IF tOOLS
Discover your connection

Manual

MSB-RS485

Version 4.6.0

Inhaltsverzeichnis

1

Analysis of RS422/485 Bus systems

In contrast to other busses RS422 or RS485 define only the electrical characteristics. All further protocol levels can be specified freely. So beside the physical features an analysis also has to regard the different protocols.

RS485 and RS422 (or. EIA-422 and EIA-485) are mostly used synonymously because of their great similarity, where the EIA-422 standard is seen as a subset of EIA-485. But this is correct only partially.
Both standards use a pair of twisted wires to transmit the inverted and non-inverted levels of a one bit signal. The receiver reconstructs the original data signal from the difference of both signal levels. In this way common mode distortions do not have much effect on the transmission which leads to a significant higher noise immunity.
As a consequence all data and handshake lines are designed as wire pairs. However no standardized terminal assignments for EIA-422 and EIA-485 exist.

A EIA-422 connection generally consists of two pairs of wires for send and receive and a common ground line which is conform to the classical EIA-232 connection. In case of RTS/CTS Handshake two additional wire pairs have to be used. EIA-422 primarily was developed to overcome the limitations of EIA-232 connections.
With EIA-422 Full-Duplex point-to point transmissions and Multidrop networks can be realized. The latter allows the unidirectional connection of up to 10 receivers, in which the transmission takes place in one direction only from the sender to the maximal 10 receiver.

EIA-485 was designed as a bidirectional bus system for up to 32 (and more[1]) participants. Data can be transmitted optionally over a singe pair of wires Half-Duplex (the so-called 2-wire technology or short 2-wire) or in a Fullduplex capable way with two separate send and receive wire pairs (4-wire).
In a 2-wire system all sender and receiver are connected together through a single pair of wires. The main advantage of the 2-wire technology is its Multi-Master capability. Each bus device can exchange data with any other device.
A well known application based on a 2-wire system is the PROFIBUS.

[1] depending on the so-called unit load it may be up to 256 participants

4-wire busses (for instance the DIN-Messbus) are solely used as Master-Slave systems. That means that the data output of the master is connected to all data inputs of the slaves through a single wire pair. The data outputs of the slaves are all connected to the second wire pair which leads to the data input of the master.
In both variants only one device can drive (send), all other devices have to set their sender into tri state mode.
Some EIA-485 devices automatically care for a correct implementation of the tri state status (tri state when no data is sent), others have to be explicitly set into tri state by software control.

Physically both interfaces are almost identical so that EIA-485 devices can be used without problems in EIA-422 systems. But this is not possible the other way round, because EIA-422 drivers do not have the tri state mode which is necessary to operate multiple devices on one wire pair.

The analysis of a EIA-422/485 connection does not only have to care about the different connection varieties (and the bus signals). The EIA-422/485 specifications do not make a statement about the protocol levels. So a number of different transfer protocols are established, protocols with asynchronous (UART based) and synchronous serial data transmission.
Protocols with synchronous data transmissions use different kinds of bit coding, with or without synchronizing clock, and use special coding hardware.
In contrast the asynchronous transmission technologies are based on the UART which is the standard for serial interfaces and is installed in every PC and microcontroller. Because of its simply way of connecting (with EIA-232 or USB to EIA-422/485 converters) these protocols are widespread why in the following we focus on UART based asynchronous protocols. These are some of them:

1. **Din-Messbus**
2. **Modbus ASCII**
3. **Modbus RTU**
4. **Profibus**
5. **Application specific protocols**

The kind of the protocol plays a very important role not only for the logging and evaluation of the communication but also for the right choice of the analysis tools.

After this short side-note to the specification of EIA-422/485 the basis is laid for the question: Which possibilities for analysis of EIA-485[2] communications are available and where are they appropriate for?

Because of the simple connection of EIA-485 busses, based on asynchronous data transmission, to a standard PC the following techniques for logging and evaluation are possible with appropriate software:

1. **Data logging by the serial driver and additional software of the PC**

[2] EIA-485 Analysis tools are also appropriate for EIA-422 connections. In the following chapters we speak about EIA-485 only, meaning both standards.

2 **Bus-tap by one EIA-485 converter (2-wire bus)**

3 **Bus-tap by two EIA-485 converters (4-wire bus)**

4 **Sampling of the bus lines by special additional hardware**

1.1 Special serial driver software

If one of the participants of the communication is a PC (usually the master) an appropriate driver can be installed to protocol every sent and received data byte. This procedure only allows the logging and detecting of the data bytes which are processed by the serial driver of the operating system. The disadvantages are:

Data losses due to buffer overflows are not detected.

A precise time stamping is not possible. Indeed the received and sent data bytes are signalized by interrupt. But the interrupt is processed by the operating system after a not exact predictable time delay.

Therefore the time measurement of such Sniffer programs, most time in the millisecond range, have to be regarded with suspicion. The times are the times when the operating system handles the input and output of the characters. They are not the moment the character is really present on the data line.
A statement about correct data and protocol timing can only be done with care. Some busses as Modbus RTU or Profibus define a send pause as the start or end of the data telegram. Data within a telegram have to be transmitted without gaps.
Information about the tri-state condition get lost since the serial driver can process the two logical bus states only. Errors caused by data collision by reason of multiple sending bus participants can not be detected.

1.2 Bus-tap 2-Wire Bus

The PC is connected via an appropriate EIA-485 interface converter (EIA-232 to EIA-485 or USB to EIA-485) as an additional bus participant to the bus.
This variant allows the logging of all transmitted data bytes with e.g. Hyperterm. The disadvantages:

Since send and receive data run over the same wires it is not possible to distinguish between sent and received data. A detailed examination of the telegrams is necessary to detect the data direction.

The behavior of the time measurement is the same as mentioned under 1.1.

The connection of a EIA-485 converter is normally done as a serial COM port (either as a virtual COM port in case of a USB to EIA-485 converter or direct in case of a EIA-232 to EIA-485 interface converter).
In both cases the information about the tri-state condition gets lost with the consequence that bus errors, caused by multiple active senders, can not be recognized.

1.3 Double bus-tap 4-Wire bus

The send and receive lines are separately recorded. However this way needs two EIA-485 converter and special drivers since the recorded data bytes have to be marked with a time stamp or unique number to synchronize both data streams.

The data direction is correctly recognized but the data sequence is not always clear because of the interrupt delays, explained above. This possibility was used under MS-DOS since this operating system allowed the direct real-time processing of the interrupts.
The statements about time measurement and the tri-state condition are the same as under 1.2.

