erfolgt damit als: CH1=none, CH1=invalid, CH1=inactive oder CH1=0V.

13.3.1.2 Formulierung eines Datenfehlers

Datenfehler treten nur in Verbindung mit einem Datenkanal auf. Als Ziel muß deshalb A oder B eingesetzt werden.

Eingabe	Beschreibung
break, frame, parity	Datum in Datenkanal A oder B ist ein Break oder weist einen Frame bzw. Parity Fehler auf. Parity Fehler werden mit A=parity oder B=parity gefunden.

13.3.1.3 Formulierung eines Datenwerts

Jedes Datenbyte von Datenkanal A und B (d.h. die an dem zugehörigen Differenzsignaleingang aufgezeichneten Daten) kann wahlweise auf Gleichheit oder auf gesetzte Bits geprüft werden. Letzteres ist sinnvoll, wenn bestimmte Bitkombinationen in den Daten nicht vorkommen dürfen bzw. einen durch das Protokoll definierten Zweck erfüllen.
Beliebige Daten werden mit dem * Symbol gekennzeichnet.
Der LevelFinder stellt vier Arten von Eingabemöglichkeiten zur Verfügung.

Eingabe	Beschreibung
A=*, B=*	Jedes Datenereignis (A oder B) liefert einen Treffer, der Datenwert ist unerheblich. Sinnvoll, wenn nach einem beliebigen Datum gesucht wird.
A='x', B='x'	Prüft das Ziel (A oder B) auf Gleichheit mit dem in Hochkommas gesetzten Zeichen. Eine Suche nach einem durch Datenkanal A empfangenen Fragezeichen wird formuliert mit: A='?'.
A=$xx, B=$xx	Prüft das Ziel (A oder B) auf Gleichheit mit dem hexadezimal angegebenen Zeichen. Eine Suche nach einem durch Datenkanal A empfangenen Fragezeichen wird hier formuliert mit: A=$3f bzw. A=$3F
A=~xxxxxxxx, B=~xxxxxxxx	Prüft das Ziel (A oder B) auf die in xxxxxxxx angegebenen gesetzten Bits. Dabei wird das Bitmuster mit den Datum UND verknüpft und anschliessend auf Gleichheit geprüft. Um ein Zeichen mit einem gesetzten Bit 7 (gezählt von 0) an dem Datenkanal B zugeordneten Differenzeingang zu finden, geben Sie B=~10000000 ein.

13.3.2 Sucheingabe und Suche

Beispielprojekt
im Verzeichnis
examples/RS485-Analyzer

Bevor Sie beginnen, öffnen Sie im Kontrollprogramm das Beispielprojekt:
`Scan-for-signal-levels.msbprj`
im Verzeichnis `examples/RS485-Analyzer`.
Es handelt sich um eine 2-Draht Segmentanalyse bei der zusätzlich der Digitalausgang eines Modbus Teilnehmers mit Hilfe des zweiten Digital IO Anschlusses aufgenommen wurde. Die Aufzeichnung enthält eine Reihe von Datenfehlern sowie diverse Kombinationen von Pegelwechsel, nach denen wir im Anschluß suchen werden.

Suche nach einem Break in Datenkanal A

- Levelfinder Dialog öffnen per Strg + F

- Textfeld anklicken und » `AND: A=break` « eingeben
- Startknopf des Dialogs klicken oder ALT+S drücken
- Den Knopf Mehr klicken um das nächsten Break zu suchen
- Den Knopf Zurück klicken um zum letzten Treffer zurückzukehren

Der sichtbare Ausschnitt des Ereignismonitors wechselt bei jedem Treffer die Position und zeigt das gefundene Ereignis als schwarz hinterlegte Zeile an.

Suche nach einem Break in Datenkanal A oder B

- Textfeld anklicken und » `OR: A=break B=break` « eingeben

Suche Break in Datenkanal CH1 (Data A) mit CH2 idle

- Textfeld anklicken und » `AND: A=break CH2=none` « eingeben

Das war einfach. Verkomplizieren wir die Suche ein wenig und suchen nach:

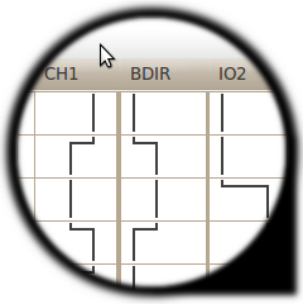
Suche nach inaktivem CH1 während CH2 low und BDIR high ist

- Textfeld anklicken und:
  » `AND: CH1=none CH2=low BDIR=high` «
  eingeben
- Den Knopf Mehr klicken um die nächsten Treffer zu finden

Sie können alle Pegelzustände, Daten und/oder Datenfehler beliebig kombinieren. Nicht möglich ist die Kombination verschiedener Verknüpfungsarten innerhalb eines Suchausdruckes. D.h. das Mischen von AND und OR Verknüpfungen. Wir werden jedoch sehen, daß AND und OR in aufeinander folgenden Ausdrücken durchaus erlaubt sind. Sequenzen von verschiedenen Suchausdrücken werden bei der Suche nach Signaländerungen verwendet. Z.B. die Suche nach einem Wechsel der Bus-Richtung BDIR bei gleichzeitig aktiver IO2 Leitung.

13.3.3 Suche nach Signaländerungen

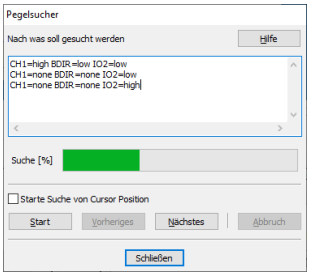
Änderungen werden beschrieben durch zwei (oder mehrere) aufeinander folgende Suchausdrücke, die jeweils den Leitungsstatus vor und nach dem Signalwechsel definieren. Insofern erweitern wir die bisherigen Sucheingaben um die Möglichkeit, mehrere in jeweils getrennten Zeilen einzugeben. Dabei ist es auch möglich, die logische Verknüpfung zu variieren.
Betrachten Sie dazu neben stehendes Bild. Von Interesse sind die Signale CH1, BDIR und IO2, wobei wir den in der idealisierten Zoomansicht dargestellte Signalwechsel suchen wollen.
Insgesamt handelt es sich um drei Zustände, die nacheinander eintreffen müssen.



Pegelwechsel gesucht?
Kein Problem mit dem integrierten LevelFinder.

1 CH1=high BDIR=low IO2=low
2 CH1=none BDIR=none IO2=low
3 CH1=none BDIR=none IO2=high

In jedem Zustand müssen alle drei Signalpegel erfüllt sein, d.h. für jeden einzelnen Suchausdruck gilt die AND (und) Verknüpfung.
Geben Sie jeden Ausdruck in einer eigenen Zeile im Textfeld des LevelFinders ein. Da die AND Verknüpfung die Voreinstellung ist, können Sie diese auch weglassen und die Zeilen exakt wie oben übernehmen. (Statt high und low können Sie auch 1 bzw 0 schreiben).



Der LevelFinder
Findet Datenbytes, Fehler und Pegelwechsel inkl. Zeitmessung.

Falsche Eingabe - was dann?

Sollten Sie bei der Eingabe einen syntaktischen Fehler machen, z.B. einen falschen Signalnamen oder ein ungültiges Zeichen, quittiert dies der LevelFinder mit einem leicht gelb hinterlegten Textfeld, sobald Sie die Suche starten.

Sie können die Suchbedingungen beliebig kombinieren. Z.B. Suchen nach einem bestimmten Datenbyte bei gleichzeitig aktiver IO2 Leitung, oder beliebiges Datenbyte von Datenkanal A gefolgt von einem Wechsel der Bus-Richtung BDIR. Der Anzahl von Suchausdrücken sind prinzipiell keine Grenzen gesetzt. Nicht desto trotz kostet jede einzelne Bedingung und jeder einzelne Ausdruck zusätzliche Rechenzeit und verlangsamt dadurch die Suche.
Der Suchmechanismus läuft parallel zur Anwendung und kann jederzeit durch Klicken auf den Abbruch Knopf beendet werden. Auch während einer evtl. länger dauernden Suche können Sie den Ereignismonitor normal bedienen.
Der LevelFinder speichert den aktuellen Suchausdruck automatisch. Dies gilt auch, wenn Sie den Ereignismonitor schliessen oder die komplette Sitzung beenden.

Suche ab bestimmter Position starten

Klicken Sie mit der Maus auf die Zeile, ab der die Suche beginnen soll und aktivieren Sie im LevelFinder *Starte Suche von Cursor Position*.

13.3.4 Suchen mit Zeitvorgaben

Pegeldauer?
Suche mit der Stopuhr

Als besonderes Feature bietet der LevelFinder eine integrierte Stopuhr, die in jedem Suchausdruck gestartet und in den folgenden Ausdrücken abgefragt werden kann. Damit ist es möglich, gezielt nach Pegelzuständen zu suchen, die nur eine bestimmte Zeit existieren. Beispielsweise eine aktive Bus-Richtung mit einer Dauer größer als 0.1s und kleiner als 0.3.

```
1 BDIR=none
2 BDIR=high watch.start
3 BDIR=none watch.time>0.1 watch.time<0.3
```

Beachten Sie, das alle Bedingungen innerhalb eines Ausdruckes per Voreinstellung AND (und) verknüpft sind. In Zeile 2 wird der Wechsel der Bus-Richtung von einem inaktiven Pegel `none` (definiert in Zeile 1) auf `high` definiert und gleichzeitig eine Stopuhr (watch) gestartet. (Genau genommen wird die Startzeit der Uhr mit der Zeit des aufgetretenen Ereignisses, dem Wechsel von BDIR auf `high`, initiiert.)

In der dritten Zeile wird nun der Wechsel des BDIR Signals zurück in den inaktiven Leitungszustand mit einer Zeitdauer größer als 0.2s und kleiner als 0.3s AND verknüpft. Zeile 3 enthält damit 3 Bedingungen, die alle erfüllt sein müssen, um einen gültigen Treffer zu liefern. Positive BDIR Signalflanken, die später als 0.3s erfolgen, werden dadurch ignoriert.

Anzeige der Treffer im Signalmonitor

Das Beispiel enthält exakt zwei Stellen, die obige Bedingungen erfüllen, sehr schön zu sehen in der Signal Darstellung. Öffnen Sie dazu einfach einen Signalmonitor und stellen Sie ihn auf 'Synchronisieren' mit anderen Views. Mit jedem Suchtreffer springt der Marker an die betreffende Signalposition.

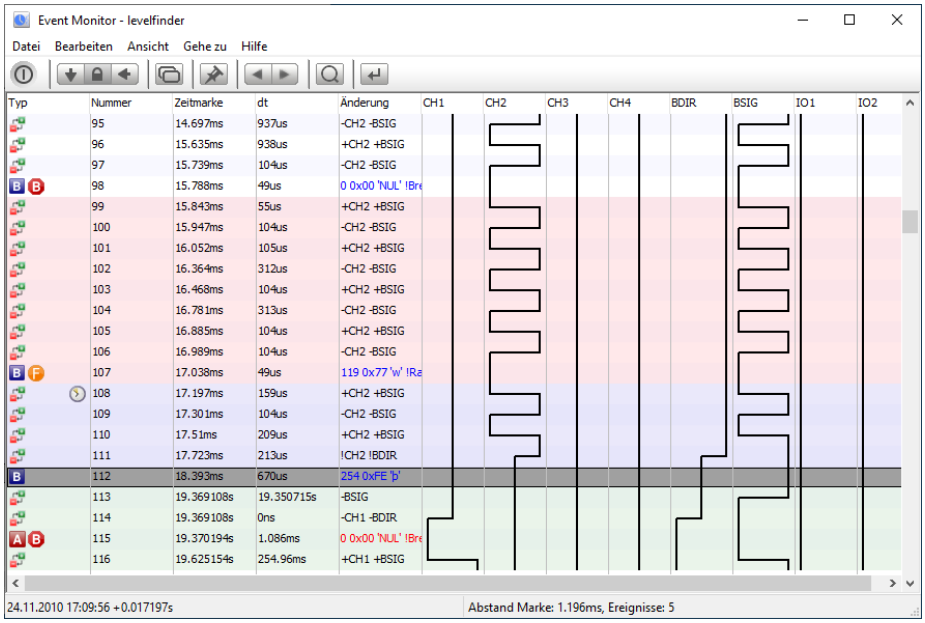
13.4 Eine Auswahl markieren und speichern

Bestimmte Funktionen des Ereignismonitors wie z.B. die Speicherung von Ereignissen als eigenständige Aufzeichnungsdatei, der Export im CSV Format zur späteren Bearbeitung durch externe Programme oder aber als Region beziehen sich immer auf eine zuvor definierte Auswahl.
Die Auswahl beliebiger Ereignissequenzen entspricht dem üblichen Vorgehen anderer Programm. Links-Klicken Sie die erste zur Auswahl gehörende Zeile in der Listendarstellung. Anschliessend scrollen Sie per Mausrad oder Scrollbalken durch die Aufzeichnung bis zum gewünschten letzten Ereignis der Auswahl. Dieses Links-Klicken Sie bei gleichzeitiger gedrückter Umschalt-Taste.
Um zum letzten Ereignis Ihrer Auswahl zu gelangen können Sie auch den LevelFinder oder Gehe-zu Dialog verwenden. Wichtig ist nur, daß Sie das letzte Ereignis mit gleichzeitig gedrückter Umschalt-Taste anklicken.
Beachten Sie, daß immer nur zusammenhängende Bereiche ausgewählt werden können, nicht beliebige einzelne Ereignisse.
Mit `Date→Speichere Auswahl unter...` können Sie eine Auswahl als eigene Aufzeichnungsdatei `*.msblog` sichern um diese später gezielt mit der Software zu analysieren. Auf diese Weise lassen sich gerade bei sehr großen Datenmengen die von eigentlichem Interesse befindlichen Abschnitte extrahieren.

13.4.1 Eine Auswahl als Region speichern

Eine Region dient zur Kennzeichnung bestimmter Abschnitte in der Aufzeichnung, die von besonderem Interesse sind. Im Gegensatz zu einer Auswahl gelten Regionen für alle aktiven Analysetools. Sobald eine Auswahl als Region hinzugefügt wird, ist diese für alle Analysefenster sichtbar.
Regionen werden im Ereignismonitor als verschieden farbige Bereiche dargestellt und bleiben unabhängig vom Ereignismonitor bestehen solange sie nicht explizit gelöscht werden.
Um eine Auswahl als Region zu speichern drücken Sie einfach die F4 Taste, oder klicken Sie unter `Bearbeiten→Auswahl in Region kopieren`. Es stehen maximal acht Regionen zur Verfügung. Unter `Ansicht→Regionen Dialog anzeigen` können Sie jederzeit vorhandene Regionen ein/ausge-

blenden oder wieder entfernen.



Regionen sind Teil der Aufzeichnung und werden als solche beim Speichern in der Aufzeichnungsdatei `*.msblog` mit gesichert.

13.4.2 Eine Auswahl als CSV Datei exportieren

Datenexport
Speichern als CSV für externe Bearbeitung

Der Ereignismonitor erlaubt Ihnen, eine beliebige Auswahl der aufgezeichneten Ereignisse als Komma getrennte Liste (CSV - Comma Separate Values) abzuspeichern. Sie werden von dieser Möglichkeit Gebrauch machen, wenn Sie Ereignisse in ein Tabellenkalkulations Programm wie z.B. Microsoft Excel, Gnumeric oder Open Office Calc importieren wollen.

An dieser Stelle werden Sie vielleicht die Frage einwerfen: Warum sollte ich die aufgezeichneten Ereignisse in Excel (oder andere Programme) importieren wollen?

Angenommen, Sie wollen die Ereignisse nach den größten Pausen zwischen zwei gesendeten Datenbytes sortieren. Oder aber, Sie wollen einfach eine Statistik der Ereignisse (oder Daten) erstellen. Anforderungen dieser Art sind die Domäne der Tabellenkalkulations Programme. Der Ereignismonitor gibt Ihnen die Möglichkeit, diese auch zu nutzen.

Markieren Sie zunächst eine Auswahl der von Ihnen zu exportierenden Ereignisse. Mit Strg + A können Sie auch alle bis dato aufgezeichneten Ereignisse auswählen.

Klicken Sie anschliessend im Menü Datei den Eintrag 'Exportiere als CVS'.

In dem sich öffnenden Exportdialog können Sie aus der linken Liste der verfügbaren Werte einen beliebigen Wert auswählen indem Sie ihn einfach anklicken und anschliessend mit dem 'Pfeil nach rechts' Knopf in die Liste der zu exportierenden Werte verschieben. Wiederholen Sie dies einfach mit allen von Ihnen benötigten Werten.

Um die Reihenfolge der exportierenden Werte zu ändern klicken Sie auf den zu verschiebenden Wert (in der Liste der exportierenden Werte) und bewegen ihn, mit dem 'Pfeil nach oben' bzw. 'Pfeil nach unten' Knopf in die gewünschte Richtung.
Entsprechend können Sie einen in der Exportliste markierten Eintrag mit dem 'Pfeil nach links' Knopf wieder entfernen, d.h. er wird wieder in die Auswahlliste zurück gelegt.
Abschliessend geben Sie einen Dateinamen ein, unter welchem die Exportdatei gespeichert werden soll und klicken den 'OK' Knopf um den Export zu starten.

Exportdialog

Die Liste der exportierbaren Ereignisse umfaßt folgende Werte:

Bezeichnung	Beschreibung
Nummer	Ereignisnummer, beginnend bei 0
Ereignistyp	Die folgenden Typen sind definiert: A=Daten an Port A, B=Daten an Port B, L=Leitungsänderung.
Zeit(s)	Zeitpunkt des Ereignisses als Offset zum Aufzeichnungsbeginn in Sekunden mit Mikrosekunden Genauigkeit.
Daten	Der Dezimalwert eines Datenereignisses (9bit) im Bereich (0...511).
Daten/Status	Enthält entweder das Datenbyte (9 Bit) oder den Tri-State Zustand jeder einzelnen Leitung ⇒1
Fehler	Datenübertragungsfehler, enthält im Fehlerfall ein B (Break), F (Frame) oder P (Parity).
dt gleichen Typs	Zeitdifferenz zum vorherigen Ereignis des gleichen Typs in Sekunden, z.B. 0.251518.
dt zuletzt	Zeitdifferenz in Sekunden zum letzten aufgetretenen Ereignis, d.h. zur letzten aufgezeichneten Änderung in der Leitung, z.B. 0.000217.
Datum Zeit	Absolutes Datum, Zeit und Mikrosekunden des Ereignisses, z.B. 2014-08-20 09:49:31+148762s.
Änderung	Ausgabe der Veränderungen verglichen zum letzten Ereignis, wie in der Splate Veränderung dargestellt..
Signalpegel	Ausgabe der Veränderungen/Zustände aller Leitungen wie im Ereignismonitor angezeigt. Die grafische Darstellung der Pegelwechsel wird dabei in einer entsprechende Textsymbolik dargestellt. ⇒2

*CH1,*CH2,*CH3,...	Exportiert den Tri-State Pegel des entsprechenden Signals als eine nummer -1, 0, +1. Das führende '*' dient zur Unterscheidung eines Signalnamens von anderen Feldnamen. `-1` entspricht -12V bzw. mark oder 'logisch 1' `0` ist ein ungültiger bzw. inaktiver Leitungszustand `+1` entspricht +12V bzw. space oder 'logisch 0' ⇒3

[1] Datenstatus

Der Inhalt dieses Feldes ist abhängig vom Datentyp. Handelt es sich um ein übertragenes Datenbyte enthät es in den unteren 9 Bit den Wert des aufgezeichneten Datenbytes. Die oberen 7 Bits sind 0.

Bit	Unused							Data Byte							Bit
15							8	7							0

Im Falle einer aufgezeichneten Leitungsänderung enthalten die oberen 8 Bit den Pegelzustand der jeweiligen Leitung, d.h. high oder low. Die unteren 8 Bit repräsentieren den Gültigkeitszustand der einzelnen Leitungen. Ist er '0', befindet sich die Leitung in einem inaktivem Zustand, dies entspricht einem Pegel zwischen -0.7V...+0.7V.

Bit 15	Line State						Bit 8	Bit 7	Valid State						Bit 0
CH1	CH2	CH3	CH4	BDIR	BSIG	IO1	IO2	CH1	CH2	CH3	CH4	BDIR	BSIG	IO1	IO2

[2] Textsymbolik der Pegelzustände

Gesendete Daten und Leitungsstatus sind zwei verschiedene Informationen und bestehen aus einer unterschiedlichen Anzahl von Feldern. Daten werden als Hexwert und mit entsprechender ASCII bzw. Kontrolbezeichnung dargestellt, während die Leitungen aus acht einzelnen Status bzw. Übergangssequenzen gelistet sind.
Um die für einen CSV Export gleiche Anzahl von Spalten zu erreichen, sind sowohl die Daten als auch die Zustände aller Leitungen innerhalb zweier "..." zusammengefasst. Der Zustand bzw. Zustandswechsel einer Leitung wird durch folgende Zeichen beschrieben.

- `^` : High Pegel
- `-` : Inaktiver Pegel
- `v` : Low Pegel

Eine Abfolge von `-v` beschreibt einen Wechsel von invalid zu low level, während ein Pegelwechsel von high auf low mit `^v` signalisiert wird. Nachfolgender Auszug soll dies verdeutlichen:

```
"Number","Type(Event)","Time(s)","SignalLevels"
0,L,17.359117,"-vCH1,-^CH2,-^CH3,-vCH4,-vBDIR,-^BSIG,-^IO1,--IO2",
1,L,18.408774,"vvCH1,^^CH2,^^CH3,vvCH4,vvBDIR,^vBSIG,^^IO1,--IO2",
2,L,18.408911,"vvCH1,^^CH2,^vCH3,vvCH4,vvBDIR,vvBSIG,^^IO1,--IO2",
3,L,18.409014,"vvCH1,^^CH2,v^CH3,vvCH4,vvBDIR,vvBSIG,^^IO1,--IO2",
```

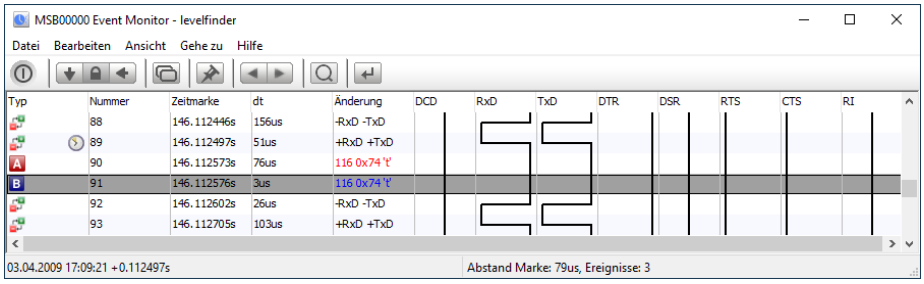
```
4,L,18.409118,"vvCH1,^^CH2,^vCH3,vvCH4,vvBDIR,vvBSIG,^^IO1,--IO2",
5,L,18.409845,"vvCH1,^^CH2,v^CH3,vvCH4,vvBDIR,vvBSIG,^^IO1,--IO2",
6,A,18.410003,"-vCH1,-^CH2,-^CH3,-vCH4,-vBDIR,-vBSIG,-^IO1,--IO2",
```

[3] Signalnamen bei Export

Beachten Sie, daß die Leitungen nicht unbedingt mit obigen Vorgaben bezeichnet sein müssen, da Sie jeder Leitung einen indviduellen Namen geben können, der dann an Stelle der hier verwendeten Vorgabe erscheint. Siehe auch Signalnamen im Kontrollprogram.

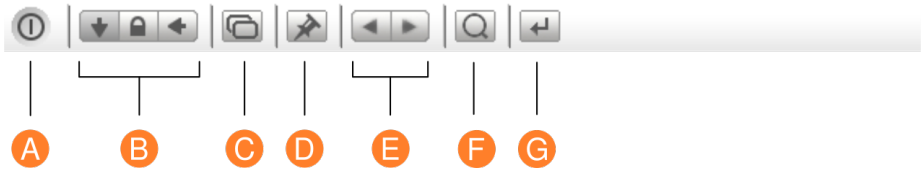
13.5 Zeitabstände messen

Jedes Ereignis kann per Rechtsklick als Startzeit markiert werden. Rechts neben dem Ereignistyp wird in diesem Fall ein Uhrensymbol eingeblendet und in der Statuszeile erscheint zu jeweiligen Cursorposition die Zeitdifferenz zum gegebenen Startereignis.
Ein nochmaliger Rechtsklick auf das Startereignis entfernt die Markierung wieder.



13.6 Die Werkzeugleiste

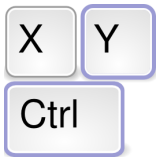
Die Werkzeugleiste dient zum schnellen Zugriff der am meisten benötigten Funktionen. Einige davon sind bei allen Views indentisch, andere spezifisch für den Ereignismonitor.



A	**Ende:** Speichert alle Einstellungen und schließt das Fenster
B	**Anzeigemodus:** Je nach Anzeigemodus zeigt der Fensterinhalt immer das aktuelle (zuletzt aufgenommene) Ereignis, ist verriegelt oder aktualisiert den Inhalt mit anderen Fenstern.
C	**Neue Ansicht:** Öffnet ein neues Fenster mit dem gleichen Ausschnitt und identischen Einstellungen.
D	**Default Einstellung:** Speichert die aktuellen Datenmonitor Einstellungen als Vorgabe wenn ein neues Datenmonitor Fenster geöffnet wird.

E	**Ereignis abhängiges blättern:** Springt zum vorherigen oder nächsten Ereignis des gleichen Typs wie an der aktuellen Cursor Position.
F	**Ereignis Suche** Öffnet den LevelFinder Dialog zur Ereignissuche.
G	**Gehe zu...** Öffnet den Gehe zu... Dialog um sichtbaren Ausschnitt per Ereignisnummer oder Zeitvorgabe zu wählen.

13.7 Kurzbefehle



Tastenkommandos der wichtigsten Funktionen

Aktion	Kurzbefehl
Online Hilfe zum Ereignismonitor	F1
Öffnet den Suchdialog des Levelfinders	Strg + F
Gehe zu... Dialog öffnen	Strg + G
Springe zu gesetztem Zeitmarker	Strg + T
Alle Einträge (Ereignisse) auswählen	Strg + A
Auswahl ausheben	Strg + Umschalt + A
Auswahl als Region speichern	F4
Springe zu vorherigem Ereignis gleichen Typs	Strg + Pfeil nach oben
Springe zu nächstem Ereignis gleichen Typs	Strg + Pfeil nach unten

14
Der Protokollmonitor

Mit der Analyse von Protokollen betreten Sie die nächste Ebene der Kommunikation. Die scheinbar willkürlich auftretenden Daten werden nach Ihren Regeln geordnet und gruppiert. Ausgabefunktionen erlauben Ihnen dabei, die einzelnen Datensequenzen individuell zu formatieren und farblich darzustellen.

Der Austausch von Daten zwischen zwei oder mehreren Kommunikationspartner erfolgt i.a. nach einem definiertem Protokoll, welches das Format der übertragenen Daten sowie deren Inhalt und Bedeutung festlegt. Die kleinste in sich geschlossene Dateneinheit wird dabei als Telegramm oder Datagramm Datagramm bezeichnet.
Während der Datenmonitor die übertragenen Daten in der Reihenfolge ihres Auftretens und ohne jegliche Interpretation ihres Inhaltes anzeigt (was je nach Untersuchung seine Vorteile besitzt), betritt man mit der Analyse von Protokollen und Datagrammen quasi die nächste Ebene der Kommunikation.

Der vom Analysator aufgenommene Datenstrom muss in einzelne Datensequenzen oder Datagramme unterteilt und diese dann angezeigt werden. Da es keine festen Regeln zur Definition eines Datagrammes gibt, kommen in der Praxis die unterschiedlichsten Realisierungen vor. Sie reichen von einem einfachen End of String (EOS) Zeichen, Start (STX) und Ende (ETX) Markierungen, bis zur Verwendung von zeitlichen Pausen (Modbus-RTU, Profibus), Lauflängenkodierung (RTL) und anderem.

Um die Interpretation oder Analyse der Kommunikation zu vereinfachen sollte jedes Telegramm die in ihm übermittelten Informationen möglichst lesbar darstellen. Dies gilt u.a. für die Adresse (Bus Teilnehmer), Funktions-Code, die Daten in den verschiedensten Formaten, Prüfsumme, Telegramm Begrenzer und anderes.

Es ist offensichtlich, das selbst eine breite Palette an vordefinierten Protokoll Templates nicht alle Anforderungen erfüllen kann. Besonders dann, wenn die Analyser Software mit individuellen Protokolldefinitionen oder Vorgaben konfrontiert wird. Der Protokollmonitor delegiert deshalb beides, die Auftrennung des Datenstromes in einzelne Telegramme sowie die Benutzer spezifische Darstellung der Telegramme an eine integrierte Skriptsprache.
Lua hat bereits im Datenmonitor seine herausragenden Fähigkeiten unter Beweis gestellt. So ist es nur konsequent, diese Skriptsprache auch in den Dienst

Modbus

PROFIBUS®

P-NET

...und mehr...

Lua

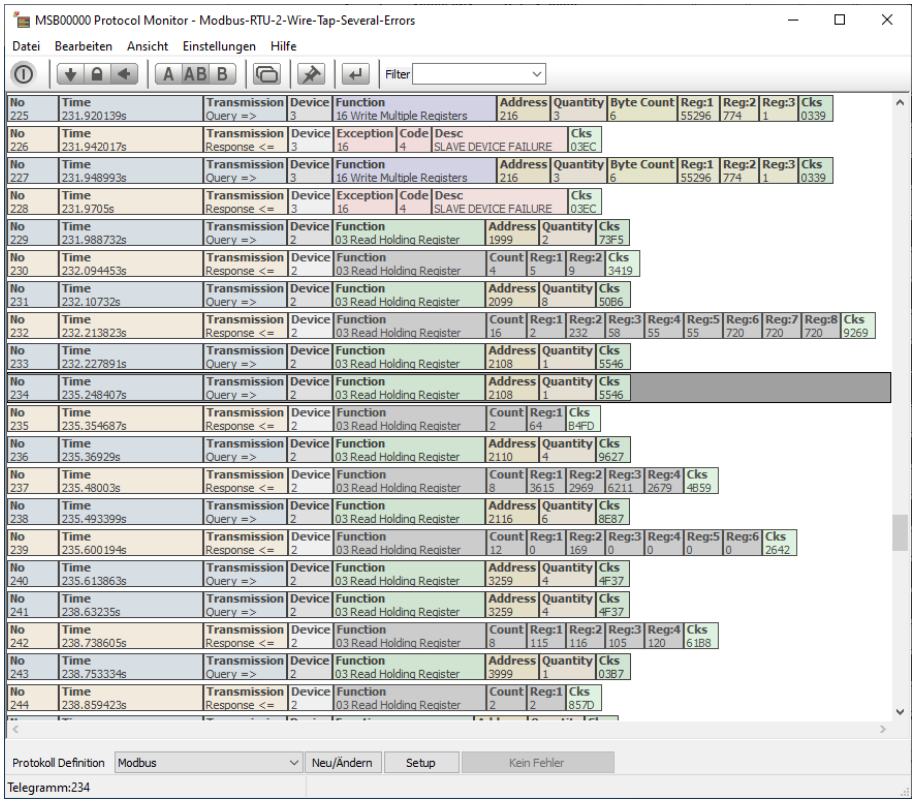
Lua Version 5.3

des Protokollmonitors zu stellen.

Lua bietet Ihnen die Möglichkeit, Protokoll Templates mit dem vollen Lua Sprachumfang zu realisieren. Sie können eigene Prüfsummenfunktionen schreiben, (neben dem bereits vorhandenen Checksum Modul, welches die gängigsten Algorithmen abdeckt), Funktionsnummern mit entsprechenden Namen ersetzen und Telegramme ausblenden die Sie nicht interessieren.
Und all dies ist interaktiv während einer laufenden Aufzeichnung möglich. Die Aufnahme selbst wird davon nicht beeinflusst. Ihre Modifikationen am Template werden direkt auf die Daten angewandt, das Resultat unmittelbar dargestellt.
Wir werden das Schreiben von Templates später im Detail diskutieren. Zunächst wenden wir uns dem Protokollmonitor in Aktion zu.

14.1 User Interface

Beim Protokollmonitor wurde bewusst auf unnötiges Beiwerk verzichtet und das Design auf optimales und intuitives Arbeiten mit Telegrammen ausgerichtet. Der Hauptteil des Programm Fensters dient deshalb der Darstellung der übertragenen Telegramme. Hier zu sehen am Beispiel einer Modbus RTU Übertragung.



14.1.1 Telegramm Fenster

Der Protokollmonitor stellt jede Sequenz (oder jedes Datagramm) in einer einzelnen Zeile dar. Wo ein Telegramm im Datenstrom beginnt oder endet wird

durch die entsprechend ausgewählte Protokoll Definition entschieden. Das gleiche gilt - wie bereits erwähnt - für die Darstellung des Telegramm Inhalts.

Jedem Telegramm kann optional eine der folgenden Informationen voran gestellt werden: Die Telegrammnummer, die Telegrammzeit (absolut und relativ zum Aufnahmestart), die Zeitlänge des Telegramms sowie der zeitliche Abstand zum vorherigen Telegramm. All dies kann jederzeit im Einstelldialog individuell geändert werden.

Datum/Zeitformat und Index

Sie können jedem Telegramm zusätzliche Informationen voranstellen. Die Auswahl erfolgt unter Einstellungen→Protokollmonitor einrichten und ist unabhängig vom verwendeten Template

Die vorangestellten Informationen geben gleichzeitig Auskunft über die Herkunft des Telegramms, d.h. ob ein Telegramm an Datenkanal A (roter Text) oder Datenkanal B (blauer Text) empfangen wurde. In einer typischen Bus Applikation finden Sie üblicherweise die Telegramme mehrerer Teilnehmer an Datenkanal A, andere empfangen an Datenkanal B. Da Sequenzen nicht immer abwechselnd auftreten, wird jede Sequenz getrennt nach der Datenrichtung nummeriert. Unvollständige Sequenzen, d.h. Datagramme, deren Endebedingung noch nicht erfüllt ist, z.B. weil noch kein Endezeichen empfangen wurde, werden mit einer Interpunktion ... hinter der Telegramm Nummer gekennzeichnet.
Beachten Sie dass diese nur bei Anzeige der Telegramm Nummer angezeigt wird.

14.1.2 Synchronisierung


Anzeige Modi

Allen MultiView Programmen ist gemeinsam, dass Sie die Ansicht synchronisieren, sperren oder per Autoscroll die jeweils zuletzt aufgenommenen Daten anzeigen lassen können. Dies gilt auch für den Protokollmonitor. Linksklicken Sie einfach das gewünschte Datagramm, um deren Ansicht bei anderen Views einzublenden. Das aktuelle Datagramm wird dabei hervorgehoben.
Entsprechend reagiert die Protokollansicht auf eine Synchronisierung durch andere Views. Die Sequenz, die Teil der Synchronisierung ist, wird mit der aktuellen Zeilenauswahl versehen.

14.1.3 Datenrichtung

Datenquellen

IFTOOLS Analyser besitzen zwei unabhängige Datenkanäle welche jeweils einer Quelle oder Richtung der übertragenen Daten zugewiesen werden. Bei Voll-Duplex Bus Systemen wie RS422 (aber auch RS232) ist die Bedeutung hierfür klar. Andere Feldbusse verwenden oft nur eine gemeinsame Verbindung für alle Bus Teilnehmer (Halb-Duplex).
Die Analyser sind in der Lage auch hier eine Zuordnung der Datenrichtung auf physikalischer Ebene vorzunehmen. Je nach Protokoll ist dies aber nicht immer nötig. Nicht desto trotz können Sie im Protokollmonitor jederzeit eine Auswahl der Telegramm nach Datenrichtung A, B oder A+B vornehmen.


Fenster kopieren

14.1.4 Aktuelle Ansicht in neuem Fenster öffnen

Stellen Sie sich vor, Sie möchten ein oder mehrere aufeinander folgende Telegramme mit denen an einer völlig anderen Position in der Aufzeichnung vergleichen.

Offensichtlich können Sie dies nicht in einem Protokollmonitor Fenster tun. Sie können aber die aktuelle Ansicht in einem neuen Protokollmonitor öffnen indem Sie den 'Clone' Icon in der Werkzeugleiste klicken.

Anschließend haben Sie zwei Fenster und können nur unterschiedliche Regionen der Aufzeichnung gegenüberstellen.

14.1.5 Aktuelle Fenstereinstellungen als Vorgabe verwenden

Die Analyser Software speichert Größe, Position und individuelle Fenstereinstellungen automatisch per Vorgabe wenn Sie die Anwendung beenden.

Sie können allerdings auch die Einstellungen mit denen ein neuer Protokollmonitor gestartet wird festlegen. Dies beinhaltet die ausgewählte Protokoll Definition, Schriftart, Datenrichtung, Synchronisation usw.

Indem Sie das Pin Icon in der Werkzeugleiste klicken, werden Ihre aktuellen Einstellungen als Vorgabe für alle weiteren Protokollmonitor Fenster gespeichert und bei Start eines neuen Protokollmonitors automatisch angewendet.

14.1.6 Zu einer Telegramm Nummer springen

Der Protokollmonitor nummeriert alle Telegramme in der Reihenfolge ihres Auftretens. Sie können ein bestimmtes Telegramm, z.B. mit der Nummer 10204 gezielt anspringen indem Sie das 'Gehe zu' Icon in der Werkzeugleiste klicken oder STRG + G drücken. In dem Dialogfenster geben Sie die gewünschte Nummer (oder Telegramm Position) ein. Der Protokollmonitor zentriert seine Anzeige danach um das gewünschte Telegramm.

Negative Eingaben erzwingen einen Sprung an die Startposition. Eingaben größer als die Anzahl vorhandener Telegramme lassen die Anzeige zum zuletzt aufgenommenen Telegramm springen.

Beachten Sie, das der Anzeigemodus nicht auf automatischem Scrollen steht, da sonst sofort wieder das zuletzt aufgenommenen Telegramm eingeblendet wird!

14.1.7 Filter Eingabe

Die Filtereingabe in der Werkzeugleiste ist ziemlich speziell und wird im Abschnitt 14.4 genauer erklärt. An dieser Stelle sei soviel gesagt:

Eine Filterung der angezeigten Telegramme hängt hauptsächlich von dem verwendeten Protokoll ab. Sie können z.B. nur nach Kommando Funktionen filtern (Modbus), wenn die Protokoll Spezifikation ein solches Telegrammfeld vorsieht. Die entsprechende Filter Funktion ist deshalb Teil des Protokoll Template. Nicht alle Templates bieten eine Filterung an. Aber wenn dies der Fall ist, können Sie die Filtereingabe zur Eingabe bestimmter Filter Kriterien nutzen.

Im Falle unseres Modbus Beispiels können Sie damit Diagnostik Telegramme ausblenden, oder z.B. nur die Kommunikation zwischen Master und einem bestimmten Teilnehmer (Slave Adresse) anzeigen lassen.

14.1.8 Bereich auswählen

Mit der Export oder Copy And Paste Funktion können Sie einen beliebigen Ausschnitt des Protokolls in anderen Anwendungen weiter verarbeiten. Dazu

müssen Sie lediglich den gewünschten Bereich markieren.
Die Auswahl erfolgt analog der Dateiauswahl in Ihrem Betriebssystem. Platzieren Sie den Cursor per Linksklick auf der ersten Zeile des auszuwählenden Abschnittes. Anschließend verschieben Sie den sichtbaren Ausschnitt bis zum Ende des gewünschten Bereichs und markieren das Ende der Auswahl mit einem Linksklick bei gleichzeitig gedrückter Umschalt Taste. Der Bereich wird daraufhin grau hinterlegt.
Alle Zeilen markiert die Tastenkombination Strg+A.

14.2 Protokoll Vorlagen

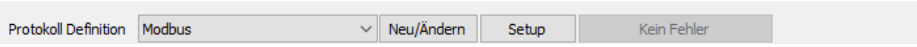
Die Analyser Software enthält bereits viele Templates (Protokoll Vorlagen) für die meist verwendeten Feldbus Protokolle. Weitere kommen mit neuen Software Versionen hinzu. Die aktuell unterstützten Protokolle sind:

- 3964(R)
- BACnet
- DF1
- DNP3
- Executive (Vending machines)
- IEC60870-5-101
- IEC60870-5-103
- MDB/IPC
- Modbus ASCII & RTU
- MOVILINK
- NMEA
- P-Net
- Profibus
- SAE-J1587
- SAE-J1922
- SMA-NET
- SSI (synchron, PLUS Analyser)
- USS

Ergänzt wird diese Liste um zusätzliche - mehr generische - Templates. Dabei handelt es sich um Vorlagen für Protokolle mit 9-Bit Werten, mit bestimmten Start und/oder Ende Zeichen oder einem Break (LIN Bus) als Telegrammstart. Jedes der verfügbaren Protokoll Templates kann als Vorlage für eigene Protokolle verwendet werden.

14.2.1 Ein Protokoll Template auswählen

Sie können jederzeit eine anderes Protokoll auf die aufgenommenen Daten anwenden indem sie den Auswahlknopf unter dem Telegramm Fenster klicken. Die sich öffnende Liste enthält alle verfügbaren Templates (auch die von Ihnen hinzugefügten). Die Anpassung der Protokoll Darstellung erfolgt dabei unmittelbar nach Auswahl eines Listeneintrages.



Ganz wichtig! Egal welches Protokoll Sie anwenden, die eigentliche Aufzeichnung bleibt immer davon unbeeinflusst!
Damit können Sie verschiedenste Protokoll Spezifikationen einfach ausprobieren,indem Sie diese nacheinander aus der Liste auswählen. Und das vor allem ohne die Aufzeichnung beenden oder neu starten zu müssen. Ein ungemeiner Vorteil bei der Analyse von unbekannten oder proprietären Protokollen sehr hilfreich.
Sollte keines der vorhandenen Templates Ihrem Protokoll entsprechen, bietet sich entweder die Modifikation eines vorhandenen oder das Schreiben eines komplett neuen Templates an. In beiden Fällen steht Ihnen der IFTOOLS Support mit Rat und Tat zur Seite.

14.2.2 Ein Protokoll Template editieren

Diese Fähigkeit des Protokollmonitors, Sequenzen mittels der Skriptsprache Lua zu definieren geht weit über eine fest vorgegebene Auswahlliste hinaus. Sie erfordert allerdings auch einen gewissen Lernprozess bezüglich der Sprachsyntax und ist bei trivialen Problem manchmal hinderlich.
Die vorhandenen Templates bieten deshalb einen guten Start für eigene Entwürfe. Neue Templates werden automatisch beim Speichern zu der Liste der verfügbaren Templates hinzugefügt und können dann genauso einfach ausgewählt werden wie die bereits vorhandenen.

Um ein Template zu modifizieren, klicken Sie den Knopf Neu/Ändern auf der rechten Seite der Protokoll Auswahl und das aktuelle Template wird im Editor geöffnet.
Ab Version 5.0 ist der Editor nicht länger Bestandteil des Protokollmonitors sondern ein eigenständiges Programm, dass von allen Analyser Views zum Schreiben und Testen von Lua Skripten verwendet wird.
Der Editor enthält alle Features die Sie von einem wirklich guten Editor gewöhnt sind. Darüber hinaus bietet er aber die Möglichkeit, eigenen Lua Code unabhängig vom Protokollmonitor (oder anderen Views in einem Sketch Buffer auszuführen. Dies können ausgewählte Zeilen sein, oder komplette Skripte.
Und: Vordefinierte Code Gerüste erleichtern Ihnen das Schreiben neuer Templates oder genereller Lua Module. Der Editor wird ausführlich in einem eigenen Kapitel 17 beschrieben.

Anfänglich werden Sie vermutlich nur kleine Dinge in einem Template ändern. Die Farbe einer Telegramm Struktur, die Definition eines Ende Zeichens oder die Idle (Pause) Zeit zwischen Telegrammen (wie bei Modbus RTU).
Sobald Sie Ihre Änderungen im Editor speichern werden diese automatisch vom Protokollmonitor angewendet und im Telegramm Fenster sichtbar (auch

bei einer aktiven d.h. laufenden Aufzeichnung).
Die Anzeige etwaige Code Fehler erfolgt dabei in der Statuszeile des Protokollmonitors.

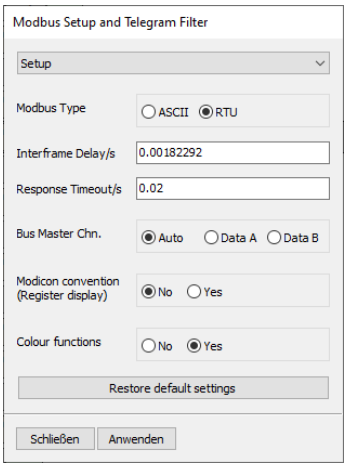
Template anwenden

Beachten Sie, das eine Änderung der Split oder 'Aufteilungs-Regel' eine komplette Neuinterpretation der aufgenommenen Daten erfordert und in Abhängigkeit von der Größe der Aufzeichnung etwas länger dauern kann. Eine Veränderung der eigentlichen Telegramm Darstellung wirkt sich dagegen nur auf die gerade angezeigten Telegramme aus und ist unmittelbar sichtbar.

Sie können ein vorhandenes Template auch jederzeit kopieren indem Sie es im Editor öffnen und unter einem anderen Namen abspeichern.

14.2.3 Individueller Protokoll Setup

Die überwiegende Mehrzahl der existierenden Feldbus Protokolle ist parametrisierbar. Zum Beispiel ist bei einer IEC60601-5-101 Anwendung die Anzahl der Adressbytes wählbar zwischen einem und zwei Bytes. Ein anderes Beispiel ist Modbus RTU und die sogenannte Interframe Idle Zeit (die Zeit, die zwischen zwei aufeinander folgenden Telegrammen pausiert werden muss). Diese Zeit wird in Modbus Anwendungen oftmals sehr individuell ausgelegt.
Beide genannten Beispiele wirken sich natürlich direkt auf das Parsen des Datenstroms aus. Die Protokollmonitor Template API bietet Ihnen deshalb die Möglichkeit, eigene ganz Protokoll spezifische Setup Dialoge zu schreiben. Mit diesen können Sie ein gewähltes Protokoll dann individuell anpassen ohne den Template Code in irgendeiner Form verändern zu müssen. Ein existierender Dialog vorausgesetzt, können Sie diesen jederzeit per Klick auf den Setup Knopf öffnen. Das Bild auf der rechten Seite zeigt als Beispiel den Setup Dialog des Modbus Templates.
Ein Setup Dialog muss nicht zwingend vorhanden sein. Falls nicht, bekommen Sie einen entsprechenden Hinweis, wenn Sie den Knopf klicken. Das Template Dialog Feature beschreiben wir in allen Details im Kapitel 20.



Ein Setup Dialog hier für Modbus

14.2.4 Eigene Templates definieren

Der neue Editor ist Dreh- und Angelpunkt für alle Lua Skripte in der Analyser Software. Er ist auch zuständig, wenn Sie ein neues Protokoll Template anlegen möchten (deshalb der Knopf Neu/Ändern).
Im Editor klicken Sie dazu auf das 'Neue Datei anlegen' Symbol in der Werkzeugleiste oder drücken Sie STRG + N.
Um Ihnen den Einstieg in eigene Skripte zu erleichtern legt der Editor je nach Anwendung ein entsprechendes Code Gerüst an. Bei der Auswahl des Skript Typs gegen Sie deshalb in dem sich öffnenden Skript Ersteller Dialog 'ProtocolView' vor (näheres hierzu finden Sie im Editor Kapitel 17).

14.2.5 Speicherort der Protokoll Templates

Der Speicherort der Protokoll Vorlagen (sowie aller anderer Lua Skripte) hat sich mit Version 5.0 geändert.

Linux Anwender finden Sie nun unter:

```
~/.IFTOOLS/SerialAnalyzer/7.0.2/Templates/ProtocolView
```

Für Windows Anwender ist der Speicherort:

```
C:\Users\USERNAME\AppData\Roaming\IFTOOLS\SerialAnalyzer\7.0.2\Templates\ProtocolView
```

Dies ist aber nur von informeller Bedeutung. Wenn Sie ein neues Protokoll Template anlegen oder ein vorhandenes unter einem anderen Namen speichern sorgt der Editor dafür, das diese immer im richtigen Verzeichnis abgelegt werden. Und das mit gutem Grund:
Der Protokollmonitor prüft dieses Verzeichnis nach neuen Skripten sobald Sie den Protokoll Template Auswahlknopf klicken. Alle gefundenen Templates werden dabei alphabetisch in der Auswahlliste angezeigt.
Sie können - natürlich - Templates auch unter einem anderen Ort speichern. Z.B. um Sie auf einen anderen Datenträger zu kopieren oder anderweitig zu verwenden.
Die Analyser Software allerdings berücksichtigt nur Skripte in dem vorgegebenen Ort!

14.2.6 Ein Template importieren

Manchmal möchten Sie vielleicht ein Template aus einer anderen Quelle verwenden. Z.B. ein Template von einem Kollegen oder aus dem Internet (der IFTOOLS Download Seite). Wie bereits erwähnt, müssen Sie das Template im richtigen Verzeichnis speichern, damit der Protokollmonitor es findet.
Importieren oder Hinzufügen eines neuen Templates ist einfach. Ziehen Sie dazu einfach das gewünschte Template (Dateiendung msbtml) in das offene Telegramm Fenster des Protokollmonitors (drag and drop). Das ist schon alles! Das Programm speichert automatisch das Template im entsprechenden Verzeichnis und wendet es auf die aktuelle Aufzeichnung an.
Im falle eines Fehlers bekommen Sie eine entsprechende Meldung im Fehlerknopf der Statuszeile. Das Programm warnt Sie zudem, wenn ein Template gleichen Namens bereits existiert.

14.3 Template Sprachsyntax

Jedes Protokoll Template (Datei bzw. Skript) muss mindestens zwei Funktionen bereitstellen. Die Erste extrahiert aus den eintreffenden oder aufgenommenen Daten die einzelnen Datagramme bzw. Telegramme. Diese enthält den Code der nötig ist, um zu entscheiden wann ein Telegramm beginnt und wann es endet.
Die zweite Funktion kontrolliert das Erscheinungsbild jedes Telegramms in einem extrem großen Umfang. Hier spezifizieren Sie wie der Telegramminhalt (oder Teile davon) dargestellt werden. Zum Beispiel: Sie können eine Bytefolge in andere Zahlenformate umwandeln, die Prüfsumme validieren oder einzelne Bereiche mit einem Bezeichner versehen. Und Sie können beliebige Abschnitte unterschiedlich einfärben.

Neben diesen beiden Funktionen existiert noch eine dritte, optionale Filterfunktion. Sie erlaubt Ihnen in Verbindung mit dem Filterelement in der Toolbar bestimmte Telegramm auszublenden.
Zunächst werden wir uns aber auf die zwei essentiellen Dinge konzentrieren, die ein Template leisten muss:

1. **Aufsplitten des Datenstroms in einzelne Telegramme**
2. **Individuelle Anzeige der Telegramme**

14.3.1 Aufsplitten des Datenstroms in einzelne Telegramme

Bei der Definition eines Protokoll Templates muss die erste Frage lauten:
Wann startet ein Telegramm und wann endet es?
Manchmal reicht eine Ende-Bedingung aus, z.B. bei einem Carriage Return und/oder einem Linefeed, oder wenn ein Wechsel in der Datenrichtung auftrat. Aber oftmals ist die reale Welt komplizierter. Denken Sie an binäre Protokolle mit definierten Zeitpausen zwischen den einzelnen Telegrammen wie Modbus RTU, Profibus oder ähnliche.

Der Protokollmonitor behandelt die komplette Auftrennung in der Funktion `split` wie im folgenden gezeigt.

```
1 function split(data, interval, alternation, string, filter)
2     -- here are your split instructions and their return states
3     return STATE
4 end
```

Diese Funktion wird jedesmal aufgerufen wenn ein neues Datenbyte aufgenommen oder aus einer vorhandenen Aufzeichnung gelesen wird und muss einen der folgenden Zustände zurückgeben:

1. STARTED → Ein neues Telegramm beginnt
2. MODIFIED → Das neue Byte ändert nichts außer die Telegrammlänge
3. COMPLETED → Das Telegram ist komplett
4. REMOVED → Entfernt das aktuelle Telegramm zu Filterzwecken
5. MARKED → Vormerken des aktuellen Bytes für späteren Telegrammstart

REMOVED überflüssig?

Das Filtern von Telegrammen durch Rückgabe von REMOVED wird nicht länger empfohlen, da es einige Nachteile ausweist. Dies betrifft vor allem bei komplizierteren Filter Szenarien. Ein bessere und deutlich einfachere Lösung wird im Abschnitt 14.5 beschrieben!

Es wird schnell klar, das die Auswertung nur des aktuellen Datenbytes nicht ausreicht um einen gültigen Start oder das Ende eines Telegramms zu erkennen. Denken Sie an ein EOS (End Of String) bestehend aus mehr als einem Zeichen. Oder: Ein Telegramm spezifiziert durch ein bestimmtes Start- UND Endezeichen.
Dies ist der Grund, warum die `split` mit zusätzlichen Parametern aufgerufen wird. Diese sind:

1. data → Das aktuelle Datenbyte (bis zu 9 Bits)

2 interval → (kurz intval), die Zeitdifferenz zum letzten Byte in Sekunden (mit Mikrosekunden Auflösung)
3 alternation → (kurz alter), wahr wenn sich die Datenrichtung geändert hat
4 string → (kurz str), alle bisher im aktuellen Telegramm empfangenen Bytes als String
5 filter → Die aktuelle Auswahl in der Toolbar Filtereingabe

Sie können die einzelnen Parameter umbenennen sofern eine andere Bezeichnung für Sie mehr Sinn ergibt. Sie dürfen allerdings nicht die Reihenfolge ändern. Erlaubt ist, unnötige Parameter vom der rechten Seite ab wegfallen zu lassen ohne das dabei Lücken entstehen.

Kompliziert? Sehen wir uns einige Beispiele an. Stellen Sie sich ein einfaches Protokoll vor, in welchem jedes Telegramm durch ein Linefeed abgeschlossen wird.

```
1  function split(data, intval, alter, str)
2      if #str == 1 then return STARTED end
3      if data == 10 then return COMPLETED end
4      return MODIFIED
5  end
```

Der Filter-Parameter wird nicht benötigt und wir können ihn deshalb weglassen (Zeile 1).
Unser Beispiel verwendet kein spezielles Startkriterium. Mit anderen Worten: Ein neues Telegramm beginnt sobald das vorherige mit einem Linefeed beendet wurde und ein beliebiges anderes Zeichen empfangen wird.
Der Parameter `str` ist ein Lua String und enthält alle Zeichen des aktuellen (wenn auch unvollständigen) Telegramms. Im Falle des ersten Bytes ist die Länge des Telegramms 1 und wir geben STARTED zurück. Lua verwendet einen speziellen Längen-Operator '#' um die Anzahl von Zeichen innerhalb eines String (einer Zeichenkette) zu ermitteln (Zeile 2). Alternativ können Sie die Abfrage der Länge auch wie folgt formulieren:

```
if str:len() == 1 then ...
```

Zeile 3 definiert die Ende Bedingung. Das Telegramm ist komplett, wenn ein Linefeed empfangen wird (das aktuelle Zeichen wird in `data` übergeben). Wir vergleichen deshalb den Parameter `data` mit Linefeed (der Zeichenwert ist dezimal 10) und liefern COMPLETED zurück, wenn dies zutrifft.
In allen anderen Fällen (wenn es sich um ein beliebiges anderes Zeichen handelt und bereits mindestens ein Zeichen empfangen wurde, d.h. die aktuelle Telegrammlänge ist größer als 1) kehrt die Funktion mit MODIFIED zurück.

Im nächsten Schritt passen wir das Beispiel an ein Protokoll an, welches für jedes Telegramm ein Startzeichen sowie ein Endestring spezifiziert, wie es u.a. bei Modbus ASCII üblich ist.

STX	Data	EOS
':'	ASCII coded data as 0-9 and A-F	CR LF

Jedes Telegramm wird mit einem Doppelpunkt ':' eingeleitet (der ASCII Wert ist dezimal 58), gefolgt von den eigentlichen Daten, wobei das Datenfeld nur die

Zeichen 0-9 und A-F erlaubt). Das Ende des Telegramms wird mit einem Carriage Return und Linefeed (CRLF) markiert. Die entsprechende `split` Funktion ist dann:

```
1  function split(data, intval, alter, str)
2      if data == 58 then return STARTED end
3      if str:find("\r\n") then return COMPLETED end
4      return MODIFIED
5  end
```

Zeile 2 vergleicht alle neuen Datenbytes mit dem Wert des ':' und liefert STARTED sobald ein Doppelpunkt erkannt wurde.
Zeile 3 durchsucht die bis dato im Telegramm befindlichen Zeichen nach der Zeichenkette CRLF (`"\r\n"` ist das Lua Äquivalent zu CRLF). Ein gefundenes CRLF bedeutet das Telegramm ist vollständig und liefert den Rückgabewert COMPLETED.
Alle anderen Zeichen werden als Telegrammdaten behandelt und modifizieren lediglich das Telegramm (Resultat ist MODIFIED).

Soweit zur Verwendung der Parameter `data` und `str`. Was aber ist mit den übrigen Funktionsargumenten `intval` und `alter`?
Stellen Sie sich vor, Sie haben ein Protokoll mit abwechselnd gesendeten Telegrammen. Jeder Telegrammstart (und Ende) ist dadurch gekennzeichnet, das sich die Datenrichtung geändert hat. Hier ist die passende `split` Funktion:

```
1  function split(data, intval, alter, str)
2      if alter then return STARTED end
3      return MODIFIED
4  end
```

Der Parameter `alter` ist immer wahr (true) wenn die Quelle des aktuellen Datenbytes (die Datenrichtung) gewechselt hat. Alles was wir tun müssen ist in diesem Fall STARTED zurückzugeben.

Einige Protokolle geben eine definierte Pause (spezifiziert als inaktive Zeit, in der keine Daten übertragen werden) als Telegramm-Begrenzer vor, z.B. das Modbus RTU Protokoll. Der Vorteil eines solchen Ansatzes ist: Ein Telegramm benötigt keine 'speziellen' Start- und/oder Endezeichen und kann deshalb alle binäre Zeichen zur Übertragung verwenden. (Es gibt keine Datenbytes die gesondert interpretiert werden müssen.

Telegramme definiert durch Sendepausen

Modbus RTU spezifiziert eine Sendepause von 3.5 Byte. Dies ist die Zeit, die zur Sendung von 3.5 Bytes mit der verwendeten Baudrate benötigt wird.
Und hier kommt der letzte Parameter `intval` ins Spiel. `intval` ist die Zeitdistanz zwischen dem aktuell und zuletzt empfangenen Byte in Sekunden. Die Auflösung ist - wie immer - 1μs.
Die Übertragungszeit für ein Byte oder Zeichen hängt von der eingestellten Baudrate ab. Glücklicher Weise bietet der Protokollmonitor entsprechende Funktionen zur Umrechnung. Aber zunächst der `split` Funktionscode:

```
1  function split(data, intval, alter, str)
2      if intval > transmission.bytepause( 3.5 ) then return STARTED end
3      return MODIFIED
```

```
4 end
```

Der Code sollte klar genug sein - mit Ausnahme von `transmission.bytepause`. Der Protokollmonitor erweitert den Lua Sprachstandard um eigene Funktionsmodule. `bytepause` liefert die benötigte Zeit zur Sendung der übergebenen Anzahl von Datenbytes für die aktuelle Baudrate in Sekunden. Damit reicht es die verstrichene Zeit seit dem letzten empfangenen Zeichen mit der berechneten Pause zu vergleichen um eine gültige Startbedingung zu erkennen.
Sie finden eine detaillierte Beschreibung des `transmission` Moduls auf Seite 273.

Aber halt! Was ist mit dem COMPLETED Status?
Das aktuelle Telegramm ist komplett (COMPLETED) sobald eine entsprechende Pause erkannt wurde. Gleichzeitig wird ein neues Telegramm eingeleitet (STARTED). In diesem Fall kennzeichnet der Protokollmonitor die bisherigen Telegrammdaten automatisch als fertig (COMPLETED) sobald er ein Neues startet.
Unvollständige Telegramme werden als eine Reihe von Punkten (Interpunktion) im vorangestellten Index oder Nummer Feld angezeigt. Sie können diese im Einstelldialog ein- oder ausblenden.

Spezialfall: Telegramme mit nur einem Byte

Telegramme bestehend aus nur einem einzelnen Zeichen sind insofern ein spezieller Fall, da sie beide Zustände repräsentieren: Ein STARTED wie auch ein COMPLETED Status. Ausnahmen hiervon sind lediglich Protokolle mit einer Sendepause als Telegramm-Begrenzung wie Modbus RTU oder ProfiBus[1], da dort die Telegramm-Begrenzer unabhängig von einem bestimmten Zeichen sind.
Alle anderen Protokolle mit einem definierten EOS (End of String) müssen der besonderen Bedeutung solcher Einzelbyte Nachricht Rechnung tragen. Ein Beispiel:
In Ihrem Protokoll wird jedes Telegramm mit einem Linefeed abgeschlossen. Daneben sind aber auch einzelne Linefeeds als Kurzbestätigung (Acknowledge) erlaubt.
Ohne die besondere Berücksichtigung einzelner Linefeeds wird der Protokollmonitor das erste auftretende Linefeed (LF) solange als unvollständiges Telegramm werten bis ein weiteres LF empfangen wird. Hier die entsprechende split Funktion aus dem EOSwithLF Template:

```
1 function split( data, intval, alter, str )
2     if #str == 1 then return STARTED end
3     if data == 10 then return COMPLETED end
4     return MODIFIED
5 end
```

Da das erste eintreffende Zeichen als der Beginn eines neuen Telegramms gewertet wird, wird ein LF in diesem Fall genauso behandelt (und nicht als Endezeichen). Eventuell schlagen Sie jetzt einfach vor, die Zeilen 2 und 3 zu tauschen - das löst das Problem aber nicht. Dann wird ein LF nur angezeigt werden, wenn es am Ende von davon unterschiedlichen Zeichen empfangen

[1] ProfiBus verwendet eine Einzelbyte Nachrichten (SC) als Kurzbestätigung. SC Nachrichten bestehen lediglich aus dem Zeichen e5h.

wurde, aber kein einzelnes LF wie es als Kurzbestätigung vorkommt.
Die split Funktion kann immer nur ein Byte nach dem anderen verarbeiten. Im Falle eines Einbyte-Telegramms müsste die Funktion aber gleichzeitig STARTED und COMPLETED zurückgeben. Zum Glück ist dieses Problem einfach zu lösen indem Sie genau dies tun:

```
1  function split( data, intval, alter, str )
2      if #str == 1 then
3          if data == 10 then
4              return STARTED + COMPLETED
5          else
6              return STARTED
7          end
8      else
9          if data == 10 then
10             return COMPLETED
11         end
12     end
13     return MODIFIED
14 end
```

Zeile 4 liefert ein STARTED und COMPLETED wenn ein Linefeed empfangen wurde (Zeile 3) und wenn es das erste Zeichen des Telegramms (Zeile 2) ist. Nur wenn beide Bedingungen erfüllt sind handelt es sich um ein einzelnes LF.

Telegramme mit einer Startsequence bestehend aus mehreren Bytes

Es existieren Protokolle die statt einem einzelnen Zeichen eine Zeichenkette oder Bytefolge als Telegrammstart spezifizieren. So verwendet z.B. DNP3 die Datenbytes 0x05 und 0x64 (in hexadezimaler Schreibweise) als Start- und Synchronisation-Header. Ein einfaches DNP3 Telegramm sieht wie folgt aus:

Start 0x05	Start 0x64	Length	Control	Destination	Source	CRC

Von der Tatsache einmal abgesehen, dass ein solches Protokoll natürlich jedwede Verwendung der Startsequenz im restlichen Telegramm unterbinden muss, steht die `split` Funktion hier vor einem Problem.

Betrachten Sie dazu folgendes Szenario: Die `split` Funktion wird mit dem Byte 0x05 aufgerufen, welches - eventuell - den Beginn eines neuen Telegramms markiert. In diesem Fall liefert die Funktion als Rückgabewert STARTED, aber Vorsicht!
Was wenn das nachfolgende Zeichen nicht 0x64 entspricht? In diesem Fall ist das zuvor zurück gegebene STARTED schlicht falsch. Sie können natürlich erst auf das zweite Zeichen warten und dann beide mit der Sequenz 0x05 0x64 vergleichen. Aber auch hier liefert ein STARTED nicht das gewünschte Ergebnis. Vielmehr wird das 0x05 an das letzte Telegramm angehängt während das neue Telegramm mit 0x64 beginnt.
Was wir benötigen ist ein Mechanismus um ein beliebiges Byte als möglichen Start (eines neuen Telegramms) zu 'markieren' um dann später zu entscheiden, ob und wann dieses Byte als Telegrammstart gewertet wird.

Hier kommt der Rückgabestatus MARKED ins Spiel. Ein zurück gegebenes MARKED startet kein neues Telegramm. Vielmehr wird eine Analyse der folgenden Bytes fortgeführt als wäre nichts passiert. Erst wenn die `split` Funk-

tion STARTED liefert (und nur STARTED) wird das zuvor per MARKED gekennzeichnete Byte (bzw. die Byte Position im Datenstrom) als neuer Telegrammstart verwendet. Ein Beispiel:

```
1  function split( data )
2      if data == 100 and last == 5 then
3          return STARTED
4      end
5      last = data
6      if data == 5 then
7          return MARKED
8      end
9      return MODIFIED
10 end
```

Sieht trivial aus, aber wie funktioniert es im Detail?
Der Trick liegt in der Verwendung einer globalen Variable `last` die immer das zuvor empfangene Datenbyte enthält. Per Vorgabe sind Variablen in Lua immer global und werden mit `nil` initiiert (siehe auch 14.3.1).
Unter normalen Umständen empfehlen wir deshalb alle Variablen mit dem Zusatz `local` zu definieren um eine ungewollte Mehrfachverwendung zu vermeiden. In diesem Fall macht die Verwendung einer globalen Variable das Handling aber deutlich einfacher.

In der ersten Funktionszeile (Zeile 2) prüfen wir ob das gerade empfangene Datenbyte (übergeben via Parameter `data`) 0x64 entspricht UND gleichzeitig das vorherige Byte ein 0x05 war.
Beim ersten Aufruf der `split` Funktion ist `last` nicht definiert. Lua erzeugt die Variable deshalb on-the-fly und weist ihr den Wert `nil` zu. Da die Variable nicht `local` ist bleibt sie und ihr Inhalt danach bestehen. Ist die Bedingung wahr, bedeutet das ein neues Telegramm und Zeile 3 liefert STARTED zurück. Anschließend wird das aktuelle Datenbyte in der globalen Variable `last` für den nächsten Aufruf von `split` gespeichert (Zeile 5).
Die folgende Bedingung in Zeile 6 sorgt dafür, das ein neuer Telegramm Start (ausgelöst durch die Rückgabe von STARTED in Zeile 3) immer dem entsprechenden 0x05 zugewiesen wird.
Ist `data` weder 0x64 noch 0x05 liefert `split` am Ende MODIFIED um das aktuelle Datenbyte an die interne Telegramm Sequenz anzufügen.

Es gibt nur eine interne MARKED Position!

Ein per split zurück gegebenes MARKED überschreibt eine eventuell vorhandene frühere MARKED Position und STARTED liefert deshalb als Telegrammstart immer das letzte MARKED Byte im Datenstrom.

Globale und lokale Variablen

Variablen in Lua sind per default global und nach ihrer ersten Deklaration von überall im Skript sichtbar bzw. zugänglich. Dies kann bei vermeintlich unabhängigen Variablen gleichen Namens zu merkwürdigen Resultaten führen. Betrachten Sie dazu folgende Zeilen:

```
1  function chksum( data )
```

```
2     n = 0
3     for i=1,#data do
4         n = n + data:byte( i )
5     end
6     return n % 255
7  end

8  function out()
9      -- the current telegram
10     tg = telegrams.this()
11     -- query the current telegram length
12     n = tg.size()
13     -- a simple checksum
14     sum = chksum( tg:string() )
15     -- telegrams less than 8 byte need a special handling
16     if( n < 8 ) then
17         -- do something with small telegrams
18     end
19 end
```

In Zeile 12 weisen wir die aktuelle Telegrammlänge der Variable `n` zu und berechnen anschließend die Prüfsumme des Telegramms, indem wir der Funktion `chksum` das Telegramm als Lua String übergeben. Und hier lauert das eigentliche Problem!
Die `chksum` Funktion verwendet intern ebenfalls eine Variable `n` zur Aufsummierung der einzelnen Bytewerte. Aus der Sicht des Lua Interpreters existiert diese ABER bereits (deklariert durch die vorherige Zuweisung der Telegrammlänge). `n` wird deshalb in Zeile 2 (in der `chksum` Funktion) zuerst auf 0 gesetzt und danach mit den Bytes des Strings `data` aufaddiert.
Bei der Abfrage in Zeile 16 enthält `n` nicht mehr die Telegrammlänge sondern die Summe der Telegrammbytes! Nicht gerade das was wir beabsichtigten!
Sie können natürlich die Variable `n` in der chksum Funktion einfach umbenennen. Dies ist aber vor allem bei umfangreicheren Templates nicht immer so leicht möglich und bedeutet einen nicht zu unterschätzender Pflegeaufwand.
Eine bedeutend einfachere Lösung ist: Variablen, die nur innerhalb einer Funktion benötigt werden, als 'lokal' zu deklarieren. Lua reserviert hierfür das Schlüsselwort **local**. Auf unser Beispiel angewandt reicht die Änderung von Zeile 2:

```
1  function chksum( data )
2     local n = 0
3     for i=1,#data do
4         n = n + data:byte( i )
5     end
6     return n % 255
7  end
```

Die lokale Variable `n` existiert jetzt nur noch innerhalb der Funktion `chksum` und hat mit der Telegrammlänge `n` in Zeile 12 nichts mehr gemeinsam.
Aber was ist mit der Variable `i`?
Die Laufvariable innerhalb einer **for** Schleife hält Lua generell lokal. Sie ist nur innerhalb der Schleife sichtbar und wird anschließend gelöscht.

Sie sehen: Es ist generell immer eine gute Idee ALLE Variablen als **local** zu deklarieren und nicht lokale Variablen, die Sie über mehrere Funktionen hinweg verwenden mit einer entsprechenden Kennzeichnung zu versehen. Z.B. `g_n` (globale n).

Mehr Informationen über das aktuelle Datenereignis

Die an `split` übergebenen Parameter sollten für die meisten Anwendungsfälle völlig ausreichen. In ganz speziellen Situation werden Sie aber eventuell zusätzliche Informationen über das aktuelle Datenereignis benötigen, ohne die Sie die Telegramme nicht korrekt aus dem Datenstrom extrahieren zu können. Z.B. die Richtung oder Quelle (und nicht nur einen Wechsel derselben), den genauen Zeitstempel (und nicht nur die Distanz zum vorherigen Datenereignis).
Da das Hinzufügen weiterer Parameter u.U. die Kompatibilität zu älteren Templates einschränkt, werden diese zusätzlichen Informationen durch das `event` Modul zur Verfügung gestellt. Das Modul wird ausführlich in Lua Modul Abschnitt 14.8.3 beschrieben.
Hier ein Beispiel wie Sie die Quelle des aktuellen Datenbytes innerhalb `split` ermitteln können. Vorausgesetzt wird ein Protokoll mit unterschiedlichen EOS Zeichen abhängig von der Datenrichtung. Alle Telegramme an Port A (CH1) verwenden ein CR als EOS, während Telegramme an Port B (CH2) mit einem LF beendet werden.

```
1  function split( data, intval, alter, str )
2      local eos = 13
3      if event.dir() == 2 then eos = 10 end
4      if #str == 1 then return STARTED end
5      if data == eos then return COMPLETED end
6      return MODIFIED
7  end
```

Split - Zusammenfassung

Die `split` Funktion verspricht eine Anpassung an fasst alle möglichen Protokolle. Obgleich sie i.a. nur aus einigen wenigen Zeilen besteht ist ein besseres Verständnis der Skriptsprache Lua von Vorteil wenn Sie Ihre eigenen Templates schreiben. In Kapitel 18.1 finden Sie eine komplette Einführung in diese faszinierende Sprache.

Das Auftrennen des Datenstromes in einzelne Telegramme ist die eine Sache. Im nächsten Abschnitt lernen Sie wie Sie die Ausgabe der Telegramme ganz individuell nach Ihren Vorgaben steuern.

14.3.2 Individuelle Darstellung der Telegramme

Die komplette Formatierung, d.h. wie die Telegramme im einzelnen dargestellt werden, erfolgt in der Funktion `out`. Diese Funktion wird jedesmal aufgerufen, wenn der Protokollmonitor ein Telegramm im Telegrammfenster ausgibt. Gleichzeitig reduziert dies den Rechenaufwand, da der Code nur für die gerade sichtbaren Telegramme ausgeführt wird.

```
1  function out()
2      -- your formating instructions
3  end
```

Das Prinzip des neuen Ausgabe-Mechanismus basiert auf einer Menge von individuellen rechteckigen Boxen angeordnet in einer Zeile, wobei jede Zeile ein Telegramm repräsentiert.
Jede dieser Boxen enthält einen Titel (caption) und einen Textinhalt (content), beides spezifiziert durch den Anwender. Text- und Hintergrundfarbe sind frei

definierbar. Titel und Text sind Lua Strings und können das Ergebnis einer beliebigen Operation mit den Daten des zugehörigen Telegrammes sein. Das gleiche gilt für die Farben. Ein Beispiel:
Das folgende Bild zeigt ein einzelnes Modbus RTU Telegramm realisiert mit dem neuen Mechanismus. Die verschiedenen Elemente des Telegramms wie Geräteadresse, Funktionsnummer etc. werden in einzelnen Boxen dargestellt. Die letzte Box enthält die validierte CRC16 Prüfsumme, wobei grün hier für einen korrekten CRC16 Wert steht.

Die Größe (Breite) einer Box berechnet sich aus dem Inhalt. Jede neue Box wird automatisch rechts neben der vorherigen Box angefügt. Das bedeutet: Die Position der Boxen in einer Zeile ergibt sich aus der Reihenfolge, in welcher diese in der Lua `out()` Funktion aufgerufen werden. Der erste Aufruf einer Box zeichnet die Box ganz links (am Anfang der Zeile), der zweite Aufruf die Box rechts neben der ersten und so weiter...
Eine einfache Box ist wie folgt definiert:

```
1  function out()
2      local telegram = telegrams.this()
3      box.text{ caption="Time", text=telegram:time()}
4  end
```

Time
0.137345

Eine einfache Box

Beachten Sie! Wir verzichten hier darauf, die zugehörige Split Funktion zu zeigen und konzentrieren uns ausschließlich auf die Telegrammausgabe. Später werden wir das Box Modell an einem Beispiel inklusive der Split Funktion diskutieren.
Der obige Code liefert eine einfache Box wie am Rand gezeigt. Der Titel oder die Überschrift ist 'Time' (wir wollen die Zeitmarke des Telegramms anzeigen). Der Inhalt ist das Resultat des `telegram:time()` Aufrufs den wir später noch erläutern. So lange wir keine Farben spezifizieren wird die Box mit schwarzem Text/Rahmen und weißem Hintergrund ausgegeben.

Wir wollen nun die Ausgaben mit den eigentlichen Daten des Telegramms ergänzen.

```
1  function out()
2      local telegram = telegrams.this()
3      box.text{ caption="Time", text=telegram:time() }
4      box.text{ caption="Data (hex)", text=telegram:dump{} }
5  end
```

Dazu fügen wir in Zeile 4 eine zweite Box hinzu. Anstelle die einzelnen Datenbytes einzeln zu extrahieren 'bitten' wir das Telegramm seine Daten in Form eines Hexdumps auszugeben. Die Telegramm Funktion `dump` liefert einen beliebigen Datenausschnitt des zugehörigen Telegramms als Hexdaten String. Per Voreinstellung (ohne Parameter) wird die vollständige Datensequenz des Telegramms verwendet.
Die geschweiften Klammern des `dump` Aufrufs weisen, wie bei `box.text`, darauf hin, das die Funktion benannte Parameter erwartet.

Time	Data (hex)
2.339189	03a 030 032 030 032 043 034 00d 00a

Time	Data (hex)
2.351468	03a 030 032 030 032 030 046 031 035 036 041 034 032 037 044 00d 00a

Die Box mit den Hexdaten erscheint hinter der Time Box da sie NACH der Time Box aufgerufen wird. Alle Datenbytes werden per Voreinstellung als 3-stellige Hexwerte angezeigt (wohlgemerkt, der MSB-Analyser unterstützt 9 Bit Daten).
Sie werden `dump` mit Sicherheit sehr häufig verwenden, da sie die einfachste Möglichkeit darstellt, die Telegrammdaten auszugeben ohne sich Gedanken über die Telegrammlänge bzw. den Inhalt machen zu müssen.

Zugriff auf die Telegrammdaten

Die `out()` Funktion wird für jedes einzelne im Telegrammfenster darzustellende Telegram ausgeführt. Zeigt das Telegrammfenster z.B. die ersten zehn von `split` generierten Telegramme, wird `out()` zehnmal ausgerufen, beginnend mit dem Telegramm Nr.1 und endend mit Telegramm Nr.10.
Wenn Sie anschließend durch die Aufzeichnung scrollen und das Telegrammfenster einen beliebigen Abschnitt innerhalb der Aufnahme einblendet, sagen wir einmal die Telegramme von 1201 bis 1217, so erfolgt der `out()` Aufruf für die Telegramme 1201, 1202, ... bis das letzte sichtbare Telegramm 1217 erreicht ist.
Das Modul `telegrams` liefert dazu die zur aktuellen Zeile oder zum aktuellen `out()` Aufruf zugehörigen Telegrammdaten mit:

```
local telegram = telegrams.this()
```

So zeigen die folgenden Programmzeilen

```
local telegram = telegrams.this()
box.text{ caption="Number", text=telegram:number() }
```

die Telegrammnumber eines jeden Telegramms welches gerade sichtbar ist und in der Funktion `out` verarbeitet wird. Das `telegrams` Modul bietet allerdings mehr als nur den Zugriff auf das aktuelle Telegramm.
Stellen Sie sich vor, die Darstellung eines Telegramms hängt von Informationen in zuvor empfangenen Telegrammen ab. Oder Sie benötigen die 'Antwortzeit' zwischen dem aktuellen und vorherigen Telegramm, um zu entscheiden ob es sich um eine Anforderung oder eine Antwort handelt[2].

Das `telegrams` Modul bietet einen wahlfreien Zugriff auf alle Telegramme, von der allerersten aufgenommenen Sequenz bis zu dem gerade in `out` Funktion verwendeten Telegramm[3].
Der Zugriff auf ein beliebiges Telegramm erfolgt einfach mit:

```
local telegram = telegrams.at( index )
```

Dabei adressiert der Parameter `index` ein Telegramm auf zweierlei Weise.
Ein positiver Index (absolute Adresse oder Position innerhalb der Aufzeichnung) gibt das Telegramm mit dem angebenen Index bzw. Nummer zurück. So liefert ein Index von 1 das allererste Telegramm , ein Index von 100 das

[2] Das Modbus RTU Template verwendet eine solche Vorgehensweise
[3] Der Telegramm Zugriff ist damit nicht länger auf das aktuelle und vorherige Telegramm begrenzt wie in früheren Programmversionen.

Hunderste. Sollte kein Telegramm an dem angegebenen Position existieren erhalten Sie ein klassisches Lua `nil`.

Weit interessanter sind allerdings 'negative' Indexe. Ein negativer Indexwert steht für 'relative Adressierung' und zählt rückwärts vom aktuell in `out()` behandelten Telegramm.
So bedeutet ein Index von -1 das aktuelle Telegramm in `out()` und `telegrams.this()` ist lediglich ein etwas eingängigerer Alias dafür. Ein Index von -2 greift auf das vorherige Telegramm zu. Auch hier existiert mit `telegrams.prev()` ein Alias. Anwender mit Lua Background dürften negative Indexe bereits aus mehreren Lua String Funktionen geläufig sein.
Die folgenden Code Zeilen demonstrieren wie Sie die Antwortzeit zwischen dem vorherigen und aktuellen Telegramm ermitteln können:

```
1 function out()
2     local tcurr = telegrams.this()
3     local tprev = telegrams.prev()
4     if not tprev then
5         tprev = tcurr
6     end
7     local dt = tcurr:time() - tprev:time()
8 end
```

Während `telegrams.this()` immer ein gültiges Ergebnis liefert, ist dies beim Zugriff auf den Vorläufer des allerersten Telegramms nicht gegeben. `tprev` bekommt einen `nil` Wert zugewiesen, sobald Sie zum Anfang der Aufzeichnung scrollen. Da `nil` keine gültigen Telegrammdaten darstellt, belohnt Sie Lua ohne die Abfrage in Zeile 4 hier mit einem leeren Telegramm Fenster.

In den allermeisten Fällen werden Sie in `out()` nur auf das entsprechend zugehörige Telegramm per `telegrams.this()` zugreifen. Es gibt allerdings auch Protokolle bei denen die Reaktion eines Busteilnehmers von einem zuvor empfangenen Telegramm abhängt. Um ein solches Verhalten korrekt nachzubilden müssen Sie durch die davor liegenden Telegramme iterieren und diese entsprechend auswerten.
Die Zugriffszeit von `telegrams.at(index)` ist linear, nicht desto trotz ist es keine gute Idee über eine unbegrenzte Anzahl von Telegrammen zu iterieren, da dies leicht zu Endlosschleifen führt die der Lua Interpreter mit einem 'Overrun of allowed executions' bestraft. Vermeiden Sie dies indem Sie z.B. die Anzahl der `telegrams.at(index)` Zugriffe innerhalb einer Schleife auf eine von vorn herein bestimmte Anzahl begrenzen.

Wie Sie auf das aktuelle oder auch beliebig andere Telegramm in `out()` zugreifen haben wir nun im Detail erörtert. Wenden wir uns jetzt dem eigentlichen Ergebnis zu, dem Rückgabewert `telegram`.

Der Protokollmonitor spezifische Lua Typ `telegram` repräsentiert ein einzelnes Telegramm. Dieser ist am besten mit einem Container (oder einem Objekt) zu vergleichen welches alle zu diesem Telegramm relevanten Information enthält. D.h. Zeit, Länge, Richtung usw.
Die Variablen `tcurr` und `tprev` im vorherigen Beispiel sind vom Typ `telegram`.
Verwechseln Sie den Typ `telegram` nicht mit einem Modul. `telegram` entspricht eher einen normalen Lua Datentyp wie z.B. einem String oder einer

Zahl und ist immer das Ergebnis eines zuvor erfolgten `telegrams` Modul Aufrufes. Sie können das Ergebnis (den Typ `telegram`) einer Variablen zuweisen - wie in unserem Beispiel - oder direkt verwenden.
Die folgenden beiden Beispiele liefern die gleiche Ausgabe. Als erstes ein Ansatz ohne Variable.

```
1 box.text{ caption="Number", text=telegrams.this():number() }
2 box.text{ caption="Time", text=telegrams.this():time() }
3 box.text{ caption="Length", text=telegrams.this():size() }
```

Das ist leicht machbar, führt aber zu drei identischen und daher unnötigen Aufrufen von `telegrams.this()`. Ein besser Ansatz ist:

```
1 local tg = telegrams.this()
2 box.text{ caption="Number", text=tg:number() }
3 box.text{ caption="Time", text=tg:time() }
4 box.text{ caption="Length", text=tg:size() }
```

Der Unterschied zwischen . und : in Lua

In den Beispielen haben wir eine Menge Punkte '.' und Doppelpunkte ':' verwendet und Sie haben sich eventuell bereits gefragt wo der Unterschied in der Syntax zwischen beiden liegt.
Ein Punkt in `telegrams.this()` greift auf die Funktion `this` des `telegrams` Moduls zu. Ein Modul ist - einfach gesprochen - organisiert als eine Tabelle und die Funktion `this` ist einer von mehreren Tabelleneinträgen. Der Punkt hier verweist auf den Eintrag `this` in der Tabelle `telegrams`. Insofern können Sie ein Modul auch als ein Kollektion zusammengehöriger Funktionen betrachten.

Was ist aber mit `tg:number()` ? Der Ausdruck erscheint ähnlich, und zwar der Aufruf einer Funktion `number()` des `tg` 'Objekts'.
Der Begriff 'Objekt' ist hier ganz bewußt gewählt. `tg` ist eine Variable oder ein Objekt vom Typ `telegram` aber es ist KEIN Modul! Mit dem Doppelpunkt ':' weisen Sie Lua an die Funktion (hinter dem Doppelpunkt) der zugehörigen Variable oder des Objekts (benannt vor dem Doppelpunkt) auszuführen. `tg:number()` liefert deshalb die Nummer eines ganz bestimmten mit `tg` assozierten Telegramms.

```
1 local tg = telegrams.this()
2 -- the number of the current telegram
3 tg:number()
4 tg = telegrams.prev()
5 -- now it's the number of the previous telegram
6 tg:number()
```

In obigem Beispiel wird die Variable `tg` zunächst mit dem aktuellen Telegramm initialisiert und danach 'seine' Telegrammnummer erfragt. Anschließend wird `tg` das vorherige Telegramm zugewiesen. `tg` ist nun identisch mit diesem Telegramm und eine erneute Abfrage 'seiner' Nummer liefert nun eine andere um eins niedriger Telegrammnummer.