1.4 Sampling

This method needs additional independent hardware to simultaneously sample all signal lines and produce correspondent results. The advantages:

Because the sampling and evaluation is independent of the connected devices and the PC invalid tri-state conditions, wrong baudrates or UART settings are clearly detected and logged.

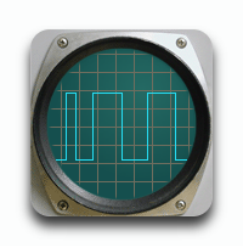
Furthermore the parallel sampling of all signal lines allows precise time stamps for the data lines and optional handshake lines like RTS/CTS for point to point connections. This is mandatory for examination of protocols which have to follow exact timing rules.
More: Even jittering or slightly deviating baudrates of single bus devices can be detected.

Sampling analyzers combine the advantages of protocol analyzers with the features of a digital scope and offer beside the logging of the data transfer also the physical or logical display of the line levels.

2
MSB-RS485 Analyzer

The MSB-RS485 Analyzer is an essential tool for analysis and optimizing of RS485/422 connections. As an autonomous device it gathers exact information about every line change with micro second precision, independent from the PC and its operating system.
Equipped with a multitude of visualization tools it allows a detailed view into every RS485/422 communication and detects conditions which can be recorded by a true 'hardware solution' only.

The Analyzer MSB-RS485 samples all eight signals simultaneously with a sampling rate of maximum 16 MHz. Thereby all events, that means level changes on the line, are stamped with the exact time in a micro second precise resolution.
As events all changes of the line levels are regarded, including the tri-state. That means a change from 0 (space) to 1 (mark) and from an inactive (tri-state) to an active logical level (and of course vice versa).
The recorded data are transferred via USB to the PC at which a 500kByte cache memory serves as a buffer to avoid data losses.
The recording is not limited. Date types with 64 bytes for the μs precise time stamps guarantee any length of recording, even with this time resolution (if you define 500000 years as unlimited).The recording depends only on the maximum allowed file length (disc capacity).



Max. 16MHz sampling cares for clear details and μsec precise time stamps

2.1 Advantages of a hardware solution

The MSB-RS485 Analyzer offers the capabilities of a logic analyzer, combined with a very low price. With it the disadvantages of pure software solutions are avoided by the direct evaluation of the signal changes in an independent hardware.
Analyzing solutions, based on software, depend on the not constant reaction and computing times of interrupts in the operating system. The usage of EIA-422/485 to USB converters add unpredictable delays of the USB subsystem. Furthermore the hardware input Fifos of the PCs are normally limited to 16 characters and 115200 Baud. If the interrupt handling is too slow characters will get lost because of the input buffer overflow.
The resulting time stamps are the times of the interrupt execution, not the real time of the occurred events.

Correspondingly the chronological relationship between the send and receive data is imprecise if two EIA-485 converter are used for logging of a full duplex 4-wire connection (T+, T-, R+, R-).

Especially if using protocols with a an exact to follow timing relationship (e.g. max. pause between the single data bytes of a telegram or pause between the telegrams themselves like requested by ModBus RTU or Profibus)) you will not be sure if the measured timing is the real bus timing.

In contrast to traditional converters the MSB-RS485 analyzer also supports protocols with 9 bit data length. 9-bit values are used for certain binary protocols to differ between data and address bytes or to indicate the frame or telegram start.

By connecting the EIA-485 bus via serial interface all information about the tri-state condition get lost. The consequence is that data collisions caused by multiple simultaneously active driving senders can not be detected. The MSB-RS485 detects the tri-state level correct even if the differential signal is drawn to a certain rest level (Idle, Pull up, pull down resistors).

The MSB-RS485 analyzer and pure software solutions in comparison:

Feature	MSB-RS485	Software solution
Detects invalid levels (tristate)	yes	no
Any Baudrate 1 Baud to 1 MBaud	yes	no
Real time stamps	yes	no
Time resolution 1 micro second	yes	no
Display of the real level changes	yes	no
Automatic detection of baudrate and protocol	yes	no
Supports protocols with 9 Bit data word length	yes	no
Correct time relationships between data and control lines	yes	no
Detects baudrate jitter and wrong bit times	yes	no

2.2 Innovative software concept

Already while recording any section of the data transfer can be investigated. This includes the physical display of the signals in different time resolutions (scope display) as well as the display of the transferred data bytes.

The software of the analyzer is designed as a Multi-Process Architecture. While the control program controls the recording the transferred data can already

be checked, evaluated and searched in any number of analysis windows in different time resolutions.

In this way the logical signal of a 2-wire bus or of both data lines can be followed and at the same time an earlier section can be watched in a higher resolution. That is also possible in all other analysis windows and allows the comparison of transferred data at different moments.

Extensive search mechanisms allow the search for defined data sequences, where also complex search requests are possible. That is done via regular expressions and could be:
All data strings which start with an 'A' and end with 'Z'. Also a search for defined levels or level changes can be done. The MSB-RS485 analyzer is supplied via USB from the PC and is appropriate for mobile operation when using a Laptop or notebook.

MultiPlatform Support
Versionen für Windows und Linux

Application specific protocols and telegrams can be displayed with the help of the integrated script language LUA.

Analyzer and software run under Microsoft Windows or Linux. MultiPlatform Support Version for Windows and Linux.

2.3 Application fields

The analyzer MSB-RS485 finds its use for logging and evaluating of asynchronous data transmissions based on the EIA-422/485 specifications. This includes 2-wire, 4-wire and EIA-422 full duplex connections inclusive handshake lines.

The high chronological time resolution of one microsecond allows a precise timing analysis of the watched communication and detailed information about the reaction times in EIA-422/485 protocols.

By the active sampling of all wire pairs bus conflicts, evoked by faulty implementation of the tri-state condition, can be clearly detected. This is also possible if the data bus is drawn to the usual idle level of 200mV by pull up, pull down resistors.

Typical application are:

- Industrial interface applications
- Fabric automation
- Industrial networks
- Building services engineering
- Maschine controlling and automation technics.
- Embedded devices

3

Features & Benefits

The MSB-RS485 analyzer offers all necessary features for an effective examination of EIA-422/485 connections. In particular for debugging, recording, tests and 'reverse engineering'.