Die folgende Regel möge als kleine Eselsbrücke dienen:

Verwenden Sie ':' wenn Sie sagen können: ≫ Variable, bitte tue dies für mich ≪

Einen Telegramminhalt untersuchen

Sie können verschiedenste Telegramm Informationen abfragen indem Sie die entsprechende Funktion aufrufen. Alle verfügbaren Telegramm Funktionen sind im Abschnitt 14.8.7 aufgelistet. Eine dieser Funktionen sei hier explizit genannt, da sie gerade bei der Analyse unbekannter Telegramminhalte einen sehr schnellen Überblick auf die im Telegramm enthaltenen Daten liefert. Es handelt sich um die Funktion `dump{}`. Sie haben sie bereits früher in diesem Kapitel kennengelernt.
`dump` gibt den Inhalt 'seines' Telegramms als Lua String aus. In diesem sind alle Datenbytes entweder in hexadezimaler oder dezimaler Schreibweise und durch ein Leerzeichen getrennt aufgelistet (Hexdump). Ein `dump{}` Aufruf akzeptiert folgende benannte Parameter, hier mit ihren entsprechend Defaultwerten:

```
telegram:dump{first=1, last=-1, sep=' ', base=16, width=3, max=size/2}
```

Ohne Parameter liefert `dump` den kompletten Inhalt (first=1, last=-1) als 3-stellige (width=3) Hexwerte (base=16), getrennt durch ein Separator Leerzeichen (sep=' ').
Der Parameter max begrenzt die maximal Anzahl Datenbytes und gibt lediglich die erste und letzte Hälfte von den in max spezifizierten Bytes aus.
Nehmen wir ein Telegramm mit der Bytefolge:

3A	30	32	30	32	30	46	31	35	36	41	34	32	37	44	0D	0A

Und eine einfache `out()` Funktion:

```
1  function out()
2      local tg = telegrams.this()
3      box.text{ caption="Time", text=tg:time() }
4      box.text{ caption="Data (hex)", text=tg:dump{} }
5  end
```

Die Ausgabe ist dann:

Time	Data (hex)
2.351468	03A 030 032 030 032 030 046 031 035 036 041 034 032 037 044 00D 00A

Mit dem Parameter *base* können Sie die Zahlenbasis von hexadezimal auf dezimal ändern indem Sie `base=10` übergeben.

```
4  box.text{ caption="Data (dec)", text=tg:dump{ base=10 } }
```

Time	Data (dec)
2.351468	058 048 050 048 050 048 070 049 053 054 065 052 050 055 068 013 010

Da das Telegramm nur 8-Bit Werte enthält, begrenzen wir als nächstes die Stellenanzahl bei der hexadezimalen Darstellung auf 2 Digits.

```
4  box.text{ caption="Data (hex)", text=tg:dump{ width=2 } }
```

Time	Data (hex)
2.351468	3A 30 32 30 32 30 46 31 35 36 41 34 32 37 44 0D 0A

Soweit so gut. Stellen Sie sich nun vor Sie wollen die letzten beiden Bytes (das CRLF) als individuelle End-Of-String Box anzeigen. `dump` hält dazu die zwei Positionsparameter *first* und *last* bereit, mit welchen Sie einen beliebigen Ausschnitt auswählen können. Sie können eine Byte Position als absoluten

Wert angegeben, z.B. first=1 für das allererste Byte im Telegramm. Oder Sie zählen rückwärts mit einem negativen Positionswert.

```
1  function out()
2      local tg = telegrams.this()
3      box.text{ caption="Time", text=tg:time() }
4      box.text{ caption="Data (hex)", text=tg:dump{ first=1, last=-3,
           width=2 } }
5      box.text{ caption="EOS", text=tg:dump{ first=-2, last=-1, width=2 }
           }
6  end
```

Das letzte Byte wird mit -1 adressiert. Um die beiden letzten Bytes auszuwählen übergeben wir einen Bereich von *first=-2* und *last=-1*. Entsprechend beenden wir die Datenausgabe in Zeile 4 mit dem letzten Byte vor dem CRLF, d.h. Position -3 (drei von hinten gezählt).

Time	Data (hex)	EOS
2.351468	3A 30 32 30 32 30 46 31 35 36 41 34 32 37 44	0D 0A

Wie Sie sehen bieten negative Indexe einen sehr konfortablen Weg um Daten relativ zum Telegrammende auszugeben ohne sich dabei mit der Telegrammlänge abmühen zu müssen.

Der Parameter *sep* ist leicht verständlich. Mit seiner Hilfe können Sie das Leerzeichen zwischen den Zahlenwerten durch jedes andere Zeichen oder String ersetzen. Und - natürlich - auch komplett entfernen mit:

```
4  box.text{ caption="Data (hex)", text=tg:dump{ width=2, sep='' } }
```

Time	Data (hex)
2.351468	3A30323032304631353641343237440D0A

Der Protokollmonitor behandelt Telegramme ohne Längenbegrenzung. Gleichwohl ist es manchmal ziemlich lästig horizontal durch eine Riesenmenge von `dump{}` produzierten Daten zu scrollen. Hier kommt der letzte Parameter *max* ins Spiel.
max definiert die maximale Anzahl der von `dump{}` ausgegebenen Daten, die erste Hälfte von *max* am Anfang, die zweite Hälfte am Ende. Die übrigen Daten werden als Anzahl zwischen zwei Aufzählungspunkten angezeigt. Ein Beispiel:

```
4  box.text{ caption="Data (hex)", text=tg:dump{ width=2, max=4 } }
```

Data (hex)
3A 30 ...[13]... 0D 0A

Tutorial
Tutorial.msbprj

Ihr eigenes Template Schritt für Schritt

Für ein besseres Verständnis der nächsten Schritte empfehlen wir Ihnen die Projektdatei `Tutorial.msbprj` im `Examples\ProtocolView` Verzeichnis zu laden bzw. per Doppelklick zu starten. Das Beispiel funktioniert auch ohne angeschlossenen MSB-Analyser und bietet Ihnen die Möglichkeit, die folgenden Template Veränderung direkt im Protokollmonitor anzusehen.

Das Beispielprojekt enthält die Aufzeichnung eines einfachen Protokolls wobei jedes Telegramm mit einem Doppelpunkt ':' eingeleitet und mit der Sequenz CRLF (Carriage Return und Linefeed) beendet wird. Eventuell haben Sie es

bereits in den oben abgebildeten EOS Box gesehen (die Daten 013 010).
Wir haben ein solches Protokoll bereits bei der Erklärung der `split` Funktion beschrieben und können es hier einfach übernehmen. Zur Erinnerung:

```
1  function split(data,intval,alter,str)
2      if data == 58 then return STARTED end
3      if str:find("\r\n") then return COMPLETED end
4      return MODIFIED
5  end
```

Unser einfaches Protokoll spezifiziert zudem eine Geräteadresse des Telegramm Empfängers, eine Funktionsnummer sowie die eigentlichen Daten und eine einfache Prüfsumme. Alles in allem erinnert es ein wenig an Modbus ASCII.
Das Projekt Template ist schreibgeschützt. Sie müssen deshalb zunächst eine Kopie des Templates erstellen indem Sie bei aufgeklapptem Template Editor auf den [+] Knopf klicken und einen neuen Template Namen eingeben. Z.B. 'MyTutorial' oder ähnliches.

Als ersten werden wir jedes Telegramm entsprechend seiner Herkunft farblich unterschiedlich kennzeichnen um die Datenrichtung leichter erkennen zu können. Wie üblich wählen wir rot für Telegramme empfangen an Port A (CH1) sowie blau für alle Daten von Port B (CH2).

Wie bereits erwähnt kann jede Box eine individuelle Text/Rahmen- sowie Hintergrundfarbe erhalten. Die Farbwerte werden als RGB (Red Blue Green) Werte übergeben, z.B. `0xAABBCC`. Das erste Byte (AA) definiert den Rotanteil (in Schritten von 0...255), das zweite den Grünanteil (BB) und das letzte Byte den Anteil der blauen Farbe (hier CC). Schwarz ist in dieser Definition 0x000000, weiß 0xFFFFFF.
Alle Telegramme empfangen an Port A sollen mit rotem Text auf hellrotem Hintergrund dargestellt werden. Die Telegramme an Port B entsprechend in blauer Schritt auf hellblauem Hintergrund. Ok, los geht's:

```
1  function split( data, intval, alter, str )
2      if data == 58 then return STARTED end
3      if str:find("\r\n") then return COMPLETED end
4      return MODIFIED
5  end
6
7  function out()
8      -- telegram colors
9      local textcolors = { 0xFF0000, 0x0000FF }
10     local backcolors = { 0xFFEEDD, 0xDDEEFF }
11
12     -- access the current telegram
13     local tg = telegrams.this()
14
15     -- select the text and background color depending on the data
           source
16     local fc = textcolors[ tg:dir() ]
17     local bc = backcolors[ tg:dir() ]
18
19     -- display time
20     box.text{ caption="Time", text=tg:time(), fg=fc, bg=bc }

21
```

```
22      -- display all data as hex
23      box.text{ caption="Data (hex)", text=tg.dump{}, fg=fc, bg=bc }
24  end
```

Zur Vollständigkeit haben wir die `split` Funktion ebenfalls mit aufgelistet.
Zeile 9 ist eine typische Lua Tabelle (oder Array) mit zwei Einträgen für die Textfarbe (je nach Richtung). Zeile 10 definiert entsprechend zwei Farben für den Hintergrund.
In Zeile 13 holen wir das zuständige Telegramm und weisen es der Variable `tg` zu. Die Funktion `tg:dir()` liefert die Quelle oder Richtung des Telegramms als Wert 1 (Port A/Ch1) oder 2 (Port B/CH2). Das Resultat wird in den Zeilen 16 und 17 genutzt um für das aktuelle Telegramm Text- und Hintergrundfarbe auszuwählen. Zu guter Letzt übergeben wir diese Farben als zusätzliche Parameter *fg* und *bg* an die beiden Box Aufrufe (Zeile 20 und 23).
Drücken Sie jetzt die Taste F5 - voila - die bislang schwarz/weiße Telegramm-Anzeige wechselt in eine bunte Darstellung.

Time	Data (hex)
2.339189	03a 030 032 030 032 043 034 00d 00a

Time	Data (hex)
2.351468	03a 030 032 030 032 030 046 031 035 036 041 034 032 037 044 00d 00a

Übrigens: Jede Änderung an dem Template wird automatisch bei Ausführung mittels F5 gespeichert.
Als nächstes wollen wir die beiden Telegramm Begrenzer ':' (3Ah) und die End-Of-Frame Sequenz CRLF gesondert hervorheben um etwaige Abweichungen im Telegramm durch Übertragungsfehler besser erkennen zu können.

```
1   function split( data, intval, alter, str )
2       if data == 58 then return STARTED end
3       if str:find("\r\n") then return COMPLETED end
4       return MODIFIED
5   end
6
7   function out()
8       -- telegram colors
9       local textcolors = {0xFF0000,0x0000FF}
10      local backcolors = {0xFFEEDD,0xDDEEFF}
11
12      -- access the current telegram
13      local tg = telegrams.this()
14
15      -- select the text and background color depending on the data
            source
16      local fc = textcolors[tg:dir()]
17      local bc = backcolors[tg:dir()]
18
19      -- display time
20      box.text{caption="Time",text=tg:time(),fg=fc,bg=bc}
21
22      -- start colon
23      box.text{caption="SOF",text=string.char(tg:data(1)),bg=fc,fg=bc}

24
25      -- display all data as hex
26      box.text{caption="Data (hex)",text=tg:dump{first=2,last=-3},fg=fc,
            bg=bc}
27
28      -- end of frame CRLF
```

```
29      box.text{caption="EOF",text=tg:dump{first=-2},fg=bc,bg=fc}
30  end
```

Zur Anzeige des Startzeichens ':' verwendet Zeile 23 eine normale Text Box. Der Titel (caption) der Box ist SOF (Start Of Frame). Als Boxinhalt wird das erste Byte des Telegramms mit `tg:data(1)` abgefragt. Dies liefert bei einem korrekten Telegramm den Dezimalwert 58.
Besser wäre hier die direkte Darstellung als Doppelpunkt. Wir wandeln deshalb den Bytewert mit: `string.char(tg:data(1))` in seine Zeichenrepräsentation (ASCII) um[4].
Das Ende der Telegramm Sequenz benötigt keine Konvertierung. Zur besseren Lesbarkeit geben wir beide Zeichen einfach als Box mit invertierten Farben aus (Zeile 29).
Zum Schluss müssen wir nur noch bei der Darstellung der eigentlichen Daten die Länge anpassen (Zeile 26). Die anzuzeigenden Daten beginnen jetzt an Position 2 (die erste ist das SOF) und enden mit dem letzten Byte vor dem CRLF bzw. Position -3. Das Resultat unserer Modifikationen sehen Sie im folgenden Bild:

Time	SOF	Data (hex)	EOS
2.339189	:	030 032 030 032 043 034	00d 00a
2.351468	:	030 032 030 032 030 046 031 035 036 041 034 032 037 044	00d 00a

Die Daten werden als 3-stellige Hexwerte angezeigt. Dies ist die Voreinstellung da der MSB-Analyser 9 Bit Daten unterstützt.
Unser Beispiel enthält keine 9 Bit Datenwörter, wir können deshalb die Darstellung auf 2 Stellen begrenzen. Der Parameter `width` erlaubt die Angabe von einer abweichenden Stellenzahl.
Hier die angepasste Zeile 26:

```
26  box.text{caption="Data (hex)",text=tg:dump{first=2,last=-3,width=2},fg=
        fc,bg=bc}
```

...und das Resultat für das 'rote' Telegramm:

Time	SOF	Data (hex)	EOS
2.339189	:	30 32 30 32 43 34	00d 00a

Handhabung von Daten im base16 Format

Im weiteren Verlauf werden wir uns nun einige Fähigkeiten des Protokollmonitors ansehen, die weit über das hinaus gehen, was in den früheren Versionen möglich war.
Unser Beispiel simuliert die Bus Kommunikation zwischen einem Sender und zwei Geräten (Sensoren) die kontinuierlich die Temperatur, den Luftdruck und die Luftfeuchtigkeit messen.
Stellen Sie sich vor, jedes dieser beiden Geräte steht an einem unterschiedlichen Standort. Der Sender (oder Bus Master) erfragt bei jeden Teilnehmer in wahlloser Reihenfolge eine dieser Informationen. Die Antwort ist eine Fliesskommazahl. Mit Ausnahme des Startzeichens ':' sowie dem CRLF am Ende werden alle Datenbytes des Telegramms als zwei ASCII Zeichen kodiert (hex

[4] Das string Modul ist Teil der Lua Sprache.

Format oder Base16). Ein Beispiel:
Das Byte 5Bh wird als zwei Zeichen 35h und 42h übertragen (35h = '5', 42h = 'B' in ASCII).
Zu guter Letzt sorgt eine einfache Prüfsumme für die Integrität der übertragenen Daten. Ein einzelnes Telegramm sieht damit wie folgt aus:

Start	Address	Function	Data	Checksum	End
:	2 chars	2 chars	0 or 8 chars	2 chars	CRLF

Beachten Sie das die Anfragen des Master ein leeres Datenfeld besitzen!

In einem ersten Schritt wandeln wir die im Base16 (oder Hex-ASCII) Format vorliegenden Daten zurück in ihre binäre Represenatation. Dies erleichert uns die nachfolgende Bearbeitung der im Telegramm enthaltenen Informationen. Sie können - natürlich - eine kleine Lua Funktion schreiben, die dies übernimmt. Allerdings bietet der Protokollmonitor für solche Zwecke ein eigenes `base16` Modul um solche Daten flexibel zu handhaben. Das Module ist auf Seite 261 detailiert beschrieben.
Zur Rückgewinnung der Originaldaten in einem base16 kodierten String übergeben Sie diesen einfach der Funktion `base16.decode(string)`.

```
1  local tg = telegrams.this()
2  local bindata = base16.decode( tg:string():sub( 2, -3 ) )
```

`tg:string()` liefert alle in einem Telegramm enthaltenen Bytes als Lua string. Der Doppelpunkt ':' und die End-Of-String Sequenz CRLF sind allerdings nicht Teil der in Base16 kodierten Daten. Dekodiert werden muss der Teilstring vom zweiten Byte (Position 2) bis zum dritt-letzten Byte (Position -3). Die Extraktion eines Teilstrings ist in Lua eine häufig benötigte Anwendung und Lua's string Modul offeriert hierzu eine geeignete Funktion: `sub(first,last)`.
Das einzige was wir tun müssen ist die `sub` Funktion direkt vom Ergebnis via `tg:string():sub(2,-3)` aufzurufen.

Beachten Sie bitte! Da Lua Strings nur normale Bytes enthalten können, werden alle 9-Bit Daten im Telegramm auf normale Bytes reduziert. Wenn Ihr Protokoll explizit 9-Bit Werte verwendet, müssen Sie auf die einzelnen Daten per `telegram:data(index)` zugreifen.

Das Resultat der Umwandlung wird in Zeile 2 der Variable `bindata` zugewiesen, welche wir für alle weiteren Schritte verwenden wollen. Der Inhalt der binären Sequenz ist:

Address	Function	Data	Checksum
1 byte	1 byte	0 or 4 bytes	1 byte

Sie können die einzelnen Bytes eines Lua Srings mit `string:byte(index)` erfragen. Diese Funktion arbeitet ähnlich der `telegram:data(index)` und liefert das Byte an der angegebenen String Position. Eine Anzeige von Adresse und Funktion ist damit einfach zu realisieren:

```
box.text{ caption="Address", text=bindata:byte(1), fg=fc, bg=bc }
box.text{ caption="Function", text=bindata:byte(2), fg=fc, bg=bc }
```

Die Prüfsumme ist das letzte Byte in der Binärsequenz und es wäre schön diese in hexadezimaler Notation zu sehen. Dazu übergeben wir das Checksum Byte der string format Funktion:

```
box.text{caption="Chksum",
        text=string.format("%02X",bindata:byte(-1)),fg=fc,bg=bc}
```

Wie bereits erwähnt unterscheidet unser Protokoll zwischen Anfragen (Requests) und Antworten (Responses). Nur die Antwort Telegramme enthalten ein zusätzliches Datenfeld mit einem 4-Byte umfassenden Fliesskommawert. Antwort-Telegramme sind durch ihre Länge von 17 Byte (Originaltelegramm in Base16 Kodierung) bzw. durch eine Binärsequenz von 7 Bytes (Adresse=1 Byte, Funktion=1 Byte, Daten=4 Bytes, Checksum=1 Byte) gekennzeichnet. Um sie von einer Anfrage zu unterscheiden reicht eine Prüfung der Telegramm- oder `bindata` Länge aus.

```
if tg:size() == 17 then ... end
if #bindata == 7 then ... end
```

Es macht keinen Unterschied welche davon Sie wählen. Da wir aber im weiteren Verlauf auf die `bindata` Sequenz zugreifen wählen wir die zweite Variante.

```
function out()
    -- telegram colors
    local textcolors = { 0xFF0000, 0x0000FF }
    local backcolors = { 0xFFEEDD, 0xDDEEFF }

    -- access the current telegram
    local tg = telegrams.this()
    local bindata = base16.decode( tg:string():sub( 2, -3 ) )

    -- select the text and background color depending on the data
        source
    local fc = textcolors[ tg:dir() ]
    local bc = backcolors[ tg:dir() ]

    -- display time
    box.text{ caption="Time", text=tg:time(), fg=fc, bg=bc }

    -- start colon
    box.text{ caption="SOF", text=string.char( tg:data( 1 ) ), bg=fc,
        fg=bc }

    -- the address field
    box.text{ caption="Address", text=bindata:byte(1), fg=fc, bg=bc }

    -- the function number field
    box.text{ caption="Function", text=bindata:byte(2), fg=fc, bg=bc }

    -- is it a response?
    if #bindata >= 7 then

        -- display the response data as hex
        box.text{caption="Data (hex)",text=tg:dump{ first=3,last=6,
            width=2},fg=fc,bg=bc}

    end
```

```
33
34      -- the checksum byte is always the last byte in bindata
35      box.text{caption="Chksum",text=string.format("%02X",bindata:byte
            (-1)),fg=fc,bg=bc}
36
37      -- end of frame CRLF
38      box.text{ caption="EOF", text=tg:dump{ first=-2, width=2 }, fg=bc,
            bg=fc }
39
40  end
```

Zeile 27 prüft die Länge und damit den Typ. Im Falle einer Antwort (`bindata` enthält mindestens 7 Zeichen) wird in Zeile 30 eine zusätzlicher Datenbox ausgegeben. Das Ergebnis:

Time	SOF	Address	Function	Checksum	EOS
2.339189	:	2	2	C4	0D 0A

Time	SOF	Address	Function	Data (hex)	Checksum	EOS
2.351468	:	2	2	0F 15 6A 42	7D	0D 0A

Die Anzeige der Funktion als Nummer ist in den meisten Fällen ausreichend. Aber wäre es nicht bequemer die Funktionsnummer mit einer eingängigeren Beschreibung oder einem Namen zu versehen, so dass die Bedeutung eines Telegramms sofort ersichtlich ist? In unserem Beispiel sind die Funktionen wie folgt nummeriert:

1 Temperature
2 Moisture
3 Pressure

Eine Lua Funktion, die einen Text abhängig von einem Eingabeparameter zurück liefert, könnte so aussehen[5].

```
1  function GetFunctionName( number )
2      local names = { "Moisture", "Humidity", "Pressure" }
3      return names[ number ]
4  end
```

Zeile 2 erzeugt eine Lua Tabelle (oder Array) mit drei Texteinträgen (Funktionsbeschreibungen). Da `names` als **local** deklariert ist kann auf diese Tabelle von außerhalb nicht zugegriffen werden. Dies vermeidet Konflikte wenn Ihr Template noch an anderer Stelle eine Variable `names` verwendet und ist deshalb die empfohlene Vorgehensweise!
Lua indiziert Tabellen beginnend mit 1. Zeile 3 liefert den Eintrag entsprechend des übergebenen Parameters *number*.

In Lua ist das verschachtelte Anlegen von Funktionen erlaubt. D.h. die Funktion `GetFunctionName` kann sowohl innerhalb von `out()` stehen als auch außerhalb definiert werden. Zur besseren Lesbarkeit des Skripts sollten Sie aber nur relativ kleine Funktion in `out()` platzieren und bei umfangreicheren Funktionen (oder mehreren) diese z.B. am Ende des Skriptes anlegen.

```
1  function out()
2      function inside()
3          -- do something
```

[5] Wir gehen davon aus, das die übergebene Funktionsnummer gültig ist.

```
4       end
5       -- call the function
6       inside()
7   end
```

Lassen Sie uns die einzelne Code Schnipsel zusammentragen. Das folgende Listing zeigt die signifikanten Modifikationen. Wir fügen die Funktion `GetFunctionName()` innerhalb `out()` hinzu und rufen diese mit der Funktionsnummer des Telegramms in Zeile 24 auf.

```
1   function out()
2
3       function GetFunctionName( number )
4           local names = { "Moisture", "Humidity", "Pressure" }
5           return names[ number ]
6       end
7
8       -- access the current telegram
9       local tg = telegrams.this()
10      local bindata = base16.decode( tg:string():sub( 2, -3 ) )
11      ...

23      -- the function number field
24      box.text{caption="Function",text=GetFunctionName(bindata:byte(2)),
            fg=fc,bg=bc}
25      ...
26  end
```

Und hier das Ergebnis für die ersten beiden Telegramme:

Time	SOF	Address	Function	Checksum	EOS
2.339189	:	2	Moisture	C4	0D 0A

Time	SOF	Address	Function	Data (hex)	Checksum	EOS
2.351468	:	2	Moisture	0F 15 6A 42	7D	0D 0A

Die Anzeige der Funktionsnamen ist lediglich ein Beispiel wie bestimmte Abschnitte eines Telegramms in ein von Menschen lesbares Format umgewandelt werden können. Genauso einfach ließe sich noch statt der Geräteadresse der Gerätename ausgegeben.
Wir werden uns aber im Folgenden ganz auf die Daten selbst konzentrieren.

In unserem Protokoll beantworten die Geräte den per Funktionsnummer erfragten Wert als Fliesskommazahl. Die Zahl entspricht dabei der Temperatur, der Feuchtigkeit oder dem Luftdruck.
Das Fliesskommaformat besteht aus vier Bytes im `bindata` String (bzw. acht Bytes in der ursprünglichen Telegramm Sequenz). Das obige Bild zeigt die 4 Bytes in der Data (hex) Box des Antwort Telegramms.
Unsere nächste Aufgabe besteht darin, eine bestimmte Anzahl von Bytes in eine Zahl umzuwandeln.

Bytefolgen in Zahlen umwandeln

Bei der Analyse von Protokollen ist es oftmals erforderlich eine bestimmte Bytesequenz in eine Zahl zu konvertieren. Die verwendeten Zahlwerte können dabei in den unterschiedlichsten Formaten vorliegen: Integer (Ganzzahl) Werte werden als zwei oder vier Bytes übertragen, Fließkommazahlen in Folgen von vier (einfache Genauigkeit) oder acht Bytes (doppelte Genauigkeit).

Und: Selbst die Reihenfolge ist relevant. Manche Protokolle übertragen das höcherwertigste Byte zuerst (Big Endian), andere senden als erstes das niederwertigste Byte (Low Endian).
Glücklicher Weise enthält der Lua Interpreter eine mächtige Funktion um all diese verschiedenen Typen zu verarbeiten.

Die Funktion `string.unpack` erwartet zwei obligatorische Parameter: Die Bytesequenz als Lua String, und wie diese umzuwandeln ist. Ein optionaler dritter Parameter spezifiziert die Position innerhalb der Bytefolge ab der die Konvertierung starten soll, falls diese vom Stringanfang abweicht. Die Funktion `unpack` ersetzt die ältere `bunpack` und ist Teil von Lua's String Module mit der Implementierung von Lua 5.3 in der Analyser Software.

```
val1,...,pos = string.unpack( format, sequence, position )
```

Die Funktion liefert immer mindestens zwei Werte zurück. Der erste bzw. die ersten Werte enthalten immer das/die Resultat(e) der im Formatstring spezifizierten Transformationen. Der letzte Wert (pos) gibt die Position im umzuwandelnden String an, ab der die nächste Umwandlung erfolgt.
Bevor wir zu unserem Tutorial zurückkehren, hier einige Beispiele die Ihnen eine Vorstellung der Arbeitsweise von `string.unpack` vermitteln.

```
1  seq = "\248\036\001\000\154\153\045\065"
2  i,pos = string.unpack( "<i", seq )
3  f,pos = string.unpack( "<f", seq, pos )
```

In Zeile 1 generieren wir eine Zeichenfolge mit den einzelnen Zeichen in dezimaler Notation. So erzeugt ein "\255" ein einzelnes hex FF Byte und die Eingabe "\104\101\108\108\111" ist gleichbedeutend mit dem String "hello". Die dezimale Notation erlaubt uns Sequenzen mit Bytewerten zu erzeugen die nicht unbedingt Teil der normalen Tastatur sind.

Die obige Sequenz ist nicht zufällig gewählt.
Die ersten vier Bytes repräsentieren eine 32-Bit Integerzahl mit dem Wert 75000 in Little-Endian Ordnung (also das niederwertigste Byte zuerst). Die Bytes 5...8 sind das binäre Äquivalent der Zahl 10.85, ebenfalls im Little-Endian Format (LE). Die folgende Tabelle zeigt die Bytefolge in hexadezimaler Schreibweise:

LE Integer 75000				LE Float 10.85			
F8	24	01	00	9A	99	2D	41
1	Byte position						8

Und nun lassen Sie uns sehen wie `string.unpack` aus dieser Aneinanderreihung von Bytes echte Zahlen reproduziert. Wir beginnen mit dem 32-Bit Integer Wert.

```
2  i,pos = string.unpack( "<i", seq )
```

Das erste Argument eines `string.unpack` Aufrufes ist immer der Formatstring (hier `"<i"`). Er definiert, wie der im zweiten Parameter übergebene String umgewandelt werden soll. Der dritte - optionale - Parameter erlaubt die Angabe einer Position in diesem String, ab der die Umwandlung beginnen soll. Da die Vorgabe gleichbedeutend mit dem Stringanfang ist können wir hier auf

diesen verzichten.
Die eigentliche 'Magie' oder Vielfalt von `string.unpack` liegt in seinem Formatstring. Jeweils ein einzelnes Zeichen ist einem ganz bestimmten Datentyp zugeordnet. Ein '<' oder '>' davor definiert die Byteordnung. Ein '<' steht für Little Endian, ein '>' bedeutet eine Interpretation als Big Endian.
Der Formatparamter versteht eine ganze Reihe unterschiedlichster Typen. Sie sind alle detailiert in der frei verfügbaren Lua 5.3 Online Dokumentation beschrieben. Sie finden diese hier: http://www.lua.org/manual/5.3/ oder im entsprechenden Abschnitt **??**.

In unserem Beispiel interpretiert `string.unpack` die ersten vier Bytes als Vorzeichen behaftete 32 Bit Zahl in Little-Endian, d.h. niedriges Byte zuerst (`"<i"`). Der Aufruf liefert zwei Rückgabewerte. Der erste enthält den dekodierten Integer Wert (75000), der zweite die Position im Eingabestring nach der Dekodierung, hier also 5 (das fünfte Byte, nachdem die ersten vier Bytes = 32 Bit verarbeitet wurden).
Die Umwandlung in Zeile 3 startet an der Position die wir im vorherigen Aufruf zurück erhalten haben. Da es sich bei dem erwarteten Zahlentyp um eine 32-Bit Fließkommazahl in Little-Endian Ordnung handelt, übergeben wir als Formatstring `"<f"`.

```
3  f,pos = string.unpack( "<f", seq, pos )
```

Das Resultat ist erneut ein Wertepaar. pos zeigt auf das neunte Byte (das auf die Fließkommazahl folgende Byte) und `f` enthält den Fliesskommawert 10.85.
In unserem Beispiel folgen beide Zahlenwerte (integer und float) direkt aufeinander. In einem solchen Fall können wir die Extraktion auch in einem Rutsch durchführen und auf die 'Mitnahme' des Positionsparameters ganz verzichten, und zwar sowohl im Aufruf als auch bei der Ergebnis Zuweisung. (Lua ist es egal, ob alle Rückgabewerte verarbeitet werden. Es muss nur die richtige Reihenfolge eingehalten werden).

```
i, f = string.unpack( seq, "<i<f" )
```

Fantastisch - nicht wahr!
Und da Ihnen der Protokollmonitor erlaubt, mit diversen Formatparametern zu 'spielen' und gleichzeitig die Auswirkung auf die Telegrammdarstellung zu sehen, ist es besonders einfach Telegrammabschnitte auf bestimmte Datentypen hin zu prüfen. Handelt es sich um ein Ganz- oder Fließkommazahl, um Big oder Little-Endian? Die Antwort ist immer nur eine kurze Änderung des Formatparameters entfernt - ein gewaltiger Vorteil gerade bei unbekannten Protokollspezifikationen.

Ok, lassen Sie uns nach diesem kleinen Exkurs zu unserem Tutorial zurück kehren. Wir hatten bereits die Hex ASCII Daten in ihre binäre Repräsentation umgewandelt. Sie erinnern sich, daß ein Antwort Telegramm den angeforderten Wert (übergeben als Funktionsnummer) als Fließkommazahl enthält. Hier noch einmal die Struktur des Antwort Telegramms:

Address	Function	Float Number	Checksum
1 byte	1 byte	4 bytes	1 byte

Die folgenden Zeilen fassen alle bisherigen Modifikationen in der `out()` Funktion zusammen:

```
1  function out()
2
3      function GetFunctionName( number )
4          local names = { "Moisture", "Humidity", "Pressure" }
5          return names[ number ]
6      end
7
8      -- telegram colors
9      local textcolors = { 0xFF0000, 0x0000FF }
10     local backcolors = { 0xFFEEDD, 0xDDEEFF }
11
12     -- access the current telegram
13     local tg = telegrams.this()
14     local bindata = base16.decode( tg:string():sub( 2, -3 ) )
15
16     -- select the text and background color depending on the data
           source
17     local fc = textcolors[ tg:dir() ]
18     local bc = backcolors[ tg:dir() ]
19
20     -- display time
21     box.text{ caption="Time", text=tg:time(), fg=fc, bg=bc }
22
23     -- start colon
24     box.text{ caption="SOF", text=string.char( tg:data( 1 ) ), bg=fc,
           fg=bc }
25
26     -- the device address
27     box.text{ caption="Address", text=bindata:byte(1), fg=fc, bg=bc }
28
29     -- the function number
30     box.text{caption="Function",text=GetFunctionName(bindata:byte(2)),
           fg=fc,bg=bc}
31
32     if #bindata >= 7 then
33         -- it is a response
34         local value = string.unpack( "<f", bindata, 3 )
35         box.text{ caption="Value", text=value, fg=fc, bg=bc }
36     end
37
38     -- the checksum byte is always the last byte in bindata
39     box.text{caption="Chksum",text=string.format("%02X",bindata:byte
           (-1)),
40              fg=fc,bg=bc}
41
42     -- end of frame CRLF
43     box.text{ caption="EOF", text=tg:dump{ first=-2 }, fg=bc, bg=fc }
44 end
```

Die entsprechende Zeile 34 sollte jetzt leicht verständlich sein. Im Falle eines Reponse Telegramms extrahieren wir die Fließkommazahl (an Position 3) und zeigen den Wert in einer zusätzlichen Box an.
Die Überprüfung der Checksum heben wir uns für später auf. Sie werden vielleicht bemerkt haben, daß wir die Ausgabe der End-Of-String Sequenz per relativer (negativer) Indexierung gelöst haben. Dadurch müssen wir uns nicht um die unterschiedlichen Längen von Anfrage und Antwort Telegrammen kümmern. Die EOS Zeichen CRLF sind immer an der 'vorletzten' Position (-2) zu

finden. Das Resultat für die allerersten beiden Telegramme ist damit:

Time	SOF	Address	Function	Checksum	EOS
2.339189	:	2	Moisture	C4	0D 0A

Time	SOF	Address	Function	Value	Checksum	EOS
2.351468	:	2	Moisture	58.520565032959	7D	0D 0A

Verglichen mit der ersten Anzeige ist das deutlich informativer und verständlicher. Wir sind damit aber noch nicht am Ende. Unser Template enthält noch einigen Spielraum für weitere Verbesserungen. Zum Beispiel: Die Fließkommazahl wird mit zu vielen Nachkommastellen dargestellt. Und es wäre natürlich schön die Prüfsumme zu validieren.
Lua enthält ein integriertes `string` Modul welches neben den üblichen String-Operationen wie suchen, ersetzen und regulären Ausdrücken auch eine C ähnliche Formatfunktion bietet.

```
35  box.text{caption="Data",text=string.format("%.2f",value),bg=bc,fg=fc}
```

`string.format` unterstützt eine ganze Reihe von Variablentypen und Optionen. Detaillierte Informationen dazu finden Sie in einem der Online Manuals die am Ende des Kapitels aufgelistet sind. Hier verwenden wir die Format(ierungs)-Anweisung `"%.2f"`, die eine gegebene Fliesskommazahl ('f') mit 2 Nachkommastellen ausgibt.
Als kleine Übung können Sie eine Funktion erstellen, die abhängig von der Funktionsnummer einen entsprechenden Formatstring inklusive physikalischer Einheit zurückgibt. Eine mögliche Lösung sehen Sie hier:

```
1  function GetFunctionFormat( number )
2      local formats = { "%.2f Deg", "%.2f%%", "%.2fmBar"}
3      return formats[ number ]
4  end
5
6  if #bindata >= 7 then
7      -- it is a response
8      local value = string.unpack( "<f", bindata, 3 )
9      box.text{caption="Value",
10             text=string.format(GetFunctionFormat(bindata:byte(2)),
                   value),
11             bg=bc,fg=fc}
12 end
```

Mit Ausnahme des `"%.2f%%"` sollte der Code eigentlich selbsterklärend sein. Das Prozentzeichen wird bei Formatanweisungen als Platzhalter für die übergebene Variable benutzt. Wenn Sie das Prozentzeichen mit ausgeben möchten (hier im Falle der Luftfeuchtigkeit) müssen Sie die Platzhalterbedeutung durch ein weiteres Prozentzeichen ausschalten, d.h. einfach gesprochen zwei Prozentzeichen im Formatstring angeben.

Prüfsumme validieren

Unsere Einführung in den neuen Template Mechanismus ist damit fast am Ende. Zu guter Letzt wollen wir noch die Prüfsumme der einzelnen Telegramme validieren. Dabei sollen korrekte Prüfsummen grün (für OK) und fehlerhafte in einem hellen Orange ausgegeben werden.
Die Prüfsumme in unserem Protokoll wird einfach gebildet durch Addieren alle Bytes beginnend mit dem ersten Adressbyte und mit dem letzten Datenbyte endend. Der Doppelpunkt sowie das CRLF werden nicht berücksichtigt. Die

Prüfsumme ist auf 8 Bit beschränkt, Überläufe bei der Addition werden deshalb ignoriert.
Die Funktion zur Bildung der Prüfsumme sieht wie folgt aus:

```
1  function Checksum( data )
2      local sum = 0
3      for i=1,#data do sum = sum + data:byte( i ) end
4      -- discard the carries
5      return sum % 256
6  end
```

Die Checksum Funktion wird aufgerufen mit der zu prüfenden Bytefolge und liefert den niederwertigen 8 Bit Wert der Summe zurück.

```
1  local chksum = Checksum( tg:string():sub( 2, -5 ) )
```

Der startende Doppelpunkt ist nicht Teil der Prüfsumme. Wir übergeben deshalb die Position 2 als Substring Startwert. Zudem müssen die Prüfsumme selbst (2 Bytes) und das CRLF (ebenfalls 2 Bytes) ausgeschlossen werden. D.h. die Substring Endposition ist 4 Byte weniger als die Telegrammlänge bzw. das fünfte Byte gezählt von hinten.
Abschließend vergleichen wir die kalkulierte Prüfsumme mit der im Telegramm enthaltenen und geben entweder eine 'gute' Checksum Box oder eine 'schlechte' Checksum Box aus.

```
1  local chksum = Checksum( tg:string():sub( 2, -5 ) )
2
3  if chksum == bindata:byte( -1 ) then
4      box.text{ caption="Chksum", text=string.format("%02X",chksum),
5                fg=0xEEEEEE, bg=0x00aa00 }
6  else
7      box.text{ caption="Chksum failed!", text=string.format("%02X",
           chksum),
8                fg=0xcc0000, bg=0xffcc00 }
9  end
```

Das komplette Beispiel mit allen hier gemachten Ergänzungen finden Sie als `complete-sample.msbtml` im `examples\ProtocolView` Verzeichnis. Die Beispielaufzeichnung enthält zudem zur Demonstration unserer Checksum Validierung eine falsche Prüfsumme im dritten Antworttelegramm.

Benannte Parameter

Hier noch ein paar ergänzende Worte zu der Übergabe von Parametern in Lua. Sie werden sicher bemerkt haben, dass einige Funktionen mit folgender Parameter Konvention aufgerufen wurden:

```
parametername = value
```

Dies ist nicht Lua-typisch, aber in vielerlei Hinsicht mehr als hilfreich. Gerade bei Funktionen mit einer ganzen Reihe von Parametern trägt dieser Ansatz deutlich zur besseren Lesbarkeit bei. Und: Sie müssen sich keine Gedanken über eine etwaige Aufrufreihenfolge der Funktionsargumente machen. Ein Beispiel:

```
1  box.block( "Func", "Command", 0xFFAAAA, 0x0000FF )
```

Ohne eine Blick ins Handbuch ist die Bedeutung der einzelnen Parameter nicht so ohne weiteres ersichtlich. Was ist die Überschrift, was der Text oder die Textfarbe?
Hier der gleiche Aufruf allerdings mit benannten Parametern:

```
1  box.block{ caption="Func", text="Command", fg=0xFFAAAA, bg=0x0000FF }
```

Die Bedeutung ist offensichtlich, auch wenn Sie sich merken müssen, das fg für die Vordergrund/Textfarbe (foreground) und bg für die Hintergrundfarbe (background) steht.
Beachten Sie, dass benannte Parameter im Gegensatz zu normalen Funktionsargumenten immer in geschweiften Klammern `{...}` stehen müssen, da Lua diese als übergebene Tabelle auswertet. Der korrekte Aufruf lautet eigentlich `function({...})`, aber die äußeren Klammern sind in diesem Fall optional und Sie können Sie weglassen.

14.4 Filterung

Die Filterung der angezeigten Telegramme ist ein oft verlangtes Feature. Aber wie sollen Telegramme mit unbekanntem Protokoll gefiltert werden? Aus der Sicht des Protokollmonitors macht eine vordefinierte Liste von Filtern keinen großen Sinn, da jeder Protokolltyp eigene Bedingungen daran stellt, was angezeigt (gefiltert) und was in der Ausgabe unterdrückt werden soll.
Um ein Beispiel zu geben: Sie haben einen Feldbus und wollen nur die Telegramme eines bestimmten Teilnehmers (mit einer spezifischen Geräteadresse) sehen. Oder es interessieren Sie nur bestimmte Telegrammtypen (Funktionsnummern etc.). In beiden Fällen muss der Filtermechanismus in der Lage sein, die Adresse oder Funktionsnummer aus den Telegrammen zu extrahieren und mit den vorgegebenen Filterparametern vergleichen.
Auf Grund der Protokollvielfalt wird schnell klar, das der Filtermechanismus Teil des Templates sein muss.

14.4.1 Anzeigen und verbergen kompletter Telegramme

Damit kommen wir zum letzten Parameter in der `split` Funktion. Als Erinnerung hier noch einmal der Aufruf der Funktion:

```
1  function split(data,inval,alter,str,filter)
2      -- you split code
3      return STATE
4  end
```

Der `filter` Parameter ist einfach ein Textstring der dem aktuell ausgewählten Eintrag der Filtereingabe in der Werkzeugleiste entspricht. Er ermöglicht damit beliebige Eingaben als Text an die `split` Funktion zu übergeben.
Warum die Filterung in der `split()` Funktion und nicht einfach in der Ausgabe?
Das Telegramm Ausgabefenster behandelt NUR die Telegramme die gerade im Fenster sichtbar sind. Es kann keine Telegramme entfernen (ausfiltern) ohne dabei eine Diskrepanz zwischen wirklich vorhandenen und angezeigten Telegrammen zu erzeugen. So können Sie z.B. alle Telegramme ausfiltern, nichts desto trotz sind diese 'intern' vorhanden und die vertikale Bildlaufleiste (Scrollbar) macht Ihnen das mehr als deutlich indem Sie dann durch eine vermeintlich

leere Telegrammliste scrollen können.
Es gibt Anwendungsfälle für die Filter Eingabe, die die Darstellung der Telegramme und damit die `out()` Funktion betrifft. Diese sind nicht zu verwechseln mit der 'Filterung' bestimmter Telegramme aus dem Datenstrom, da sie sich nur auf die aktuell sichtbaren Telegramme im Fenster bezieht. Wir kommen im nächsten Abschnitt darauf zurück.

Der Filtermechanismus ist am besten - wie üblich - an einem Beispiel erklärt. Laden Sie das Tutorial Projekt erneut und wählen Sie im Protokollmonitor das Template 'Tutorial-Complete'. Kopieren Sie das Template mit dem [+] Knopf, damit Sie es als neues Template editieren können.

Die Tutorialaufzeichnung enthält die Kommunikation eines Masters mit zwei Geräten. Die Geräte haben die Adressen 1 und 2 (nur als Erinnerung).
Und nun stellen Sie sich vor, Sie könnten einfach nur die Kommunikation zwischen dem Master und dem ersten Gerät sehen oder nur alle Telegramme die eine bestimmte Anfrage, z.B. nach der Temperatur betreffen.
Bestimmte Telegramme zu filtern heißt, die unerwünschten Telegramme aus der internen Liste des Protokollmonitors zu entfernen. Dafür bietet die `split` Funktion den besonderen Rückgabestatus REMOVED. Zunächst wollen wir nur die Telegramme darstellen, die an das erste Gerät bzw. von diesem gesendet werden (Geräteadresse 1). Um dies zu gewährleisten müssen wir alle Telegramme mit einer abweichenden Adresse unterdrücken. In unserem Fall reicht es, alle Telegramme mit der Adresse 2 zu entfernen. Die Adresse ist als zwei Hex ASCII Zeichen im zweiten und dritten Byte des Telegramms kodiert. Im folgenden das allererste Telegramm wie es im Datenmonitor dargestellt wird.

3A	30	32	30	32	43	43	0D	0A	:0202C4..

Wir erreichen dies, indem wir einfach nach der Zeichenkette "02" an der 2ten Position im Telegramm suchen. Oder in Lua formuliert:

```
1  if str:find("02") == 2 then ...
```

Der Parameter `str` repräsentiert alle bis dato empfangenen Zeichen des Telegramms als String und ist damit ein idealer Kandidat für Lua's string Modul. Diese Abfrage ist damit jedesmal wahr, wenn es sich um ein Telegramm an bzw. vom 2ten Gerät handelt. Da wir diese nicht sehen wollen, geben wir REMOVED zurück (siehe Zeile 3 im folgenden Listing).

```
1  function split( data, intval, alter, str, filter )
2      if data == 58 then return STARTED end
3      if str:find("02") == 2 then return REMOVED end
4      if str:find("\r\n") then return COMPLETED end
5      return MODIFIED
6  end
```

Wenn Sie die `split` Funktion angepasst haben drücken Sie F5 (oder klicken Sie auf den Ausführungsknopf in der Werkzeugleiste) und - voila - alle nun angezeigten Telegramme betreffen nur noch die Kommunikation zwischen Master und dem ersten Gerät.
Wenn Sie jetzt in Zeile 3 den String "02" mit "01" ersetzen werden nur noch die Telegramme des zweiten Gerätes gezeigt.

Allerdings ist es ziemlich mühselig, jedesmal das Template zu editieren um zwischen den verschiedenen Telegrammansichten hin- und herzuwechseln. Hier kommt das Filtereingabefeld in der Werkzeugleiste ins Spiel.

Weiter oben haben wir bereits die Verbindung zwischen dem Filtereingabefeld und der `split` Funktion angedeutet. Die aktuelle Eingabe im Filter Control wird der split Funktion per Parameter `filter` zur Verfügung gestellt. Wir müssen nur den Suchstring "02" in Zeile 3 durch den filter Parameter ersetzen wie im nachfolgenden Code gezeigt.

```
1  function split( data, intval, alter, str, filter )
2      if data == 58 then return STARTED end
3      if str:find(filter) == 2 then return REMOVED end
4      if str:find("\r\n") then return COMPLETED end
5      return MODIFIED
6  end
```

Ohne eine gültige Filtereingabe (z.B. einem leeren Eintrag) oder einer nicht passenden Geräteadresse wird die REMOVED Bedingung immer ignoriert und das Telegramm in der Anzeige dargestellt. Sobald aber der im Filter Control eingegebene Text im Telegramm an Position 2 gefunden wird, kehrt die `split` Funktion mit REMOVED zurück und das Telegramm mit der entsprechenden Adresse wird unterdrückt.
Sie können das leicht prüfen sobald Sie die Änderungen in Ihrem Template übernommen haben. Geben Sie im Filter Control 01 ein und drücken Sie Enter. Alle Telegramme mit Adresse 1 verschwinden aus dem Telegrammfenster. Ändern Sie nun die Eingabe auf 02 und schließen Sie die Eingabe mit Enter ab. Alle nun angezeigten Telegramme haben die Adresse 1.

Lassen Sie uns unser Beispiel noch ein wenig ausbauen und einen Filtermechanismus für die drei verschiedenen Funktionsnummern implementieren.

1 Temperature
2 Moisture
3 Pressure

Die Funktionsnummer wird als Hex ASCII in den Bytes 4 und 5 des Telegramms übertragen.
Der Anwender sollte im Filter Control die Funktion auswählen können, die er sehen möchte. (Im Beispiel mit den Geräteadressen war das genau umgekehrt). D.h. im Falle einer Eingabe von 1 (Temperatur) sollen alle Telegramme bezüglich Feuchtigkeit oder Druck per REMOVED aus der Anzeige verschwinden.

```
1  function split( data, intval, alter, str, filter )
2      if data == 0x3A then return STARTED end
3      if filter == "1" then
4          if str:sub( 4, 5 ) == "02" or str:sub( 4, 5 ) == "03" then
5              return REMOVED
6          end
7      elseif filter == "2" then
8          if str:sub( 4, 5 ) == "01" or str:sub( 4, 5 ) == "03" then
9              return REMOVED
10         end
```

```
11      elseif filter == "3" then
12          if str:sub( 4, 5 ) == "01" or str:sub( 4, 5 ) == "02" then
13              return REMOVED
14          end
15      end
16      if str:find( "\r\n") then return COMPLETED end
17      return MODIFIED
18  end
```

Der obige Code filtert alle Telegramme mit Ausnahme der im Filter Control eingegebenen Nummer aus. Aber manchmal ist es schwer sich an die richtige Nummer zu erinnern. Vor allem dann, wenn das Protokoll eine weitaus größere Zahl von Funktionsnummern unterstützt.

Wir werden deshalb im nächsten (und letzten Schritt) das Filtern noch etwas vereinfachen, indem wir das Filter Control mit einer Liste von Auswahlmöglichkeiten vorbesetzen. Dazu werden wir eine weitere Funktion neben `split` und `out` einführen. Diese spezielle `filters` Funktion erlaubt Ihnen die Filtereingabe in der Werkzeugleiste mit beliebigen Einträgen zu füllen. Die Definition dieser Funktion ist simpel:

```
1   function filters()
2       return "Show all,Temperature,Moisture,Pressure"
3   end
```

Die `filters` Funktion muss immer einen einzelnen Textstring zurückgeben, wobei jeder auszuwählende Eintrag durch ein Komma getrennt ist. Der erste Eintrag erscheint als ganz oben, der letzte ganz unten in der Auswahlliste des Filter Controls.

Sie können die `filters` Funktion an beliebiger Stelle des Skripts platzieren, nicht aber innerhalb einer anderen Funktion. Wir empfehlen diese am Anfang des Templates einzugeben.
Die Einträge in dem zurückgelieferten String erscheinen in dem Filter Control sobald der interne Lua Interpreter das Skript ausgeführt hat. In unserem Fall sind es die Einträge: Show all, Temperatur, Moisture und Pressure.

Jetzt müssen wir nur noch die REMOVED Bedingungen dahingehend ändern, dass wir an Stelle der Funktionsnummern die in `filters` definierten Einträge verwenden. (Sie finden das komplette Template im Examples Verzeichnis als `tutorial-complete-with-filtering.msbtml`).

```
1   function split( data, intval, alter, str, filter )
2       if data == 0x3A then return STARTED end
3       if filter == "Temperature" then
4           if str:sub( 4, 5 ) == "02" or str:sub( 4, 5 ) == "03" then
5               return REMOVED
6           end
7       elseif filter == "Moisture" then
8           if str:sub( 4, 5 ) == "01" or str:sub( 4, 5 ) == "03" then
9               return REMOVED
10          end
11      elseif filter == "Pressure" then
12          if str:sub( 4, 5 ) == "01" or str:sub( 4, 5 ) == "02" then
13              return REMOVED
14          end
```

```
15      end
16      if str:find( "\r\n") then return COMPLETED end
17      return MODIFIED
18  end
```

Mit den entsprechenden Anpassungen in den Zeilen 3, 7 und 11 im obigen Listing ist der Anwender nun in der Lage, die von ihm gewünschten Telegrammtypen einfach per Mausklick aus der Filtereingabe auszuwählen. Und da der Filtermechanismus immer Teil eines Protokoll-Templates ist kann jedes Protokoll mit exakt den Filtermöglichkeiten ausgestattet werden, die es erfordert.

14.4.2 Zwischen verschiedenen Telegramm Darstellungen wählen

Wie bereits zuvor erwähnt. Es macht keinen Sinn, die Ausgabe bestimmter Telegramm Typen in der `out()` Funktion zu unterdrücken. Die Telegramme werden - im Gegensatz zur Filterung innerhalb der `split()` Funktion - nicht aus der internen Liste entfernt, sondern als 'leere' Zeile repräsentiert.
Allerdings kann hier das Filter Control verwendet werden, um den Anwender abhängig von der Filtereingabe zwischen verschiedenen Darstellungsarten wählen lassen zu können. Z.B. könnte bei Protokollen, die wiederum in ein anderes Protokoll eingebettet sind, der äußere Protokollrahmen ein- oder ausgeblendet werden. Oder eine etwas einfachere Anwendung, die uns zurück zu unserem Beispiel führt.
Angenommen, Sie wollen bei der Anzeige der der Sensorwerte (Temperatur, Feuchtigkeit und Druck) zwischen metrischen und Angloamerikanischen Maßeinheiten wählen können. D.h je nach Auswahl im Filter Control: Die Anzeige des Druckes wahlweise in **psi** oder **bar** und die Temperatur entweder in °**C** oder °**F**. (Die Feuchtigkeit bleibt für beide Maßsysteme in Prozent).

Beginnen wir damit, die beiden Auswahl Varianten für die Filtereingabe zu definieren. Dies erfolgt analog zu unserem bisherigen Filter Beispiel mit:

```
1  function filters()
2      return "Metric,Anglo-American"
3  end
```

Die Filtereingabe wird - wie bereits bei der `split()` Funktion als neuer Parameter `filter` an die `out()` Funktion übergeben. In der `out()` Funktion können Sie die Eingabe entsprechend vergleichen und je nach Resultat unterschiedliche Ausgabevarianten zur Verfügung stellen. In unserem Fall die Ausgabe der Werte in einem metrischen Maßsystem oder umrechnen in das Angloamerikanische System.

Das folgende Code Beispiel soll Ihnen eine Vorstellung davon vermitteln, wie das Ganze funktioniert. Alle im Tutorial verwendeten Werte (Temperatur, Druck, Feuchtigkeit) werden in der Funktion `GetFunctionValue()` formatiert, d.h. Anzahl der Dezimal- und Nachommastellen sowie physikalischen Einheiten.
`GetFunctionValue()` wird mit zwei Parametern aufgerufen, einer Nummer, die den Typ definiert und der Wert selbst.
In Zeile 4 prüfen wir die Filtereingabe ('Metric' oder 'Anglo-American'). Die metrische Auswahl erfordert keine zusätzlichen Schritte da die übertragenen Werte bereits im metrischen System vorliegen. Dies entspricht dem Code im Tutorial. Die neuen Änderungen beginnen im folgenden `else` Block ab Zeile 8.

Zuerst ändern wir die physikalischen Einheiten und fügen zwei Nachkommastellen für den Druck in **psi** hinzu. Zeile 10 ist ein besonders schönes Beispiel, wie einfach und elegant in Lua auch kompliziertere Sachverhalte gelöst werden können. Hier müssen wir je nach Typ (Temperatur, Druck oder Feuchtigkeit) den Wert in das angloamerikanische Maßsystem umrechnen.
Zeile 10 definiert dazu eine sogenannte Hash-Tabelle oder assoziatives Array mit drei anonymen Funktion (Funktionen ohne Namen). Jede dieser Funktionen wird über den Typ Index aufgerufen. Dies geschieht automatisch in Zeile 18,19 und analog zur Typ anhängigen Auswahl der Formatierung. (Beides könnte man innerhalb der `convs` zusammen fassen). Vergleichen Sie dazu den Aufruf in Zeile 6 ohne Umrechnung.

```
1  function out( filter )
2      -- skip unchanged code here
3      function GetFunctionValue( number, value )
4          if filter == "Metric" then
5              local formats = { "%.2fC", "%.2f%%", "%.2fmBar"}
6              return string.format( formats[ number ], value )
7          else
8              -- conversion factor for Temparature, Moisture and Pressure
9              local formats = { "%.2fF", "%.2f%%", "%.2fpsi"}
10             convs = {
11                 -- function to convert celsius to fahrenheit
12                 [1] = function(c) return c * 1.8 + 32 end,
13                 -- the mositure is always in percent
14                 [2] = function(m) return m end,
15                 --  function to convert mbar to psi
16                 [3] = function(p) return p * 0.01450377 end
17             }
18             return string.format( formats[ number ],
19                                   convs[number](value) )
20         end
21     end
```

Ersetzen Sie die im Tutorial Template vorhandene `GetFunctionValue()` durch die modifizierte Version, fügen Sie die neue `filters()` Funktion hinzu und ändern Sie `out()` zu `out(filter)`. Nach Ausführen des geänderten Templates können Sie nun zwischen den beiden Maßsystem hin- und herschalten.

14.5 Neuer Filter Mechanismus

Vor der Veröffentlichung der Beta Version 6.1.0 musste eine Telegramm Filterung ausschließlich in der `split()` Funktion erfolgen. Dieser Ansatz hat allerdings einige gravierende Nachteile:
Zum einen kann ein Telegramm nur dann ausgefiltert bzw. entfernt werden (durch Rückgabe von REMOVED), solange das aktuell bearbeitete Datenbyte in `split(data,...)` Teil des zu entfernenden Telegramms ist. Sobald das an `split()` übergebenes Datenbyte zu einem neuen Telegramm gehört, kann auf das vorherige Telegramm nicht mehr zugegriffen und dieses damit auch nicht mehr entfernt (ausgefiltert) werden. Ein kleines Beispiel soll dies näher erläutern:
In Modbus RTU werden Telegramme durch eine Sendepause der Länge von 3.5 Bytes separiert, die sogenannte Interframe-Idle-Time. Ist der zeitliche Abstand des aktuellen Bytes zum vorherigen Byte größer als diese Idle Time (siehe Zeile 2 im folgenden Code), wird es in der `split()` Funktion als Start einer

neuen Telegramm Sequenz gewertet. Hier die entsprechende split Regel:

```
1  function split( data, intval, alter, str )
2      if intval > transmission.bytepause(3.5) then
3          -- data is the first byte of the next telegram
4          return STARTED
5      end
6      return MODIFIED
7  end
```

Stellen Sie sich nun vor, Sie möchten alle Telegramme mit einer falschen Prüfsumme herausfiltern. Ungültige Prüfsummen sind meistens ein Zeichen tiefer liegender Fehler in einer Modbus Übertragung und deshalb sehr ernst zu nehmen. Um dies zu erreichen müssen Sie alle Telegramme mit einer korrekten Prüfsumme verbergen bzw. entfernen.
Modbus RTU spezifiziert eine CRC16 Prüfsumme (Cyclic Redundancy Check aller Telegramm Bytes als 16 Bit Wert), die am Ende eines Telegramms als zwei aufeinander folgende Bytes angehängt wird. Die Details tun hier nichts zur Sache. Wichtig ist allein die Tatsache, dass Sie die Prüfsumme erst dann validieren können, wenn das Telegramm in der `split()` als vollständig erkannt wurde. Und genau hier liegt das Problem!

Der einzige Weg um ein vollständiges Telegramm zu erkennen ist, wenn das erste Byte des Folgetelegramms an die `split()` Funktion übergeben wurde[6].
Da Sie in der `split()` aber nicht gleichzeitig ein REMOVED für das vorherige Telegramm und ein STARTED für das aktuelle Datenbyte zurückgeben können, ist die Situation unlösbar.
Und es gibt weitere Szenarien die die Grenzen der aktuellen Filter Implementierung zeigen. Zum Beispiel: Sie wollen nur unbeantwortete Modbus Requests seitens des Masters sehen. Um dies zu realisieren müssen Sie alle Telegramm Paare entfernen, die aus einer Anfrage (Request) und korrekter Antwort (Response) bestehen. Leider können Sie dies aber erst dann beurteilen, wenn beide Telegramme bereits in der `split()` Funktion verarbeitet wurden. Was wiederum bedeutet: Sie müssten bereits zuvor verarbeitete Telegramme per REMOVED entfernen - was nicht funktioniert!
Noch ein letztes Beispiel bevor wir uns dem neuen Filter Mechanismus widmen - Modbus Exceptions:
Die Modbus Spezifikation schreibt vor, dass ein Slave mit einer Exception antwortet, wenn er die Anfrage des Masters aus irgendwelchen Gründen nicht bedienen kann. Typischerweise Dinge wie nicht implementierte Funktionalität oder ungültige Registerzugriffe etc. In solchen Fällen ist es für eine Analyse extrem wichtig, nur die Master Anforderungen mit der Exception Antwort zu sehen und alle anderen Telegramme zu verbergen.
Aber auch hier müssen Sie zunächst die Komplettierung der Antwort abwarten, bevor Sie entscheiden können, ob die Antwort keine Exception ist und damit BEIDE (der vorherigen Request sowie die gültige Antwort) entfernt werden können!

Zentrale Fähigkeiten des neuen Filter Mechanismus sind deshalb:

[6] Sie können natürlich die Länge eines Telegramms durch Auswertung des Telegramm Inhalts ermitteln. Dies ist aber sehr zeitaufwendig und funktioniert nicht bei beschädigten oder ungültigen Telegrammen.

1 Sie arbeiten ausschließlich mit kompletten Telegrammen und müssen sich nicht länger um das Splitten des Datenstroms in einzelne Telegramme kümmern. Dies ist weiterhin Aufgabe der `split()` Funktion.

2 Sie können auf alle bis dato empfangenen Telegramme zugreifen. D.h. Sie können nicht nur das zuletzt empfangen Telegramm sondern auch die davor untersuchen und entfernen.

Der neue Mechanismus ist als eine einzelne Funktion `split_complete(no)` realisiert und und wirkt sich auf ältere Protokoll Templates in keiner Weise aus.

```
1 table function split_complete( no )
2     -- query direction, data sequence and time of the last telegram
3     local dir, seq, time = sequences.get( no )
4     -- examine the telegram content
5     ...
6 end
```

Die Funktion `split_complete` wird jedes mal aufgerufen, wenn ein Telegramm von der `split()` Funktion als komplett markiert wurde. Sie brauchen sich nicht mehr selbst darum zu kümmern (insbesondere weil es teilweise recht aufwendig sein kann, wie wir im vorherigen Abschnitt gesehen haben). Gleichzeitig wird die `split()` wieder auf die ursprüngliche Aufgabe reduziert - nämlich den Datenstrom in einzelne Telegramme aufzuteilen.
Der übergebene Parameter `no` enthält die Telegramm Nummer des zuletzt komplettierten Telegramms. Er fungiert als Index, um nicht nur das letzte, sondern auch auf früher empfangene (wohlgemerkt alles komplette) Telegramme zugreifen zu können. Dies beinhaltet die Telegramm Richtung (Ursprung), den Telegramm Inhalt (als Lua String) sowie die Telegramm Zeit (Zeitstempel des ersten Telegramm Bytes). Ein Zugriff auf das aktuelle (zuletzt komplettierte) Telegramm gelingt einfach per:

```
1 local dir, seq, time = sequences.get( no )
```

Der genaue Wert von `no` ist unerheblich - er ändert sich permanent im Laufe einer Aufzeichnung. Wichtig ist, dass `no` immer die Telegramm Nummer des letzten komplettierten Telegramms enthält und $no - 1$ das des vorletzten Telegramms usw. Es ist - natürlich - Aufgabe des Template Programmierers dafür zu sorgen, dass der Ausdruck $no - n \geq 0$ niemals ungültig ist!

Bislang haben wir noch nicht darüber gesprochen, wie nun genau unerwünschte Telegramme in der `split_complete()` Funktion entfernt werden. Der neue Mechanismus wurde u.a. entworfen, um nicht nur einzelne, sondern auch einen Bereich von Telegrammen (z.B. Anfrage/Antwort Paare) zu entfernen. Der Rückgabewert dieser Funktion ist deshalb eine Lua Tabelle mit zwei Werten `{from,to}`.

Dies lässt sich am besten mit einigen 'realen' Beispielen erklären:
Als erstes wollen wir alle Telegramme in einer Modbus Übertragung/Aufzeichnung entfernen - mit Ausnahme eines ganz bestimmten Busteilnehmers (Slave) dessen Kommunikation uns interessiert. Die Addressierung eines Slaves erfolgt in Modbus durch seine Geräteadresse. Diese ist im ersten Byte eines jeden Modbus Telegramms kodiert - sowohl im Master Request als auch in der Slave Response. Dies macht es besonders einfach, da wir nicht zwischen Requests und Responses unterscheiden müssen.

Um nur die Telegramme zu sehen, die an einen bestimmten Busteilnehmer gesendet bzw. von diesem beantwortet wurden, müssen alle Telegramme mit abweichender Geräteadresse entfernt werden. Dabei reicht es, jeweils das erste Byte eines jeden Telegramms zu untersuchen. Zum besseren Verständnis hier eine vereinfachte Modbus RTU Telegramm Darstellung:

ADDR	FUNC	DATA PAYLOAD	CRC16
1 byte	1 byte	0 to 252 bytes	2 bytes

Die eigentlichen Telegramm Daten (die Datennutzlast oder Payload) interessiert uns hierbei nicht. Unser Augenmerk gilt nur der Geräteadresse, dem ersten Byte.

```
1  local addr = 5
2  function split_complete( no )
3      -- query direction, data sequence and time of the last telegram
4      local dir, seq, time = sequences.get( no )
5      -- check the address byte
6      if seq:byte(1) ~= addr then
7          return {no,no}
8      end
9  end
```

Der Filter Code ist recht einfach. In Zeile 4 fragen wir die Telegramm Informationen des aktuell komplettierten Telegramms ab und weisen den Telegramm Inhalt der lokalen Variablen `seq` zu.
In Zeile 6 vergleichen wir die Adresse mit dem in Zeile 1 vorgegebenen Wert (hier Adresse 5). Ist die Bedingung falsch, geben wir eine Lua Tabelle mit dem Bereich der Telegramm Nummern zurück, die wir entfernen wollen (Zeile 7). Da es sich nur um ein einziges Telegramm handelt, ist der Ausdruck `{form,to}` hier gleichbedeutend mit `{no,no}`.
Sie können die Funktion `split_complete(no)` auch ohne Rückgabewert verlassen. Sobald Sie aber ein oder mehrere Telegramme entfernen wollen, müssen Sie immer eine Lua Tabelle mit einem gültigen dem ersten und letzten Index der gewünschten Telegramme zurückgeben[7].
Beachten Sie, dass eine (einfache) Tabelle in Lua definiert ist als Komma separierte Aufzählung in geschweiften Klammern!

Unser nächstes Beispiel macht Gebrauch von diesem Feature. Diesmal wollen wir eine Multi-Drop-Bus (MDB) Übertragung untersuchen. Dieses Protokoll findet hauptsächlich Verwendung bei Verkaufsautomaten (vending machines). Jedes mal wenn Sie eine Münze in einen solchen Automaten werfen um sich z.B. eine Tasse Kaffee oder einen Schokoriegel zu 'ziehen', kommunizieren die internen Komponenten untereinander mit diesem Protokoll. Die Details sind dabei nicht von Bedeutung. Uns interessiert hier nur eine Besonderheit dieses Protokolls, die ein gutes Beispiel zur Entfernen zweier aufeinander folgenden Telegramme abgibt.

Das MDB Protokoll ist wie Modbus ein Master/Slave Protokoll. Der Master initiiert die Kommunikation immer mit einer Anfrage oder einem Kommando. Nicht alle Master Anfragen müssen von dem Slave (den jeweiligen Komponenten

[7] Wenn Sie z.B. nur das vorletzte Telegramm entfernen wollen, ist der Rückgabe Ausdruck `return {no-1,no-1}`

des Warenautomats) beantwortet werden. Und manchmal ist die Antwort nur eine ein Byte lange Bestätigung (Acknowledge).

Eine typischer Master Anfrage ist das POLL Kommando. Stark vereinfacht dient es dazu, die adressierte Komponente nach irgendwelchen (Zustands) Änderungen abzufragen. In den meisten Fällen ist der Status unverändert und die Komponente (der Slave) antwortet mit einem einfachen ACK, einem einzelnes $00h$ Byte mit gesetztem Mode Bit. Dazu muss man wissen, dass MDB ein 9-Bit Protokoll ist und das Parity Bit als 9tes Mode Bit verwendet wird. Das ist für unser Beispiel aber nicht weiter von Belang.
POLL Requests mit einfacher ACK Antwort nehmen einen großen Teil der Kommunikation ein. Für die Analyse wäre es deshalb sinnvoll, diese im Protokollmonitor ausblenden zu können.

Unser Ziel ist also alle POLL Anfragen des Masters mit einer einfachen $00h$ byte ACK Antwort seitens des Slaves zu entfernen. Der folgende Code macht genau das:

```
1  -- Master/Peripheral direction
2  MASTER = 1   -- Master on CH1
3  PERI = 2     -- Peripheral on CH2
4
5  function split_complete( no )
6      local dir, seq = sequences.get( no )
7      if no > 0 and dir == PERI and #seq == 1 and seq == "\x00" then
8          -- device answered with ACK, if the former telegram
9          -- is a POLL, remove both
10         local lastDir, lastSeq = sequences.get( no - 1 )
11         if #lastSeq == 2 and isPoll( lastSeq:byte(2) ) then
12             return {no-1,no}
13         end
14     end
15 end
```

Zeile 2 und 3 definieren zunächst eindeutige Namen für die Quelle/Ursprung eines Telegramms. In unserem Beispiel ist der Master mit CH1 des Analysers verbunden, alle Slaves (Komponenten oder Peripherie Geräte) mit CH2.
Mit jedem Aufruf der `split_complete(no)` Funktion weisen wir Richtung und Telegramm Inhalt den lokalen Variablen `dir` und `seq` zu, siehe Zeile 6.
In der nächsten Zeile garantieren wir, dass zumindest immer 2 Telegramme vorhanden sind ($n > 0$), um gefahrlos auf das vorletzte Telegramm $n - 1$ zugreifen zu können. Wir haben dies bereits in einem früheren Abschnitt thematisiert. In der gleichen Zeile prüfen wir zudem, ob es sich bei dem aktuellen Telegramm um ein einfaches ACK handelt (ein einzelnes Byte mit Inhalt $00h$)[8].

Wenn wahr (also ein ACK seitens des Slaves) fragen wir in Zeile 10 Richtung und Inhalt des zuvor empfangenen Telegramms (der Master Request) ab. Ein POLL Kommando besteht immer aus zwei Bytes. Slave Adresse und Kommando sind im ersten Byte kombiniert, das zweite enthält eine einfache Prüfsumme. Der eigentlich POLL Kommando Wert ist im Multi-Drop-Bus keine Konstante, sondern variiert mit dem Slave Typ. Zur Vereinfachung haben wir den Check, ob es sich um ein POLL Kommando Handelt, als eigene Funktion

[8]Wir vergleichen den Telegramminhalt seq mit einem Null Byte String, das gleiche kann man aber auch mit `seq:byte(1) == 0` erreichen.

`isPoll(...)` ausgelagert. Sie finden den vollständigen Code im MDB Template.

Wichtig ist allein, dass bei einem passenden Telegramm Paar, also POLL Request $no - 1$ gefolgt von einer ACK Response no, beide Telegramme entfernt werden, indem wir diesen Bereich als `{no-1,no}` Tabelle zurückgeben. Siehe Zeile 12.

14.6 Individuelle Filter Dialoge

Die Funktion `split_complete()` bietet eine Menge neuer Filter Möglichkeiten, um ein Protokoll Template an die eigene Anwendung anzupassen. All dies wäre aber unvollständig, wenn wir für jede Änderung der Filter Parameter den Template Code jedes Mal extra neu editieren müssten.
Hier kommt das GUI Dialog Feature des Protokollmonitors ins Spiel. Das Handbuch widmet diesem eigenes Kapitel, siehe 20, weshalb wir uns hier auf die grundlegenden Details konzentrieren. Und damit zurück zu unserem ersten Beispiel, der Filterung von Modbus Telegrammen nach Geräteadressen.

```
1  local addr = 5
2  function split_complete( no )
3      -- query direction, data sequence and time of the last telegram
4      local dir, seq, time = sequences.get( no )
5      -- check the address byte
6      if seq:byte(1) ~= addr then
7          return {no,no}
8      end
9  end
```

In unserer Lösung mussten wir zur Anpassung der Geräteadresse Zeile 1 ändern. Schöner wäre es, einen Dialog zur Verfügung zu haben, der die Einstellung der Adresse OHNE Editieren des Template Codes ermöglicht.
Die `split_complete()` Funktion wird vom selben Lua Interpreter ausgerufen, der auch für die Aufsplittung des Datenstroms per `split()` in einzelne Telegramme verantwortlich ist. Die Ausführung des Dialog Codes erfolgt jedoch von einem davon unabhängigen Interpreter. Lua Variablen müssen deshalb zwischen diesen beiden per `widgets` Umgebung bzw. Namensraum ausgetauscht werden.
Klingt zunächst kompliziert - ist es aber nicht. Im Grunde sind diese 'shared' Variablen nichts anderes als globale Variablen. Sie müssen lediglich die gewünschten Variablen dem `widgets` Namensraum zuweisen indem Sie dem Variablennamen ein `widgets.` voranstellen. Also `widgets.VARIABLE_NAME`
Der folgende Template Code soll das verdeutlichen:

```
1  -- preset the used GUI variables.
2  if not widgets.FILTER_DEVICE_ADDRESS then
3      widgets.FILTER_DEVICE_ADDRESS = 1
4  end
5  if not widgets.FILTER_DEVICE_ADDRESS_ENABLE then
6      widgets.FILTER_DEVICE_ADDRESS_ENABLE = 0
7  end
8
9  function split_complete( no )
10     -- apply address filter only when enabled via checkbox
11     if widgets.FILTER_DEVICE_ADDRESS_ENABLE == 1 then
12         -- query direction, data sequence and time of the last telegram
```

```
13          local dir, seq, time = sequences.get( no )
14          -- check the address byte
15          if seq:byte(1) ~= widgets.FILTER_DEVICE_ADDRESS then
16              return {no,no}
17          end
18      end
19  end
20
21  function dialog()
22      widgets.SetTitle( "Modbus Device Filter" )
23      -- switch on/off the address filter
24      widgets.CheckBox{ name="wxFilterDeviceAddressEnable",
25                        label="Device Address", row=1, col=1,
26                        value = widgets.FILTER_DEVICE_ADDRESS_ENABLE == 1
                              }
27      -- input the device address
28      widgets.SpinCtrl{ name="wxFilterDeviceAddress",
29                    min=1, max=255, row=1, col=2,
30                    value = widgets.FILTER_DEVICE_ADDRESS }
31  end
32
33  function apply()
34      -- filter device address (enable checkbox and address field)
35      widgets.FILTER_DEVICE_ADDRESS = widgets.GetValue(
36          "wxFilterDeviceAddress")
37      widgets.FILTER_DEVICE_ADDRESS_ENABLE = widgets.GetValue(
38          "wxFilterDeviceAddressEnable")
39      return "Reload"
40  end
```

Kapitel 20 enthält die nötigen Details. Wir beschränken uns hier auf das Wesentliche.
In den Zeilen 2-7 definieren wir die globalen Variablen, die wir zum Austausch zwischen der `split_complete()` Funktion und dem Dialog Code benötigen. Dies sind die eigentliche Geräteadresse sowie ein Flag um die Filterung nach Adressen ein/auszuschalten.
Die `split_complete()` Funktion verwendet nun für den Vergleich die mit dem Dialog geteilte Geräteadresse (Zeile 15). Zudem wird die Filterung nur ausgeführt, wenn das zugehörige Flag wahr ist (Zeile 11).
Die Funktion `dialog()` definiert - wie der Name bereits vermuten lässt - das Dialog User Interface. Zur Ein/Ausschaltung der Filterung dient eine einfache CheckBox (Zeile 24) und ein SpinCtrl ermöglicht die Eingabe einer Modbus Geräte Adresse im Bereich 1..255 (Zeile 28).
Wie bereits erwähnt: Sie finden alle Details und eine Menge weiterer Beispiele im Kapitel 20. Dort wird auch thematisiert, wie Sie die Dialog Eingaben persistent speichern können.

Zu guter Letzt die `apply()` Funktion. Diese wird immer dann aufgerufen, wenn der Anwender im Dialog den 'Anwenden' Knopf klickt. Sie ist damit zuständig, die globalen Variablen mit den Eingaben des Benutzers zu aktualisieren. Dies geschieht für die Adresse in Zeile 35 und für die Filter Aktivierung in Zeile 37. Anschließend geben wir noch den String 'RELOAD' zurück, der den Protokollmonitor veranlasst, die aufgezeichneten Daten neu einzulesen und per `split_complete()` zu filtern.

14.7 Telegramm Export

Die MSB-RS485-PLUS Software ist für die meisten Anwendungsfälle sehr gut gerüstet. Ungeachtet dessen gibt es aber immer wieder Situationen in denen Sie die durch den Analyser aufgenommenen Daten - hier die Telegramme - mit anderen Tools oder in externen Applikationen verarbeiten müssen. Z.B. in Programmen die spezialisiert sind auf spezielle Dinge die die Analyser Software naturgemäß nicht leisten kann.
Der Protokollmonitor unterstützt dank der Möglichkeit eigener, Anwender definierter Templates quasi eine unbegrenzte Anzahl von Protokollen. Jedes dieser Protokolle kommt mit ganz unterschiedlichen Daten und Telegrammstrukturen die beim Export zu berücksichtigen sind.
Gelöst wird dieser Spagat zwischen unbegrenzten Templates auf der einen und einem fest integrierten Exportdialog auf der anderen Seite durch einen intelligenten Mechanismus, der es Ihnen erlaubt IHRE Telegrammdaten in anderen Applikationen zu nutzen. In den meisten Fällen arbeitet dieser Mechanismus 'out of the box'. Nichtsdestotrotz ist es manchmal sinnvoll zu wissen, wie der Protokollmonitor die zu exportierenden Daten aus den Templates ermittelt. Vor allem dann, wenn Sie die Daten in einem Tabellenkalkulationsprogramm wie z.B. Excel oder Open Office Calc verarbeiten möchten.

14.7.1 Welche Daten werden exportiert?

In den vorangegangenen Abschnitten haben Sie alles über das zugrunde liegende Box Modell erfahren. Jede Box besteht demnach aus einer Bezeichnung (caption) und den zugehörigen Informationen bzw. Daten. Durch die Benennung einer bestimmten Information haben Sie den damit verbundenen Daten bereits einen Namen gegeben. Ein Beispiel:
Sie haben ein Telegrammfeld (d.h. eine Box) die die Geräteadresse anzeigt. Vermutlich haben Sie das Feld mit 'Address' benannt oder ähnlich. Und es ist nur logisch, das Sie diese Information (die Geräteadressen) unter dem gleichen Namen exportieren wollen.
Die Zuweisung `caption="Feldname"` im Template wird damit zum elementaren Bestandteil des Exportmechanismus. Jedesmal wenn Sie den Exportdialog öffnen extrahiert das Programm alle Boxbezeichner (caption) und speichert diese in einer internen Liste. Die im Einstelldialog aktivierten Präfix Informationen stehen dabei am Anfang der Liste. Diese Liste wird Ihnen im Exportdialog präsentiert und Sie können aus dieser alle oder einen Teil für den Export auswählen.
Mit Ausnahme der Präfixe sind alle Exportfelder alphabetisch geordnet. Dies liegt daran, das die Reihenfolge der `caption="..."` Zuweisungen im Template nichts über die spätere Feldposition in der Telegrammdarstellung aussagt.
Der eigentliche Exportprozess verläuft ähnlich der Telegrammdarstellung und ist nur abhängig vom gewählten Exportformat.

1 **Export as CSV**

2 **Export as HTML**

3 **Export as Text**

4 **Export as Latex**

Alle im Telegrammfenster ausgewählten Telegramme werden dem Lua Interpreter zugeführt. Die Skript Engine schreibt die Daten gemäß den Feldbezeichnungen in die richtige Spalte einer CSV (Comma Separated Values) Datei oder generiert einen HTML Tabelleneintrag mit exakt den gleichen Text- und Hintergrundfarben wie sie in der Telegrammanzeige verwendet werden. Nicht ausgewählte Telegramme werden im Export nicht berücksichtigt.
Beachten Sie: Jedes der Exportformate dient einem unterschiedlichen Zweck. In einer CSV Datei repräsentiert jedes definierte Telegrammfeld (jede Box) eine eigene Spalte, wobei nicht jedes Telegramm auch alle Felder besitzt. Beispielsweise können bestimmte Telegramme zusätzliche Datenfelder enthalten, während andere - kurze Telegramme - nicht mehr als ein Acknowledge darstellen. Deshalb werden alle Felder (Spalteneinträge) die nicht im aktuellen Telegramm enthalten sind mit einem leeren String "" versehen.
Der HTML Export dient hauptsächlich zu Dokumentationszwecken. Jedes Telegramm sollte genau so dargestellt werden, wie es im Protokollmonitor angezeigt wird. Dabei wird für jedes ausgewählte Telegramm eine HTML Tabelle erzeugt die alle Felder des Telegramms umfasst - sofern diese im Exportdialog zuvor ausgewählt wurden.

14.7.2 Der Exportdialog

Bevor Sie einen Export starten müssen Sie zuerst die gewünschten Telegramme auswählen. Ohne einen ausgewählten Bereich von Telegrammen wird das Telegramm unter der aktiven Cursorposition verwendet. Den Exportdialog öffnen Sie mit Strg+E oder mit Klick auf den Exporteintrag im Dateimenü.
Das Exportdialog Fenster präsentiert Ihnen eine Liste aller verfügbaren Telegrammfelder sowie der eingeschalteten Präfixe. Jeder Eintrag ist per Default aktiviert. Sie können jeden Eintrag in der Liste individuell für den Export aus bzw. einschalten. Oder aber sie (de)selektieren alle Einträge in einem Rutsch mit Klick auf den entsprechenden Knopf unter der Liste.
Das vorgegebene Exportformat ist CSV (Comma Separated Values). Sie können aber genauso gut HTML als Exportformat wählen.
Der Export startet sobald Sie einen gültigen Dateinamen eingegeben bzw. den Vorschlag des Dialog (`telegrams` mit der entsprechenden Endung `csv` oder `html`) übernommen haben. Sie können den Export jederzeit beenden indem Sie den 'Abbruch' Knopf links neben der Fortschrittsanzeige in der Statuszeile klicken. Da der Export 'parallel' zum Programm verläuft müssen Sie auch während eines längeren Exports die Arbeit mit der Analyse der Telegramme nicht unterbrechen.

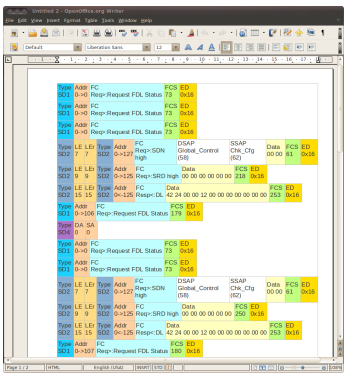
14.7.3 Export als CSV Datei

Stellen Sie sich vor Sie müssten die maximale Zeit zwischen einer Anfrage und einer Antwort herausfinden. Oder Sie benötigen eine Statistik der Häufigkeit ganz bestimmter Telegramm Typen. Diese und ähnliche Fragestellungen kann Ihnen ein Tabellenkalkulationsprogramm wie z.B. Microsoft Excel® oder Open Office Calc weitaus besser beantworten.
Der Protokollmonitor bietet deshalb einen einfachen Weg um alle angezeigten Telegramm Informationen als CSV Datei mit Komma getrennten Werten zu exportieren.
Alle Spalteneinträge sind dabei in Anführungszeichen eingeschlossen und können durch die meisten Tabellenkalkulationsprogramme gelesen werden. Die

Kopfzeile der CSV Datei enthält die Spaltennamen wie sie aus den `caption="..."` Zuweisungen in dem Template extrahiert wurden.
Wenn Sie nur an wenigen Telegrammdaten interessiert sind deaktivieren Sie alle unnötigen Felder im Exportdialog. Die erzeugte CSV Datei wird dadurch kleiner, übersichtlicher und die Dauer des Exportprozesses reduziert.

14.7.4 Export als HTML

Der HTML Export ist hauptsächlich für Dokumentationszwecke gedacht. Der Protokollmonitor exportiert alle ausgewählten Telegramme als ein gültiges HTML Dokument wobei jedes einzelne Telegramm durch eine HTML Tabelle repräsentiert wird. Die Darstellung der Tabelle entspricht exakt der Telegrammansicht, d.h. einschließlich der Text- und Boxfarben.
Die meisten Textverarbeitungsprogramme sind in der Lage HTML Dateien zu importieren. Open Office Anwender können die exportierte HTML Datei einfach per Drag And Drop in Ihrem Dokument einfügen (siehe Bild).



Open Office Writer mit HTML Telegramm Export

14.7.5 Export als Text

Die Ausgabe der ausgewählten Telegramme als Text unterscheidet sich insofern von einem CSV Export, als das nur die im jeweiligen Telegramm vorhandenen Felder (Boxen) berücksichtigt werden. Die einzelnen Telegrammfelder werden dabei als `Label(VALUE)` Ausdrücke in die Exportdatei geschrieben.
Diese Art von Export ist - je nach weiter führender Applikation - leichter zu verarbeiten und empfiehlt sich vor allem bei Dokumentation/Report Tools die nur reine Texteingaben erlauben. Hier ein beispielhafter Auszug eines Textexport aus einer Modbus RTU Aufzeichnung:

```
Src(Master) Dest(1) Fnc(Read Coils) Desc(Addr=0, Quantity=10) Cks(0DBC)
Src(1) Dest(Master) Fnc(Read Coils) Desc(Byte count=2) Data(00 00) Cks(FCB9)
```

14.7.6 Export als Latex

Diese Exportart ist hauptsächlich für Anwender gedacht, die LaTeX als Textverarbeitung bzw. zur Dokumentation verwenden. Jedes ausgewählte Telegramm wird dabei als `tabular` Umgebung ausgegeben. Die einzelnen Telegrammfelder entsprechen einer Tabellenspalte (Zelle) und sind in den aktuellen Telegrammfarben gefärbt. Dazu müssen die folgenden LaTeX Pakete `color`, `colortbl` and `xcolor` am Anfang des LaTeX Dokuments eingebunden werden. Diese Pakete sollten in jeder LaTeX Distribution vorhanden sein bzw. leicht nachinstallierbar sein. Folgendes Kommando bindet die erforderlichen Pakete ein:

```
\usepackage{ color , colortbl , xcolor }
```

Einzelne Telegrammfelder können vor dem eigentlichen Export einzeln ein/ausgeschaltet werden. Sie können allerdings ebenso leicht später in der eigentlichen LaTeX Tabelle noch editiert werden.

14.7.7 Ein paar besondere Hinweise zur Feld- oder Boxbezeichnung

Wir erwähnten bereits: Die Exportfelder (Spalten in CSV) werden durch die `caption="..."` Zuweisungen benannt. Solange Sie reinen Text und eindeutige Bezeichnungen verwenden wird das Resultat dem entsprechen was Sie erwarten.