- **Simultaneous sampling of all lines by external hardware :** Exact measurement of all EIA-422/485 signals with a precision of 1μsec and a maximum sampling rate of 16 MHz, independent form the PC operating system. No wrong time stamps or event sequences due to delayed or not answered system interrupts (software solutions).
- **Any baudrate with FLEXUART:** High-precise set and measurement of standard and non-standard baudrates in the range from 1 Baud up to 1 MBaud with a resolution of 0.1% of value. Recording and analysis with any, even unusual, baudrates. Detection of asynchronous or drifting baudrates between sender and receiver.
- **Automatic protocol detection :** Simple check and analysis of any communication with unknown connection parameters.
- **Supports 9 Bit Data words :** Recording and analysing also of protocols with 9 Bit data word length.
- **Scope-like display of the data lines :** Simultaneous display of the logical signals as well as the transferred data. That makes the error analysis and search easy for transmission errors, i.e. improper bit rates (jitter) or wrong data formats. Measuring of the real signals with the integrated bit ruler.
- **Segment-Analysing :** Direction specific analysis of single bus segments or bus participants and therewith isolating of erroneous send devices by transparent bus disconnection.
- **2 digital input/output channels :** Recording of two additional control lines (signals), output of the bus direction or bus state (active/inactive) for triggering of external measuring equipment.
- **Protocol template :** Define own rules how your data shall be displayed or visualize any application specific protocols.
- **Data analysis in real time :** Examination of the connection already while recording the data.
- **Detection of invalid line levels :** Detecting of open lines, invalid Tri-States and bus conflicts.

- **Framing, Parity, Break Detection :** Direct analysis of error conditions and the reactions of end devices thereon.
- **Pattern search with regular expressions :** Makes the search for any data sequences possible with wild card characters and time distances or pauses between data strings.
- **Integrated LevelFinder :** Finds any static level, level change or error condition. Combined with the search of defined data bytes it is a precious tool to analyze hardware protocols.
- **Integrated Lua script language :** To define, visualize, compute (check sum test) and convert the recorded data.
- **MultiView concept :** Simultaneous analysis of the recorded data at different positions with multiple tool windows. That's a very powerful help to compare the transferred data with their logical signal level or to compare different sections of the data stream.
- **Copy And Paste :** Simple copying of recorded protocol or data sequences into other applications for further evaluation or documentation purposes.
- **Data export as CSV:** For further evaluation of the logged data in Microsoft Excel or other spread sheet programs. That makes the full toolset of these programs available for statistic examination, sorting and other computations.
- **Direct display of the data stream by green Leds :** Additional indication of the data flow, quick check for a correct data connection.
- **Future-proof by modern FPGA technology :** Integrated state of the art gate array technology allows permanent advancements and adaption to different applications. The updating is done simply at start of the software.
- **Synchronize of two analysers with mikrosecond precision:** The internal Link jack provides the user with a time synchronious recording of two different RS232/RS422/485 connections.
- **Internal memory of 512 kByte :** USB transfer buffer of 512 kB for measurement data to avoid data losses while recording at high baudrates.
- **Multi-Platform Support :** The MSB-RS485 software is delivered as 'native binary' for Microsoft Windows and Linux. No emulation, no additional libraries, no installation of .NET® or Java®.
- **Multi-Language Support :** German and English language support. The selection is done automatically according to the used operating system, but can be changed manually.
- **Multi-Process Architectur :** The splitting of different functions into different programs or processes guarantees high data security while recording and provides a better adaption to the system resources and CPU load time.
- **Compact housing with USB connector :** No additional power supply necessary. Mobile operation even with laptop.

4 Specifications

General

Protocol analyzer for recording and analysis of asynchronous EIA-422/485 connections (2-wire, 4-wire, half- and fullduplex) by parallel sampling with a maximum of 16 MHz. Precise measurement of all data and bus signals with a resolution of 1μs.
Decoding of the bus lines T+, T–, R+, R– including the Tri-State with any baudrate in the range from 1 Baud to 1 MBaud.
Automatic detection of baudrate and protocol.

EIA-422/485 Measurement

- **Any baudrate with FlexUart**
  High-precise setting and measuring of standard and non-standard baudrates in the range from 1 Baud to 1 MBaud with a resolution of 0.1% of the set resp. measured value.
- **Data formats**
  Parameter for serial data transmission: 5 to 9 data bits, parity off, even, odd, constant 0 or constant 1.
- **Logical line state**
  Logical level (A-B): 1 (V+), 0 (V-), invalid (–0.7V <In <+0.7V)
- **Time resolution**
  All lines are exactly sampled and marked with 1μs time stamps, independent of the operating system of the PC.

EIA-422/485 connectors

- **Signal levels**
  Standard EIA-422/485 level $\pm$0.2V to $\pm$12 V, ESD protected inputs 12kOhm, Common Mode $\pm$7V. Detection of the tri-state level of differential signals below $\pm$0.7V
- **Bus Connectors**
  Connector: 2* Phoenix MC 1,5/ 6ST-3,5 with 2mm screw connectors, 6 pins each.
- **Intern connections**
  All connections from Port 1 and 2 are connected through high speed transceivers and are automatically switched in correspondence to the selected connection mode and data direction.

Additional features

- **Auxiliary In-Outputs**
  Two additional terminals, each individually switchable for recording of external signals or for outputting of bus status signals. Input: 0-5V, Trigger level 1,65V, 25KOhm Pull down Output: 0/5V ca. 10mA
- **Cache**
  Internal cache memory of 512 kB for buffering of measuring data when recording data with high transfer rates.
- **Status LEDs**
  Leds for displaying of: red: recording status and buffer load, green: bus data flow.

Power supply

The analyzer is directly supplied from the USB cable. The consumption is about 200mA. USB Ground is the same as EIA-422/485 Ground.
No external power supply necessary.

Supported Operating Systems

- **Windows**
  Windows 2000, Windows XP, Vista (32 and 64 Bit)
- **Linux**
  All Linux with kernel from 2.4.18 and installed Gtk2 libraries (are standard). In case of doubt you can test the Linux version from our download page. 32 and 64 Bit Systems.

Dimensions

- **Abmessungen**
  100mm x 50mm x 25mm (Length, width, height)
- **Weight**
  ca. 100g

Scope of delivery

- **Analyser**
  MSB-RS485 analyzer device.
- **Connection Set**
  Connection set consists of:
  2* 6-pin Phoenix screw connector
  4* termination resistors 120 Ohm if analyzer is end devcie
  4* short circuit wires for various connection variants.
  1* Screwdriver for Phoenix connectors
  1* USB Cable for connection to PC

- **Software**
  CD for Windows and Linux, Manual as online help and PDF document in German and English.

Requirements

- **Graphical display**
  Graphics board and monitor with at least 1024x768 pixel resolution and 16 bit color depth or more.
- **Disk space**
  100 MByte empty space for the software installation plus additional space for the recording files.
- **Memory**
  256 Mbyte or more
- **USB connector**
  One empty USB 1.1 or 2.0 connector (full speed).

5
Program Installation

The MSB-Analyzer software is available for Microsoft Windows as well as for Linux. Both versions are contained on the program CD and are both offered to your system for installation. What you have to regard is mentioned in the following chapter.

The MSB-Analyzer is connected via USB to the PC and communicates through a virtual COM port. Under Microsoft Windows the respective VCOM driver is installed automatically.
Linux distributions from kernel 2.4.18 already contain the right module ftdi sio, functional for the analyzer.
The software is bilingual (German and English) and can be installed even without analyzer, e.g. to evaluate earlier captured data. Or if you want to check out the capabilities of the Analyzer by examining the enclosed sample files.

Installation under Windows

Close all running applications before inserting the CD-ROM. Do not connect the MSB-Analyzer before you insert the CD-ROM. Connect the analyzer after the program installation is finished.

1 **Insert the installation CD-ROM**
The IFTOOLS product installer is invoked. When it does not automatically start double click onto the `My computer` symbol on your desktop or open it from the start menu. Double click onto the IFTOOLS-Setup-CD icon to start the IFTOOLS product installer.

2 **Selecting the product**
In the product list at the left side click on the relating analyzer (MSB-RS232 or MSB-RS485). Start the software installation inclusive the necessary driver with a single click onto
`installation (Version` 4.6.0`)`. Probably it takes a moment until the installer is displayed.

3 **Install the software**
Proceed according to the hints on the screen. The necessary driver is automatically installed together with the operating program.

5.1 Installation under Linux

Modern Linux distributions offer the same comfort for program installations as Windows. Just as under Windows this behavior has to be activated before. Your Linux system has to mount the CD as executable. If this is the case the installation runs as under Windows.

1 **Insert the installation CD-ROM**
The IFTOOLS product installer is invoked. Depending on the distribution you are asked if you want to activate the autorun feature. Answer with 'yes'. If the installer does not automatically start read the following chapter 'Manual installation under Linux'.

2 **Selecting the product**
Click on the relating analyzer (MSB-RS232 or MSB-RS485) in the selection list at the left side. Start the software installation with a click onto `Installation (Version 4.6.0)`.

3 **Install the software**
Proceed according to the hints on the screen. The necessary kernel module is part of all kernel since kernel version 2.4.18 and does not have to be installed.

5.2 Manual installation under Linux

If the IFTOOLS product installer does not start after inserting of the CD please follow these steps:

1 **Open a console**

2 **Copy the installation file onto your desktop**
Enter the following command:

```
cp PATH_TO_CDROM/programs/msb/msb-4.6.0-linux-installer.run ~/Desktop
```

3 **Make the installation file executable**
by setting the executable flag with:

```
chmod +x ~/Desktop/msb-4.6.0-linux-installer.run
```

4 **Start the installation file**
via mouse click (double click). Alternatively you can also run the installer in the text mode in case of the graphical installer did not start:

```
sudo ~/Desktop/msb-4.6.0-linux-installer.run --mode text
```

5.3 Installation for all users

The installation needs only `root` rights if you like to install the software for all users (system wide). For instance: if the relating software files should be wrote to `/opt` or `/usr/local`. In this case start the installation file under KDE with:

```
kdesu ~/Desktop/msb-4.6.0-linux-installer.run
```

Under Gnome either with:

```
gnomesu ~/Desktop/msb-4.6.0-linux-installer.run
```

or

```
gksu -k ~/Desktop/msb-4.6.0-linux-installer.run
```

Alternatively you can also start the installer in the text mode. This makes sense if the graphical installer did not start or you want to execute it simply by `root` or via `sudo` command:

```
sudo ~/Desktop/msb-4.6.0-linux-installer.run --mode text
```

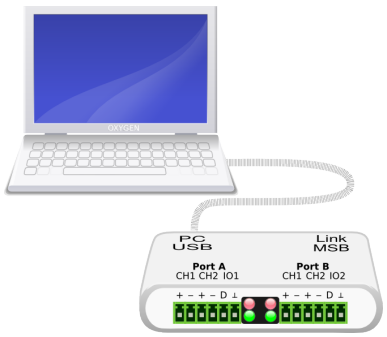
5.4 Program Updates

IFTOOLS issues software updates in irregular intervals with new features and improvements. These updates are free of charge can can be loaded from the following address https://iftools.com/download
The update are complete program versions which also contain in the Windows version the current driver. Updates can be installed in parallel to your current MSB-Analyzer program version. Windows user only have to execute the update file.
Under Linux it is necessary to make the file executable and then to start it like described above.

6 Connection of the Analyzer

How do I connect the analyzer to my PC? How do I insert it into the the connection I want to monitor? What is the meaning of the LEDs? These and other questions will be answered in the following chapter.

The EIA-422/485 specification do not specify a certain association to the connector pins. Therefore the type of connection is depending on the application.
To allow an easy adaption of the analyzer to different bus systems the MSB-RS485 is equipped with two 6-pin sockets for Phoenix connectors with screw terminals. An appropriate connection kit including plug connectors, termination resistors, wires and screw driver is enclosed.
The connector marked with PC USB is used to connect the analyzer with an unassigned USB port of your PC where your analyzer software is or shall be installed. The energy support is made through the USB cable so that you do not have to use an extra power supply. In this way you can easily use the analyzer together with a laptop in mobile operation, the current consumption is about 250 mA.



6.1 Definition of the Signal lines

Unfortunately the naming of the both twisted lines of an EIA-422/485 connection is not consistent.
The EIA-485 specification defines line A as the not inverted signal or '+' terminal and line B as the inverted or '−' terminal.
This is in conflict with the A/B naming of a number of transceiver manufacturers. Details can be found at: http://en.wikipedia.org/wiki/Rs485. Even if their naming of the signals A/B is in contrast to the standard, it is wide-spread. To

avoid more confusion the terminals of the analyzer MSB-RS485 are simply marked with '+' and '−', which correspondents to the not inverted and the inverted signal of a EIA-RS485 wire pair.
So connect the not inverted bus line with the '+' input and the inverted line with the '−' input of the analyzer.
The following table lists some of the most used line names:

EIA-485	MSB-RS485	Customize naming
A+	+	TX+, TX+/RX+, D+, Data+, (Y)
B−	–	TX–, TX–/RX–, D-, Data-, (G)
A+	+	RX+[1]
B−	–	RX-[2]

1,2 only for full duplex 4-wire systems.