Aber betrachten Sie einmal folgende Box mit einem zusammengesetztem Feld bzw. Boxnamen aus Text und Funktionsnummer. (Die genaue Beschreibung der Funktion wird dabei als Box-Textvariable übergeben).

```
1  box.text{caption="Fnc (".. tg:data(2) ..")", text=GetFuncDesc(tg:data(2))
       }
```

Die Anzeige des Telegramms könnte wie folgt aussehen:

Func (8) This is function 8

Leider kann der Exportmechanismus nicht die komplette Feldüberschrift (caption) zusammenbauen. Dafür müsste er das Template auf alle vorhandenen Telegramme anwenden BEVOR der eigentliche Export starten kann.
Hinzu kommt: Ein solche Bezeichnung mag für die Darstellung im Protokollmonitor ideal sein, führt aber beim CSV Export zu einer verwirrenden Anzahl von Spalten (eine Spalte für jede unterschiedliche Funktionsnummer). Hier hätten Sie vermutlich gerne nur eine Spalte 'Funktionsnummer' - richtig?
Zur Erinnerung: Der Exportmechanismus durchsucht das Template nach Zuweisungen der Form `caption="NAME"`. Im obigen Beispiel wird der Feldname (caption) als `"Fnc ("` extrahiert und der restliche Ausdruck verworfen. Wenn der rechte Teil der Zuweisung (der NAME) bereits mit einem ausführbaren Ausdruck beginnt, z.B. `caption=tg:data(1).."-Typ"` schlägt die Suche komplett fehl und die Box-Bezeichnung bzw. der Feldname wird weder in der Liste der zu exportierenden Werten auftreten noch in der Exportdatei selbst.
Das Gleiche gilt wenn Sie Variablen für den Box/Feld-Titel verwenden.

```
1  label = "Chksum OK"
2  if ChecksumTest() == false then
3      label = "Chksum fails"
4  end
5  box.text{caption=label, text=GetChecksumByte()}
```

Auch hier wird die Suche nach einem Muster der Form `caption="..."` fehlschlagen und keins von beiden möglichen Felder in der Exportliste hinzugefügt werden.
Andere Effekte sind vielleicht weniger offensichtlich. Nichtsdestotrotz ist es wichtig, auch diese im Hinterkopf zu behalten. Der Exportmechanismus kann naturgemäß nicht unterscheiden ob eine caption Zuweisung innerhalb einer auskommentierten Funktion auftritt oder Teil eines niemals ausgeführten Codes ist. In beiden Fällen wird der Exportdialog die entsprechenden Felder zur Auswahl anbieten. Aber dies führt im schlimmsten Fall lediglich zu einer Spalte mit leeren Strings in einer CSV Datei.
Fazit: Vermeiden Sie zusammengesetzte Ausdrücke als Boxbezeichnung und verwenden Sie stattdessen nur reinen Text!

14.8 Protokollmonitor spezifische Lua Erweiterungen

Der folgende Abschnitt beschreibt alle im Protokollmonitor verfügbaren Module, Funktionen, Erweiterungen und Datentypen die nicht Bestandteil des allgemeinen Lua Sprachumfangs sind. Diese sind ausschließlich im Protokollmonitor implementiert bzw. hinzugefügt wurden. Lua bietet - natürlich - eine Menge

weiterer Module und Funktionen die wir hier aus Platzgründen nicht behandeln können.

- **box Modul :** Das box Modul ist für die Anzeige der Telegrammdaten verantwortlich.
- **debug Modul :** Mit den Funktionen des Debug Moduls können Sie spezielle Debug Informationen im Debug Fenster ausgegeben. Insbesonders bei der Fehlersuche ist dies u.U. von großem Nutzen.
- **event Modul :** Das event Modul ist nur innerhalb der split Funktion verfügbar und liefert zusätzliche Informationen zum aktuell empfangenen Datenereignis.
- **linestates Modul :** Das linestates Modul liefert Informationen über einen vorherigen Pegelwechsel sowie Anzahl der Pegeländerungen.
- **protocol Modul :** Liefert Informationen zur aktuell verwendeten Baudrate, Anzahl der Daten-, Stopbits und Parity.
- **record Modul :** Liefert Information rund um die aktuelle Aufzeichnung. Momentan ist nur die Abfrage der Startzeit sowie des aktuellen Bus-Wirings (Bus Anschluss) implementiert.
- **sequences Modul :** Das sequences Modul ist nur in der `split_complete()` Funktion verfügbar und erlaubt Ihnen den Zugriff alle bereits empfangenen und komplettierten Telegramm Sequenzen. Es ist insbesondere zur Implementierung von Telegramm Filtern gedacht.
- **shared Modul :** Der Protokollmonitor verwendet zwei unabhängige Lua Interpreter zur Aufsplittung der Telegramme, getrennt nach Datenrichtung. Das shared Modul erlaubt Ihnen, Daten zwischen diesen beiden Interpretern auszutauschen.
- **telegram Typ :** Ein Protokollmonitor spezifischer Datentyp der alle Informationen zu einem ganz bestimmten Telegramm enthält.
- **telegrams Modul :** Das telegrams Module erlaubt Ihnen den wahlfreien Zugriff auf alle bis dato empfangenen Telegramme. Das Ergebnis ist immer eine Variable vom Typ `telegram`. Es ersetzt die Module tg und tgprev, die auf den Zugriff des aktuellen und vorherigen Telegramms begrenzt waren. Das `telegrams` Modul ist nur in `out()` verfügbar.
- **widgets Modul :** Erlaubt Ihnen das Erstellen von individuellen Template User Interfaces (Dialogen) um z.B. Setup oder Filter Parameter einzustellen. Das Modul bietet zudem einen Mechanismus um Variablen zwischen dem Dialog und restlichen Template Code auszutauschen und Dialog Einstellungen persistent zu speichern. Die Details sind in einem eigenen Kapitel beschrieben, siehe 20.

Die einzelnen Typen, Funktionen und Module in alphabetischer Reihenfolge:

14.8.1 Das box Modul

Das `box` Modul enthält alle für die Ausgaben im Telegrammfenster nötigenen 'Boxen'. Eine allgemeine Textbox erlaubt die Darstellung beliebiger Informationen wie z.B. Text, Zahlen, Datensequenzen in einem farbigen Rechteck inklusive Überschrift.
Jede Box besitzt eine individuelle Text- und Hintergrundfarbe, übergeben als benannte Parameter `fg` und `bg`. Die Voreinstellung ist schwarzer Text (und Umriß) auf weißem Hintergrund.
Die 'space' Box ist eine in der Weite frei definierbare leere Box (Lücke) um aufeinanderfolgende Boxen gezielt zu separieren.

Funktion	Beschreibung
setup	Neu! Erlaubt das globale Vorbesetzen von Box Farben durch Label/Farb Paare sowie die Erhöhung der Box Textzeilen.
space	Fügt einen leeren Zwischenraum mit einer Weite definiert als Pixel oder Anzahl von Zeichen an.
text	Eine allgemeine Box mit frei definierbarem Label (Bezeichnung), Textinhalt, Vordergrund- (Text und Rahmen) und Hintergrundfarbe.

box.setup

Mit `box.setup` können Sie bestimmte box Parameter global als Vorgabe definieren. Diese Angaben überschreiben dabei alle individuellen Box Parameter, insbesondere die Box Hintergrundfarbe. Damit ist es möglich, die Farbe aller Telegramm Felder mit dem Label 'Checksum' an einer Stelle vorzugeben, ohne dies wieder und wieder angeben zu müssen.

Example

```
box.setup{
    lines=3,
    colours = {
        ["Checksum"] = 0x00FF00,
        ["Source"] = 0x80FFFF
    }
}
```

Und! Ab Version 5.0 ist die Anzahl der Textzeilen in einer Box nicht länger auf 2 Zeilen (Caption und Text) beschränkt. Möglich sind nun bis zu 4 Zeilen (Caption und bis zu 3 Zeilen Text).
Die Angabe von `lines` bezieht sich dabei auf alle Zeilen (Caption/Überschrift und Text).

Example

```
box.setup{ lines=4 }
box.text{ caption="TEST", text="line1\nline2\nline3"}
```

Beachten Sie die zwei Zeilenumbrüche `\n` in Zeile 2 im obigen Beispiel.

box.space

Fügt eine leere Box mit definierter Weite an. Ohne Angabe wird eine Breite von 10 Pixel verwendet. Die Weite (oder Breite) der Box kann als Anzahl von Pixeln oder Zeichen spezifiziert werden. Letzteres berücksichtigt die aktuelle Schriftgröße. D.h. der leere Zwischenraum 'wächst' mit der Schriftgröße (Zooming).

box.space{ *em=0, px=10* }

- **em:** Die Breite definiert als Anzahl 'M' Zeichen.
- **px:** Die Breite definiert in Pixel. Die maximale Pixelanzahl ist 100.

Example

```
1  function out()
2      -- insert a small space with the width of two space (blank)
           characters
3      box.space{ em=2 }
4      -- insert an empty space with the width of 50 pixel
5      box.space{ px=50 }
6  end
```

box.text

Zeichnet eine allgemeine Box mit individuellen Farben, Boxbezeichnung und Boxinhalt.

box.text{ *caption=STRING, text=STRING [, fg=RGB, bg=RGB], em=0, px=0* }

- **caption:** Ein Textstring als Boxbezeichnung.
- **text:** Die Boxdaten als beliebiger Textstring.
- **fg:** Optionale RGB Farbe für die Text- und Rahmenfarbe, default ist schwarz.
- **bg:** Optionale RGB Farbe für den Box Hintergrund, default ist weiß.
- **em:** Optionale Breite definiert als Anzahl 'M' Zeichen.
- **px:** Optionale Breite in Pixel. Die Vorgabe ist die Textbreite.

Beispiel

```
1  function out()
2      box.text{ caption="Caption", text="Some text", px=100,
3                fg=0xFF0000, bg=0xAADDFF }
4  end
```

14.8.2 Das debug Modul

Der Protokollmonitor besitzt ein eingebautes Debug Fenster um spezielle Zustände in Ihrem Skript anzuzeigen. Dies kann der Inhalt einer bestimmten Variable sein, oder das Resultat einer Operation. Das Debug Modul enthält hierzu

alle nötigen Funktionen um beliebige Werte oder Texte auszugeben. Zusätzlich können Sie die Debug-Ausgabe per Skript anhalten, zu einem geeigneten Zeitpunkt wieder aufnehmen und Ausgaben zusammenfassen. Z.B. bei wiederholenden Debug Meldungen. Um das Ausgabefenster für Debug Meldungen zu öffnen drücken Sie einfach:
Strg + Alt + D

Function	Description
clear	Löscht den aktuellen Inhalt im Debug Fenster.
print	Gibt die übergebenen Argumente im Debug Fenster aus. Sie können beliebig viele Argumente (Text oder Wert) per Komma getrennt im Funktionsaufruf angeben.
resume	Setzt eine zuvor unterbrochene Debug Ausgabe wieder fort.
summarize	Wenn aktiviert sammelt die Debug Ausgabe alle identischen Meldungen und gibt diese mit Angabe einer entsprechenden Wiederholungszahl aus.
suspend	Stoppt (suspendiert) die Debug Ausgabe. Alle weiteren Ausgaben werden unterdrückt bis die Ausgabe per `debug.resume` fortgesetzt wird.
timeprompt	Zeigt die aktuelle Zeit (hh:mm:ss) am Anfang jeder Debug Ausgabe. Sie können dies ein- bzw. ausschalten, indem Sie dieser Funktion true oder false übergeben.

debug.clear

Löscht den aktuellen Inhalt des Debug Ausgabefensters.

debug.clear()

Beispiel

```
1  -- a global counter holding the number of error responses
2  errorCounter = 0
3
4  function split( data, intval, alter, str, filter )
5      if alter or intval > transmission.bytepause(3.5) then
6          -- a function byte > 0x80 means an error response in Modbus
7          if #str > 1 and str:byte(2) >= 0x80 then
8              errorCounter = errorCounter + 1
9              debug.clear()
10             debug.print( "Error Counter: "..errorCounter )
11         end
12         return STARTED
13     end
14     -- all other bytes extend the current telegram
15     return MODIFIED
16 end
```

Das gezeigte Beispiel zählt alle Fehlermeldungen in einer Modbus Übertragung. Fehlermeldungen sind durch ein gesetztes MSB (höchstwertiges Bit) im Funktionswert (2tes Byte) gekennzeichnet.

Da nur die `split()` Funktion 'alle' übertragenen Datenbytes sieht, muss der Code für den Fehlerzähler in der split Funktion eingefügt werden.

debug.print

Gibt die per Komma separierten Argumente in dem Protokollmonitor internen Debugfenster aus.

debug.print(*param1*,*param2*,...)

- **param:** Durch Komma getrennte Liste der Parameter.

Beispiel

```
function out()
    local tg = telegrams.this()
    if tg:size() > 10 then
        -- output the time and size of the current telegram
        debug.print( "Time:"..tg:time(), "Size:"..tg:size() )
    end
end
```

Vermeiden Sie die intensive Nutzung von debug.print

Jede Ausgabe im Debug Fenster benötigt eine gewisse Zeit und verlangsamt deshalb die Ausführung des Template Skripts.

debug.resume

Setzt eine zuvor ausgesetzte Debugausgabe wieder fort. Im Beispiel wurde die Ausgabe nach Empfang eines Telegramms mit einem ersten Byte ungleich hex 10 gestoppt und erst mit Empfang eines Telegramms mit hex 10 als Startbyte wieder aufgenommen.

debug.resume()

Beispiel

```
function out()
    local tg = telegrams.this()
    if tg:data(1) == 0x10 then
        -- first output the debug message
        debug.print( "Data:"..tg:data(1),"Size:"..tg:size() )
        -- then suppress any other output
        debug.resume()
    else
        -- enable the debug output again
        debug.suspense()
    end
end
```

debug.summarize
Fasst identische Debug Meldungen zusammen und gibt sie erst aus, wenn eine davon unterschiedliche Meldung auftritt. Die wiederholten Meldungen werden wie folgt dargestellt:

```
THE DEBUG MESSAGE
The previous message repeated n times.
```

`n` bezeichnet die Anzahl der aufgetretenen Wiederholungen.
Gewöhnlich reicht es eine Anweisung wie `debug.summarize(true)` am Anfang des Templates zu setzen. D.h. außerhalb von `split()` und `out()`, da diese Funktion nur einmal aufgerufen werden muss (siehe Zeile 1).

debug.summarize()

Beispiel

```
1  debug.summarize( true )
2  debug.timeprompt( true )
3
4  function split( data, intval )
5      if intval > transmission.bitpause( 33 ) then return STARTED end
6      return MODIFIED
7  end
8
9  function out()
10     -- your output code...
11 end
```

debug.suspend
Unterdrückt alle weiteren Debugausgaben via `debug.print` bis ein Aufruf von `debug.resume` diese wieder freigibt. Siehe das vorherige Beispiel (resume).

debug.timeprompt
Aktiviert oder deaktiviert die zusätzliche Zeitangabe wann die Debugausgabe erfolgte. Voreingestellt ist eine Ausgabe ohne Zeit.
Wenn eingeschaltet wird jeder Ausgabe die aktuelle Zeit im Format hh:mm:ss vorangestellt. Ein Beispiel:

```
12:24:48: My debug message
```

Siehe auch Zeile 2 im vorherigen Beispiel.

14.8.3 Das event Modul
Das `event` Modul bietet innerhalb der `split` Funktion Zugriff auf zusätzliche Informationen die nicht als Parameter übergeben werden.
Bitte beachten Sie! Eine Verwendung außerhalb der `split` Funktion ist nicht erlaubt und erzeugt einen Fehler.

Funktion	Beschreibung
data	liefert das 9 Bit Datenbyte.

dir	liefert die Quelle oder Richtung des aktuellen Datenereignisses.
isbreak	liefert true wenn es sich bei dem aktuellen Datenerereignis um ein break handelt.
level	liefert den aktuellen Pegel des übergebenen Signals zum Zeitpunkt des Datenereignisses.
number	gibt die aktuelle Ereignisposition oder Nummer zurück. Gezählt wird am 0.
time	gibt den Zeitstempel des Datenereignisses in Sekunden seit Beginn der Aufnahme zurück.

event.data

Liefert den Datenwert des aktuellen Datenereignisses als 9 Bit Wert. Entspricht dem Parameter data und ist hier nur der Vollständigkeit halber aufgeführt.

event.data()

Example

```
1  function split( data, intval, alter, str )
2      -- test for LF
3      if event.data() == 0x0A then
4          return COMPLETED
5      end
6      return MODIFIED
7  end
```

event.dir

Liefert die Richtung bzw. Quelle des aktuellen Datenereignisses als eine Integerzahl mit dem folgenden möglichen Werten: 1: Port A (CH1), 2: Port B (CH2).

event.dir()

Beispiel

```
1  function split( data, intval, alter, str )
2      local eos = 13
3      if event.dir() == 2 then eos = 10 end
4      if #str == 1 then return STARTED end
5      if data == eos then return COMPLETED end
6      return MODIFIED
7  end
```

event.isbreak

Liefert true wenn es sich bei dem aktuellen Datenereignis um ein Break handelt. Breaks werden vom Analyzer als NULL Bytes empfangen. Mit `event.isbreak()` können Sie zwischen normalen NULL Bytes und echten Breaks unterscheiden.

Dies ist insbesondere bei Protokollen interessant, die ein Break als Telegramm Trenner verwenden.

event.isbreak()

Example

```
function split( data, intval, alter, str )
    if event.isbreak() then return STARTED end
    return MODIFIED
end
```

event.level

Dient zur Abfrage eines beliebigen Signalpegel, der während des Auftretens des Datenereignisses vorlag. Das zugehörige Signal wird als Nummer 1...8 (entsprechend der Anzeige im Kontroll-Programm) übergeben. Das Resultat ist einer der folgenden Werte: 1: High Pegel, -1: Low Pegel, 0: ungültiger (invalid) Pegel

event.level(*signal=NUMBER*)

- **signal:** Signal- oder Leitungsnummer (1...8)

Beispiel

```
function split( data, intval, alter, str )
    -- the RI signal marks a special one byte broadcast telegram
    if event.level(8) == 1 then return STARTED+COMPLETED end
    if #str == 1 then return STARTED end
    if data == eos then return COMPLETED end
    return MODIFIED
end
```

event.number

Gibt die Position des aktuellen Ereignisses in der Aufzeichnung/Aufnahme zurück, beginnend mit 0 für das erste Ereignis. Diese Funktion wird nützlich, wenn Sie wissen möchten, ob z.B. seit dem letzten Ereignis bestimmte Pegelwechsel aufgetreten sind. Letzteres hängt davon ab, welche Ereignisse Sie für die Aufzeichnung aktiviert haben.

event.number()

Example

```
lastNumber = 0
function split( data, intval, alter, str )
    if event.number() ~= lastNumber + 1 then
        lastNumber = event.number()
```

```
5           -- a line state event has occurred since the last call
6           return COMPLETED
7       end
8       lastNumber = event.number()
9       return MODIFIED
10  end
```

event.time

Liefert die Zeit in Sekunden wann das Datenereignis relativ zum Start der Aufzeichnung aufgetreten ist. Das Resultat ist eine Fliesskommazahl mit Mikrosekunden Genauigkeit.

event.time()

Beispiel

```
1   function split( data, intval, alter, str )
2       -- remove all telegrams in the first 5s of the record
3       if event.time() < 5.0 then return REMOVED end
4       if #str == 1 then return STARTED end
5       if data == eos then return COMPLETED end
6       return MODIFIED
7   end
```

14.8.4 Das linestates Modul

Aufgrund ihres Designs 'sieht' die `split` Funktion keine Ereignisse außer den übertragenen Daten. Bei Protokollen die den Start bzw. das Ende eines Telegramms mit einem Pegelwechsel einer anderen Leitung markieren ist aber die Information über vorherige Pegeländerungen unabdingbar. Dies wird z.B. bei sogenannten Radio RTU (Remote Terminal Units) verwendet um per RTS oder CTS das Trägersignal für die Datenübertragung ein- bzw. wieder auszuschalten. Das `linestates` Modul füllt diese Lücke.
Bitte beachten Sie! Wie bereits beim `event` Modul ist eine Verwendung außerhalb der `split` Funktion ist nicht erlaubt und erzeugt einen Fehler.

Function	Description
changed(*signo*)	liefert true wenn sich der Pegel der angegebenen Leitung (Signalnummer 1...8) seit der letzten Anfrage geändert hat.
count(*signo*)	gibt die Anzahl aller Pegeländerungen der angegeben Leitung (Signalnummer (1...8) seit Aufzeichnungsbeginn zurück.

linestates.changed

Liefert true wenn sich der Pegel der angegebenen Leitung (Signals) seit dem letzten Aufruf dieser Funktion geändert hat. Die Leitungen oder Signalnummern werden von 1 bis 8 gezählt und entsprechen der Signalreihenfolge wie

sie im Display des Kontrollprogrammes angezeigt werden.
Ein Signalwechsel wird immer dann erkannt, wenn das Signal seinen Tri-State Zustand ändert. Das schließt einen logischen Signalwechsel (high, low) als auch einen Wechsel von oder zu einem inaktiven Signalpegel ein.

linestates.changed(*signo*)

- **signo** Leitungs- bzw. Signalnummer.

Example

```
function split( data, intval, alter, str )
    local RTS = 6
    local CTS = 7
    if event.dir() == 1 then
        if event.level( RTS ) ==  1 and linestates.changed( RTS ) then
            return STARTED
        end
    else
        if event.level( CTS ) ==  1 and linestates.changed( CTS ) then
            return STARTED
        end
    end
    return MODIFIED
end
```

linestates.count

Liefert die Anzahl der Pegelwechsel der angegebenen Leitung (Signals) seit Start der Aufzeichnung. Die Leitungen oder Signalnummern werden von 1 bis 8 gezählt und entsprechen der Signalreihenfolge wie sie im Display des Kontrollprogrammes angezeigt werden.
Ein Signalwechsel wird immer dann erkannt, wenn das Signal seinen Tri-State Zustand ändert. Das schließt einen logischen Signalwechsel (high, low) als auch einen Wechsel von oder zu einem inaktiven Signalpegel ein.

linestates.count(*signo*)

- **signo** Leitungs- bzw. Signalnummer.

Example

```
rtsChanges = 0
function split( data, intval, alter, str )
    local RTS = 6
     if linestates.count( RTS ) > 5 then
         rtsChanges = 0
         return STARTED
    end
    return MODIFIED
end
```

14.8.5 Das sequences Modul

Dieses Modul ist nur innerhalb der `split_complete()` Funktion verfügbar. Es dient dazu, auf den Inhalt, die Datenrichtung und den Zeitstempel (des ersten Telegramm Bytes) bereits empfangener und komplettierter Telegramm Sequenzen zuzugreifen.

Unterschied zwischen sequences.get und telegrams.at

Es existiert noch eine andere Methode um auf empfange Telegramme zuzugreifen. Sie kennen diese aus der `out()` Funktion. Warum deshalb noch eine neue?
Erstens: `telegrams.at(index)` liefert nicht nur komplette sondern auch unvollständige Telegramme.
Zweitens: Da `sequences.get(no)` für jedes empfangene Telegramm aufgerufen wird ist es extrem optimiert um möglichst wenig Performance Verluste zu verursachen. Es liefert deshalb auch kein `telegram` Objekt zurück, sondern lediglich drei Werte (Richtung, Daten und Zeitstempel).

Function	Description
get	liefert Telegramm Richtung (Quelle), Daten (Lua String) und Zeitstempel des Telegramms mit dem angegebenen Index.

sequences.get

Liefert Richtung, Daten und Zeitstempel eines bereits empfangenen und komplettierten Telegramms. Der Zugriff erfolgt über einen Index der in Relation zur übergebenen Telegramm Nummer der `split_complete(no)` Funktion steht.

sequences.get(*index*)

- **index:** Telegramm Nummer.

Example

```
1  local addr = 5
2  function split_complete( no )
3      -- query direction, data, time of the last received telegram
4      local dir, seq, time = sequences.get( no )
5      -- check the address byte
6      if seq:byte(1) ~= addr then
7          return {no,no}
8      end
9  end
```

14.8.6 Das shared module

Der Protokollmechanismus verwendet für jede Datenrichtung einen eigenen, unabhängigen Lua Interpreter. Dadurch müssen Sie sich in der `split` Funktion keine Gedanken über die Datenquelle des übergebenen Datenbytes machen. Und auch die interne Telegramm Repräsentation (alle bislang empfange-

nen Zeichen, übergeben als `str`) beziehen sich immer auf eine Datenrichtung. Die Verwendung zweier Interpreter macht zudem die Arbeitsweise der Template Funktionen um einiges einfacher. Dies hat allerdings auch seinen Preis:
Obwohl globale Variablen auch außerhalb der `split` Funktion existieren (entweder außerhalb deklariert bzw. nicht als `local` definiert), können Sie diese dennoch nicht dazu verwenden um Informationen zwischen den beiden Interpretern auszutauschen. Oder anders gesagt:
Wenn beide `split` Funktionen, die eine aufgerufen durch Daten an Port A (CH1), die andere durch Daten an Port B (CH2), auf eine gemeinsame Variable zugreifen sollen, kommt das Modul `shared` ins Spiel.
Alle im Modul `shared` abgelegten Variablen sind von beiden Interpretern zugänglich. So können Sie z.B. dort eine Variable anlegen wenn ein bestimmtes Datenbyte an Port A empfangen wurde und den Wert dieser Variable bei der Verarbeitung von Port B Daten verwenden.

Funktion	Beschreibung
shared.get	Liefert den Wert der Variable mit dem angegebenen Namen.
shared.set	Speichert den Wert unter dem angegebenen Name.

shared.get
Liefert den Wert/Inhalt der Variable mit dem übergebenen Namen oder nil falls keine Variable dieses Namens existiert.

shared.get(*name*)

- **name:** Der Name der Variable als ein Lua String.

shared.set
Legt eine neue Variable mit dem angegebenen Namen an und weist ihr den übergebenen Wert zu. Sollte bereits eine Variable dieses Namens existieren, wird deren Inhalt mit dem neuen Wert überschrieben.

shared.set(*name, value*)

- **name:** Der Name der Variable als Lua String.
- **value:** Ein beliebiger Lua Wert (Zahl, Boolean, String).

Das folgende Beispiel verwendet ein imaginäres Protokoll. Dabei wird jedes Telegramm mit einem Doppelpunkt ':' eingeleitet und mit einem LF (Linefeed) beendet.
Stellen Sie sich nun ein spezielles 'Life Ping' Telegramm vor welches lediglich eine Antwort als 'Lebenszeichen' anfordert. In unserem Beispiel wollen wir sowohl die 'Life Ping' Anforderung als auch die zugehörige Antwort aus der Telegramm Darstellung entfernen (REMOVED). Und um das ganze noch etwas zu verkomplizieren sollen die 'Life Ping' Antworten von anderen Antworten nicht zu unterscheiden sein.
Ein 'Life Ping' Telegramm sei als leeres Telegramm spezifiziert. D.h. Das Telegramm besteht lediglich aus dem Doppelpunkt ':' gefolgt von einem LF.

Um eine 'Life Ping' Antwort von einer identischen Antwort auseinander halten zu können, müssen wir den Empfang eines 'Life Ping' in einer Variable vermerken. Im Falle der späteren Antwort können wir anhand der gespeicherten Variable den Anfragetyp unterscheiden.
Da Anfrage (Life Ping) und Antwort aus verschiedenen Datenquellen erfolgen können, ist die Verwendung des `shared` Moduls unabdingbar. Hier der entsprechende Code:

Example

```
1  function split( data, intval, alter, str )
2      if data == 58 then return STARTED end
3      if data == 10 then
4          if shared.get("LifePing") then
5              return REMOVED
6          end
7          if #str == 2 then
8              shared.set("LifePing", true )
9              return REMOVED
10         else
11             shared.set("LifePing", false )
12         end
13         return COMPLETED
14     end
15     return MODIFIED
16 end
```

Das Beispiel erscheint auf den ersten Blick ein wenig konstruiert. Es verdeutlicht aber sehr schön den Einsatz des `shared` Moduls zum Austausch von Informationen zwischen beiden unabhängig arbeitenden Lua Interpretern.
Zeile 2 löst ein neues Telegramm aus sobald ein Doppelpunkt (ASCII Wert ist 58 dezimal) empfangen wurde. Und zwar unabhängig von der Datenrichtung.
Zeile 3 prüft auf das Ende eines Telegramms indem es das aktuelle Datenbyte mit LF (dezimal 10) vergleicht. Sobald ein vollständiges Telegramm vorliegt (`data` ist 10) prüfen wir, ob es die Antwort auf ein vorheriges 'Life Ping' ist. In diesem Fall ist die gemeinsam genutzte Variable `LifePing` wahr (true) und wir können das Telegramm in Zeile 5 entfernen (REMOVED).

In Zeile 7 merken wir uns den Erhalt eines 'Life Ping' Telegrammes indem wir die Telegrammlänge mit 2 vergleichen (':' und LF). Handelt es sich um ein 'Life Ping' Telegramm wird die `shared` Variable `LifePing` auf wahr gesetzt (Zeile 8) und mit REMOVED wird das Telegramm aus der Anzeige entfernt (Zeile 9).
Bei allen anderen Telegrammlängen wird der Variable `LifePing` false zugewiesen. Da das Telegramm mit Erhalt von LF komplett ist, erfolgt in Zeile 13 die Rückgabe COMPLETED.
Alle empfangenen Datenbytes mit Ausnahme von ':' und LF werden an die interne Telegramm Repräsentation angehängt und deshalb in Zeile 15 die Funktion mit MODIFIED beendet.

Beachten Sie bitte! Der obige Code wird nicht mit normalen globalen Variablen arbeiten, da jeder der beiden Interpreter seine 'eigenen' globalen Variablen verwendet und ein Zugriff darauf durch den jeweils anderen Interpreter nicht

möglich ist. Die Verwendung des `shared` Moduls ist die einzige Möglichkeit um Daten zwischen beiden Interpretern auszutauschen.

14.8.7 Der telegram Typ

Der Lua Datentyp `telegram` ist eine Art Behälter der die Eigenschaften eines beliebigen Telegramms in der Aufzeichnung enthält. `telegram` ist immer das Resultat einer Telegrammabfrage durch das `telegrams` Modul (beachten Sie den Plural im Modulnamen).
Sie können jede Telegramm spezifische Information durch Aufruf der zugehörigen Telegrammfunktion ermitteln. Dies entspricht einem Objekt orientierten Ansatz. Eine zusätzliche dump Funktion bietet eine sehr einfache und elegante Möglichkeit, den kompletten Inhalt des Telegramms zu erfassen und darzustellen.

Funktion	Beschreibung
data	Liefert das Datenbyte an der angegebenen Position (bis zu 9 Bit). Positive Werte indizieren das Datenbyte gezählt vom Telegrammstart (Position=1 bedeutet das erste Byte im Telegramm). Negative Indexe zählen von hinten (-1 greift auf den letzten Datenwert im Telegramm zu).
datatime	Liefert den Zeitstempel des angegebenen Telegramm Bytes in Sekunden mit Mikrosekunden Genauigkeit. Die Indizierung des Datenbytes erfolgt wie in obiger Funktion data.
dir	Abfrage der Datenrichtung des Telegramms. Ein Resultat von 1 bedeutet, das das Telegramm an Port A (CH1) empfangen wurde, eine 2 Port B (CH2).
dump	Liefert einen Lua String mit einer Datenauflistung des angegebenen Telegrammbereichs in hexadezimaler oder dezimaler Notation. `dump` ist besonders hilfreich wenn ein schneller Überblick über den Telegramminhalt erforderlich ist oder ein definierter Telegrammbereich als 'Rohdaten' angezeigt werden soll.
duration	Liefert die Zeitlänge bzw. Dauer des Telegramms in Sekunden. Dies entspricht der Zeit zwischen erstem Startbit und letztem Stopbit.
geterror	Liefert den Fehlerstatus (frame, parity, break) des Datenbytes an der gegebenen Telegramm Position: Mögliche Resultate sind: 0:Kein Fehler, 1:Frame, 2:Parity, 4:Break.
isbreak	Liefert true wenn es sich bei dem indizierten Datenbyte um ein echtes Break handelt.
number	Abfrage der Telegramm Nummer gezählt ab 1.
size	Liefert die Länge des Telegramms in Datenbytes. Beachten Sie, das ein Telegramm auch 9-Bit Werte enthalten kann die ebenfalls als ein Datenbyte gezählt werden.
string	Liefert den Telegramminhalt als Lua String. Da Lua Strings nur 8-Bit Werte enthalten können, werden etwaige 9-Bit Werte auf 8 Bit reduziert.

time	Liefert die Zeit wann das Telegramm relativ zum Beginn der Aufzeichnung empfangen wurde. Das Resultat ist in Sekunden mit einer Genauigkeit von 6 Nachkommastellen (Auflösung in Mikrosekunden). Beispiel: Ein Wert von 25.034198 bedeutet, das das Telegramm exakt 25.034198 Sekunden nach Start der Aufnahme aufgetreten ist.

telegram:data

Liefert den Datenwert des Telegramms an der angegebenen Position. Wie üblich zählt der Positionswert von 1 an. Negative Positionen adressieren die Daten vom Telegrammende. Da die MSB-RS485-PLUS 9 Bit Datenprotokolle unterstützt ist der Rückgabewert im Bereich 0...511.

telegram:data(*INDEX)*

- **INDEX** Index des angeforderten Datenwertes.

Beispiel

```
1  function out()
2      -- access the current telegram
3      local tg = telegrams.this()
4      -- shows the first byte in the telegram as decimal value
5      box.text{ caption="First", text=tg:data( 1 ) }
6      -- shows the last byte in the telegram as decimal value
7      box.text{ caption="Last", text=tg:data( -1 ) }
8  end
```

telegram:datatime

Liefert den Zeitstempel des angegebenen Telegramm Bytes. Die Indizierung startet wie üblich mit 1. Negative Indexe adressieren die Daten vom Telegrammende.

telegram:datatime(*INDEX)*

- **INDEX** Index des angeforderten Datenbytes

Example

```
1  function out()
2      -- access the current telegram
3      local tg = telegrams.this()
4      -- shows the pause between the stop bit of the first byte and the
5      -- start bit of the second byte (substract sending time)
6      local delay = tg:datatime( 2 ) - tg:datatime( 1 ) - transmission.
           bytepause( 1 )
7      box.text{ caption="Pause 1-2", delay }
8  end
```

telegram:dir

Liefert die Quelle oder Richtung des Telegramms. Ein Wert von 1 gibt Port A (oder CH1) als Datenquelle an, ein Wert von 2 bedeutet, das das Telegramm an Port B (CH2) empfangen wurde.

telegram:dir

Beispiel

```
1  function out()
2      -- access the current telegram
3      local tg = telegrams.this()
4
5      if tg:dir() == 1 then
6          -- do something with data form port A
7      else
8          -- telegram received at port B
9      end
10 end
```

telegram:dump

Erzeugt einen Lua String mit einer Datenauflistung des angegebenen Telegrammbereichs in 3-stelliger hexadezimaler oder dezimaler Notation, getrennt jeweils durch ein vorgegebenes Zeichen oder String. Ohne Argumente wird der komplette Telegramminhalt verwendet. Die vorgegebene Zahlenbasis ist hex (16), des Default Trennzeichen ein Leerzeichen.

telegram:dump{ *first=1, last=-1, base=16, width=3, sep=' ', max=LEN* }

- **first:** Spezifiziert den ersten Datenwert in der hex dump Ausgabe, Default ist der allererste Datenwert im Telegramm (1).
- **last:** Definiert den letzten im hex dump zu verwendeten Datenwert, vorgegeben ist der letzte im Telegramm (-1 oder `telegram:size()`).
- **base:** Die zu verwendende Zahlenbasis, Vorgabe ist hex (base=16).
- **width:** Die Anzahl der Stellen bei der Ausgabe der Daten, Default ist 3 Stellen (nötig für 9-Bit Werte). In den meisten Fällen werden Sie aber in Verbindung mit der Hex Darstellung `width=2` übergeben.
- **sep:** Ersetzt das vorgegebene Trennungsleerzeichen durch einen beliebigen anderen Charakter oder String. Ein leerer String unterdrückt die Ausgabe eines Trennungszeichens komplett.
- **max:** Limitiert die maximale Anzahl der angezeigten Daten im hex dump. Ein Wert von `max=4` gibt lediglich die ersten und letzten beiden Datenwerte aus. Die übrigen (nicht angezeigten Datenwerte) werden als Anzahl zusammengefaßt. Die Vorgabe von max entspricht der Telegrammlänge.

Example

```
function out()
    -- access the current telegram
    local tg = telegrams.this()
    -- show the complete telegram content as hex dump
    box.text{ caption="Data (hex)", text=tg:dump{} }
    -- shows the last two bytes as hex without separator and 2 digit
    box.text{ caption="EOS", text=tg:dump{first=-2, width=2, sep=''}
    -- shows the second byte as a decimal value
    box.text{ caption="Second", text=tg:dump{ first=2, last=2, base=10
        }
end
```

telegram:duration

Die Telegrammdauer oder Länge in Sekunden. Das Resultat ist eine doppelt genaue Fließkommazahl mit der üblichen Genauigkeit in Mikrosekunden.

telegram:duration()

Beispiel

```
function out()
    -- access the current telegram
    local tg = telegrams.this()
    -- display the duration of the telegram
    box.text{ caption="Length (s)", text=tg:duration() }
end
```

telegram:geterror

Liefert einen Wert ungleich Null wenn das Datenbyte an der angegebenen Telegramm Position vom Analyser mit einem Fehlerstatus markiert wurde. Rückgabewerte sind: 0: Kein Fehler, 1: Rahmenfehler, 2: Paritätsfehler, 4: Break

telegram:geterror(*INDEX)*

- **INDEX** Index des angeforderten Bytes.

Example

```
function out()
    -- access the current telegram
    local tg = telegrams.this()
    local err = tg:geterror(1)
    if err then
        local t = {
          [1] = "Frame",
          [2] = "Parity",
          [4] = "Break"
       }
       box.text{ caption="Error", text = t[err] or "UNKNOWN:"..err }
    end
end
```

telegram:isbreak

Gibt true zurück, wenn es sich bei dem Daten (NULL) Byte an der indizierten Position um ein echtes Break handelt. Ansonsten false.

telegram:isbreak(*INDEX)*

- **INDEX** Index des angeforderten Bytes.

Beispiel

```
1  function out()
2      -- access the current telegram
3      local tg = telegrams.this()
4      -- distinguish between a normal null byte and a break
5      if tg:data( 1 ) == 0 then
6          if tg:isbreak( 1 ) then
7              box.text{ caption="BREAK", text=tg:data( 1 ) }
8          else
9              box.text{ caption="NULL", text=tg:data( 1 ) }
10         end
11     end
12 end
```

telegram:number

Die aktuelle Telegramm Nummer, d.h. um welches Telegramm es sich in der aktuellen Aufzeichnung handelt.

telegram:number()

Beispiel

```
1  function out()
2      -- access the current telegram
3      local tg = telegrams.this()
4      -- show the current telegram number
5      box.text{ caption="Number", text=tg:number() }
6  end
```

telegram:size

Dient zur Abfrage der Telegrammlänge. Beachten Sie, das auch 9-Bit Werte als ein Datenwert gezählt werden.

telegram:size()

Beispiel

```
1  function out()
2      -- access the current telegram
3      local tg = telegrams.this()
```

```
4       -- show the size of a telegram
5       box.text{ caption="Length", text=tg:size() }
6   end
```

telegram:string

Liefert den Inhalt des Telegramms als Lua String zurück.
Beachten Sie: Ein Lua String kann zwar beliebige Bytes im Bereich 0...255 enthalten, aber keine 9 Bit Werte. Eventuell vorhandene 9-Bit Werte werden deshalb auf 'normale Bytes' reduziert.
Im Gegensatz zu den früheren Modulen `tg` und `tgprev` akzeptiert die 'neue' string Methode keine Bereichsparameter und gibt immer den ganzen Telegramminhalt als String zurück.
Da das Lua String Modul bereits leistungsfähige Funktionen zur String Extraktion bietet, besteht auch keine Notwendigkeit diese erneut zu implementieren. Und: Lua erlaubt das Indizieren von Teilstringen auch vom 'Stringende', was ein zusätzlicher Vorteil ist.

telegram:string()

Beispiel

```
1   function out()
2       -- access the current telegram
3       local tg = telegrams.this()
4       -- extract the bytes 2...5 as a Lua string
5       local data = tg:string():sub(2,5)
6       -- query the last two EOS bytes
7       local eos = tg:string():sub(-2,-1)
8   end
```

telegram:time

Liefert den Zeitstempel des Telegramms. Der Zeitstempel ist definiert als die Zeit in Sekunden, die seit Start der Aufnahme vergangen ist, wenn das erste Byte des Telegramms empfangen wurde. Das Resultat ist ein Fliesskommawert mit doppelter Genauigkeit und der üblichen Auflösung von 1 Mikrosekunde.

telegram:time()

Beispiel

```
1   function out()
2       -- show the response time to the former telegram
3       local tc = telegrams.this()
4       local tp = telegrams.prev()
5       -- handle not existing previous telegram (at very first position)
6       if not tp then tp = tc end
7       box.text{ caption="Response time", text=tc:time() - tp:time() }
8   end
```

14.8.8 Das telegrams Modul

Das `telegrams` Modul bietet einen einfachen und vor allem wahlfreien Zugriff auf alle bis dato aufgenommenen Telegramme.
Der große Vorteil: Im Gegensatz zu den bisherigen (und nun obsoleten) Modulen `tg` und `tgprev` ist der Telegrammzugriff nun nicht länger nur auf das aktuelle und vorherige Telegramm beschränkt. Vielmehr ermöglicht `telegrams` Ihnen jetzt auch Telegramme zu behandeln, deren Verarbeitung von einem zu einem früheren Zeitpunkt empfangenen Telegramm abhängig ist[9].

Ein Beispiel: Sie müssen ein Telegramm differenziert interpretieren falls ein zuvor empfangenes Telegramm von Typ XYZ vorliegt. In einem Bussystem mit mehreren Teilnehmern ist dies nicht zwangsläufig das vorherige Telegramm. D.h. Sie brauchen eine Möglichkeit über die zuvor empfangenen Telegramme zu 'iterieren' um das passende Telegramm zu finden.

Der Zugriff auf ein beliebiges Telegramm erfolgt einfach per Aufruf der Modul Funktion `telegrams.at(index)` wobei index die Nummer des gewünschten Telegramms ist.
Der Rückgabewert (oder das Rückgabeobjekt) ist immer vom Typ `telegram` (siehe 14.8.7). Dieser spezielle Typ agiert als ein Interface und stellt Ihnen die gleichen Funktionen zur Verfügung, die Sie bereits von dem Modulen `tg` und `tgprev` gewöhnt sind.
`telegrams.at` ist die einzige Funktion die Sie für den Zugriff auf ein bestimmtes Telegramm benötigen. Da es sich beim Abruf des aktuellen oder vorherigen Telegramms aber um die am meisten benötigten Operationen handelt, bietet das Modul zwei entsprechende Alias Funktion. Die folgende Tabelle listet alle Modul Funktionen auf:

Funktion	Beschreibung
at	liefert das Telegramm mit der angegebenen Nummer bzw. Position.
this	liefert das aktuell in `out()` behandelte Telegramm. Dies ist ein Alias und gleichbedeutend mit dem Aufruf für `telegrams.at(-1)`.
prev	liefert das vorherige in `out()` behandelte Telegramm. Entspricht dem Aufruf `telegrams.at(-2)`.

telegrams.at

Die Funktion `telegrams.at(index)` akzeptiert absolute und relative Indexe und liefert das zugehörige Telegramm - oder nil, wenn Sie einen ungültigen Index übergeben. Der (Zeit)Aufwand für den Telegrammzugriff ist immer linear. Es macht keinen Unterschied ob Sie das zuletzt empfangene oder allererste Telegramm abfragen.
Absolute Telegrammnummern sind immer positiv. Sie starten mit einem Index von 1 (erstes Telegramm) und enden mit der Nummer des letzten aufgenommenen Telegramms.
Relative Index hingegen zählen vom aktuell in der `out()` Funktion bearbeiten

[9] Bisher konnten Sie in out() nur auf das aktuelle und vorherige Telegramm zugreifen.

Telegramm und werden als negative Werte übergeben. Ein Index von -1 liefert das aktuelle Telegramm (und macht das `tg` Modul obsolete), ein Index -2 das vorherige (bisher per `tgprev` realisiert), ein Index von -3 das vorvorletzte (immer bezogen auf das in `out()` behandelte Telegramm), usw.
Da in der `out()` Funktion immer nur das EINE Telegramm bezogen auf die entsprechende Zeile im Ausgabefenster bearbeitet wird, ist eine relative Indexierung viel praktischer, da Sie nicht über die zugehörige absolute Telegrammnummer nachdenken müssen.

telegrams.at(*index*)

- **index:** Der Index oder die Nummer des angeforderten Telegramms. Ein positiver Index zählt vom Beginn der Aufnahme, ein negativer Index zählt rückwärts vom aktuell in `out()` behandelten Telegramm.

Beispiele

```
1  function out()
2      -- query the current telegram
3      local telegram = telegrams.at( -1 )
4      -- show the telegram time
5      box.text{ caption="Time", text=telegram.time() }
6  end
```

Die folgenden Programmzeilen berechnen den Zeitabstand zwischen dem aktuellen (Index -1) und zuvor empfangenen (Index -2) Telegramms. Statt beide Telegramme separat als lokale telegram Variablen zu speichern, erfolgt der Zugriff auf die Zeitinformationen direkt:

```
1  function out()
2      -- show the time difference between the current and previous
           telegram
3      box.text{ caption="dt",
4                text=telegrams.at(-1):time() - telegrams.at(-2):time() }
5  end1
```

14.9 Einstellungen

Im Einstelldialog finden Sie zusätzliche Möglichkeiten, um die Darstellung der Telegramme an Ihre persönlichen Anforderungen anzupassen. So können Sie jedem Telegramm aus einer Liste von sogenannten Präfixen bestimmte Zusatzinformationen voranstellen (Telegrammnummer, Datum/Uhrzeit, etc.), die verwendete Schrift ändern und eine andere Hintergrundfarbe für das Telegrammfenster auswählen. Der Einstelldialog wird per Klick auf:
`Einstellungen→Protokollmonitor einrichten...`
geöffnet.

14.9.1 Anzeige zusätzlicher Telegramminformationen

Der Protokollmonitor bietet eine Reihe von Zusatzinformationen die Sie den Telegrammen voranstellen können. Selbstverständlich könnten Sie dies auch durch ein paar Zeilen Lua Code realisieren, aber manchmal ist einfacher - und

auch gewollt - bestimmte Informationen zu den Telegrammen kurz mal einzuschalten und später wieder aus der Telegrammanzeige zu entfernen.
Folgende Zusatzinformationen sind im Präfix Reiter des Einstelldialogs auswählbar:

1. **Telegramm Number**
   Die aktuelle Telegramm beginnend mit 1 und unabhängig von der Telegrammrichtung.
2. **Telegramm Zeit**
   Die Empfangszeit des ersten Telegrammbytes relativ zum Aufnahmestart in Sekunden.
3. **Telegramm Datum und Uhrzeit**
   Die absolute Zeit und Datum des Telegramms. Zeit und Datum werden in dem jeweils lokalen Format (abhängig von Ihrer PC Einstellung) und mit zusätzlichem Mikrosekundenanteil angezeigt.
4. **Telegramm Dauer**
   Entspricht der zeitlichen Länge des Telegramms in Sekunden, gemessen vom ersten Startbit bis zum letzten Stopbit.
5. **Zeitdistanz zum vorherigen Telegramm**
   Die 'Pausezeit' zwischen dem aktuellen und vorherigen Telegramm. Gemessen als die Zeit zwischen dem letzten Stopbit des vorherigen und dem Startbit des aktuellen Telegramms.

Jede Änderung der Auswahl wird unmittelbar im Telegrammfenster angezeigt.

14.9.2 Änderung der Schriftart

Telegramm Schriftart

Die Änderung der aktuellen Schrift wirkt sich direkt auf die Box-Darstellung aus. Sie können eine kleinere Schrift wählen, wenn Sie mehr Informationen in einer Zeile angezeigt bekommen möchten. Oder Sie favorisieren eine größere Schriftart wegen der besseren Lesbarkeit.
Der Schriftauswahl Dialog erlaubt Ihnen Schriftart, Still und Größe getrennt auszuwählen. Alle Änderungen wirken sich unmittelbar auf die Darstellung aus und werden automatisch gespeichert.

Änderung der Schrift per Maus und Tastatur

Neben dem Schriftdialog existiert eine noch weitaus einfachere Möglichkeit, die Schrift anzupassen. Halten Sie die Strg Taste gedrückt während Sie das Mausrad drehen. Oder drücken Sie die Tastenkombination Strg+[+] bzw. Strg+[−] um die Darstellung zu vergrößern oder zu verkleinern. Strg+[0] schaltet auf die voreingestellte Größe zurück.

14.9.3 Eine andere Hintergrundfarbe

Background color

Per Templateskript können Sie die Farbe der Telegramme via Box-Modell vorgeben. Wenn Sie zusätzlich die Farbe des Telegrammfensters (den Hintergrund) ändern wollen, öffnen Sie den Farbdialog und klicken Sie auf den Farbknopf. Anschließend können Sie eine beliebige Farbe auswählen die dann als neue Hintergrundfarbe übernommen wird.

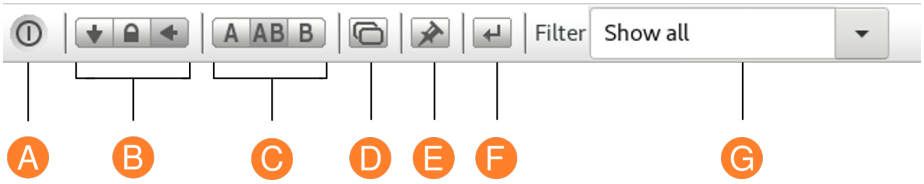
14.9.4 Lua Kompatibilität

Lua Kompatibilität

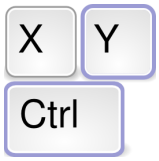
Um immer die bestmögliche Protokollverarbeitung zu ermöglichen ist es manchmal unausweichlich die Lua Funktionalität auch auf Kosten der Abwärtskompatibilität zu ändern. Wir tun dies nicht leichtfertig und brechen mit bisherigen Versionen nur dann wenn der Vorteil außer Frage steht. In solchen Fällen geben wir Ihnen die Gelegenheit Ihre eigenen Templates möglichst ohne großen Aufwand an die Änderungen anzupassen.
Per Voreinstellung akzeptiert der Lua Interpreter auch obsolete Funktionen zumindest für einen gewissen Zeitraum (oder Folgeversionen). Um Ihre eigenen Templates auf den neuesten Stand zu bringen, deaktivieren Sie einfach den Kompatibilitätsschalter in diesem Dialog. Der Protokollmonitor zeigt Ihnen dann alle relevanten Programmzeilen an, die er in einer der nächsten Versionen nicht mehr akzeptieren wird. Weitere Details über die abgekündigten Funktionen und Module finden Sie im Abschnitt 14.12.2.

14.10 Die Werkzeugleiste

Die Werkzeugleiste dient zum schnellen Zugriff der am meisten benötigten Funktionen. Einige davon sind bei allen Monitoren identisch, andere spezifisch für den Protokollmonitor.



A	**Ende:** Speichert alle Einstellungen und schließt das Fenster
B	**Anzeigemodus:** Je nach Anzeigemodus zeigt der Fensterinhalt immer das aktuelle (zuletzt aufgenommene) Ereignis, ist verriegelt oder aktualisiert den Inhalt mit anderen Fenstern.
C	**Datenrichtung:** Der Protokollmonitor kann beide Datenrichtungen (Datenkanal A und Datenkanal B) gemeinsam oder einzeln anzeigen, z.B. um diese getrennt in zwei einzelnen Fenstern darzustellen.
D	**Neue Ansicht:** Öffnet ein neues Fenster mit dem gleichen Ausschnitt und identischen Einstellungen.
E	**Default Einstellung:** Speichert die aktuellen Protokollmonitor Einstellungen als Vorgabe wenn ein neues Protokollmonitor Fenster geöffnet wird.
F	**Gehe zu Dialog:** Öffnet den Gehe zu Telegramm Nummer Dialog.
G	**Filter Control:** Erlaubt die Auswahl und Übergabe beliebiger Filter Kriterien an das aktuelle Template.



Tastenkommandos der wichtigsten Funktionen

14.11 Kurzbefehle

Aktion	Kurzbefehl
Online Hilfe zum Protokollmonitor	F1
Zeigt die Telegramme in einem neuen Fenster	Strg + N
Alle Telegramme auswählen	Strg + A
Auswahl aufheben	Umschalt + Strg + A
Ausgewählte Zeilen exportieren	Strg + E
Gehe zu Telegramm-Nummer Dialog anzeigen	Ctrl + G
Vergrößert die aktuelle Telegrammdarstellung	Strg + +
Verkleinert die aktuelle Telegrammdarstellung	Strg + −
Kehrt zur normalen Telegrammvergrößerung zurück	Strg + 0
Öffnet das interne Debug Ausgabefenster	Strg + Alt + O
Fenster schließen	Strg + Q

14.12 Änderungen zu vorherigen Versionen

Lua Module und Sprach Ergänzungen unterliegen einer steten Weiterentwicklung, ausgelöst durch neue Analyser Fähigkeiten und durch die Unterstützung neuer Feldbus Protokolle. Dabei versuchen wir die Änderungen möglichst abwärts kompatibel zu halten.
Wo dies nicht der Fall ist und wie Sie Ihren Code entsprechend anpassen können zeigt der folgende Abschnitt.

14.12.1 Inkompatible Änderungen

Mit Version 5.0 wurden die folgenden Module umbenannt (hauptsächlich für ein besseres Verständnis oder um der offiziellen Lua 5.3 Dokumentation zu entsprechen).

- bit → bit32
- cfg → config
- protocol → transmission

Sie können in Ihren Templates entweder alle Modul Aufrufe umbenennen (verwenden Sie dazu einfach den Suche/Ersetze Dialog im Editor) oder Sie fügen folgende Zeilen zu Beginn Ihrer Templates ein:

```
1 bit = bit32
2 cfg = config
3 protocol = transmission
```

14.12.2 Obsolete Funktionen und Module

Die folgenden Funktionen wurden schon früher als 'obsolete' gekennzeichnet. Sie sind nach wie vorher vorhanden, werden aber in einer der nächsten Software Versionen endgültig entfernt und nicht länger unterstützt.

Im folgenden Abschnitt erfahren Sie, wie Sie Ihre eigenen Templates updaten und obsoleten Programmcode durch die neuen und weitaus leistungsfähigeren Module und Funktionen ersetzen[10].
Zunächst eine Übersicht der obsoleten Module:

- `tg` - Zugriff auf das aktuelle Telegramm in der Funktion `out()`
- `tgprev` - Zugriff auf das vorherige Telegramm in der Funktion `out()`
- `hex` - Konvertiert die Hex ASCII Daten des aktuellen Telegramms in Binärdaten
- `box.hexdata` - Gibt einen Abschnitt der Daten im aktuellen Telegramm als Hex Dump aus

Vermutlich fragen Sie sich jetzt, was daran schlecht ist?
Die Schwäche liegt im Design. Hier werden reine Ausgabe- oder Konvertierungsmodule mit einem festgelegten Telegrammzugriff vermischt, hier als aktuell blau hervorgehoben.
So verarbeiten sowohl das `hex` Modul als auch die `box.hexdata` Funktion NUR das gerade aktuelle Telegramm. Sie können keinen Hexdump des vorherigen Telegramms erzeugen. Genauso wenig können Sie Hex ASCII Daten anderer Telegramme außer dem aktuellen konvertieren. In beiden Fällen müssen Sie dies durch eigene Lua Funktionen realisieren.
Mehr noch: `tg` und `tgprev` begrenzen die Telegramm Verarbeitung innerhalb `out()` auf das aktuelle und vorherige Telegramm. Sobald Sie auf ein früheres Telegramm zugreifen wollen haben Sie verloren.

Das neue `telegrams` Modul macht dem ein Ende und bietet Ihnen einen wahlfreien Zugriff auf ALLE Telegramme, die bislang empfangen wurden wenn `out()` ausgeführt wird. Es ist daher nur ein konsequenter Schritt, das die Nachfolger von `box.hexdata` und `hex` ihre Abhängigkeit von `tg` abstreifen und mit beliebigen Telegrammen zusammenarbeiten.

Im folgenden soll uns das Modbus ASCII Telegramm 'Write Single Register' als Beispiel dienen. Wir werden dabei den Ausgabecode Schritt für Schritt an die neuen Lua Komponenten anpassen. Das Telegramm hat die nachstehende Struktur:

:	Addr	Func	RegHi	RegLo	ValHi	ValLo	LRC	CR	LF
1 char	2 chars	2 chars	2 chars	2 chars	2 chars	2 chars	2 chars	1 char	1 char

Mit Ausnahme des Startzeichens ':' und der Endesequenz CRLF werden alle Telegrammdaten in den hexadezimalen Zeichen 0-9, A-F (Hex ASCII bzw. base16 codiert) übertragen. Hier eine exemplarische Bytefolge wie sie z.B. im Datenmonitor angezeigt wird:

3A	30	31	30	36	30	30	31	39	30	33	33	45	39	46	0D	0A

Eine frühere Lösung mit den obsoleten Modulen sah ungefähr so aus:

```
1 box.text{caption="Start",text=string.char(tg.data(1))}
2 box.text{caption="Addr",text=hex.byte{ pos=2 }}
```

[10] Die aktuellen Protokolltemplates und Beispiele sind bereits angepaßt und können als Vorlage verwendet werden.

```
3  box.text{caption="Func",text=hex.byte{ pos=4 }}
4  box.text{caption="Register",text=hex.int16{pos=6,order="BE"}}
5  box.text{caption="Value",text=hex.int16{pos=8,order="BE"}}
6  box.text{caption=''LRC'',text=string.format("%02X",hex.byte{pos=tg.size
       ()-3})}
7  box.hexdata{caption="End",pos=tg.size()-1,len=2,width=2}
```

Im ersten Schritt ersetzen wir das `tg` Modul und wandeln alle hex ASCII Zeichen (die grünen und gelben Abschnitte) in eine rein binäre Sequenz zurück.

```
1  local tele = telegrams.this()
2  local bindata = base16.decode(tele:string():sub(2,-3))
```

Zeile 1 erfragt das aktuelle Telegramm und weist es der Variable `tele` zu. Diese Variable enthält damit alle Informationen, die zuvor per `tg` Modul verfügbar waren. Im Gegensatz zu `tg` kann sich die Variable aber auch auf beliebige andere Telegramme beziehen.
In Zeile 2 extrahieren wir den gelb/grünen Abschnitt aus dem Telegramm String per Lua's (sub)string Funktion und übergeben ihn an die `base16` decode Funktion. Der gelb/grüne Abschnitt beginnt mit dem 2ten Byte (Index 2) und endet mit dem drittletzten (Index -3). Das Result ist eine Binärsequenz der gelben und grünen Bytes.
Bitte beachten Sie! Sie können nicht das komplette Telegramm zurückwandeln, da der Doppelpunkt des Startzeichens als auch das CRLF keine gültigen Hex ASCII (oder Base16) Zeichen sind!

Mit den originalen Binärdaten zur Hand gibt es keine Notwendigkeit die verschiedenen Telegrammbestandteile wie z.B. Adresse, Funktion, Register, Wert oder LRC Prüfsumme einzeln aus dem Hex ASCII Format zu konvertieren. Die Funktion `string.unpack`[11] in Zeile 3 erledigt dies weitaus eleganter und in einem Rutsch.

```
1  local tele = telegrams.this()
2  local bindata = base16.decode(tele:string():sub(2,-3))
3  local adr,fnc,reg,val,lrc,pos = string.unpack( "bb>H>Hb", bindata )
```

`string.unpack` entpackt (unpack) die ihm übergebene Datensequenz gemäß dem zweiten Formatstring Parameter `"bb>H>Hb"`. Übersetzt heißt das sinngemäß:

1. liefere das erste Zeichen als Byte (b) (adr)
2. liefere das zweite Zeichen als Byte (b) (fnc)
3. fasse das 3te und 4te Byte als vorzeichenlosen 16 Bit Wert (H) mit dem höherwertigen Byte zuerst (>) zusammen und gebe ihn zurück (reg)
4. fasse das 5te und 6te Byte als vorzeichenlosen 16 Bit Wert (H) mit dem höherwertigen Byte zuerst (>) zusammen und gebe ihn zurück (val)
5. liefere das 7te Zeichen als Byte (b) (lrc)
6. gebe IMMER das Ende der Umwandlung im String zurück (pos)

Insgesamt liefert der Aufruf 6 Resultate. Das letzte Resultat ist immer die Position an der die Extrahierung endete. Dies ist gleichbedeutend mit der Position ab der ein eventuell folgender `unpack` Aufruf weiter machen soll.
Zeile 3 fasst alle Resultate in den entsprechenden Variablen zusammen und wir können diese dann ohne weitere Bearbeitung einfach ausgeben.

[11] Beachten Sie! `string.unpack` ersetzt das frühere `bunpack`

```
1  local tele = telegrams.this()
2  local bindata = base16.decode(tele:string():sub(2,-3))
3  local adr,fnc,reg,val,lrc,pos = string.unpack( "bb>H>Hb", bindata )
4  box.text{ caption="Addr", text=adr }
5  box.text{ caption="Func", text=fnc }
6  box.text{ caption="Register", text=reg }
7  box.text{ caption="Value", text=val }
8  box.text{ caption="LRC", text=string.format("%02X",lrc) }
```

Übrig bleiben jetzt noch das Start ':' Zeichen und die CRLF Endesequenz. Der Doppelpunkt ist das erste Zeichen im Originaltelegramm und wir können es ähnlich wie im obsoleten Code handhaben. Sehen Sie dazu Zeile 4 im folgenden Listing.
Die neue `telegram:dump` Funktion ersetzt den restriktiven `box.hexdata` Aufruf in Zeile 10. Die `dump` Funktion 'gehört' immer zu einer zuvor zugewiesenen `telegram` Variable und greift nicht mehr intern auf `tg` zu.

```
1  local tele = telegrams.this()
2  local bindata = base16.decode(tele:string():sub(2,-3))
3  local adr,fnc,reg,val,lrc,pos = string.unpack( "bb>H>Hb", bindata )
4  box.text| caption)"Start", text=string.char( tele:data(1) )
5  box.text{ caption="Addr", text=adr }
6  box.text{ caption="Func", text=fnc }
7  box.text{ caption="Register", text=reg }
8  box.text{ caption="Value", text=val }
9  box.text{ caption="LRC", text=string.format("%02X",lrc) }
10 box.text{ caption="End", text=tele:dump{ first=-2, width=2 } }
```

15
Der Signalmonitor

Der MSB-RS485-PLUS Analyser tastet alle Signale mit bis zu 200 Mhz ab. Das Ergebnis liefert der Signalmonitor. Analog zu einem Digitalscope können Sie beliebige Ausschnitte anfahren und in unterschiedlicher Vergrößerung untersuchen.

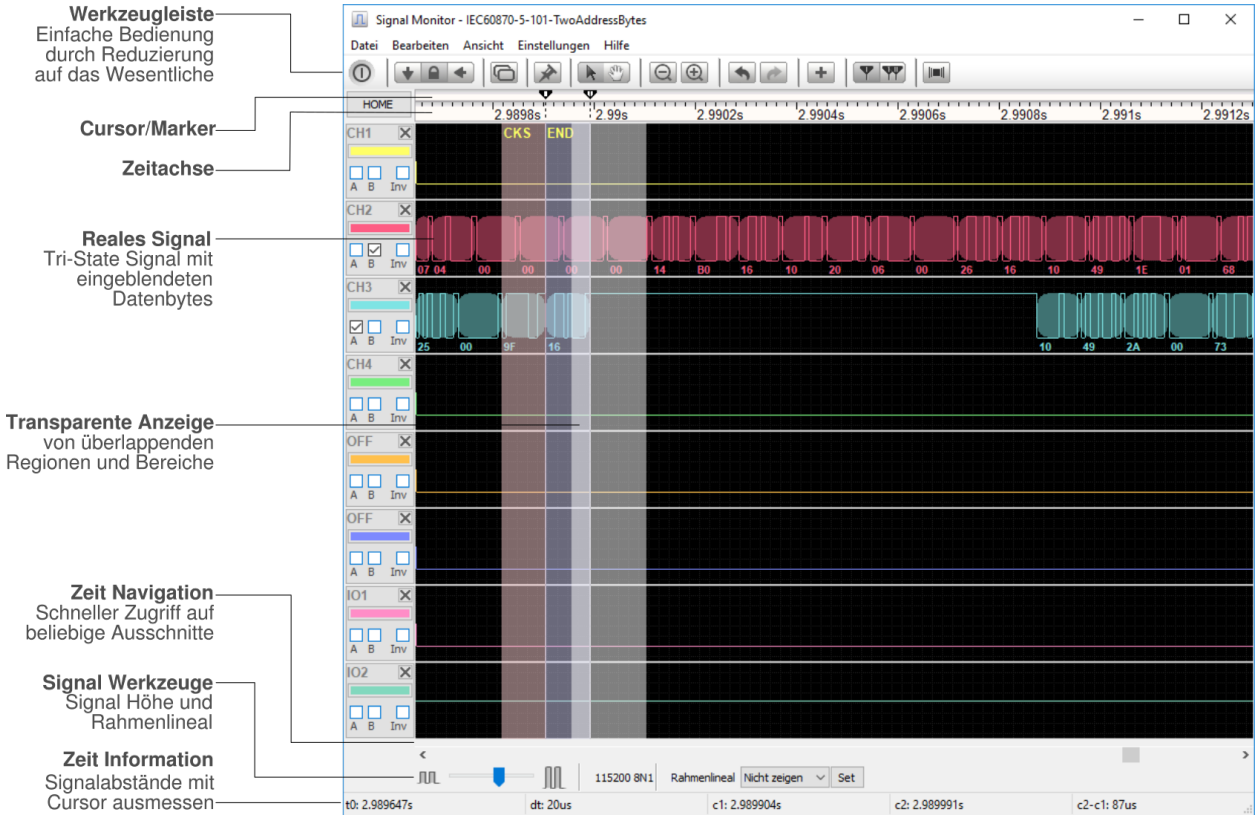
Zur Analyse von seriellen Datenverbindungen reicht es oftmals nicht aus, nur die übertragenen Datenbytes zu betrachten. Gerade bei Bus Systemen kommt es auf ein reibungsloses Zusammenspiel aller Komponenten an. Dies erfordert die korrekte Parametrisierung und Einhaltung der Protokollspezifikationen von allen Bus Teilnehmer. Die mögliche Fehlerursachen sind entsprechend vielfältig.

Durch falsche Einstellungen oder eine Fehlfunktion der Sende-Hardware können ungültige Daten auf den Bus gelegt werden.
Auftretende Datenkollisionen durch mehrere, gleichzeitig aktive Sender (Stichwort Bus-Freigabe/Belegung) oder ungültige Datensequenzen (Frames) durch falsches Timing sind allein mit Aufzeichnung der Daten nicht korrekt zu beurteilen. Das gleiche gilt für Hardware Handshakes.

Um ein (Fehl)Verhalten solcher Verbindungen beurteilen zu können, trägt der Signalmonitor die Pegel aller Leitungen über die Zeit auf. Die Zeitauflösung beträgt dabei 10ns.

Alle von der MSB-RS485-PLUS aufgezeichneten Leitungen werden dabei parallel dargestellt, wobei jedes Leitungssignal einzeln ein-/ausgeschaltet und die Reihenfolge der Darstellung individuell variiert werden kann. Die Arbeitsweise des Signalmonitors kann am besten mit einem digitalen, 8 kanaligem Speicheroszilloskop verglichen werden. Im Gegensatz zu einem Oszilloskop ist die Aufzeichnungstiefe, d.h. die Zeitdauer der Aufzeichnungstiefe , d.h. die Zeitdauer aufgenommenen Signale nur durch den Festplattenplatz (und die Rechenleistung Ihres Computers) beschränkt.
Durch den Aufruf mehrerer Signalmonitore können Sie die aufgezeichneten Signale (Pegelveränderungen der einzelnen Leitungen) an unterschiedlichen Stellen und mit unterschiedlicher Zeitauflösung untersuchen bzw. vergleichen. Daneben eignet sich der Signalmonitor besonders bei der Beurteilung des Zeitverhaltens von gesendeten Datenbytes oder bei Fragen nach der Kontinuität der gesendeten Daten. Im einfachsten Fall zeigt der Signalmonitor die

Pegeländerungen einer gerade aktiven Datenverbindung und liefert damit bereits entscheidende Hinweise auf das Funktionieren bzw. nicht funktionieren einer laufenden Datenübertragung.



15.1 Die Signaldarstellung

Die Signalanzeige ist unterteilt in drei Abschnitte. Direkt unter der Werkzeugleiste befinden sich die Cursor-Leiste (siehe Cursor) sowie die Zeitachse. Die Zeitachse liefert Ihnen die genaue Position und Auflösung des gerade sichtbaren Signalausschnittes. Um die Ablesung zu erleichtern werden bei allen Zeitangaben überflüssige Stellen durch Prefixe ersetzt. Aus 0.012570s wird z.B. 12.57ms.
Im Anschluß an die Zeitachse erfolgt die Signaldarstellung. Für alle Signale gilt der gleiche Ausschnitt sowie die gleiche Auflösung (Zeitbasis). Um ein Signal an mehreren Positionen zu untersuchen, starten Sie einfach einen weiteren Signalmonitor. Sie können die aktuelle Ansicht duplizieren, indem Sie den Clone Knopf in der Werkzeugleiste anklicken. Dadurch wird eine neue Instanz des Signalmonitors gestartet, der exakt die gleichen Einstellungen wie der aktuelle besitzt. Oder Sie starten einen neuen Signal Monitor mit den zuletzt verwendeten Einstellungen aus dem Kontrollprogramm.

Vergleichen von Signalabschnitten

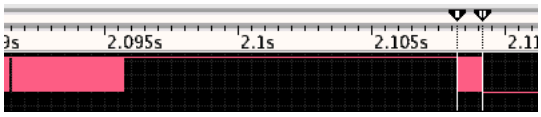
Sie können beliebig viele Signalausschnitte gleichzeitig betrachten, auch bereits während einer laufenden Aufzeichnen. Öffnen Sie dazu einfach neue Signalmonitor Fenster mit dem gewünschten Abschnitt.

Jedes sichtbare Signal wird durch seinen Namen am linken Rand beschrieben. Den Signalnamen können Sie jederzeit im Einstelldialog des Kontrollprogramms ändern. Die Reihenfolge, in welcher die Signale dargestellt werden, ist individuell einstellbar. Das ermöglicht wichtige Signalleitungen direkt übereinander anzuzeigen. Zusätzlich können unwichtige Signale ausgeblendet werden. Alle Signal relevanten Einstellungen (inklusive der Signalanordnung) erfolgen direkt im zugehörigen Signalkontrollfeld links vom eigentlichen Signal, siehe Abschnitt 15.5.

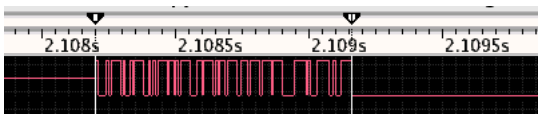
Individuelle Signalanzeige

Signalreihenfolge, Sichtbarkeit, Invertierung, Daten Overlay und Signalfarbe können einfach und direkt in der zugehörigen Signalkontrollfeld eingestellt werden!

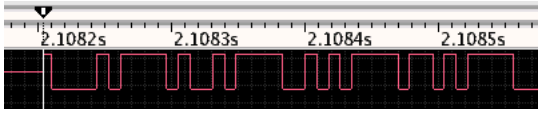
Der sichtbare Signalbereich wird durch seine Position als Zeitdifferenz vom Anfang der Aufzeichnung und seinen sichtbaren Ausschnitt definiert. Der sichtbare Ausschnitt errechnet sich aus der Größe des Signalfensters in Bildschirm Pixel und der Zeitbasis, d.h. wieviele usec (Millionstel Sekunden) pro Pixel dargestellt werden. Je größer die Zeitbasis, desto größer ist die Anzahl von Mikrosekunden, die innerhalb des sichtbaren Fensters angezeigt werden.
Um einen kompletten Überblick über die bis dato aufgenommenen Ereignisse zu haben, klicken Sie den Knopf HOME in der oberen linken Ecke oder drücken Sie die Tastenkombination Strg+Pos1. Die Zeitbasis wird dann automatisch so gewählt, daß möglichst alle Ereignisse in ein Signalfenster passen. Je nach gewählter Zeitbasis können mehrere Ereignisse, Pegelwechsel innerhalb eines Bildschirm Pixels aufgetreten sein. Der Signalclient zeichnet dann eine senkrechte Linie oder ein senkrechtes Band anstelle eines einzelnen Pegels. Folgende Bildausschnitte sollen das verdeutlichen:



Zeitbasis 0.5ms



Zeitbasis 50μs



Zeitbasis 10μs

Alle drei Darstellungen betreffen den gleichen Signalausschnitt, allerdings mit einer von oben nach unten kleiner werdenden Zeitbasis, bzw. höheren Auflösung.

15.2 Navigation

Die Scrolleiste unter den Signalen gibt Ihnen einen Überblick über die Position und Größe des dargestellten Signalausschnittes im Vergleich zum Gesamtsignal. Der 'Anfasser' der Scrolleiste repräsentiert dabei die Größe, seine Position den Offset des angezeigten Ausschnitts.
Daneben ermöglicht Ihnen die Scrolleiste ein Navigieren durch das Gesamtsignal. Die Pfeilknöpfe am linken und rechten Ende der Scrolleiste scrollen den Signalausschnitt in Raster- bzw. 10 Raster Schritten, oder sie verschieben das Signal mit dem Anfasser. Mit den Pfeiltasten läßt sich das Signal ebenfalls verschieben, siehe Kurzbefehle.
Position und Vergrößerung (Zeitbasis) werden in den beiden linken Statusfeldern angezeigt. Unterhalb der Scrolleiste sehen Sie einen Schieberegler (Slider), mit dem Sie die Signalhöhe aller Signale verringern können. Im allgemeiner werden Sie diese Funktion nicht benötigen. Sie ist sinnvoll, wenn Sie den Namen einer eingeblendeten Region nicht lesen können, weil er vom Signal verdeckt wird. In diesem Fall verringern Sie einfach die Signalhöhe.

15.2.1 Navigation und Zoom mit dem Mausrad

Die Navigation mit dem Mausrad bietet sozusagen einen 'mittleren' Gang um das Signal zu verschieben. Halten Sie dazu die Umschalt Taste gedrückt, während Sie am Mausrad drehen. Das Signal wandert je nach Drehrichtung um jeweils 10 Raster nach links oder rechts. Ohne gedrückte Umschalt Taste bewegt sich das Signal in 1 Raster Schritten. Bei gleichzeitig gedrückter Strg Taste wird das Signal um den Mauszeiger vergrößert bzw. verkleinert.

15.2.2 Verschieben mit dem Hand-Werkzeug

Das Hand Werkzeug erlaubt das pixelweise Verschieben des Signals. Klicken Sie dazu auf das Hand Icon in der Toolbar. Der Cursor wechselt zu einem Handsymbol. Um das Signal nach rechts oder links zu bewegen, fassen Sie das Signal, indem Sie die linke Maustaste gedrückt halten und das Signal (bei gedrückter) Taste in die gewünschte Richtung ziehen. Der Mauscursor hat dabei die Form einer zupackenden Hand.

15.3 Die Zeitbasis

Die Zeitbasis entspricht dem Vergrößerungsgrad, mit welchem das Signal dargestellt wird. Die kleinste Zeitbasis ist 10ns, d.h. 10 Nanosekunden pro Raster und bedeutet 1ns pro angezeigtem Pixel (1 Raster ist 10 Pixel breit). Das Signal ist stark vergrößert, bei einer Bildschirmauflösung von 1024 sind das ungefähr 1μs des Signals (sofern Sie das Signalmonitor Fenster maximiert haben).

Sind die Pegeländerungen im Millisekunden oder Sekundenbereich werden Sie eher eine größere Zeitbasis wählen um einen zeitlich größeren Ausschnitt überschauen zu können. Durch Anklicken der beiden Lupen Symbole in der

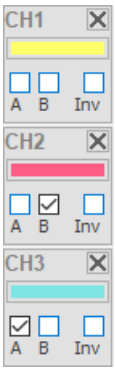
Werkzeugleiste wird die Zeitbasis jeweils auf den nächst größeren bzw. kleineren Wert gesetzt. Dasselbe erreichen Sie durch die Tastenkombination Strg + Pfeil nach oben und Strg + Pfeil nach unten bzw. durch Drehen des Mausrades.

Sie können auch einen bestimmten Ausschnitt des Signals vergrößern, indem Sie den Ausschnitt mit der gedrückter linker Maustaste auswählen. Bewegen Sie dazu den Mauszeiger an den Anfang des gewünschten Bereiches und drücken Sie die linke Maustaste. Halten Sie die Taste gedrückt und bewegen Sie den Mauszeiger an das Ende des Ausschnitts. Ein Rechteck zeigt Ihnen dabei mit jeder Bewegung die aktuelle Auswahl. Sobald Sie die linke Maustaste loslassen, wird dieser Bereich vergrößert dargestellt.

15.4 Undo und Redo

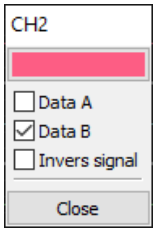
Alle Ausschnittsvergrößerungen können beliebig zurückgenommen (undo) oder nach einem Zurücknehmen wieder hergestellt (redo) werden.
Auf diese Weise können Sie einen interessanten Signalabschnitt beliebig vergrößern, z.B. um einen Cursor möglichst genau zu plazieren und anschliessend wieder zu einer der vorherigen Ansichten zurückkehren, indem Sie auf das Rückg. Symbol in der Werkzeugleiste klicken oder einfach Strg + Z eingeben.
Die ursprüngliche vor einem Rückgängig machen vorhandene Ansicht wird wieder hergestellt, wenn auf das Wiederh. Symbol geklickt wird. Beide Symbole sind als inaktiv (grau) gekennzeichnet, wenn keine weiteren Undo/Redo Schritte mehr möglich sind. Undo und Redo sind nur bei der Ausschnittsvergrößerung relevant. Ein normales Vergrößern oder Verkleinern des Signals ist jederzeit möglich, sodaß hier ein Undo/Redo unnötig ist.

15.5 Signalkontrollfeld



Signalkontrollfeld

Jedes angezeigte Signal besitzt auf der linken Seite sein eigenes Kontrollfeld. Dieses ersetzt mit Version 5.0 den bisherigen Einstelldialog und ermöglicht einen deutlich bequemeren und schnelleren Zugriff zu allen Signal relevanten Eigenschaften.
Hier können Sie einfach die Signalfarbe ändern, Daten einblenden, das Signal invertieren oder ganz ausschalten. Ausgeblendete Signale können Sie jederzeit im Einstelldialog des Signalmonitors wieder einschalten. Sie können sogar einzelne Kontrollfelder per Drag and Drop nach unten oder oben ziehen und damit die angezeigte Signalreihenfolge neu ordnen. All dies per einfachem Klick.
Falls die Höhe des Signalkanals zu klein für die korrekte Darstellung aller Kontrollelemente ist (z.B. wenn Sie die Höhe des Signalmonitor Fensters reduziert haben), öffnet sich bei Klick auf den Knopf für die Signalfarbe ein entsprechender Dialog mit den entsprechenden Eingabe Elementen. Der Dialog ist ebenfalls auf der rechten Seite abgebildet.



Signalkontroll Dialog

15.5.1 Signal entfernen/ausblenden

Sie können ein Signal jederzeit einfach per Klick auf den Schliessen Knopf (X) im Signalkontrollfeld ausblenden. Dies macht z.B. Sinn, wenn keine Signalpegel für diesen Kanal aufgenommen wurden oder Sie nicht an dem Signal interessiert sind.

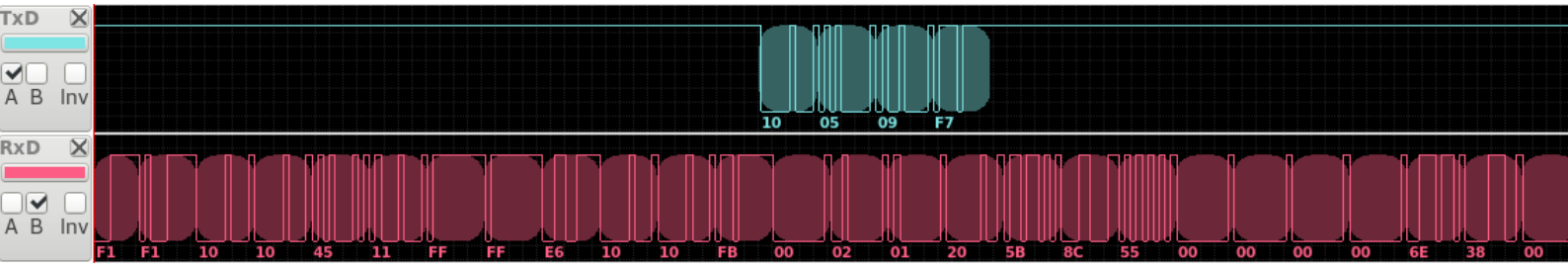
Die eigentliche Aufzeichnung wird davon nicht beeinflusst! Ausgeblendete Signale sind per Kontrollfeld in der Signalanzeige nicht mehr erreichbar (sie sind ja ausgeblendet). Sie können diese aber im Einstelldialog erneut aktivieren, siehe Abschnitt 15.6.

15.5.2 Signalfarbe

Der Signalmonitor definiert acht unterschiedliche Signalfarben die am besten zu den vorgefertigten Farbthemen passen. Sie können diese aber nach belieben ändern indem Sie einfach den Farbknopf im zugehörigen Kontrollfeld klicken. Das sich öffnende Fenster präsentiert den OS typischen Farbdialog wo Sie entweder eine bereits vorhandene Farbe auswählen oder eine neue erstellen können.

15.5.3 Daten Overlay

Jedes Signal kann mit den empfangenen Daten von Datenkanal A oder B überlagert werden. Der Signalmonitor berechnet die zugehörigen Datenrahmen (die Zeit vom Start- bis zum Stopbit) und überlagert die Daten als transparente, abgerundete Rechtecke mit entsprechendem Hexwert des jeweiligen Datenbytes.



Data overlay

Die abgerundeten Ecken dienen als sichtbare Abgrenzung zwischen den einzelnen Datenbytes. Anfang und Länge werden dabei von der internen Zeitmessung des Analysers bestimmt. Ist der überlagerte Datenbereich nicht passend zum Tri-State Signalverlauf, deutet dies auf eine falsche Einstellung der Datenrate entweder bei Ihrer Aufzeichnung oder einem bzw. mehrerer Bus Teilnehmer hin.

15.5.4 Signal Invertierung

Jedes Signal kann individuell invertiert werden indem Sie das zugehörige **Inv** Kästchen im Kontrollfeld des Signals aktivieren. Per Voreinstellung erfolgt die Darstellung so, wie das Signal vom Analyser aufgezeichnet wird.
Beachten Sie, daß diese Invertierung lediglich die Darstellung betrifft und keinerlei Einfluss auf die aufgezeichneten Signale und Daten hat. Nicht desto trotz kann eine invertierte Darstellung manchmal recht hilfreich sein. Z.B. wenn Sie sich über den korrekten Anschluss der Bus Leitungen unklar sind, und die übertragenen Signale zunächst mit dem integrierten Rahmenlineal überprüfen wollen (siehe Abschnitt 15.8).

15.5.5 Signalreihenfolge ändern

Die Signalreihenfolge entspricht per Default der Anzeige im Kontrollprogramm. Sie können diese aber sehr einfach und intuitive per Drag and Drop an Ihre

eigenen Wünsche anpassen.
Klicken Sie dazu auf den Namen im Kontrollfeld des Signals welches Sie verschieben möchten. Halten Sie die linke Maustaste gedrückt und ziehen Sie den Namen in das Kontrollfeld, an welchem Sie Ihr Signal stattdessen einfügen möchten.
Beim Verschieben des Signals in eine höhere Position wird das Signal vor diesem eingefügt. Wenn Sie das Signal nach unter verschieben, erfolgt die Darstellung dann hinter diesem.
Der Name des zu verschiebenden Signals wird während des Drag and Drop am Mauszeiger eingeblendet.

Signals neu anordnen
via drag and drop

15.6 Allgemeine Einstellungen

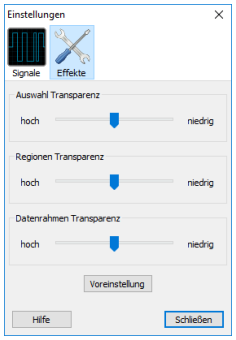
Sie werden den Einstelldialog vermutlich meistens dann aufrufen, wenn Sie ein zuvor aus der Anzeige entferntes Signal wieder aktivieren wollen.
Neben der Möglichkeit einzelne Signale ein/auszuschalten finden Sie hier auch allgemeine Farbeinstellungen. Dazu gehören u.a. die Hintergrund- und Rasterfarbe sowie die Auswahl eines Farbthemas!
Aktuell bietet der Signalmonitor zwei Farbthemen an. Ein 'Klassik Thema', welches der alten Farbgebung entspricht, und ein 'Dunkles Thema'. Letzteres sorgt für ein Erscheinungsbild ähnlich moderner Digital Oszilloskops mit diskretem Hintergrund Raster und speziell ausgewählten Signalfarben. Letztendlich ist es aber immer eine Frage des persönlichen Geschmacks.
Falls Ihnen keines der Farbthemen zusagt, können Sie Hintergrund, Raster und Signalfarben ganz nach Ihren Vorlieben einrichten.
Zukünftige Versionen werden weitere Farbthemen bereitstellen - und Ihnen vielleicht sogar die Generierung eigener Themen erlauben.

Allg. Einstellungen
und Farbthemen

15.6.1 Grafikeffekte

Der Signalmonitor verwendet Transparenz Effekte zur Darstellung eines ausgewählten Signalbereichs sowie zur Einblendung von Regionen und Daten Overlays. Der Grad der Transparenz ist wie je nach Anwender und Anwendung unterschiedlich und kann hier entsprechend vorgegeben werden.

- **Auswahl:** Bestimmt die Durchlässigkeit des per Maus ausgewählten Signalbereichs. Ein niedriger Wert erhöht die Sichtbarkeit der Auswahl, verdeckt aber zunehmend das Signal dahinter.
- **Regionen:** Regionen werden ebenfalls über das Signal gelegt. Auch hier gilt: Ein niedriger Wert erhöht die Sichtbarkeit des Signals, macht aber die Region selbst schlechter erkennbar.
- **Daten Overlay:** Regelt die Transparenz der eingeblendeten Daten und Datenrahmen. Ein niedriger Wert blendet das Daten Overlay aus, während ein zu hoher Wert das Signal überdeckt.



Grafikeffekte
für Transparenz

15.7 Cursor Steuerung

Jeder Signalmonitor besitzt zwei Cursor I und II die beliebig innerhalb des sichtbaren Bereichs über das Signal bewegt werden können. Um einen Cursor zu verschieben klicken Sie auf das entsprechende Cursor Symbol, ein auf dem Kopf stehendes Dreieck über der Signal Zeitachse und bewegen die Cursorlinie bei gedrückter Maustaste auf die von Ihnen gewünschte Position. Wenn

beide Cursor auf der selben Position stehen, können Sie nur den zuletzt aktiven sehen, da er den darunter liegenden verdeckt. Allerdings wird in diesem Fall immer Cursor I aktiviert. Um den zweiten Cursor zu verschieben, halten Sie die Umschalt Taste gedrückt während Sie auf die übereinander liegenden Cursor drücken. Jetzt können Sie Cursor II verschieben und Cursor I bleibt an Ort und Stelle.

Cursor Auswahl

Mit gedrückter Umschalt Taste wird bei übereinander liegenden Cursor der Zweite aktiviert.

Plazierte Cursor behalten ihre Signal spezifische Position, auch wenn Sie einen anderen Signalausschnitt auswählen. Cursor außerhalb des sichtbaren Bereichs werden dabei am linken oder rechten Rand dargestellt, je nachdem, in welcher Signalrichtung sie sich befinden. Ihre aktuelle Position können Sie in der Statuszeile ablesen. c1 bedeutet Cursor I, c2 Cursor II. Zusätzlich zur Position der einzelnen Cursor wird auch ihre Zeitdifferenz in der Statuszeile eingeblendet.
Damit ist es möglich beliebige Signalbereiche ausmessen, z.B. die Dauer einer aktiven Leitung etc.
Um mehrere Signalausschnitte zu vergleichen, können Sie den durch die beiden Cursor markierten Signalbereich einer sogenannten Region zuweisen. Klicken Sie dazu den '+Region' Knopf in der Werkzeugleiste oder drücken Sie F4. Maximal können 8 Regionen definiert werden. Der Bereich zwischen den Cursor wird dabei farblich hinterlegt. Mehr über Regionen erfahren Sie im Kapitel Regionen. Sie können auch beide Cursor gleichzeitig an eine andere Signalstelle bewegen, z.B. um die Zeitdauer zweier Signaländerungen zu vergleichen. Die Cursor werden verschränkt, indem Sie den 'c1+c2' Knopf in der Toolbar betätigen. Solange dieser Knopf gedrückt bleibt, werden immer beide Cursor bewegt, egal welchen von beiden Sie verschieben.

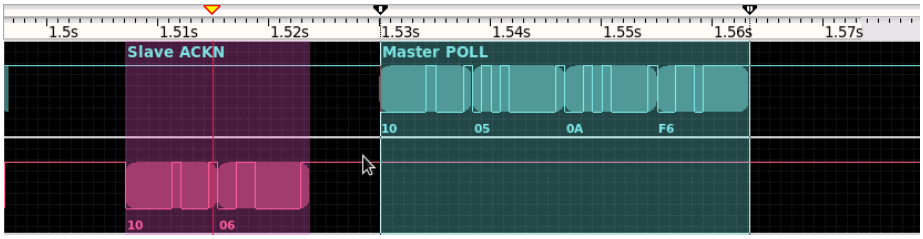
15.7.1 Auswahl

Der Signalabschnitt zwischen beiden Cursorn stellt zu jedem Zeitpunkt die aktuelle Auswahl dar. Sie müssen also nicht erst explizit eine Auswahl treffen. Da die Cursor ihre Signal relevante Position nicht verändern, bleibt ihre Position und ihr Abstand untereinander auch bei Veränderung der Signalansicht bestehen. Beides wird in der Statuszeile angezeigt.
Alle Operationen, die sich auf die aktuelle Auswahl beziehen, betreffen damit immer den Signalbereich zwischen den Cursorn.
Um eine Auswahl als Region zu definieren klicken Sie auf das '+Region' Symbol in der Werkzeugleiste oder drücken Sie die Taste F4.

15.7.2 Regionen

Regionen werden im Signalmonitor farbig und transparent über die eigentlichen Tri-State Signale gelegt.
Das folgende Bild zeigt zwei Regionen, wobei die hellblaue durch die beiden Cursor eingerahmt wird und vermutlich von diesen ausgewählt wurde. Da Regionen übergeordnete Bereiche sind und immer für alle Analysefenster gleichzeitig gelten, lassen sich auf diese Weise gezielt Signalausschnitte markieren

und in den unterschiedlichen Betrachtungsarten, die die einzelnen Analysefenster bieten, untersuchen.

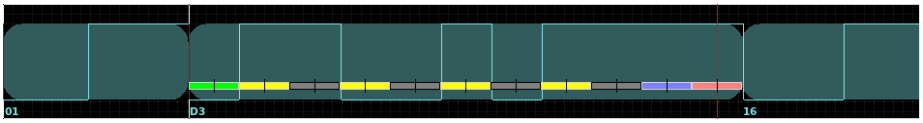


Bei dem rot-gelben Dreieck in der Cursor Leiste handelt es sich im übrigen um das aktuelle Synchronisierungs Ereignis, empfangen von einem anderen Analysefenster. Jede Region kann individuell mit einem Namen versehen werden. Im Beispiel hier die linke Region mit 'Slave ACKN', die rechte mit 'Master POLL'. Der Ausschnitt ist dabei einer DF1 Übertragung entnommen. Regionen werden im Detail in Kapitel 16 erläutert.

15.8 Datenrahmen mit Rahmenlineal ausmessen

Bei der Analyse asynchroner Übertragungen ist es manchmal hilfreich, einzelne Datenrahmen einer genaueren Überprüfung zu unterziehen. Z.B. um zu sehen, das das Parity Bit gemäß den Vorgaben gesetzt ist, die Daten zwischen Start- und Stopbit korrekt übertragen wurden - oder evtl. sogar eine Abweichung des Bit Taktes (Bit Weite oder Jitter) vorliegt.
In einer asynchronen Übertragung synchronisieren die Bus Teilnehmer ihre interne Takterzeugung für Signalabtastung mit der fallenden Flanke des Startbits. Der maximale Taktunterschied zwischen Sender und Empfänger darf dabei 2% nicht überschreiten, ungültige Daten wären sonst die Folge. Abhängig vom verwendeten Datenformat (Datenbits, Parity) muss eine solche Abweichung nicht unbedingt zu einem Parity oder Rahmenfehler führen und ist ohne genaue Betrachtung des Datensignals nur schwer zu finden.

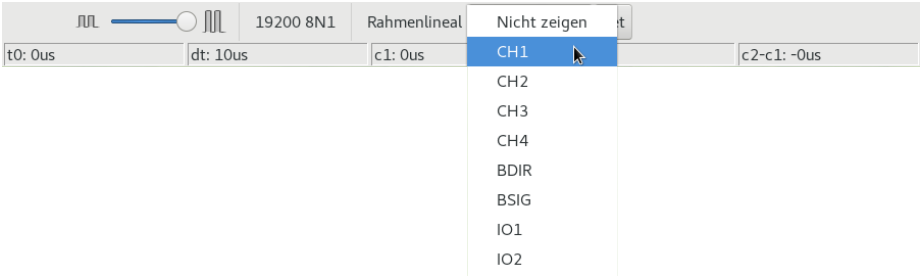
Genau für diese Zwecke enthält der Signalmonitor ein sogenanntes Rahmenlineal, siehe auch folgendes Bild. Sie können sich das Lineal als eine Art Massband vorstellen, welches einzelne Markierungen für das Startbit, die eigentlichen Datenbits (wie spezifiziert), das Parity Bit (falls verwendet) und Stopbit enthält.
Per Voreinstellung übernimmt das Lineal die bei der Aufzeichnung gemachten Baudrate und Datenformat Einstellungen.



Das Bildbeispiel zeigt ein Rahmenlineal für das Datenformat 8E1. Die Markierungen oder Felder sind: 1 Startbit (grün), 8 Datenbits (abwechselnd gelb, dunkelgrau), das Parity Bit (gerade Parität) in blau und das Stopbit (rot). Zusätzlich zeigt das Lineal die Abtastposition als vertikale Linie in der Mitte der

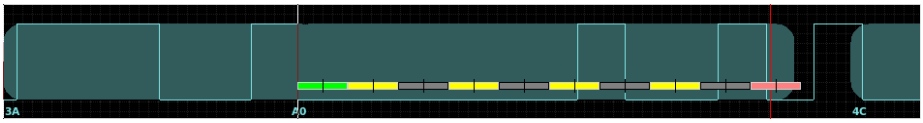
einzelnen Felder an.

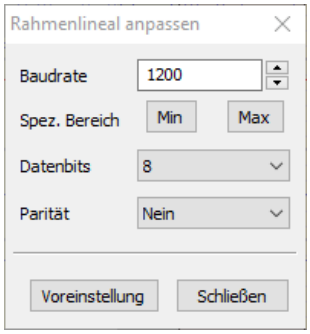
Sie können das Rahmenlineal einem beliebigen Signal zuweisen, indem Sie das entsprechende Signal in der Auswahlliste des Rahmenlineals (unterhalb des horizontalen Scrollbalkens) anklicken.



Anschließend können Sie das Lineal durch Ziehen von Cursor 1 (siehe vorheriger Abschnitt) genau mit der fallenden Flanke des Startbits positionieren. Die fallende Startbitflanke ist durch den eingeblendeten Datenwert (D3) bzw. dem linken Rand des angedeuteten Datenrahmens (hellroter Hintergrund im Bild) markiert.
Das niedrigste Bit wird immer zuerst übertragen und folgt unmittelbar auf das Startbit. Hier sehen wir eine Bitsequenz von 11001011 was hexadezimal De entspricht. Die Anzahl der Bits mit logisch 1 ist 5 und ungerade, das Parity Bit deshalb logisch 1 (gerade Parität). Das Stopbit muss immer logisch 1 sein und ist ebenfalls korrekt. Auch der Bittakt bzw. die Bitbreite entspricht exakt den Vorgaben.

Im Falle einer abweichenden Bitrate z.B. seitens des Senders, startet jedes auf das Startbit folgende Bit im Datenrahmen entweder etwas früher oder später. Dabei addieren sich die einzelnen Abweichungen zu einer evtl. nicht mehr tolerierbaren Verschiebung. Das Resultat ist eine falsche Bitzuweisung durch den Empfänger beim Abtasten und damit ein letztendlich ein falscher Datenwert. Im folgenden Bild wird ein als hexadezimal 20 übertragenes Datenbyte als hexadezimal A0 mit falschen Stopbit erkannt.



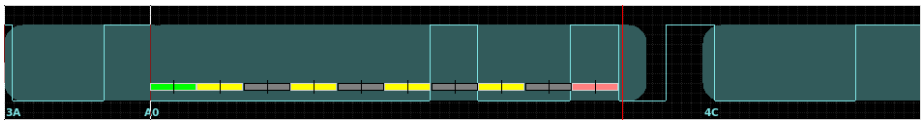


Rahmenlineal Dialog
zur Lineal Anpassung

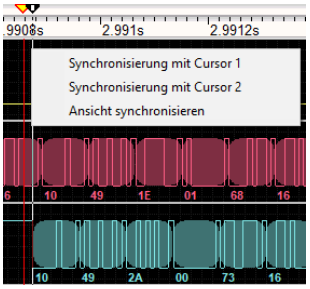
15.8.1 Rahmenlineal anpassen

Sie können die Darstellung des Rahmenlineals jederzeit an eine abweichende Baudrate oder an ein anderes Datenformat anpassen. Klicken Sie dazu einfach auf den **Set** Knopf rechts neben der Rahmenlineal Auswahl um den Rahmendialog zu öffnen. Per Vorgabe verwendet der Dialog das Datenformat und die Baudrate aus den Aufnahme Einstellungen.
Im Rahmendialog (siehe Randbild) können Sie eine beliebige andere Baudrate vorgeben oder einfach den Rahmen auf das Minimum bzw. Maximum des tolerierten Bereichs setzen. Zudem können Sie die Anzahl der Datenbits sowie die Parität (Parity Bit) verändern. Alle Einstellung wirken sich unmittelbar auf

die Anzeige des Lineals aus und erlauben so eine einfache Überprüfung der Signalübertragung ausgewählter Datenbytes.
Per Rahmenlineal ist es einfach zu prüfen ob das Signal des ausgewählten Datenbytes den Vorgaben entspricht und welche Bitrate verwendet wird - und dass ohne umständliches Ausmessen und Umrechnen der Bitbreite mit Hilfe der Cursor. In unserem Beispiel ist die zum Signal (Datenrahmen) passende Baudrate 20600.



15.9 Synchronisierung



Synchronisierung anderer Views per Cursor

Jedes Analysefenster kann seine Ansicht mit anderen synchronisieren (Siehe hierzu Synchronisierung der Datenansicht).
Wie sich der Signalmonitor beim Empfang eines Sync.-Signals (durch ein anderes, die Eingabe besitzendes Analysefenster) verhält, bestimmen die bei jedem Analysefenster identischen Sync. Knöpfe in der Cursor Steuerung.
Per default ist die Ansicht des Signalmonitors verriegelt, d.h. es reagiert auf keine Änderungen durch andere Analysetools. Klicken Sie auf das 'Sync' Symbol, und in der Cursorleiste erscheint ein rot-gelbes Dreieck, welches die Position des aktuellen Synchronisierungs Ereignis anzeigt. Bei Einschalten des 'Scroll' Symbols wird der Signalmonitor immer auf das zuletzt aufgetretene Ereignis synchronieren, d.h. die Signalansicht so wählen, daß immer das letzte Ereignis bzw. Pegelwechsel sichtbar ist.
Der Signalmonitor kann aber nicht nur auf eine Synchronisierung reagieren, er kann auch selbst eine Synchronisierung erzwingen.
Klicken Sie dazu im Signalbereich die rechte Maustaste um das Sync. Menü zu öffnen. Die Einträge sind mehr oder weniger selbst erklärend:

1 **Synchronisierung mit Cursor 1**
Synchronisiert wird auf das erste auf Cursor 1 folgende Ereignis

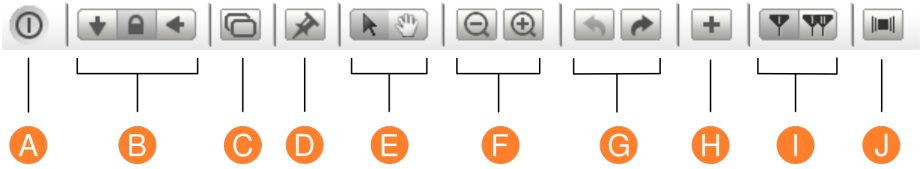
2 **Synchronisierung mit Cursor 2**
Synchronisiert wird auf das erste auf Cursor 2 folgende Ereignis

3 **Ansicht synchronisieren**
Als Synchron Ereignis wird der aktuelle Signalausschnitt genommen (genau genommen der erste Pegelwechsel vom linken Rand an gesehen)

Warum wird auf das nächstfolgende Ereignis und nicht auf die aktuelle Cursor Position synchronisiert?
Synchronisiert wird immer auf Ereignisse, nicht auf eine bestimmte Zeitmarke im Signal bzw. in der Aufzeichnung. Da der Cursor im Gegensatz zu den anderen Analysefenstern auch zwischen zwei Ereignissen stehen kann (z.B. zwischen einem Signalpegelwechsel innerhalb von wenigen Mikrosekunden) muß der auf den Cursor nächst folgende Pegelwechsel als Ereignis verwendet werden.

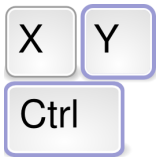
15.10 Die Werkzeugleiste

Die Werkzeugleiste dient zum schnellen Zugriff der am meisten benötigten Funktionen. Einige davon sind bei allen Views indentisch, andere spezifisch für den Signalmonitor.



A	**Ende:** Speichert alle Einstellungen und schließt das Fenster
B	**Anzeigemodus:** Je nach Anzeigemodus zeigt der Fensterinhalt immer das aktuelle (zuletzt aufgenommene) Ereignis, ist verriegelt oder aktualisiert den Inhalt mit anderen Fenstern.
C	**Neue Ansicht:** Öffnet ein neues Fenster mit dem gleichen Ausschnitt und identischen Einstellungen.
D	**Default Einstellung:** Speichert die aktuellen Signalmonitor Einstellungen als Vorgabe wenn ein neues Signalmonitor Fenster geöffnet wird.
E	**Maussteuerung:** Die Maus kann wahlweise zur Auswahl Vergrößerung oder zur Verschiebung des Signals (Handsymbol) verwendet werden.
F	**Signal vergrößern:** Vergrößert oder verkleinert den gerade sichtbaren Ausschnitt in 1, 2, 5 Faktoren, indem es die nächste jeweils kleinere oder größere Zeitbasis wählt.
G	**Undo/Redo:** Nimmt die letzte Ausschnittsveränderung (Zooming, Ausschnittvergrößerung) zurück bzw. stellt diese wieder her.
H	**Region hinzufügen:** Speichert Bereich zwischen beiden Cursor als neue Region.
I	**Cursor verschränken:** Die Cursor können wahlweise einzeln oder zusammen (verschränkt) bewegt werden.
J	**Region Dialog:** Öffnet den Regiondialog um z.B. Regionen ein/auszuschalten, zu löschen oder ihnen einen Namen zu geben.

15.11 Kurzbefehle



Tastenkommandos der wichtigsten Funktionen

Aktion	Kurzbefehl
Online Hilfe zu Signalmonitor	F1
Auswahl/Vergrößerung rückgängig machen	Strg + Z
Letzte Auswahl/Vergrößerung wieder herstellen	Strg + Y
Bereich zwischen Cursor als Region hinzufügen	F4

Bewegt Ausschnitt 1 Raster Richtung Signalende	Pfeil nach rechts
Bewegt Ausschnitt 1 Raster Richtung Signalanfang	Pfeil nach links
Bewegt Ausschnitt 10 Raster Richtung Signalanfang	Umschalt + Pfeil nach links
Bewegt Ausschnitt 10 Raster Richtung Signalende	Umschalt + Pfeil nach rechts
Bewegt Signalausschnitt um 1 Raster horiontal	Mausrad
Bewegt Signalausschnitt um 10 Raster horiontal	Umschalt + Mausrad
Ansicht vergrößern	Strg + +
Ansicht verkleinern	Strg + −
Verkleinern/Vergrößern um Mausposition	Strg + Mausrad
Komplette Ansicht	Strg + Pos1
Springe zu erstem Ereignis	Pos1
Springe zu letztem Ereignis	End
In einem neuen Fenster anzeigen	Strg + Umschalt + N
Einstellungen speichern und Fenster schliessen	Strg + Q

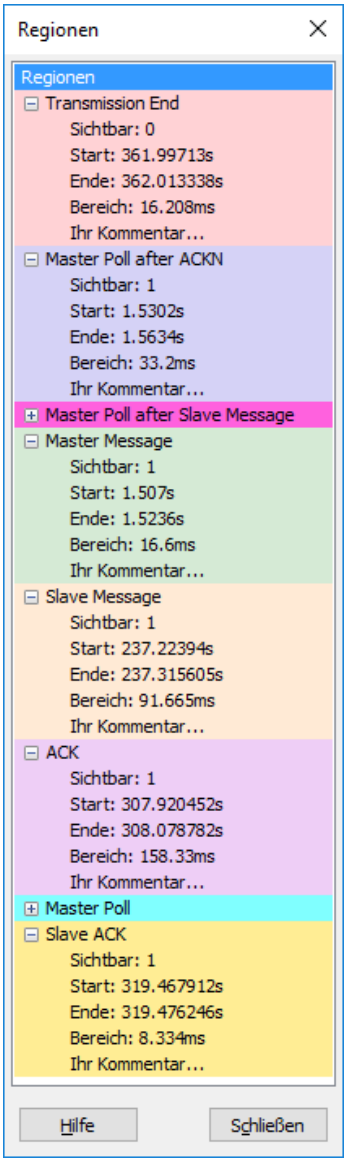
16 Regionen

Um besonders interessante Bereiche der Aufzeichnung schnell wieder zu finden, können diese als Region markiert werden. Regionen sind in allen Ansichten präsent, so dass eine im Datenmonitor markierte Region ebenfalls in der Signaldarstellung berücksichtigt wird. Regionen können gezielt angesprungen werden.

Regionen sind ausgewählte Bereiche, die in allen Analysefenstern angezeigt werden. Dabei kann jedes Analysefenster einen ausgewählten Bereich als Region definieren und diesen Bereich damit gleichzeitig in anderen Fenstern sichtbar machen. Da die verschiedenen Analysetools unterschiedliche Sichtweisen der aufgezeichneten Daten repräsentieren, kann eine Region auch völlig unterschiedlich definiert werden. Sei es im Signalmonitor durch die Auswahl eines bestimmten Signalabschnittes, im Datenmonitor durch eine bestimmte Datensequenz oder das Auftreten einzelner Zeichen und im Protokollmonitor durch eine Auswahl von Telegrammen.

Interessant wird diese Wechselwirkung, wenn Sie einen in einem Analysefenster definierten Abschnitt in einer anderen Ansicht betrachten wollen, z.B. den physikalischen Signalverlauf (Signalmonitor) einer Datensequenz (Datenmonitor). Sobald Sie in einem Analysefenster einen ausgewählten Bereich als Region hinzufügen wird diese an die Liste der vorhandenen Regionen angehängt.
Der Regionen Dialog ist dabei wie eine Liste von Lesezeichen über die Sie gezielt besonders interessante Bereiche der Aufzeichnung einfach anspringen können.
Regionen sind Teil einer Aufzeichnung und werden beim Sichern der Aufzeichnung automatisch in dieser gespeichert.

Die Analyser Software erlaubt die Definition von 8 Regionen. Jede Region kann individuell bezeichnet und wahlweise ein/ausgeblendet werden. Mit Ausnahme des Leitungsstatus Monitors öffnen Sie den Regiondialog unter:
`Ansicht→Regionendialog anzeigen` aus jedem Analysefenster heraus. Ein bereits geöffneter Regiondialog wird dabei automatisch in den Vordergrund geholt.
Die rechte Abbildung zeigt den Regiondialog mit insgesamt 8 Regionen, wobei die Regionen 3 (Magenta) und 7 (Cyan) eingeklappt sind. Alle Regionen sicht sichtbare, angezeigt durch den Wert 1 in der `Sichtbar` Eigenschaft.



Regionen Dialog

16.1 Region ein/ausschalten

Jede Region kann individuell ein- oder ausgeblendet werden. Dies ist vor allem sinnvoll, wenn sich mehrere Regionen in der Ansicht eines Fensters überlappen und trotz Transparenz eine Zuordnung erschweren. Um den Sichtbarkeitszustand einer Region umzuschalten klicken Sie einfach auf die `Sichtbar` Eigenschaft. Ein Wert von 1 bedeutet eine eingeblendete Region, ein Wert von 0 eine ausgeblendete Region.

16.2 Eine Region löschen

Um eine Region zu löschen, wählen Sie diese zunächst aus, indem Sie auf den Region Namen klicken und dann die Taste Entf auf Ihrer Tastatur drücken. Bevor die Region endgültig entfernt wird, müssen Sie dies noch einmal bestätigen.
Das Entfernen einer Region wirkt sich in keinerlei Weise auf die aufgenommenen Daten aus. Es ist, als ob Sie ein Lesezeichen aus einem Buch entfernen. Die Seiten des Buches bleiben davon unberührt.

16.3 Namensgebung für Regionen

Per Voreinstellung werden Regionen in der Reihe Ihres Hinzufügens mit Region1 bis Region8 bezeichnet. Sie können jeder Region allerdings einen mehr selbst erklärenden Namen geben indem Sie den Namen doppelklicken und dann editieren. Mit Enter wird der eingegebene Name Region übernommen.
Ein Name kann beliebige Zeichen enthalten und bis zu 200 Zeichen umfassen, wobei zu lange Namen aus Gründen der Lesbarkeit eher vermieden werden sollten. Etwaige Beschreibungen zur Region können zu diesem Zweck im Kommentar Feld eingegeben werden[1].

16.4 Regionen anspringen

Bestimmte Bereiche der Aufzeichnung werden als Regionen ausgewählt, weil es sich um wichtige Abschnitte handelt. Und diese wollen Sie natürlich schnell in den sichtbaren Ausschnitt eines Analysefensters z.B. des Signal- oder Datenmonitors holen.
Eventuell wollen Sie auch einfach zwei Regionen vergleichen.
Der Regiondialog unterstützt deshalb den gleichen Mechanismus zur Synchronisierung wie die einzelnen Analysefenster. Allerdings kann er eine Synchronisierung nur initiieren. Klicken Sie dazu einfach mit der Maus auf den Startwert um den Anfang der Region in die Analysefenster zu holen, bzw. den Endwert, wenn Sie das Ende der Region betrachten wollen.
Beachten Sie, daß auch hier nur die Analysefenster auf dieses 'Synchronsignal' reagieren, deren Synchronisierung aktiviert ist.

[1] Das Kommentar Feld wird in einer späteren Version unterstützt

Region in Fenstern einblenden

Regionen können blitzschnell in den Anzeigebereich eines Analysefensters geholt werden, indem mit der linken Maustaste auf den Start- oder Endwert der Region geklickt wird. Das Fenster Modus muss dabei als synchronisierbar eingestellt sein!

16.5 Speicherung der Regionen

Regionen werden als Teil der Aufzeichnung (Aufnahme Datei msblog) automatisch mit abgespeichert. Sollten Sie ungesicherte Regionen haben, werden Sie beim Beenden des Programms oder beim Start/Laden einer neuen Aufzeichnung entsprechend gewarnt.

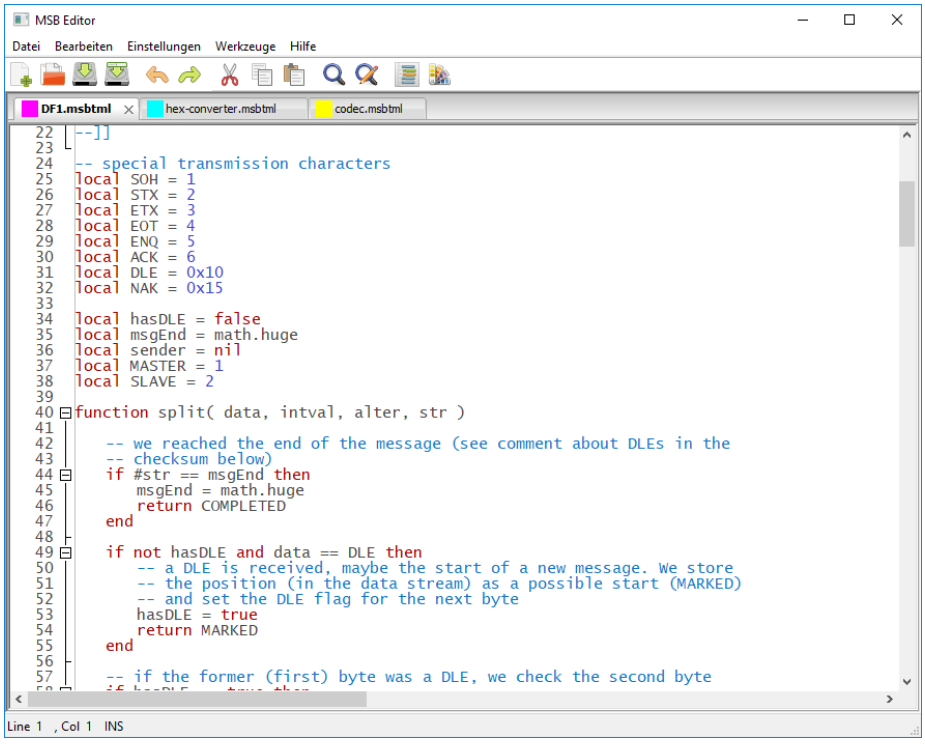
16.6 Region Eigenschaften

In der folgenden Tabelle sind alle Regionen Eigenschaften noch einmal der Übersicht halber aufgelistet. Alle Eigenschaften mit Ausnahme des Bereichs (der nur informellen Charakter hat) sind anklickbar.

Eigenschaft	Bedeutung
Region Name	Editierbarer Region Name
Sichtbar	0: Region ist unsichtbar (ausgeblendet), 1: Region wird in den diversen Programm Fenstern (Views) angezeigt.
Start	Der Start der Region in Sekunden relativ zum Aufnahmestart. Ein Klick bewirkt, das die synchronisierten Views den Beginn der Region einblenden.
Ende	Das Ende der Region in Sekunden relativ zum Aufnahmestart. Ein Klick bewirkt, das die synchronisierten Views das Ende der Region einblenden.
Bereich	Die Zeitdauer der Region in Sekunden (nicht klickbar).
Kommentar	Zusätzliche Beschreibung der Region, wird aktuell noch nicht unterstützt.

17 Der Editor

Lua wird ein zunehmend wichtigerer Bestandteil der Analyser Software. Die Vorteile einer extrem anpassungsfähigen Skriptsprache zur Definition von Protokoll spezifischen Telegrammen wurde mit Version 5.0.0 komplett auf den Datenmonitor übertragen und erlaubt nun auch dort die Verarbeitung von Protokoll relevanten Rohdaten.
Ein externer, voll ausgestatteter Lua Editor ist dabei die logische Konsequenz.

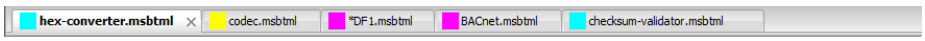


Der in der Analyser Software enthaltene Editor ist nicht nur ein einfacher Lua Skript Editor. Er besitzt vielmehr alle Vorzüge die Sie von einem guten Programm Editor erwarten können. Unter anderem Code Folding, Syntax Highligh-

ting, ein Multi-Dokument Interface, unbegrenztes Undo/redo und vieles mehr. Daneben erstellt er automatisch Code Gerüste für Protokoll Templates, Datenmonitor Skripte und Lua Module zur Verwendung in den ersten beiden.

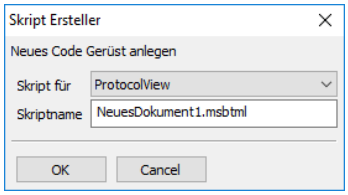
17.1 Den Editor starten

Gestartet wird der Editor aus dem Daten- bzw. Protokollmonitor indem Sie einfach den Knopf Neu/Editieren neben dem aktuell verwendeten Skript anklicken. Der Editor erscheint entweder mit dem aktuellen Template/Skript oder zeigt dieses als neuen Tab in einem bereits geöffneten Editor.
Durch das Multi-Dokument Interface (ein Tab pro Skriptdatei) können Sie einfach mehrere Skripte vergleichen und/oder Code Passagen zwischen ihnen austauschen.
Der Editor versieht dabei jeden Tab mit einem unterschiedlcih farbigen Rechteck, je nachdem, ob es sich um eine Skriptdatei für den Daten-, Protokollmonitor oder ein Lua Modul handelt.



Skripte für den Datenmonitor (d.h. im Datenmonitor Templates Ordner) werden mit einer Cyan Box angezeigt, Protokoll Templates mit Magenta und Lua Modul Dateien gelb. Dateien mit ungespeicherten Änderungen haben zudem ein vorangestelltes '∗' im Tab.
Sie können einzelne Skriptdateien einfach schliessen indem Sie auf den X im Tab klicken. Hat die Datei ungespeicherte Änderungen bekommen Sie eine entsprechende Warnung. Dies gilt auch, wenn Sie den Editor einfach beenden. Der Editor beendet sich niemals ohne Sie zuvor über veränderte Skripte zu informieren und Ihnen entsprechende Maßnahmen (Speichern) anzubieten.

17.2 Ein neues Skript anlegen



Skript Ersteller

Im Gegensatz zu vorherigen Versionen werden neue Lua Skripte (Templates) jetzt nur noch ausschließlich im Editor angelegt. Dies vermeidet verschiedene Versionen ein und derselben Skriptdatei z.B. bei zwei parallel geöffneten Protokollmonitoren.
Beim Anlegen einer neuen Datei frägt Sie der Editor nach dem Verwendungszweck des Skriptes (Datenmonitor, Protokollmonitor oder Modul) und liefert Ihnen ein entsprechendes Code Gerüst.
Die neue Skriptdatei wird dann voll allen Views (Daten- oder Protokollmonitoren) gemeinsam verwendet die diese ausgewählt haben.

Eine neue Skriptdatei auswählen

Beachten Sie! Eine neu angelegte Skriptdatei ist nicht auswählbar, bevor diese im Editor gespeichert wurde.

17.3 Interaktives Programmieren

Als integrierter Teil des MSB-Analyser Programmes interagiert der Editor automatisch mit den offenen Programm Fenstern (Views) deren Skript gerade edi-

tiert wird. Insbesondere um die Telegramm Struktur im Protokollmonitor oder die Lua Ausgabe in Datenmonitor interaktiv zu verändern. Dies macht das Programmieren von Telegramm Darstellungen oder Verarbeiten der Daten im Datenmonitor zu einem erstaunlich einfachen Vorgang.
Sobald Sie ein modifiziertes Skript gespeichert haben, werden die Änderungen automatisch in dem daten-/Protokollmonitor Fenster angewendet, dies dieses Skript ausgewählt haben. Klicken Sie einfach den Speichern Knopf in der Werkzeugleiste oder drücken Sie STRG+S und - Voila - die entsprechenden Views aktualisieren ihre Darstellung.

17.3.1 Lua Skriptfehler

Folgende Fehler können auftreten: Allgemeine Lua Fehler wie falsche Schlüsselworte (z.B. `functns` anstelle von `functions`), Modul fehler (Aufruf eines nicht existierenden Moduls oder Modulfunktion) und Laufzeitfehler. Letztere entstehen z.B. beim Teilen durch Null oder dem Zugriff auf eine nicht existierende oder ungültige (nil) Variable.
Der Editor selbst enthält keinen Lua Intepreter und kann daher weder das Skript noch die Ausführung testen. Dies obligt den eigentlichen Views, die die Skriptdatei verwenden. Fehler werden deshalb nicht im Editor sondern in dem entsprechenden Daten- bzw. Protokollmonitor mit Angabe der möglichen Ursache sowie Zeilennummer angezeigt.

17.4 Hervorheben individueller Schlüsselworte

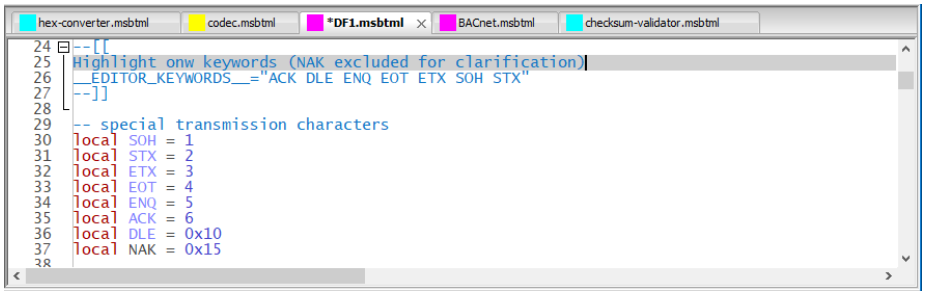
Lua Skripte, vor allem Protokoll Templates können schnell recht umfangreich werden. Die Programmierin ist deshalb über jede Hilfe dankbar, die das Lesen und Pflegen des Codes einfacher macht. Ein probates Mittel ist das Hervorheben von Lua Schlüsselworten, Variablen und Typen.
Der Editor erweitert diese Fähigkeit um die Möglichkeit, eigene beliebige Wörter hinzuzufügen, die dann ebenfalls in einer anderen Farbe hervorgehoben werden. Die neuen Wörter werden als Leerzeichen separierter String der Variable `__EDITOR_KEYWORDS__` zugewiesen.
Da der Editor das Skript nicht ausführt sondern nur nach einem Muster:

```
__EDITOR_KEYWORDS__="keyword1 keyword2 ..."
```

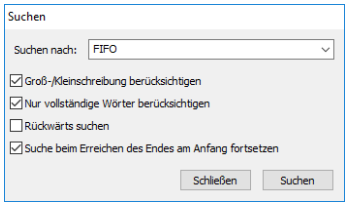
sucht, können Sie die Zuweisung auch in einen Kommentar stecken um etwaige Seiteneffekte zu vermeiden.
Das folgende Bild zeigt das Resultat der Zuweisung:

```
__EDITOR_KEYWORDS__="ACK DLE ENQ EOT ETX NAK SOH STX"
```

in einem `DF1` Protokoll Template.



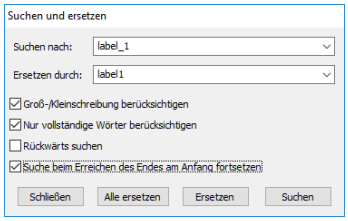
17.5 Suchen



Suche Dialog

Das Durchsuchen eines Programms nach bestimmten Begriffen oder Wortteilen ist oft verbunden mit zusätzlichen Angaben. Gesucht wird entweder ein vollständiges Wort oder nur der Teil eines Textes. Die Suche muss Groß- und Kleinschreibung entweder berücksichtigen oder ignorieren. Und die Suche soll rückwärts erfolgen und dann erneut von hinten starten.
Der MSB-Analyser Editor bietet einen einfachen nicht desto trotz aber leistungsfähigen Suchmechanismus. Klicken Sie einfach das Suchsymbol in der Werkzeugleiste oder STRG+F um den Suchdialog zu öffnen.
Der Dialog speichert dabei Ihre Suchangaben, so dass Sie diese beim nächsten Mal nicht erneut eingeben müssen.

17.6 Suchen und Ersetzen



Suchen & Ersetzen Dialog

Beim Schreiben von Programm Code ist es übliche Praxis Variablen oder Funktionsnamen später noch einmal umzubenennen um dem Namen eine bessere Bedeutung zu verleihen. Der MSB-Analyser Editor enthält hierfür einen mächtigen Such und Ersetzen Dialog, der es Ihnen erlaubt, Text Schritt für Schritt oder in einem Rutsch zu ersetzen. Ersteres ist besonders hilfreich, wenn Sie die Ersetzungen im Einzelnen erst prüfen möchten, bevor der Textaustausch erfolgt. Dabei können Sie einfach von einer gefundener Textstelle zur nächsten Springen und wahlweise den Text ersetzen oder den Text ignorieren.
Der Dialog unterstützt alle Vorgaben des Suchdialogs und merkt sich ebenfalls Ihre Angaben.
Sie können den Such & Ersetzen Dialog entweder per Werkzeugleiste oder STRG+H aufrufen.

17.7 Code Folding

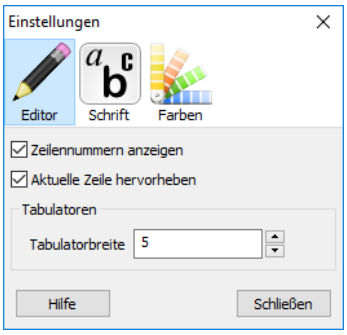
Code Folding ist ein nettes Feature when Ihr Skript eine Menge verschiedener Funktion oder anderer Code Blöcke wie z.B. Tabellen enthält. Wenn Sie das Code Folding in der Werkzeug Leiste (oder per ALT+T) aktivieren, wird jeder Anweisungsblock bis auf die erste Zeile eingeklappt. Im Falle einer Funktion ist das der Funktionsname, Tabellen werden auf die erste Tabellenzeile reduziert. Jede eingeklappte Block wird am linken Rand mit einem ⊞ gekennzeichnet. Sie können dort beliebige Blöcke durch Klicken ein- oder ausklappen. Oder aber alle Blöcke durch Klicken des Icons in der Werkzeugleiste bzw. ALT+T.

Suchen & Ersetzen bei eingeklapptem Code

Ein eingeklappter Code Block ist für den Suchdialog nicht 'sichtbar'. Allerdings wirkt sich das Ersetzen aller vorkommenden Texte auch auf eingefaltete Code Passagen aus!

17.8 Editor Einstellungen

Die Standard Einstellungen des Editors können im Einstelldialog nach eigenen Vorgaben angepasst werden. Dies betrifft u.a. die Eingabe bzw. Editiervorgaben wie Tabulator Einrückung, Hervorheben der aktuellen (Cursor) Zeile, Ein-/Ausblenden der Zeilennummern, die Schriftart (Vorgabe ist die Monospace Systemschrift und das Farbschema.
Der Editor bietet zwei unterschiedliche Farbthemen. Ein Klassik Thema mit weißem Hintergrund sowie ein dunkles Thema für Anwender, die eine helle Schrift auf dunklem Hintergrund bevorzugen.
Alle Änderungen werden sofort angewendet und für die aktuelle Sitzung gespeichert wenn der Dialog geschlossen wird.



Settings dialog

17.9 Farbassistent

Insbesondere der Protokollmonitor verwendet verschiedene Farben um Telegramminhalte zu illustrieren. Aber auch der Datenmonitor unterstützt Farben um bestimmte Datensequenzen zu markieren bzw. hervorzuheben.
Farben werden dabei in RGB Werten (rot, grün, blau Anteil) angegeben. In Lua als ein Integer Wert `0xRRGGBB` wobei RR, GG und BB den roten, grünen und blauen Anteil als Hexwert im Bereich von 00...FF entspricht. Ein reines Rot z.B. als 0xFF0000.
Die richtige Farbe nur durch Variieren des Farbwertes auszuwählen ist ziemlich umständlich und zeitraubend. Der Editor unterstützt Sie hier mit einem Farbassistenten (oder Farbcoder). Sie wählen einfach Ihre gewünschte Farbe und der entsprechende Integer Farbwert wird dann automatisch an der Cursorposition im Code eingefügt.
Sie können den Farbassistenten entweder in der Werkzeugleiste anklicken oder per CTRL+Alt+C öffnen.

17.10 Speicherort der Skriptdateien

Alle Skripte, sowohl die mit gelieferten als auch die von Ihnen geschriebenen werden per Voreinstellung im Applikations Datenverzeichnis des Anwenders abgelegt. Unter Linux ist dies:

```
~/.IFTOOLS/SerialAnalyzer/7.0.2/Templates/
```

Unter Windows:

```
C:\Users\USERNAME\AppData\Roaming\IFTOOLS\SerialAnalyzer\7.0.2\Templates\
```

Die Skriptdateien oder Templates werden dabei je nach Anwendungszweck (zugehöriges View) in unterschiedlichen Verzeichnissen einsortiert. Neben den

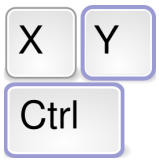
Verzeichnissen ProtocolView, DataView, Trigger und Modules gibt es einen zusätzlichen Ordner SwitchEditor. Letzterer enthält allerdings nur Schaltpläne der Schaltoption und keine Skriptdateien.
Sie können natürlich ein Skript auch unter einem beliebigen anderen Ort speichern. Z.B. wenn Sie das Skript mit einer anderen Anwendung weiter verarbeiten oder sich mit einem Kollegen austauschen wollen. Am besten verwenden Sie hierzu 'Als Kopie speichern...' im Dateimenü.
Beachten Sie aber, das sowohl Daten- wie auch Protokollmonitor nur Skripte im voreingestellten Verzeichnis verarbeiten und ausführen können, weil sie nur in diesem Verzeichnis Skripte überwachen.

Scripte speichern

Daten- und Protokollmonitor erwarten die Skripte in dem jeweiligen oben genannten vorgegebenen Verzeichnis. Andernfalls werden neue Skripte weder erkannt noch ausgeführt!

17.11 Editor Tastenkürzel

Die Anwendung des Editors ist so einfach wie möglich gehalten. Die allermeisten Funktionen sind dabei direkt in der Werkzeugleiste oder per rechtem Mausklick(auswählen, kopieren, einfügen) abrufbar. Ein paar wenige Tastenkürzel sind nichtdestotrotz wert gemerkt zu werden, da sie Ihnen einige zusätzliche Mausklicks ersparen.



Tastenkürzel
mit den wichtigsten Funktionen

Action	Short key
Ausgewählten Text in die Zwischenablage kopieren	Strg + C
Öffnet den Suchdialog	Strg + F
Öffnet den Such & Ersetzen Dialog	Strg + H
Ein/Ausschalten des Code Foldings	ALt + T
Eine neue Skriptdatei anlegen	Strg + N
Ein Skript in den Editor laden/öffnen	Strg + O
Das aktuelle Skript speichern und ein View Update triggern	Strg + S
Das aktuelle Skript unter einem anderen Namen speichern	Umschalt + Strg + S
Text aus der Zwischenablage an der Cursor Position einfügen	Strg + V
Ausgwählten Text ausschneiden und in die Zwischenablage kopieren	Strg + X
Redo des letzten Undo	Strg + Y
Undo der letzten Änderung	Strg + Z

Editorschrift vergrößern (zoom in)	Strg + ⊕
Editorschrift verkleinern (zoom out)	Strg + ⊖
Editorschrift vergrößern oder verkleinern via Mausrad	Strg+Wheel

18 Schnelleinstieg in Lua

Lua ist eine der schnellsten Skriptsprachen und auf Grund seines einfachen und klaren Designs sehr einfach zu lernen. Lua enthält einige wenige aber um so mächtigere Konzepte die es zur ersten Wahl bei der Erweiterung der Analyser Software machen.
Dieses Kapitel gibt Ihnen einen ersten Überblick über die Sprache und die Möglichkeiten, die sich dadurch bei der Analyse von Datenaufzeichnungen und Protokollen ergeben.

18.1 Erste Schritte

Lua ist eine Programmiersprache mit einer Reihe beeindruckender Eigenschaften und dabei gleichzeitig sehr schnell, kompakt und mit einfacher Syntax. Lua existiert mittlerweile in verschiedenen Versionen. In der Analyser Software verwenden wir Lua Version 5.3. Jetzt ist es an der Zeit ein wenig mehr über diese außergewöhnliche Sprache zu lernen.

Lua Version 5.3

Sie können alle nachfolgenden Beispiele im integrierten Lua Skripteditor selbst testen. Starten Sie dazu einfach die Analyser Software (ein Analyser Gerät ist dazu nicht nötig) und öffnen Sie den Lua Editor im Menü 'Ansicht' des Kontrollprogramms.

Der Lua Skript Editor erlaubt das Ausführen einzelner ausgewählter Code Zeilen einfach per Umschalt + F5. Oder Sie drücken F5 um das komplette Skript auszuführen. Die Code Evaluierung arbeitet in allen geöffneten Lua Dateien. Allerdings bietet der Editor mit einem speziellen ***SKETCH*** Buffer eine noch einfache Möglichkeit, eigenen Lua Code zu testen ohne dazu vorhandene Skripte modifizieren zu müssen.
Der Inhalt dieses Buffers wird automatisch beim Schließen des Buffers oder Editors gespeichert und wiederhergestellt wenn Sie den ***SKETCH*** Buffer erneut öffnen.

Die meisten Beschreibungen von Programmiersprachen beginnen mit dem traditionellen "Hello World". Wir wollen diese Gepfogenheit beibehalten und unser erstes Skript ein freundliches 'Hello World' ausgeben lassen
Öffnen Sie dazu den Sketch Buffer im Editor Lua Menü und geben Sie folgende Zeile ein:

```
1  print("Hello World")
```

Drücken Sie F5. Der Editor öffnet ein Lua Ausgabefenster und führt den Inhalt des Sketchbuffers aus. In der Ausgabe erscheint unsere Begrüßung `Hello world`.

Die Lua `print` Funktion ist eine schnelle und einfache Möglichkeit den Inhalt einer Variable oder hier eines Strings zu Testzwecken auszugeben. `print` wird mit einem oder mehrerer durch Kommas getrennter Parameter aufgerufen. Die Ausgabe mehrerer Resultate erfolgt dabei getrennt durch ein TAB Zeichen. Ein Beispiel:

```
1  print("Result of 1/3", 1/3 ) --> Result of 1/3   0.33333333333
```

Hier rufen wir die `print` Funktion mit zwei Argumenten aus. Das erste Argument ist ein in zwei Gänsefüßchen eingeschlossenen String, das zweite das Ergebnis eines arithmetischen Ausdrucks. Lua behandelt alle Zahlen intern im Fließkommaformat (siehe Seite 239. Das Resultat ist deshalb eine Realwert und keine Ganzzahl.

Die `print` Funktion kann auch als einfacher einzeiliger Taschenrechner dienen. Und da Lua die Eingabe von hexadezimalen Zahlen erlaubt, auch zur einfacher hexadezimal/dezimal Umrechnung. Betrachten Sie dazu folgende Zeilen:

```
1  print( 0xFFFF * 2 ) --> 131070
2  print( string.format( "%x", 123456 ) --> 1E240
3  print( string.format( 0xFFFF ~ 0x3000) ) --> CFFF
```

Hexadezimale Werte werden in Lua (wie auch in vielen anderen Sprachen) durch ein vorangestelltes `0x` angegeben, siehe Zeile 1. Die Ausgabe erfolgt immer dezimal. Sie können dies aber jederzeit mit Hilfe der String Format Funktion ändern. Diese arbeitet ähnlich der `printf` Funktion in C/C++.
Zeile 2 verwendet die vorzeichenlose hexadezimal Notation `"%x"` um das Ergebnis mit der Zahlenbasis 16 (hexadezimal) auszugeben.

Vermutlich fragen Sie sich an dieser Stelle was der Punkt zwischen `string` und `format` bedeutet?
Lua fasst alle Funktionen zur String Manipulation in einem separaten Module oder Bibliothek zusammen. Um eine Funktion innerhalb eines Moduls auszuführen, müssen Sie diese mit dem vorangestellten Modul bzw. Bibliotheksnamen UND dem Funktionsnamen getrennt durch einen Punkt aufrufen.

Zeile 3 zeigt am Beispiel des 'Exklusiv Oder' Operators die Verwendung der Lua internen bitweise oder Operatoren. Seit Lua Version 5.3 unterstützt Lua alle Bit Operatoren, die Sie auch von anderen Sprachen kennen. Mehr Informationen darüber finden Sie auf Seite 251.

18.1.1 Verwendung von Funktionen

`print` ist eine in Lua fest integrierte Funktion (neben vielen anderen). Aber Sie können jederzeit beliebig viele eigene Funktion hinzufügen und anschließend in Ihren Skripten verwenden. Alle Information über Lua Funktion finden Sie

auf Seite 254. Wir erweitern unser kleines Beispiel nun um eine Begrüßungs-Funktion.

```
1  function greeting( text, name )
2    print( text.." "..name )
3  end
4
5  greeting( "Hello", "Bob" )
```

Geben Sie alle Zeilen im SKETCH Buffer des Editors ein und drücken Sie F5 um das Skript auszuführen.
Die Funktion `greeting` wird mit zwei Argumenten aufgerufen. Dem Begrüßungstext und einem Namen. In der Funktion werden beide übergebenen Parameter und ein Leerzeichen `" "` mit Hilfe des Lua String Verkettungs Operators `..` zusammengefügt und ausgegeben.

```
Hello Bob
```

18.1.2 Funktionen mit mehreren Rückgabewerten

Unsere `greeting` Funktion hatte keinen Rückgabewert. Sie gibt lediglich die zusammengefügten Argumente aus. Im Gegensatz zu anderen Sprachen erlaubt Lua Funktionen mit einer beliebigen Anzahl von Rückgabewerten, siehe Seite 254. Stellen Sie sich eine Funktion vor, die zwei Integer Zahlen dividiert und sowohl den Quotient als auch den Rest liefert[1].

```
1  function divide( dividend, divisor )
2    local remainder = dividend % divisor
3    local quotient = ( dividend - remainder ) / divisor
4    return quotient, remainder
5  end
6
7  print( divide( 10, 3 ) ) --> 3   1
```

Da Lua alle Zahlen als Fliesskommawerte behandelt können wir nicht einfach 10 durch 3 teilen. Wir erhielten 3,3333333333333. Stattdessen berechnen wir zunächst den Rest mit Hilfe des Lua Modulo Operators `%` in Zeile 2.
Indem wir den Rest vom Dividend vor dem Teilen abziehen (Zeile 3) erhalten wir ein ganzzahliges Ergebnis für den Quotienten.

Das Schlüsselwort **local** in Zeile 2 und 3 schränkt die 'Sichtbarkeit' bzw. den Anwendungsbereich der Variablen `remainder` und `quotient` auf den umgebenden Code Block (hier die Funktion `divide` ein. Näheres dazu auf Seite 138.
Zeile 4 letztendlich gibt beide Werte per Komma getrennt zurück.

18.1.3 String Verarbeitung und Manipulation

Lua Strings können beliebige Zeichen und Bytewerte enthalten. Das Stringende ist nicht durch ein besonderes Zeichen gekennzeichnet. Eine Sonderbehandlung des Null Bytes wie z.B. bei C/C++ ist daher nicht erforderlich. Und: Unter Lua ist es besonders einfach, Zahlen und Strings zu mischen. Dies macht Lua Strings zu einem idealen Datentype um Protokoll Telegramme zu repräsentieren bzw. zu generieren.

Betrachten Sie das folgende String Beispiel:

[1] Dies ist lediglich ein Beispiel und funktioniert nur bei positiven Integer Zahlen!

```
print( string.dump( "\000\001\255\128" ) ) --> 00 01 FF 80
```

Sie können einen String mit beliebigen Bytewerten instantieren bzw. an einen String anfügen, indem Sie den 3-stelligen Dezimalwert des Bytes mit einem vorangestellten Backslash eingeben. In obigem Beispiel startet der String mit einem Null Byte, gefolgt von den binären Werten 1, 255 und 128.
Die `dump` Funktion ist eine Analyser Erweiterung des Lua `string` Moduls. Sie erlaubt die Ausgabe eines Strings in hexadezimaler Notation (neben anderen) und ist sehr hilfreich wenn Sie eigens generierte Telegramm Sequenzen auf ihre Korrektheit überprüfen wollen.

Das nächste Beispiel zeigt anhand einer einfachen Checksum Funktion wie Sie auf beliebige Bytes innerhalb eines Strings zugreifen können.
Die meisten Feldbus Protokolle verwenden Prüfsummen um die Korrektheit der übertragenen Datensequenzen sicherstellen zu können. Ein einfacher Prüfsummen Algorithmus ist die Module 256 Summe über alle zu sendenden Datenbytes[2]. Eine entsprechende Lua Funktion (Zeile 1...7) sieht wie folgt aus:

```
1  function Checksum( data )
2    local chksum = 0
3    for i=1,#data do
4      chksum = chksum + string.byte( data, i )
5    end
6    return chksum % 256
7  end
8
9  print( Checksum( "hello world" ) ) --> 92
```

Geben Sie die obigen Zeilen im Sketch Buffer des Editors ein und pressen Sie F5. Zeile 9 ruft die Checksum Funktion mit der Datensequenz bzw. dem Lua String `"Hello world"` auf. Das Resultat ist 92.

Wie funktioniert es?
Im Funktionsrumpf legen wir zunächst eine lokale Variable `chksum` an und initialisieren sie mit 0. Zeile 3 iteriert über alle im übergebenen String enthaltenen Bytes (oder Zeichen) wobei `i` den Index (die Position) des jeweiligen Zeichens enthält.

Beachten Sie hier: Im Gegensatz zu anderen Programmiersprachen zählt Lua Tabellen und String Positionen (Indexe) von 1 an und nicht von 0!
`#` in Zeile 3 ist in Lua als sogenannter Längen Operator definiert. Er gibt die Länge eines Strings oder Arrays zurück. Zum Beispiel liefert:

```
print( #"hello world" ) --> 11
```

das Resultat 11.
Zeile 3 zählt daher die Variable `i` von 1 (dem ersten Zeichen im String) bis 11 (dem letzten Zeichen).
Um den Bytewert eines Zeichens zu ermitteln verwenden wir die Lua String Funktion `string.byte(i)`[3].

[2] Optional verwendet z.B. in IEC60870-5-103
[3] Die byte Funktion bietet einen optionalen zweiten Index Parameter um die Werte gleich mehrerer Zeichen zu ermitteln. Hier verwenden wir die einfache Variante

In Zeile 4 werden alle Byte bzw Zeichenwerte des Strings aufaddiert nacheinander aufaddiert. Eine alternative, Objekt orientierte Variante um die `byte` aufzurufen ist:

```
chksum = chksum + data:byte( i )
```

Lua bietet mit dem Doppelpunkt `:` eine spezielle Syntax um auf Funktionen oder Variablen eines 'Objekts' zuzugreifen. Im Gegensatz zu einem Modul ist ein Objekt als eine Instanz eines Datentyps zu verstehen. In unserem Fall ist die übergebene Datensequenz ein String Objekt und nicht zu verwechseln mit dem String Modul.
So wie der Punkt bei einer Modulfunktion den Funktionsnamen innerhalb Moduls klassifiziert, weist der `:` Lua an, die Funktion rechts vom Doppelpunkt des zugehörigen Objekts bzw. Datentyps links davon aufzurufen. In unserem Fall ist der Datentype von `data` ein String und Lua ruft die Funktion `string.byte( data )` auf, wobei `data` automatisch übergeben wird. Beide folgenden Aufrufe sind somit identisch:

```
data:byte( i )
string.byte( data, i )
```

Wenn Sie mit Objekt orientierten Sprachen wie C++ oder JavaScript vertraut sind, können Sie den Doppelpunkt einfach wie den Punkt in diesen Programmiersprachen ansehen.
Zeile 6 letztendlich liefert die untersten 8 Bit der Prüfsumme indem es den Module 256 Wert von `chksum` zurück gibt.

Normalerweise wird die Prüfsumme am Ende der zu übertragenen Datensequenz angehängt. Wenn Sie die Prüfsumme eines empfangenen Telegramms verifizieren wollen, müssen Sie die Prüfsumme aller Bytes mit Ausnahme der angehängten Prüfsumme bilden. D.h. Sie müssen einen bestimmten Ausschnitt des Telegramm Strings an die Checksum Funktion übergeben.

Mit `string.sub` enthält Lua einen mächtige Funktion um beliebige Zeichenfolgen aus einem String zu extrahieren. Start und Ende werden als Indexe übergeben wobei negative Indexe vom Stringende zählen. Dies macht es besonders einfach z.B. die letzten beiden Zeichen einer Zeichenkette zu ermitteln ohne die Stringlänge berücksichtigen zu müssen. Die folgenden Beispiele zeigen die Idee hinter diesem Konzept:

```
1  seq = "Hello world!"
2  print( string.sub( seq, 7 ) )     --> world!
3  print( string.sub( seq, 1, 5 ) ) --> Hello
4  print( string.sub( seq, -6 ) )    --> world!
5  print( seq:sub( -6, -2 ) )        --> world
```

Die String Methode (oder Funktion) `sub` wird mit drei Parametern aufgerufen:

```
string.sub( STRING, FROM, TO )
```

wobei TO optional ist und per Voreinstellung den Index des letzten Zeichens enthält.
Zeile 1 liefert den Teilstring von Position 7 (dem 'w') bis zum Stringende (default). Erinnern Sie sich, das Lua Indexe mit 1 starten.
Zeile 2 resultiert in den ersten 5 Zeichen des Strings (1 bis 5).

In Zeile 3 wird der Anfang des Teilstrings von hinten indiziert. -1 ist gleichbedeutend mit dem letzten Zeichen, -6 ist 'w' vom Ende gezählt. Auch hier ist der zweite Index per Voreinstellung das Stringende.
Zeile 4 verwendet einen Objekt orientierten Aufruf und begrenzt das Resultat mit dem zweiten Index -2 auf das vorletzte Zeichen.

18.1.4 Datenstrukturen in Lua

Feldbus Protokoll Spezifikationen enthalten oft nummerische Konstanten um Kommandos, Fehler Codes oder Statusmeldungen als Zahlenkombination zu übertragen. Dabei wird einem bestimmten Wert (Schlüssel) eine eindeutige Bedeutung bzw. Beschreibung (Wert) zugewiesen. Z.B. einem Fehlercode eine Fehlerbeschreibung. Oder einem Digitaleingang ein Wert OFF oder ON. In der Informatik spricht hier auch von key/value Paaren.
Um den zugeordneten Text (Wert) einer Zahl (Schlüssel) zu ermitteln kann man eine Lua Funktion wie folgt verwenden:

```
1 function GetDigitStateText( state )
2   if state == 0 then
3     return "OFF"
4   elseif state == 1 then
5     return "ON"
6   end
7   return ""
8 end
```

Dies mag für eine geringe Anzahl von Varianten noch funktionieren. Bei einer großen Anzahl von `if...elseif` Abfragen wird der Code aber sehr schnell unleserlich und Fehler anfällig.
Die meisten Programmiersprachen bieten zur Speicherung von Schlüssel/Wert Paaren sogenannte Assoziative Arrays. Dabei dient der Schlüssel als eindeutiger Index für den zugewiesenen Wert. Der Schlüssel kann eine einfache Zahl sein (ein Fehlercode) aber auch ein beliebiger Sting. Bei letzterem wird aus dem String eine eindeutige Zahl, der sogenannte Hash Wert gebildet, der dann als Index fungiert.

Lua bietet mit den sogenannten `table` eine eigene und sehr mächtige Implementierung eines assoziativen Arrays. `table` sind zudem der einzige und wichtigste Datenstruktur Mechanismus in Lua. Sie können eine `table` verwenden um gewöhnliche Arrays nachzubilden (auch solche mit einer Indizierung ab 0), sowie Stacks, Queues, Funktionen und Symboltabellen und vieles mehr. Selbst Lua Module sind als `table` organisiert.
Wenn wir die Funktion `string.dump` aufrufen bedeutet es nichts anderes als das Lua die Funktion mit dem Index (Hash Wert) `dump` in der string Modul Tabelle ausführt.
Weitere Informationen zu `table` finden Sie auf Seite 243. Wir wollen uns hier zunächst auf die grundsätzlichen Funktionen konzentrieren und betrachten Tabellen einfach als eine Liste von Schlüssel/Werte Paare.

Sie können eine einfache Tabelle erzeugen per:

```
1 days = { "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun" }
2 for i=1,#days do
3   print( days[ i ] )
```

```
4 end
```

Falls kein Index 'Schlüssel' angegeben wird verwendet Lua eine normale Indizierung beginnend mit dem Wert 1. Im oberen Beispiel hat der erste Tabelleneintrag den Index 1, der letzte den Index 7.
In Zeile 2...4 wird über alle Tabelleneinträge iteriert (der Längen Operator `#` liefert die Anzahl der Einträge) und die abgekürzten Wochentage ausgegeben.

Wir können die Indexe (Schlüsselwerte) auch ändern um eine Indizierung von 0 zu ermöglichen.

```
1  days = {
2    [ 0 ] = "Mon",
3    [ 1 ] = "Tue",
4    [ 2 ] = "Wed",
5    [ 3 ] = "Thu",
6    [ 4 ] = "Fri",
7    [ 5 ] = "Sat",
8    [ 6 ] = "Sun"
9  }
10 for i=0,#days do
11   print( days[ i ] )
12 end
```

Dabei weisen wir explizit jedem Index (Schlüssel) 0...6 den entsprechenden Wochentag als Wert zu. Die `for` Schleife in Zeile 10 beginnt nun mit 0, der Längen Operator `#` liefert 6 (den letzten gültigen Index in der Tabelle).
Beachten Sie! Das Ergebnis des Längen Operators `#` kann abweichen, wenn die Tabelle keine kontinuierlichen Indexe besitzt. Wir werden dies im Abschnitt Tabellen auf Seite 243 im Detail erörtern.

Zurück zu unseren Beispiel der Digitaleingänge. Statt einer Funktion mit vielen `if...elseif` erzeugen wir eine Tabelle und 'assoziieren' jeden Digitalwert mit einem dazugehörigen Text (String).

```
IOState = {
  [ 0 ] = "OFF",
  [ 1 ] = "ON"
}
```

Unsere `GetDigitStateText` Funktion vereinfacht sich damit zu:

```
1  function GetDigitStateText( state )
2    local tbl = {
3      [ 0 ] = "OFF",
4      [ 1 ] = "ON"
5    }
6    return tbl[ state ]
7  end
```

Wenn Sie die Funktion mit einem ungültigen Wert aufrufen, z.B. 2, wird der Wert **nil** zurückgegeben, was unter Lua gleichbedeutend ist mit 'dies ist ein ungültiger Wert'.
Die `print` Funktion ignoriert geflissentlich nil Argumente. Aber der Lua Interpreter wird einen Fehler generieren sobald Sie versuchen einen **nil** Wert weiter zu verarbeiten, z.B. in einer mathematischen Funktion[4]:

[4] Lua fasst alle mathematischen Funktionen in dem math Modul zusammen.

```
print( nil ) --> nothing
print( math.sin( nil ) ) --> bad argument #1 to 'sin'
                         --> (number expected, got nil)
```

Die Rückgabe eines nil Wertes wenn ein String erwartet wird, ist keine sonderlich gute Idee. In den meisten Fällen wird das Resultat in einer anderen Funktion weiter verarbeitet oder ist Teil einer String Operation. Im folgenden Skript stoppt der Lua Interpreter die Ausführung mit dem Fehler: *'attempt to concatenate a nil value'* weil Sie keinen nil Wert an einen String anfügen können.

```
print( "IO1="..GetDigitStateText( 2 ) )
```

Dieses Manko ist allerdings leicht zu korrigieren. Lua behandelt **nil** Werte wie ein logisches **false**. Damit können wir unsere Funktion gegen ungültige Parameter absichern und immer einen gültigen String zurückgeben:

```
1 function GetDigitStateText( state )
2   local tbl = {
3     [ 0 ] = "OFF",
4     [ 1 ] = "ON"
5   }
6   return tbl[ state ] or "INVALID STATE"
7 end
```

Wir erweitern Zeile 6 mit einer ODER (**or**) Verknüpfung und geben entweder einen gültigen Wert aus der Tabelle ODER (falls `table[ state ]` **nil** ist) 'INVALID STATE' zurück. Sie können natürlich auch einen beliebigen anderen, z.B. leeren Rückgabe String wählen.

18.1.5 Wiederverwendung von Code mit Lua Modulen

Sie können eigene Funktionen natürlich immer per Copy & Paste dort einfügen, wo Sie diese gerade brauchen. Es macht allerdings mehr Sinn Funktionen mit ähnlichem Zweck in einem separaten Modul zusammenzufassen und in einer eigenen Modul Datei als Bibliothek zu speichern.
Ein checksum Modul hierbei ist ein gutes Beispiel. Der in der Analyser Software verwendete Lua Interpreter kommt mit einem eigenen checksum Modul welches die gängigsten Algorithmen enthält. Gleichwohl gibt es eine Vielzahl weiterer checksum Varianten, wie Module 256 in unserem vorherigen Beispiel, die Sie evtl. in einem eigenen checksum Modul speichern möchten

Der Lua Skript Editor hilft Ihnen bei der Erstellung eines neuen Lua Moduls mit einem entsprechenden Code Gerüst. Klicken Sie dazu im Editor das 'Neue Skriptdatei' Icon in der Werkzeugleiste oder drücken Sie STRG + N.
In dem Skript Ersteller Dialog wählen Sie Skript für 'Module' und ersetzen den vorgegebenen Dateiname mit `mychecksums.msbtml`. Klicken Sie 'OK' und der Editor öffnet einen neuen Buffer (Tab) mit bereits eingefügtem Code Gerüst.

Ersetzen Sie in einem zweiten Schritt alle Modul Platzhalter `ModuleName` mit `MyChecksums`.
Anschließend fügen Sie noch unsere checksum Funktion hinzu und ersetzen den Funktionsnamen durch `MyChecksums.Mod256Sum`. Die Modul Datei sollte jetzt wie folgt aussehen:

```
1  local MyChecksums = {}
2
3  function MyChecksums.Mod256Sum( data )
4    local chksum = 0
5    for i=1,#data do
6      chksum = chksum + data:byte( i )
7    end
8    return chksum % 256
9  end
10
11 function MyChecksums.version()
12    return "1.0.0"
13 end
14
15 return MyChecksums
```

Module sind in Lua nichts anderes als Tabellen (eine weitere Anwendung für dieses extrem leistungsfähige Konzept).
In Zeile 1 erstellen wir eine leere Tabelle wobei der Name unerheblich ist. Anschließend fügen wir unsere Checksum Funktion in der Tabelle hinzu, indem wir dem Funktionsnamen getrennt durch einen Punkt dem Tabellennamen voranstellen. Lua verwendet den Funktionsnamen als Index für den späteren Modul Zugriff.
In unserem Beispiel sind es zwei Funktionen: `Mod256Sum` und `Version`.
Zeile 15 liefert die komplette Tabelle mit all Ihren Funktionen. Dabei wird die Tabelle, nicht aber ihr Name zurückgegeben. Der Name der Tabelle ist deshalb nicht von belang, solange er im Modul selbst einheitlich verwendet wird.

Speichern Sie die Modul Datei und wechseln Sie zurück in den SKETCH Buffer. Geben Sie hier folgende Zeilen ein:

```
1  checksums = require "MyChecksums"
2  print( checksums.Mod256Sum( "Hello world" ) )
```

Das Lua Schlüsselwort **require** in Zeile 1 lädt unsere neue Modul Datei und weist dieses (die Modul Tabelle) der Variable `checksums` zu.
Das es sich in der Tat um eine Tabelle handelt können Sie mit folgendem Einzeiler selbst überprüfen:

```
print( type( checksums ) ) --> table
```

(Die Lua `type` Funktion gibt den Typ des übergebenen Arguments als String zurück.

Sie können jederzeit weitere Prüfsummen Funktionen zu Ihrem checksum Modul hinzufügen. Sie müssen lediglich darauf achten, dem Funktionsnamen immer den Namen der lokalen Modul Tabelle getrennt durch einen Punkt voran zu stellen. Nur dadurch wird die Funktion Teil der vom Modul bereitgestellten Tabelle.

```
1  function MODULENAME.FUNCTIONNAME(...)
2    -- function code ...
3  end
```

Funktionen ohne führenden Modul Namen verhalten sich wie lokale Funktion. Sie sind außerhalb des Moduls nicht sichtbar und können nur innerhalb des Moduls aufgerufen werden. Weitere Modul Informationen finden Sie auf Seite 256.

Test Einschränkungen

Der im Editor eingebaute Lua Interpreter kann nur Lua Module ausführen, die nicht von einer besonderen View Umgebung abhängen. So ist z.B. das Ausführen des vom Protokollmonitor bereit gestellten `box` Moduls nicht möglich!

18.2 Die Lua Sprache

Jede Programmiersprache besteht aus einem Satz von Zutaten wie Operatoren, Schlüsselwörter (keywords), Funktionen und einigen Regeln, wie diese anzuwenden sind. Alles zusammen nennt man die Sprachsyntax. Sie definiert, wie korrekte (aber nicht unbedingt fehlerlose oder sinnvolle) Programme zu schreiben sind.
In diesem Kapitel geben wir einen Überblick über die Skriptsprache Lua, die von ihr unterstützten Operatoren, Schlüsselwörter und einige hilfreichen Module die wir in der Analyser Software integriert haben.

18.2.1 Groß-/Kleinschreibung in Lua

Lua ist eine Sprache die zwischen Groß- und Kleinschreibung unterscheidet. So ist **while** ein reserviertes Wort (ein sogenanntes Schlüsselwort), während `WHILE` oder `While` zwei unterschiedliche Bezeichnungen einer Variable oder Funktion sein können. Dies ist bei modernen Sprachen allgemein üblich und sollte Sie deshalb nicht sonderlich verwirren.

18.2.2 Leerzeichen und Zeilenende

Lua ignoriert jegliche Leerzeichen (einschließlich Tabulatorzeichen) sofern sie nicht Teil einer Zeichenkette (siehe 18.2.4.6) sind. Lua schreibt auch keine besondere Einrückung vor (wie z.B. Python). Sie können Ihren Programmcode ganz nach belieben formatieren (oder zur besseren Lesbarkeit strukturieren).
Lua verwendet kein spezielles Zeilenende, Zeilenumbrüche spielen deshalb in der Lua Syntax keine Rolle. Der Lua Interpreter erkennt das Ende einer Anweisung unabhängig von einem Zeilenende. D.h. eine Zeile kann mehrere Anweisungen enthalten bzw. eine Anweisung über mehrere Zeilen verteilt sein.
Wenn Sie mehrere Anweisungen in einer Zeile optisch trennen wollen, können Sie dazu das Semikolon ';' verwenden. Es spielt aber - wie gesagt - keine Rolle.

```
1  x = 1 y = 2  --> nicht sehr leserlich aber ok
2  x = 1; y = 2 --> besser
3  z = x
4  +
5  Y            --> z = 3
```

18.2.3 Kommentare

Kommentare werden in Lua mit einem doppelten Bindestrich `--` eingeleitet und reichen bis zum Ende der Zeile. Verwenden Sie Kommentare um Ihre Skripte zu dokumentieren oder um einzelne Zeilen beim Erstellen und Testen vorübergehend auszukommentieren.
Mehr noch: Lua unterstützt zusätzlich blockweises auskommentieren wobei ein

Kommentarblock mit `--[[` beginnt und mit `--]]` endet. Dies machte es besonders einfach, mehrere Zeilen ein/auszukommentieren, wie wir im folgenden zeigen:

```
1  x = 1
2  --[[
3  x = 10
4  --]]
5  print( x ) --> 1
```

Um die Auskommentierung des kompletten Blockes rückgängig zu machen reicht ein einfacher Bindestrich am Anfang des Kommentarblockes. Blockstart- und Blockende-Bezeichnungen werden damit als einfache Zeilenkommentare gesehen und die Anweisungen dazwischen als regulärer Code ausgeführt.

```
1  x = 1
2  ---[[
3  x = 10
4  --]]
5  print( x ) --> 10
```

18.2.4 Datentypen

Lua ist eine Programmiersprache mit dynamischer Typbindung. D.h. Sie müssen den Typ einer Variable oder Wertes nicht explizit angeben, Lua 'verwaltet' den korrekten Typ automatisch für Sie.
Lua unterstützt acht verschiedene Grundtypen, wir beschränken uns allerdings auf die folgenden sechs:

- Zahlen
- Boolesche Werte
- Zeichenketten
- nil
- Tabellen
- Funktionen

Zu Programmbeginn feste 'Werte' werden dabei als 'Konstanten' bezeichnet. Eine Konstante ist nicht Ergebnis einer Anweisung sondern fest im Programmtext kodiert. Konstante können Zahlen sein (ganze Zahlen und Fließkommazahlen, wobei Lua intern nicht zwischen beiden unterscheidet), Zeichenketten oder die boolean Werte **true** and **false**.

18.2.4.1 Zahlen

Lua vereinfacht die Anwendung von so unterschiedlichen Zahlentypen wie ganze Zahlen, Fließkommazahlen, Fließkommazahlen mit doppelter Genauigkeit indem es intern nur einen Typ verwendet. Zahlenwerte in Lua sind generell vom Typ Fließkommazahlen mit doppelter Genauigkeit und werden intern automatisch umgewandelt.
Zumindest war das so bevor Lua die bitweise Operatoren einführte, was intern einen zweiten - Ganzzahl oder Integer - Typ erforderlich machte.
Dies betrifft allerdings nur sehr spezielle Fälle und wir werden diese in einem eigenen Abschnitt 18.2.4.2 im Detail diskutieren. Zunächst betrachten wir alle Lua Zahlen weiterhin ausschließlich als vom Typ Real bzw. Fliesskomma Zahl.

```
1  print( 1 )     --> 1
2  print( -12 )   --> -12
3  print( 100000000000 )  --> 100000000000
```

Beachten Sie das Zahlen niemals zu Ganzzahlen gerundet werden. So ist:

```
1  print( 10 / 3 )  --> 3.3333333333333
```

Wenn Sie nur den Integer Teil einer Zahl benötigen, können Sie dazu die `modf` des `math` Moduls verwenden. Diese Funktion liefert ähnlich unserem Beispiel am Anfang beides zurück: Den Vorkomma- und Nachkommateil einer gegeben Fliesskommazahl.

```
print( math.modf( 10 / 3 ) ) --> 3      0.33333333333333
```

Wie können diese Funktion nutzen um unsere eigene Fliesskomma zu Integer Konvertierung zu schreiben.

```
function integer( number )
    int, dec = math.modf( number )
    return int
end
print( integer( 10 / 3 ) ) --> 3
```

Wie bereits erläutert: Lua wandelt einen Typ automatisch in einen anderen, wenn nötig. In den Code obigen Beispielen zumeist von einem Fliesskommawert in einen String. Dies funktioniert aber auch in der gegensätzlichen Richtung. Ein Beispiel:

```
s = "1.25"
print( 2.0 * s + 2.5 ) --> 5.0
```

Hier wird der String `"1.25"` zuerst in einen Fliesskommawert konvertiert, anschließend mit 2.0 multipliziert und zu 2.5 addiert bevor das Resultat (jetzt eine Fliesskommazahl) in der `print` Funktion wieder zurück in einen Lua String gewandelt wird.

18.2.4.2 Integer versus Fliesskomma

Wir haben es bereits erwähnt: Bitweise Operatoren machen wenig Sinn mit Fliesskommazahlen. Deshalb versucht Lua alle Bitweise Operanden zunächst in Integer Typen umzuwandeln. Das Resultat einer bitweisen Operation ist dabei ebenfalls immer ein Integer Wert.
Um das ganze stringent zu machen führt Lua zu diesem Zweck intern eine zweite Zahlen Repräsentation ein. Neben dem bereits existierenden Fliesskomma Typ existiert nun ein zweiter Integer oder Ganzzahl Typ. Allerdings mit dem Preis, dass mache bislang automatischen Typumwandlungen nun nicht länger funktionieren.
Im folgenden einige Beispiele als Anhaltspunkt, falls Ihr bisheriger Lua Code nach einem Analyser Programm Update nicht mehr funktionieren sollte:

```
-- Lua 5.2 using the bit32 library
print( bit.rshift( 1.5, 1 ) ) --> 1
-- Lua 5.3 with integrated bitwise operators
print( 1.5 >> 1 ) --> error, number has no integer representation
-- Lua 5.2
print( string.format( "%d\n", 10 / 3 ) ) --> 3
print( string.format( "%04x", 1.25) ) --> 0001
-- Lua 5.3
print( string.format( "%d\n", 10 / 3 ) ) --> bad argument #2 to 'format
    ' (number has no integer representation)
```

```
print( string.format( "%04x", 1.25 ) ) --> bad argument #2 to 'format'
    (number has no integer representation)
```

In solchen Situation wäre es natürlich sehr schön, man könnte den Zahlentyp oder das Ergebnis einer Operation ermitteln. Zum Glück bietet Lua dazu gleich zwei Möglichkeiten. Die erste (`type()`) liefert den allgemeinen Typ einer Variablen, d.h. ob es sich dabei um einen String, eine Funktion, Tabelle oder eben um eine Zahl handelt.

```
x = "1.25"
print( type( x ) ) --> string
print( type( x + 1.0 ) ) --> number
print( type( type ) ) --> function
```

Aktuell sind wir natürlich mehr am eigentlichen Typ eines Zahlenwertes interessiert. Handelt sich bei einer Zahl um einen Integer oder Fliesskommawert. Die Lösung hierfür ist die `math.type()` Funktion.

```
print( math.type( 1.25 ) ) --> float
print( math.type( 1 ) ) --> integer
```

Beachten Sie, das `math.type()` nil zurück gibt, wenn es sich bei dem übergebenen Wert um keine Zahl handelt!

```
print( math.type( "1.2"  ) ) --> nil
```

So weit so gut - ab was können Sie tun, wenn eine Funktion oder Operation definitiv einen Integer Wert erwartet, Ihr Code aber eine Fliesskommazahl liefert? Siehe das `string.format(...)` Beispiel zuvor. In so einem Fall müssen Sie den Typ explizit selbst umwandeln.
An dieser Stelle ist es gut zu wissen, dass Lua eine automatische Typumwandlung von Fliesskomma zu Integer erlaubt, sofern der Fliesskommawert KEINE Nachkommastellen enthält. Also Zahlen wie `1.0` oder `2E03`.

```
print( string.format( "%04X", 1.0  ) ) --> 0001
print( string.format( "%04X", 1E02  ) ) --> 0064
```

Aber:

```
x = 1E-02
print( string.format( "%04X", x  ) ) --> bad argument #2 to 'format' (
    number has no integer representation)
```

Sie ahnen es vielleicht schon. Die Lösung besteht einfach darin, die Nachkommastellen zu entfernen. Das können Sie entweder mit der math.floor() Funktion oder noch einfacher mit dem Lua `//` Operator.

```
x = 100.5
print( string.format( "%04X", x // 1 ) ) --> 100
```

18.2.4.3 Hexadezimale Konstanten

Auch wenn Lua intern mit Fließkommazahlen arbeitet wollen Sie doch manchmal Zahlenwerte in anderen Formaten vorgeben, z.B. als Hexwert.

```
1  print( 0x1234 )     --> 4660
```

Lua erlaubt die Eingabe beliebiger hexadezimaler Zahlen mit einem führenden `0x`.

18.2.4.4 Fließkommakonstanten

Wie die meisten Programmiersprachen unterstützt Lua bei der Zahleneingabe die traditionelle und wissenschaftliche Notation. Fließkommakonstanten können daher wahlweise mit Dezimalpunkt und/oder Dezimalexponent angegeben werden.

```
1  print( -0.05 )    --> -0.05
2  print( 10E-2 )    --> 0.1
3  print( 1.25E+6 )  --> 1250000
```

18.2.4.5 Boolesche Konstanten

Ein boolescher Datenwert entspricht den klassischen logischen Zuständen und ist daher entweder **true** oder **false**. Ist ein boolescher Wert nicht true muss er false sein und umgekehrt. Boolesche Werte werden i.a. zur Repräsentation der Ergebnisses logischer oder konditionaller Operationen verwendet.

```
1  print( 2 > 1 ) --> true
2  x = 2 < 4
3  print( x ) --> false
```

18.2.4.6 Zeichenketten

Zeichenketten haben in Lua die übliche Bedeutung; sie entsprechen einer Reihe von einzelnen Zeichen. Im Gegensatz zu anderen Sprachen unterstützt Lua jedoch 8 Bit Zeichen in Zeichenketten. Dies hat einen entscheidenden Vorteil: Zeichenketten (strings) können in Lua 'jedes' beliebige Zeichen enthalten, d.h. auch ein Null Byte (in C das Stringende). Mit anderen Worten: Sie können beliebige d.h. auch binäre Daten in Lua Zeichenketten speichern ohne besondere Einschränkungen/Ausnahmen befürchten zu müssen.
Zeichenketten werden durch einfache Hochkommas, doppelte Hochkommas oder doppelte eckicke Klammern definiert.

```
1  print( "It's your code" ) --> It's your code
2  print( 'He says:"Hi"' )   --> He says: "Hi"
3  print( [[Hello\nWorld]] ) --> Hello
4                                World
```

Warum so verschiedene Arten der String-Definition?
Die Verwendung der speziellen Zeichen Hochkomma und doppeltes Hochkomma innerhalb einer Zeichenkette wird dadurch vereinfacht. Wählen Sie einfach das Zeichen zur Spezifikation Ihrer Zeichenkette, welches Sie in dieser nicht verwenden wollen.
Zeichenketten innerhalb zweier eckigen Klammern erlauben zudem die Unterdrückung von sogenannten Escape Sequenzen.

18.2.4.7 Zeichenketten mit Escape Sequenzen

Lua definiert folgende Escape Sequenzen innerhalb von Zeichenketten:

Escape Sequenze	Beschreibung
\a	Glocke
\b	Rücktaste
\f	Formfeed
\n	Zeilenumbruch (Linefeed)

\r	Carriage Return
\t	horizontaler Tabulator
\v	vertikaler Tabulator
\\	Backslash
\"	doppeltes Hochkomma
\'	einfaches Hochkomma
\ddd	Zeichen mit dem numerischen Dezimalwert ddd

Die folgenden Beispiele zeigen ihre Verwendung:

```
1  print( 'It\'s your code' )       --> It's your code
2  print( "He says:\"Hi\"" )        --> He says: "Hi"
3  print( "Tab1\tTab2" )            --> Tab1    Tab2
4  print( "Two backslashes \\\\" ) --> Two \\
5  print( "Hello\nworld'' )         --> Hello
6                                       world
7  print( [[Hello\nworld]] )        --> Hello\nworld
```

Mittels \ddd können Sie jedes beliebige Byte als numerischen Dezimalwert in einem String einfügen. Für eine Zeichenkette bestehend aus den Zeichen 0...3 ist es: "\000\001\002\003".

18.2.4.8 nil

nil ist ein spezieller Typ in Lua und bezeichnet einen undefinierten Wert. Jede Variable besitzt einen **nil** Wert solange ihr kein Wert zugewiesen wird.

```
1  print( x ) --> nil
```

Aber mehr noch: Lua verwendet **nil** wenn eine Variable nicht weiter verwendet wird. Indem Sie einer Variable den Wert **nil** zuweisen wird diese von Lua automatisch entfernt bzw. gelöscht.

18.2.5 Tabellen

Einer der mächtigsten Datentypen in Lua sind tables (Tabellen). Es handelt sich dabei um assoziative Arrays, die - anders als echte Arrays - einen beliebigen, auch nicht numerischen Schlüssel (key) zur Adressierung der einzelnen Werte (value) verwenden. Mehr noch: Da Funktionen in Lua ebenfalls einen normalen Datentyp darstellen, können Tabellen zur Implementierung eines Objekt orientierten Verhaltens verwendet werden. Dies ist allerdings nicht Teil dieser Anleitung.

Die Größe der Tabellen ist nicht festgelegt oder beschränkt und wächst mit dem Hinzufügen neuer Einträge. Wenn Sie eine Tabelle nicht länger benötigen können Sie sie einfach per Zuweisung von **nil** aus dem Speicher entfernen.
Starten wir mit einigen Beispielen die etwas mehr Licht in das Verständnis von Tabellen bringen sollten. Als erstes definieren wir eine einfache Liste mit drei Früchten:

```
1  fruits = { "apple", "banana", "orange" }
2  print( fruits[2] ) --> banana
```

Die Anweisung in Zeile 1 initialisiert eine Tabelle mit drei Einträgen. Das erste Feld `fruits[1]` enthält den Text (oder Zeichensequenz) "apple", das zweite `fruits[2]` "banana" und das dritte "orange". Beachten Sie, das in Lua Indexe generell mit 1 starten und nicht mit 0 (wie z.B. in C).
In diesem Beispiel fungiert die Tabelle als eine einfache Liste. Sie können jederzeit neue Einträge hinzufügen mit:

```
fruits[ #t+1 ] = "pear"
```

Die Anweisung in Zeile 3 verwendet Lua's internen Längen-Operator `#` (siehe auch Abschnitt 18.2.9.6) um den letzten Eintrag in der Liste zu ermitteln. Alternativ können Sie auch die Funktion `table.insert(table,value)` aus dem `table` Modul benutzen.

```
1  t = { 1, 2, 3, 4, 5 }
2  table.insert( t, 6 ) --> 1, 2, 3, 4, 5, 6
3  print( "Count: ", #t )  --> Count: 6
```

Um einen Tabelleneintrag an beliebiger Position einzufügen ohne dazu bereits vorhandene Einträge kompliziert verschieben zu müssen, können Sie die gleiche Funktion mit einem zusätzlichen Position Parameter aufrufen.
`table.insert(`*table*`,`*position*`,`*value*`)`.

```
1  t = { 1, 2, 3, 4, 5 }
2  table.insert( t, 4, 44 ) --> 1, 2, 3, 44, 4, 5
3  print( "Count: ", #t )      --> Count: 6
```

Ein Element an einer bestimmten Position entfernen Sie mit:
`table.remove(`*table*`,`*position*`)`.

```
1  t = { 1, 2, 3, 4, 5 }
2  table.remove( t, 4 ) --> 1, 2, 3, 5
3  print( "Count: ", #t )  --> Count: 4
```

Sie können jederzeit ein bestimmtes Element in der Tabelle oder Liste durch einen neuen Wert ersetzen. Jedes Tabelleneintrag ist durch den Index-Operator `[]` referenzierbar. Um den zweiten Eintrag mit 200 zu überschreiben reicht z.B. ein einfaches: `t[2]=200`.