6.2 Internal Signal Processing

The MSB-RS485 Analyzer has four differential inputs CH1 to CH4 which are simultaneously sampled and recorded. For every channel the physical signal is available as display of the logical states.
Two integrated UARTs handle the decoding of the serial data stream into single data bytes. These UARTs are automatically connected to two of the four differential inputs CH1 to CH4. By this variable connecting a number of connection types and analysis functions can be implemented.

In this way the MSB-RS485 Analyzer allows besides a plain tapping of the bus lines also to feed the bus through the analyzer. This is done by splitting the bus into two parts whose ends are connected to CH1 and CH2. Both UARTS care for the independent decoding of both bus segments while the bus data flows bidirectionally through the analyzer.

A combined mode where one bus is split and a second one is tapped allows the filtering of single bus devices even in full duplex 4-wire connections.

Two additionally generated tri-state signals support this way of analysis by delivering information about the validity and direction of the bus data and the common (fed through) data signal. Both tri-state signals are listed in the following table:

Notation	Mark (1)	Space (0)	invalid
Bus-Dir (Data direction)	CH2 → CH1	CH1 → CH2	Bus undriven
Bus-Signal	fed through Bus-(Data)-Signal	fed through Bus-(Data)-Signal	no data

6.3 Digital In/Outputs

The MSB-RS485 Analyzer offers two additional digital IO-channels which can be optionally used as auxiliary inputs for recording of logic signals or as outputs for indication of status information.
The latter allows the output of the bus data direction and the validity of the bus either of single segments or of the fed through bus lines. The following settings are possible, separately for both IO-channels:

IO-Type	Description
Input	Input with pull down resistor
Input	Input with pull up resistor
Output	Output static 0
Output	Output static 1
Output	Bus data direction between CH1 and CH2 (segment analysis): 0 : CH1 → CH2 1 : CH2 → CH1
Output	Activity of the bus through CH1 and CH2 (segment analysis): 0: inactive (Tri-state), 1: active
Output	Bus activity of CH1 1: active, 0: inactive (tri-state)
Output	Bus activity of CH2 1: active, 0: inactive (tri-state)
Output	Bus activity of CH3 1: active, 0: inactive (tri-state)
Output	Bus activity of CH4 1: active, 0: inactive (tri-state)

6.4 Bus Termination and Tapping

EIA-422 are usually designed as full duplex point-to-point connections, sometimes together with additional line pairs for hardware handshake.
Whereas a EIA-485 connection is generally implemented as a multi-master capable 2-wire system (half duplex) or as a full duplex 4-wire system based on a master-slave configuration.

All serial bus systems have in common that the signals allow no inferences on direction and source of the data.
The MSB-RS485 analyzer offers the possibility to split the inspected connection into two segments and/or bus participants.
With this so called 'segment-analysis' the data of a single (or several) bus devices can purposefully be monitored independently of the rest of the bus and can be directly assigned without regarding the protocol.

The MSB-RS485 Analyzer has 4 differential Inputs and 2 integrated FLEXUART's available. These Uarts are connected to the inputs according to the selected connection mode and process the decoding of the serial data stream into single data bytes.

To make the connection and assignment as simple as possible the analyzer offers a selection of various combinations for bus system and taping (the connec-

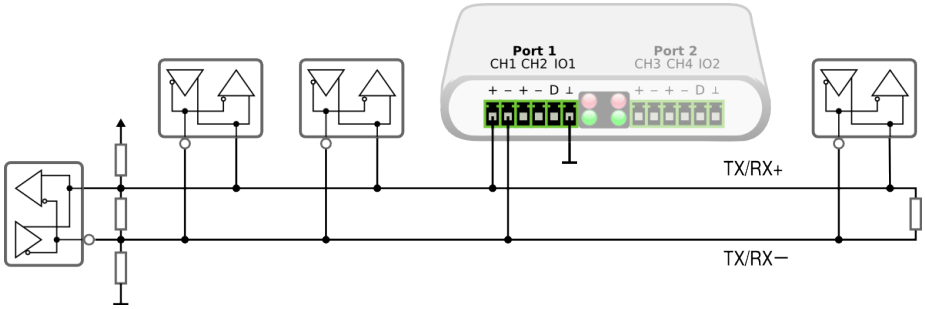
tion mode or short wiring), which are in accordance with the following variants.

6.5 Tapping 2-wire system

For connecting a 2-wire system the bus resp. the wire pair is simply connected to the terminals at Port1, CH1. CH2 and the pins of Port2 remain unconnected. Connect the inverted line of the bus to the '–' input of CH1 and the not inverted line to the terminal '+' of CH1.

The name 2-wire system implies the use of the line pair only but the correct treatment of ground is mandatory. If the inspected bus has a signal ground line connect it to the correspondingly marked terminal at port 1.
This simple method of tapping does not need additional terminating resistors.

Tapping 2-wire
for a half duplex bus
(Modbus, ProfiBus)



In this connection mode the analyzer records all transmitted data independently of source and direction. To get more information about the sender of the data you have to know the corresponding used protocol.

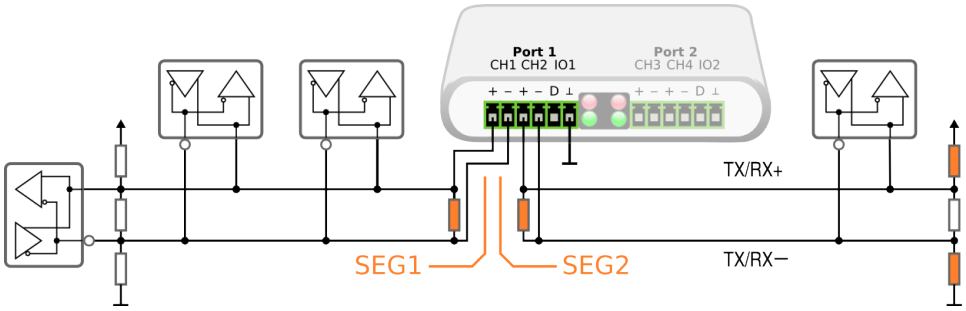
6.6 Segment Analysis 2-wire system

In contrast to the plain tapping of the data signal the MSB-RS485 is inserted into the bus. In doing so the bus is split into one segment on the left side and one segment on the right side of the analyzer.
This kind of wiring is more complex but it has some advantages over the plain tapping.