Entsprechend einfach ist die Abfrage beliebiger Einträge, in unserem Beispiel `v=t[2]` enthält v den Wert des zweiten Feldes. Ist das Feld nicht vorhanden wird **nil** zurück gegeben. Konsequenter Weise führt jede Zuweisung an ein nicht existierendes Feld zur Anlegung desselben.

Die bisherigen Tabellen Beispiele verwendeten Zahlenindexe und übergaben lediglich die Werte. Dies entspricht der Standardvorgabe von Lua. Wie oben erwähnt können Sie allerdings auch beliebige andere Datentypen als Index (key) und Werte verwenden. Stellen Sie sich einen Datentyp vor, der einen Punkt in einem Koordinatensystem repräsentiert:

```
1  point = { x = 5, y = 10 }
2  print( point.x, point.y ) --> 5    10
```

Hier wird eine Tabelle zur Speicherung der beiden Koordinaten x und y benutzt. Diese Art Beziehung zwischen Schlüssel (x oder y) und Wert (der eignetliche Koordinatenwert) in einem Array wird auch Dictionary genannt.

18.2.5.1 Diskontinuierliche Tabellen mit Löchern

Betrachten Sie dazu folgendes Beispiel:

```
1  errorCodes = {
2    [ 1 ] = "General error",
3    [ 2 ] = "Invalid command",
4    [ 3 ] = "Invalid function",
5    [ 7 ] = "Wrong checksum",
6    [ 11 ] = "Timeout"
7  }
8  print( #errorCodes ) --> 3 !!!
```

Was passiert hier?
Lua initialisiert alle nicht verwendeten Tabellen Indexe mit **nil**. Bei nicht kontinuierlichen Tabellen wertet Lua den ersten ungültigen Index (nil) als Tabellenende. In unserem Beispiel ist der Index 4 nil weil kein entsprechender Tabelleneintrag vorhanden ist. Dies gilt auch für die Index 5...10. `#` stoppt deshalb bei Index 4 und das Ergebnis ist 3.

Eine ähnliche Situation liegt vor, wenn Sie eine Tabelle als 'Dictionary' verwenden und der Schlüssel (Index) aus Strings anstelle von Zahlen besteht.

```
1  translations = {
2    [ "apple" ] = "Apfel",
3    [ "banana" ] = "Banane",
4    [ "orange" ] = "Orange",
5    [ "pear" ] = "Birne"
6  }
7  print( #translations ) --> 0 !!!
```

Lua berechnet in diesem Fall für jeden Schlüssel einen speziellen Hash Wert. Dieser dient als Positionsindex des zugehörige Wertes in der Tabelle, d.h an welcher Position der Wert abgelegt ist[5]. Ein perfekter Hash Algorithmus erzeugt für jeden beliebigen String eine einzigartige Nummer. Die Hash Zahl kann dabei jeden Wert der möglichen Strings annehmen. Im Beispiel oben ist keiner der Hash Werte 1. Der allererste Index Eintrag in der `translations` Tabelle ist daher nil und der Längen Operator stoppt bereits beim ersten Eintrag und liefert uns 0.
Wenn Sie wirklich die Anzahl aller Einträge in einer nicht kontinuierlichen Tabelle benötigen, können Sie den Operator `#` nicht verwenden. Stattdessen müssen Sie über alle Einträge iterieren. Wir werden dies im nächsten Abschnitt erläutern. Merken Sie sich an dieser Stelle nur:

Verwenden Sie den # Operator bei Tabellen mit Vorsicht

Der `#` Operator arbeitet nur mit ein-dimensionalen Arrays (Tabellen ohne speziellen Index). Vermeiden Sie den Längen Operator bei Tabellen die evtl. Löcher enthalten!

Zum Glück ist dies nicht weiter tragisch weil Sie vermutlich nur sehr selten die Anzahl der Einträge in einer nicht kontinuierlichen Tabelle bzw. 'Dictionary' benötigen werden. Stattdessen greifen Sie auf die einzelnen Einträge über ihre Schlüssel Indexe zu. Die Größe der Tabelle spielt dabei keine Rolle und Lua kümmert sich selbst darum.

[5] Dies ist eine vereinfachte Darstellung, das Prinzip ist aber das gleiche!

18.2.5.2 Tabellen Iteration

Sie können über den Inhalt einer numerischen (ein-dimensionalen und kontinuierlichen) Tabelle iterieren wie wir es bereits bei Strings vorgemacht haben.

```
1  fruits = {"apple","banana","orange","pear"}
2  for i=1,#fruits do
3    print( fruits[i] ) --> apple banana orange pear
4  end
```

Dies funktioniert auch wenn Sie eine Index Zählung von 0 verwenden. Und sogar wenn Sie die Einträge unsortiert vornehmen.

```
1  fruits = {
2      [1] = "banana",
3      [2] = "orange",
4      [3] = "pear",
5      [0] = "apple",
6  }
7  for i=0,#fruits do
8    print( fruits[i] ) --> apple banana orange pear
9  end
```

Es ist lediglich wichtig, das die Indexe stetig und ohne Lücken sind. Das folgende Beispiel funktioniert deshalb nicht!

```
1  fruits = {
2      [2] = "apple",
3      [3] = "banana",
4      [4] = "orange",
5      [5] = "pear",
6  }
7  for i=0,#fruits do
8    print( fruits[i] ) --> !
9  end
```

Der erste Index in einer ein-dimensionalen (numerischen) Tabelle ist standardmäßig immer 1 solange Sie nicht explizit mit 0 beginnen (wie zuvor gezeigt). Hier startet der Index allerdings mit 2 und Index 1 ist somit **nil**. Der Längen Operator `#` liefert deshalb die Länge 0 und `print` in Zeile 8 wird niemals aufgerufen.

Lua bietet zum Durchlaufen von Tabellen zwei Iteratoren `ipairs` und `pairs`. Beide werden mit einer Tabelle als Argument aufgerufen und kehren jedes Mal mit dem nächsten key/value (Schlüssel/Werte) Paar zurück (daher der Name). `ipairs` wird hauptsächlich bei numerischen Tabellen verwendet und stoppt automatisch beim ersten Index/Schlüssel der keiner Zahl oder nil entspricht. Mit `ipairs` können wir unser erstes Beispiel wie folgt umschreiben:

```
1  fruits = {"apple","banana","orange","pear"}
2  for i,v in ipairs(fruits) do
3    print( i, v )
4  end
```

Zeile 2 ruft den Iterator auf und speichert das Ergebnis in den Variablen i (Index oder Schlüssel) und v (Wert). Der Iterator stoppt sobald er auf einen nil Wert trifft oder das Tabellenende erreicht. Zeile 3 gibt beim Durchlauf jeweils das neue key/value Ergebnis aus:

```
1    apple
2    banana
```

```
3    orange
4    pear
```

Weitaus interessanter ist der zweite Iterator `pairs`. Er ist speziell zur Anwendung mit assoziativen Tabellen gedacht und damit besonders für Tabellen mit 'Löchern' geeignet. Er funktioniert aber auch bei einfachen Tabellen mit numerischem Index.

```
1  fruits = {
2      [2] = "apple",
3      [3] = "banana",
4      [4] = "orange",
5      [5] = "pear",
6  }
7  for i,v = pairs( fruits ) do
8    print( i, v )
9  end
```

Das Result ist:

```
3    banana
2    apple
4    orange
5    pear
```

Wahrscheinlich ist Ihnen bereits aufgefallen, das die Ausgabe in unsortierter Reihenfolge erfolgt. Auch wenn die Indexe von 2...5 eine sortierte Sequenz anzeigen werden Sie intern dennoch als Eingabe für den Hash Algorithmus verwendet. Und so wenig der resultierende Hash Wert dem ursprünglichen Index Key (Schlüssel) entspricht so ist es die interne Ordnung in der Tabelle.

18.2.5.3 Tabellen sortieren

Wie wir gesehen haben können assoziative Tabellen in Lua nicht nach ihren Schlüsseln bzw. Indexen sortiert werden. Aber Sie können eine zweite numerische (d.h. ein-dimensionalen) Tabelle erzeugen um die Schlüssel dort zu speichern und zu sortieren. Anschließend iterieren Sie über diese zweite sortierte Tabelle um auf die Schlüssel/Werte Paare der ersten zuzugreifen:

```
1  fruits = {
2      [2] = "apple",
3      [3] = "banana",
4      [4] = "orange",
5      [5] = "pear",
6  }
7  a = {}
8  for i,v in pairs( fruits ) do a[#a+1] = i end
9  table.sort(a)
10 for i,v in ipairs(a) do print( i, fruits[v] ) end
```

Das Resultat ist nun:

```
2    apple
3    banana
4    orange
5    pear
```

Erklärung: Zeile 7 erzeugt eine neue leere Tabelle `a` in der wir die Indexe bzw. Schlüssel (und nur diese) unserer `fruits` Tabelle speichern.
In Zeile 8 iterieren wir über `fruits` und fügen jeden Index (Schlüssel) am

Ende der Tabelle `a` an. Der Inhalt von `a` wird dadurch: `{3,2,4,5}` (siehe Ausgabe zuvor).
Tabelle `a` ist eine numerische ein-dimensionale Tabelle[6] und kann mit der Tabellen Funktion `table.sort` in Zeile 9 sortiert werden. Der Inhalt von `a` wird dadurch zu `{2,3,4,5}`.
In Zeile 10 durchlaufen wir die sortierten Indexe um die zugehörigen Werte aus `fruits` abzurufen und auszugeben.

Wir können diesen Code dazu benutzen unseren eigenen Sortier Iterator zu schreiben. Diesen können wir dann immer verwenden, wenn wir einen sortierten Tabellen Durchlauf benötigen. Der Iterator muss dazu lediglich bei jedem neuen Aufruf das nächste in der Sortierreihenfolge vorliegende Schlüssel/Werte Paar zurückgeben.

```
1  function sortedPairs( t )
2      local a = {}
3      for i in pairs(t ) do a[#a + 1] = i end
4      table.sort( a )
5      local i = 0
6      return function()
7          i = i + 1
8          return a[i], t[a[i]]
9      end
10 end
11
12 for i,v in sortedPairs( fruits ) do
13     print( i, v )
14 end
```

Der entscheidende Code liegt in den Zeilen 5...9.
`sortedPair` gibt selbst kein Schlüssel/Wert Paar zurück. Vielmehr delegiert es diese Aufgabe an eine anonyme Funktion[7] in Zeile 6 welche das nächste Paar als Resultat liefert.

Funktionen dieser Art werden auch Factory Funktionen genannt.
Wenn wir `sortedPair` das erste mal in der **for** Schleife in Zeile 12 aufrufen, sortiert diese die Indexe der übergebenen Tabelle `t` in einer neuen Tabelle `a`. Außerdem initialisiert sie einen lokalen index Zähler `i` mit 0. Der Zähler ist von außen nicht zugänglich da als **lokal** spezifiziert, wird von Lua aber im Speicher gehalten, da die anonyme Funktion auf diesen zugreift.

In Zeile 6 liefert `sortedPair` schließlich die eigentliche (anonyme) Iterator Funktion, die die sortierten Indexe in Tabelle `a` zur Rückgabe der Schlüssel/Werte Paare verwendet.
Mit anderen Worten: `sortedPair` 'fabriziert' eine neue Iterator Funktion inklusive der nötigen Variablen und Initialisierungen. Daher der Name 'Factory' Funktion.

In der **for** Schleife in Zeile 12 bewirkt daher jeder Aufruf von `sortedPair` letztendlich den Aufruf der anonymen Funktion in Zeile 6. Letztere 'merkt' sich

[6] Numerische Tabellen haben in Lua keinen 'Schlüssel' oder speziellen Index. Sie sind eine kontinuierliche Liste von Einträgen und bilden dadurch ein ein-dimensionales Array.
[7] sf Eine anonyme Funktion ist eine Funktion ohne Namen.

den zuletzt benutzten Index im Zähler `i` um bei erneutem Aufruf das nächste Schlüssel/Wert Paar liefern zu können.

18.2.6 Bezeichner

Ein Bezeichner (Identifier) ist in Computer Sprachen ein Name, der sich auf etwas anderes bezieht, z.B. eine Variable oder eine Funktion. Einige Bezeichner werden von Lua für interne Zwecke reserviert, die sogenannten Schlüsselworte (keywords). Wir kennen bereits die booleschen Werte **true** und **false**). Andere sind die fest eingebauten Funktionen wie z.B. `print`.
Die Regeln für zulässige Bezeichner oder Namen entsprechen denen anderer Sprachen. Ein Bezeichner kann in Lua aus einer beliebigen Folge aus Buchstaben, Zahlen und Unterstrichen bestehen, wobei der Name nicht mit einer Zahl beginnen darf[8].
Gültige Bezeichner oder Variablennamen sind:

```
x          y          ABC        t1         _nm
aVeryLongVariableName          the_last_result
```

Ungültige Bezeichner führen zu einem Fehler

```
1  print( 2n )      --> malformed number near '2a'
```

18.2.7 Schlüsselworte

Die folgenden Schlüsselworte sind reserviert und können nicht als Variablen- oder Funktionsnamen verwendet werden:

```
end        false      for        function   if
in         local      nil        not        or
repeat     return     then       true       until      while
```

Erinnern Sie sich? Lua unterscheidet zwischen Groß- und Kleinschreibung, daher ist **and** ein reserviertes Wort während `And` und `AND` einfach gültige (wenn auch wenig sinnvolle) Bezeichnungen für zwei unterschiedliche Variablen sind!

18.2.8 Variablen

Variablen sind vergleichbar mit Behältern die beliebige Daten enthalten können. Jeder Behälter hat einen eindeutigen Namen über den auf den Inhalt zugeriffen werden kann. In Lua können Variablen eine einzelne Zahl, einen Text mit Millionen von Zeichen oder eine Tabelle enthalten. Der Name des Behälters muß ein gültiger Bezeichner sein (siehe oben).
Variablen müssen nicht vor der ersten Anwendung deklariert werden. Sobald Lua eine neue Variable findet, wird diese automatisch angelegt.

18.2.8.1 Zuweisung

Anmerkung! Vor der ersten Zuweisung ist der Wertes einer Variable immer nil. Zuweisungen sind das zulässige Mittel um eine Variable (oder Tabelle) zu initialisieren oder deren Wert zu ändern.

[8]Die Analyser Software und Lua selbst verwenden Namen beginnend mit einem _ für interne Zwecke. Wir empfehlen deshalb generell bei Variablen ein _ als erstes Zeichen zu vermeiden.

```
1  if x == nil then
2      x = 1
3  end
4  print( x )        --> 1
```

Wie bereits erwähnt besitzt Lua eine dynamische Typbindung. Es ist deshalb nicht nötig, daß Sie bei der Zuweisung Aussagen über den zuzuweisenden Typ machen müssen. Lua passt den Typ einer Variable automatisch an den zugewiesenen Typ an.

```
1  x = 1
2  x = "Hello World"
3  print( x )        --> Hello World
```

Lua unterstützt zudem Mehrfach-Zuweisungen. Das bedeutet: Eine Anzahl von Werten wird gleichzeitig einer Anzahl von Variablen zugewiesen. Wir werden dieses nette Feature in einem späteren Abschnitt im Zusammenhang mit Funktion und mehreren Rückgabewerten diskutieren. Für die neugierigen Leser unter Ihnen hier ein kleines Beispiel welches den Wert zweier Variablen austauscht ohne eine zusätzliche Variable zur Zwischenspeicherung zu verwenden:

```
1  x = 5
2  y = 10
3  x,y = y,x
4  print( x, y ) --> 10    5
```

18.2.8.2 Globale und lokale Variablen

In Lua existieren drei Arten von Variablen: Globale Variablen, lokale Variablen und Tabellen (wir diskutieren Tabellen später).
Per Default sind alle Variablen global was bedeutet: Auf jede Variable kann während der Programm Laufzeit zugegriffen werden. Globale Variablen sind Teil eines 'globalen' Speicherorts oder besser gesagt Teil einer globalen Tabelle.
Wohin gegen lokale Variablen nur in ihrem jeweiligen Kontext oder Ausführungsblock sichtbar sind, in dem Sie deklariert wurden.

```
1  y = 10
2  if x == nil then
3      local y = 5
4      x = 1
5  end
6  print( x, y )       --> 1     10
```

Es ist eine übliche Strategie für bessere Programme, lokale Variablen einzusetzen, wann immer sie nicht global verwendet werden. Zudem beugen lokale Variablen unliebsame Seiteneffekten vor. Beispielsweise sollten Sie in Funktionen möglichst nur lokale Variablen verwenden.

18.2.9 Operatoren

Operatoren sind Symbole, die Berechnungen auslösen, wenn sie auf Variablen, Werte oder Ergebnisse von Ausdrücken angewendet werden. Lua unterstützt arithmetische, bitweise, Vergleichs- und logische Operatoren. Zusätzlich einen sehr nützlichen Operator zur Verkettung von Zeichenfolgen (Strings).

18.2.9.1 Arithmetische Operatoren

Lua verwendet die üblichen arithmetischen Operatoren: die binären + (Addition), – (Subtraktion), * (Multiplikation), / (Division), % (Modulo), ^ (Exponent) und unäre – (Negation) Operatoren.

```
+    -    *    /    %    ^    //
```

Der letzte Operator // ist ein spezieller Divisor-Operator und gibt einen gerundeten Quotienten zurück (gegen minus unendlich, was dem Grundwert der Division seiner Operanden entspricht).
Und nicht vergessen! In Lua werden alle Zahlen intern durch Fließkommazahlen mit doppelter Genauigkeit dargestellt.

18.2.9.2 Bitweise Operatoren

Der in der Analyser Software verwendete Lua Interpreter unterstützt die folgenden bitweisen Operatoren:

- & bitweise UND
- | bitweise ODER
- ~ bitweise exklusiv ODER
- >> rechts schieben
- << links schieben
- ~ unäres bitweises NICHT

Bitte beachten Sie: All bitweisen Operatoren wandeln ihre Operanden in eine Integer Zahl um, bevor der eigentliche Operator angewendet wird. Das Ergebnis ist immer eine Integer Zahl.

18.2.9.3 Vergleichsoperatoren

Vergleichsoperatoren vergleichen Variablen, Werte oder Ausdrücke und liefern immer ein **true** oder **false**. Lua unterstützt die folgenden vergleichenden- bzw. relationalen Operatoren:

```
<    >    <=   >=   ==   ~=
```

Der Gleichheits-Operator == testet auf Gleichheit, das Gegenteil ist der Operator ~= der auf Ungleichheit prüft. Sie können alle Operatoren auf beliebige zwei Werte anwenden, d.h. Zahlen und auch Zeichenketten (alle Vergleichsoperatoren arbeiten auch mit Zeichenketten). Sind beide Werte von unterschiedlichem Typ behandelt sie Lua als ungleich.
Beachten Sie: Der Wert 0 liefert im Gegensatz zu anderen Sprachen kein false.

```
1 print("abc" < "def" ) --> true
2 print( 0 or true)     --> 0
3 print( false or true) --> true
```

18.2.9.4 Logische Operatoren

Lua bietet die folgende logischen Operatoren in Ausdrücken: **and**, **or** und **not**. Alle logischen Operatoren verhalten sich alle auf gleiche Art und Weise. Alle liefern entweder ein **true** oder **false**. Als Spezialfall wird der Wert **nil** als **false** verarbeitet.

and und **or** bieten eine Kurzauswertung: Sie werten den zweiten Operanden nur aus, wenn es auf Grund des ersten Wertes nötig ist. Ein paar Beispiele:

```
1  print( 4 and 5 )          --> 5
2  print( 4 or 5 )           --> 4 (short-cut evaluation)
3  print( false and true )   --> false (short-cut evaluation)
4  print( a and 1 )          --> nil, because a wasn't specified
5  print( not false )        --> true
```

18.2.9.5 String Verkettungs-Operator

Zwei aufeinander folgende Punkte `..` bezeichnen den String Verkettungs-Operator in Lua. Ähnlich dem binären '+' Operator addiert dieser zwei Zeichenketten zu einer, indem er zweite an die erste anhängt. Zahlen werden dabei automatisch in Strings umgewandelt. Beachten Sie aber, das auf eine Zahl mindestens ein Leerzeichen folgen muß, da Lua sonst das erste der beiden Punkte `..` als Dezimalpunkt betrachtet und der ganze Ausdruck ungültig wird.

```
1  print( "Hello " .. "World" ) --> Hello World
2  print( 100 .. "sec" )       --> 100sec
```

Der Verkettungs-Operator liefert 'immer' eine neue Zeichenkette und läßt die beiden Operanden unverändert.

18.2.9.6 Der Längen-Operator

Der Längen-Operator ist definiert als `#` und liefert die Anzahl der Zeichen einer Zeichenkette oder die Anzahl von Einträgen in einer Tabelle (sofern diese keine Lücken ausweist).

```
1  print( #"Hello World" ) --> 11
```

18.2.9.7 Rangfolge

Die folgende Tabelle listet alle Lua Operatoren in der Reihenfolge ihrer Priorität (von niedrigster zu höchster Priorität).

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

Wenn Sie Zweifel über die Ausführungsreihenfolge habe, verwenden Sie Klammern. Das macht Ihren Code nicht nur lesbarer sondern vermeidet unnötiges Nachschlagen in diesem Handbuch.

18.2.10 Kontrollstrukturen

Kontrollstrukturen steuern den Programmfluß und sind integraler Bestandteil einer jeden Programmiersprache.
Lua stellt folgende Kontrollstrukturen zur Verfügung: **if** für bedingte Ausführungen, sowie **for**, **repeat** und **while** für wiederholende Ausführungen. Mit Ausnahme von **repeat** müssen alle mit einem expliziten **end** beendet werden.
repeat Ausführungsblöcke werden mit einem **until** geschlossen.

18.2.10.1 if then else

if wertet einen Ausdruck aus und führt abhängig von dem Ergebnis entweder die Anweisungen im **then** Abschnitt oder die Programmzeilen unter **else** aus. Letztere ist optional und kann entfallen wenn für eine nicht zutreffende Bedingung nichts zu tun ist.

```
1  if x < 0 then
2      x = 0
3  else
4      x = math.sqrt( x )
5  end
```

Kurze Bedingungsausdrücke können in einer Zeile geschrieben werden:

```
1  function max( a, b )
2      if a > b then return a else return b end
3  end
```

Lua kennt keine switch Anweisung wie sie z.B. unter C üblich sind um zwischen mehreren Alternativen zu wählen. Mit einer Kette von **if**, **elseif** Anweisungen ereichen Sie allerdings das gleiche Resultat.

```
1  if a >= 100 then
2      exp = 2
3  elseif a >= 10 then
4      exp = 1
5  else
6      exp = 0
7  end
```

18.2.10.2 while

Der **while** Befehl führt einen Block von Anweisungen aus, solange die Bedingung wahr ist. Üblicherweise wird zuerst die Bedingung geprüft, bevor der Block ausgeführt wird. D.h. der Block wird niemals ausgeführt wenn die Bedingung falsch ist.

```
1  local x = 0
2  while x < 10 do
3      x = x + 1
4  end
```

18.2.10.3 repeat

Der **repeat** Befehl führt den folgenden Anweisungsblock aus, solange die Bedingung wahr ist. Im Gegensatz zu **while** wird die Bedingung erst nach Ausführung des Blocks geprüft, dieser deshalb auf jeden Fall mindestens einmal ausgeführt. Beachten Sie, das eine **repeat** Anweisung mit **until** abgeschlossen werden muß.

```
1  local x = 0
2  do
3      x = x + 1
4  until x < 10
```

18.2.10.4 Numerisches for

Lua besitzt zwei Arten der **for** Anweisung. Wir beschränken uns hier auf die erste und eingängigere numerische Variante. Das numerische **for** besitzt eine Laufvariable der ein Startwert zugewiesen wird, einem Endwert sowie optional einer Schrittweite, in der der Startwert bis zum Erreichen des Endwerts erhöht wird. Die Schrittweite ist per Vorgabe 1 wenn sie nicht explizit angegeben wird.

```
1  for var=from,to,step do
2      --> do something
3  end
```

Beispielsweise

```
1  local m = 0
2  for n=0, 9, 0.1 do
3      m = m + 1
4  end
5  print( m )  --> 5.5
```

18.2.10.5 break

Die **break** Anweisung bricht die Anweisungen innerhalb einer **for**, **repeat** oder **while** Schleife ab und setzt die Ausführung nach der Schleife fort.

```
1  local m = 0
2  for n=0, 9, 0.1 do
3      m = m + 1
4      if m == 2.5 then
5          break
6      end
7  end
8  print( m )  --> 2.5
```

18.2.11 Funktionen

Jede Programmiersprache - selbst sehr einfache - verwenden Funktionen. Lua ist hier keine Ausnahme. Eine Funktion führt eine bestimmte Aufgabe oder Berechnung aus und liefert die Ergebnisse zurück.
Sie lesen richtig - Ergebnisse, plural! Funktionen in Lua können eines oder mehrere Resultate zurück geben.
In beiden Fällen übergeben Sie der Funktion eine Liste von in Klammern eingeschlossenen Argumenten oder Parametern. Wenn die Funktion keine Argumente benötig wird einfach eine leere Liste definiert durch `()` angegeben.

18.2.11.1 Funktionsaufruf

Einfach ausgedrückt wird eine Funktion durch ihren Namen und eine optionale Anzahl von Argumenten aufgerufen. Sie können eine Funktion mit mehr Parametern als angegeben aufrufen. In diesem Fall werden nur die spezifizierten verwendet. Ein Funktionsaufruf mit zu wenig Parametern liefert allerdings einen Fehler.
Sie kennen bereits einige vordefinierten Funktionen wie z.B. `print(...)` oder

`math.sqrt(x)`. Die meisten sind Teil eines Moduls oder anderweitig durch die Analyser Software definiert.

18.2.11.2 Funktionsdefinition

Funktionen werden ganz konventionell mit dem Schlüsselwort **function** definiert.

```
1  function fnc( arg1, arg2, ... )
2      --> the function body
3  end
```

Das folgende Beispiel definiert eine Maximum-Funktion, die das jeweils größere der beiden übergebenen Werte zurück liefert.

```
1  function max( n1, n2 )
2      if n1 > n2 then return n1 else return n2 end
3  end
```

Funktionen müssen nicht unbedingt einen Wert (oder mehrere) zurückgeben. In diesem Fall können Sie einfach die **return** Anweisung weg lassen oder die Funktion mit einem einfachen **return** ohne weitere Werte beenden.
Wie bereits erwähnt können Funktionen in Lua mehrere Ergebnisse zurück geben. Dies ist ein großer Vorteil, da Sie die Ergebnisse nicht in irgendeinem Konstrukt (Behälter) zusammenfassen müssen den Sie dann als Resultat zurück liefern. Es besteht auch keine Notwendigkeit, Ergebnisse in globalen Variablen außerhalb der Funktion zu speichern, was zudem zu unangenehmen Seiteneffekten führen kann.
Eine Reihe von vordefinierten Funktionen machen von dieser Fähigkeit gebrauch. Beispielsweise liefert die Funktion `math.modf(x)` aus dem math Modul den ganzen und den gebrochenen Anteil der übergebenen Zahl x zurück.

```
1  print( math.modf( 5.125 ) ) --> 5      0.125
```

Die Definition einer Funktion mit mehreren Rückgabewerten ist genauso einfach wie die jeder anderen. Das folgende Beispiel zeigt die Unterschiede. Stellen Sie sich eine Funktion vor, die zwei polare Koordinaten (Radius und WInkel) in das karthesische Koordinatensystem (x und y) umwandeln soll.

```
1  function polar2cartesian( radius, angle )
2      x = radius * math.sin( math.rad( phi ) )
3      y = radius * math.cos( math.rad( phi ) )
4      return x,y
5  end
```

Anstelle irgendeines Behälters oder einer Tabelle liefern wir einfach beide Werte als vielfaches Resultat zurück.

18.2.11.3 Rekursive Funktionen

Als nächstes wollen wir eine Funktion zur Berechnung der Fakultät einer beliebigen Zahl schreiben (dem mathematischen Äquivalent zu $n!$):

```
1  function faculty( n )
2      if n == 0 then
3          return 1
4      else
5          return n * faculty( n - 1 )
6      end
7  end
8
9  print( faculty( 5 ) ) --> 120
```

Rekursive Funktionen bieten oftmals elegante und was Code Zeilen anbelangt - kompakte Lösungen. Sie haben allerdings ihren Preis.

Betrachten Sie dazu den `faculty` Code. Mit jeder Erhöhung der übergebenen Zahl `n` wächst die Rekursionstiefe. Für eine Programmiersprache bedeutet dies: Jedes mal wenn eine Funktion aufgerufen wird und nicht zurückkehrt erhöht sich der Stack Verbrauch[9]. Ist der verfügbare Stack Speicher aufgebraucht stürzt das Programm in der Regel ab!

Sie können natürlich die Rekursionstiefe limitieren indem Sie für `n` nur eine maximale Größe zulassen. Dieser Ansatz erfordert aber ein diszipliniertes Verhalten für jeden Programmierer - und wird gerne einmal vergessen.
Zum Glück besitzt die Lua Implementierung der Analyser Software einen integrierten 'Wächter'. Ändern Sie dazu den Aufruf in Zeile 9 zu:

```
print( faculty( 30 ) ) --> [string "--[[..."]:11: stack overflow
```

Die Analyzer Software limitiert die maximale Rekursionstiefe auf einen Wert der einen guten Kompromiss zwischen Speicherverbrauch und Prozessor Last darstellt.
Dies ist besonders wichtig wenn Sie Protokoll Templates erstellen die in Echtzeit ausgeführt werden!

Die meisten rekursiven Funktionen können auch nicht-rekursiv implementiert werden. Eine alternative `faculty` Funktion könnte so aussehen:

```
1  function faculty_non_recursive( n )
2      local sum = 1
3      if n <= 1 then
4          return 1
5      else
6          for i=2,n do
7              sum = sum * i
8          end
9      end
10     return sum
11 end
12
13 print( faculty_non_recursive( 100 ) ) --> 9,3326215443944e+157
```

Diese Variante konsumiert hauptsächlich Prozessor Zeit. Der Stack Verbrauch ist minimal, verwendet werden lediglich die Variablen `sum` und der Iterator Zähler `i`.

18.2.12 Module

Ein Modul ist ein Zusammenstellung von Funktionen für bestimmte Zwecke. Sie kennen bereits das Modul `string`, `math` und `table`. Daneben existieren spezielle - Analyser spezifische - Module welche nicht Teil des Standard Lua sind. Wir beschreiben diese in einem eigenen Kapitel 19.
Und: Vergessen Sie nicht Ihre eigenen in Lua geschriebenen Module wie unser `checksum` Beispiel zuvor.

[9] Der Stack ist Teil des normalen Speichers und dient u.a. zur Speicherung der lokalen Variablen in Funktionen.

Aus der Sicht von Lua ist ein Modul schlicht eine Tabelle, die verschiedene Funktionen, Modul-Variablen (Funktionen sind für Lua ebenfalls nur ein Variablentyp), Modul-Konstanten etc. enthält. Eine Modul Funktion wird deshalb wie ein Tabellenelement mit dem Modulnamen gefolgt von einem Punkt und der eigentlichen Funktionsbezeichnung aufgerufen.

Wie Sie Lua um eigene Module erweitern können war bereits ein Thema im Abschnitt 18.1.5. Hier geben wir deshalb nur einen Überblick über die Standard Module, die von der Analyzer Lua Implementierung zur Verfügung gestellt werden. Alle Module beziehen sich dabei auf die Lua Version 5.3.

18.2.12.1 Standard Module

Die folgenden Standard Module werden von dem in der Analyser Software integrierten Lua Interpreter unterstützt. Sie arbeiten in allen Views wenn nicht anders angegeben.

Modul	Beschreibung
coroutine	Coroutine bieten eine unabhängige Programm Ausführung in Form eines kooperativen Multitasking. Sie sind Standard Bestandteil von Lua aber wir empfehlen diese nur einzusetzen wenn Sie wissen was Sie tun.
math	Die mathematische Bibliothek. Sie enthält alle vom C Standard definierten mathematischen Funktionen.
os	Teilweise unterstützt! `os` bietet Zugriff auf bestimme OS spezifische Funktionen wie Datum, Zeit und Länder/Spracheinstellung (locale). Andere Funktionen zur Ausführung externer Programme, Umbenennen und Löschen von Dateien sind dagegen ausgeschlossen.
string	Enthält eine ganze Reihe von Funktionen zur Manipulierung, zum Suchen/Ersetzen und Extrahierung von Zeichenketten. Reguläre Ausdrücke werden ebenfalls unterstützt.
table	Die Tabellen Bibliothek mit speziellen Funktionen zur Bearbeitung von Tabellen (Assoziativen Arrays) und Listen (Tabellen mit numerischem Index).

Folgende Lua Module Werden von der Analyser Lua Implementierung NICHT unterstützt.

```
debug (ersetzt durch eigenes debug Modul)
io
```

Im `os` Modul sind die nachstehenden Funktionen verfügbar:

```
os.clock
os.date
os.difftime,
os.setlocale
os.time
```

Eine kurze aber sehr gute Referenz (allerdings nur in englisch) über Lua und seine Standard Module als PDF Datei finden Sie unter:

http://lua-users.org/wiki/LuaShortReference
Die Referenz ist ebenfalls Teil der MSB-RS485-PLUS Software und wird bei der Installation im `doc` Verzeichnis abgelegt.

18.3 Lua Einschränkungen

Rechenintensive Lua Skripte oder Endlosschleifen können dazu führen, daß das zugehörige View nicht mehr oder nur sehr träge reagiert. Die integrierte virtuelle Lua Maschine (Lua VM) bricht deshalb Anweisungen oder Skripte, die ein bestimmtes Zeitmaß überschreiten, einfach ab und Sie werden durch eine entsprechende Meldung über diesen Umstand informiert:
Probieren Sie einmal folgendes im SKETCH Buffer des Editors:

```
1  local x = 0
2  while true do
3      x = x + 1
4  end


--> [string "local x = 0..."]:-1: overrun of allowed executions
```

Mit dem `config` Modul können Sie die erlaubte Anzahl von Ausführungen (internen Lua Befehlen) in einem vorgegebenen Bereich einstellen. Wie die Rekursionstiefe stellt dieser einen guten Kompromiss zwischen CPU Last und Ausführungszeit (in der das Skript die Anwendung blockiert) dar.
Das `config` Modul ist im Abschnitt 19.2.6 beschrieben.

18.4 Lua Referenzen

Dieses Kapitel hat nicht den Anspruch eine gute Einführung in Lua zu ersetzen. Es enthält lediglich die nötigsten Information um Ihnen die ersten Schritte mit Lua innerhalb der MSB-RS485-PLUS Software zu erleichtern. Und es gibt Ihnen vielleicht eine vage Vorstellung der Eigenschaften die Lua so besonders auszeichnen.
Die Skriptsprache Lua ist frei verfügbar und sehr gut dokumentiert. Im Internet finden Sie eine Vielzahl von Programmcode, Beispielen und Anleitungen. Eine sehr gute (wenn nicht die beste!) Quelle ist die Lua Website:

http://www.lua.org

Der folgende Link bietet eine gute Einführung in diverse Lua Themen:

http://lua-users.org/wiki/TutorialDirectory

Eine Online Lua Anleitung für Version 5.3, wie sie in der Analyser Software verwendet wird, ist hier erhältlich:

http://www.lua.org/manual/5.3/

19 Lua Analyser Erweiterungen

Um die Analyser Fähigkeiten bestmöglich unter Lua ausnutzen zu können bietet die Software zusätzliche Module und Datenstrukturen. Manche davon sind explizit bestimmten Views vorbehalten, andere können ohne Einschränkung in allen Views verwendet werden.

19.1 Übersicht der Module

Die folgende Tabelle listet alle zusätzlichen Module in alphabetischer Reihenfolge, ihre Verfügbarkeit in den Views und eine kurze Beschreibung. Die Module werden anschließend detailliert beschrieben und ihre Anwendung anhand von Beispielen genauer erläutert.
Module die exklusiv nur bestimmten Views vorbehalten sind (z.B. Protokoll- oder Datenmonitor) werden in den jeweiligen Kapiteln behandelt und hier nur der Vollständigkeit halber und kursiv aufgeführt.

Name	Usage	Description
base16	Allgemein	(De)Codierungs-Funktionen für base16 Sequenzen wie z.B. in Modbus ASCII Übertragungen.
bit32	OBSOLETE!	Dieses Modul stellte Bitweise Operationen für 32 Bit Werte zur Verfügung. Durch die Einführung der neuen bit-weise Operatoren in Lua wird dieses nun nicht länger benötigt und in einer der nächsten Versionen entfernt. Siehe 19.2.2 für die Details.
box	*ProtocolView*	*Nur Protokollmonitor! Das Box Modul ist verantwortlich für die Darstellung der Telegramme im Protokollmonitor.*
bpack	OBSOLETE!	Diese Funktion wurde durch die nun offiziell in Lua verfügbare `string.pack` Funktion ersetzt. Siehe auch 19.2.3.
bunpack	OBSOLETE!	Umkehrung von `bpack`. Auch diese Funktion wurde durch die nun in Lua vorhandene `string.pack` Funktion ersetzt. Siehe auch 19.2.3.

Obsolete Module bit32 und b(un)pack

checksum	Allgemein	Enthält checksum Algorithmen für Modbus RTU (CRC16), Modbus ASCII (LRC), BACNet (CRC8 und CRC16), DNP3, CRC16 CCITT (Kermit) und weitere.
config	Allgemein	Das `config` Modul dient zur Anpassung interner Lua Interpreter Einstellungen. Ersetzt das frühere `cfg` Modul.
data	*DataView*	*Nur Datenmonitor! Bietet wahlfreien Zugriff auf die aufgenommen Daten sowie die Möglichkeit, Datenbytes relativ zur Cursorposition hervorzuheben.*
debug	*DataView, ProtocolView*	*Ermöglicht die Ausgabe von beliebigen Text oder Variablen (Inhalt) für Debug Zwecke. Die verfügbaren Funktionen sind identisch für den Daten- als auch Protokollmonitor. Da der Aufrufkontext aber unterschiedlich ist, werden diese getrennt im jeweiligen Datenmonitor bzw. Protokollmonitor Kapitel beschrieben.*
record	Allgemein	Liefert Informationen zur aktuellen oder geladenen Aufzeichnung, z.B. Zeitpunkt der Aufzeichnung, Bus Anschluss, Signalnamen und Analyser Typ mit dem die Aufzeichnung erstellt wurde.
string.dump	Allgemein	Erweitert das Lua `string` Modul um die Möglichkeit, strings als hexadezimale oder dezimale Datensequenz ausgeben zu können.
telegrams	*ProtocolView*	*Nur Protokollmonitor! Das `telegrams` Modul erlaubt wahlfreien Zugriff auf alle aufgenommenen Telegramm Sequenzen.*
transmission	Allgemein	Liefert Informationen über die Übertragungseinstellungen der aktuellen oder geladenen Aufzeichnung. U.a. Baudrate, Anzahl Datenbits, Parität und Stopbits. Es ersetzt das frühere `protocol` Modul!

19.2 Allgemeine Erweiterungen für alle Views

Die folgenden Module oder Funktionen funktionieren in allen Views und auch im SKETCH Buffer des Editors (SKETCH Beispiel).
Beachten Sie! Einige Beispiele sind zum besseren Verständnis gleichwohl für bestimmte Views geschrieben. In diesen Fällen wird das in der Kopfzeile des Beispiels explizit angezeigt.

19.2.1 Das base16 Modul

Das `base16` Modul erlaubt die Umwandlung beliebiger im base16 (Hex ASCII) vorliegender Telegramminhalte in ihre originale Binärdaten. Dies betrifft insbesondere das Modbus ASCII Protokoll oder SRecord Übertragungen.

Funktion	Beschreibung
decode	Dekodiert eine im base16 Format vorliegende Datensequenz und liefert dessen binären (Original)Inhalt als Lua String zurück.
encode	Wandelt einen gegebenen Lua String in seine base16 Repräsentation und liefert das Ergebnis als neuen String zurück.

19.2.1.1 base16.decode

Dekodiert eine Datenreihe im base16 Format in seine ursprüngliche Binärsequenz und gibt diese als String zurück. Die Umwandlung/Dekodierung stoppt automatisch wenn das Ende des übergebenen Strings erreicht wurde oder ein ungültiges Zeichen auftrat.

base16.decode(*string*)

- **string:** Eine im Base16 Format vorliegende Datensequenz.

Protokollmonitor Beispiel

```
1  function out()
2      -- extract the binary data of a Modbus-ASCII telegram
3      local tg = telegrams.this()
4      -- A Modbus ASCII telegram starts with a colon ':' and ends with
           CRLF.
5      -- The data in between (byte 2...third last) is coded in base16
6      local bindata = base16.decode( tg:string():sub( 2, -3 ) )
7  end
```

19.2.1.2 base16.encode

I.a. werden Sie diese Funktion nicht benötigen, da sie einen Lua String in base16 kodiert. Sie ist allerdings dann von Nutzen wenn Sie eine bestimmte Bytefolge im base16 Format sehen möchten, z.B. um diese mit den empfangenen Telegrammdaten vergleichen zu können.

base16.encode(*string*)

- **string:** Ein Lua String der ins Base16 Format überführt werden soll.

Protokollmonitor Beispiel

```
1  function out()
2      local seq = "hello world"
3      box.text{ caption="Base16", text=base16.encode( seq ) }
4  end
```

Sketch Beispiel

```
1  local seq = "hello world"
2  print( base16.encode( seq ) ) --> 40 20 00 00 00 00 03 E8
3  print( base16.decode( base16.encode( seq ) ) ) --> Hello world
```

19.2.2 Das bit32 Modul

bit32 ist obsolete

Mit der Integration von Lua 5.3 wird die Verwendung des bit32 Moduls obsolete. Aus Gründen der abwärts Kompatibilität wird es von der Analyser Software noch eine Zeit lang unterstützt. Deshalb hier - nach wie vor - die Modul Beschreibung.
Denken Sie allerdings daran, dass das Modul in einer der nächsten Version entfernt wird. Die neuen Lua bitweise Operatoren sind nicht zu viel einfacher anzuwenden - der Code ist dadurch auch viel leichter zu lesen[1].

Ein Hinweis zur Bit Breite: Das bit32 Module ist auf 32 Bit begrenzt (daher der Name). Größere Werte werden automatisch auf 32 bit gekürzt entsprechend dem Rest der Teilung durch 2^{32}. Dies gilt auch für die neuen Lua bitweise Operatoren unter Windows. Nur die 64-Bit Linux Programm Versionen erlauben das (bitweise) Rechnen mit echten 64 Bit Integer Werten! Und nun die Beschreibung des obsoleten `bit32` Moduls:

Auf Protokoll bzw. Datenebene werden Sie häufiger mit dem Problem konfrontiert, einzelne Bits auszuwerten oder - z.B. im Zusammenhang mit Berechnungen von Prüfsummen - Datenbytes bitweise zu modifizieren.
Das `bit32` Modul erweiterte den integrierten Lua Interpreter um die folgenden Funktionen:

Funktion	Beschreibung
band	liefert die bitweise UND Verknüpfung der beiden Parameter x1 und x2, z.B. `bit32.band(0xFF,0x01)`
bor	liefert die bitweise ODER Verknüpfung der beiden Parameter x1 und x2, z.B. `bit32.bor(0xFF,0x01)`
bxor	liefert die bitweise XOR (exklusiv oder) Verknüpfung der beiden Parameter x1 und x2, z.B. `bit32.bxor(0xFF,0x0F)`
bnot	das Resultat ist die logische Negation jedes einzelnen Bits (auch Einerkomplement). Dabei wird jede 1 durch eine 0 ersetzt und umgekehrt. Z.B. `bit32.bnot(0x55)`
lshift	verschiebt den übergebenen Wert x bitweise um n Bits nach links, z.B. `bit32.lshift(0x100,2)`
rshift	verschiebt den übergebenen Wert x bitweise um n Bits nach rechts, z.B. `bit32.rshift(0x1FF,1)`

Sketch Beispiel

[1] Ein bit32 Modul wurde in Lua Version 5.2 eingeführt. Seit Lua 5.3 sind Bitweise Operatoren Teil der Sprache.

```
1  print( string.format( "%X", bit32.band( 0xFF, 0x03 ) ) ) --> 3
2  print( string.format( "%X", bit32.bor( 0x03, 0x10 ) ) ) --> 13
3  print( string.format( "%X", bit32.bxor( 0x1001, 0x1000 ) ) ) --> 1
4  print( string.format( "%X", bit32.bnot( 0x1001 ) ) ) --> FFFFEFFE
5  print( string.format( "%X", bit32.lshift( 0x1000, 2 ) ) ) --> 4000
6  print( string.format( "%X", bit32.rshift( 0x4000, 2 ) ) ) --> 1000
```

19.2.3 Die bpack und bunpack Funktionen

bpack und bunpack sind obsolete!

Beide Funktionen sind ursprünglich der Lua lpack Bibliothek entliehen. Zwecks abwärts Kompatibilität sind sie weiterhin integraler Bestandteil des Lua Interpreters in der Analyser. Allerdings werden wir sie in einer der nächsten Versionen entfernen, da Lua nun die selbe Funktionalität im eigenen `string` Modul bietet.
Die folgende Beschreibung sowie die Hinweise unter 19.2.4 mögen deshalb als Hilfe dienen, eigene Skripte und Templates für zukünftige Versionen upzudaten.

Die Funktionen `bpack` und `bunpack` geben Ihnen die Möglichkeit beliebige Bytefolgen in Strings und vice versa umzuwandeln.
Die Arbeitsweise von `bunpack` ähnelt dabei der scanf Funktion in C und ist vermutlich die Funktion von beiden, die Sie am meisten verwenden werden. Entsprechend eines Formatstrings extrahiert diese aus eine übergebene Bytefolge (Lua String) ein oder mehrere Zahlen. Lua ist bei der Rückgabe von Funktionsaufrufen nicht auf einen einzelnen Wert beschränkt. `bunpack` kann deshalb beliebig viele Zahlenwerte in einem Aufruf zurück geben.
Mittels eines optionalen dritten Parameters können Sie den Start der Konvertierung unabhängig vom Beginn der Bytefolge festlegen.

```
pos,val1,... = bunpack( sequence, format, position )
```

Im folgenden sind die meist verwendeten und von `bunpack` verstandenen Format/Umwandlungs-Zeichen aufgelistet.

Format	Beschreibung
`b`	Interpretiert das nächste Byte als einzelnen nicht Vorzeichen behafteten Byte (8-Bit) Wert.
`c`	Wandelt das nächste Byte in einen Vorzeichen behafteten (8-Bit) Wert um.
`d`	Wandelt die nächsten 8 Bytes in eine Fließkommazahl mit doppelter Genauigkeit (64 bit).
`f`	Interpret eine Folge von 4 Bytes als Fließkommanzahl, einfache Genauigkeit, 32 bit.
`H`	Transformiert die nächsten 2 Bytes in eine vorzeichenlose 16-Bit Zahl (unsigned short).
`h`	Transformiert die nächsten 2 Bytes in eine Vorzeichen behaftete 16-Bit Zahl (short).

`I`	Wandelt eine Reihe von 4 Bytes in eine vorzeichenlose 32-Bit (unsigned long) Zahl.
`i`	Wandelt eine Reihe von 4 Bytes in einer Vorzeichen behaftetete 32-Bit Zahl (long)
`>`	Interpretiert die Bytefolge mit dem höchstwertigen Byte zuerst (big-endian order).
`<`	Interpret die Bytefolge mit dem niederwertigsten Byte zuerst (little-endian order).

bunpack(*sequence, format, position=1*)

- **sequence:** Eine Bytesequenz als Lua String mit den zu extrahierenden (unpack) Zahlen.
- **format:** Formatanweisung, enthält die Art und Reihenfolge, in der die Zahlen extrahiert werden sollen.
- **position:** Eine vom ersten Byte in der Sequenz abweichende Position, ab der die Umwandlung erfolgen soll.

Beispiel

Gegeben sei das Kommando 'Write Single Register' in einer Modbus RTU Übertragung. Der Aufbau des Telegramms ist wie folgt:

Dev	Fnc	Reg HI	Reg LO	Value HI	Value LO	CRC HI	CRC LO

Dieses Modbus Telegramm instruiert einen Busteilnehmer eine 16-Bit Zahl in das angegebene Register zu schreiben. Das Register ist adressiert mit einer 16-Bit Nummer. Die Registeradresse ist in dem 3ten und 4ten Byte, der Registerwert im 5ten und 6ten Byte. Die letzten beiden Bytes enthalten eine CRC16 Prüfsumme. Alle Bytefolgen sind im Big-Endian Format abgelegt.
Mit `bunpack` können Sie alle nötigen Informationen (u.a. Registeradresse, -Wert und Prüfsumme) in einem einzigen Schritt extrahieren.

```
1 function out()
2     -- extract the binary data of a Modbus-ASCII telegram
3     local tg = telegrams.this()
4     -- assume its a Write Single Register telegram
5     local pos,dev,fnc,reg,val,crc = bunpack(tg:string(),"bb>H>H>H",1)
6     end
7 end
```

Die Umwandlung (oder Extraktion) startet mit dem ersten Byte der Sequenz (Position 1) und interpretiert die folgenden Bytes gemäß der Instruktionen im Formatstring. Am Ende liefert die Funktion die Position für nachfolgende `bunpack` Aufrufe (pos) und füllt alle übrigen Varaiblen auf der linken Seite des Ausdrucks mit den einzelnen Umwandlungsergebnissen.

Dev	Fnc	Reg HI	Reg LO	Value HI	Value LO	CRC HI	CRC LO
B	B	>H		>H		>H	

Die Anzahl der Variablen auf der linken Seite muss nicht exakt der vom Formatstring geforderten Ergebnisse entsprechen. Lua sorgt dafür, das nur die vorhandenen Variablen 'gefüllt' werden. Der folgende Code ist deshalb völlig korrekt, läßt aber das Ergebnis der CRC16 Umwandlung unberücksichtigt.

```
local pos,dev,fnc,reg,val = bunpack(tg:string(),"bb>H>H>H",1)
```

bpack(*format, value1, values2, ...*)

- **format:** Das anzuwendende Umwandlungsformat für die nachfolgenden Zahlen/Werte.
- **value1, values2, ...:** Eine Komma separierte Liste von Lua .

Sketch Beispiel

```
1 seq = string.dump( bpack( ">f>I", 2.5, 1000 ) )
2 print( seq )  --> 40 20 00 00 00 00 03 E8
```

Zugegeben: `bpack` (bzw. das diese ersetzende `string.pack`) werden Sie vermutlich - wenn überhaupt - selten benötigen. Um aber auf die schnelle einmal im Editor Sketch Buffer die String Repräsentation einer Fließkommazahl oder eines 64 Bit Integer Wertes zu prüfen, ist diese als Hilfsmittel nicht zu unterschätzen.

19.2.4 string.pack und string.unpack

Wie bereits erwähnt: Mit der Integration von Lua Version 5.3 ist die Verwendung von `bpack` und `bunpack` obsolete. Lua bietet die gleiche Funktionalität jetzt als Teil des eigenen `string` Moduls. Und wenn man es genau betrachtet, macht dies auch Sinn. In beiden Funktionen wird schließlich ein String als Eingabe bzw. Ausgabe verwendet.
Die Arbeitsweise der neuen `string` Modul Funktionen ist fast identisch zu `bpack` und `bunpack`. Das gilt auch für die Format Angabe. Sie finden alle nötigen Details in der frei verfügbaren Lua online Dokumentation:
http://www.lua.org/manual/5.3/manual.

Der einzige Unterschied, den Sie beim Portieren von `bunpack` zum neuen `string.unpack` beachten müssen liegt in der abweichenden Reihenfolge der Aufrufparameter sowie der Rückgabewerte. Hier der direkte Vergleich:

```
pos, val1, ... = bunpack( sequence, format, position )
val1, ..., pos = string.unpack( format, sequence, position )
```

Der Parameter `position` ist in beiden Fällen optional!
Im Gegensatz zu `bunpack` ist der zurück gelieferte Positionswert (an dem die Transformation endete) nun der Letzte (bisher war es der Erste). Außerdem wird der `format` Parameter jetzt - identisch mit der `bpack` und `string.pack` Funktion - als erstes übergeben. Gefolgt vom eigentlichen Eingabe String. Dies berücksichtigt können wir unser früheres Beispiel einfach wie folgt umschreiben:

```
1 function out()
2     -- extract the binary data of a Modbus-ASCII telegram
3     local tg = telegrams.this()
```

```
4       -- assume its a Write Single Register telegram
5       local dev,fnc,reg,val,crc,pos=string.unpack("bb>H>H>H",tg:string())
6       end
7   end
```

19.2.5 Das checksum Modul

Der Protokollmonitor unterstützt z.Z. die folgenden Prüfsummen Algorithmen, weitere werden in nächster Zukunft folgen. Wie Sie Ihren eigenen Prüfsummenalgorithmus erstellen beschreibt Abschnitt 14.3.2.
Alle Funktionen des checksum Moduls erwarten einen Lua String als Parameter und liefern die Prüfsumme über alle in diesem String enthaltenen Datenbytes entsprechend dem ausgewählten Algorithmus. Das Ergebnis ist eine Integerzahl.
Beachten Sie, das die Reihenfolge bei 16 Bit Werten von der Applikation abhängig ist. So überträgt Modbus RTU (crc16) das niederwertige vor dem höherwertigen Byte.

Funktion	Beschreibung
crc8_bacnet	8 Bit Prüfsumme wie sie in BACNet (Header) Telegrammen verwendet wird.
crc16_bacnet	16 Bit Prüfsumme wie sie in BACNet Übertragungen verwendet wird.
crc16_ccitt_kermit	Berechnet die crc16 Prüfsumme der gegebenen Datensequenz, allerding mit einem anderen Startwert (Initialisierung) wie in CCITT Kermit üblich.
crc16_df1	Berechnet die crc16 Prüfsumme wie sie im Allen-Bradley DF1 Protokoll verwendet wird. Zurück gegeben wird ein 16 Bit Wert.
crc16_dnp3	Berechnet die crc16 Prüfsumme wie sie im DNP3 Protokoll verwendet wird. Zurück gegeben wird ein 16 Bit Wert.
lrc	Liefert die Prüfsumme der gegebenen Datensequenz als Longitudinal Redundancy Check (LRC), wie bei Modbus ASCII verwendet.
crc16_modbus	Berechnet die Modbus RTU (CRC16) Prüfsumme der übergebenen Datensequenz und gibt sie als 16 Bit Integerzahl zurück.

19.2.5.1 checksum.crc8_bacnet

Ein in BACNet (Header) Telegrammen verwendeter Prüfsummenalgorithmus. Das Resultat ist ein einzelnes Byte (8 Bit Wert).

checksum.crc8_bacnet(*String*)

- **String:** Die Daten als Lua String

Sketch Beispiel

```
print( checksum.crc8_bacnet( "Hello world" ) ) --> 157
```

Protokollmonitor Beispiel

```
function out()
    local tg = telegrams.this()
    -- checksum header crc
    local header_crc = checksum.crc8_bacnet(tg:string():sub(3,8) )
    ...
    -- checksum data crc
    local datalength = tg:data(6) * 256 + tg:data(7)
    local data_crc = checksum.crc16_bacnet(tg:string():sub(9, 9+
        datalength+1) )
end
```

19.2.5.2 checksum.crc16_bacnet

16 Bit CRC BACNet Prüfsummenalgorithmus. Das Resultat ist ein 16 Bit Integer Wert.

checksum.crc16_bacnet(*String*)

- **String:** Die Daten als Lua String

Sketch Beispiel

```
print( checksum.crc16_bacnet( "Hello world" ) ) --> 20985
```

Protokollmonitor Beispiel

```
function out()
    local tg = telegrams.this()
    -- checksum header crc
    local header_crc = checksum.crc8_bacnet(tg:string():sub(3,8) )
    ...
    -- checksum data crc
    local datalength = tg:data(6) * 256 + tg:data(7)
    local data_crc = checksum.crc16_bacnet(tg:string():sub(9, 9+
        datalength+1) )
end
```

19.2.5.3 checksum.crc16_ccitt_kermit

Liefert die CRC16 CCITT (Kermit) Prüfsumme des gegebenen Datenstrings als 16 Bit Integer.

checksum.crc16_ccitt_kermit(*String*)

- **String:** Die Datensequenz als Lua String

Sketch Beispiel

```
print( checksum.crc16_ccitt_kermit( "Hello world" ) ) --> 41426
```

Protokollmonitor Beispiel

```
function out()
    -- the following code checks the content of the entire message
        except
    -- for the last two byte (which are the checksum itself)
    local cks = checksum.crc16_ccitt_kermit(telegrams.this():string():
        sub(1,-3))
    box.text{ caption="Checksum", cks }
end
```

19.2.5.4 checksum.crc16_df1

CRC Allen-Bradley DF1 Prüfsummenalgorithmus. Das Resultat ist ein 16 Bit Integer Wert.

checksum.crc16_df1(*String*)

- **String:** the data as a string

Protokollmonitor Beispiel

```
function out()
    local tg = telegrams.this()
    -- extract the application data and substitute DLE DLE
    local data = tg:string():sub(6,-5):gsub(''\016\016'','\016'')
    -- checksum validation, the CRC16 is calculated from the STN (3th
        byte),
    -- STX (5th byte), the application data AND the final ETX
    local data = tg:string():sub(3,3)..tg:string():sub(5,5)..data..
                 tg:string():sub(-3,-3)
    local cks = checksum.crc16_df1(data)
    box.text{ caption="Checksum", cks }
end
```

19.2.5.5 checksum.crc16_dnp3

Liefert die CRC16 Prüfsumme des gegebenen Datenstrings als 16 Bit Integer wie im DNP3 Protokoll spezifiziert.

checksum.crc16_dnp3(*String*)

- **String:** Die Datensequenz als Lua String

Protokollmonitor Beispiel

```
function out()
    -- the following code checks the content of the entire message
        except
    -- for the last two byte (which are the checksum itself)
    local cks = checksum.crc16_dnp3(telegrams.this():string():sub(1,-3)
        )
```

```
5     box.text{ caption="Checksum", cks }
6 end
```

19.2.5.6 checksum.lrc

Ein Prüfsummenalgorithmus basierend auf dem Longitudinal Redundancy Checking wie er z.B. in Modbus ASCII Übertragungen verwendet wird. Das Resultat ist ein einzelnes Byte (8 Bit Wert).

checksum.lrc(*String*)

- **String:** Die Daten als Lua String

Protokollmonitor Beispiel

```
1 function out()
2     -- in Modbus ASCII each byte is sent as a two ASCII characters but
3     -- the checksum is calculated before encoding the message. So we
4     -- must decode it first with base16.decode
5     local bindata = base16.decode( telegrams.this():string():sub(2,-3)
          )
6     local cks = checksum.lrc( bindata )
7     box.text{ caption="Checksum", cks }
8 end
```

19.2.5.7 checksum.crc16_modbus

Liefert die Modbus RTU Prüfsumme der Datensequenz als 16 Bit Integer.

checksum.crc16_modbus(*String*)

- **String:** Die Datensequenz als Lua String

Das nächste Beispiel ist wieder für den Editor Sketch Buffer. Wir konstruieren ein Modbus RTU Master Kommando in Zeile 2 (Lua erlaubt die Eingabe beliebiger binärer 8-Bit Werte als Dezimalzahl per `\ddd`. Zeile 3 dient uns einfach zur nochmaligen Überprüfung.
Die Prüfsumme wird über alle Datenbytes mit Ausnahme der letzten beiden (der eigentlichen Checksum) berechnet. Wir verwenden die Lua `string.sub` Funktion um alle Bytes bis auf die letzten beiden zu extrahieren. Die Prüfsumme wird im Hex Format ausgeben. Low und High Byte sind vertauscht, da Modbus RTU das Low Byte for dem High Byte überträgt.

Sketch Example

```
1 -- Modbus RTU Read Holding Register, address = 2, count = 5
2 seq = "\001\003\000\002\000\005\036\009"
3 print( seq:dump() ) --> 01 03 00 02 00 05 24 09
4 cks = checksum.crc16_modbus( seq:sub( 1, -3 ) )
5 print( string.format( "%04x", cks ) ) --> 0924
```

Protokollmonitor Beispiel

```
1  function out()
2      -- calculates the checksum over the whole telegram except for the
3      -- for the last two byte (which are the checksum itself)
4      local cks = checksum.crc16_modbus(telegrams.this():string():sub
           (1,-3))
5      box.text{ caption="Checksum", cks }
6  end
```

19.2.6 Das config Modul

Das `config` Modul ist hauptsächlich dafür gedacht, die internen Einstellungen des Lua Interpreters zu ändern. Beachten Sie: Diese Vorgaben sind mit Bedacht gewählt, eine Änderung kann evtl. zu unerwünschtem Verhalten führen.
Ein Parameter - und bislang der einzige - ist die maximale Anzahl von erlaubten Ausführungseinheiten für ein Skript. Ohne eine solche Begrenzung könnte der Interpreter in eine Endlosschleife laufen und die Applikation unbedienbar machen.

Achtung! Das cfg Modul wurde in config umbenannt

Für ein besseres Verständnis wurde in der Analyser Version 5.0 das cfg Modul in config umbenannt.

Betrachten Sie dazu folgendes Beispiel:

```
1  while true do end
```

Die Zeile wird niemals beendet und blockiert das entsprechende Programm Fenster. Auf Grund des MultiView Konzepts sind die aktive Aufzeichnung sowie alle anderen offenen Views davon nicht betroffen. Trotzdem ist es ärgerlich, da Sie das blockierende Fenster per Taskmanager beenden müssen.
Um dies zu vermeiden beendet der Lua Interpretor die Ausführung eines Skriptes nach einer voreingestellten Anzahl von Ausführungseinheiten. Diese Anzahl hängt von der verfügbaren Rechenleistung des PCs ab, weshalb wir einen eher konservativen Wert voreingestellt haben.
Es passiert selten, aber sollte Ihr Skript CPU intensiven Code enthalten, z.B. bei verschlüsselten Telegrammen und/oder aufwendigen Checksum Berechnungen, könnte die folgende Fehlermeldung erscheinen:

```
Overrun of allowed executions!
```

Sie können dieses Verhalten jederzeit reproduzieren indem Sie obige Zeile im Sketch Buffer des Editors ausführen!

Die maximal erlaubte Anzahl der Ausführungseinheiten ist mit 10000 vorbesetzt und ausreichend selbst für kompliziertere Skripte. Wenn Sie mehr benötigen, fügen Sie folgende Zeile am Beginn Ihrers Template Skripts ein:

```
1  config.setmaxop(1000000)
```

Der erlaubte Zahlenbereich liegt zwischen 10000...1000000.

19.2.7 Das record Modul

Mit dem `record` Modul können Sie bestimmte Informationen zur aktuellen Aufzeichnung abfragen. Diese könnten dann wichtig werden, wenn Ihr Code vom verwendeten Analyser Typ oder dem Bus Anschluss abhängt. Zudem liefert das `record` Modul noch die eingestellten Signalnamen sowie die Startzeit der Aufzeichnung.

Funktion	Beschreibung
analyzer	liefert den für die Aufnahme verwendeten Analyzer Typ. 0 : MSB-RS232, 1 : MSB-RS485, 2 : MSB-RS232-PLUS, 3 : MSB-RS485-PLUS
buswiring	Zurück gegeben wird der eingestellte Bus Anschluss: 0 : 2-Draht-Abgriff, 1 : 2-Draht-Segment, 2 : 4-Draht-Abgriff, 3 : 4-Draht-Segment
signalnames	liefert alle 8 Signalnamen als Liste von Signal1 zu Signal8. Ein Beispiel: `s1,s2,s3,s4,s5,s6,s7,s8 = record.signalnames()`
starttime	liefert den Beginn der Aufzeichnung in Sekunden seit dem 1. Januar 1970 00:00:00 (wie bereits im datetime Modul verwendet).

19.2.7.1 record.analyzer

Gibt den aktuellen Analyser Typ zurück. Dies ist entweder der aktuell verwendete oder der Analyser mit dem eine geladene Aufzeichnung gemacht wurde. Diese Funktion ist hauptsächlich zum Behandlung von verschiedenen Analyser Typen in Ihren Skripten gedacht, insbesondere der neuen PLUS Serie.

record.analyzer()

SKETCH Example

```
analyzers = {
  [ 0 ] = "MSB-RS232",
  [ 1 ] = "MSB-RS485",
  [ 2 ] = "MSB-RS232-PLUS",
  [ 3 ] = "MSB-RS485-PLUS"
}
print( analyzers[ record.analyzer() ] )
```

19.2.7.2 record.buswiring

Liefert den aktuell eingestellten (nur RS485 Analyser) bzw. in einer geladenen Aufzeichnung gespeicherten Bus Anschluss.

record.buswiring()

Protokollmonitor Beispiel

```
1  function out()
2      local tg = telegrams.this()
3      if record.buswiring() == 1 or record.buswiring() == 3 then
4          -- we can use the tg:dir() to distinguish between request and
               response
5      end
6  end
```

19.2.7.3 record.signalnames

Liefert die verwendeten Signalnamen zurück, definiert entweder im Einstelldialog des Kontrollprogrammes oder in der geladenen Aufzeichnungsdatei.

record.signalnames()

SKETCH Example

```
1  local signames = {record.signalnames()}
2  print( signames[3]) -->  TxD
```

19.2.7.4 record.starttime

Liefert die Sekunden die seit der Epoche (00:00:00 UTC, 1. Januar, 1970) verstrichen sind. Sie können die `os.date` Funktion verwenden um das Datum nach Ihren eigenen Wünschen zu formatieren und auszugeben.

record.starttime()

Protokollmonitor Beispiel

```
1  function out()
2      local tg = telegrams.this()
3      local t = record.starttime() + tg:time()
4      box.text{ caption="Date/Time", text = os.date( "%X %x", t ) }
5      -- returns something like 08:50:44 16.04.2013
6  end
```

19.2.8 Die string dump Erweiterung

Diese Funktion liefert einen 'Hex dump' aller im String enthaltenen Bytes. Die Arbeitsweise ist vergleichbar mit `telegram:dump`, aber nicht auf den Telegramm Typ beschränkt und ermöglicht damit z.B. auch 'Hex dumps' von base16 Konvertierungen.

19.2.8.1 string.dump

Erzeugt einen neuen Lua String der alle im String enthaltenen Datenbytes als 2-stellige Hexadezimal- oder 3-stellige Dezimalwerte getrennt durch ein Separatorzeichen auflistet. Die Vorgabe ist Hex (base=16), der Default Separator ist ein Leerzeichen.
Beachten Sie bitte! `string.dump` ist nicht Teil des allgemeinen Lua Sprachumfangs und arbeitet nur innerhalb der Analyser Software.

string.dump(*str, base, sep*)

- **str:** Der Lua String der als Hex dump ausgegeben werden soll.
- **base:** Die verwendete Zahlenbasis, Vorgabe ist hex (base 16).
- **sep:** Ersetzt das vorgegebene Trennungsleerzeichen durch ein beliebig anderes Zeichen bzw. Zeichenfolge. Ein leerer String unterbindet die Ausgabe eines Trennungszeichens komplett.

Protokollmonitor Beispiel

```
1  function out()
2      -- access the current telegram (a Modbus ASCII telegram)
3      local tg = telegrams.this()
4      -- convert the telegram content in its binary representation
5      local bindata = base16.decode( tg:string():sub(2,-3) )
6      -- show the complete telegram content as hex dump
7      box.text{ caption="Data (hex)", text=string.dump( bindata ) }
8      -- or in a more object orientated manner, dec output and ':'
           separator
9      box.text{ caption="Data (hex)", text=bindata:dump( bindata, 10, ":"
           ) }
10 end
```

19.2.9 Das transmission Modul

Das `transmission` Modul ersetzt das bisherige `protocol` Module und kommt immer dann gelegen, wenn Sie Informationen über das zugrunde liegende Übertragungsprotokoll benötigen, z.B. die Baudrate, Anzahl der Datenbits oder welche Parität benötigen.

protocol Modul wurde in transmission umbenannt

Mit der Analyser Version 5.0 wurde das protocol Modul in transmission Modul umbenannt um Missverständnisse mit der Protokollschicht im Protokollmonitor zu vermeiden.

Funktion	Beschreibung
baudrate	Liefert die Baudrate der aktuellen bzw. geladenen Aufzeichnung.
bitpause	Diese Funktion berechnet die benötigte Zeit um die angegebene Anzahl von Bits zu senden und berücksichtigt dabei die aktuelle Baudrate. Profibus z.b. definiert eine Sendepause von 33 Bits als Telegramm Trennung. Das Resultat ist in Sekunden(bruchteilen).
bytepause	Ähnlich der vorhwerigen Funktion. Hier wird allerdings die benötigte Zeit zur Sendung für die angegebenen Bytes berechnet, wobei neben der aktuellen Baudrate auch Start-, Parity und Stopbit mit berücksichtigt werden. Modbus RTU verwendet eine Sendepause von 3.5 Byte als Telegramm Trennung. Das Resultat ist in Sekunden(bruchteilen).

databits	Liefert die Anzahl der verwendeten Datenbits. Das Resultat ist ein Wert im Bereich 5...9.
parity	Der Rückgabewert ist die aktuell verwendete Parity Einstellung, kodiert wie folgt: None = 0, Odd = 1, Even = 2, Mark = 3, Space = 4.

19.2.9.1 transmission.baudrate

Liefert die in der aktuellen Aufzeichnung verwendete Baudrate.

transmission.baudrate()

Protokollmonitor Beispiel

```
1  function split( data, intval, alter, str )
2      -- start a new telegram after a pause of 33 bits
3      if intval > 33 / transmission.baudrate() then
4          return STARTED
5      end
6      return MODIFIED
7  end
```

19.2.9.2 transmission.bitpause

Berechnet die notwendige Zeit um die angegebene Anzahl von Bits zu senden.

transmission.bitpause(*bits*)

- **bits** number of paused bits.

Protokollmonitor Beispiel

```
1  function split( data, intval, alter, str )
2      -- Profibus specifies a pause of 33 bits as a telegram delimiter
3      if intval > transmission.bitpause( 33 ) then
4          return STARTED
5      end
6      return MODIFIED
7  end
```

19.2.9.3 transmission.bytepause

Berechnet die nötige Zeit um die gegebene Anzahl von Bytes zu senden.

transmission.bytepause(*bytes*)

- **bytes** number of paused bytes.

Protokollmonitor Beispiel

```
function split( data, intval, alter, str )
    -- Modbus RTU specifies a pause of 3.5 bytes as a telegram
        delimiter
    if intval > transmission.bytepause( 3.5 ) then
        return STARTED
    end
    return MODIFIED
end
```

19.2.9.4 transmission.databits

Liefert die eingestellte Anzahl von Datenbits (Wortlänge) der aktuellen Aufzeichnung.

transmission.databits()

Protokollmonitor Beispiel

```
function output()
    local tg = telegrams.this()
    if transmission.databits() > 8 then
        -- discards the 9th bit and and uses a warning red background
        box.text{ caption="9Bit", text=tg:data % 256, bg=0xFF0000, fg=0
            }
    else
        box.text{ caption="8Bit", text=tg:data }
    end
end
```

19.2.9.5 transmission.parity

Achtung! Diese Funktion liefert die in der aktuellen Aufzeichnung verwendete Parity Einstellung NICHT das Parity-Bit eines einzelnen Bytes.

transmission.parity()

Protokollmonitor Beispiel

```
function out()
    if transmission.parity() ~= 2 then
        -- do something when parity is not even
        box.text{ caption="Warning", text="We need an even parity" }
        return
    end
end
```

19.3 View abhängige Lua Module

Einige Module arbeiten nur in der Umgebung bestimmter Views. In der Übersichtstabelle ganz am Anfang sind sie kursive gelistet.
Zum Beispiel benötigen das `box` und `telegrams` Modul den Protokollmonitor um auf die Telegramm Daten und das Telegramm Fenster zugreifen zu können. Das gleiche gilt für das `databytes` Modul. Dieses erlaubt wahlfreien Zugriff auf alle im Datenmonitor dargestellten Bytes und kann deshalb nicht außerhalb

des Datenmonitors verwendet werden.

Alle diese Module sind eng mit den jeweiligen Views verknüpft. Wir beschreiben Sie deshalb im Detail in den jeweiligen View Kapiteln.

`box` Modul siehe 14.8.1
`telegrams` Modul siehe 14.8.8
`data` Modul siehe 12.7
`debug` Modul siehe Protokollmonitor 14.8.2 sowie Datenmonitor 12.7.2

20
Lua Protokoll Dialoge

Das Design des Lua Protokoll-Template Mechanismus verspricht ein Höchstmaß an Flexibilität. Allerdings fehlt es an einer Möglichkeit das Verhalten des Templates interaktiv zu verändern. Zum Beispiel um bestimmte Protokoll Parameter anzupassen oder Such/Filter Regeln zu verändern.
Ein einfache Lua GUI Framework schließt diese Lücke.

Stellen Sie sich vor, Sie analysieren eine Modbus RTU Übertragung mit abweichenden Interframe Delay Zeiten. Das ist nicht unüblich, zwingt Sie aber das Modbus Template entsprechend anzupassen. Insbesondere wenn Sie verschiedene Modbus Feldbusse analysieren müssen, kann dies ziemlich aufwendig werden.
Ein anderes Beispiel ist das IEC60870-5-101 Protokoll welches in mindestens zwei Varianten existiert. Optional kann bei diesem Protokoll die Adresse eines Gerätes als ein Byte oder in zwei Byte kodiert sein.
Jedes Mal wenn Sie ein abweichendes Protokoll Template benötigen, müssen Sie den Template Editor starten, die entsprechende Code Zeile suchen und gemäß Ihren Vorgaben korrigieren. Sie können - natürlich - verschiedene Versionen eines Templates pflegen, die sich evtl. nur durch einige Konstanten unterscheiden. Dies kann aber nur eine Notlösung sein mangels einer besseren Alternative.

Was wir benötigen ist eine flexible Möglichkeit Protokoll spezifische Parameter einzugeben ohne dazu den Template Code ändern zu müssen. Mit anderen Worten ein grafisches User Interface (GUI).
Klar ist, dass jede Benutzer Interaktion abhängig vom ausgewählten Protokoll (Template) ist. Eine Einstellung von Modbus Parametern ist sicherlich anders als die von - sagen wir - Profibus. Die Implementierung eines grafischen User Interfaces muss deshalb Teil des verwendeten Protokoll Templates sein. Schließlich beeinflusst jede Benutzer Interaktion die aktuelle Telegramm Darstellung im Protokollmonitor.

Mit dem Protokollmonitor können Sie individuelle Einstellung/Setup-Dialoge für jedes einzelne Protokoll direkt im zugehörigen Template realisieren und diese dann jederzeit per Klick auf den 'Setup' Knopf unter der Telegramm Ansicht zu öffnen/starten.
Die Anzahl der zu verwendenden oder benötigten Eingabefelder/Elemente (Check-

boxen, Auswahlschalter, Texteingabe, etc.) sowie deren Anordnung ist allein Ihnen überlassen. Elemente können dabei interagieren, z.B. um Eingaben abhängig von bestimmten Einstellungen zu erlauben oder zu verbieten.

Zudem können Sie vorgeben ob eine bestimmte Interaktion des Benutzers nur die Darstellung der angezeigten Telegramme beeinflusst (z.B. Umschaltung zwischen hexadezimaler oder dezimaler Checksum Darstellung) oder ob eine komplette Neu-Evaluierung der Daten erforderlich ist (Änderung der Telegramm Start/Ende Bedingung in der `split` Funktion).

20.1 Wie funktioniert es?

Bevor wir uns detailliert mit der Dialog Programmierung beschäftigen, hier zunächst ein kurzer Überblick wie ein Dialog aufgebaut ist und wie Sie die Benutzer Eingaben an das Protokoll Template übergeben.

Ein Dialog wird durch Grafikelemente definiert, die Sie in Lua hinzufügen. Diese werden dann vom Protokollmonitor in einem Dialogfenster angezeigt und - ganz wichtig - mit einem zusätzlichen 'Anwenden' Knopf versehen wird.
Letzterer involviert den Lua Code, der für die Übergabe der Benutzer Eingaben an das eigentliche Template verantwortlich zeichnet. Damit besteht ein typischer Protokolldialog aus zwei Teilen.

1. Die Lua Funktion `dialog` enthält den Code für das grafische Benutzer Interface (GUI).
2. Die Lua Funktion `apply` wird durch den 'Anwenden' Knopf des Dialogs aufgerufen und aktualisiert das Protokoll Fenster.

Ein Dialog kann - natürlich - weitaus mehr Funktionen enthalten. Diese zwei sind aber essentiell für einen funktionstüchtigen Eingabedialog.
Sobald Sie den Setup Knopf klicken, wird die `dialog` Funktion in einem unabhängigen Lua Interpreter ausgeführt. Dies gewährleistet eine strikte Trennung und Entkopplung der grafischen Bedienoberfläche von der eigentlichen Datenstrom Evaluierung (Telegramm Extrahierung).
In der `apply` Funktion fragen Sie die Dialog Elemente nach deren aktuellem Status (Benutzer Eingaben) ab und übergeben diese dann an das eigentliche Protokoll Skript.

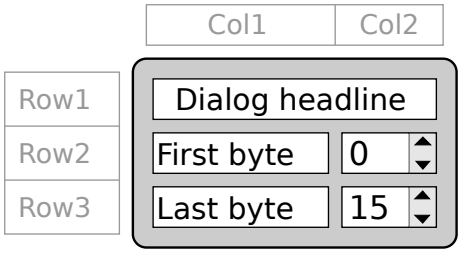
20.2 Das Dialog Gerüst

Wann immer es um grafische Bedienoberflächen geht ist eine der ersten Fragen: Wie werden die verschiedenen Bedienelemente angeordnet?

Hier gibt es mehrere Ansätze. Einer ist die absolute Positionierung der Elemente. Dabei wird Ort/Position und Größe innerhalb des Dialogs fest vorgegeben. Dies ist allerdings sehr umständlich und aufwendig.
Die Analyser Software benutzt einen deutlich eleganteren Weg um die einzelnen Bedienelemente anzuordnen. Dabei wird die verfügbare Fläche des Dialogs durch ein unsichtbares Raster überzogen. Breite und Höhe sowie Anzahl

der Spalten und Zeilen sind nicht limitiert und werden automatisch an die Bedienelemente angepasst.
Sie können sich das Raster wie ein Setzkasten vorstellen. Jedes Fach enthält ein einzelnes Bedienelement, wobei ein Bedienelement sich auch über mehrere horizontalen Fächer erstrecken kann. Durch diese Anordnung werden Benutzer freundliche und klare Bedienoberflächen erreicht.

Um ein Bedienelement in einem bestimmten 'Fach' zu platzieren übergeben Sie einfach die Spalten- und Zeilennummer des Faches. Das Raster wird dabei automatisch an die Breite und Höhe des neuen Elements angepasst.
Mehr noch:
Sie können jederzeit ein Element durch ein anderes ersetzen, (z.B. als Folge einer Interaktion durch den Anwender), indem Sie einem bestimmten 'Fach' einfach ein neues Element zuweisen. Betrachten Sie dazu folgendes Bild:



Dieser kleine Dialog besteht aus zwei Spalten und drei Zeilen, d.h. einem Raster von 2 x 3 und dient zur Eingabe eines definierten Zahlenbereich von/bis (first/last). Die erste Zeile wird komplett von der Kopfzeile ausgefüllt. (Das Element wurde mit einem entsprechenden `span=2` Parameter eingefügt).
Das linke 'Fach' oder Feld in der zweiten Zeile ist mit einem Text Label 'First Byte' besetzt. Das rechte enthält das zugehörige Eingabeelement für die ersten Zahl. Verwendet wird ein sogenanntes 'Spin Control'. Dieses erlaubt das inkrementelle Ändern durch schrittweises hoch-/runter zählen.
Die dritte Zeile besteht aus den gleichen Elementen, allerdings mit anderem Text Label und einem weiteren Spin Control Element zur Eingabe des zweiten Zahlenwertes.
Nicht verwendete Felder zwischen den einzelnen Elementen bleiben leer. Dies erlaubt ein Gruppieren von mehreren Elementen durch einfache räumliche Trennung.
Und hier ist der zugehörige Lua Code:

```
1  function dialog()
2      -- the headline label spanned over two columns
3      widgets.Label{ name="headline", text="Dialog headline",
4                     row=1, col=1, span=2 }
5      -- label and first byte input control
6      widgets.Label{ name="label1", text="First byte", row=2, col=1 }
7      widgets.SpinCtrl{ name="first", row=2, col=2,
8                        min=0, max=255, value=0 }
9     -- label and last byte input control
10     widgets.Label{ name="label2", text="Last byte", row=3, col=1 }
11     widgets.SpinCtrl{ name="last", row=3, col=2,
12                       min=0, max=255, value=15 }
13 end
```

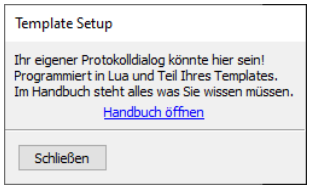
Machen Sie sich keine Sorgen, wenn Sie diese Zeilen nicht auf Anhieb verstehen. Wir werden im Folgenden all die nötigen Details erklären. Hier ging es lediglich darum, Ihnen einen kurzen Eindruck zu verschaffen, wie Sie einen kompletten Dialog mit ein paar Code Zeilen realisieren können.

20.3 Einen Template Dialog erstellen

Die gesamte Codierung Ihres Dialogs muss in der Funktion `dialog` erfolgen. Dies ist die einzige Lua Funktion, die der Interpreter bei der Ausführung der grafischen Bedienoberfläche aufruft.
Im Falle komplexerer Dialoge können Sie Teile des Codes auch in anderen Funktionen auslagern. Aber jede von diesen muss zumindest einmal aus dem Kontext der `dialog` Funktion aufgerufen werden.
Der Protokollmonitor fasst alle unterstützten Grafikelemente in einem Modul namens `widgets` zusammen. Ein Lua Modul ähnelt einer Bibliothek in anderen Programmiersprachen. Sie können es sich als eine Sammlung von Funktionen speziell für Bedienoberflächen vorstellen.
Um eine bestimmte Modulfunktion auszuführen erwartet Lua den Modulnamen, gefolgt von einem Punkt und dem eigentlichen Funktionsnamen. Für einen klickbaren 'Knopf' (Button) zum Beispiel:

```
1  widgets.Button{ PARAMETER... }
```

Neue Sprach Features werden am besten an einem Beispiel erklärt. Öffnen Sie deshalb zunächst das Dialog Tutorial Projekt im Beispiel (Examples) Verzeichnis (`Tutorial-Dialogs`).
Sie können es per Doppelklick aus Ihrem Dateibrowser starten. Oder sie klicken im Dateimenü des Kontrollprogramm den Eintrag 'Beispiele', wählen den Ordner Tutorial und öffnen dort die Projektdatei. Je nach ausgewähltem Analyser Typ wird das Programm neu gestartet.
Sollten Sie noch nicht das Beispiel zur Template Programmierung im Protokollmonitor Kapitel gelesen haben (siehe 14.3.2), ist es jetzt eine gute Idee, dies nachzuholen.



Leerer Dialog

Beginnen wir nun Schritt für Schritt mit der Erstellung eines eigenen Dialogs. Am Anfang ist dies nicht mehr als eine leere (oder nicht vorhandene) `dialog` Funktion.

```
1  function dialog()
2  end
```

Wenn Sie den Setup im Protokollmonitor klicken, erscheint ein Dialog wie links abgebildet. Der Dialog ist noch 'leer'. Er wird mit einem Default Titel angezeigt und enthält einen Link auf das entsprechende Kapitel im Handbuch (nämlich dieses).
Sie können die `dialog` Funktion an beliebiger Stelle in Ihrem Template einfügen. Allerdings nicht innerhalb einer anderen Funktion (was Lua explizit erlauben würde). Der Inhalt von `dialog` ist nicht begrenzt auf `widgets` Funktionen. Sie können allen möglichen Lua Code einbauen. Vermeiden Sie aber zeitaufwendige Berechnungen, da Ihr Dialog sonst unbedienbar wird.

20.3.1 Bedienelemente hinzufügen

In unserer Beispiel Aufzeichnung werden von einem Bus Master regelmäßig drei Sensoren abgefragt. Diese liefern jeweils Werte für Temperatur, Feuchtigkeit und Luftdruck. Die Telegramm Spezifikation ist einfach und ähnelt dem von Modbus ASCII. Jedes Telegramm beginnt mit einem Doppelpunkt ':' und endet mit einem Wagenrücklauf und Zeilenvorschub (Carriage return, line feed) oder CRLF. Hier ein Beispiel:

Time	SOF	Address	Function	Checksum	EOS
2.339189	:	2	Moisture	C4	0D 0A

Time	SOF	Address	Function	Value	Checksum	EOS
2.351468	:	2	Moisture	58.52%	7D	0D 0A

Wir haben bereits in Kapitel 14.4.2 darüber diskutiert, wie man bei der Telegramm Ausgabe wahlweise zwischen einer metrischen oder Anglo-Amerikanischen Darstellung der Messwerte hin- und her schalten kann. Dabei hatten wir als Lösung ein Filter vorgeschlagen.
Mangels einer besseren Alternative war dies vertretbar. Jetzt implementieren wir dies mit einem echten Dialog. Wir starten mit einem Auswahl zwischen 'Metrisch' oder 'Anglo-Amerikanisch'.
Um exakt eine von mehreren Möglichkeiten auszuwählen wird i.a. eine sogenannte Radio Box verwendet. Analog zu den Sendertasten eines alten Radioempfängers kann der Anwender nur eine aus mehreren Möglichkeiten auswählen. Fügen Sie jetzt folgende Code Zeilen am Ende des Template Skripts hinzu:

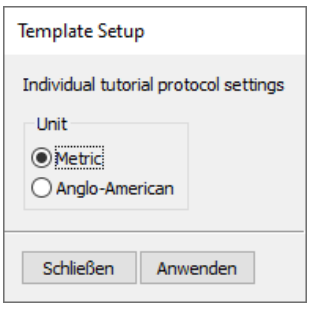
```
1 function dialog()
2     widgets.RadioBox{ name="unit", col=1, row=1,
3                       label="Unit"
4                       choices={"Metric","Anglo-American"}}
5 end
```

Jedes GUI Element (oder im folgenden Widget) benötigt zumindest einen einmaligen Namen und eine Position (Raster Spalte und Zeile). Im Augenblick genügt es zu wissen, das der Name zum späteren Zugriff auf das Widget dient und deshalb nicht doppelt vergeben werden darf.
Speichern Sie die Änderung und klicken Sie dann den Setup Knopf um den Dialog zu testen.

Dialog testen

Sie können Ihren Dialog jederzeit durch Klick auf den Setup Knopf testen sobald Sie Ihre Code Änderungen gespeichert haben.

Im Dialog sehen Sie eine Radio Box in der Sie wahlweise 'Metric' oder 'Anglo-American' auswählen können. Mehr passiert allerdings nicht. Das Skript selbst hat noch keine Idee was Sie im Dialogfenster eingeben.
Wie Sie Ihre Eingabe oder Auswahl dem Rest des Protokoll Templates mitteilen ist Gegenstand des nächsten Abschnitts. Bleiben wir noch ein wenig bei den Dialog Elementen um das ein oder andere Detail beim Arrangement sowie Bedienbarkeit zu erörtern.



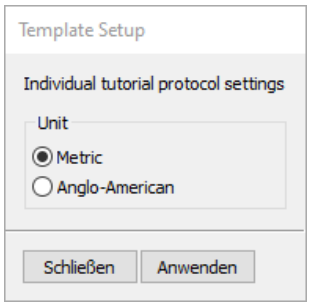
Dialog mit Auswahl

Dazu fügen wir als erstes einen beschreibenden Text über der Radio Box hinzu und korrigieren den Feld Parameter. Da die Überschrift in Zeile 1 steht, muss der Auswahlschalter für die Einheiten in der zweiten Zeile platziert werden.

```
1  function dialog()
2      widgets.Label{ text="Individual tutorial protocol settings",
3                     col=1, row=1 }
4      widgets.RadioBox{ name="unit",
5                        label="Unit",
6                        col=1, row=2,
7                        choices={"Metric","Anglo-American"}}
8  end
```

Das Resultat sehen Sie im Bild 'Dialog mit Auswahl'.
Der Dialog weist als Vorgabe jedem Element die kleinst mögliche Fläche zu. Wie Sie im Bild sehen verwendet die Radio Box nur einen Teil der durch die Überschrift vorgegebenen Breite. Dies ist nicht nur unschön, es ist auch schlechter GUI Still!

Sie können den Algorithmus für die Anordnung (Sizer genannt) aber auch instruieren, den verfügbaren Platz komplett durch das Widget ausfüllen zu lassen. Der entsprechende Widget Parameter lautet `fill=true`.



Filled radio box

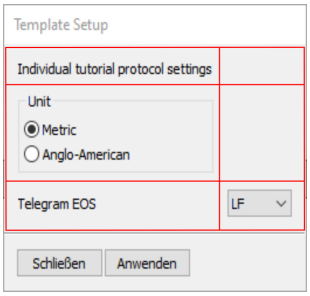
```
1  function dialog()
2      widgets.Label{ text="Individual tutorial protocol settings",
3                     col=1, row=1 }
4      widgets.RadioBox{ name="unit",
5                        label="Unit",
6                        col=1, row=2, fill=true,
7                        choices={"Metric","Anglo-American"}}
8  end
```

Das sieht schon besser aus! Der Auswahlschalter für die Einheit füllt nun die komplette Spalte und damit Breite des Dialogs aus. Letztere vorgegeben durch das Label Widget in der ersten Zeile des Rasters.

Das aktuelle Dialog Arrangement ist ein Raster mit einer Spalte und zwei Reihen. Aber wie kann sich ein Widget über mehrere Spalten erstrecken? Der `fill` Parameter beschränkt sich schließlich nur auf ein einzelnes Feld.
Wir hatten bereits in der Einleitung 20.2 kurz den Parameter `span` erwähnt.
Zum besseren Verständnis ergänzen wir den Dialog zunächst um ein weiteres Widget Element zur Auswahl verschiedener EOS Varianten. Sie erinnern sich? Unsere Telegramm Spezifikation sagt, das jedes Telegramm mit einem CRLF endet. Stellen Sie sich vor, dies wäre nur eine unter verschiedenen Varianten, z.B. CRLF, LF, CR oder LFCR.
Wir könnten dies mit einer weiteren Radio Box lösen, nehmen aber diesmal ein sogenanntes `Choice` Element.

```
1  function dialog()
2      widgets.Label{ text="Individual tutorial protocol settings",
3                     col=1, row=1 }
4      widgets.RadioBox{ name="unit",
5                        label="Unit",
6                        col=1, row=2, fill=true,
7                        choices={"Metric","Anglo-American"}}
8      widgets.label{ text="Telegram EOS",
9                     col=1, row=3 }
10     widgets.Choice{ name="eos",
```

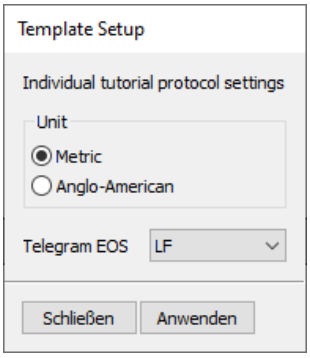
```
11                  col=2, row=3, fill=true,
12                  choices={"LF", "CR", "LFCR", "CRLF" } }
13  end
```



Dialog Raster 2 x 3

Da das `Choice` Widget kein Label zur Verfügung stellt, ergänzen wir ein solches mit einem weiteren `Label` Widget in Spalte 1 und platzieren `Choice` in Spalte 2 (Code Zeile 9 und 11).
Das Dialog Framework verhält sich exakt wie erwartet. Alle Elemente oder Widgets mit Ausnahme der EOS Auswahl (`Choice`) werden in Spalte 1 angeordnet. Das EOS Widget ist per `col=2` in der zweiten Spalte platziert - was nicht so schön aussieht (siehe Bild 'Dialog Raster 2 x 3').
Hier kommt nun der `span` Parameter ins Spiel. Indem wir im Aufruf des Überschrift Widgets und der Radio Box zusätzlich den Parameter `span=Spalten` übergeben, instruieren wir den Dialog Sizer diese Elemente über die entsprechenden Spalten zu 'spannen' (hier 2). Der folgende Programmcode zeigt die Verbesserungen in Zeile 3 und 6. Daneben das resultierende Erscheinungsbild des Dialogs.

```
1   function dialog()
2       widgets.Label{ text="Individual tutorial protocol settings",
3                      col=1, row=1, span=2 }
4       widgets.RadioBox{ name="unit",
5                         label="Unit",
6                         col=1, row=2, fill=true, span=2,
7                         choices={"Metric","Anglo-American"} }
8       widgets.Label{ text="Telegram EOS",
9                      col=1, row=3 }
10      widgets.Choice{ name="eos",
11                      col=2, row=3, fill=true,
12                      choices={"LF", "CR", "LFCR", "CRLF" } }
13  end
```



Dialog verbessert

Der zugrunde liegende Algorithmus ändert die Größe aller Widgets Elemente, so dass Sie möglichst wenig Platz beanspruchen ohne dabei abgeschnitten zu werden. Das Resultat ist ein aufgeräumter, klarer Dialog ohne unnötige Freiflächen.
Wenden wir unsere Aufmerksamkeit nun der Frage zu, wie die User Eingaben an die Stellen im Skript weiter gegeben werden, die für die Extrahierung der Telegramme aus dem Datenstrom sowie für die Telegramm Anzeige im Protokollfenster zuständig sind. Dazu müssen wir zunächst den aktuellen Status der Widgets abfragen.

20.3.2 Dialog Eingaben auswerten

Wir hatten bereits erwähnt, dass jedes Widget Element einen eindeutigen Namen bekommen muss. Dies ist nicht ganz korrekt. Alle `Label` Elemente in unserem Dialog z.B. haben gar keinen Namen.
Ein Widget Element benötigt immer nur dann einen eindeutigen Namen, wenn Sie später darauf zugreifen möchten. Und Labels sind in der Regel kein Objekt irgendeiner Benutzer Interaktionen. Im Gegensatz dazu müssen wir aber in der Lage sein z.B. den aktuellen Wert der Radio Box auszulesen um die Einstellung weiter verarbeiten zu können.
Zurück zu unserem Beispiel. In unserem Dialog kann der Anwender das Einheitensystem und/oder die EOS (End of String) Sequenz ändern. In der Einleitung sprachen wir bereits davon, dass jede `dialog` Funktion gewöhnlich immer von

einer `apply` Funktion begleitet wird[1], die jedes Mal ausgeführt wird wenn der Anwender den 'Anwenden' Knopf klickt.
Die `apply` Funktion ist damit der geeignete Ort um die Anwender Eingaben abzufragen und an den für die Telegramm Evaluierung und Anzeige zuständigen Code weiter zu reichen.
Beginnen wir mit dem Code:

```
1 function apply()
2     unit = widgets.GetValue( "unit" )
3     eos = widgets.GetValue( "eos" )
4     debug.clear()
5     debug.print( "Unit="..unit, "EOS="..eos )
6 end
```

Die Funktion fragt die aktuellen Einstellungen der Radio Box und der EOS Auswahl ab und weist diese den globalen Variablen `unit` und `eos` zu.
Beachten Sie! Alle Lua Variablen sind global solange Sie dies nicht explizit mit dem Schlüsselwort `local` anders deklarieren!
Zur Prüfung geben wir in Zeile 5 die aktuellen Einstellungen im Lua Debug Fenster aus. Zeile 4 dient dazu den Debug Fensterinhalt zuvor zu löschen. Sie können das Debug Fenster einfach im 'Ansicht' Menü des Protokollmonitor oder per Strg + Alt + O öffnen. Sobald Sie den 'Anwenden' Knopf des Dialogs klicken aktualisiert das Skript die Debug Ausgabe. Dies macht das Debug Fenster zu einem wertvollen Tool bei der Template und Dialog Programmierung.

Die von den Sensoren gelieferten Werte Temperatur, Feuchtigkeit und Druck werden in der Funktion `GetFunctionValue()` formatiert. Für ein Wechsel des Einheitensystems müssen wir diese zunächst um ein 'Anglo-Amerikanisches' Wertesystem erweitern.

```
1  function out( filter )
2      -- skip unchanged code here
3      function GetFunctionValue( number, value )
4          if unit == "Metric" then
5              local formats = { "%.2fC", "%.2f%%", "%imBar"}
6              return string.format( formats[ number ], value )
7          else
8              -- conversion factor for Temparature, Moisture and Pressure
9              local formats = { "%.2fF", "%.2f%%", "%.2fpsi"}
10             convs = {
11                 -- function to convert celsius to fahrenheit
12                 [1] = function(c) return c * 1.8 + 32 end,
13                 -- the mositure is always in percent
14                 [2] = function(m) return m end,
15                 --  function to convert mbar to psi
16                 [3] = function(p) return p * 0.01450377 end
17             }
18             return string.format( formats[ number ],
19                                   convs[number](value) )
20         end
21     end
```

Der Code wurde bereits im Kapitel 14.4.2 detailliert erläutert. Hier in Kürze:
In Zeile 4 wird die globale Variable `unit` mit 'Metric' verglichen und falls dies

[1] Es gibt Ausnahmen, wenn Sie z.B. jede Änderung im Dialog durch eine callback Funktion behandeln.

der Fall ist, die Sensorwerte wie gehabt formatiert. Wenn nicht, erfolgt die Umrechnung der Temperatur von Grad Celsius zu Fahrenheit und des Druckes von Bar zu psi. Die Feuchtigkeit ist ein Prozentwert und immer gleich.
Speichern Sie die Änderungen und rufen Sie den Dialog neu auf. Wechseln Sie dann das Einheitensystem in der Radio Box. Vergessen Sie nicht den 'Anwenden' Knopf zu klicken um die Änderungen zu übernehmen und die `apply` Funktion zu triggern.

Sie werden feststellen - das nichts passiert! Außer das die Radio Box Ihre letzte Änderung repräsentiert, zeigt die Telegramm Anzeige jetzt nur noch die Anglo-Amerikanische Einheiten - egal was Sie anklicken. Was läuft hier schief?

Seien Sie versichert, dass der Code völlig korrekt ist. Allerdings haben wir ein kleines, nicht desto trotz aber sehr wichtiges Detail unterschlagen. (Und mit völliger Absicht!).
Ein Protokoll Template Skript wird von mehreren Lua Interpretern ausgeführt die alle ihre 'eigenen' globalen Variablen besitzen. Ein Interpreter ist zuständig für die GUI und 'kümmert' sich explizit um die `dialog` und `apply` Funktionen. Dieser 'besitzt' die globalen Variablen `unit` und `eos`.

Der für die Darstellung der Telegramme (durch die Ausführung von `output`) verantwortliche Interpreter kennt keine `unit` Variable. Aus seiner Sicht ist die Variable `nil` (wie jede in Lua nicht initialisierte Variable) was letztendlich zum Aufruf des Anglo-American Block führt.

Interessant - nicht wahr? Aber sicher nicht die Antwort, die Sie erwartet haben. Lassen Sie uns zunächst die Gründe für dieses - auf den ersten Blick ungewöhnliche - Software Design erörtern bevor wir auf unser Beispiel zurück kommen.

20.3.3 Datenaustausch zwischen GUI und Template

Das Extrahieren der Telegramme in der `split` Funktion sowie deren Darstellung im Telegramm Fenster durch `output` erfolgen durch unabhängige Lua Interpreter. Da jeder Lua Interpreter seine eigenen globalen Variablen (Namensraum) besitzt (tatsächlich sind diese in einer individuellen Tabelle organisiert), können globale Variablen nicht zum Datenaustausch verwendet werden.

Auf den ersten Blick mag dieser Ansatz recht ungewöhnlich erscheinen. Aber dieses Design macht sehr viel Sinn wenn Sie folgendes bedenken:
Das Extrahieren der Telegramme hat absolute Priorität und sollte unter keinen Umständen durch die Bedienung des Dialogs beeinflusst, verzögert oder gar gestoppt werden.
Zudem bietet ein solcher Ansatz eine klare Trennung zwischen Daten/Protokoll Verarbeitung und Bedienoberfläche. Und vor allem: Der die `split` Funktion ausführende Lua Interpreter kommt ohne zusätzlichen Ballast in Form von grafischen Modulen/Funktionen etc. aus - was ihn extrem schnell macht!

D.h. selbst wenn Sie eine globale Variable `x` in der `split`, `output` und `apply` Funktion verwenden, sind diese - in Wirklichkeit - vier verschiedene Variablen! Vier - weil jedes Protokoll Template von bis zu vier Lua Interpretern ausgeführt

wird. Dies passiert nicht immer gleichzeitig. Z.B. gelangt der für die GUI zuständige Interpreter nur dann zum Einsatz, wenn Sie den Setup Knopf klicken.
Zwei Interpreter 'kümmern' sich um die `split` Funktion - einer für jede Datenrichtung. Auch hier kann es sein, das nur ein Interpreter zum Einsatz kommt, weil Sie nur an einem Kanal Daten empfangen. Und auch die `output` Funktion ist in einem eigenen Interpreter separiert.
Aber wenn es sich hier auch um unabhängige Lua Skript Interpreter handelt, so teilen sie sich doch den gleichen Skript Code. Was es sehr leicht macht gemeinsame Funktionen für alle bereit zu stellen.

Dieses spezielle Design macht den Protokollmonitor extrem schnell, sehr robust gegen Code Fehler (bedenken Sie hier, das es mögliche Fehler im Anwender Skript sind) und vermeidet das unbeabsichtigte Mischen von globalen Variablen (was ein typisches Problem bei vielen Programmiersprachen ist).
Wie aber können wir nun Daten zwischen zwei (oder mehr) Lua Instanzen austauschen, wenn globale Variablen keine Option sind? Oder genauer gesagt: Wie können die im Dialog eingegebenen Werte an den Rest des Templates weiter gegeben werden?

Das `widgets` Modul bietet einen einfachen Mechanismus für den Datenaustausch beliebiger Daten (Zahlen, Strings, Boolesche Werte) zwischen der GUI und den anderen Interpretern. Dieser basiert auf einer anonymen Tabelle im `widgets` Modul, die automatisch in allen Lua Interpretern eingeblendet wird (quasi wie ein gemeinsamer Speicherbereich unterschiedlicher Prozesse).
Sobald Sie unter `widgets` bzw. im `widgets` Namensraum eine Variable deklarieren, kann auf diese von allen anderen Interpretern zugegriffen werden. In unserem Fall:

```
1  function apply()
2      widgets.unit = widgets.GetValue( "unit" )
3      widgets.eos = widgets.GetValue( "eos" )
4  end
```

Wenn wir die Funktion `GetFunctionValue` in gleicher Weise verändern werden sich die in den Telegrammen angezeigten Werte aktualisieren, sobald wir eine andere Einheit wählen und Anwenden klicken.

```
1  function out()
2      ...
3      function GetFunctionValue( number, value )
4          if widgets.unit == "Metric" then
5              local formats = { "%.2fC", "%.2f%%", "%imBar"}
6              return string.format( formats[ number ], value )
7          else
8      ...
9  end
```

Und da alle Benutzer Eingaben in einer internen Tabelle gespeichert sind, ist es einfach, den aktuellen Dialog Inhalt mit einem einzigen Funktionsaufruf zu speichern oder erneut zu laden. Mehr im Abschnitt 20.3.8.

20.3.4 Aktualisieren oder neu laden

Ein Wechsel der Systemeinheiten wirkt sich nur auf die aktuelle Anzeige der Telegramme aus. Ein erneutes Extrahieren der Telegramme (neu Laden der

Aufzeichnung) aus dem Datenstrom ist nicht nötig, da sich die Telegramme als Bytefolge mit definiertem Anfang und Ende nicht ändern. Das bedeutet:
Die `apply` Funktion triggert lediglich ein Neu-zeichnen der sichtbaren Telegramme im Telegramm Fenster. Dies ist das Standard Verhalten und gleichzeitig das Stichwort für das noch nicht berücksichtigte zweite Widget in unserem Dialog. Die EOS Auswahl.
Die vier Einträge im `Choice` Widget sind: LF, CR, LFCR und CRLF (welches wir in unserem Tutorial verwenden). Das vom Anwender ausgewählte EOS muss an die `split` Funktion übergeben werden, da es ein entscheidendes Kriterium zur Extrahierung (Splitten) der Telegramme aus dem Datenstrom ist. Zuvor muss das EOS allerdings in die korrekte Bytefolge umgewandelt werden. Ein LF steht für das Zeichen hex 10, CR für hex 13. In Lua auch geschrieben als `"\n"` und `"\r"`. C(++) Programmieren kennen dies bereits.

```
1  function apply()
2      local t = {
3          ["LF"] = "\n",
4          ["CR"] = "\r",
5          ["LFCR"] = "\n\r",
6          ["CRLF"] = "\r\n"
7      }
8      widgets.unit = widgets.GetValue( "unit" )
9      widgets.eos = t[widgets.GetValue( "eos" )]
10     return "RELOAD"
11 end
```

Die Umwandlung von einem String in einen anderen. z.B. CRLF in `"\r\n"` ist in Lua sehr einfach zu realisieren. Wir legen dazu eine lokale Tabelle mit entsprechenden Key/Value Paaren an. Zeile 9 liefert dann zu jedem ausgewählten EOS die zugehörige binäre Sequenz.

Zeile 10 ist neu! Bislang haben wir in `apply` keine Wert zurück gegeben, was für den Protokollmonitor gleich bedeutend mit dem Standard Verhalten ist. Nämlich lediglich die sichtbaren Telegramme neu auszugeben. Eine Änderung des EOS betrifft aber ALLE Telegramme, nicht nur die im Programmfenster gerade dargestellten. Mit anderen Worten: Der Protokollmonitor muss den gesamten Datenstrom neu laden und in einzelne Telegramme zerlegen! Der Rückgabewert `"RELOAD"` bewirkt genau das was er sagt.

Als letzter Schritt müssen wir jetzt nur noch in der `split` Funktion das ausgewählte EOS berücksichtigen. Ersetzen Sie dazu die Zeile:

```
1  if str:find( "\r\n") then return COMPLETED end
```

mit:

```
1  if str:find( widgets.eos ) then return COMPLETED end
```

Das Skript verhält sich nun wie erwartet. Sie können zwischen 'Metric' und 'Anglo-American' wechseln und einen aus vier EOS Strings auswählen. Dabei gibt es nur einen kleinen Wermutstropfen:
Egal ob Änderung des EOS oder des Einheitensystems - jedes mal lädt der Protokollmonitor die komplette Aufzeichnung neu. Dies kann gerade bei sehr großen Datenmengen ziemlich zeitaufwendig werden.
Besser wäre es das erneute Laden der Daten auf die Fälle zu begrenzen, bei denen sich die zu Grunde liegende Telegramm Struktur auch wirklich geändert

hat - z.B. ein Wechsel des EOS. Und ansonsten lediglich die sichtbaren Telegramme neu auszugeben - was extrem schnell ist.
In unserem Dialog Beispiel wäre es schön, wenn der Anwender beim Klick der Radio Box unmittelbar das Resultat sieht. Der nächste Abschnitt zeigt Ihnen wie das geht.

20.3.5 Einen Aktionsbehandler definieren

Ein Aktionsbehandler oder 'action handler' ist eine Funktion, die automatisch aufgerufen wird, wenn der Anwender ein Dialogelement verändert. Diese auch als 'callback' bezeichneten Funktionen sind dadurch besonders hilfreich um unmittelbar auf Benutzer Eingaben zu reagieren.
Beispiele hierfür sind das Klicken eines Knopfes oder das (De)Aktivieren von anderen Elementen abhängig von der gerade getätigten Interaktion. Oder aber das Aktualisieren des Telegramm Fensters, was uns zu unserem Beispiel zurück bringt.
Mit dem Dialog Framework können Sie beliebige Funktionen als callback definieren. Und das auf eine sehr einfache Art und Weise. Die Definition einer callback Funktion ist wie folgt:

```
function callback_NAME( value )
    -- do something
end
```

Das wichtige Detail hier ist der NAME. Dieser entspricht dem Namen des Elements oder Widgets, für den Sie einen 'action handler' einrichten wollen[2]. Sobald Sie eine callback Funktion für ein bestimmtes Widget im Code eingefügt haben, wird diese bei jeder Interaktion mit eben diesem Widget (geklickt, ausgewählt, modifiziert) unmittelbar ausgeführt.
In unserem Beispiel wollen wir sofort auf einen Wechsel der Einheiten in der Radio Box 'unit' reagieren. Der Name der callback Funktion ist damit vorgegeben als `callback_unit`. Sehen Sie sich dazu folgende Code Zeilen an:

```
function callback_unit( state )
    debug.print( state )
    widgets.unit = state
end
```

Der interne callback Mechanismus übergibt zudem den aktuellen Status, Auswahl oder Inhalt des Widgets als String Parameter. Dieser Datentyp wurde bewusst gewählt um allen möglichen Varianten von Dialog Elementen gerecht zu werden. Denken Sie an Widgets für Texteingaben oder Auswahllisten.
Hier ist es besonders praktisch. Der callback Parameter `state` enthält bereits den Wert 'Metric' oder 'Anglo-American' und wir müssen diesen nur einfach der globalen Variable `widgets.unit` zuweisen - fertig!
Mit jedem Wechsel der 'unit' Radio Box wechselt nun auch sofort die aktuelle Telegramm Anzeige[3].
Die Debug Ausgabe ist nur zu Testzwecken da und soll Ihnen lediglich eine Vorstellung vermitteln, wie Sie den Parameter der Funktion verwenden können. Vergessen Sie aber nie: Er ist immer von Typ Lua String!
Die Umwandlung von einem Datentyp in einen anderen erfolgt in Lua meistens

[2] Und ist ein weiterer Grund, den Namen des Widgets sorgfältig zu wählen!
[3] Das Dialog Framework triggert generell eine Aktualisierung der Anzeige sobald irgendein callback aufgerufen wird.

völlig automatisch und Sie müssen sich nicht weiter darum kümmern. Es gibt aber auch Situationen in denen Sie explizit z.B. einen String in eine Zahl umwandeln müssen. Dies ist u.a. der Fall, wenn Lua nicht weiß (und auch nicht vorhersehen kann) welcher Datentyp im weiteren Verlauf benötigt wird. Die Umwandlung eines Strings in eine Zahl erfolgt per:

```
x = tonumber( string )
```

20.3.6 Dialog Variablen initialisieren

Bislang haben wir uns keine Gedanken über die Initialisierung unserer Dialog Variablen gemacht. Das ändert sich, sobald Sie den aktuellen Protokollmonitor als neues Fenster öffnen (klonen). Drücken Sie dazu einfach
Strg + Umschalt + N.
Das neue Protokollfenster begrüßt Sie mit folgendem Skript Fehler:
`"bad argument #1 to find..."`. Was ist passiert?

Lassen Sie uns kurz Revue passieren wie Lua Ihr Skript interpretiert.
Der allererste Code bzw. Funktion die Lua ausführt, sobald Sie einen neuen Protokollmonitor starten oder ein neues Template auswählen, ist `split`. Diese ist essentiell, weil alle weiteren Aktionen von der korrekten Extrahierung der Telegramme aus dem Datenstrom abhängen. In unserem Beispiel:

```
function split( data, intval, alter, str )
    if #str == 1 then return STARTED end
    if str:find( widgets.eos ) then return COMPLETED end
    return MODIFIED
end
```

In der `split` Funktion verwenden wir die Lua String Methode `find` um nach dem übergebenen EOS String zu suchen. Das Modul `widgets` mit der anonymen Tabelle der Dialog Variablen ist Teil des Interpreters, der `split` ausführt. Aber: Solange der Skript Dialog nicht geöffnet und dessen `apply` Funktion nicht aufgerufen wurde bleibt diese Tabelle leer - kein `widgets.eos` Eintrag! Denn dieser wird erst in der `apply` Funktion angelegt!
Das Resultat ist: `find` wird mit einem `nil` Parameter aufgerufen und verursacht eine entsprechende Fehlermeldung.
Sie können das einfach überprüfen. Öffnen Sie den Dialog und klicken Sie 'Anwenden'. Der Protokollmonitor zeigt anschließend die Telegramme wie erwartet und ohne Fehler!

Um in Zukunft solche Fehlermeldungen zu vermeiden müssen wir die `widgets` Variablen für alle Interpreter deklarieren und initialisieren. Dies erreichen wir, indem wir den Variablen im globalen Namensraum des Skripts, d.h. außerhalb jedweder Funktion, einen Startwert zuweisen (initialisieren). Hier am Beispiel der `widgets.eos` Variable.

```
widgets.eos = "\r\n"

function split( data, intval, alter, str )
    if #str == 1 then return STARTED end
    if str:find( widgets.eos ) then return COMPLETED end
    return MODIFIED
end
```

Damit extrahiert der Protokollmonitor alle Telegramme mit einem endenden CR-LF solange der Anwender nichts anderes eingibt.

`widgets.eos` ist aber nicht die einzige Dialog Variable in unserem Skript. In der `output` Funktion greifen wir auf eine weitere Variable `widgets.unit` zu. Auch diese ist zum Zeitpunkt des ersten Aufrufs nicht initialisiert und damit `nil`. Der Unterschied hier: Lua erlaubt den Vergleich von `nil` Werten. D.h.:

```
if widgets.unit == "Metric" then
```

liefert false und der Anweisungsblock für die 'Anglo-American' Formatierung kommt zur Ausführung.
Es ist deshalb gute Praxis generell ALLE Dialog Variablen am Anfang des Skripts zu initialisieren.

```
widgets.eos = "\r\n"
widgets.unit = "Metric"
```

Initialisieren Sie alle Dialog Variablen

Um Fehler durch nicht initialisierte Dialog Variablen zu vermeiden sollten Sie diese IMMER zu Beginn des Skript mit gültigen Werten initialisieren!

Sie werden feststellen, das Ihr Dialog nun immer mit den gleichen Einstellungen startet, egal welche davon gerade in der Telegramm Anzeige angewendet werden. Das ist nachvollziehbar, da der Dialog noch nirgends mit den aktuellen Einstellungen vorbesetzt wird.
Wenn Sie den Setup Knopf klicken, wird ein neuer Lua Interpreter gestartet und führt die `dialog` Funktion aus. Er endet mit Schließen des Dialogs.
Es obliegt Ihrer Verantwortung als Programmierer, die Dialog Elemente mit den aktuell angewendeten Werten zu initiieren. Wie der nächste Abschnitt zeigen wird, ist das zum Glück recht einfach.

20.3.7 Dialog Einstellungen

Mit Dialog Einstellungen sind alle veränderbaren Parameter gemeint, die der Anwender in einem Dialog beeinflussen kann. Wie diese an den Rest des Skripts zur Protokoll Verarbeitung weitergegeben werden haben wir bereits herausgearbeitet. Jetzt geht es um den umgekehrten Weg. Die Dialog Elemente mit Ihrem letzten Status voreinstellen.
Auch wenn der ausführende Dialog Interpreter nach Schließen desselben aufhört zu existieren, sind die aktuellen Dialog Einstellungen dennoch in Form der globalen Variablen `widgets.eos` und `widgets.unit` weiter verfügbar.
Um den Dialog in seinen letzten Zustand zu versetzen, müssen wir seine einzelnen Elemente mit diesen Werten beim Start (in der Funktion `dialog`) initialisieren.

```
1 function dialog()
2     widgets.Label{ text="Individual tutorial protocol settings",
3                    col=1, row=1, span=2 }
4     widgets.RadioBox{ name="unit",
5                       label="Unit",
6                       col=1, row=2, fill=true, span=2,
7                       choices={"Metric","Anglo-American"} }
```

```
 8      widgets.Label{ text="Telegram EOS",
 9                     col=1, row=3 }
10      widgets.Choice{ name="eos",
11                      col=2, row=3, fill=true,
12                      choices={"LF", "CR", "LFCR", "CRLF" } }
13      -- initialize dialog elements with their last settings
14      widgets.SetValue( "unit", widgets.unit )
15      local t = {
16          ["\n"] = "LF",
17          ["\r"] = "CR",
18          ["\n\r"] = "LFCR",
19          ["\r\n"] = "CRLF",
20      }
21      widgets.SetValue( "eos", t[widgets.eos] )
22  end
```

Beachten Sie! Die Variable `widgets.eos` enthält den EOS String als binäre Sequenz. Wir müssen diesen zuerst in sein entsprechendes Pendant CR, LF, CRLF oder LFCR 'zurück' verwandeln bevor wir das Widget damit initialisieren können. Auch hier ist der Ansatz der gleiche: Eine Lua Tabelle mit Key/Value Paaren, nur diesmal mit vertauschten Einträgen (ab Zeile 15).

Wenn Sie nach Speichern des Skripts den Dialog ausprobieren, startet dieser aber trotzdem immer mit den gleichen Einstellungen ('Metric' und CRLF als EOS). Das ist allerdings auch nicht verwunderlich.
Erinnern Sie sich? Wir haben exakt diese Werte zur Initialisierung der globalen Variablen `widgets.eos` und `widgets.unit` verwendet.
Bei der Ausführung des Dialogs führt der Interpreter auch allen anderen 'erreichbaren Code' in dem Skript aus. Unter anderem die Zuweisung der `widgets` Dialog Variablen bei deren Initialisierung. Dies passiert auch dann, wenn die Variablen durch einen vorherigen Dialog Aufruf bereits existieren.
Um das zu verhindern dürfen diese Variablen nur dann initialisiert werden, wenn Sie noch nicht angelegt wurden.

```
if not widgets.eos then
    widgets.eos = "\r\n"
end
if not widgets.unit then
    widgets.unit = "Metric"
end
```

Mit diesen Änderungen arbeitet der Dialog nun exakt wie er soll. Sie können beliebige Änderungen im Dialog vornehmen, die dann vom Protokollmonitor entsprechend angewendet werden. Sobald Sie den Dialog erneut öffnen, spiegelt dieser die aktuellen Einstellungen wider.

20.3.8 Dialog Einstellungen speichern

Die vom Anwender gemachten Einstellungen werden bislang nur innerhalb des Protokollmonitor gespeichert. Sobald Sie das Programmfenster schließen sind alle Einstellungen weg.
Dies ist gerade bei komplexeren Dialogen ärgerlich. Das Framework enthält deshalb zwei Funktionen, um den Inhalt alle `widgets` Variablen in einer Datei zu speichern und aus dieser erneut zu laden.
Der Dateiname wird aus dem aktuellen Namen der Aufzeichnung und der Dateiendung `msbini` gebildet. Die Datei selbst wird im gleichen Ordner wie das

Skript abgelegt. Sie brauchen sich deshalb über den Dateinamen und Dateipfad keine Gedanken zu machen.
Das einzige was Sie tun müssen, ist folgende Zeile am Ende der `apply` und `callback_unit` Funktion einzufügen[4].

```
widgets.SaveSettings()
```

Um die gespeicherten Einstellungen zu restaurieren rufen Sie - ganz wichtig - nach der Initialisierung der `widgets` Variablen das Gegenstück auf:

```
if not widgets.eos then
    widgets.eos = "\r\n"
end
if not widgets.unit then
    widgets.unit = "Metric"
end
widgets.LoadSettings()
```

Sie sollten immer zuerst die `widgets` Variablen initialisieren bevor Sie die Funktion `widgets.LoadSettings()` aufrufen. Dies garantiert, das die Variablen auch dann korrekt gesetzt sind, wenn die Datei mit den gespeicherten Werten noch gar nicht existiert[5]

msbini Dateiname

Vielleicht wundern Sie sich gerade, warum der Dateiname zur Speicherung der Einstellungen vom Namen der Aufzeichnung und nicht dem Template Namen abgeleitet wird. Nun, die Antwort ist einfach:
Stellen Sie sich vor, die arbeiten gleichzeitig mit zwei Aufzeichnungen und identischen Protokoll Template. Die erste Aufnahme verwendet allerdings ein CR als EOS, die zweite ein LF. Würden die per Dialog eingestellten Protokoll Parameter in einer vom Template Namen abgeleiteten Datei gespeichert werden, müssten Sie das EOS beim Wechsel der Aufzeichnung immer wieder neu anpassen. So werden die Einstellungen 'zugehörig' zur Aufnahme gespeichert, was auch richtig ist.

20.4 Fortgeschrittene Positionierung und Interaktion

Alle Widgets in einem Dialog sind in einem unsichtbaren Raster ausgerichtet. Sie können die Position (definiert als Raster Spalte und Zeile) direkt angeben (wie wir es in unserem Beispiel gemacht haben) oder aber als Resultat einer Berechnung. Dies gilt auch für die übrigen Widget Parameter.
Nehmen wir ein Feld von 3x3 Knöpfen. Ihr erster Ansatz wäre vermutlich ähnlich dem folgenden:

```
1  function dialog()
2      widgets.Button{ name="B1/1", label="B1/1", col=1, row=1 }
3      widgets.Button{ name="B2/1", label="B2/1", col=2, row=1 }
4      widgets.Button{ name="B3/1", label="B3/1", col=3, row=1 }
5      widgets.Button{ name="B1/2", label="B1/2", col=1, row=2 }
6      ...
7  end
```

Aber anstelle der neun Zeilen `widgets.Button{...}` Code können wir die Buttons auch in einer Schleife erzeugen.

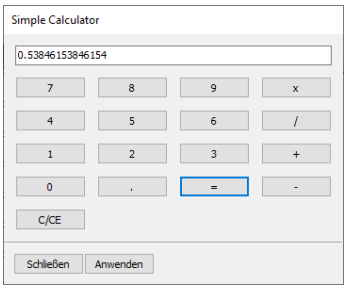
[4] Im Grunde überall da, wo Sie die Dialog Einstellungen anwenden
[5] Was der Fall ist, solange der Einstelldialog nicht mindestens einmal angewendet wurde.

```
1  function dialog()
2      for row=1,3 do
3          for col=1,3 do
4              local s = "B"..col.."/"..row
5              widgets.Button{ name=s, label=s, row=row, col=col }
6          end
7      end
8  end
```

Der interessante Teil dieses kleinen Code Schnipsel ist Zeile 4. Bei Operationen mit verschiedenen Datentypen erfolgt die Typ Umwandlung durch Lua mehr oder weniger automatisch. Strings und Zahlen können damit sehr einfach durch den Verknüpfung Operator `..` zu neuen Strings kombiniert werden.
In Zeile 4 verwenden wir die Spalte und Zeile um einen eindeutigen Namen `s` für den Button zu erzeugen. Jeder Name beginnt mit einem 'B' (String), der Spalte (Lua wandelt den Zahlenwert automatisch in einen String um), gefolgt von einem '/' und der Zeile.
Das Resultat wird dann in Zeile 5 als Name und Label Parameter dem Button Widget übergeben.

Wie Sie diese Technik zum Bau von gut aussehenden Dialogen verwenden können, zeigt das Beispiel `Tutorial-Calculator`. Dieses Skript enthält einen kleinen aber voll funktionsfähigen Taschenrechner. Es ist aber kein Protokoll Template! Sein ausschließlicher Zweck ist, die erweiterten Möglichkeiten des Dialog Framework zu demonstrieren.
Hier ist der Code für den Dialog.

```
1  function dialog()
2      widgets.TextCtrl{ name="display", col=1, row=1, span=4, fill=true,
3                        datatype=DEC_NUMBER }
4      local labels={ "7", "8", "9", "x",
5                     "4", "5", "6", "/",
6                     "1", "2", "3", "+",
7                     "0", ".", "=", "-" }
8      local i = 1
9      for y=2,5 do
10         for x=1,4 do
11             widgets.Button{ name=labels[i], label=labels[i], col=x, row=
                   y }
12             i = i + 1
13         end
14     end
15     widgets.Button{ name="Clear", label="C/CE", col=1, row=6 }
16 end
```



Simpler Taschenrechner

In dem Taschenrechner Beispiel kombinieren wir fest vorgegebene Positionen (Anzeige in Zeile 2 und 3 sowie Löschtaste C/CE in Zeile 15) mit einem per Schleife initialisierten 4x4 Raster welches die Zahlen- und Operatoren enthält (Zeile 9..14).
Das Anzeige `TextCtrl` nimmt die komplette erste Reihe des Rasters ein (`span=4`). Die Schleife startet deshalb mit Reihe 2 und iteriert bis zur Raster Reihe 5.
Jede Reihe besteht aus vier Spalten. Die innere Schleife (Zeile 10) zählt deshalb von 1 bis 4.
Alles in allem besteht das komplette Taschenrechner Interface gerade mal aus 16 Zeilen Code.

20.4.1 Erweitere callbacks

Mit Ausnahme des `TextCtrl` Widgets (Display) besteht die GUI des Taschenrechners ausschließlich aus Buttons. Sobald der Anwender einen Zahlenknopf klickt, wird die entsprechende Zahl an den Wert im Display ein- oder angefügt. Beim Klick auf eine Arithmetik Taste wird die Operation gespeichert und nach Eingabe des nächsten Wertes mit der Taste (=) angewendet.
Zuletzt der (C/CE). Dieser löscht bei Betätigung alle Eingaben und damit das Display.

Wir haben bereits gelernt, dass wir jedem Widget eine individuelle callback Funktion zuweisen können. Dies ist auch hier der richtige Ansatz. Allerdings werden 17 callbacks (für 17 Tasten) im Code schnell ziemlich unübersichtlich. Insbesondere da einige Tasten im Grunde immer das gleiche tun. Die Zahlentasten z.B. fügen immer ihre Nummer an der Wert im Display.
Ein einzelnes callback für alle Nummern Tasten wäre hier mehr wünschenswert. Jedes mal, wenn der Anwender eine beliebige Taste klickt soll eine ganz bestimmte Funktion ausgeführt werden. Die Aktion soll dabei von der gedrückten Taste abhängen dürfen.

Der callback Mechanismus des Dialog Framework verhält sich exakt so.
Wenn ein Button geklickt wird (oder generell eine Interaktion mit einem Widget stattfindet), prüft der Lua Dialog Interpreter zunächst, ob ein individuelles callback `callback_NAME` vorliegt und führt den dortigen Code aus.
Im Falle der Buttons wird anschließend nach einer weiteren callback Funktion `callback_all_buttons` gesucht und - falls vorhanden - auch diese noch ausgeführt. Die Funktion ist wie folgt definiert:

```
1 function callback_all_buttons( name )
2 end
```

Als Parameter wird der Name des angeklickten Buttons übergeben.
Das macht es einfach auf jede Taste entsprechend zu reagieren. Öffnen Sie dazu das `Tutorial-Calculator` Template im Editor. Der Code ist gut dokumentiert.

20.5 Bereits existierende Widgets ändern

Bislang haben wir noch nicht thematisiert, wie man ein bereits vorhandenes Widget Element im Nachhinein modifiziert.
Mit der widgets Funktion `SetValue(name,value)` können wir zwar den Wert (oder Inhalt), den ein Widget repräsentiert, voreinstellen oder aktualisieren. Was aber, wenn Sie - zum Beispiel - den Eingabebereich eines SpinCtrl ändern müssen?[6]
Ein Aufruf per `Setvalue(name,value)` beeinflusst nur den Inhalt, nicht aber den Bereich definiert durch `min` und `max`. Um es ein wenig klarer zu machen, hier ein typischer SpinCtrl Code in einem Dialog:

```
widgets.SpinCtrl{ name="char", row=2, col=2, min=0, max=255, value=0 }
```

[6]Wir haben ein SpinCtrl im allerersten Bespiel zur Erklärung des Dialog Rasters verwendet - erinnern Sie sich?

Das angelegte SpinCtrl sei gedacht für die Eingabe eines beliebigen Zeichens per 8-Bit Wert im Bereich 0 bis 255. Durch Interaktion mit einem anderen Dialogelement muss dieses nun aber auf die Eingabe druckbarer Zeichen reduziert werden. D.h. das SpinCtrl darf nun nur noch Zeichen im Bereich 30 (Leerzeichen) bis 127 (letztes gültiges Zeichen in der normalen ASCII Tabelle) zurückgeben.
Es wäre ziemlich aufwendig (und wenig benutzerfreundlich), für jeden Widget Typ eine eigene Update Funktion zu definieren. Ein weitaus besserer Ansatz ist das entsprechende Widget einfach durch ein neues des gleichen Typs aber mit unterschiedlichen Parametern zu ersetzen. In unserem Fall die min und max Parameter.
Da alle Dialog Elemente (Widgets) in einem Raster organisiert sind, bedeutet dies nichts anderes, als ein Element an einer bestimmten Position durch ein anderes oder geändertes zu überschreiben:

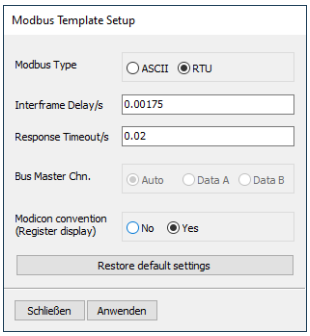
```
1  local col, row = widgets.GetPosition( "char" );
2  widgets.SpinCtrl{ name="char", row=row, col=col,
3                    min=30, max=127, value=0 }
```

Sie können - natürlich - die Zeilen- und Spaltenwerte direkt kodieren. Das ist aber ziemlich fehleranfällig. Besser ist es, die Position des zu ersetzenden Elements per widgets Funktion `GetPosition(name)` zu erfragen, wie in Zeile 1 gezeigt. In Zeile 2 überschrieben wir das an dieser Stelle befindliche SpinCtrl dann einfach mit einer neuen SpinCtrl Instanz, allerdings jetzt mit geändertem Bereich 30...127.
Wir können unser kleines Beispiel noch ein wenig erweitern. Z.B. indem wir den ursprünglichen Inhalt/Wert nach Überschreiben des Elements restaurieren. Natürlich nur, wenn er noch immer im neuen Bereich liegt:

```
1  local col, row = widgets.GetPosition( "char" );
2  local val = widgets.GetValue( "char" )
3  widgets.SpinCtrl{ name="char", row=row, col=col,
4                    min=30, max=127, value=val }
```

20.6 Weiterführende Beispiele

Dieses Tutorial zeigte Ihnen lediglich ein rein theoretisches Beispiel. Wenn Sie an einem echten Protokoll Template interessiert sind und was Sie dort mit dem Dialog Framework erreichen können, schauen Sie sich das Modbus Projekt an. Oder wählen Sie einfach Modbus im Protokollmonitor und klicken Sie anschließend Setup.
Danach können Sie sich den Code im Editor ansehen, er ist gut dokumentiert.



Modbus dialog

20.7 Unterstützte Dialog Elemente und Widgets

Alle hier aufgeführten GUI Elemente sind Teil der Analyser Lua Software und funktionieren nicht in reinen Lua Umgebungen.

Jedes Widget unterstützt die folgende Liste von Parametern. Obgleich alle Parameter im Lua Sinne optional sind (der Lua Interpreter wird bei fehlenden Parametern keinen Fehler melden), müssen mache dennoch angegeben werden, damit das Programm korrekt funktioniert.

Zum besseren Verständnis haben wir die Parameter deshalb entsprechend gekennzeichnet:

⊗ Ein zwingender Parameter
⊙ Ein optionaler Parameter

20.7.1 Benannte Parameter

Ein paar Worte bezüglich der Parameter Übergabe. Sie haben sicher bemerkt, dass alle Widget Parameter nachstehender Konvention folgen:

```
PARAMETERNAME=VALUE
```

Das ist nicht Lua typisch. Benannte Parameter haben aber einen entscheidenden Vorteil. Sie sind besser verständlich, der Code besser lesbar und damit weniger Fehler anfällig. Ein Beispiel:

```
1 widgets.Button( "MyButton", "Press", 1, 3, true )
```

Ohne Blick in das Handbuch ist diese Zeile nur schwer zu verstehen. Was ist der Widget Name, was das Button Label? Ist 1 die Spalte oder Reihe und was bedeutet true?
Im Gegensatz dazu folgender Code mit benannten Parametern:

```
1 widgets.Button{ name="MyButton",label="Press",col=1,row=3,fill=true}
```

Die Bedeutung der Parameter ist offensichtlich.
Benannte Parameter werden IMMER von geschweiften Klammern `{...}` eingerahmt, da sie intern als Lua Tabelle übergeben werden. Korrekter Weise müssten Sie `({...})` schreiben. Die äußeren Klammern sind aber für Lua optional und Sie können Sie bedenkenlos weg lassen.

20.7.2 Allgemeine Widget Parameter

Alle Widgets verstehen zumindest nachfolgende benannte Parameter. Benannt heißt: Sie übergeben diese als Schlüssel/Werte Paare wie zuvor erläutert:

```
name="MyName"
col=1
fill=true
```

Die Parameter nach Wichtigkeit geordnet:

⊗ name : (Fast) jedes Widget benötigt einen individuellen Name um später auf es zugreifen zu können. Z.B. um den Wert oder eine Auswahl zu erfragen.
⊗ col : Spezifiziert die Spalte in der das Widget platziert werden soll. Die Spalten werden ab 1 gezählt. Default Wert ist 1.
⊗ row : Gibt den Nummer der Reihe an, in welcher das Widget platziert werden soll. Auch die Reihe wird ab 1 gezählt. Default ist 1.
⊙ datatype : Insbesondere das Widget zur Texteingabe kann zur Eingabe verschiedenster Datentypen verwendet werden. Z.B. dezimale oder hexadezimale Zahlenwerte, aber auch normaler (ASCII) Text oder HEX Strings. Mit dem `datatype` Parameter können Sie den Bereich der erlaubten Zeichen festlegen und wie der eingegebene Wert bei einer Abfrage behandelt werden soll. Gültige Parameter sind:

`BIN_NUMBER, DEC_NUMBER, FLOAT_NUMBER, HEX_NUMBER, ASCII_STRING, HEX_STRING`

- datalen : Definiert die Anzahl der maximal erlaubten Zeichen. Nur diese werden bei Abfrage später berücksichtigt.
- fill : Setzen Sie diesen Wert auf true, wenn das Widget den komplett verfügbaren Platz im Raster ausfüllen soll.
- span : Mit den 'span' Parameter können Sie ein Widget über die angegebene Anzahl von Spalten ausdehnen.

Neben diesen allgemeinen Parametern erlauben manche Widgets noch weitere individuelle Argumente. Diese werden in den jeweiligen Widget Abschnitten im Detail behandelt.

20.7.3 Button

Ein simpler Knopf mit Text Label.

```
widgets.Button{ name=STRING, label=STRING, col=NUM, row=NUM,
                fill=BOOL, span=NUM }
```

- name : der Button Name als Lua String
- col : die Rasterspalte als Integer
- row : die Rasterzeile als Integer
- label : das Button Label
- fill : Raster vollständig ausfüllen, true oder false
- span : die Anzahl der zu verwendenden Spalten als Integer

Example

```
1  function dialog()
2      -- a button on top left2
3      widgets.Button{ name="MyButton", label="Press", col=1, row=1 }
4  end

5  -- this callback is executed every time the button is clicked
6  function callback_MyButton()
7     -- do something...
8  end
```

20.7.4 CheckBox

Eine Checkbox (Kontrollkästchen) ist eine beschriftetet Box (Kästchen) welches entweder wahr (Häkchen gesetzt) oder falsch (kein Häkchen) sein kann.

```
widgets.CheckBox{ name=STRING, label=STRING, col=NUM, row=NUM,
                  fill=BOOL, span=NUM }
```

- name : der Checkbox Name als Lua String.
- col : die Rasterspalte als Integer integer
- row : die Rasterzeile als Integer

- label : eine optionale Beschriftung auf der rechten Seite der Checkbox
- fill : Raster vollständig ausfüllen, true oder false
- span : die Anzahl der zu verwendeten Spalten als Integer
- value : der vordefinierte (gesetzte) Zustand der CheckBox, true oder false

Example

```
function dialog()
    -- a button we want to stretch or shrink
    widgets.Button{ name="MyButton", label="Press", col=1, row=1 }
    -- a checkbox to toggle a fill parameter
    widgets.CheckBox{ name="MyCheckBox", label="Stretch the button",
      col=1, row=2, value=false }
end

-- this callback is executed every time the checkbox is clicked
function callback_MyCheckBox( selection )
    -- query the position of the button widget
    row,col = widgets.GetPosition( "MyButton" )
    -- recreate the button with the new fill parameter
    widgets.Button{ name="MyButton", label="Press",
                    col=col, row=row, fill=(selection=="true") }
end
```

20.7.5 Choice

Das Choice (Auswahl) Widget präsentiert Ihnen eine Liste von Strings aus der Sie einen auswählen können. Nur der aktuell ausgewählte wird anschließend angezeigt.

```
widgets.Choices{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
                 choices={STRING1, STRING2 [,...]}}
```

- name : der Name des Widgets als Lua String.
- col : die Rasterspalte als Integer integer
- row : die Rasterzeile als Integer
- choices : eine Lua Tabelle der Auswahlmöglichkeiten mit zumindest einem String Eintrag
- fill : Raster vollständig ausfüllen, true oder false
- span : die Anzahl der zu verwendeten Spalten als Integer
- value : die voreingestellte Auswahl übergeben als Lua String

Example

```
function dialog()
    -- a button we want to stretch or shrink
    widgets.Button{ name="MyButton", label="Press", col=1, row=1 }
    -- a checkbox to toggle a fill parameter
    widgets.Choice{ name="MyChoice", col=1, row=2,
                    choices={ "Shrink button", "Stretch button"},
                    value = "Stretch button" }

end

-- this callback is executed every time a choice is made
function callback_MyChoice( selection )
    -- query the position of the button widget
    row,col = widgets.GetPosition( "MyButton" )
    -- recreate the button with the new fill parameter
    widgets.Button{ name="MyButton", label="Press",
                    col=col, row=row,
                    fill=(selection=="Stretch button") }
end
```

20.7.6 Label

Ein Label Widget wird immer dann verwendet, wenn ein statischer (nicht interaktiver) Text im Dialog angezeigt werden soll. Im allgemeinen ein erklärender Text oder ein Label für ein anderes Widget Element.

```
widgets.Label{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
               text=STRING}
```

- ⊗ name : der Name des Widgets als Lua String.
- ⊗ col : die Rasterspalte als Integer integer
- ⊗ row : die Rasterzeile als Integer
- ⊗ text : der anzuzeigende Text als Lua String
- ⊙ fill : Raster vollständig ausfüllen, true oder false
- ⊙ span : die Anzahl der zu verwendeten Spalten als Integer

Example

```
function dialog()
    -- a explaining text for the button
    widgets.Label{ name="MyLabel", text="You can click me",
                   col=1, row=1 }
    -- the button
    widgets.Button{ name="MyButton", label="Disable me",
                    col=1, row=2 }
end

-- this callback is executed every time the button is clicked
function callback_MyButton()
    -- disable the button
    widgets.Enable( ''MyButton", false )
    -- and adapt the label text
    local col, row = widgets.GetPosition( ''MyLabel" )
    widgets.Label{ name="MyLabel", text="You cannot click me anymore",
                   col=col, row=row }
end
```

20.7.7 Line

Eine einfache Linie. Meistens verwendet um unterschiedliche Dialog Elemente visuell zu trennen. Eine Linie füllt immer die komplette Breite des ihr zugewiesenen Rasters. Mit dem `span` Parameter können Sie die Linie über mehrere Spalten strecken.

```
widgets.Line{ name=STRING, col=NUM, row=NUM, span=NUM }
```

- ⊗ col : die Rasterspalte als Integer integer
- ⊗ row : die Rasterzeile als Integer
- ⊙ span : die Anzahl der zu verwendeten Spalten als Integer
- ⊙ name : der Name der Linie. Dieser kann meistens entfallen, es sei denn Sie wollen die Linie später z.B. ein- oder ausblenden

Example

```
function dialog()
    widgets.Label{ name="MyLabel", text="A label", col=1, row=1 }
    widgets.Button{ name="Button1", label="Press me", col=2, row=1 }
    widgets.Button{ name="Button2", label="Or me", col=3, row=1 }
    -- draw a line over all columns (span=3)
    widgets.Line{ span=3, col=1, row=2 }
end
```

20.7.8 RadioBox

Die RadioBox hat Ihren Namen von den Stationstasten alter Radioempfänger. Sie dient zur Auswahl exakt einer von mehreren sich gegenseitig ausschließenden Möglichkeiten. Die Radiobox kann vertikal (auswählbare Einträge untereinander) oder horizontal (auswählbare Einträge nebeneinander) ausgerichtet werden.

```
widgets.RadioBox{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
                  choices={STRING1, STRING2 [,...]}}
```

- ⊗ name : der Name des Widgets als Lua String.
- ⊗ col : die Rasterspalte als Integer integer
- ⊗ row : die Rasterzeile als Integer
- ⊗ choices : eine Lua Tabelle der Auswahlmöglichkeiten mit zumindest einem String Eintrag
- ⊙ fill : Raster vollständig ausfüllen, true oder false
- ⊙ span : die Anzahl der zu verwendeten Spalten als Integer
- ⊙ orientation : Die RadioBox Orientierung, entweder 'vertical' (Default) oder 'horizontal'.
- ⊙ value : die voreingestellte Auswahl der RadioBox als Lua String

Example

```
function dialog()
    widgets.RadioBox{ name="MyRadioBox", col=1, row=1,
                      choices={ "vertical", "horizontal" },
                      value="horizontal" }
end

-- each click changes the orientation of MyRadioBox
function callback_MyRadioBox( selection )
    widgets.RadioBox{ name="MyRadioBox", col=1, row=1,
                      choices={ "vertical", "horizontal" },
                      orientation=selection,
                      value=selection }
end
```

20.7.9 Spacer

Das Spacer Widget ist einfach ein leeres Element. Es ist besonders nützlich, wenn Sie ein bereits vorhandenes Element entfernen - oder einfach ausgedrückt - mit 'Nichts' überschreiben wollen.

```
widgets.Spacer{ name=STRING, col=NUM, row=NUM, span=NUM }}
```

- ⊗ name : der Name des Widgets als Lua String.
- ⊗ col : die Rasterspalte als Integer integer
- ⊗ row : die Rasterzeile als Integer
- ⊙ span : die Anzahl der zu verwendeten Spalten als Integer

Example

```
function dialog()
    widgets.Button{ name="MyButton", col=1, row=1,
                    label="Click me", fill=true }
    widgets.RadioBox{ name="MyRadioBox", col=1, row=2,
                      choices={ "Show button", "Hide button" } }
end

-- Show or hide the button according to the RadioBox selection
function callback_MyRadioBox( selection )
    if selection == "Show button" then
        widgets.Button{ name="MyButton", col=1, row=1,
                        label="Click me", fill=true }
    else
        widgets.Spacer{ col=1, row=1 }
    end
end
```

20.7.10 SpinCtrl

Das SpinCtrl dient zur Eingabe eines ganzzahligen Wertes in einem definierten Minimum/Maximum Bereich. Es kombiniert eine Texteingabe mit zwei Inkrement/Dekrement Knöpfen. Letztere dienen zur schrittweisen Erhöhung bzw. Erniedrigung des angezeigten Wertes. Der Wert wird dabei automatisch korrigiert wenn das Minimum oder Maximum unter/überschritten wird.

```
widgets.SpinCtrl{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
                  min=NUM, max=NUM, value=NUM}}
```

- ⊗ name : der Name des Widgets als Lua String.
- ⊗ col : die Rasterspalte als Integer integer
- ⊗ row : die Rasterzeile als Integer
- ⊙ fill : Raster vollständig ausfüllen, true oder false
- ⊙ min : der Minimum Wert, Default ist 1
- ⊙ max : the Maximum Wert, Default ist 100
- ⊙ value : der anfängliche Wert, Default ist der Minimum Wert
- ⊙ span : die Anzahl der zu verwendeten Spalten als Integer

Example

```
1  function dialog()
2      widgets.Label{ name="MyLabel", text="Valid numbers are 1...100",
3                     col=1, row=1 }
4      widgets.SpinCtrl{ name="MySpinCtrl", min=1, max=100,
5                        col=1, row=2, value=10 }
6  end

7  -- always called when the value in the SpinCtrl was changed
8  function callback_MySpinCtrl( value )
9      local num = tonumber( value )
10     if num >= 100 then
11         widgets.Label{ name="MyLabel", text="Maximum number reached",
12                        col=1, row=1 }
13     elseif num <= 1 then
14         widgets.Label{ name="MyLabel", text="Minimum number reached",
15                        col=1, row=1 }
16     else
17         widgets.Label{ name="MyLabel", text="Valid numbers are
               1...100",
18                        col=1, row=1 }
19     end
20 end
```

20.7.11 Table

Das Table Widget ist ein Eingabeelement organisiert als Tabelle. Die Anzahl der Spalten und Zeilen ist frei definierbar. Jede Tabellenzelle dient dabei als Texteingabe für beliebige Strings oder Zahlen. Sie können jede Zelle mit einem bestimmten Wert voreinstellen oder eine Lua `table` übergeben.
Eine solche Darstellung ist besonders für Bussysteme interessant, bei denen die Teilnehmer ihre Daten in Register Tabellen strukturieren - wie z.B. Modbus.

```
widgets.Table{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
               cols=NUM, rows=NUM, preset=STRING,
               choices={STRING1, STRING2 [,...]} }
```

- ⊗ name : der Name des Widgets als Lua String.
- ⊗ col : die Rasterspalte als Integer integer

- ⊗ row : die Rasterzeile als Integer
- ⊙ fill : Raster vollständig ausfüllen, true oder false
- ⊙ span : die Anzahl der zu verwendeten Spalten als Integer
- ⊙ cols : die Anzahl der Spalten, Default ist 1
- ⊙ rows : die Anzahl der Zeilen, Default ist 1
- ⊙ preset : Der Anfangswert für jede Zelle, voreingestellt ist ein leerer String
- ⊙ content : ein Lua Array oder table mit einer bestimmten Anzahl von Strings oder Zahlenwerten. Die Zuweisung startet mit der ersten Zelle und dem ersten Eintrag des übergebene Arrays und stoppt entweder mit der letzten Zelle der Tabelle oder dem letzten Array Eintrag.

Example

```
1  function dialog()
2      -- the number of columns
3      local tcols = 3
4       -- the number of rows
5      local trows = 20
6      -- an empty table holding the default values
7      local tvalues = {}
8
9      -- fill the table with incrementing numbers starting with 1
10     for i=1,tcols*trows do
11         tvalues[i ] = i
12     end

13     -- special mark of the first and last table entry
14     tvalues[ 1 ] = "FIRST"
15     tvalues[ #tvalues ] = "LAST"
16     -- create a table widget and pass the tvalues table as initial
           values
17     -- to initiate all table cells
18     widgets.Table{ name="table", col=1, row=1, cols=tcols, rows=trows,
19                    preset="FFFF", content=tvalues }
20 end
```

20.7.12 TextCtrl

Dialoge benötigen oft ein Feld zur Texteingabe. Dabei sind die eingegebenen Zeichen je nach Verwendung entsprechend zu filtern um ungültige Eingaben zu vermeiden. Beispielsweise darf ein TextCtrl zur Zahleneingabe nur gültige Ziffern mit und ohne Dezimalpunkt erlauben. Eine hexadezimale Adresse nur die Zeichen 0-9 und A-F und binäre Eingaben nur aus '0' und '1' bestehen.
Das TextCtrl ist hier das Mittel der Wahl. Und: Mit dem TextCtrl können Sie nicht nur den Typ der Eingabe festlegen sondern auch die erlaubte Länge. Z.B. bei der Eingabe eines hexadezimalen 16 Bit Wertes max. 4 Stellen.

```
widgets.TextCtrl{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
                  datatype=TYPE, datalen=NUM, value=STRING }
```

- ⊗ name : der Name des Widgets als Lua String.
- ⊗ col : die Rasterspalte als Integer integer

- ⊗ row : die Rasterzeile als Integer
- ⊙ fill : Raster vollständig ausfüllen, true oder false
- ⊙ span : die Anzahl der zu verwendeten Spalten als Integer
- ⊙ datatype : Spezifiziert die Art der einzugebenden Daten. Das TextCtrl unterstützt die Zahlentypen BIN_NUMBER (binär), DEC_NUMBER (dezimal), FLOAT_NUMBER (Eingabe von Zahlen mit Dezimalpunkt) und HEX_NUMBER (hexadezimal). Zudem ASCII_STRING (normaler Text) und HEX_STRING (eine Sequenz aus Hex Zeichen). Das Eingabe/Tasten-Filter wird dabei automatisch gesetzt. Default ist ASCII_STRING.
- ⊙ datalen : Gibt die gültige Länge der eingegebenen Daten an. Z.B. wie viele der eingegebenen Zeichen sollen bei der Abfrage berücksichtigt werden.
- ⊙ value : der Startwert, Standard ist ein leerer String

Example

```
1  function dialog()
2      widgets.RadioBox{ name="MyRadioBox", col=1, row=1, label="Mode",
3                        orientation="horizontal",
4                    choices={ "ASCII", "HEX", "DEC", "BIN" } }
5      widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
6                        datatype=ASCII_STRING, fill=true }
7  end
8
9  function callback_MyRadioBox( mode)
10     local filter = ASCII_STRING
11     if mode== "DEC" then filter = DEC_NUMBER
12     elseif mode== "HEX" then filter = HEX_NUMBER
13     elseif mode== "BIN" then filter = BIN_NUMBER
14     end
15     widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
16                       datatype=filter, fill=true }
17 end
```

20.8 Widgets Funktionen

Einige der hier beschriebenen Funktionen haben Sie bereits kennengelernt als wir die im Dialog eingegebenen Daten abgefragt oder die aktuellen Einstellungen wiederhergestellt haben. Alle sind Teil des `widgets` Moduls und benötigen immer einen Widget Namen.
Da die Anzahl der Parameter hier überschaubar ist (nämlich nur 2), werden die Parameter direkt übergeben (zwischen zwei runden Klammern `(...)`) und nicht wie üblich als NAME=WERT Paar. Einzige Ausnahme hiervon ist die Funktion `SetDialogSize`.
Alle Funktion in alphabetischer Reihenfolge:

20.8.1 Clear

Ein Aufruf von `widgets.Clear()` entfernt (löscht) alle bislang existierenden Widget Elemente in dem Dialog. Diese Funktion wird insbesondere dann nützlich, wenn Sie einen Dialog mit mehreren 'Dialog Seiten' realisieren wollen. Da die aktuelle Dialog Implementierung keine 'verschachtelten' Dialog Elemente

erlaubt (also Elemente mit unterschiedlichen Besitzern - sprich Dialog Seiten), müssen Sie alle Elemente einer nicht sichtbaren Dialogseite entfernen bzw. per Spacer überschreiben bevor Sie die der sichtbaren Seite hinzufügen - ein aufwendiges Unterfangen!
Die `Clear()` Funktion liefert Ihnen den ursprünglichen 'leeren' Zustand - eine leere Dialogseite, die Sie neu mit Widget Elementen füllen können. Sobald Sie auf eine andere Dialogseite umschalten, machen Sie dort einfach das selbe. Erst alle Elemente per `Clear()` löschen, anschließend die Elemente hinzufügen, die die sichtbare Dialogseite ausmacht.
Das Modbus Template ist hierfür ein gutes Beispiel. Es unterteilt den Dialog in einen generellen Modbus Protokoll Einstelldialog und eine zweite Seite zur speziellen Modbus Telegramm Filterung.
Das folgende (kleinere) Code Beispiel gibt Ihnen einen ersten Eindruck wie es funktioniert.

```
1  if not widgets.DIALOG_PAGE then
2      widgets.DIALOG_PAGE = "Filter"
3  end
4
5  function dialog()
6      callback_page( widgets.DIALOG_PAGE )
7  end
8
9  function dialog_filter()
10     widgets.Clear()
11     widgets.RadioBox{ name="page", choices={"Setup","Filter"},
12                       row=1, col=1, fill=true, span=2,
13                       orientation="horizontal", value="Filter" }
14     widgets.Label{ text="This is the FILTER page", row=2, col=1,
15                    span=2, fill=true }
16     widgets.Label{ text="Device Address", row=3, col=1, fill=true }
17     widgets.SpinCtrl{ name="wxFilterAddr", row=3, col=2, fill=true,
18                       min=1,max=16 }
19 end
20
21 function dialog_setup()
22     widgets.Clear()
23     widgets.RadioBox{ name="page", choices={"Setup","Filter"},
24                       row=1, col=1, fill=true,
25                       orientation="horizontal", value="Setup" }
26     widgets.Label{ text="This is the SETUP page", row=2, col=1,
27                    fill=true }
28     widgets.CheckBox{ name="test", label="Test mode",
29                       row=3, col=1, fill=true }
30 end
31
32 function callback_page( page )
33     if page == "Setup" then
34         dialog_setup()
35     else
36         dialog_filter()
37     end
38     -- store selected dialog page
39     widgets.DIALOG_PAGE = page
40     widgets.SaveSettings()
41 end
42
43 function apply()
44     local page = widgets.GetValue( "page" )
45     if page == "Setup" then
```

```
46          -- do apply only setup settings
47      else
48          -- do apply only filter settings
49      end
50  end
```

Zeile 1..3 definiert eine nicht flüchtige Variable zur Speicherung der aktuell ausgewählten Dialogseite, siehe auch 20.8.8. Der Dialog selbst besteht aus zwei unterschiedlichen Seiten. Jede ist in Ihrer eigenen Funktion `dialog_filter()` und `dialog_setup()` kodiert. Die Auswahl erfolgt über eine RadioBox, die in jeder Funktion (Dialogseite) zur Verfügung gestellt wird (Zeile 11 und 23). Die ursprüngliche `dialog()` Funktion ruft in Zeile 6 abhängig vom Wert der Variable `widgets.DIALOG_PAGE` den entsprechende Seiten Code auf. Das gleiche passiert, wenn der Anwender die RadioBox klickt per `callback_page`.

Da hinzugefügte Dialogelemente weder gelöscht oder entfernt werden können, bleibt Ihnen normaler Weise nichts anderes übrig, als alle Elemente durch die der jeweils anderen Dialogseite zu ersetzen. Entweder durch ein entsprechend neues Element oder durch einen Spacer, wenn der Platz im Raster ungenutzt bleiben soll. Dies ist allerdings eine ziemlich aufwendige und teilweise auch kaum machbare Lösung. Vor allem, wenn sich die Raster der beiden Dialog Seiten in der Anzahl der Spalten und Reihen unterscheiden.
Jede der beiden Dialog Seiten löscht deshalb zu Beginn alle bisherigen Widget Elemente durch die Funktion `widgets.Clear()` um einen sauberen (leeren) Anfangszustand des Dialogs herzustellen.
Da dies auch die RadioBox zur Auswahl der Dialogseite einschließt, muss diese in jeder Seite separat als erstes definiert werden (Zeile 11 und Zeile 23). Natürlich können Sie statt einer RadioBox auch ein anderes Widgets Element zur Seitenauswahl verwenden. Z.B. ein Choice Widget oder eine Reihe von Buttons. Das bleibt alleine Ihnen überlassen, am Mechanismus ändert sich dabei nichts.

Noch ein Wort zum `apply` Mechanismus. Da durch den Aufruf von `Clear()` alle Elemente des nicht sichtbaren Dialogs entfernt wurden, können Sie in der `apply` Funktion nur auf den Inhalt der gerade sichtbaren - existierenden - Elemente zugreifen. Ein Aufruf von `widgets.GetValue(...)` mit dem Namen eines nicht existierenden Dialogelements liefert nil. Deshalb ist es sinnvoll, in der `apply` Funktion zunächst die aktuell sichtbare Dialogseite abzufragen und nur den Widget Status der dortigen (Benutzer Eingaben) abzufragen und anzuwenden.

20.8.2 Enable

Mit dieser Funktion können Sie beliebige Dialog Elemente für die User Eingabe sperren. Per Voreinstellung sind alle Elemente aktiviert. Ein deaktiviertes Element erscheint ausgegraut.

```
widgets.Enable( NAME, STATUS )
```

- NAME : der Name des Widgets als Lua String.
- STATUS : der neue Status des Widgets, true oder false

Example

```
function dialog()
    widgets.RadioBox{ name="MyRadioBox", col=1, row=1, label="Mode",
                      orientation="horizontal",
                  choices={ "ENABLED", "DISABLED" } }
    widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
                      datatype=ASCII_STRING, fill=true }
end

function callback_MyRadioBox( mode)
    local enable = true
    if mode== "DISABLED" then enable = false
    else enable = true
    end
    widgets.Enable( "MyTextInput", enable )
end
```

20.8.3 GetPosition

Liefert die Position eines Widgets in dem Layout Raster. Diese Funktion ist insbesondere bei dynamisch erzeugten Layouts sehr nützlich.

```
POSITION = widgets.GetPosition( NAME )
```

⊗ NAME : der Name des Widgets als Lua String.
= POSITION : Die Position als Wertepaar Spalte, Reihe.

Example

```
function dialog()
    for row=1,4 do
        for col=1,4 do
            local name = "B"..col.."x"..row
          if col == 3 and row == 2 then name="PRESS" end
          widgets.Button{ name=name, label=name, col=col, row=row }
        end
    end
end

function callback_PRESS( control)
    local col,row = widgets.GetPosition( "PRESS" )
    widgets.Label{ name="PRESS", text="Ready", col=col, row=row }
end
```

Beachten Sie hier die Rückgabe von zwei Werten. Dies ist eine der vielen Spezialitäten von Lua.

20.8.4 GetValue

Erfragt den (eingegebenen) Wert des angegebenen Widget Elements. Das Resultat hängt von Typ des Widgets ab. Es kann eine Zahl sein (SpinCtrl), ein boolescher Wert (CheckBox), ein String (TextCtrl, Choice, RadioBox) oder eine String Tabelle/Array (Table).

Da Lua in den meisten Fällen eine automatische Typumwandlung durchführt, ist das aber nicht weiter von Belang.

```
VALUE = widgets.GetValue( NAME )
```

⊗ NAME : der Name des Widgets als Lua String.

= VALUE : Der interne Wert des Widgets, der Datentyp kann variieren, s.o.

Example

```
function dialog()
    widgets.Label{ name="MyLabel", text="Input a hex number", col=1,
        row=1 }
    widgets.TextCtrl{ name="MyInput", col=2, row=1, datatype=HEX_NUMBER
        }
end

function apply()
    -- pass the input to the send mechanism
    return widgets.GetValue( "MyInput" )
end
```

20.8.5 IsEnabled

Prüft ob das genannte Widget für Interaktionen bereit (aktiv) ist oder nicht (inaktiv).

```
RESULT = widgets.IsEnabled( NAME )
```

⊗ NAME : der Name des Widgets als Lua String.

= RESULT : liefert true wenn das Widget aktiv ist, ansonsten false.

Example

```
function dialog()
    widgets.Button{ name="MyButton", label="Toogle inout field", col=1,
        row=1 }
    widgets.TextCtrl{ name="MyInput", col=1, row=2, fill=true}
end

function callback_MyButton()
    widgets.Enable( "MyInput", widgets.IsEnabled( "MyInput" ) == false
        )
end
```

20.8.6 SetValue

Setzt den internen Wert des angegebenen Widgets.

```
widgets.SetValue( NAME, VALUE )
```

- NAME : der Name des Widgets als Lua String.
- VALUE : der neue Wert bzw. Zustand des Widgets.

Example

```
1  function dialog()
2      widgets.Button{ name="MyButton", label="Default value", col=1, row
           =1 }
3      widgets.SpinCtrl{ name="MySpinCtrl", col=1, row=2, fill=true}
4  end

5  function callback_MyButton()
6      widgets.SetValue( "MySpinCtrl", 50 )
7  end
```

20.8.7 SetDialogSize

Unter bestimmten Umständen kann es nötig sein, die Größe des Dialogs unabhängig vom internen 'Raster' vorzugeben. Mit dieser Funktion können Sie Breite und Höhe in Pixel explizit setzen.

```
widgets.SetDialogSize{ width=400, height=500 }
```

- width : die neue Breite des Dialogs in Pixel.
- height : die neue Höhe des Dialogs in Pixel.

Example

```
1  function dialog()
2      widgets.SetDialogSize{ width="600", height="400" }
3      widgets.RadioBox{ name="MyRadioBox", col=1, row=1, label="Mode",
4                        orientation="horizontal",
5                    choices={ "ENABLED", "DISABLED" } }
6      widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
7                        datatype=ASCII_STRING, fill=true }
8  end
```

20.8.8 SetTitle

Der im Fensterrahmen angezeigte Dialogtitel ist per Vorgabe 'Template Setup'. Sie können diesen aber jederzeit mit dieser Funktion individuell anpassen.

```
widgets.SetTitle( title )
```

- title : der neue Titel des Dialogs als Lua String.

Example

```
function dialog()
    widgets.SetTitle("Tutorial Protocol Settings")
    widgets.RadioBox{ name="unit",
                      label="Unit",
                      col=1, row=2, fill=true, span=2,
                      choices={"Metric","Anglo-American"} }
    widgets.Label{ text="Telegram EOS",
                   col=1, row=3 }
    widgets.Choice{ name="eos",
                    col=2, row=3, fill=true,
                    choices={"LF", "CR", "LFCR", "CRLF" } }
end
```

LoadSettings

Diese Funktion stellt den zuvor mit `SaveSettings()` abgespeicherten Inhalt der `widgets` Variablen Tabelle wieder her.

```
RESULT = widgets.LoadSettings()
```

= RESULT : true wenn die Einstellungen korrekt geladen wurden, sonst false.

Example

```
if not widgets.LoadSettings() then
    debug.print("OOPS")
end
...
function split(...)
end
```

SaveSettings

Sichert alle im `widgets` Namensraum angelegten Variablen in einer Datei. Der Dateiname wird aus dem aktuellen Namen der Aufzeichnung abgeleitet und bekommt die Endung `msbini`. Die Datei selbst wird im Template Ordner abgelegt und enthält alle Variablen als Name/Wert Paare:

```
eos="%013%010"
unit="Metric"
timeout=0.01
```

Steuerzeichen oder Zeichen außerhalb des normalen ASCII Bereichs werden durch die Zeichenkette `%ddd` ersetzt, wobei `ddd` dem ASCII Code des zu ersetzenden Zeichen entspricht. Im obigem Beispiel bedeutet `%013` das Wagenrücklauf Zeichen (CR), `%010` der Zeilenvorschub (LF). Das Zeichen % selbst wird als `%037`gespeichert.

```
RESULT = widgets.LoadSettings()
```

= RESULT : true bei korrekter Speicherung, sonst false.

Example

```
function apply()
    local t = {
        ["LF"] = "\n",
        ["CR"] = "\r",
        ["LFCR"] = "\n\r",
        ["CRLF"] = "\r\n"
    }
    widgets.unit = widgets.GetValue( "unit" )
    widgets.eos = t[widgets.GetValue( "eos" )]
    if not widgets.SaveSettings() then
          debug.print( "OOPS" )
     end
    return "RELOAD"
end
```

21 Lua Module

Lua Module entsprechen den Bibliotheken in anderen Computer Sprachen. Einige Module haben Sie bereits kennengelernt. Z.B. das math, string oder widgets Modul. Die ersten Beiden sind integraler Bestandteil der Lua Sprache. Das widgets Modul wurde als fest eingebautes Modul von der MSB-RS485-PLUS Software hinzugefügt. Hier lernen Sie, wie Sie eigene Module schreiben können um diese dann später in Ihren Skripten wieder zu verwenden.

Ein Modul ist in der Regel eine Sammlung von Funktionen, die alle einem gemeinsamen Zweck dienen. So enthält das `widgets` Modul alle nötigen Funktionen um eine grafische Bedienoberfläche zu bauen.
In Lua werden alle Modul Funktionen in einer Tabelle gespeichert bzw. verwaltet. Der Aufruf einer Modul Funktion ist dabei nichts anderes als der Zugriff auf ein Tabellen Element/Variable. Der Tabellenname entspricht dem Modulnamen und sorgt dafür, dass die dort gespicherten Funktionen von Funktion mit zufällig gleichem Namen unterschieden werden können[1].
Betrachten wir dazu folgendes Code Beispiel:

```
1  function Sleep( t )
2      -- will not executed!
3  end
4  time.Sleep( 0.5 )
```

Während der Ausführung dieser Zeilen sucht Lua nach der Funktion `Sleep` in der bereits geladenen Tabelle `time`. Die zu Beginn definierte Funktion `Sleep` wird NICHT aufgerufen!

Aber moment! Was bedeutet in diesem Zusammenhang 'die bereits geladene Tabelle'?
Im Falle eines fest eingebauten (built-in) Modules wie z.B. `string` oder `widgets` wurden diese Modul Tabellen bereits beim Start vom Lua Interpreter automatisch geladen. Ihre Funktionalität steht Ihnen damit ohne zusätzlichen Code direkt zur Verfügung. Individuelle Module (von Ihnen geschrieben) können nicht einfach vom Interpreter vorgeladen werden, da der Interpreter nichts über diese weiß. Es ist Ihre Aufgabe dies zu tun. Der Lua Funktionsaufruf um ein Modul zuladen ist:

```
require "modname"
```

[1] In anderen Sprachen wie z.B. C++ kennt man das Konzept als 'name space'.

Einfach ausgedrückt sucht die Funktion `require` in den vorgegebenen Pfaden[2] nach einer Datei mit dem angegebenen Namen und führt diese aus. Das Resultat wird von Lua zwischen gespeichert. Bei erneutem Laden erhalten Sie deshalb das zuvor gespeicherte Resultat zurück.
Ein Modul kann - natürlich - nur eine einzige Anweisung enthalten:

```
-- minimal.lua
debug.print( "I'm a module" )
```

In diesem Fall erhalten Sie eine Debug Ausgabe sobald Sie das Modul per `require` in Ihrem Skript laden. Aber nur einmal! Bei erneutem Laden des Moduls erkennt Lua automatisch, dass das Modul bereits geladen ist. Es wird deshalb nicht erneut ausgeführt!

```
require "minimal"  --> outputs "I'm a module"
require "minimal"  --> nothing!
```

Module mit direkt ausgegeführten Code (also keine Funktionsdefinitionen) sind perfekt geeignet um einmalige Initialisierungen durchzuführen. Z.B. um Ihr Programm mit vordefinierten Konstanten (Feldbus Protokoll Bezeichner oder ähnliches) zu versorgen. Aber solche Module repräsentieren nicht wirklich ein Modul im Sinne einer zuvor erwähnten Funktionssammlung.

21.1 Ein Modul schreiben

Wie bereits gesagt enthält ein Modul i.a. eine Reihe von Funktionen, die in einer internen Modul Tabelle organisiert sind. Stellen Sie sich ein kleines Modul vor, welches die beiden Funktionen `min` und `max` zur Verfügung stellt. Beide Funktionen werden mit jeweils zwei Zahlen aufgerufen und liefern entweder den kleineren oder größeren Wert zurück. Beide Funktionen sollen Teil eines neuen Moduls `algorithm` sein. Hier zunächst der Modul Code:

```
1 -- algorithm.lua
2 return {
3     min = function(a,b) if a > b then return b else return a end end,
4     max = function(a,b) if a < b then return b else return a end end
5 }
```

Erinnern Sie sich. Lua unterscheidet nicht zwischen Zahlen, Strings oder Funktionen. All diese sind für Lua sogenannte First-Class Werte[3] und Sie können eine Funktion genause leicht als Tabelleneintrag speichern wie jeden anderen Typ. In unserem Beispiel weisen wir den beiden Tabelleneinträgen `min` und `max` einfach die entsprechende Funktion zu.
Das Modul liefert beim Laden per `require` eine Tabelle zurück, die Lua intern in seiner eigenen Modul Tabelle speichert. Um allerdings später auf diese Tabelle (das Modul) zugreifen zu können, benötigen wir noch eine Art Referenz auf das geladene Modul. Wie bereits zuvor gesagt liefert `require` das Resultat des geladenen Modul Codes. Hier die Tabelle mit den beiden Einträgen `min` und `max`, die Sie einfach einer als Referenz dienenden Variable zuweisen.

```
local algorithm = require "algorithm"
```

[2] Wo Lua nach Modulen sucht ist Teil eines späteren Abschnitts
[3] Die Bezeichnung First-Class Werte bedeutet, dass in Lua eine Funktion die gleichen Rechte besitzt wie Strings oder Zahlen und deshalb gleichberechtigt in Variablen oder Tabellen gespeichert werden können.

Danach können Sie jede Modul Funktion so einfach aufrufen wie jeden anderen Zugriff auf einen Tabelleneintrag. Um den min oder max Wert eines Zahlenpaares zu ermitteln, reicht ein:

```
local algorithm = require "algorithm"
debug.print( algorithm.min(1,2) ) --> outputs 1
debug.print( algorithm.max(1,2) ) --> outputs 2
```

Beachten Sie! Wir haben die Referenz nach dem Modul Namen benannt. Das ist übliche Praxis, aber letztendlich Ihre Entscheidung.

Ein Module wie oben zu codieren (Funktionsrümpfe direkt in der Tabelle) mag für sehr kleine Funktionen noch in Ordnung gehen. Es ist aber sicherlich keine gute Praxis für komplexere Funktionen. Ein weitaus besserer Ansatz ist zunächst eine leere Tabelle zu definieren und die Funktionen dann später hinzuzufügen:

```
1  local m = {}
2
3  function m.min( a, b )
4     if a > b then return b else return a end
5  end
6
7  function m.max( a, b )
8     if a > b then return a else return b end
9  end
10
11 return m
```

Zeile 1 erzeugt eine leere und lokale (d.h. nur im Modul sichtbare) Tabelle. Wir können anschließend jede Funktion (wie auch jede beliebige Variable) der Tabelle hinzufügen, indem wir einfach den Tabellennamen getrennt mit einem Punkt '.' dem Funktions- oder Variablennamen voran setzen. Alternativ können Sie natürlich auch schreiben:

```
1  local m = {}
2
3  m.min = function( a, b )
4     if a > b then return b else return a end
5  end
6
7  m.max = function( a, b )
8     if a > b then return a else return b end
9  end
10
11 return m
```

Aber der erste Ansatz ist eleganter.

Auf Funktionen oder Variablen, die nicht Teil der Modul Tabelle sind, kann von außen nicht zugegriffen werden. Sie verhalten sich damit wie nicht öffentliche Funktionen/Variablen in einer C++ Klasse. Um die Bedeutung dieses Konzepts zu verdeutlichen erweitern wir das `algorithm` Modul um eine Funktion zur Berechnung der Fakultät einer gegebenen Zahl.

```
1  function m.fact( n )
2     if n < 0 then return nil
3     elseif n == 1 then return 1
4     else return n * m.fact( n - 1 )
5     end
```

```
6  end
```

Die Funktion `fact` ist eine rekursive Funktion und ruft sich n mal selbst auf (siehe Zeile 4). Und wie es bei rekursiven Funktion üblich ist, beherbergen sie alle die Gefahr eines Stack Überlaufs! Hier allerdings wird dies nicht passieren, da die Funktion vor einem möglichen Überlauf bereits die größte mögliche Fließkommazahl mit n=170 erreicht und 'unendlich' (inf) zurück gibt.
Nehmen wir aber einmal an der Stack ist limitiert. In diesem Fall müssen wir die Anzahl der rekursiven Aufrufe von `fact` auf einen festen, durch den vorhandenen Stack vorgegebenen, Wert begrenzen. Die maximale Anzahl wird durch die interne (private) Variable `MAX_STACK` mit der Vorgabe 100 spezifiziert. Private deshalb, weil sie nicht Teil der zurückgegebenen Tabelle ist.

```
1  local MAX_STACK = 100
2
3  function m.fact( n )
4     if n < 0 or n >= MAX_STACK then return nil
5     elseif n == 1 then return 1
6     else return n * m.fact( n - 1 )
7     end
8  end
```

Nun benötigen wir nur noch eine Funktion, um dieses Limit von außen verändern zu können.

```
1  function m.set_stack_limit( n )
2     if n > 2 and n <= 100 then
3        MAX_STACK = n
4     end
5  end
6
7  function m.fact( n )
8     if n < 0 or n >= MAX_STACK then return nil
9     elseif n == 1 then return 1
10    else return n * m.fact( n - 1 )
11    end
12 end
```

Die Funktion `set_stack_limit` ist somit die einzige Möglichkeit das Stack Limit von außen zu kontrollieren. Und das ausnahmslos nur in einem sinnvollen Bereich.