The analyzer becomes the interface between any two bus segments. The data, flowing through this interface, are collected together with their direction so that they can be clearly assigned to the corresponding segment. If the segment consists of only one bus device the data sent from this device can be easily assigned to this device independently from the remaining bus communication - even without having to know the used protocol.

Split the bus at the required point and connect the wire pair of the first segment to the terminals of the first analyzer channel Port1, CH1 (CH1+, CH1–)
The second bus segment is connected to the second analyzer channel Port1, CH2 (CH2+,CH2–).
The bus is now split into two segments. Please note that you possibly have to terminate the new bus ends. In this case you can directly connect the resistors to the terminals of CH1, CH2.

The same applies for pull up/down resistors. By the splitting the bus one segment is now without these resistors for setting the idle level. They are normally attached at one bus end only and their values are system dependent. Please check if they have to be added.



2-wire Segment Analysis
for half duplex bus (Modbus, ProfiBus)

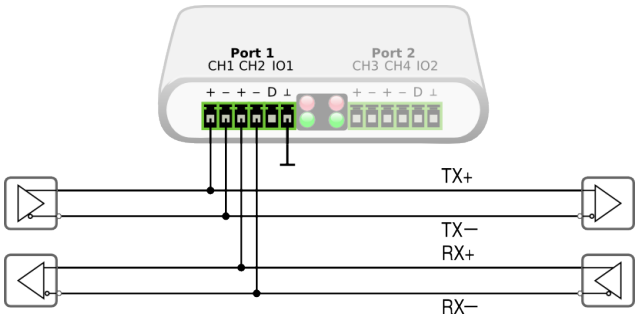
The analyzer records the data of both segments independently while the data of both directions are passed fully transparent. Data of the first segment are marked as coming from channel 1 with the internal name 'A', data of the second segment are marked as coming from channel 2 with the internal name 'B'. A and B are initially two different data sources within the analyzer and are assigned to the physical input channels according to the selected wiring.

6.7 Tapping 4-wire system

In point-to-point connections like used for EIA-422 transmissions over long distances (EIA-232 replacements) only two bus devices are available communicating over two different send and receive channels. A double tapping is sufficient and guarantees the correct recording of the data direction.

This kind of wiring is also used for analysis of full duplex EIA-485 connections (as DIN-Messbus, Master-Slave) if you do not need to watch a special bus device singularly and if you can assign the data to the bus participants by evaluating the protocol.

Connect the send line pair to the terminals CH1+, CH1- of Port 1 and the receive line pair to CH2+,CH2- of Port 1.



Tapping 4-wire
full duplex EIA-422/485 (Din Messbus)

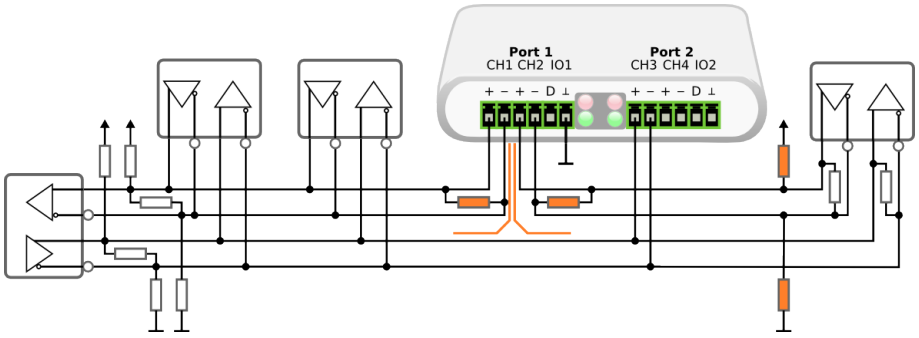
All data from channel 1 are named as A and all data from channel 2 are named B.

6.8 Segment Analyse 4-wire system

Bus systems with full duplex 4-wire connection (Master-Slave bus, Din-Messbus) also use separate send and receive channels.
While the master is connected as sender (Masterbus) to the receivers (Slaves) these return their answers on the second channel (Slavebus) to the master. To monitor the send data from the master a single tapping of the master bus is sufficient.
In contrast the slaves share one channel to send their data back to the master. With the help of the segment analysis a singular device can intentionally be separated and its communication with the master monitored without regarding the other devices.

Like for the 2-wire segment analysis you have to split the slave bus at the appropriate point.
Both segments of the slave bus have to be connected to CH1 and CH2 of Port 1. Please note that possibly the ends have to be terminated as explained in 6.6.
Additional you have to consider about existing pullup resistance. The tapping of the master bus is connected to CH3 at Port 2. A termination is not necessary.

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



In this configuration all data sent by the slaves on both segments are gathered at CH1 and CH2 and internally named as A. The data from the master, received at CH3 are named B.
The assignment of the data A to a segment or a certain bus device (CH1 or CH2) is done via an additional internally generated bus direction signal. This signal can be evaluated to visualize and differing the sender, resp. the active sending segment.

6.9 Signal assignment

The MSB-RS485 Analyzer has 10 data display channels for the visualization of the recorded information.
Two data channels are used for the display of both bus data A and B which are generated by the UARTs.
Further 8 logical channels are used to display the tri-state levels of the differential signal inputs CH1 to CH4. They also display the bus activity and data

direction and the two digital auxiliary inputs. The assignment of the display channels varies according to the selected connection mode (wiring).

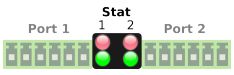
Gray entries indicate signals which are recorded by the analyzer but are not used in the selected connection mode. However they can be used for recording of additional signals like handshake lines.
The following table shows the available signal information. You do not have to use this table for your data evaluation, the analyzer software automatically names the display channels depending on the selected wiring.