In diesem Beispiel 'versteckten' wir nur eine normale Variable. Es ist aber immer eine gute Idee auch Funktionen, die nur innerhalb des Moduls verwendet werden, nicht in der Modul Tabelle einzutragen.

21.2 Modul Pfad

Der Pfad unter welchem die MSB-RS485-PLUS Software beim Aufruf von `require` nach einem Modul sucht ist abhängig vom Betriebssystem:
Unter Linux ist es:

```
~/.IFTOOLS/SerialAnalyzer/7.0.2/Templates/Modules
```

Unter Windows:

```
C:\Users\USERNAME\AppData\Roaming\IFTOOLS\SerialAnalyzer\7.0.2\Templates\Modules
```

Sie können das einfach selbst testen. Rufen Sie dazu `require` mit einem nicht vorhandenen Modul Namen auf:

```
algorithm = require "algoorithm"
```

Das Resultat ist eine Lua Fehlermeldung inklusive der Angabe des gültigen Suchpfades (hier unter Linux):

```
[string "--[[..."]:8: module 'algoorithm' not found:
no field package.preload['algoorithm']
no file '~/.IFTOOLS/SerialAnalyzer/6.0.2/Templates/Modules/algoorithm.lua'
```

Die letzte Zeile ist der Hinweis, wo Lua nach einem Modul sucht.
Sie können Module sogar in verschiedenen Unterverzeichnisse organisieren. Z.B. wenn Sie eine neue Version eines Moduls implementieren, die alte aber bis zur Fertigstellung bereithalten wollen. In diesem Fall ergänzen Sie den Modul Parameter im `require` Aufruf durch den Namen des Unterverzeichnisses. Übertragen auf unser Modul:

Sie haben zwei Versionen des `algorithm` Moduls, 1.0 und 1.1. Im Modul Verzeichnis sind diese gespeichert unter `Modules/1.0/algorithm.lua` und `Modules/1.1/algorithm.lua`. Die gewünschte Version wird geladen mit:

```
algorithm = require "1.0/algorithm"
```

Sie können sogar eine Variable zur Umschaltung zwischen den Versionen verwenden:

```
local modpath = "1.0/algorithm"
algorithm = require( modpath )
```

Beachten Sie hier aber!
Beim Übergeben des Moduls (inklusive optionalem Pfad) als Variable müssen Sie diese in umschließende Klammern setzen um Lua klar zu machen, dass es sich hierbei um einen Funktionsparameter handelt.
Und: Lua konvertiert alle Modul Pfade intern passend zum OS. Es macht also keinen Unterschied, ob Sie unter Windows oder Linux arbeiten. Sie sollten Pfade immer per '/' trennen! Das ist nicht nur Plattform unabhängig, es sieht auch besser aus als:

```
-- in a string a backslash must be input as double backslashes
local modpath = "1.0\\algorithm"
algorithm = require( modpath )
```

22 Zwei Verbindungen aufzeichnen

Sie haben zwei Verbindungen (RS232 und/oder RS422/485) die Sie gemeinsam betrachten bzw. untersuchen müssen, z.B. Ein- und Ausgangsdaten eines Protokollumsetzers, verschiedene Bussegmente, oder generell voneinander abhängige Datenübertragungen. Wie Sie hier vorgehen und was Sie dabei beachten müssen zeigt dieses Kapitel.

Zur gleichzeitigen Aufzeichnung zweier getrennter Verbindungen benötigen Sie zwei MSB-Analyser. Dies ist allerdings nur eine Voraussetzung. Um zwei Aufzeichnungen vergleichen zu können müssen die aufgenommenen Daten in einer genauen zeitlichen Beziehung zueinander stehen.
Ohne diese können weder exakte Aussagen über die zeitliche Abfolge der aufgezeichneten Datensequenzen gemacht, noch die Gleichzeitigkeit bestimmter Ereignisse überprüft werden. Z.B. wann wurde ein Datenbyte oder eine Bytefolge in der einen Verbindung relativ zu den Daten der anderen Verbindung gesendet? Was passierte in beiden Verbindungen zu einem exakten Zeitpunkt?

22.1 Technische Voraussetzungen

Eine der herausragenden Fähigkeiten der MSB-Analyser ist die exakte Messung und Visualisierung des Zeitverhaltens einer Verbindung mit Mikrosekunden Genauigkeit. Diese Präzession ist erforderlich um auch bei höheren Baudraten korrekte Ergebnisse zu liefern und gilt auch für die gemeinsame Analyse zweier Verbindungen. Was heißt das im Einzelnen?

Stellen Sie sich zwei 'Agenten' vor, die in ein streng abgesichertes Gebäude eindringen wollen und dazu den genauen Ablauf der Wachen an unterschiedlichen Standorten beobachten müssen. Bevor Sie beginnen führen beide einen Uhrenvergleich durch. Dies geschieht in der von normalen Uhren gewährleisteten Genauigkeit von 1 Sekunde. Beide Agenten verfügen zudem über Uhren die auch nach einem Tag nicht mehr als 1 Sekunde voneinander abweichen. Nicht auszudenken was passieren würde, wenn bereits nach mehreren Minuten beide Uhren auseinander laufen würden. Anschließend begeben sich beide an Ihre Position.
Jeder einzelne notiert den Zeitpunkt (Sekunden genau) des Wachwechsels, Wachganges etc. Da die Beobachtung mehrere Tage in Anspruch nimmt, synchronisieren beide Agenten Ihre Uhren erneut indem einer der Agenten exakt um Mitternacht einen kurzen Funkimpuls sendet.

Für die erfolgreiche Umsetzung ihres Planes ist es unabdingbar jeden Schritt der Wachen Sekunden genau zu kennen.

Das gleiche Szenario, allerdings mit einer weitaus höheren Genauigkeit stellt sich bei der simultanen Aufzeichnung zweier Verbindungen. Der Uhrenvergleich steht hier für den 'gleichzeitigen' Start der Aufnahme (wobei gleichzeitig hier auf die millionstel Sekunde genau meint).

Die Uhren beider MSB-Analyser sind natürlich um Größenordnungen genauer als die Armbanduhren der beiden Agenten. Nicht desto trotz laufen auch diese bedingt durch kleine Unterschiede in den Schwingquarzen stetig auseinander und müssen deshalb in regelmäßigen Abständen neu synchronisiert werden.
Die MSB-Analyser verwenden deshalb für beides, den simultanen Start der Aufzeichnungen als auch für die regelmäßige Abstimmung ihrer Uhren eine zusätzliche Synchron-Verbindung.
Jeder MSB-Analyser besitzt hierfür eine sogenannte 'MSB-Link' Buchse in RJ45 Ausführung. Um zwei Analyser zu synchronisieren werden diese einfach mit einem handelsüblichen Netzwerkkabel (1:1) über diese Buchsen miteinander verbunden.
Es spielt dabei keine Rolle, ob die beiden Analyser an einem gemeinsamen oder an verschiedenen PCs betrieben werden. Die PC's müssen auch nicht einem gemeinsamen Netzwerk angehören. Die einzige Einschränkung liegt in der Länge des Synchron Kabels mit dem die Geräte verbunden sind.[1] Die Synchronisierung wirkt sich lediglich auf den Beginn der Aufzeichnung und die genaue Einhaltung einer gemeinsamen Zeitbasis aus. D.h. die Zeitstempel der aufgenommenen Ereignisse beider Analyser sind auf die millionstel Sekunde genau vergleichbar.

Darüber hinaus arbeiten beide Analyser völlig autark. D.h. Sie können gleichzeitig unterschiedliche Protokolle und Ereignisse vorgeben und aufzeichnen (Baudrate, Datenformat etc.). Mehr noch: Bei Verwendung eines MSB-RS232 und eines MSB-RS485 Analysers können auch RS232 und RS422/485 Verbindungen simultan aufgenommen und untersucht werden, beispielsweise Schnittstellenwandler etc.

22.2 Master Slave Betrieb

Es macht wenig Sinn, die Aufzeichnung der synchronisierten Analyser einzeln zu starten. Vor allem, wenn beide Geräte an unterschiedlichen PCs laufen, die evtl. sogar räumlich getrennt voneinander sind.
Start, Pause und Stop der gemeinsamen Aufnahme erfolgen deshalb über einen zuvor als 'Master' definierten Analyser.
Dieser ist vom Anwender frei wählbar. Der zweite, über das Synchron Kabel verbundene Analyser wird dabei automatisch zum 'Slave' und steuert seine Aufzeichnung simultan zum Master - Mikrosekunden genau.
Beide synchronisierten Analyser können so konfiguriert werden, dass die aufgenommenen Daten automatisch bei Aufzeichnungsstop an einem zuvor angegebenen Speicherort abgelegt werden. Das kann die lokale Festplatte des mit dem jeweiligen Analyser verbundenen PC's sein, aber auch ein beliebiges

[1] Getestet wurden CAT6 Netzwerkkabel mit 100m Länge.

Netzlaufwerk.
Die Aufnahmedateien werden dabei nach Seriennummer und Datum/Zeit des Aufnahmestarts benannt. Zusätzlich können Sie ihnen aber noch eine beliebige Zeichenkette (Prefix) voranstellen.

22.3 Einrichten einer synchronen Aufzeichnung

Sie haben jetzt eine ungefähre Vorstellung wie eine Synchron-Aufzeichnung von statten geht. Kommen wir deshalb zum praktischen Teil. Stellen Sie sich vor, Sie haben zwei RS232 Verbindungen, die Sie synchron aufzeichnen wollen. Der Einfachheit halber erfolgt die Aufzeichnung mit einem einzelnen PC. D.h. beide Analyser sind an einem gemeinsamen PC angeschlossen.

Verbinden Sie zunächst beide Geräte über die Link MSB Buchsen mit einem Standard Netzwerkkabel. Für längere Verbindungen empfehlen wir Netzwerkkabel der Kategorie CAT-6, allerdings sollten für die meisten Anwendungen CAT-5 Kabel völlig ausreichen.

Warnung!
Bitte beachten Sie, das der Analyser nicht per MSB Link Buchse mit einem PC Netzwerk Anschluss verbunden werden darf! Dies hat mit großer Wahrscheinlichkeit einen Defekt des Gerätes zur Folge!

Starten Sie die Analyser Software über den Desktop Start Icon. Bei mehreren mit dem PC verbundenen Geräten müssen Sie den gewünschten Analyser aus einer Liste auswählen ("Wähle angeschlossenen Analyser").

Wiederholen Sie den letzten Schritt für den zweiten Analyser.
(Die gleiche Vorgehensweise gilt auch bei an getrennten PC's betriebenen Analysatoren wobei jeder PC eine Anwendung der Analyser Software startet.)

Platzieren Sie beide Kontrollprogramme (jedes verbunden mit einem Analyser) auf dem Bildschirm.
Noch arbeiten beide Geräte völlig unabhängig voneinander. D.h. Sie können die Aufzeichnung jedes einzelnen Analysers einzeln starten, pausieren oder stoppen. Da beide Geräte unterschiedliche Verbindungen (mit verschiedenen Verbindungsprotokollen bzw. Verbindungstypen wie RS232 oder RS422/485) aufnehmen können, müssen Sie diese jeweils zunächst konfigurieren. Dies geschieht analog zur Aufzeichnung einer nicht synchronen Aufnahme. D.h. Sie geben alle nötigen Verbindungsparameter, Auswahl der aufzuzeichnenden Ereignisse etc. im Einstelldialog, getrennt für beide Analyser, vor.
Per Voreinstellung speichern beide Analyser ihre aufgenommenen Daten auf dem Desktop, sobald die synchrone Aufzeichnung vom Master (d.h. von Ihnen) gestoppt wurde. Sie können den Speicherort allerdings auch beliebig vorgeben indem Sie im Einstelldialog unter 'Autospeichern' einen anderes Verzeichnis auswählen.
Der Dateiname wird vom Analyser Programm vorgegeben um auch bei wiederholter Abspeicherung Fehler durch bereits vorhandene Dateien auszuschließen und später die Dateien einem Analyser einwandfrei zuordnen zu können.

Aufnahme Speicherort ist unter Autospeichern frei wählbar, z.B. ein Netzlaufwerk

Die Dateinamen entsprechen folgender Form, hier als Beipiel eine Aufnahme des Analysers mit der Seriennummer MSB01060, gestartet am 16 April 2014 um 15:32,17.

```
MSB01060-20140416153217.msblog
```

Sie können diesem allerdings noch eine beliebige Zeichenkette als Prefix voranstellen, z.B. MASTER oder SLAVE.

Master und Slave
Auswahl des Aufnahme Masters

Nachdem Sie beide Geräte konfiguriert haben müssen Sie jetzt nur noch den Master für die synchrone Aufnahme vorgeben (vorausgesetzt Sie haben beide Analyser per Netzwerkkabel verbunden).
Aktivieren Sie dazu einfach im Einstelldialog des Gerätes, welches die Master Rolle übernehmen soll, unter Aufnahme die Auswahl 'Analysator ist Master'.
In der Anzeige wird über der laufenden Aufnahmezeit das Wort 'Master' eingeblendet. Zeitgleich zeigt das mit dem Master verbundene Gerät den Status 'Slave' in seiner Anzeige und deaktiviert die Knöpfe und Menüeinträge zur Aufzeichnungssteuerung.

Schliessen Sie den Einstelldialog und klicken Sie im Kontrollprogramm des Masters auf den Aufnahmeknopf. Beide Geräte wechseln in den Record Modus, angedeutet durch eine entsprechende Darstellung des Knopfes sowie der roten LEDs am Gerät selbst.
Klicken Sie den Pause Knopf des Masters um die Aufnahme beider Geräte pausieren zu lassen.
Mit Klick auf den Stop Knopf wird die Aufzeichnung beendet. Beide Analysatoren, Master und Slave, speichern die evtl. vorhandenen Daten auf Ihrem Desktop (oder jedem anderen von Ihnen vorgegebenen Verzeichnis).

Sie können den Vorgang beliebig oft wiederholen. Sobald Sie den Aufnahmeknopf erneut klicken, starten beide Analyser eine neue Aufnahme und speichern diese bei Stop automatisch als zwei weitere Dateien.

Diese Vorgehensweise unterscheidet sich auch nicht, wenn Sie beide Analysatoren an 'getrennten' Rechnern betreiben die eventuell sogar in verschiedenen Räumen stehen. Einzige Bedingung ist eine direkte Verbindung durch das Synchron Kabel.

22.4 Auswertung/Analyse einer synchronen Aufzeichnung

Die MSB-Analyser Software ist darauf optimiert, eine einzelne Aufnahme durch verschiedenste Views zu betrachten (MultiView Konzept). Das Laden mehrerer Aufnahmedateien (oder Projekte) ist nicht vorgesehen, da zwei oder mehrere z.T. völlig verschiedene Aufzeichnungen innerhalb einer Anwendung keinen Sinn ergeben. Man denke hier nur an Aufnahmen einer RS232 UND RS485 Verbindung[2].
Wie aber nun zwei synchrone Aufnahmen analysieren?

[2]Letztendlich entspricht eine laufende MSB-Analyser Programmanwendung immer EINER Aufzeichnung. Das gleiche gilt im übrigen für Audio und Video Anwendungen.

Die MSB-Analyser Software erweitert dazu die bereits vorhandene Kommunikation zwischen den Views einer einzelnen Anwendung (Sitzung) konsequent auf mehrere, parallel laufende Anwendungen. D.h. so wie die Signaldarstellung dem Cursor des Datenmonitors folgt, folgen bzw. synchronisieren sich auch die Views zweier getrennt laufender MSB-Analyser Programme.
Dies hat eine Reihe von entscheidenden Vorteilen:

- Vergleichende Analyse auch völlig unterschiedlicher Aufzeichnungen (Verbindungsart, Protokoll, ...).
- Synchrones Springen und paralleles Einblenden von bestimmten Aufnahmebereichen in beiden Aufzeichnungen (z.B. Suche nach Ereignis in Aufzeichnung A und Einblendung der zugehörigen Signalsequenz in Aufzeichnung B).
- Flache Lernkurve, kein neues Bedienkonzept, keine zusätzlichen Menüs.

Die Analyse zweier synchronisierter Aufzeichnungen unterscheidet sich deshalb nicht sonderlich von der Untersuchung einer einzelnen Aufnahme. Statt nur einer laufenden MSB-Analyser Anwendung starten Sie jetzt für die Master und die Slave Aufnahme einfach zwei getrennte Programme.
Zur gemeinsamen Analyse benötigen Sie keinen verbundenen Analyser, die Untersuchung der Aufzeichnungen kann gewohnt im Offline Modus erfolgen. Klicken Sie dazu einfach nacheinander auf die vom Master und Slave gemachten Aufnahmen.
Beide Anwendungen bieten Ihnen den gewohnten Zugriff auf die entsprechende Aufzeichnung. Die zu einer Aufzeichnung oder Anwendung gehörenden Views synchronisieren ihre Ansichten durch entsprechendes Aktivieren des Sync. Modus in der Werkzeugleiste.

Ext. Synchronisierung aktivieren in den Aufnahme Einstellungen

Um die einzelnen Views zwischen BEIDEN laufenden Anwendungen zu synchronisieren, müssen Sie dies allerdings zuerst freigegeben. Per Voreinstellung sind Views generell gegen Synchron-Ereignisse von außerhalb gesperrt.
Die Freigabe erfolgt für alle Views zentral im Einstelldialog des Kontrollprogrammes unter 'Allgemein'e Einstellungen. Aktivieren Sie dort den Punkt 'Externe Synchronisation erlauben'.

Mit Freischaltung der externen Synchronisation 'empfängt' das Kontrollprogramm die entsprechenden Mausklicks oder Ereignisse (wie z.B. Suchresultate, Regionauswahl, etc.) einer parallel laufender Analyser Anwendungen und reicht diese an 'seine' geöffneten Views weiter. Jedes View mit aktiver Synch. Einstellung wird auf diese Ereignisse reagieren und seine Darstellung aktualisieren.
Auf diese Weise können Sie zu jedem Zeitpunkt innerhalb der Master Aufnahme den entsprechenden Abschnitt der Slave Aufzeichnung in der von Ihnen gewünschten Darstellung (View) einblenden und umgekehrt.

Synchrone Darstellung der einzelnen Views in beiden Aufzeichnungen

22.5 Synchronisierung von mehr als zwei Analysern

Was ist, wenn Sie mehr als zwei Analyser synchronisieren wollen. Dies ist zwar eher selten, nicht desto trotz können bestimme Anwendungen das synchrone Aufzeichnen von mehr als zwei Übertragungen erforderlich machen.
Wie bereits beschrieben ist die Idee hinter der Synchronisierung, das der Master die Uhren der verbundenen Slaves an seine eigene Uhr adaptiert indem er

MSB-Link-Doubler
verdoppelt Sync-Signal

in regelmäßigen Zeitabständen einen Synchron Pulse über die MSB Link Verbindung sendet.
Indem dieses Signal in zwei weitere gesplittet wird, kann der Master bereits zwei Slaves synchronisieren.
IFTOOLS bietet einen solchen Splitter als MSB-Link-Port-Doubler an. Wenn der Splitter mit dem MSB-Link Anschluss des Master Gerätes verbunden wird, können zwei Analyser als Slaves synchronisiert werden. Jeder weiterer hinzugefügte Splitter ermöglicht die Synchronisierung eines weiteren Analysers.
Der Splitter sowie eine detaillierte Beschreibung ist unter MSB-Link-Port-Doubler im IFTOOLS Webshop erhältlich.

22.6 Zusammenfassung

Die vergleichende Aufnahme bzw. Analyse zweier getrennter Verbindungen erfordert eine sehr genaue zeitliche Referenz um die aufgenommenen Daten und Ereignisse in Beziehung zueinander zu setzen.
Diese Kapitel zeigte Ihnen warum das so ist, welche technischen Voraussetzungen erfüllt sein müssen und wie eine solche Aufnahme mit den MSB-Analyser durchgeführt wird.
Hier noch einmal die nötigen Schritte ohne zusätzlichen technischen Ballast:

22.6.1 Synchrones Aufzeichnen

1. Verbinden Sie beide zu synchronisierenden Analyser über die Link MSB Buchse mit einem handelsüblichen Netzwerkkabel.
2. Schließen Sie beide Analyser gemeinsam oder getrennt an einen bzw. zwei PC's an.
3. Starten Sie für beide Geräte ein separates MSB-Analyser Programm.
4. Stellen Sie für beide Analyser individuell Verbindungsparameter und aufzuzeichnende Ereignisse ein.
5. Prüfen Sie, ob die automatische Speicherung nach Stop der synchronen Aufzeichnung (Einstellung 'Autospeichern') aktiviert ist und geben Sie ggf. einen Speicherort vor.
6. Definieren Sie eines der Geräte im Einstelldialog des zugehörigen Programmes unter 'Aufnahme' als Master.
7. Starten Sie die Synchron-Aufzeichnung im Master Kontrollprogramm.
8. Die Aufzeichnung wird ebenfalls durch den Master beendet wobei die Aufnahmen automatisch getrennt gesichert werden.

22.6.2 Synchrones Auswerten/Analysieren

Zur Auswertung zweier synchron aufgenommener Aufzeichnungen benötigen Sie keinen angeschlossenen Analyser. Allerdings müssen beide MSB-Analyser Programm Instanzen auf dem gleichen Rechner laufen, da eine Synchronisierung der Views nicht über Rechnergrenzen hinweg möglich ist (im Gegensatz zur Synchronisierung der Aufnahme).

1. Doppelklicken Sie nacheinander beide (Master und Slave) Aufzeichnungen bzw. starten Sie zwei MSB-Analyser Programm Instanzen und laden Sie diese in das Kontrollprogramm.

2 Aktivieren Sie in beiden Programmen im Einstelldialog unter ’Allgemein’ den Punkt ’Externe Synchronisation erlauben’.

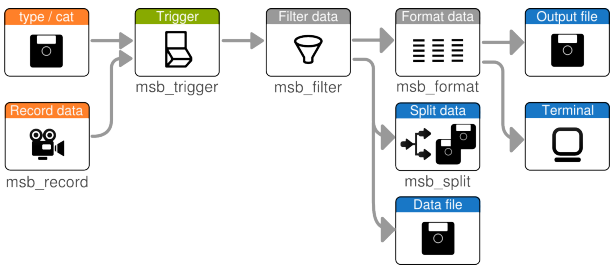
3 Plazieren Sie beide Kontrollprogramme und die von Ihnen benötigten zugehörigen Views auf Ihrem Bildschirm.

4 Navigieren Sie wie gewohnt durch beide Aufzeichnungen. Views in Sync. Modus werden dabei automatisch ihren Inhalt am aktuell untersuchten Zeitabschnitt ausrichten.

23
Kommandozeilen API

Sie wollen die Aufzeichnung einer Verbindung automatisieren und die aufgenommenen Daten in Ihrer eigenen Applikation weiter verarbeiten oder in einem bestimmten Format speichern bzw. ausgeben.
Eine Langzeitaufnahme soll in mehreren aufeinanderfolgenden Dateien gespeichert oder nachträglich aufgesplittet werden.
Der Analyser soll von Ihrer Anwendung gesteuern werden.
Die MSB-Analyser Software enthält hierzu eine Reihe von leistungsfähigen Tools.

Nach der Installation der Analyser Software befinden sich im Installationsverzeichnis neben den Programmen für die Bedienung der MSB-RS485-PLUS und Darstellung bzw. Analyse der aufgenommenen Daten noch eine Reihe weiterer kleiner 'Helferlein'.
Alle diese Programme arbeiten per Kommandozeile und können auch als Teil von Batchdateien oder Shell Skripten verwendet werden.
Gemäß der Unix Philosophie: *"Mache nur eine Sache und mache sie gut"* erfüllt jedes dieser Programme einen bestimmten Zweck. Durch ihre Fähigkeit Datenströme aus der Standardeingabe zu lesen und diese auf der Standardausgabe wieder ausgegeben zu können, können die Programme zu beliebigen Verarbeitungsketten kombiniert werden.
Mehr noch: Sie können sie mit einer Vielzahl weiterer Programme verwenden, die ebenfalls Daten per Standardein/ausgabe verarbeiten können.



Die Programme im Überblick:

- **msb_record**
  Steuert den verbundenen Analyser und schreibt alle empfangenen Analyserdaten auf die Standardausgabe oder in eine angegebene Datei.

- **msb_format**
  Liest gültige Analyserdaten von der Standardeingabe, formatiert diese und schreibt das Ergebnis auf die Standardausgabe oder in eine Datei.
- **msb_filter**
  Die von der Standardeingabe gelesenen Daten werden nach vorgegebenen Regeln gefiltert und die passierten Daten auf der Standardausgabe wieder ausgegeben.
- **msb_split**
  Liest gültige Analyserdaten von der Standardeingabe und schreibt diese in eine oder mehrere Dateien (splitten).
- **msb_trigger**
  Prüft die von der Standardeingabe gelesenen Daten gegen ein oder mehrere vordefinierte Trigger-Bedingungen. Ist die Trigger-Bedingung erfüllt werden die Daten an die Standardausgabe durchgereicht, ansonsten verworfen.

23.1 Beliebige Verarbeitungsketten mittels Pipes

Eine Verarbeitungskette besteht immer aus einer Datenquelle und einer Datensenke (Empfänger). Zwischen beiden können andere Programme den Datenstrom (hier die vom Analyser aufgenommenen Daten) manipulieren. Die Verknüpfung der einzelnen Programme erfolgt dabei über den für Windows und Linux identischen Pipe Operator '|'.

```
DATENQUELLE | MANIPULATOR1 | MANIPULATOR2 | ... | DATENSENKE
```

23.1.1 Datenquellen

Datenquellen stehen am Anfang einer Vearbeitungskette und 'liefern' die nötigen Daten auf der Standardausgabe, hier vor allem das Tool `msb_record` . Genauso geeignet ist aber auch die Ausgabe einer bereits vorhandenen Aufzeichnungsdatei durch das Kommando `type` (Windows) bzw. `cat` (Linux). Beispielsweise:
`type recordfile.msblog` bzw. `cat recordfile.msblog`

23.1.2 Manipulatoren

Als Manipulatoren werden Programme bezeichnet, die den eingelesenen Datenstrom in irgendeiner Form verändern bevor sie diesen wieder ausgeben. Ein typischer Manipulator könnte bestimmte durch den Analyser aufgenommenen Ereignisse entfernen oder die Daten in ein anderes Format umwandeln bevor er diese wieder ausgibt und damit in der Verarbeitungskette weiter reicht. Durch eine Kette mehrerer miteinander verbundener Manipulatoren läßt sich die Bearbeitung beliebig erweitern oder ergänzen. So könnten zunächst alle unerwünschten Daten entfernt und anschließend der Rest umgewandelt werden, z.B. mit dem Tool `msb_format`.

23.1.3 Datensenken

Datensenken definieren das Ende der Verarbeitungskette. Eine typische Datensenke ist die Standardausgabe (die nichts anderes macht, als die empfangenden Daten auf dem Bildschirm bzw. innerhalb der Kommandokonsole auszugeben) oder eine Datei die das Ergebnis der Verarbeitung speichert.
Ein Datensenke könnte aber auch Ihre Anwendung sein, indem Sie die Daten

einfach einliest und gemäß Ihren Vorgaben verarbeitet, beispielsweise LabView.
Bei `msb_split` handelt es sich um eine typische Datensenke. Die Daten werden nicht weiter gereicht sondern als Datei(en) gespeichert.

23.1.4 Ein paar Beispiele

Das Programmverzeichnis der MSB-Analyser Software wird während der Installation automatisch dem Suchpfad für ausführbare Programme hinzugefügt. Einem ersten Versuch (auch ohne verbundenem Analyser) steht deswegen nichts im Wege.
Öffnen Sie ein Kommandoeingabefenster (Konsole) und wechseln Sie in das Beispielverzeichnis `examples` der Analyser Software.
Leiten Sie eine Aufzeichnung aus dem Beispielverzeichnis in das `msb_format` (pipen). Beispielsweise:

```
type DataView\9bit.msblog | msb_format
```

Linux Anwender verwenden natürlich statt dem `type` Befehl das Kommando `cat`. `type Datei` entspricht dabei der Datenquelle, `msb_format` dem Manipulator. Die Ausgabe (der Empfänger bzw. die Datensenke) ist die Konsole die das Ergebnis auf Ihrem Bildschirm ausgibt.

Verwenden Sie das Tool `msb_split` um die gleiche Datei in kleine Häppchen zu zerlegen.

```
type DataView\9bit.msblog | msb_split -n1000
```

Ohne zusätzliche Parameter generiert das Split Programm zwei Dateien mit den Namen xaa.msblog sowie xab.msblog.
Detailierte Informationen zu den einzelnen 'Helferlein' finden Sie in den folgenden Abschnitten.

23.2 Aufzeichnen mit msb_record

Wie der Name `msb_record` bereits andeutet ist dieses Tool für die Steuerung und Aufzeichnung der Daten des verbundenen MSB-Analyser zuständig. Gleichzeitig fungiert es als 'Datenquelle' für alle anderen Tools.
Aufgerufen ohne weitere Parameter sucht das Programm nach einem mit dem PC verbundenen Analyser, überträgt nötigenfalls die Firmware und startet eine Aufzeichnung, wobei Baudrate und Protokoll mit '115200 8N1' und Aufnahme aller übertragenen Datenbytes voreingestellt sind.
Sollte das Programm keinen oder mehrere Analyser erkennen, zeigt es dies mit einer entsprechenden Meldung an. In letzterem Fall können Sie einen Analyser gezielt mit Angabe der Seriennummer auswählen.

Die Ausgabe der aufgenommenen Daten erfolgt direkt auf der Standardausgabe um sie weiteren Tools zur Verfügung zu stellen. Ein kleines Beispiel, eingegeben auf der Kommandozeile soll dies verdeutlichen:

```
msb_record | msb_format
```

Die vom Analyser aufgenommenen Daten werden mit dem Pipe Operator '|' direkt an das nächste Programm in der Kommandokette übergeben[1]. Dieses liest die Daten aus der Pipe, verarbeitet sie und gibt das Ergebnis seinerseits wieder auf der Standardausgabe aus. In diesem Fall - ohne weitere Programm Parameter - einfach als informelle Liste.

```
1 3.501328 A "104 0x68 'h' "
2 3.501414 A "101 0x65 'e' "
3 3.501501 A "108 0x6C 'l' "
4 3.501588 A "108 0x6C 'l' "
5 3.501675 A "111 0x6F 'o' "
```

Um das Programm abzubrechen drücken Sie einfach Strg+C.

Im Allgemeinen werden Sie `msb_record` entweder in Kombination mit einem anderen Tool verwenden oder die aufgenommenen Daten direkt in eine Datei schreiben. Letzteres können Sie auf zwei Arten erreichen. Entweder indem Sie die Ausgabe in eine Datei umleiten:

```
msb_record > output.msblog
```

oder eine Datei als Ausgabeziel angeben:

```
msb_record -o output.msblog
```

23.2.1 Verbindungsparameter und Ereignisse

Wir haben das Rekord Tool bisher ohne Berücksichtigung der Verbindungseigenschaften aufgerufen und stillschweigend die Standardeinstellungen verwendet. Letztere zeichnet zwar alle übertragenen Daten auf, nicht aber die Pegelveränderungen der einzelnen Leitungen.
Angenommen Sie haben eine Verbindung mit 38400 Baud, 7 Datenbits und einer geraden Parität. Die Anzahl der Stopbits ist für den Analyser unerheblich, der Vollständigkeit halber nehmen wir hier 2 Stopbits an.
Neben den reinen Datenbytes wollen Sie außerdem die Pegelwechsel der Datenleitungen (hier RxD, TxD) sowie der Handshakeleitungen RTS und CTS aufzeichnen. Der Aufruf des Rekorders ist dann:

```
msb_record --baudrate=38400 --protocol=7E2 --logsignals=2,3,6,7
```

oder in kurzer Schreibweise:

```
msb_record -b 38400 -p 7E2 -l 2,3,6,7
```

Wir werden später eine weitere Möglichkeit zur Parameter Übergabe mittels einer Konfigurationsdatei kennenlernen. Wichtig an dieser Stelle:
Alle Programmparameter müssen VOR einem folgenden Pipe Operator '|' stehen. D.h.:

```
msb_record -b 38400 -p 7E2 -l 2,3,6,7 | msb_format
```

Dies gilt für alle Programme.

[1] Sie können `msb_record` natürlich auch alleine aufrufen. Da es sich bei den ausgegebenen Daten um binäre Werte handelt macht dies aber wenig Sinn.

23.2.2 Einbindung in eigene Applikationen

Das ist alles ganz schön, aber wie binden Sie den Analyser nun in Ihrer eigenen Anwendung ein?
Folgenden Voraussetzungen müssen dazu erfüllt sein:

1. Aufruf eines beliebigen Kommandos aus Ihrer Anwendung.
2. Einlesen einer von einem anderen Prozess geöffneten Datei.

Das klingt zunächst kompliziert, ist es aber nicht.
Starten Sie einfach die von Ihnen gewünschte Verarbeitungskette mit dem entsprechenden Befehl (Systemaufruf) der von Ihnen verwendeten Anwendung bzw. Programmiersprache. In C zum Beispiel die Funktionen `system` oder `popen`, LabView bietet hierfür das System Exev VI.

Beim Aufruf externer Kommandos gibt es i.a. zwei Möglichkeiten. Der Aufrufer 'wartet' auf das Ende des Kommandos oder das Kommando (der Prozess) läuft entkoppelt und unabhängig von dem Aufrufer. Letzteres ist wichtig wenn Ihre Anwendung nicht auf das Ausführungsende der Verarbeitungskette warten möchte (und damit blockiert ist).

Sobald der Rekorder von Ihrer Anwendung gestartet wurde und die aufgenommenen Daten mit dem von Ihnen gewünschten Format in eine Datei schreibt, können Sie damit beginnen aus die gewünschten Informationen aus letzterer zu lesen. Dies kann z.B. zeilenweise geschehen.
Je nach Programmiersprache oder Anwendung können Sie die Ausgabe der Verarbeitungskette auch ohne den Umweg über das Schreiben in eine Datei direkt in Ihre Applikation einlesen.

Im allgemeinen wird die gestartete Verarbeitungskette oder Befehlsfolge automatisch beendet, wenn Sie Ihre Anwendung schließen.
Es gibt aber noch eine weitere Möglichkeit, den laufenden Rekorder zu beeinflussen.

23.2.3 Remote Kommandos

Das `msb_record` Tool enthält zur Kommunikation mit anderen Anwendungen eine einfache und leicht umzusetzende Interprozess-Kommunikation die für beide Plattformen (Linux und Windows) gleichermassen funktioniert.
Um einem laufenden Rekord Programm ein Kommando zu übermitteln reicht es aus, `msb_record` erneut mit dem Parameter '-r Kommando' aufzurufen (siehe Programm Parameter), bzw. aus Ihrer Anwendung den entsprechenden Befehl per Systemaufruf aufzuführen.

Öffnen Sie dazu einfach zwei Konsolen (Kommandozeileneingaben) und starten Sie in einer eine Aufnahme mit dem Kommando:

```
msb_record | msb_format
```

Der angeschlossene MSB-Analyser wird initialisiert und anschliessend automatisch die Aufzeichnung gestartet, signalisiert u.a. durch ein permanentes Leuchten der roten LED 1. Es spielt dabei keine Rolle ob am Analyser gerade Daten anliegen.

Wechseln Sie nun in die zweite Konsole und stoppen Sie die Aufzeichnung mit:

```
msb_record -r stop
```

Das die laufende Aufzeichnung wirklich gestoppt wurde erkennen Sie an den erneut wechselseitig blinkenden roten LEDs des Analysers.
Um die Aufzeichnung fortzusetzen, wiederholen Sie den Aufruf, nun allerdings mit dem Kommando 'start':

```
msb_record -r start
```

Das Kommando `msb_record -r quit` beendet das `msb_record` Programm bzw. die gesamte Verarbeitungskette und schließt die Datei mit den bislang gespeicherten Daten.

23.2.4 Synchrones Aufzeichnen mit zwei oder mehreren Analyser

Jeder MSB-Analyser verfügt über eine sogenannte Link MSB Buchse, mittels derer die Aufnahmen von zwei oder sogar sogar mehreren Geräten (bei Verwendung des IFTOOLS MSB Port-Link Doublers) mit einer Genauigkeit von einer Mikrosekunde synchronisiert werden können. Wir haben diesen speziellen Anwendungsfall und seine Vorteile im Detail in Kapitel 22 beschrieben.
Synchronisierte Aufzeichnungen sind aber auch mit den Kommandozeilen Tools möglich. Wie das geht erklärt der folgende Abschnitt.

Nehmen wir an Sie haben zwei MSB-Analyser und beide sind über die MSB Link Buchsen verbunden. Wenn Sie die grafische Oberfläche verwenden starten Sie zunächst eine Anwendung für jeden Analyser. In einem zweiten Schritt wählen Sie einen von beiden als 'Master'. Der zweite Analyser wird dadurch automatisch zum 'Slave'. Eine korrekte Einstellung vorausgesetzt können Sie anschließend die Aufnahme in der 'Master' Anwendung starten.

Kommandozeilen Tools arbeiten von Natur aus anders.
Das Programm `msb_record` besitzt weder einen Dialog zur Auswahl von 'Master' oder 'Slave' noch einen Knopf um die Aufzeichnung zu starten wenn beide Analyser verbunden und betriebsbereit sind. Sie müssen deshalb dem jeweiligen Analyser per Programm Parameter mitteilen, ob er als 'Master' oder 'Slave' arbeiten soll. Und da Sie mehrere Analyser mit Ihrem PC verbunden haben müssen Sie zusätzlich die Serienummer mit angeben.
Beide Kommandos - für den 'Master' als auch den 'Slave' - werden in zwei unabhängigen Programmterminals bzw. DOS Eingabefenstern ausgeführt.

Als erstes geben wir das Aufnahmekommando für den 'Master' mit der Seriennummer MSB01000 ein. Der Einfach halber verwenden wir die Voreinstellungen und leiten die Ausgabe direkt in das Formatierung-Tool weiter. Bitte denken Sie daran, zuvor die Seriennummer an Ihr eigenes Gerät anzupassen.

```
msb_record -nMSB01000 --sync-mode=master | msb_format
```

Sobald Sie das Kommando mit der Enter Taste bestätigen fordert Sie das Programm auf, ZUERST den 'Slave' zu starten und DANN die Enter Taste erneut

zu drücken um die Aufzeichnung zu starten. Was bedeutet das denn?

Ein synchrone Aufzeichnung setzt den Austausch bestimmter Informationen zwischen den beiden MSB-Analyser voraus. Insbesondere Zeit relevante Daten. Lassen Sie uns deshalb ein zweites Terminal- bzw. DOS Kommandoeingabefenster öffnen und den 'Slave' starten mit:

```
msb_record -nMSB02000 --sync-mode=slave | msb_format
```

Nochmals: Die Seriennummer hier ist nur ein Platzhalter und Sie müssen sie mit der S/N Ihres zweiten Gerätes ersetzen!
Beide Geräte sind nun aufnahmebereit, angezeigt durch das wechselseitige Blinken der roten LEDs. Aber noch wichtiger! Der 'Slave' ist aktiv und wartet nur noch auf die Timing Daten und das Startkommando des Master Gerätes.
Wenn alles soweit eingerichtet ist, dann drücken Sie jetzt die Enter Taste im Master Eingabefenster.
Die folgenden zwei Dinge passieren:

1 Der Master überträgt seine korrekte Startzeit an den Slave[2].

2 Der Master sendet das Start Kommando und versorgt im weiteren Verlauf den Slave periodisch mit Synchron-Impulsen über das Link Kabel.

Danach laufen beide Kommandos (in beiden Eingabefenstern) unabhängig voneinander und verhalten sich nicht anders als eine normale (nicht synchrone) Aufnahme. Sie können das Kommando beliebig mit Parametern oder zusätzlichen Tools erweitern, die Ausgabe in eine Datei schreiben oder in mehrere Dateien aufsplitten.
Drücken Sie Strg+C in jedem Kommandofenster um die Master bzw. Slave Aufnahme zu beenden.

23.2.5 Eine synchrone Aufzeichnung fernsteuern

Die Durchführung einer synchronen Aufzeichnung per Kommandozeile mag für kleine oder seltene Anwendungen ausreichen. Davon abgesehen können Sie die Aufzeichnung natürlich auch per Skript starten. Genau hier lauert aber ein kleines Problem. Wie können Sie die vom Master erwartete Enter Taste eingeben wenn das Kommando innerhalb einer Batchdatei oder eines Skriptes ausgeführt wird?

Sie können - natürlich - eine Prozess Pipe verwenden und die Enter Taste an den Master Prozess umleiten. Dies ist aber nicht ganz trivial. Zudem gibt es zum Glück noch eine einfachere Lösung: Das Starten der Aufnahme per Fernsteuerkommando:
Zunächst eine kleine Batchdatei für Windows Anwender:

[2]Denken Sie daran: Master und Slave müssen nicht zwangsläufig auf dem gleichen Computer laufen.

```
1 rem Synchronous record
2 echo "{} Initiate master..."
3 start msb_record.exe -i -nMSB01000 --sync-mode=master --paused
    ↪ -o master.msblog
4 timeout 2 >nul
5 echo "{} Initiate slave..."
6 start msb_record.exe -i -nMSB02000 --sync-mode=slave -o
    ↪ slave.msblog
7 timeout 2 >nul
8 echo "Start synchronous record..."
9 msb_record.exe -r start
```

Und hier die Linux Variante:

```
1 #!/bin/bash
2 echo "Initiate master..."
3 msb_record -i -nMSB01000 --sync-mode=master --paused
    ↪ 2>>/dev/null -o master.msblog &
4 sleep 2
5 echo "Initiate slave..."
6 msb_record -i -nMSB02000 --sync-mode=slave 2>>/dev/null -o
    ↪ slave.msblog &
7 sleep 2
8 echo "Start record"
9 msb_record -r start
```

Die Vorgehensweise ist für beide Betriebssysteme ähnlich.
Als erstes starten wie einen Hintergrundprozess für den Master (Zeile 3) und weisen ihn an, auf ein Start Kommando zu warten, indem wir den Parameter `--paused` anfügen.
Windows (bzw. die DOS Kommandoeingabe) verwendet das `start` Kommando, während Linux Anwender einfach ein Kaufmanns-Und '&' an die Kommandozeile anhängen um diese in den Hintergrund zu versetzen. In Linux leiten wir zudem die stderr (2) Ausgabe nach /dev/null um.

Ein im Hintergrund laufender Prozess bedeutet: Die Kommandozeile wird losgelöst vom dem Skript ausführenden Terminal (oder DOS Fenster) gestartet. Das Kommando kann deshalb nicht blockieren und die Ausführung wird unmittelbar mit der nächsten Skriptzeile fortgesetzt.
Zeile 4 (und 7) stellen der Programm Initialisierung ein paar Sekunden zur Verfügung. Die DOS Kommandoeingabe besitzt keinen entsprechendes 'sleep' Befehl, aber `timeout` erfüllt denselben Zweck[3].
Der 'Slave' wird in Zeile 6 gestartet und ebenfalls als Hintergrundprozess ausgeführt.

Zu diesem Zeitpunkt sind beide Analyser aufnahmebereit und der 'Master' wartet lediglich auf den Auslöser. Anstelle einer gedrückten Enter Taste (unmöglich, da der Prozess von jeglicher Tastatur entkoppelt ist) senden wir ihm in Zeile 9 per 'Fernsteuerung' ein 'start' Kommando.
Beide Aufnahmeprozesse (Master und Slave) laufen anschließend völlig un-

[3]timeout ist unter XP nicht verfügbar. Eine Alternative um eine Pause von 2s zu simulieren ist das Kommando: ping 127.0.0.1 -n 3 >null

abhängig voneinander. Der Master schreibt seine Daten in die per `-o` angegebene Ausgabedatei master.msblog. Der Slave speichert seine Daten in slave.msblog.
Die interne Synchronisation der MSB-Analyser durch das per MSB Link verbundene Kabel garantiert, dass die aufgenommenen Ereignisse in beiden Aufzeichnungen zeitlich mit einer Genauigkeit von einer Mikrosekunde übereinstimmen.

23.2.6 msb_record Programm Parameter

Der Aufruf des Programmes erfolgt mit:

```
msb_record [OPTION]...
```

`[OPTION]` kann ein oder mehrere der folgenden Programm Parameter enthalten. Wenn keine Parameter angegeben werden, werden die Programmvorgaben verwendet, die Ausgabe erfolgt auf der Standardausgabe.
Um einem laufenden Hintergrundprozess ein Kommando zu schicken, reicht die Angabe des Kommandos in Verbindung mit dem Remote Parameter '-r' aus.

Argument	Beschreibung
--baudrate=*rate*	OBSOLETE! Bitte verwenden Sie in Zukunft den Parameter `-b` oder `--bitrate`, siehe unten!
-b *rate* --bitrate=*rate*	Bitrate der aufzunehmenden Verbindung.
--bit-order	Bitfolge: MSB (höherwertiges Bit) zuerst=1 (Vorgabe), MSB zuletzt=0, Parameter nur für Manchester Übertragungen sinnvoll.
--clock-delay	Verzögertes Abtasten der Datenbits um einen $\frac{1}{2}$ Takt bei langen Leitungen die an der Taktgrenze sind. Vorgabe ist aus.
-c *file* --config-file=*file*	verwende Einstellungen aus der angegebenen Konfigurationsdatei.
-C --create-config-file	erzeugt die Konfigurationsdatei `msb_tools.config` im aktuellen Verzeichnis.
--disable-dataA	Schaltet die Aufzeichnung der Daten an Port 1 aus. Vorgabe ist ein.
--disable-dataB	Schaltet die Aufzeichnung der Daten an Port 2 aus. Vorgabe ist ein.
--doubled-frame	Verifizierung der Daten durch doppelte Übertragung des Datenframes. Selten verwendet, nur SSI. Vorgabe ist aus.

--frame-bits=*Bits*	Bit Anzahl oder Takte innerhalb eines Daten Rahmens (5...63). Notwendig bei der Aufzeichnung synchroner Bus Systeme.
-h --help	Ausgabe aller verfügbaren Programmparameter.
-i --initiate	Übertragen der Firmware auch wenn sie bereits geladen ist.
--io1=*operation*	Verwendung des Hilfskanals IO1 (nur MSB-RS485 und MSB-RS485-PLUS). Die möglichen Werte sind abhängig vom verwendeten Analyser Typ und der eingestellten Bus Übertragungsart. Die nachfolgende Tabelle im Abschnitt 23.2.6.1 zeigt alle gültigen Varianten. Beispiel (Ausgabe Bus Richtung): `msb_record --io1=3`
--io2=*operation*	Verwendung des Hilfskanals IO2 (nur MSB-RS485). Gültige Einstellwerte siehe 23.2.6.1. Beispiel (Ausgabe Bus Gültigkeit): `msb_record --io2=4`
-l *list* --log-signals=*list*	Leitungssignale, die vom Analyser aufgezeichnet werden sollen. Die Leitungen werden von 1...8 durchnummeriert so wie sie im Display Kontrollprogramm angezeigt werden. Z.B. -l 2,3 oder --log-signals=2,3,6,7.
-L --logic-mode	Schaltet die Eingänge auf Logiksignalpegel um (nur MSB-RS232), Vorgabe sind RS232 Signalpegel.
--memory-test	führt einen interen Speichertest des angeschlossenen Analysers aus.
--nice=*niceness*	Der nice Parameter steuert den Verbrauch der CPU Leerlauf Zeit. Gültige Werte sind 0...10. Ein Wert von 0 bedeutet annähernd 100% CPU (Last) Zeit, die Voreinstellung ist 1. Ein Wert von 0 ist nur bei sehr hoher Datenrate und Datenlast zu empfehlen. Z.B. mit `msb_record --nice=0`
-n *serno* --serno=*serno*	Verwendet den Analyser mit der angegebenen Seriennummer serno.

--output-buffering	Aktiviert die interne Ausgabepufferung. Die vom Analyzer aufgenommenen Ereignisse werden dabei gepuffert und nicht sofort in den Ausgabekanal geschrieben. Dies erhöht die Performance vor allem bei Verbindungen mit hohen Datentransraten und vermeidet Lücken in der Aufzeichnung. Beachten Sie, das bei aktivierter Pufferung die Ereignisse nicht sofort in das nächste Glied der Kommandokette gereicht werden, z.B. wenn Sie die aufgenommenen Daten in einer Konsole per msb_format ausgeben.
-o *file* --output-file=file	Datei, in welche die Ausgabe erfolgen soll. Default ist die Standardausgabe (Konsole).
--paused	Startet den verbundenen Analyser im Pause Modus. Die Aufnahme startet erst nach Empfang eine remote start Kommandos.
-p *protocol* --protocol=*protocol*	Protokoll der aufzunehmenden Verbindung als Kommbination aus Anzahl Datenbit (5...9), Parität (N)one, (E)ven, (O)dd, (0)off, (1)on, und Stopbit (1,2). Z.B. 8N1 oder 7E2.
-r *command* --remote=*command*	sendet einem bereits laufenden Programm den angeführten Befehl. Folgende Kommandos werden unterstützt: `quit` beendet Hintergrundprozess `start` setzt eine zuvor gestoppte Aufnahme fort `stop` stoppt die Aufzeichnung.
--show-analyzers	Zeigt alle am PC angeschlossenen MSB-Analyser.
--sync-mode=*mode*	Wählt die Betriebsart (autonom, master, slave) des Analysers für synchrone Aufzeichnungen. Die Vorgabe ist autonom.
-t *delay* --time-delay=*delay*	Verlangsamt den Transfer der Firmware um die angegebenen Zahl (Vorgabe ist 0, d.h. keine Verzögerung, bis maximal 100.
--transmission=*mode*	Wählt die Übertragungs-Art des angeschlossenen Bus Systems. Asynchrone, synchrone SSI oder Manchester Bus Systeme. Die möglichen `mode` Parameter sind abhängig vom verwendeten Analyser, siehe Abschnitt 23.2.6.2. Beispiel: `msb_record --transmission=sync-ssi`

--trigger-source=*src*	Wählt dir Trigger Quelle der Datenrahmen und Datenfehler Ausgabe für die entsprechende IO Hilfskanal Einstellung, siehe Parameter `--io1` und `--io2`. Mögliche Einstellungen für *src* sind: `A`, `B` und `A+B` (Vorgabe).
-u --unique-file	Speichert die aufgenommenen Daten im aktuellen Verzeichnis in einer Datei mit eindeutigem Dateinamen der Form `YYYYMMDD-HHmMMmSSs.msblog`, z.B. `20110324-03h04m41s.msblog`. Dieser Parameter ist vor allem bei einem automatischen Aufzeichnungsstart nach (Re)boot des Rechners interessant.
-v --verbose	Verbose, Ausgabe zusätzlicher Informationen.
-V --version	Ausgabe der Programmversion.
-w *wiring* --wiring=*wiring*	Setz den Bus Anschluss wiring (nur MSB-RS485). Folgende Werte sind erlaubt: 0 : 2-Draht Abgriff 1 : 2-Draht Segment 2 : 4-Draht Abgriff 3 : 4-Draht Segment

23.2.6.1 Digitale IO Einstellungen

Die verfügbaren Werte zur Konfiguration der beiden digitalen IO Hilfs- Ein/-Ausgänge hängt von der gewählten Übertragungsart und dem verwendeten Analyser Typ ab. Die folgende Tabelle zeigt alle Varianten. Die erste Spalte repräsentiert dabei die dem entsprechenden `--io`$[1,2]$ Parameter zuzuweisende Zahl.

Number	Description	Transmission	Analyzer
0	Eingang mit Pull Down	alle	MSB-RS485 MSB-RS485-PLUS
1	Ausgang: Statisch 0	alle	MSB-RS485, MSB-RS485-PLUS
2	Ausgang: Statisch 1	alle	MSB-RS485, MSB-RS485-PLUS
3	Ausgang: Bus Richtung	alle	MSB-RS485, MSB-RS485-PLUS
4	Ausgang: Bus gültig	alle	MSB-RS485, MSB-RS485-PLUS
5	Ausgang: CHN1 gültig	alle	MSB-RS485, MSB-RS485-PLUS

6	Ausgang: CHN2 gültig	alle	MSB-RS485, MSB-RS485-PLUS
7	Ausgang: CHN3 gültig	alle	MSB-RS485, MSB-RS485-PLUS
8	Ausgang: CHN4 gültig	alle	MSB-RS485, MSB-RS485-PLUS
9	Eingang mit Pull Up	alle	MSB-RS485, MSB-RS485-PLUS
10	Fehlerausgabe (Parity, Frame) bei asynchronen Bus Systemen unabhängig von Datenkanal	alle	MSB-RS485, MSB-RS485-PLUS
11	Ausgang +5V/50mA	alle	MSB-RS485-PLUS
12	Fehlerausgabe synchrone Busse wobei IO1 Bus A und IO2 Bus B repräsentiert	Synchron SSI Manchester	MSB-RS485-PLUS
13	Ausgabe Datenrahmen Signal, dabei ist IO1 Bus A und IO2 Bus B zugeordnet	Synchron SSI Manchester	MSB-RS485-PLUS

23.2.6.2 Transmission Parameter

Die unterstützten Feldbus Übertragungsarten hängen vom verwendeten Analyser ab. Die älteren Modelle MSB-RS232 und MSB-RS485 erlauben lediglich die Analyse von asynchronen Übertragungen, während die neuen PLUS Typen auch synchrone und Manchester kodierte Übertragungen aufzeichnen und auswerten können. Die folgende Tabelle gibt Ihnen einen Überblick der erlaubten Werte für den `transmission` Programm Parameter:

Parameter	Transmission	Analyser
`async`	Asynchrone Übertragung	alle
`sync-ssi`	Synchrone SSI Übertragung	MSB-RS232-PLUS, MSB-RS485-PLUS
`manchester-1`	Manchester I (G.E. Thomas)	MSB-RS485-PLUS
`manchester-2`	Manchester I (IEEE 802.3)	MSB-RS485-PLUS
`manchester-t0`	Differential Manchester T0	MSB-RS485-PLUS
`manchester-t1`	Differential Manchester T1	MSB-RS485-PLUS

23.3 Formatierte Ausgabe mit msb_format

`msb_format` erlaubt die Formatierung der Analyserdaten nach Ihren Anforderung, beispielsweise als CSV (Komma separierte Liste). Ohne Parameter erhalten Sie eine Liste der aufgetretenen Ereignis mit Angaben zum Zeitpunkt, Ereignistyp, Datenbyte oder Leitungsstatus. Dies entspricht dem Formatbezeichner 'l' und ist die Voreinstellung.

Sie können dies jederzeit ändern, indem Sie dem Programm eine eigenene 'Formatanweisung' per Parameter `-F` bzw. `--format=` übergeben. Alle auf diesen folgende Zeichen werden als Formatdefinition gewertet. Ein Leerzeichen bzw. generell alle soge-

nannten 'Whitespace' Zeichen wie Tabs oder Enter beenden den Formatstring. Wenn Sie ein Leerzeichen als Teil der Ausgabe definieren wollen, müssen Sie dies quoten. Wie das geht erfahren Sie im Abschnitt 23.3.1.
Wir beschränken uns hier zunächst auf den einfachen Fall und wollen lediglich das Datenbytes nebst zugehörigem Zeitstempel ausgeben. Dabei soll in jeder Zeile die Zeit in Sekunden und das Datenbyte stehen, getrennt durch ein Komma. Der entsprechende Formatstring[4] lautet: `T,B`

Wir werden im folgenden eine Aufzeichnungsdatei als Datenquelle verwenden. Damit brauchen Sie für die folgenden Beispiele keinen angeschlossenen Analyser. Unabhängig von der Datenquelle ist die Arbeitweise identisch und gilt so auch für Aufnahmen mit `msb_record`.
Öffnen Sie (wieder) ein Kommandofenster und wechseln Sie in das DataView Verzeichnis (i.a. `msb-VERSION/examples/DataView`). Geben Sie anschließend folgenden Befehl ein:

```
type modbus-ascii.msblog | msb_format -FT,B
```

Die Ausgabe sieht wie folgt aus:

```
...
5633.304127,48
5633.305162,70
5633.306197,57
5633.307226,13
5633.308261,10
```

In der Ausgabe kann jederzeit zwischen einer ASCII Darstellung und einer binären Ausgabe gewechselt werden. Letztere ist vor allem dann sinnvoll, wenn Sie die Daten in eine Datei schreiben und durch eine andere Applikation einlesen wollen. Eine umständliche Konvertierung der ASCII Darstellung in native Programmtypen wie double oder integer kann dadurch entfallen.
Der Formatbezeichner % aktiviert die binäre Ausgabe während @ wieder auf ASCII (die Voreinstellung) zurück schaltet. Beispielsweise gibt der folgende Aufruf alle Zeichen in ihrem eigentlichen Wert aus.

```
type modbus-ascii.msblog | msb_format -FT,%B@
```

Beachten Sie, dass bei der binären Ausgabe kein Zeilenumbruch erfolgt. In diesem Beispiel schalten wir deshalb nach der binären Ausgabe des Datenbytes wieder in den ASCII Modus zurück.

```
...
5633.304127,0
5633.305162,F
5633.306197,9
5633.307226,
5633.308261,
```

Zeilenumbruch im ASCII Modus abschalten

Um die Ausgabe im ASCII Modus lesbarer zu machen wird am Ende immer ein Zeilenumbruch angehängt. Sie können dieses Verhalten aber deaktivieren, indem Sie das Programm mit dem Parameter —disable-linefeed aufrufen, oder den Formatstring mit % (Binärmodus) beenden.

[4] Eine Auflistung aller Formatbezeichner finden Sie in der Tabelle Format Parameter.

23.3.1 Ausgabe mit beliebigen Zeichen

Sie wollen per Formatstring ein nicht druckbares Zeichen einfügen, oder ein vom System unabhängigen Zeilenumbruch festlegen[5].
Verwenden Sie den Formatbezeichner *#ddd* um ein beliebiges Zeichen zu definieren, was an Stelle des Bezeichners ausgegeben werden soll. Um die Ausgabe der einzelnen Werte durch ein Tabulator zu trennen, geben Sie diesen als Dezimalwert 9 ein. Beispielsweise:

```
type modbus-ascii.msblog | msb_format -FT#009B#009S
```

Oder: Separieren der einzelnen Werte durch jeweils ein Leerzeichen (Space, Dezimalcode 32):

```
type modbus-ascii.msblog | msb_format -FT#032#032S
```

Um unter Windows einen Zeilenumbruch mit einem einzelnen Linefeed zu realisieren, geben Sie diesen als Dezimalwert (010) vor und Aktivieren zur Unterbrückung der im ASCII Modus voreingestellten systemabhängigen Zeilenumbruchsequenz am Schluss den Binärmodus:

```
type modbus-ascii.msblog | msb_format -FT#009B#009S#010%
```

Der Zeichenwert muss immer mit drei Dezimalstellen (0...9) erfolgen. Abweichende Stellenanzahl oder falsche Eingaben in Form von ungültigen Dezimalzahlen führen zu einer Fehlermeldung.

23.3.2 Ausgabe in Datei

Sie können die Ausgabe jederzeit in eine Datei umlenken. Rufen Sie dazu das Programm mit dem zusätzlichen Parameter '-o Dateiname' auf.
In der Datei landen nur die per Formatstring definierten Ausgaben, keinerlei Statusmeldungen oder Zusatzinformationen die Sie eventuell per Programmparameter angegeben haben.
Eine einfache Ausgabe in eine Datei erfolgt mit:

```
type modbus-ascii.msblog | msb_format -FT#009B#009S#010% -o test.log
```

23.3.3 Format Parameter

Die folgenden Bezeichner sind als Format Parameter definiert. Beachten Sie, dass hier nicht aufgeführte Zeichen genauso ausgegeben werden. Ausnahmen sind Whitespace Zeichen (d.h. alle Leerzeichen, Tabs, Enter), die als Ende des Formatstrings gewertet werden.
(Für ein Leerzeichen in der Ausgabe verwenden Sie #032, die Ausgabe eines Tabulators erreichen Sie mit #009).

Bez.	Bedeutung	Beschreibung
%	Binary Flag	Schaltet für alle folgenden Parameter die binäre Ausgabe ein.
@	Ascii Flag	Schaltet für alle folgenden Parameter die ASCII Ausgabe ein.

[5] Unter Linux werden Zeilen mit einem einfachen Linefeed beendet, während Windows eine Carridge Return/Linefeed Kombination verwendet.

#*ddd* Zeichen		Ausgabe eines beliebigen auch nicht druckbaren Zeichens, angegeben als 3-stelliger Dezimalwert. Erlaubter Wertebereich ist 0...255. Z.B. Zeilenende Carridge Return mit #013.
[...]	[*format*]	Ausgabe des Zeitstempels in einem Benutzer definierten Format. Weitere Information hierzu finden Sie im Abschnitt 23.3.4.
a	Änderung	Zeigt die Änderung in den Signalleitungen im Vergleich zum letzten Ereignis. Z.B. bedeutet `+TxD -RTS` ein Wechsel der TxD Leitung auf einen positiven Signalpegel und ein Wechsel der RTS Leitung auf einen negativen Signalpegel.
A	Alle Signalpegel	Status bzw. Veränderung aller Signale in einem repräsentativen Textformat wie im Ereignismonitor dargestellt. Z.B.: `-^DCD,^^TxD,^^RxD,^^DSR,-^DTR,^^CTS,^^RTS,-^RI`
b	Data-Byte	Ausgabe des aufgenommenen Datenbytes als 8 Bit Wert. In ASCII wird der Wert als 2 stellige Hexadezimalzahl mit führenden Nullen dargestellt. Z.B. 41 für das Zeichen 'A'.
B	Data-Byte	Wie 'b', allerdings wird der Wert im ASCII Mode als Dezimalzahl. Z.B. 65 für das Zeichen 'A' oder 10 für Linefeed.
d	Datum/Uhrzeit	Datum und Uhrzeit des aufgetretenen Ereignisses im ISO Format YYYY-MM-DD HH:MM:SS (ASCII Ausgabe). Im Binärmodus erfolgt die Ausgabe als 32 Bit Zahl mit Angabe der verstrichenen Sekunden seit dem 1 Januar 1970, 00:00:00 UTC (Unixzeit).
D	Excel Datum	Datum als Tage seit dem 1.1.1900. Ausgabe als 32 Bit Wert (binär) oder Dezimalzahl (ASCII).
e	Error	Übertragungsfehler. Fehler werden im ASCII Modus als Buchstabe gekennzeichnet: 'F'rame', 'P'arity' oder 'B'reak'. In Binary als 8 Bit Wert (0:kein Fehler, 1:Frame, 2:Parity, 3:Break).
i	Info	Informelle Ausgabe, vor allem zu Testzwecken. Verwenden Sie ausschließlich diesen Parameter, wenn Sie die vom Formater eingelesenen Daten überprüfen wollen.
l	Logik-Pegel	Der aktuelle logische Leitungsstatus. Ein gesetztes Bit entspricht einem logischen Pegel von 1. Die Bitzuordnung entspricht der Signalreihenfolge im Kontrollprogramm. Bit 0 ist das erste (linke) Signal, Bit 7 das letzte (rechte) Signal im Display. Die Ausgabe erfolgt entweder als 8 Bit Wert (Binärmode) oder als 2-stelliger Hexadezimalwert mit führenden Nullen. So bedeutet '7F' das alle Signale mit Ausnahme von Signal 8 einen logischen '1' Pegel aufweisen.

L	Logik-Pegel	Siehe 'l'. Im ASCII Mode erfolgt die Ausgabe als Dezimalwert. So entspricht 127 dem obigen hex 7F und ein Wert von 255 bedeutet einen logischen '1' Pegel für alle Leitungen.
M	Millisekunden	Zeitstempel des Ereignisses in Millisekunden relativ zum aktuellen Tag. Ausgabe als 32 Bit Wert (binär) oder als Dezimalzahl im ASCII Modus.
o	dt zuletzt	Ausgabe der verstrichenen Zeit seit dem letzten Ereignis in Sekunden als Dezimalzahl mit 6 Nachkommastellen oder als 8 Byte Wert (double) im Binärmodus.
O	dt gleichen Typs	Ausgabe der verstrichenen Zeit seit dem letzten GLEICHEN Ereignis in Sekunden als Dezimalzahl mit 6 Nachkommastellen oder als 8 Byte Wert (double) im Binärmodus.
P	Position	Laufender Ereigniszähler beginnend mit 0. Ausgabe binär als 32 Bit Wert, in ASCII als Dezimalzahl.
s	Datenstatus	Die Ausgabe ist abhängig vom Datentyp. Handelt es sich um ein übertragenes Datenbyte enthät es in den unteren 9 Bit den Wert des aufgezeichneten Datenbytes. Die oberen 7 Bits sind 0. Im Falle einer aufgezeichneten Leitungsänderung enthalten die oberen 8 Bit den 'logischen' Zustand der jeweiligen Leitung. Die unteren 8 Bit repräsentieren den Gültigkeitszustand der einzelnen Leitungen. Siehe auch Formatparameter 'l' und 'v' sowie Abschnitt 13.4.2. Daten oder Status werden als 4-stelliger Hexwert mit führenden Nullen im ASCII Modus ausgegeben (z.B. F12E). Im Binärmodus erfolgt die Ausgabe als 16 Bit Wert.
S	Source	Quelle oder Richtung des Datenbytes. Datenkanal A=1, Datenkanal B=2, (0 wenn es sich um kein Datenereignis handelt). Die Ausgabe erfolgt als Dezimalzahl in ASCII oder als 8 Bit Wert im Binärmodus.
t	Ereignistyp	Art des Ereignisses, im ASCII Modus bezeichnet als: A (Daten Port A), B (Daten Port B), L (Änderung des Leitungspegel). Im Binärmodus Ausgabe als 8 Bit Wert im Bereich [0...2].
T	Zeitstempel	Mikrosekunden genauer Zeitstempel des Ereignisses relativ zum Start der Aufzeichnung. Ausgabe in Sekunden als Fliesskommazahl mit 6 Nachkommastellen (ASCII) oder als 8 Byte Fliesskommawert mit doppelter Genauigkeit (double).
u	usec Anteil	Mikrosekundenanteil des Ereigniszeitstempels. Interessant um die Datum/Uhrzeit Ausgabe mit den Mikrosekunden zu ergänzen. So liefert das Format -Fd+u `2012-04-11 15:57:40+184935`. IM Binärmodus erfolgt die Ausgabe als 32 Bit Wert.

v	Valid-Pegel	Der aktuelle Güligkeitsstatus der einzelnen Signalleitungen bitweise kodiert. Leitungen mit einem korrekten Pegel enthalten ein gesetztes Bit. Die Ausgabe erfolgt als 8 Bit wert im binären Modus, oder als 2-stelliger Hexadezimalwert mit führenden Nullen in ASCII. Z.B. 'FF' wenn alle Leitungen einen gültigen Pegel besitzen.
V	Valid-Pegel	Siehe 'v'. Im ASCII Modus erfolgt die Ausgabe als Dezimalwert. Sind alle Leitungen gültig (alle Bits gesetzt) entspricht dies einem '255'.
w	Data-Word	Berücksichtigt bei der Ausgabe der Datenbytes auch 9-Bit Datenübertragungen. Im Binärmodus werden die Datenbytes als 16 Bit Werte geschrieben. Im ASCII Modus erfolgt die Ausgabe als 3 stellige Hexadezimalzahl mit führenden Nullen dargestellt, z.B. 105
W	Data-Word	Siehe 'w'. Die Ausgabe der aufgenommenen Datenbytes in ASCII erfolgt hier als Dezimalzahl, z.B. 261.
x*1...8*	signal level	Ausgabe des Tri-State Signalpegels individueller Signalleitungen. Die Signalnummer 1...8 entspricht der Signalfolge im Kontrollprogramm. Der Signalstatus wird als -1, 0, oder 1 in ASCII bzw. als vorzeichenbehafteter 8 Bit Wert im Binärmodus ausgegeben.

23.3.4 Benutzerdefinierte Datumausgabe

Eine zwischen zwei eckigen Klammern eingeschlossene Zeichenkette wird als spezielle Datum und Uhrzeit Formatanweisung behandelt. Sie erlaubt die Ausgabe von Datum und Uhrzeit des Ereignisstempels individuell nach Ihren Vorgaben. Ein Beispiel:

```
type modbus-ascii.msblog | msb_format -F"[%d.%m.%Y %H:%M:%S],u#115"
```

liefert:

```
27.08.2010 09:41:04,303098s
27.08.2010 09:41:04,304127s
27.08.2010 09:41:04,305162s
27.08.2010 09:41:04,306197s
27.08.2010 09:41:04,307226s
27.08.2010 09:41:04,308261s
...
```

Das `msb_format` Programm unterstützt die nachfolgenden Formatparameter für Anwender spezifische Datum/Zeit Ausgaben. Jeder Parameter wird mit einem % eingeleitet.
Beachten Sie! Bei Verwendung in einer Batch Datei[6] müssen Sie das % doppelt eingeben, da Sie sonst einen 'Invalid format parameter' Fehler erhalten. Der Grund: Das % Zeichen wird von dem Windows Kommando Interpreter intern zur Referenzierung der Skript Argumente benutzt.
Obiges Beispiel:

```
type modbus-ascii.msblog | msb_format -F"[%d.%m.%Y %H:%M:%S],u#115"
```

muss in einer Batch Datei deshalb geändert werden zu:

[6] Dies betrifft ausschließlich Windows

```
type modbus-ascii.msblog | msb_format -F"[%%d.%%m.%%Y %%H:%%M:%%S],u#115"
```

Beachten Sie!

Da der Kommando-Interpreter in Windows wie auch in Linux das Leerzeichen (hier zwischen Datum und Uhrzeit) als Trennzeichen zwischen zwei Parametern interpretiert, müssen Sie den Formatstring 'quoten', d.h. in Gänsefüßchen einschliessen.

Parameter	Beschreibung
%a	abgekürzter Name des Wochentages
%A	ausgeschriebener Name des Wochentages
%b	abgekürzter Name des Monats
%B	ausgeschriebener Name des Monats
%c	Datum und Uhrzeit Repräsentation gemäß den Systemeinstellungen
%d	Tag des Monats als Zahl [01...31]
%e	Monatstag als Dezimalwert mit vorangestelltem Leerzeichen bei einstelligen Werten [' 1'...'31']
%H	Stunde im Bereich [00...23]
%I	Stunde im Bereich [00...12]
%j	Tag des Jahres als Zahl [001...366]
%m	Monat als Zahl [01...12]
%M	Minute als Dezimalwert [00...59]
%p	die Zeit in 'a.m.' oder 'p.m.' Notation
%S	Sekunden als Dezimalwert [00...59]
%T	aktuelle Zeit, genau wie `%H:%M:%S`
%U	Nummer der Woche des aktuellen Jahres als Dezimalwert [00...53], beginnend mit dem ersten Sonntag als erstem Tag der ersten Woche [01]
%w	Wochentag als Zahl [0...6], beginnend mit Sonntag
%W	Nummer der Woche des aktuellen Jahres [00...53], beginnend mit dem ersten Montag als erstem Tag der ersten Woche
%y	Jahr als 2-stellige Dezimalzahl [00...99]
%Y	Jahr als 4-stellige Dezimalzahl inklusive des Jahrhunderts
%Z	Zeitzone, Name oder eine Abkürzung, z.B. CEST
%%	das % Zeichen

23.3.5 msb_format Programm Parameter

Der Aufruf des Programmes erfolgt mit:

```
msb_format [OPTION]...
```

`[OPTION]` kann ein oder mehrere der folgenden Programm Parameter enthalten. Wenn keine Parameter angegeben werden, werden die Programmvorgaben verwendet (Format `-FI`), die Ausgabe erfolgt auf der Standardausgabe.

Argument	Beschreibung
-c *Datei* --config-file=*Datei*	Verwende die angegebene Konfigurationsdatei.
--disable-linefeed	Zeilenumbruch unterdrücken.
-F *Format* --format=*Formatstring*	Ausgabeformat Definition, siehe Format Tabelle.
-h --help	Ausgabe aller verfügbaren Programmparameter.
-o *Datei* --output-file=Datei	Datei, in welche die Ausgabe erfolgen soll. Default ist die Standardausgabe (Konsole).
-s *list* --signal-names=list	Pass a comma separated list of signal names for use in the output. For instance: `--signal-names=DCD,RxD,TxD,DSR,DTR,CTS,RTS,RI` names all signals according to the RS232 DCE standard names.
--signal-rs232-dte	Predefined signal list. Names all signals according to RS232 DTE.
--signal-rs232-dce	Predefined signal list. Names all signals according to RS232 DCE.
--signal-rs485	Predefined signal list for use with the RS485 analyzer. Names all signals as like: `--signal-names=CH1,CH2,CH3,CH4,BDIR,BSIG,IO1,IO2`
-v --verbose	Verbose, Ausgabe zusätzlicher Informationen.
-V --version	Ausgabe der Programmversion.

23.4 Datenausgabe filtern mit msb_filter

In der Regel werden Sie dieses Tool verwenden um aus einer bereits vorhandenen Aufzeichnung (*.msblog) einen bestimmten Bereich, nur bestimmte Ereignisse, oder eine Kombination von beiden zu extrahieren.
Beispielsweise wenn Sie lediglich die übertragenen Datenbytes ohne die mit aufgenommenen Änderungen der Leitungspegel weiter verarbeiten möchten.

Wie alle Tools liest `msb_filter` die zu filternden Daten aus der Standardeingabe und gibt die gefilterten Daten auf der Standardausgabe wieder aus. Es ist entspricht damit einem Filter zwischen Ein- und Ausgabe. Welche Daten durchgeleitet werden, geben

Sie per Programm Parameter an.

```
type recordfile.msblog | msb_filter [Filterparameter] ...
```

Beachten Sie, das Sie zumindest einen Filterparameter angeben müssen, da das Programm nur die Daten durchreicht, die per Parameter frei gegeben wurden[7]. Ohne Angabe eines Filter Parameters werden alle Daten blockiert.

23.4.1 Daten filtern

Die folgende Kommandokette filtert aus der Datei `modbus-ascii.msblog` im `examples/DataView` Verzeichnis alle (an Port A und B) empfangenen Datenbytes und speichert diese als neue Aufzeichnungsdatei
`data-only.msblog`.

```
type modbus-ascii.msblog | msb_filter -A -B > data-only.msblog
```

Genauso gut können Sie die Ausgabe von `msb_filter` zur formatierten Ausgabe direkt an das `msb_format` Programm weiter reichen.

```
type modbus-ascii.msblog | msb_filter -A -B | msb_format
```

23.4.2 Bestimmte Leitungsereignisse herausfiltern

Neben den Daten könnnen Sie auch die aufgezeichneten Leitungsänderungen einzelner Leitungen herausfiltern. Z.B. wenn Sie alle Signaländerungen aufgenommen haben, aber nur an den Daten und den Handshake Leitungen RTS/CTS interessiert sind.
Die Auswahl der von `msb_filter` durchgereichten Signale erfolgt als Komma separierte Liste und entspricht dem `--log-signals` Parameter des Tools `msb_record`.

```
type modbus-ascii.msblog | msb_filter -A -B --pass-signals=6,7 | msb_format
```

Obige Kommandokette extrahiert zusätzlich zu den übertragenen Daten die Signaländerungen der Leitungen 6 und 7 (bei RS232 Verbindungen i.a. RTS und CTS).

23.4.3 Einen Aufzeichnungsabschnitt filtern

Mit dem weiter unten beschriebenen `msb_split` können Sie eine Aufzeichnungsdatei in einzelne 'kleinere Häppchen' zerlegen. Was aber, wenn Sie lediglich an einem ganz bestimmten Ausschnitt der Aufzeichnung interessiert sind? Z.B. an den ersten oder letzten 100000 Ereignissen. Oder an allen Ereignissen innerhalb eines bestimmten Zeitabschnitts?
Das Tool `msb_filter` enthält zwei zusätzliche Parameter zur Definition eines Abschnittes:

1 `--pass-selection=`*pos1,pos2*

pos1 und pos2 spezifizieren die Ereignisnummer des ersten und letzten Ereignisses welches durchgereicht werden soll (pass). Beispiel:

```
type modbus-ascii.msblog | msb_filter --pass-all --pass-selection=300,310
```

2 `--pass-time=`*time1,time2*

time1 und time2 definieren Begin und Ende des Abschnittes in Sekunden. Die Angabe erfolgt als Fliesskommazahl und einer gewohnten Auflösung in Mikrosekunden.

```
type modbus-ascii.msblog | msb_filter --pass-all --pass-time=3.04,3.05
```

[7] Linux Anwender verwenden natürlich statt dem `type` Befehl das `cat` Kommando.

Filter durchgängig schalten

Per Voreinstellung 'blockiert' das Filter Tool alle Daten. Bei der Definition eines bestimmten Bereiches müssen Sie deshalb entweder die durchzureichenden Ereignisse explizit auflisten, oder mit dem Parameter `--pass-all` das Filter für alle Ereignisse komplett 'durchlässig' schalten.

23.4.4 msb_filter Programm Parameter

Der Aufruf des Programmes erfolgt mit:

```
msb_filter [OPTION]...
```

`[OPTION]` kann ein oder mehrere der folgenden Programm Parameter enthalten. Es muss zumindest ein Filter Parameter angegeben werden.

Argument	Beschreibung
-a --pass-all	Keine Filterung, alle Daten und Ereignisse werden durchgereicht.
-A --pass-dataA	Nur die Daten empfangen an Port A (MSB-RS232) bwz. Channel 1 (MSB-RS485) werden durchgereicht.
-B --pass-dataB	Nur die Daten empfangen an Port B (MSB-RS232) bwz. Channel 2 (MSB-RS485) werden durchgereicht.
-c --config-file=*file*	verwende Einstellungen aus der angegebenen Konfigurationsdatei.
-h --help	Ausgabe aller verfügbaren Programmparameter.
--pass-all-signals	Alle Leitungsänderungen dürfen passieren.
--pass-signals=*list*	Liste der Signale deren Änderungen durchgereicht werden sollen. Die Signale sind von 1...8 durchnummeriert, ihre Reihenfolge entspricht der Anzeige im Kontrollprogramm. Z.B. `--pass-signals=2,3,6,7.`
-s --pass-selection=*list*	Durchleitung alle Ereignisse die in dem angegebenen Bereich liegen. Der Bereich ist definiert als erste und letzte Ereignisnummer. Beispielsweise reicht `-s 100,200` oder `--pass-selection=100,200` die Ereignisse ab Position 100 bis 200 durch.
-t --pass-time=*list*	Durchleitung aller Ereignisse die in dem angegebenen Zeitbereich liegen. Der Zeitbereich ist in Sekunden (Auflösung in Mikrosekunden) anzugeben. Das folgende Beispiel `-t 10.257,20.213` leitet alle Ereignisse durch, die in der Zeit zwischen 10.257s und 20.213s nach Beginn der Aufzeichnung aufgetreten sind.
-v --verbose	Verbose, Ausgabe zusätzlicher Informationen.

-V --version	Ausgabe der Programmversion.

23.5 Aufzeichnungen splitten mit msb_split

Bei der Aufzeichnung mit dem MSB-RS485-PLUS Analyser können schnell recht große Datenmengen entstehen. Dies gilt vor allem dann, wenn auf der Suche nach Fehlern in einer Datenkommunikation der Fehler tagelang nicht auftritt und eine Aufzeichnung im Fifo Modus aus bestimmten Gründen nicht möglich ist.
`msb_split` liest eine Rekorddatei von der Standardeingabe und zerlegt diese in kleinere Aufzeichnungsdateien, wobei Sie Größe und Bezeichnung per Programm Parameter vorgeben können.

23.5.1 Bestehende Rekorddatei splitten

Sie haben eine GByte große Rekorddatei und wollen diese in handlichere Stücke splitten, zumal Sie lediglich an den letzten Ereignissen in der Aufzeichnung interessiert sind.
Öffnen Sie dazu eine Konsole (unter Windows Eingabeaufforderung) und wechseln Sie in das Verzeichnis, welches die Rekorddatei enthält.
Geben Sie folgendes Kommando ein:

```
type record.msblog | msb_split -n1000000
```

Mit `type` wird die angegebene Rekorddatei auf der Standardausgabe ausgegeben und durch den Pipe Operator '|' in die Standardeingabe des `msb_split` Programmes geleitet.

Linux Anwender verwenden statt des `type` Programmes den `cat` Befehl.

Je nach Größe der Ausgangsdatei *record.msblog* teilt `msb_split` diese in mehrere $1000000 * 24 + 3072$ große Dateien. Jede Datei (mit Ausnahme der letzten) enthält 1000000 Ereignisse, je 24 Byte umfassend, plus einen Header von 3072 Bytes. Im aktuellen Verzeichnis befinden sich eine Reihe von neuen `msblog` Dateien der Form:

xaa.msblog, xab.msblog, xac.mbslog, ...

Sie können jede dieser Datei einzeln mit der MSB-RS485-PLUS Analyser Software untersuchen, indem Sie diese einfach in das Programm laden oder diese doppelklicken.
Per Voreinstellung nummeriert das Programm alle Dateien alphabetisch mit einem vorangestellten 'x'. Sie können dieses Verhalten jederzeit durch Angabe eines entsprechenden Programm Parameters ändern. Für eine 3stellige, numerische Numerierung verwenden Sie die Parameter `-a` und `-d` (siehe Programm Parameter unter 23.5.4).

```
type record.msblog | msb_split -a3 -d -n1000000
```

Als Resultat erhalten Sie nun: x000.msblog, x001.msblog, ...

Das führende 'x' können Sie durch einen eigenen Prefix ersetzen, indem Sie diesen als letzten Parameter an den Programmaufruf anhängen. Also:

```
type record.msblog | msb_split -a3 -d -n1000000 Test
```

Die resultierenden Dateien beginnen nun mit: Test000.msblog, Test001.msblog, usw.

Diese Art der Numerierung sagt natürlich wenig über den eventuell wichtigen Zeitabschnitt der einzelnen Dateien aus. Alternativ zu einer alphabetischen oder numerischen Benennung der Dateien können Sie auch Datum und Uhrzeit des Auftretens des ersten Ereignisses in der jeweiligen Splitdatei angegeben. Der Parameter hierfür lautet `-D`.

```
type record.msblog | msb_split -D -n1000000 Projekt-
```

Die erzeugten Dateien haben damit sinngemäß folgende Bezeichnungen:

```
Projekt-20110510_15h53m24s.msblog
Projekt-20110510_15h58m31s.msblog
Projekt-20110510_16h02m10s.msblog
...
```

Wenn Sie keinen Prefix bevorzuge, hängen Sie einfach einen 'leeren' String als letzten (PREFIX) Parameter an:

```
type record.msblog | msb_split -D -n1000000 ""
```

und erhalten:

```
20110510_15h53m24s.msblog
20110510_15h58m31s.msblog
20110510_16h02m10s.msblog
...
```

23.5.2 Laufende Aufzeichnung von `msb_record` splitten

Da das `msb_split` Programm seine Daten aus der Standardeingabe liest, können Sie auch die Ausgabe des `msb_record` Tools als Datenquelle verwenden und die aufgenommenen Ereignisse quasi sofort in kleine Häppchen unterteilen. Dies ist z.B. sinnvoll, wenn Sie sehr lange und vor allem sehr große Aufzeichnungen planen und diese erst später untersuchen müssen.

```
msb_record -b115200 -p8N1  | msb_split -a4 -d -n1000000
```

23.5.3 Nur die letzten N Dateien speichern

Stellen Sie sich folgendes Szenario vor: Sie wollen eine Aufzeichnung jede Stunde. Da Ihr Festplattenplatz limitiert ist, wollen Sie aber gleichzeitig nicht mehr als 10 Aufzeichnungen speichern. D.h. nur die Aufzeichnungen der letzten 10 Stunden. Dies entspricht einer typischen Anwendung bei der Sie einen Fehler suchen, der nur sehr sehr selten in Erscheinung tritt.
Per Voreinstellung generiert das `msb_split` Programm neue Aufzeichnungen solange es läuft und Daten aufgenommen werden. Tritt der Fehler jedoch erst nach Tagen auf, kann die Menge der aufgenommenen Record Dateien zu einem Problem werden.
Mit dem Parameter `--keep-max-files=N` können Sie die Anzahl der vorgehaltenen Aufzeichnungsdateien auf einen bestimmt Wert beschränken. Das folgende Beispiel behält nur die letzten 10 Dateien:

```
msb_record -b115200 -p8N1 | msb_split -D -t3600 --keep-max-files=10
```

Sobald die elfte Datei vom Programm erzeugt wird, wird gleichzeitig die allererste wieder gelöscht. Dies wiederholt sich dann bei jeder weiteren Stunde wenn vom Programm eine neue Aufzeichnung angelegt wird.

23.5.4 msb_split Programm Parameter

Der Aufruf des Programmes erfolgt mit:

```
msb_split [OPTION]... [PREFIX]
```

`[OPTION]` kann ein oder mehrere der folgenden Programm Parameter enthalten. Wenn keine Parameter angegeben werden, werden die Programmvorgaben verwendet.
`[PREFIX]` ist eine optional und frei wählbare Zeichenkette, die dem Dateinamen vorangestellt wird. Vorgabe ist das Zeichen 'x'.
Alle Parameter können in einer kurzen Schreibweise (ein Buchstabe mit einem führenden Bindestrich, erste Zeile) oder in der Langform (zweite Zeile) angegeben werden.

Argument	Beschreibung
-a *Länge*, --suffix-length=*Länge*	Anzahl der zu verwendenden Zeichen für den Numerierungs-Suffix. Vorgabe ist 2 Zeichen.
-c *Datei* --config-file=*Datei*	Verwende die angegebene Konfigurationsdatei.
-d, --numeric-suffix	Die Dateinamen werden numerisch benannt, die Vorgabe ist alphabetisch.
--dir=*directory*	Ausgabe der gesplitteten Dateien in dem angegebenen Verzeichnis.
-D, --date-time-suffix	Die Dateinamen werden mit dem Datum und der Uhrzeit des ersten Ereignisses in der Datei im Format `YYYYMMDD_HHhMMmSSs` versehen.
-h --help	Ausgabe aller verfügbaren Programmparameter.
-k, --keep-max-files=*N*	Nur die letzten N Aufzeichnungsdateien werden vorgehalten. Alle anderen (vorherigen) werden automatisch gelöscht.
-n, --number=*Anzahl*	Anzahl der Ereignisse pro ausgegebener Datei. Jedes Ereignis umfasst 24 Byte.
-t, --split-time=*Sekunden*	Aufgenommenen Daten in neuer Datei ablegen alle angegebene Sekunden.
-v --verbose	Verbose, Ausgabe zusätzlicher Informationen.
-V --version	Ausgabe der Programmversion.