Display channel[a]	2-wire Tap	2-wire Seg	4-wire Tap	4-wire Seg
Data A (Data A)	Data from Bus at CH1	Data from Bus segment at CH1	Data from Bus at CH1	Data from Bus segments at CH1 + CH2
Data B (Data B)	Data from Bus at CH2	Data from Bus segment at CH2	Data from Bus at CH2	Data from Masterbus at CH3
Signal 1 (CH1)	Logic signal at CH1	Logic signal at CH1	Logic signal at CH1	Logic signal at CH1
Signal 2 (CH2)	Logic signal at CH2	Logic signal at CH2	Logic signal at CH2	Logic signal at CH2
Signal 3 (CH3)	Logic signal at CH3	Logic signal at CH3	Logic signal at CH3	Logic signal at CH3
Signal 4 (CH4)	Logic signal at CH4	Logic signal at CH4	Logic signal at CH4	Logic signal at CH4
Signal 5 (BDIR)	unused	Data direction CH1 ⇔ CH2	unused	Data direction CH1 ⇔ CH2
Signal 6 (BSIG))	unused	Logic signal CH1 + CH2	unused	Logic signal CH1 + CH2
Signal 7 (IO1)	IO1	IO1	IO1	IO1
Signal 8 (IO2)	IO2	IO2	IO2	IO2

[a] The short notation as displayed in the control program in parenthesis

6.10 Lightment elements LEDs

The MSB-RS485 analyzer has four LEDs to display its operating status and the status of the data recording. They are located between both 6-pin Phoenix jacks (Port 1 and 2) and are marked with Red1, Red2, Green1 and Green2.
The red LEDs serve for the signaling of the recording status and the internal buffer load while the green LEDs show the state of the connection and data flow.



Control LEDs
for State and record control

Green LEDs
The green LEDS show the bus states. LED1 is assigned to data channel A, LED2 is assigned to data channel B.

1 **LED is off:**
Undefined bus state, drivers in tri state or lines are twisted (interchanged polarity).

2 **LED is on:**
Active bus, correct connection.

3 **LED flickers, mostly on:**
Active bus with data transfer, correspondents to the normal EIA-422 operation.

4 **LED flickers, only short pulses:**
Active bus with data transfer, but wrong polarity or EIA-485 bus with rest (tri-state) conditions. The latter is the normal condition for EIA-485.

Red LEDs
The red Leds serve as a display for the operating condition of the MSB-RS485.

1 **Both LEDs permanently on:**
The MSB-RS485 was not yet initialized by the PC. Data is not fed through (segment analysis).

2 **Both LEDs blink alternatively:**
The MSB-RS485 was initialized but is not yet active, that means no recording was started.

3 **Red 1 is on, Red 2 is off:**
Recording is active, the PC logs all interface events.

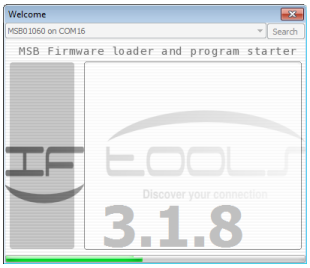
4 **Red 1 is on, Red 2 is blinking:**
The loading of the internal data buffer is displayed. The filling degrees $\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$ are indicated by different length of pauses between the blinks. The more full the memory the shorter the pauses.

5 **Both LEDs blink at the same time:**
The buffer memory is full and recording data gets lost. The duration of the data loss is recorded.

7
Program start

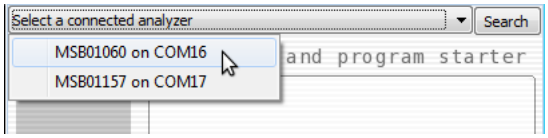
Selection of the interesting events and the recording mode (continous or Fifo-mode). Store or load recorded data. All these functions are controlled through the main program.

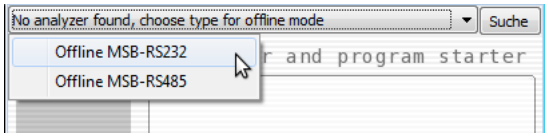
The Firmware of the MSB-RS485 ist not installed in the device but has to be loaded after powering up. This takes place only once. As long as the device is powered from the USB connection the firmware is kept active.
Therefore the loader appears as soon as you double click onto the MSB-RS485 desktop icon.
The firmware loader automatically detects if an analyzer is connected and if the firmware is already loaded or has to be transferred. After the MSB-RS485 was identified and the firmware was successfully transferred (observable at the progress bar in the lower part of the dialog window) the MSB-RS485 control program starts automatically.
If there are more than one analyzer connected with your PC the Loader will show you a selection list of all detected devices.



The first start
loads the firmware into the device

If no MSB-RS485 was found, even though it is connected to your PC, read the hints for Trouble shooting (Windows D.1, Linux E.1) in the appendix.
If you simply forgot to connect the MSB-RS485 to your PC, just connect it now and click on the 'Search' button to update the list of the detected analyzers.
You also can work without the MSB-RS485, e.g. to evaluate recorded data or to work through the Tutorial. Because the program cannot detect the kind of analyzer (MSB-RS232 or MSB-RS485) in offline mode, you have to choose the wanted type from the selection list.

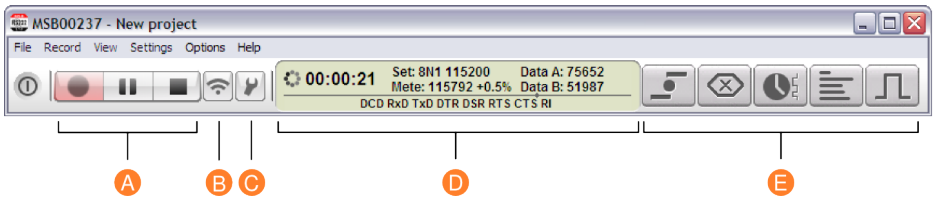




The MSB-RS485 software uses a multi process architecture. That means, that the program does not run in a single window but starts special tools according to the different tasks.
In view of this feature the start of a small control panel may seem poor. But the software shall not confuse you with not necessary windows, multiline toolbars and nested menus. Instead it shall offer you a set of easy to operate tools which are appropriate for your application.

The MSB-RS485 control program is your ≫cockpit≪ to supervise the analyzer. With it you start a recording, save or load records or projects and open the different analysis tools to examine the recorded data.

7.1 User Interface



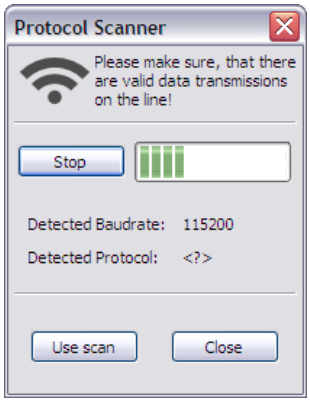
A	**Recording control:** Easy start, pause or stop a recording.
B	**Protocol Scanner:** Automatically detecting of (unknown) baudrate and protocol.
C	**Settings:** All necessary settings for the recording available with one click. No long winded navigation through complex menu items.
D	**Recording control:** Clear display of the recorded data/events, settings, record time and record state. More information with a right mouse click.
E	**Analysis tools:** Start the proper view with it's last settings.