23.6 Eine Aufzeichnung triggern mit msb_trigger

Langzeitaufzeichnungen in Verbindung mit den Kommandozeilen Tools dienen oft dazu, bestimmte und selten auftretende Ereignisse zu finden die zu Fehlern in der Kommunikation führen. Dies können z.B. plötzlich nicht mehr reagierende Busteilnehmner sein, ausgelöst durch ungültige Telegramme oder fehlerhafte Telegramminhalte.
Es ist offensichtlich, dass solche Ereignisse nicht einfach per Suchmuster (Datensequenz) detektiert werden können. Betrachten wir dazu einen Modbus Teilnehmer, der unerwartet (und auch nur einmal alle Stunden oder Tage) mit einer Fehlermeldung antwortet. Obgleich in Modbus Fehlermeldungen in zwei Bytes kodiert werden (Adress

Byte, gefolgt von Fehlernummer), kann diese Sequenz auch innerhalb des Telegramminhalts (Payload) beliebiger anderer Telegramme auftreten. Eine Trigger Bedingung ist aber nur dann wahr, wenn die gesuchten beiden Bytes an Anfang eines (Modbus RTU) Telegramms stehen.

Da es eine Unmenge unterschiedlichster Protokolle gibt verwendet das Tool `msb_trigger` den gleichen Ansatz wie der Protokollmonitor. Ein integrierter Lua Skript Interpreter erlaubt die Formulierung beliebiger - auch sehr komplexer - Trigger Bedingungen, bei der normale 'Suchmechanismen' chancenlos sind.
Das `msb_trigger` Programm folgt dabei den Regeln aller anderen Kommandozeilen Tools. Sie können es zur Triggerung einer aktuellen (laufenden) Aufzeichnung verwenden mit:

```
msb_record | msb_trigger script.lua > record.msblog
```

Oder Sie extrahieren einen für Sie interessanten Abschnitt (spezifiziert durch eine Trigger Bedingung) aus einer bereits vorhandenen Aufzeichnung:

```
type record.msblog | msb_trigger script.lua > result.msblog
```

(Linux Anwender verwenden das `cat` Kommando anstelle von `type`).

Beachten Sie bitte!
Um die Kommandozeilen Beispiele möglichst einfach zu halten verzichten wir auf alle zusätzlichen `msb_record` Parameter.
Sie können die Ausgabe (das Resultat) des `msb_trigger` Programms in andere Tools wie z.B. dem Formater (`msb_format`) oder Splitter (`msb_split`) weiterleiten.
Die Datei `script.lua` enthält die in Lua formulierte Trigger Bedingung als Funktion `trigger()`. Diese arbeitet ähnlich der Funktion `split()` im Protokollmonitor und wir werden diese im folgenden genauer betrachten.

23.6.1 Ein Trigger Skript erstellen/editieren

Sie können ein Trigger Skript mit jedem beliebigen Editor erstellen. Wir empfehlen aber den mit der Analyser Software integrierten Editor zu verwenden.
Dieser bietet nicht nur vorgefertigte Skript Code Gerüste, er erlaubt Ihnen auch das Testen und Ausführen ausgewählter Skriptzeilen direkt im Editor selbst.
Um den Skript Editor zu öffnen, starten Sie zunächst die Analyser Software und wählen im Kontrollprogramm unter 'Ansicht' den Menüpunkt 'Skript Editor'.
Im Editor selbst klicken Sie das 'Neue Datei' Symbol in der Werkzeugleiste (oder drücken Sie einfach Strg + N um den Skript Ersteller zu starten. Dort wählen Sie 'Trigger' als Skript aus.
Sie finden eine ganze Reihe von Beispielen, die Sie per Klick auf das 'Datei Öffnen' Symbol in der Editor Werkzeugleiste oder per Strg + O auswählen können. Die Trigger Skripte befinden sich im 'Trigger' Verzeichnis.

23.6.2 Eine Trigger Bedingung definieren

Standardmäßig leitet das `msb_trigger` Tool alle von der Standardeingabe gelesenen Datenereignisse (generiert von `msb_record` oder aus einer existierenden Aufzeichnung) an die im Skript definierte Lua Funktion `trigger`.

```
1 function trigger( data, intval, dir, alter )
2   -- return true if the trigger condition occurred
3 end
```

Die `trigger` Funktion wird separat für jede Datenrichtung aufgerufen. Da die Daten einzelner Telegramme auf eine Datenrichtung beschränkt sind, macht dies die Datenverarbeitung deutlich einfacher.
Neben den Datenbytes (9 Bit) werden auch die verstrichene Zeit zum vorherigen Datenbyte, die Datenrichtung sowie eine eventuelle Änderung der Datenrichtung an die `trigger()` Funktion übergeben. Nachfolgend die Aufzählung aller Funktionsparameter:

1. data → das aktuelle Datenbyte (9 Bit)
2. interval → (kurz intval), die verstrichene Zeit zum letzten Byte in Sekunden (Mikrosekunden Genauigkeit)
3. direction → (kurz dir), die Richtung oder Quelle (Absender) des aktuellen Datenbytes. 1=Data A, 2=Data B.
4. alternation → (kurz alter), true wenn sich die Datenrichtung seit letzten Aufruf geändert hat

Sie können jeden einzelnen Parameter nach eigenen Vorgaben umbenennen. Wichtig ist nur, das die Reihenfolge der Parameter eingehalten wird. Es ist außerdem erlaubt, unnötige Parameter wegzulassen, sofern danach keine weiteren Parameter verwendet werden. (Die Reihenfolge muss von links nach rechts gewährleistet sein!).

Beginnen wir mit einem einfachen Beispiel. Das folgende Lua Skript löst eine Aufzeichnung aus, sobald in einer Modbus ASCII Übertragung ein Telegramm unvollständig endet. Anstelle eines CR LF (Carriage Return, Line Feed) wird das Telegramm mit einem einzelnen CR beendet.

```
lastByte=-1
function trigger( data )
   if lastByte == 0x0D and data ~= 0x0A then
      -- trigger
      return true
   end
   lastByte = data
   return false
end
```

Über ein innerhalb der `trigger` Funktion verfügbares `event` Objekt können Sie zusätzliche Informationen über das aktuelle Ereignis (Datenbyte oder Leitungsänderung) abrufen. Z.B. den Zeitpunkt (Time Stamp) des Datenbytes sowie die aktuellen Leitungspegel.
Das `event` Modul wird detailiert im Protokollmonitor Kapitel 14.8.3 beschrieben. Es ist besonders nützlich, wenn Sie eine Aufzeichnung bei Auftreten eines bestimmten Leitungspegels oder Änderung eines solchen triggern wollen.
Um Ihnen eine Idee davon zu vermitteln löst das folgende Skript eine Aufzeichnung bei fallender Flanke des DTR Signals aus.
Dabei wird bei jeder detektierten Leitungsänderung der Pegel des DTR Signals mit Hilfe des `event` Moduls ermittelt (mögliche Tri-State Pegel -1, 0 und +1) und mit dem gespeicherten letzten Leitungspegel verglichen.

```
-- the DTR signal number
DTR = 4
-- here we store the last DTR line state
last_dtr = -1
function trigger()
   local dtr = event.level( DTR )
   -- check for a falling edge
   if dtr == -1 and last_dtr == 1 then
      -- trigger
      return true
   end
```

```
12      last_dtr = dtr
13      return false
14  end
```

Da das `msb_trigger` Tool per Default nur Datenereignisse an die `trigger` Funktion übergibt, müssen Sie den verarbeitenden Ereignistyp bzw. die Trigger Quelle per Kommandoparameter auf `--trigger-source=signal` ändern.

```
msb_record | msb_trigger --trigger-source=signal script.lua > record.msblog
```

23.6.3 Pre und Post Trigger

Wie bereits erwähnt liegt der Hauptzweck des `msb_trigger` Tools darin, die Menge der aufgenommenen Daten auf die unmittelbar interessanten Bereiche einer Aufzeichnung zu reduzieren und das Analysieren unnötig großer Datenmengen zu vermeiden.
Wie viele Daten vor und nach einer Trigger Bedingung zur Auswertung benötigt werden hängt vom jeweiligen Anwendungsfall ab. `msb_trigger` erlaubt Ihnen deshalb, beide frei zu definieren. Der Parameter `--pre-trigger` gibt dabei die Anzahl der vorherigen Daten bzw. Ereignisse an, die bei Auslösen des Triggers mit aufgezeichnet werden. Interessant vor allem dann, wenn die Kommunikation vor der Trigger Bedingung herangezogen werden muss.
Der Parameter `--post-trigger` definiert die der Daten/Ereignisse, die nach ausgelöster Triggerung noch folgen (siehe Abschnitt Programm Parameter 23.6.9). Die Aufnahme wird damit nach einer definierten Anzahl beendet und verhindert die Aufzeichnung weiterer, nicht benötigter Daten.

Auch hier ein Beispiel: Nehmen wir an, Sie suchen nach einem Fehler in einer Modbus RTU Kommunikation, bei welcher ein Teilnehmer eine falsche Prüfsumme (CRC16 checksum failure) meldet.
Das folgende Trigger Skript zerlegt den eintreffenden Datenstrom in einzelne Modbus RTU Telegramme, indem es die Inaktiv (Idle) Zeiten zwischen den einzelnen Datenbytes auswertet. Modbus RTU spezifiziert eine Idle Zeit von 3.5 Byte (die Zeit, die zur Übertragung von 3.5 Byte benötigt wird) als Telegramm Trennung. Im Falle einer größeren inaktiven Pause (idle time) enthält die globale Variable `seq` (Zeile 2) das komplette Telegramm.

Modbus RTU verwendet eine 16 Bit CRC16 Prüfsumme die in den letzten beiden Bytes eines Telegramms übertragen werden. Das Lua Skript testet zunächst die Telegrammlänge (Zeile 7) auf mindestens 2 Bytes (um den Zugriff auf nicht vorhandene Daten zu vermeiden), und vergleicht anschließend die letzten beiden Bytes mit der berechneten Prüfsumme. Die Funktion `trigger` liefert **true** wenn die Trigger Bedingung zutrifft, d.h. die empfangene Prüfsumme nicht der berechneten entspricht.

```
1  -- represents the actual telegram
2  seq = ""
3  function trigger( data, intval, dir, alter )
4      if intval > transmission.bytepause( 3.5 ) then
5          -- seq represents the current telegram, test checksum
6          -- read 16 bit ckecksum of current telegram
7          if #seq >= 2 then
8              local cks_is = seq:byte(-1) * 256 + seq:byte(-2)
9              local cks_must = checksum.crc16_modbus( seq:sub(1,-3) )
10             if cks_is ~= cks_must then
11                 return true
12             end
13         end
14         -- start a new telegram sequence
15         seq = ""
```

```
16      end
17      -- add the current data byte to the actual telegram sequence
18      seq = seq..string.char( data )
19      return false;
20  end
```

Sie finden das entsprechende Skript im `examples/API` Verzeichnis.
Modbus RTU Telegramme sind auf eine maximale Länge von 256 Bytes limitiert. In unserem Beispiel wollen wir mindestens 10 Telegramme vor und nach dem Telegramm mit der ungültigen Prüfsumme aufzeichnen. Dies gibt uns eine Pre- und Post-Trigger Anzahl von 2560.

```
msb_record | msb_trigger --pre-triger=2560 --post-trigger=2560 ↩
 script.lua > record.msblog
```

Beachten Sie, das es sich bei obiger Eingabe um **eine** Zeile handelt die nur zur Übersicht hier umgebrochen wird (↩ kennzeichnet den Zeilenumbruch).

23.6.4 Trigger Ausschnitt aus Aufzeichnung extrahieren

Das `msb_trigger` Programm dient nicht nur zur Auslösung einer aktiven Aufzeichnung. Sie können damit genauso gut eine bereits gemachte Aufzeichnung nach bestimmten Ereignissen durchsuchen und das Ergebnis zur späteren Analyse als neue Aufzeichnung abspeichern.
Der Aufruf des `msb_trigger` Tools ist identisch. Sie müssen lediglich das `msb_record` als Datenquelle durch die Ausgabe der entsprechenden Aufzeichnung ersetzen. Also: `type record.msblog` für Windows Anwender oder `cat record.msblog` für Linux User. Im Falle des vorherigen Beispiels (Windows):

```
type modbus-rtu.msblog | msb_trigger --pre-triger=2560 ↩
 --post-trigger=2560 script.lua > record.msblog
```

23.6.5 Eine Aufzeichnung nach bestimmten Ereignissen durchsuchen

Stellen Sie sich vor, Sie wollen wissen ob in einer aktiven Übertragung oder einer bereits gemachten Aufzeichnung falsche Prüfsummen - oder generell irgendwelche Fehler - vorhanden sind. Sie wollen aber keine neue Aufzeichnung analysieren, sondern das Resultat Ihrer Prüfung (Zeit und/oder Nummer des entsprechenden Telegramms) direkt auf dem Bildschirm sehen.
Das `msb_trigger` Tool bietet einen speziellen Parameter `--debug` der Ihnen nicht nur dabei hilft, Ihr Skript zu testen. Er dient zudem als 'Schalter' um die Ausgabe der aufgenommenen Ereignisse im Falle einer Trigger Auslösung zu unterdrücken und damit einen Mix aus eigenen Ausgabeanweisungen und binären Ereignisdaten zu vermeiden.

Kommen wir auf unser Modbus RTU Prüfsummen Beispiel zurück. Wir wollen nun nach Telegrammen mit ungültiger Prüfsumme suchen und das Ergebnis in Form der Telegrammzeit zusammen mit der übertragenen (falschen) und erwarteten CRC16 Prüfsumme ausgeben.
Zunächst die entsprechende `trigger` Funktion:

```
1  -- a Lua string representing the last received data of one channel
2  seq = ""
3  -- contains the time stamp of the first byte of the current telegram
4  ts = 0
5
6  function trigger( data, intval, dir, alter )
7      if alter or intval > transmission.bytepause( 3.5 ) then
8          -- seq represents the current telegram, test checksum
```

```
 9       -- read 16 bit ckecksum of current telegram
10       if #seq >= 2 then
11          local cks_is = seq:byte(-1) * 256 + seq:byte(-2)
12          local cks_must = checksum.crc16_modbus( seq:sub(1,-3) )
13          if cks_is ~= cks_must then
14            print( string.format( "%6f\tis:%04X, must:%04X",
15                                  ts, cks_is, cks_must ) )
16          end
17       end
18       -- start a new telegram sequence
19       seq = ""
20       -- store the telegram time
21       ts = event.time()
22    end
23    -- add the current data byte to the actual telegram sequence
24    seq = seq..string.char( data )
25    -- don't stop parsing
26    return false
27 end
```

Ganz wichtig! Da Sie nicht wollen, das die Verarbeitung der eingehenden Daten bei der ersten Triggerung (falsche Prüfsumme erkannt) aufhört muss die `trigger` Funktion IMMER false zurück geben (Zeile 26). Ansonsten beendet sich das Programm ohne Ausgabe. Der Grund: Der `--debug` Parameter schaltet die normale Ausgabe der empfangenen Daten ab und alle per Lua print Anweisungen gemachten Ausgaben werden nicht mehr vollendet, da das Programm zuvor beendet wird.
Sie können das obige Skript selber testen. Öffnen Sie dazu einfach ein Terminal (Linux) oder Kommando Eingabe/Fenster (Windows) in dem `examples/API` Verzeichnis und geben Sie folgendes ein:

```
type Modbus-RTU-wrong-checksum.msblog | msb_trigger ↩
--debug scan-modbus-wrong-checksum.lua
```

Das Kommando sollte folgende Ausgabe produzieren:

```
727.232679      is:982E, must:972E
904.113558      is:50A5, must:50A6
949.962685      is:528C, must:508C
```

Die verwendete Modbus RTU Aufzeichnung enthält drei Telegramme mit ungültiger Prüfsumme, aufgetreten an den angegebenen Zeiten. Die übertragene und falsche CRC16 Prüfsumme is als 'is' bezeichnet, die erwartete (berechnete) als 'must'. Sie können die Aufzeichnung jederzeit in die Analyzer Software laden und die entsprechenden Telegramme überprüfen.

23.6.6 Ein Skript zur Triggerung und Suche

Bis hierhin haben wir die folgenden Anwendungen kennengelernt:

1. Wie Sie eine Aufzeichnung triggern
2. Wie Sie Trigger Ereignisse aus einer Aufzeichnung extrahieren
3. Wie Sie eine Aufzeichnung nach bestimmten Ereignissen durchsuchen

Der dritte Punkt verwendet dabei die eingebaute Debug Funktionalität des `msb_trigger` Tools. Das Problem hierbei: Code der die Debug Funktion verwendet, d.h. mit `--debug` aufgerufen wird und eigene Ausgaben per Lua print macht, kann nicht zur Triggerung einer Aufzeichnung verwendet werden.
Der Grund: Zunächst wird per `--debug` Parameter die Weitergabe der von der Standardeingabe gelesenen Ereignisse (generiert durch `msb_record` oder Ausgabe einer

gemachten Aufzeichnung) unterdrückt und damit die Generierung einer korrekten Analyzer Aufzeichnungsdatei unterbunden. Stattdessen erfolgt die Ausgabe individueller Information (z.B. durch Lua print Anweisungen) in einem nicht kompatiblen Analyzer Aufzeichnungsformat - was in diesem Fall auch beabsichtigt ist.
Die Konsequenz ist: Zwei unterschiedliche Skripte, eins zur Triggerung und eins zur Suche bzw. Testen. Auch im Falle eher einfacherer Skripte eine doch unbefriedigende Lösung.

Im folgenden Abschnitt zeigen wir Ihnen deshalb, wie Sie diese verschiedenen Anwendungsfälle trotzdem in EINEM Skript realisieren können.
Dreh- und Angelpunkt ist dabei der Programm Parameter `--debug`. Bei Aufruf des `msb_trigger` Tools mit diesem Parameter erfolgt die Ausgabe in einem nicht kompatiblen, individuellen Format. Mit Wissen um diesen Parameter könnten wir im Skript eigene Ausgaben per Lua print (de)aktivieren und den Rückgabewert der `trigger` Funktion (true oder false) steuern.

Genau zu diesem Zweck definiert `msb_trigger` die globale Variable `DEBUG`. Sie ist true bei Programmaufruf mit übergebenem `--debug` Parameter, ansonsten false. Damit ist es recht einfach, Code für beide Anwendungsfälle (Auszeichnungs-Triggerung oder individuelle Ausgabe) in einem Skript zu vereinen. Hier erneut unser Modbus RTU Beispiel:

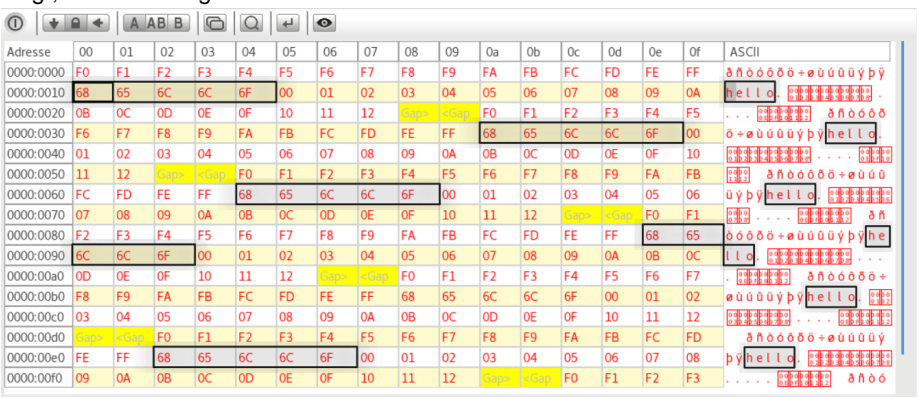
```
1  -- a Lua string representing the last received data of one channel
2  seq = ""
3  -- contains the time stamp of the first byte of the current telegram
4  ts = 0
5  -- the trigger function
6  function trigger( data, intval, dir, alter )
7     if intval > transmission.bytepause( 3.5 ) then
8       -- seq represents the current telegram, test checksum
9       -- read 16 bit ckecksum of current telegram
10      if #seq >= 2 then
11         local cks_is = seq:byte(-1) * 256 + seq:byte(-2)
12         local cks_must = checksum.crc16_modbus( seq:sub(1,-3) )
13         if cks_is ~= cks_must then
14           if DEBUG then
15             print( string.format( "%6f\tis:%04X, must:%04X",
16                                   ts, cks_is, cks_must ) )
17           else
18             return true
19           end
20        end
21      end
22      -- start a new telegram sequence
23      seq = ""
24      -- store the telegram time
25      ts = event.time()
26    end
27    -- add the current data byte to the actual telegram sequence
28    seq = seq..string.char( data )
29    return false;
30 end
```

Die entscheidende Code ist in Zeile 14. Hier prüfen wir bei erkannter Trigger- oder Such-Bedingung, ob das Skript mit dem Programm Parameter `--debug` aufgerufen wurde. Wenn ja, erfolgt die individuelle Ausgabe der Telegrammzeit sowie der empfangenen und erwarteten CRC16 Prüfsumme (Zeile 15). Anschliessend kehrt die Funktion mit **false** zurück (Zeile 29) um eine weitere Verarbeitung der empfangenen Daten zu gewährleisten.

Bei nicht gesetzter `DEBUG` Variable (false) signalisiert die Funktion einfach eine gefundene Trigger Bedingung indem es **true** zurück gibt (Zeile 18). Eine individuelle Ausgabe, die sich mit den eigentlichen Aufzeichnungsdaten vermischen würde findet dadurch nicht statt!
Sie finden das kombinierte Skript wie alle anderen auch im `examples/API` Verzeichnis.

23.6.7 Mehrfaches Triggern

Bislang haben wir uns bei der Triggerung einer Aufzeichnung nur auf einmalige Trigger Ereignisse beschränkt. Sobald die per Trigger Skript formulierte Bedingung eintrat, wurde eine Aufzeichnung mit der gegebenen Anzahl von Pre- und Post-Trigger Ereignissen erzeugt und das Programm bzw. die Verarbeitungskette beendet.
In den meisten Fällen ist dies auch ausreichend. Was aber, wenn Sie mehrere Trigger Ereignisse in einer einzelnen Aufzeichnung sammeln möchten?
Das Programm Argument `--multi-trigger` deaktiviert die einmalige Triggerung. Statt dessen fährt das Tool `msb_trigger` mit der Verarbeitung weiterer Daten fort. Sobald die im Skript definierte Trigger-Bedingung wieder zutrifft, wird erneut die per Pre- und Post-Trigger angegebene Anzahl von Ereignissen in die gleiche Ausgabedatei geschrieben.
Die resultierende Aufzeichnungs-(Record)-Datei enthält nun mehrere Datensegmente mit den jeweils aufgetretenen Trigger Ereignissen. Die einzelnen Datenabschnitt sind dabei durch eine Aufzeichnungslücke definiert durch die Zeit zwischen den Triggerungen und der gewählten Anzahl von Pre- und Post-Trigger Ereignissen getrennt.
Das `msb_trigger` Tool markiert diese Grenzen durch spezielle 'Gap' Ereignisse. Im Datenmonitor werden diese als zwei aufeinander folgende gelben Datenfelder angezeigt, siehe das folgende Bild:



Adresse	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	ASCII
0000:0000	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF	ðñòóôõö÷øùúûüýþÿ
0000:0010	68	65	6C	6C	6F	00	01	02	03	04	05	06	07	08	09	0A	hello.
0000:0020	0B	0C	0D	0E	0F	10	11	12	Gap	Gap	F0	F1	F2	F3	F4	F5	... ðñòóô
0000:0030	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF	68	65	6C	6C	6F	00	ö÷øùúûüýþhello.
0000:0040	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	
0000:0050	11	12	Gap	Gap	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	ðñòóôõö÷øùúû
0000:0060	FC	FD	FE	FF	68	65	6C	6C	6F	00	01	02	03	04	05	06	üýþÿhello.
0000:0070	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	Gap	Gap	F0	F1	ðñ
0000:0080	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF	68	65	òóôõö÷øùúûüýþhe
0000:0090	6C	6C	6F	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	llo.
0000:00a0	0D	0E	0F	10	11	12	Gap	Gap	F0	F1	F2	F3	F4	F5	F6	F7	ðñòóôõö÷
0000:00b0	F8	F9	FA	FB	FC	FD	FE	FF	68	65	6C	6C	6F	00	01	02	øùúûüýþÿhello.
0000:00c0	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	
0000:00d0	Gap	Gap	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	ðñòóôõö÷øùúûüý
0000:00e0	FE	FF	68	65	6C	6C	6F	00	01	02	03	04	05	06	07	08	þÿhello.
0000:00f0	09	0A	0B	0C	0D	0E	0F	10	11	12	Gap	Gap	F0	F1	F2	F3	ðñòó

Die schwarz umrandeten Felder markieren die Trigger Bedingung. Hier wurde nach dem Auftreten der Datensequenz 'hello' gesucht. (Wir haben die schwarzen Umrandungen zum besseren Verständnis eingefügt. In der reallen Aufzeichung existieren diese nicht). Immer zu sehen sind allerdings die gelben Felder, die den Anfang und das Ende des zu einem Trigger Ereignis gehörenden Daten- oder Ereignisbereichs markieren. Alle innerhalb der 'gelben' Lückenmarkierungen liegenden Ereignisse gehören zusammen und sind ein- und derselben Triggerbedingung zugeordnet. Sie können einzelne Triggersequenzen (Abschnitte) jederzeit einfach per Ereignismonitor in separaten Aufzeichnungsdateien speichern.

Ein typischer Aufruf von `msb_trigger` für mehrfaches Triggern sieht wie folgt aus:

```
msb_record | msb_trigger --pre-trigger=100 --post-trigger=100 ↩
--multiple-trigger multitrigger.lua > test.msblog
```

Die Anzahl der Pre- und Post-Trigger Ereignisse ist dabei mit jeweils 100 spezifiziert. Die Trigger Bedingung wird - wie üblich - per Lua Skript angegeben, hier `multitrigger.lua`.

Noch ein Wort zum Trigger Skript selbst.
Im Gegensatz zu einem Skript für einmaliges Triggern müssen Sie bei mehrfachem Triggern dafür sorgen, das Ihre im Skript definierte Trigger Bedingung nach erfolgter 'Triggerung' zurückgesetzt wird. Das folgende Beispiel zeigt das. Es triggert bei einer bestimmten Datensequenz, hier 'hello' für Kanal 1 (A) und 'world' für Kanal 2 (B).

```
1  -- a Lua string representing the last received data of one channel
2  seq = ""
3  -- the trigger function
4  function trigger( data, intval, dir, alter )
5    -- channel A uses "hello", channel B "world"
6    local pattern = {"hello","world"}
7    -- add the data to the stored string/sequence
8    seq = seq..string.char( data )
9    -- limit the size for a better search performance
10   if #seq > 2048 then seq = seq:sub( -32, -1 ) end
11   -- check if the already received bytes contain the given pattern
12   if seq:find( pattern[ dir ] ) then
13     -- MATCHED!
14     -- NOTE! We must clear the sequence buffer for the next multi-
         trigger!
15     seq = ""
16     -- trigger is true
17     return true
18   else
19     return false
20   end
21 end
```

Das Skript bzw. die Funktion `trigger` hängt dabei in Zeile 8 jedes eintreffende Datenbyte an den internen Puffer `seq` an. Anschließend prüft es in Zeile 12, ob der Puffer das gesuchte Muster ('hello' oder 'world', je nach Kanal oder Datenrichtung) enthält. Sobald dies zutrifft, ist die Trigger Bedingung erfüllt.
Bei einmaligem Triggern reicht es, den Wert true zurück zu geben. Betrachten Sie nun aber den Fall, das die Funktion bei mehrfachem Triggern erneut aufgerufen wird. In diesem Fall enthält der interne Puffer `seq` immer noch die bereits zuvor zur Triggerung führende Datensequenz. Die Funktion liefert erneut true obwohl noch keine weitere 'hello' oder 'world' Sequenz empfangen wurde.
Um dies zu verhindern, müssen Sie den Trigger Mechanismus zurücksetzen. Hier bedeutet das: Sie müssen den internen Puffer löschen, bevor Sie true zurück geben. Wir machen dies in Zeile 15.
Generell ist es ratsam, im Skript die Trigger Bedingung bei erfolgter Triggerung korrekt zu reseten, unabhängig davon, ob die das Skript für einmaliges oder mehrfaches Triggern verwenden.

23.6.8 msb_trigger spezifische Lua Erweiterungen

Die folgenden Lua Erweiterungen werden vom `msb_trigger` Programm unterstützt:

- **base16 Modul :** Funktionen zur Codierung und Dekodierung von base16 Sequenzen (i.a. verwendet in Modbus ASCII und Intel SRecord Telegrammen), siehe Lua Erweiterungen, Abschnitt 19.2.1.
- **bit32 module :** OBSOLETE! Mit der Integration der neuen Lua Version 5.3 wird das bit32 Modul nicht länger benötigt und in naher Zukunft entfernt. Benutzen Sie stattdessen jetzt einfach die neuen bitweisen Lua Operatoren.

- **checksum Modul :** Das Checksum Modul enthält fertige Berechnungsalgorithmen für Modbus RTU (CRC16), Modbus ASCII (LRC), BACNet (CRC8 und CRC16), DNP3 sowie CRC16 CCITT (Kermit), siehe Lua Erweiterungen, Anschnitt 19.2.5.
- **event Modul :** Das event Modul ist nur innerhalb der trigger Funktion verfügbar und liefert zusätzliche Informationen zum aktuell empfangenen Datenereignis, siehe Protokollmonitor, Abschnitt 14.8.3.
- **transmission Modul :** Liefert Informationen zur aktuell verwendeten Baudrate, Anzahl der Daten-, Stopbits und Parity, siehe Lua Erweiterungen, Abschnitt 19.2.9.
- **shared module :** Wie der Protokollmonitor verwendet auch das `msb_trigger` Programm zwei unabhängige Lua Interpreter zur Aufsplittung der Telegramme, getrennt nach Datenrichtung. Der Grund für dieses Implementierungsdetail und wie Sie Daten zwischen beiden Interpretern austauschen können erklärt der folgende Abschnitt 14.8.6 im Protokollmonitor Kapitel.

  Alle Module können in gleicher Weise wie im Protokollmonitor verwendet und von überall aufgerufen werden. Die einzige Ausnahme ist das `event` Modul. Dieses ist nur innerhalb der `trigger` Funktion verfügbar.

23.6.9 msb_trigger Programm Parameter

Aufruf des Programms mit: `msb_trigger [OPTION]... trigger-script`

`[OPTION]` kann einen oder mehrere der folgenden Programm Parameter beinhalten. Bei nicht angegebenem Parameter werden die Default Einstellungen verwendet. `trigger-script` ist ein Lua Skript welches die eigentliche Trigger Bedingung formuliert.
Einige der Parameter können in einer Kurzform (einzelnes Zeichen mit vorangestelltem Bindestrich) oder in einer Langform mit zwei vorangestellten Bindestrichen verwendet werden.

Parameter	Beschreibung
-c *Datei* --config-file=*Datei*	Verwende die angegebene Konfigurationsdatei.
-h --help	Ausgabe aller verfügbaren Programmparameter.
--debug	Aktiviert den Debug Modus.
--multi-trigger	Das `msb_trigger` Programm stoppt nicht nach Auftreten der ersten Triggerung sondern fährt mit der Verarbeitung und Suche nach weiteren Trigger Bedingungen fort.
--pass-through	Umgeht alle Trigger Bedingungen. Unabhängig vom ausgeführten Trigger Skript werden alle empfangenen Ereignisse unverändert wieder ausgegeben. Dies ist insbesondere für Skripte interessant, die bestimmten Informationen (Fehler Bedingungen) in der Aufzeichnung z.B. in einer Datei protokollieren wollen ohne die Aufzeichnung zu verändern.
--pre-trigger=*events*	Gibt die Anzahl der Ereignis VOR dem Triggerpunkt an, welche mit ausgegegben werden sollen, sobald eine Trigger-Bedingung eintritt. Die Voreinstellung ist 4096 Ereignisse (Daten sowie Signalereignisse).

--post-trigger=*events*	Gibt die Anzahl der Ereignisse an, die NACH dem Triggerzeitpunkt noch ausgegeben werden sollen. Voreingestellt ist keine Begrenzung, d.h. die Datenausgabe erfolgt bis zum Stop der Aufnahme.
--trigger-source=*source*	Die Art der Ereignisse, die an das Trigger Skript übergeben werden. Voreingestellter Wert ist `data` (Datenereignisse). Sie können aber auch nach bestimmten Signalbedingungen triggern. In diesem Fall verwenden Sie `signal` als Trigger Quelle (source).
-v --verbose	Verbose, Ausgabe zusätzlicher Informationen.
-V --version	Ausgabe der Programmversion.

23.7 Eine Konfigurationsdatei für alle

Bisher haben wir entweder die Default Einstellung der einzelnen Tools verwendet oder unsere Vorgaben per Programmargumente an das jeweilige Tool übergeben. Je nach Länge der Verarbeitungskette und Anzahl der Parameter kann dies recht schnell sehr unübersichtlich und auch fehlerträchtig werden.

Alle Tools lassen sich deshalb einfach mit einer einzelnen Konfigurationsdatei steuern, die sie dem Programm `msb_record` als Parameter übergeben. `msb_record` sorgt automatisch dafür, das allen weiteren Programme in der Verarbeitungskette[8] die dort gemachten Einstellungen erhalten.

Die Konfigurationsdatei ist nicht Teil der Programminstallation sondern kann jederzeit mit folgendem Kommando neu erzeugt werden.

```
msb_record -C
```

bzw.

```
msb_record --create-config-file
```

Als Ergebnis landet die Datei `msb_tools.config` im aktuellen Verzeichnis. Öffnen Sie die Datei mit einem Editor Ihrer Wahl (unter Windows beispielsweise Notepad, Linux User verwenden gedit, kate, oder natürlich vi, emacs ...).

Die Konfigurationsdatei ist ausführlich dokumentiert. Passen Sie einfach die für Sie wichtigen Parameter an und speichern Sie die Datei anschließend unter einem anderen Namen. Letzteres ist sinnvoll, da ein erneuter Aufruf von `msb_record -C` diese ohne zu zögern überschreibt.

Sie können natürlich beliebige Konfigurationsdateien anlegen, für jede Applikation eine individuelle. Dateiname und Endung sind dabei ohne Belang.

Sobald Sie die Konfigurationsdatei `msb_record` übergeben, werden alle Programm der Verarbeitungskette die dort gespeicherten Einstellungen übernehmen. Zum Beispiel:

```
msb_record -c meine-config-datei | msb_format
```

[8] Dies betrifft natürlich nur alle zur Analyser Software gehörenden Kommandozeilen Tools.

bzw.

```
msb_record --config-file meine-config-datei | msb_format
```

Nun werden Sie einwenden:
Was aber, wenn ich als Datenquelle nicht `msb_record` verwende, sondern die Ausgabe einer Rekorddatei mittels `type` bzw. `cat`?

In diesem Fall können Sie jedem Programm in der Verarbeitungskette die Konfigurationsdatei auch individuell übermitteln. Alle Tools verstehen den Parameter `--config-file` bzw. einfach nur `-c`. Sie brauchen dazu nicht einmal die Konfigurationsdatei ändern. Rufen Sie einfach das entsprechende Tool mit der gewünschten Datei auf.

```
type examples\DataView\9bit.msblog | msb_format --config-file datei
```

A

ASCII Zeichensatz

ASCII (American Standard Code for Information Interchange) ist eine Form der Zeichenkodierung, die ursprünglich für Fernschreiber entwickelt wurde, und sich in den Anfängen des Computer Zeitalters als Standard-Code für Schriftzeichen etablierte.

Bei den ersten 32 Zeichen des ASCII Codes (hex 00...1F) handelt es sich um nicht druckbare Zeichen die für Steuerzwecke reserviert sind. Steuerzeichen sind z.B. der Zeilenvorschub (Linefeed) oder der Wagenrücklauf (Carridge Return). Sie werden bei Geräten verwendet, die den ASCII Code zu Steuerungszwecken verwenden wie beispielsweise Drucker oder Terminals. Ihre Definition ist historisch begründet.

Code 20 entspricht dem Leerzeichen, 7F ist ein Sonderzeichen, welches auch als Löschzeichen DEL bezeichnet wird.

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEK	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	SP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Obige Tabelle berücksichtigt nur 7 Bits pro Byte, d.h. die ersten 128 Zeichen. Erweiterungen des ASCII Codes machen sich dies zu nutze, indem Sie in die noch möglichen weiteren 128 Zeichen für länderspezifische Kodierungen oder Grafikzeichen verwenden. Da diese aber sehr unterschiedlich ausfallen und eine ausführliche Darstellung den Rahmen hier deutlich sprengen würden, beschränken wir uns auf die standardisierte 7 Bit Version.

B

Baudratemessung

Der MSB-RS485-PLUS Analyser erlaubt die Einstellung und Messung beliebiger Baudraten in einem weiten Bereich von 1 Baud bis 1 MBaud und das mit einer Genauigkeit besser als 0.1%.

Die Messung der Baudrate erfolgt 8 mal pro Sekunde. Dabei wird die Breite von singulären 0 oder 1 Bits gemessen und gemittelt. Je mehr Bits innerhalb des Messfensters von 125ms zur Messung zur Verfügung stehen, desto genauer wird die Messung. Ein größeres Datenaufkommen kann also zu stabileren und genaueren Messwerten führen. Der Analyser erlaubt drei Arten der Baudrate Messung.

1 Automodus **(UART A + B)**

2 CH1 **(UART A)**

3 CH2 (CH3) **(UART B)**

Im Auto-Modus wird nach Zufallsprinzip die Daten an der internen UART A (CH1) oder UART B (CH2 bzw. CH3) zur Messung verwendet, je nachdem welcher Kanal zum Messstart die erste Datenflanke liefert. Dadurch kann es zu schwankenden Messwerten kommen wenn beide Kanäle etwas unterschiedliche Taktgeneratoren verwenden. Dieser Modus ist besonders geeignet, um differierenden Baudraten auf der Sende- und Empfangsleitung auf die Schliche zu kommen.

Um definiert die Übertragungsraten eines Kanales zu messen, sollte der verwendete Eingang explizit gesetzt werden. Dies entspricht der Auswahl **Daten A** bzw. **Daten B** und wird im Einstelldialog des Kontrollprogrammes vorgenommen.

Im Statusfenster wird neben der eingestellten auch die gemessene Übertragungsrate angezeigt. Letztere mit ihrer prozentualen Abweichung zur voreingestellten Baudrate. Aus Platzgründen werden Abweichungen größer als ±50% mit Out! gekennzeichnet.

$$dBaud = 100 * \frac{Baudmete - Baudset}{Baudset}$$

Negative Werte weisen dabei auf eine zu kleine Übertragungsrate hin, positive auf eine zu grosse. Viele Bitfehler in einer Übertragung können durch nicht korrekt erzeugte Raten erklärt werden. Als grober Richtwert gilt:
Abweichungen von maximal ±3% können noch toleriert und ausgeglichen werden, darüber hinausgehende Abweichungen sollten vermieden werden.

Baudrate Toleranz

Vermeiden Sie bei der Übertragung Abweichungen der Baudrate mit mehr als 3%. Bitfehler sind dann die Folge.

Durch ungenügende Steilheit der Sender in einer EIA-422/485 Verbindung und daraus resultierenden langsamen Anstiegszeiten kann die Messung bei höheren Übertragungsraten etwas zu hoch ausfallen. Dies kann dann auch ein Hinweis sein, dass die EIA-422/485 Treiber nicht genügend an die verwendete Übertragungsrate angepasst sind.

C Farben

Die MSB-RS485-PLUS Analyser Software erlaubt Ihnen an verschiedenster Stelle die Angabe eigener Farben. Eine Auswahl vordefinierter Farben finden Sie hier.

Die Angabe von Farbwerten kann sowohl in Form eines Farbnamens erfolgen (die folgenden Tabellen geben einen Überblick über die vordefinierten Farbnamen) oder durch Eingabe des RGB (rot grün blau) Wertes als hexadezimale Zahl.
Beachten Sie bitte, das Farbnamen generell mit den englischen Bezeichnungen einzugeben sind, auch wenn Sie eine deutsche Version der Software verwenden. Manche Farben bestehen aus mehreren Worten, beispielsweise 'indian red'. Das Leerzeichen ist Teil des Namens und muß explizit mit angegeben werden.
In der folgenden Aufstellung finden Sie neben dem Farbnamen auch immer den RGB Wert, den Sie alternativ eingeben können. RGB Werte können in einer kurzen und einer langen Form eingegeben werden.

C.1 RGB Wert kurze Form

Die kurze Form #RBG reduziert jeden Farbanteil auf einen Wert zwischen 0...15, in hexadezimaler Schreibweise 0...F, wobei R,G und B jeweils einer Hexzahl 0...F entsprechen. D.h. jeder Farbanteil kann in 1/15 Schritten von 100% definiert werden. Zum Beispiel ist rot #F00 und weiß #FFF. Da der jeweilige Farbanteil Werte zwischen 0...F annehmen kann, bedeutet 0 das der jeweilige Farbanteil überhaupt nicht, bzw. bei F voll vorhanden ist. In der kurzen Form sind 16 x 16 x 16 = 4096 Farben möglich.

C.2 RGB Wert lange Form

Die lange Form #RRGGBB erweitert den Wertebereich für die einzelnen Farbanteile von 16 auf 256, was einfach eine höhere Auflösung pro Farbanteil ist, nämlich 1/256 Schritte von 100%. Rot wäre hier #FF0000, weiß #FFFFFF. Dies entspricht einem Farbbereich von 256 x 256 x 256 = 16777216 möglichen Farben.

C.3 Vordefinierte Farbnamen

Die vordefinierten Farbnamen sind eine Auswahl aus einer Liste standardisierter Farben wie sie z.B. bei der Darstellung von Webseiten verwendet werden. Von den erweiterten Farben abgesehen, sollte die Eingabe von 'green' deutlich intuitiver sein, als #0F0 bzw. #00FF00. Leicht zu merken sind hier neben 'black' und 'white' vor allem die Grundfarben.

C.3.1 Grey colors

Name/Value	Color	Name/Value	Color
black #000000		dim grey #696969	
dark grey #a9a9a9		grey #bebebe	
light grey #d3d3d3		white #fffff	

C.3.2 Basic colors

Name/Value	Color	Name/Value	Color
blue #0000ff		green #00ff00	
red #ff0000		cyan #00ffff	
magenta #ff00ff		yellow #ffff00	

C.3.3 Extended colors

Name/Value	Color	Name/Value	Color
medium spring green #7fff00		forest green #228b22	
lime green #32cd32		dark green #006400	
aquamarine #70db93		spring green #00ff7f	
medium aquamarine #66cdaa		sea green #238e6b	
medium turquoise #70dbdb		dark turquoise #00ced1	
steel blue #236b8e		sky blue #3299cc	
slate blue #007fff		light steel blue #b0c4de	

Name/Value	Color	Name/Value	Color
cornflower blue #6495ed		navy #23238e	
medium blue #0000cd		dark slate blue #483d8b	
medium orchid #9370db		medium slate blue #7f00ff	
blue violet #8a2be2		dark orchid #9932cc	
purple #b000ff		orchid #db70db	
violet red #cc3299		orange red #ff007f	
maroon #b03060		salmon #6f4242	
khaki #f0e68c		wheat #d8d8bf	
medium goldenrod #eaeaad		pale green #8fbc8f	
medium sea green #426f42		medium violet red #db7093	
turquoise #adeaea		cadet blue #5f9ea0	
light blue #add8e6		midnight blue #2f2f4f	
pink #bc8fea		thistle #d8bfd8	
plum #eaadea		violet #4f2f4f	
firebrick #8a2222		brown #a52a2a	
orange #cc3232		indian red #cd5c5c	
coral #ff7f50		tan #db9370	
sienna #8e6b23		gold #ffd700	
medium forest green #6b8e23		yellow green #99cc32	
dark olive green #556b2f		green yellow #adff2f	

D

Windows Trouble-Shooting

Dieses Kapitel gibt Ihnen die nötigen Hinweise, falls Ihr Gerät nicht erkannt wird oder nicht korrekt mit der Analyser Software kommuniziert. Linux Anwender lesen bitte das entsprechende Kapitel zur Fehlerbehebung unter Linux.

Im Falle eines technischen Problems trennen Sie als erstes Ihren MSB-RS485-PLUS Analyser von jeglichen RS485 Anschlüssen und Ihrem PC. Danach prüfen Sie:

1 Wurde der Analyser warm oder heiß?

2 Zeigt der Analyser Anzeichen von Beschädigungen?

Wenn Sie beide Fragen verneinen können, folgen Sie bitte den Instruktionen in der folgenden Tabelle.
Sollte das Problem weiterhin bestehen ist eine Fehlfunktion des Gerätes nicht auszuschließen. In diesem Fall oder wenn Ihr Analyser anderweitig beschädigt ist, kontaktieren Sie IFTOOLS unter support@iftools.com. Geben Sie bitte immer die Seriennummer Ihres Gerätes mit an. Sie finden dieses auf der Unterseite des Gehäuses.

D.1 Test des Analyser PC Verbindung

Die folgenden Tests dienen dazu, die einwandfreie Kommunikation zwischen PC und Analyzer zu überprüfen. Dies beinhaltet die Verifikation einer korrekten Treiber Installation als auch der fehlerfreien USB Verbindung zum Gerät.

Verbinden Sie den Analyser erneut mit Ihrem PC, aber lassen Sie die Bus Anschlüsse offen!

Symptom	Ursache	Abhilfe
Analyser LEDs bleiben aus	Nicht installierter Treiber	Treiber installieren, siehe D.3
	Beschädigtes USB Kabel oder beschädigter USB Anschluss	USB Kabel austauschen, anderen USB Anschluss (PC) verwenden
	Defekter Analyser	Analyser austauschen
Analyser wird nicht erkannt (Programm kann Analyser nicht finden)	Fehler bei der USB Enumeration	Seriennummer bei Start angeben, siehe D.4
	Falscher Treiber oder Treiber Version	Treiber neu installieren, siehe D.3

	Treiber Konflikt mit anderen USB Geräten	Log der Geräte Erkennung prüfen, siehe D.5
Firmware Übertragung schlägt fehl	Beschädigtes USB Kabel	USB Kabel austauschen
	Übertragungsfehler	Anpassen von Transfer Geschwindigkeit und Timeouts, siehe D.4
	Fehlerhafter Analyser	Analyser austauschen
Alle LEDs blinken rot nach Firmware Übertragung	EEPROM Checksum Fehler	Kontaktieren Sie IFTOOLS, siehe D.8

Ein korrekt verbundener und fehlerlos funktionierender Analyser (Firmware erfolgreich geladen) zeigt dies durch eine gelb blinkende Status LED. an.

D.2 Prüfen der Analyser Bus Anschlüsse

Sobald Sie Ihren Analyser mit einem Bus-System verbinden (unabhängig davon, ob es sich um einen kleinen Testaufbau oder einen Feldbus in einer industriellen Anlage handelt), wird der Analyser Teil eines größeren Systems. Dies bedeutet u.U. durch den Analyser fließende Ausgleichsströme und/oder hohe Spannungsspitzen an den Dateneingängen. All dies kann die Funktion des Analysers beeinträchtigen oder ihn gar beschädigen.
Vergewissern Sie sich deshalb, das der Bus Anschluss korrekt und gemäß den Analyser Spezifikationen erfolgt, bevor Sie den Bus mit dem Analyser verbinden.

Verbinden Sie Ihren MSB-RS485-PLUS Analyser mit dem Bus und starten Sie die Software

Symptom	Ursache	Abhilfe
Der Analyser reagiert nicht mehr	Kurzschluss am Bus Eingang	Überprüfen des Bus Anschluss (Datensignal, Masse)
Der Analyser nimmt keine Daten auf	Aufzeichnung nicht gestartet	Starten der Aufzeichnung
	Aufnahme der Datenbytes deaktiviert	Einstellung der Aufzeichnung überprüfen
	Falscher Bus Anschluss	Verbindung und Status der prüfen prüfen CH1 und CH2 LEDs
Analyser stoppt Aufzeichnung	Festplatte voll oder maximale Größe der Aufzeichnung erreicht	Reduzierung der Aufnahme Ereignisse (Signal Einstellungen), Festplattenplatz freigeben
Programm Kommunikation mit Analyser stoppt unerwartet	USB Port deaktiviert durch Windows Power Management	Power Management deaktivieren, siehe D.6

D.3 Treiber Installation

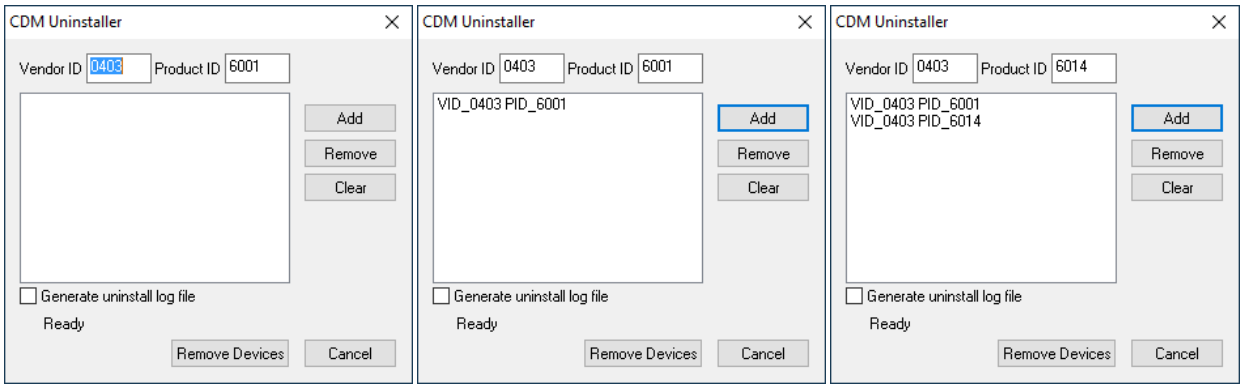
Der MSB-RS485-PLUS Analyser verwendet zur Kommunikation mit dem PC einen weit verbreiteten Chip von FTDI. Falls Sie andere USB Geräte mit dem gleichen Chip verwenden, kann ein Treiber Konflikt durch inkompatible Treiber Versionen nicht immer

völlig ausgeschlossen werden.

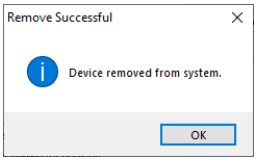
D.3.1 Treiber deinstallieren

Um auf der sicheren Seite zu sein, ist es generell besser, zunächst alle eventuellen alten Treiber zu entfernen, bevor der neue Treiber installiert wird.
Sie finden das geeignete Tool zur Treiber Deinstallation auf Ihrer IFTOOLS CDROM unter `Treiber & Tools`. Klicken Sie dort Interaktive Deinstallation.
Alternativ können Sie das Programm auch auf unserer Download Seite herunterladen.
Geben Sie dazu im Suchfeld der Seite `CDMUninstaller` ein.
Nach dem Download entpacken Sie die ZIP Datei und starten Sie das Programm `CDMuninstallerGUI.exe`.
Beachten Sie, dass Sie je nach System hierzu Administrator Rechte benötigen!

Achtung! Trennen Sie den Analyser von Ihrem PC bevor Sie den Treiber entfernen!



Der IFTOOLS Analyser verwendet wahlweise einen Kommunikations-Chip mit der USB Produkt ID 6001 oder 6014, (die Hersteller (Vendor) ID 0403 steht für FTDI). Fügen Sie deshalb zunächst beide Produkt IDs zu der Liste der zu entfernenden Gerätetreiber hinzu, wie im rechten Bild gezeigt. Anschließend klicken Sie den 'Remove Devices' Knopf um die Deinstallation zu starten.
Das Programm quittiert die Aktion mit einer entsprechenden Meldung. Fehlermeldungen wie 'Failed to remove device...' können Sie ignorieren. Die bedeutet lediglich, das der Treiber nicht existiert oder bereits entfernt wurde.



D.3.2 Treiber neu installieren

Es gibt mehrere Methoden, den geeigneten Treiber zu installieren.

1 Automatische Installation durch Windows (benötigt aktive Internet Verbindung)

2 Installation von der IFTOOLS CDROM oder einem IFTOOLS USB Stick

3 Installation durch andere Quelle

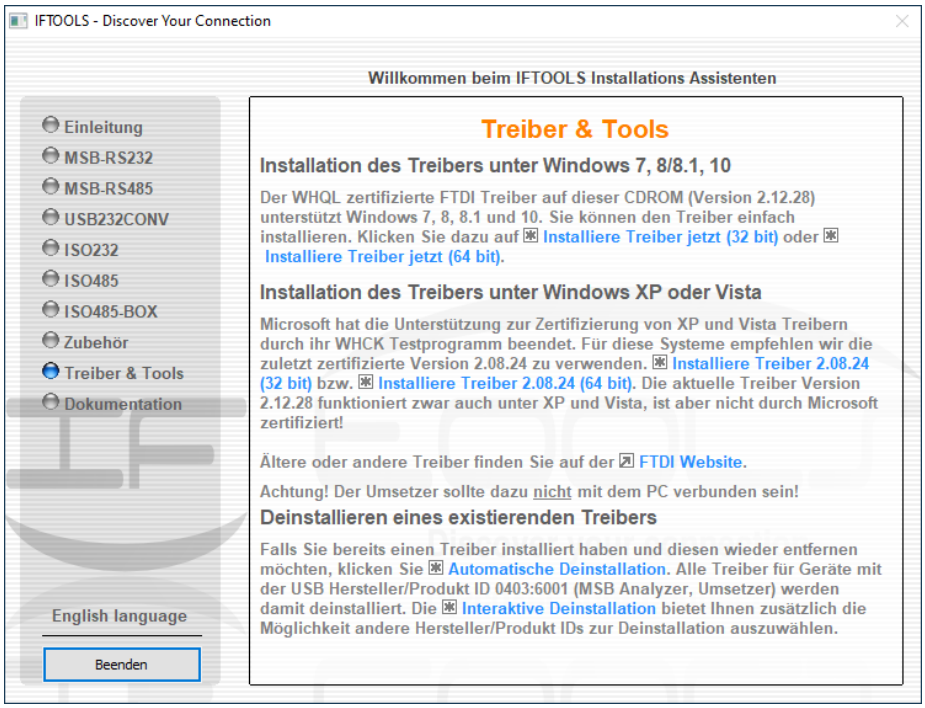
Automatische Installation durch Windows

Dies ist der einfachste Weg. Nachdem Sie den oder die alten Treiber entfernt haben, verbinden Sie den Analyser erneut mit Ihrem Windows PC und überlassen dem Betriebssystem die Arbeit. Die Installation kann einige Minuten in Anspruch nehmen. Je nach Windows Version bekommen Sie währenddessen dazu einige Statusmeldungen. Um ganz sicher zu gehen, öffnen Sie den Windows Geräte Manager (siehe D.7 und klappen Sie dort den Eintrag 'Anschlüsse (COM & LPT)' auf.

Nach erfolgreicher Installation muss dort ein weiterer Eintrag stehen. Danach können die Analyser Software starten.

Installation von CDROM oder USB stick

Legen Sie die IFTOOLS CDROM ein oder stecken Sie den IFTOOLS USB Stick an Ihren PC. Neuere Windows Versionen werden Sie um Erlaubnis zur Aufführung eines unbekannten Programms auf externem Medium fragen bevor Sie den IFTOOLS Installations Assistenten starten. In diesem Fall müssen Sie den Zugriff zunächst erlauben und dann ggf. das Programm `setup.exe` im Wurzelverzeichnis der CDROM/Stick manuell starten/anklicken.



Im Installations Assistenten wählen Sie links die Rubrik 'Treiber und Tools' aus, siehe Bild oben. Die CDROM enthält relativ aktuelle und getestete Treiber Versionen für alle Windows Arten (32 und 64 Bit OS).
Klicken Sie einfach den entsprechenden Link 'Installiere Treiber...' um die Treiber Installation zu starten.

Installation von anderen Quellen

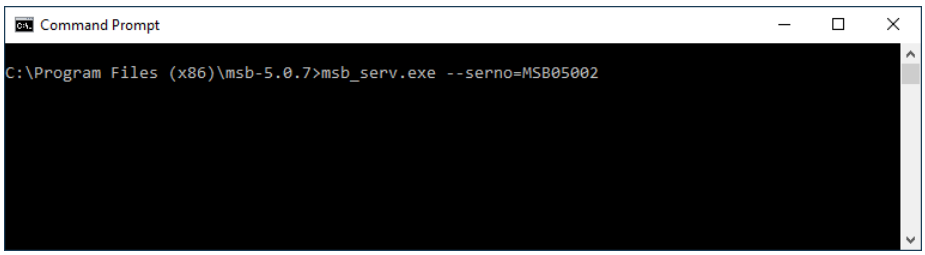
FTDI bietet die neuesten Treiber auf der VCP drivers page zum Download an. Wir empfehlen dabei die als 'setup executable' bezeichneten Treiberpakete zu verwenden, da diese eine unkomplizierte Treiber Installation durch einfaches Ausführen der Datei erlaubt.
Eine alternative Quelle für Treiber ist die IFTOOLS Download Treiber Seite. Hier können Sie ebenfalls zwischen 'ausführbaren' Treiberpaketen und solchen zur Installation per Geräte Manager wählen.
Nach dem Download des Treiber Pakets führen Sie dies entweder direkt aus (setup executable) oder geben dieses im Geräte Manager zur Treiber Installation an.

D.4 Hilfreiche Programm Parameter

Die Analyser Software, insbesondere das Kontrollprogramm `msb_serv.exe` zur Kommunikation und Steuerung des Analysers bietet einige spezielle Parameter, die die Geräte Erkennung sowie die Übertragung der Firmware gezielt beeinflussen können.
Normaler Weise werden Sie diese nie benötigen, da die voreingestellten Parameter optimal für den Geräte Zugriff eingestellt sind. Sie werden dann interessant, wenn Ihr Analyser Probleme bei der Erkennung und/oder Übertragung der Firmware zeigt.
Um die Analyser Software mit zusätzlichen Parametern zu starten, öffnen Sie ein Kommando Fenster (Command shell oder command prompt).

D.4.1 Analyzer wird nicht gefunden

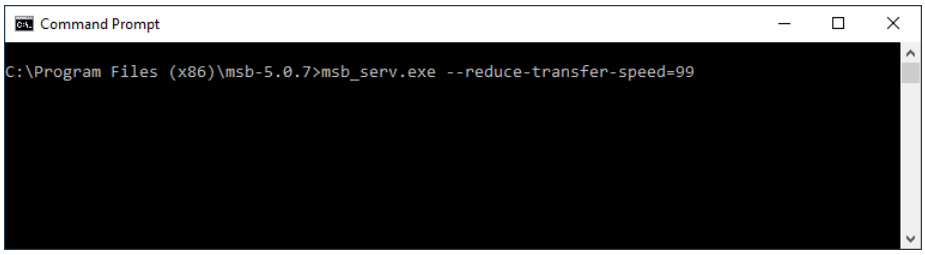
Wenn die Software den Analyser nicht findet (evtl. ausgelöst durch ein USB Enumerations Problem), reicht es manchmal bereits aus die Seriennummer des Gerätes beim Start mit `--serno=MSBxxxxx` zu übergeben, MSBxxxxx ist dabei die Seriennummer und auf der Geräte Rückseite zu finden:



Sie finden die Seriennummer auf der Unterseite des Geräts. Analyser der dritten Generation besitzen einen Barcode. Die letzten 5 Ziffern entsprechen dabei der Seriennummer. Geben Sie diese für `xxxxx` ein.

D.4.2 Firmware Übertragungsfehler

Das die Software einen Fehler bei der Übertragung der Firmware meldet (der Analyser wurde korrekt erkannt, aber die Firmware Übertragung unterbrochen) ist eher selten und meistens abhängig von der PC Hardware.
In diesem Fall kann eine Reduzierung der Übertragungsgeschwindigkeit helfen. Der entsprechende Parameter lautet: `--reduce-transfer-speed=speed`
Das folgende Beispiel überträgt die Firmware mit der geringst möglichen Geschwindigkeit:



Eine detaillierte Auflistung nebst Beschreibung aller möglichen Parameter finden Sie im Kapitel des Kontrollprogrammes 8.16. Die evtl. zusätzlich nötigen Parameter können Sie dann dem Start Icon der Analyser Software hinzufügen. Wie das geht wird im Abschnitt 8.12 erklärt.

D.5 Helfen Sie uns bei Gerätekonflikten

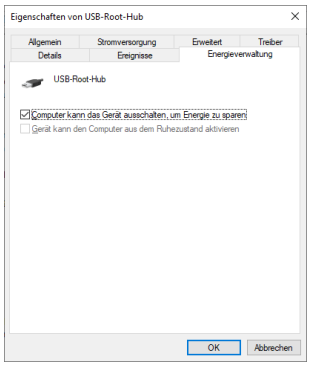
Der Firmware Loader verwendet die vom Betriebssystem während der USB Enumeration ermittelten Informationen um alle mit einem MSB-Analyser verbundenen seriellen Ports zu erkennen.
Bei den mannigfaltigen Kombinationen von existierenden USB Geräten und Treibern können wir nicht ausschliessen, dass in seltenen Fällen das Programm den Analyser nicht korrekt erkennt. Um solche Situationen bei der weiteren Programmentwicklung berücksichtigen zu können benötigen wir Ihre aktive Mithilfe.
Öffnen Sie dazu erneut eine Kommando Shell (DOS Box) und wechseln Sie in das Installationsverzeichnis. Geben Sie folgendes Kommando ein:

```
msb_serv.exe --verbose
```

Der `--verbose` Parameter erzeugt eine Reportdatei (AnalyzerScan.txt) mit zusätzlichen Informationen zur Analyser Erkennung und speichert diese auf dem Desktop. Senden Sie diese Datei anschließend an support@iftools.com.

D.6 USB Power Management deaktivieren

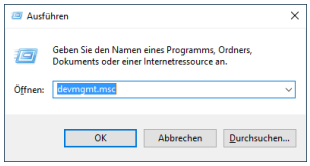
Der MSB-RS485-PLUS Analyser verliert die Verbindung zum PC und stoppt unerwartet. Dabei arbeitet das Gerät zunächst wie erwartet, bricht dann aber aus unerfindlichen Gründen die Aufnahme ab. I.a. gehen dabei auch alle LEDs aus.
Meistens liegt die Ursache des Problems an der Energie Einstellung Ihres PC's (hauptsächlich Notebooks und Laptops).
Um die Laufzeit im mobilen (Akku) Betrieb zu verlängen bzw. um generell Energie zu sparen, kann Windows USB Anschlüsse abschalten, die nicht im Betrieb sind bzw. auf eine entsprechende Anforderung nicht reagieren. Teilweise ist dieses Verhalten fehlerhaft im System oder BIOS implementiert, so dass auch aktive USB Geräte abgeschaltet werden.
Deaktivieren Sie deshalb das Power Management des USB Hubs wie folgt.



1. Starten Sie den Geräte Manager, siehe D.7
2. Klicken Sie in der Liste den Eintrag Universal Serial Bus Controller.
3. Rechtsklicken Sie den Eintrag USB Root Hub und wählen Sie Eigenschaften.
4. Klicken Sie auf Energie Verwaltung
5. Deaktivieren Sie 'Computer kann Gerät ausschalten um Energie zu sparen'
6. Wiederholen Sie dies für alle weiteren USB Root Hubs
7. Klicken Sie auf OK und schliessen Sie den Gerätemanager
8. Starten Sie Ihren Computer neu

D.7 Windows Geräte Manager

Auch wenn sich Design und Aufgaben des Geräte Managers in den letzten 20 Jahren kaum geändert haben, ist dieser mit jeder neuen Windows Version ein bisschen schwieriger zu finden. Vermutlich um das System vor falschen Hardware Einstellungen (und damit teils schwerwiegenden Eingriffen) seitens unerfahrener Anwender zu schützen. Wahr ist, dass der Geräte Manager unter normalen Umständen nur selten benötigt wird.
Nicht desto trotz ist er das erste Mittel wenn bestimmte Geräte den Dienst verweigern. Zum Glück ist die Tastenkombination zum Start des Geräte Managers seit Windows XP die gleiche. Drücken Sie einfach:



Windows Key + R

In der Eingabezeile des sich öffnenden Dialog geben Sie den Programmnamen des Geräte Managers ein: `devmgmt.msc` und drücken Sie die Enter Taste bzw. klicken Sie OK.

D.8 Andere Probleme

Sie haben ein anderes Problem, welches hier nicht genannt ist.
Mailen Sie uns unter support@iftools.com mit genauen Angaben zur Ihrem System (Windows Version, Service Pack, 32/64 Bit System) sowie der verwendeten Software Version und einer detaillierten Beschreibung Ihres Problems.
Vergessen Sie nicht, auch die Seriennummer Ihres Analysers anzugeben. Sie finden diese auf der Geräterückseite.

E

Linux Trouble-Shooting

Dieses Kapitel gibt Ihnen die nötigen Hinweise, falls Ihr Gerät nicht erkannt wird oder nicht korrekt mit der Analyser Software kommuniziert. Windows Anwender lesen bitte das entsprechende Kapitel zur Fehlerbehebung unter Windows.

Im Falle eines technischen Problems trennen Sie als erstes Ihren MSB-RS485-PLUS Analyser von jeglichen RS485 Anschlüssen und Ihrem PC. Danach prüfen Sie:

1 Wurde der Analyser warm oder heiß?

2 Zeigt der Analyser Anzeichen von Beschädigungen?

Wenn Sie beide Fragen verneinen können, folgen Sie bitte den Instruktionen in der folgenden Tabelle.

Sollte das Problem weiterhin bestehen ist eine Fehlfunktion des Gerätes nicht auszuschließen. In diesem Fall oder wenn Ihr Analyser anderweitig beschädigt ist, kontaktieren Sie IFTOOLS unter support@iftools.com. Geben Sie bitte immer die Seriennummer Ihres Gerätes mit an. Sie finden dieses auf der Unterseite des Gehäuses.

E.1 Test der Analyser PC Verbindung

Alle Standard Linux Kernel enthalten bereits alle nötigen Treiber Module um mit Ihrem IFTOOLS Analyser kommunizieren zu können. Nichtsdestotrotz lauern auf Grund der Vielzahl verschiedener Linux Distributionen und deren unterschiedlicher Umsetzung (speziell von Nutzer- und Gruppenrechte) immer wieder Fallstricke, die zu Beginn das einwandfreie Funktionieren des Analysers erschweren können.

Die folgenden Tests dienen dazu, die Anbindung Ihres Analysers an Ihrem PC und die korrekte Kommunikation zu überprüfen.

Verbinden Sie den Analyser erneut mit Ihrem PC, aber lassen Sie die Bus Anschlüsse offen!

Symptom	Ursache	Abhilfe
Analyser LEDs bleiben aus	Beschädigtes USB Kabel oder beschädigter USB Anschluss	USB Kabel austauschen, anderen USB Anschluss (PC) verwenden

	Defekter Analyser	Analyser austauschen
Analyser wird nicht erkannt (Programm kann Gerät nicht finden)	Fehlende Rechte	Überprüfen Ihrer Nutzer Rechte, siehe E.3
	Fehlende oder fehlerhafte udev Regel	Erneutes Installieren der udev Regel, siehe E.4
	Installiertes Braille Modul	Braille Modul entfernen, siehe E.5
	USB Enumerations Fehler	Log der Geräteerkennung prüfen, siehe E.7
	Andere Gründe	System log prüfen, siehe E.8
Fehler bei Firmware Übertragung	Beschädigtes USB Kabel	USB Kabel austauschen
	Übertragungsfehler	Geschwindigkeit/Timeous anpassen, siehe E.6
	Fehlerhafter Analyser	Analyser austauschen
Alle LEDs blinken rot nach Firmware Übertragung	EEPROM Checksum Fehler	Kontaktieren Sie IFTOOLS, siehe E.9

Ein korrekt verbundener und fehlerlos funktionierender Analyser (Firmware erfolgreich geladen) zeigt dies durch eine gelb blinkende Status LED. an.

E.2 Prüfen der Analyser Bus Anschlüsse

Sobald Sie Ihren Analyser mit einem Bus-System verbinden (unabhängig davon, ob es sich um einen kleinen Testaufbau oder einen Feldbus in einer industriellen Anlage handelt), wird der Analyser Teil eines größeren Systems. Dies bedeutet u.U. durch den Analyser fließende Ausgleichsströme und/oder hohe Spannungsspitzen an den Dateneingängen. All dies kann die Funktion des Analysers beeinträchtigen oder ihn gar beschädigen.
Vergewissern Sie sich deshalb, das der Bus Anschluss korrekt und gemäß den Analyser Spezifikationen erfolgt, bevor Sie den Bus mit dem Analyser verbinden.

Verbinden Sie Ihren MSB-RS485-PLUS Analyser mit dem Bus und starten Sie die Software

Symptom	Ursache	Abhilfe
Der Analyser reagiert nicht mehr	Kurzschluss am Bus Eingang	Überprüfen des Bus Anschluss (Datensignal, Masse)
Der Analyser nimmt keine Daten auf	Aufzeichnung nicht gestartet	Starten der Aufzeichnung
	Aufnahme der Datenbytes deaktiviert	Einstellung der Aufzeichnung überprüfen
	Falscher Bus Anschluss	Verbindung und Status der prüfen prüfen CH1 und CH2 LEDs
Analyser stoppt Aufzeichnung	Festplatte voll oder maximale Größe der Aufzeichnung erreicht	Reduzierung der Aufnahme Ereignisse (Signal Einstellungen), Festplattenplatz freigeben

E.3 Nutzer Rechte prüfen

Linux behandelt den Analyser als serielles USB Gerät, welches als `/dev/ttyUSBx` vom Kernel im System eingebunden wird. Um darauf zugreifen zu können, benötigen Sie bzw. das Analyser Programm Lese/Schreib Rechte. Mit Ausnahme von root ist dies nur Mitgliedern der Gruppe `dialout` (Debian basierende Linux Systeme wie z.B. Ubuntu) oder `uucp` (SuSE) erlaubt. Ob Sie die nötige Gruppen Mitgliedschaft besitzen, können Sie wie folgt prüfen. Öffnen Sie dazu ein Terminal Fenster und geben das nachstehende Kommando ein:

```
ls -l /dev/ttyUSB*
```

Die Ausgabe sollte ähnlich wie diese sein:

```
crw-rw---- 1 root dialout 188, 0 2020-08-26 14:47 /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 0 2020-08-26 14:47 /dev/ttyUSB1
```

Überprüfen Sie nun Ihre Gruppen Mitgliedschaft mit:

```
groups
```

Das folgende, beispielhafte Resultat zeigt, dass Sie Mitglied Ihrer eigenen Gruppe sind (hier als YOUR_USER_NAME), sowie Mitglied der Gruppen groups cdrom, floppy, audio, video und plugdev. Sie sind aber nicht Mitglied von dialout (oder uucp) welche für den Zugriff auf den Analyser unabdingbar ist.

```
YOUR_USER_NAME cdrom floppy audio video plugdev
```

Um Mitglied der Gruppe dialout (oder uucp) zu werden, müssen Sie das nachstehende Kommando per `sudo` ausführen. (sudo gibt Ihnen für die Ausführung root Rechte. Sollten Sie keine sudo Rechte haben, fragen Sie Ihren System Administrator). SuSE Anwender ersetzen hier `dialout` durch `uucp`!

```
sudo adduser YOUR_USER_NAME dialout
```

Beachten Sie! Um die neuen Rechte zu erlangen, müssen Sie ich als Benutzer neu anmelden. Ein Reboot ist nicht nötig!

E.4 udev Regel installieren

Beginnend mit Version 5.0 erfolgt die Analyser Kommunikation nicht mehr über einen virtuellen seriellen Port sondern ohne Umweg per direktem USB Zugriff (FTDI d2xx). Dies bewirkt eine deutliche Erhöhung der Transfer Rate und ist obligatorisch für die neue Analyser Generation (PLUS Serie).
Leider schließen sich beide Arten von Geräte Zugriff (virtueller COM Port versus direct access d2xx) unter Linux gegenseitig aus. D.h. eine Kommunikation per direktem USB Zugriff ist nicht möglich, solange das Gerät vom Kernel als virtueller COM Port (/dev/ttyUSBx) eingebunden ist.
Wie sich der Linux Kernel beim Anstecken eines neuen USB Gerätes verhält ist in weitem Maße per `udev` Regeln konfigurierbar. Eine solche Regel wird normaler Weise bei der Software Installation installiert und verhindert, dass der Kernel einen IFTOOLS Analyser als virtuellen COM Port registriert. Wenn diese Regel nicht existiert, wird der Analyser von der Software nicht erkannt. Ob diese udev Regel korrekt installiert wurde, können Sie wie folgt prüfen:

```
ls -l /etc/udev/rules.d
```

In einem Linux System mit einem angeschlossenen und funktionierenden Analyser sehen sie mehrere Einträge, die Datei (udev Regel) `10-iftools-msb.rules` ist dabei entscheidend:

```
-rw-r--r-- 1 root root   262 Jan 12 14:23 10-iftools-msb.rules
-rw-r--r-- 1 root root   620 Feb 23  2016 70-persistent-net.rules
-rw-r--r-- 1 root root 58549 Jan 17  2019 70-snap.core.rules
-rw-r--r-- 1 root root   984 Mar  4 16:10 70-snap.telegram-desktop.rules
```

Sollte diese nicht vorhanden sein, müssen Sie diese nachinstallieren. Dies geschieht am einfachsten, indem Sie ein Terminal Fenster öffnen, ins Installationsverzeichnis der Analyser Software wechseln und dort das udev Installation Skript per `sudo` ausführen:

```
cd ~/msb-5.0.9
sudo ./udev-install.sh
```

Hier der Inhalt einer korrekten udev Regel für alle IFTOOLS Analyser:

```
1 # 10-iftools-msbc.rules
2 #
3 # Apply the new rules with: sudo udevadm trigger
4 ATTRS{idVendor}=="0403",\
5 ATTRS{idProduct}=="6001|6014",\
6 ATTRS{manufacturer}=="IFTOOLS",\
7 MODE="0660",\
8 GROUP="dialout",\
9 RUN+="/bin/sh -c 'echo -n %k:1.0 > /sys/bus/usb/drivers/ftdi_sio/unbind'"
```

Diese Regel wird vom Kernel für alle USB Geräte angewendet, die einen Chip von FTDI besitzen (vendor ID für FTDI ist 0403), der Chip Typ (die Produkt ID) 6001 oder 6014 entspricht und der Produkt Hersteller (manufacturer) IFTOOLS ist. Letztere Bedingung verhindert, das USB Geräte anderer Hersteller mit gleichem Chipsatz von dieser Regel betroffen sind.

Um die korrekte Arbeitsweise der Regel zu testen, öffnen Sie erneut ein Terminal Fenster und geben Sie folgendes Kommando ein:

```
dmesg
```

Hier erkennt der Kernel ein neues USB Gerät (als Beispiel einen IFTOOLS Analyser MSB-RS485-PLUS) und hebt die Zuordnung als ttyUSB2 auf. Damit ist er frei für den direkten USB (d2xx) Zugriff.

E.5 Braille Modul entfernen

Sie haben die korrekten Zugriffsrechte auf die Gerätedatei `/dev/ttyUSBx` und die udev Regel ist installiert. Trotzdem wird der Analyser nicht erkannt.
Vergewissern Sie sich, daß auf Ihrem System kein Braille Modul (ein Blindenschrift fähiges Ausgabegerät) installiert ist. Trennen Sie dazu den Analyser von Ihrem PC und stecken Sie ihn wieder an. Öffnen Sie anschließend eine Konsole und geben Sie das

Kommando `dmesg` ein.
Ist die Ausgabemeldung ähnlich der folgenden:

```
~$ dmesg
Detected FT232BM Feb 11 16:14:59 sd kernel: [ 1575.765756] usb 3-2:
FTDI USB Serial Device converter now attached to ttyUSB0 Feb 11
16:14:59 sd kernel: [ 1575.881392] usb 3-2: usbfs: interface 0 claimed
by ftdi_sio while 'brltty' sets config Feb 11 16:14:59 sd kernel: [
1575.885485] ftdi_sio ttyUSB0: FTDI USB Serial Device converter now
disconnected from ttyUSB0
```

ist ein Braille Treiber installiert. Wenn Sie kein Braille Gerät verwenden, entfernen Sie den Treiber mit Hilfe des Paket Management Ihres System. Unter Debian basierenden Distributionen (Ubuntu) beispielsweise:

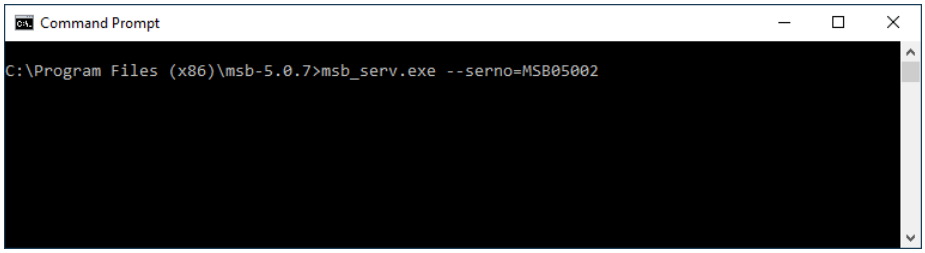
```
~# apt-get remove brltty
```

E.6 Hilfreiche Programm Parameter

Die Analyser Software und hier insbesondere das Programm, welches für die Kommunikation und Steuerung des Analysers verantwortlich ist (das Kontrollprogramm), erlauben mit einigen 'speziellen' Parametern die Geräte Erkennung sowie Firmware Transfer zu beeinflussen.
Im Normalfall werden Sie diese nicht benötigen. Die voreingestellten Werte sind optimal für die meisten Anwendungen. Sie werden dann interessant, wenn die Erkennung und/oder der Firmware Transfer fehl schlagen.
Um den Analyser mit diesen zusätzlichen Parametern zu starten, öffnen Sie eine Konsole (Terminal Fenster) und wechseln in das Analyser Software Installationsverzeichnis (i.a. `$HOME/msb-7.0.2`).

E.6.1 Analyser wird nicht erkannt

Wenn die Software den Analyser nicht findet (evtl. ausgelöst durch ein USB Enumerations Problem), reicht es manchmal bereits aus die Seriennummer des Gerätes beim Start mit `--serno=MSBxxxxx` zu übergeben, MSBxxxxx ist dabei die Seriennummer und auf der Geräte Rückseite zu finden:



Analyser der dritten Generation besitzen einen Barcode. Die letzten 5 Ziffern entsprechen dabei der Seriennummer. Geben Sie diese für `xxxxx` ein.

E.6.2 Firmware Übertragungsfehler

Das die Software einen Fehler bei der Übertragung der Firmware meldet (der Analyser wurde korrekt erkannt, aber die Firmware Übertragung unterbrochen) ist eher selten und meistens abhängig von der PC Hardware.
In diesem Fall kann eine Reduzierung der Übertragungsgeschwindigkeit helfen. Der entsprechende Parameter lautet: `--reduce-transfer-speed=speed`
Das folgende Beispiel überträgt die Firmware mit der geringst möglichen Geschwindigkeit:

```
./msb_serv -r99
```

Eine detaillierte Auflistung nebst Beschreibung aller möglichen Parameter finden Sie im Kapitel des Kontrollprogrammes 8.16. Die evtl. zusätzlich nötigen Parameter können Sie dann dem Start Icon der Analyser Software hinzufügen. Wie das geht wird im Abschnitt 8.12 erklärt.

E.7 Helfen Sie uns bei Gerätekonflikten

Der Firmware Loader verwendet die vom Betriebssystem während der USB Enumeration ermittelten Informationen um alle mit einem MSB-Analyser verbundenen seriellen Ports zu erkennen.
Bei den mannigfaltigen Kombinationen von existierenden USB Geräten und Linux Distributionen können wir nicht ausschliessen, dass in seltenen Fällen das Programm den Analyser nicht korrekt erkennt. Um solche Situationen bei der weiteren Programmentwicklung berücksichtigen zu können benötigen wir Ihre aktive Mithilfe.
Öffnen Sie dazu erneut ein Terminal Fenster und wechseln Sie in das Installationsverzeichnis. Geben Sie folgendes Kommando ein:

```
./msb_serv --verbose
```

Der `--verbose` Parameter erzeugt eine Reportdatei (AnalyzerScan.txt) mit zusätzlichen Informationen zur Analyser Erkennung und speichert diese auf dem Desktop. Senden Sie diese Datei anschließend an support@iftools.com.

E.8 System Log prüfen mit dmesg

Sobald ein neues USB Gerät angesteckt oder entfernt wird erzeugt der Linux Kernel eine Fülle nützlicher Informationen in einem internen Kernel Ringpuffer. Mit folgendem Kommando ausgeben Sie den Pufferinhalt ausgeben:

```
dmesg
```

Stecken Sie dazu Ihren Analyzer am PC an und öffnen Sie den System Log per `dmesg` Kommando in einem Terminal Fenster.
Ein unbeschädigter und funktionierender Analyser verursacht immer einige entsprechenden Zeilen in der `dmesg` Ausgabe.
Im Zweifelsfalle speichern Sie die Ausgaben mit:

```
dmesg > log.txt
```

in einer Datei und schicken diese an support@iftools.com

E.9 Andere Probleme

Sie haben ein anderes Problem, welches hier nicht genannt ist.
Wir sind sehr daran interessiert, unser Produkte/Software auf allen modernen Linux Varianten lauffähig zu machen. In Hinblick auf die große Zahl von verschiedenen Linux Distributionen ist dies nicht immer ganz einfach. Deshalb:
Mailen Sie uns unter support@iftools.com mit genauen Angaben zur Ihrem System (Linux Distribution, Desktop Umgebung, 32/64 Bit System) sowie der verwendeten Software Version und einer detaillierten Beschreibung Ihres Problems.
Vergessen Sie nicht, auch die Seriennummer Ihres Analysers anzugeben. Sie finden diese auf der Geräterückseite.

Glossar

Glossar	Beschreibung
Aufzeichnungstiefe	Die Anzahl maximaler Ereignisse oder Abtastungen, aus denen sich eine Signalaufzeichnung zusammensetzen kann, wird Aufzeichnungs- oder Speichertiefe genannt und hängt vom verfügbaren Speichermedium ab. 203
CSV	Comma Separated Values, Textdateiformat, in der der Inhalt einzelner Datensätze durch Kommas getrennt in jeweils einzelnen Zeilen abgelegt wird. 90
Datagramm	Datagramm ist ein Oberbegriff für ein Datenframe, Datenpaket oder Datensegment und beschreibt eine zusammengehärende Datensequenz mit definiertem Anfang und Länge. In Feldbussen auch als Telegramm bezeichnet. 125
ETX	End of Text, im ASCII Steuerzeichensatz hexadezimal 0x03. Markiert das Ende einer Nachricht bzw. Datagrammes. 125
Firmware	Mit Firmware wird die in einem elektronischen Gerät enthaltene und für dessen Funktion notwendige Software bezeichnet. Firmware kann fester Bestandteil der Hardware sein, oder erst beim Starten in das Gerät geladen werden. 49
FLEXUART	Eine spezielle von IFTOOLS entwickelte UART erlaubt das präzise Setzen und Messen beliebiger Baudraten im Bereich von 1 Baud bis 1 MBaud mit 0,1% Genauigkeit. 53
Halbduplex	HX, manchmal auch HDX), gleichbedeutend mit 'Wechselbetrieb'. Informationen können auf einem Kanal in beide Richtungen fließen, aber nicht gleichzeitig, sondern nur abwechselnd. 1
Lua	Lua ist eine imperative und erweiterbare Skriptsprache zum Einbinden in Programme, extrem schnell und mit sehr geringem Resourcenverbrauch. 75, 86

Glossar	Beschreibung
Multi-Master	Busteilnehmer, die selbständig auf den Bus zugreifen dürfen (im Sinne von schreiben bzw. senden), bezeichnet man als aktive Knoten oder Master, (andernfalls heißen sie passive Knoten oder Slave). Ein Bus, der mehrere Master erlaubt, heißt Multimaster-Bus. Bei einem Multimaster-Bus ist eine zentrale oder dezentrale Busvermittlung notwendig, die gewährleistet, dass zu jedem Zeitpunkt jeweils nur ein Master die Bus-Herrschaft besitzt um gleichzeitiges Senden durch mehrere Master zu verhindern. 2
Multidrop	Eine Kommunikation auf Basis des Master/Slave Prinzips wobei ein Master (Sender) mehrere Empfänger ansprechen kann, diese aber nicht antworten. 1
RTF	Ein Datei- und Datenaustauschformat für formatierte Texte, ursprünglich von Microsoft entwickelt. 89
RTS/CTS Handshake	Eine Hardware-Flusssteuerung die durch entsprechende Signalpegel der RTS bzw. CTS Leitungen realisiert ist. Die RTS/CTS Leitungen beider Teilnehmer sind gekreuzt verbunden. Durch Setzen der RTS Leitung auf logisch 1 fordert der jeweilige Empfänger einen Stop der Datensendung. Die wenigsten UARTs behandeln die Flusssteuerung tatsächlich in Hardware, so dass hier die Treiber den Zustand erkennen und entsprechend schnell reagieren müssen. 1
STX	Start of Text, im ASCII Steuerzeichensatz als hexadezimal 0x02 definiert. Markiert den Anfang einer Nachricht bzw. Datagrammes. 125
UART	Universal Asynchronous Receiver Transmitter, elektronisches Bauelement, welches zum asynchronen Sendung und Empfangen von Daten über eine serielle Datenleitung dient. Ein Synchrontakt bzw. Signal ist dabei nicht nötig.. 2, 25
USART	Universal Synchronous und Asynchronous Receiver Transmitter. Electronisches Bauelement welches Daten asynchron oder synchron senden und empfangen kann. 35
Vollduplex	(DX, manchmal auch FDX abgekürzt), gleichbedeutend mit 'Gegenbetrieb'. Die Übertragung der Information ist gleichzeitig auf zwei Kanälen (Sende-, Empfangskanal) möglich. 1

Glossar	Beschreibung
Zeitbasis	Die einem Raster (10 Pixel) entsprechende Zeitspanne. Je niedriger die Zeitbasis, desto höher die zeitliche Auflösung der Darstellung. Die niedrigste Zeitbasis im Signalmonitor beträgt 500ns, d.h. 50ns pro Pixel.. 205

Index