7.2 Select kind of connection

Select bus wiring

The MSB-RS485 Analyzer has four differential inputs which are used as data or signal loads, according to the selected wiring. Before starting the first recording you have to inform the program about how the analyzer is connected to the EIA-422/485 bus. The only thing you have to do is to select the connection mode in the setup menu. Also see section 7.5.

7.3 The first start



Protocol Scanner
Automatical detection of baudrate and protocol

Imagine the control program as a kind of recorder. To record a data connection you need at first not more than the communication parameters (baudrate, protocol). All further tools to display the transferred data or their physical level can be optionally opened or closed without influencing the recording.
The MSB-RS485 analyzer contains a so called FLEXUART core, an specially developed decoder hardware, for the serial data transmission which allows not only the measuring of the baudrate but also the detection of the used protocol. The only thing you have to do is to click onto the button to open the protocol scanner dialog.

Automatical protocol scan

A correct detection of the connection parameters implies an appropriate transmission. It is sufficient to receive data at Port 1 or Port 2. A recording does not

have to be started.
Start the automatic detection with a click onto the 'Start' button of the scan dialog.
After starting the protocol detection the MSB-RS485 analyzer at first measures the baud rate of the data at Port 1 or Port 2. In the further process the data stream is analyzed and the correct number of data bits and parity is evaluated. The complete process lasts only a few seconds and can be repeated at any time by clicking the start button. As soon as the paramters are correctly detected you can adopt the found settings with the button 'Use scan'.

Manual Protocol setup

If you already know the baudrate and protocol you can directly set them in the settings dialog. The MSB-RS485 software memorizes these settings so that you do not have to enter them any more.
All relevant parameters for the recording can be reached with one click onto the button. The most important settings are immediately shown. Settings for advanced users or more specific applications can be reached in the page selection of the dialog and are discussed in chapter 7.5.

Start/stop a recording

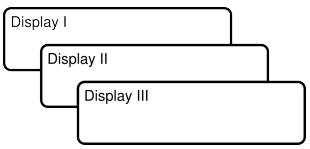
Record Pause Stop
Recording control

As soon as you determined the communication parameters one click onto the record key starts the recording . The record key starts to glow red and the active symbol in the display starts to turn.
You can halt the recording at any time by clicking on the Pause button. The recording of occuring events is discontinued until you continue the recording by another click on the Pause button.
If you want to end the recording press the Stop button. The recording is not deleted but finally stopped. If you want to start a new recording the program reminds you to first save the recorded data of the last session.
The MSB-RS485 analyzer software allows to examine the data even while recording. In this respect you will stop the recording not before you want to start a different one or to save it for a later examination.

> **Test the software without analyser**
> You can test the program even without a connected analyzer. Simply load a sample recording with Ctrl+O from the example folder in the installation directory.

7.4 Status display

The central part of the control program is the display of the current recording.
To indicate all information clearly arranged the display can be operated in three different modes. The selection is simply made by a right mouse click onto the display.
In addition to the connection data also the available recording capacity is displayed. The necessary disc capacity is depending on the data traffic and the events, selected for recording. An estimation of the space consumption shows the marker (dot) on the horizontal dividing rule. It indicates the empty space (right hand of the marker) in relation to the total available space.

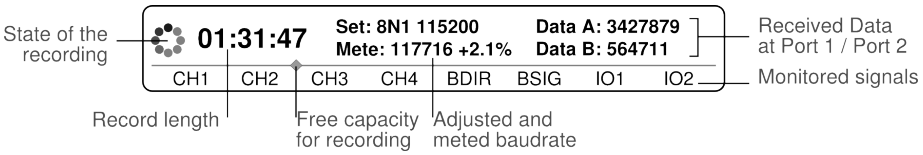


Toggle the display
with a right mouse click

The default directory is the usual Windows or Linux temporary directory. You can change it by calling the program call with an addition parameter (see also section 7.16 Additional program arguments).
The speed of the moving marker is depending on the quantity of the occuring (and selected) events as well as on the size of the empty space in the used temporary directory.

Display I

It contains in the lower part the set connection parameters (Set: baud rate, data length, parity, stop bits), the measured baudrate (Mete:) and the for logging activated input channels resp. signals.
Additionally the quantity of the transmitted data bytes are displayed depending on the direction, bus or bus segment.
The time display corresponds to the timt point of the last occurred event, relative to the start of the recording.

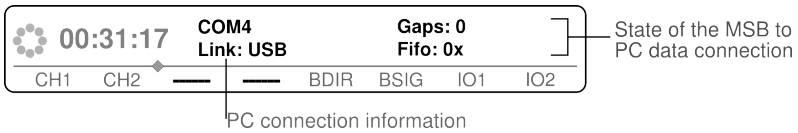


Display II

The second display informs you about the total sum of transmitted data bytes and about the quantity of the remaining events as level changes and alternations of the bus direction a.s.o.
In this display mode the time indication is not related to the last occurred event. The time is the running recording time, independent of any events.

Display III

The third display alternative serves as a control display for the connection of the Analyser to the controlling PC.
The analyzer MSB-RS485 is controlled and supplied directly through the USB connection. The USB connection is done via a virtual COM port and is shown as a normal COM connection under Windows. For Linux it is typically `/dev/ttyUSBx`.



The fields *Gaps* and *Fifo* indicated if the analyser has recorded or sent more data than the PC could handle (as a result of a too slow connection). Normally both values should be zero. Other values indicate that either the internal buffer of the analyser (Gaps) resp. the Fifo of the serial ports (UART) could not handle the data rate (overflow, lost data). In this case you should reduce the number

of recorded events.
All signal and and data lines can be individually enabled and disabled for logging. Inactive are shown as lines.
In the examples above the level changes of the input channels CH3 and CH4 are not logged.

7.5 Config a recording

In the program defaults all signals are activated and prenamed, the connection parameters are set to 8N1, 115200Baud.
These defaults probably do not correspond to the connection which shall be monitored. Before you start a recording these parameters have to be changed according to the connection settings.
You get access to these parameters through Settings→Configure Control Program... or by click onto the set symbol on the left side of the display.
Because some settings directly influence the logging they are deactivated while a logging session is running. The respective setup menus are displayed in gray.
The setup dialog is divided into the following sections:

- Connection
- Bus Wiring
- Signals
- Record
- Auto Save
- General

The recording of the bus line(s) takes place as data bytes and additional as the recording of the logical signal levels.

Connection

These settings concern the analyzed connection, not the connection between the analyser and the PC! They are mainly important if you want to record the transmitted data bytes.
This includes the baudrate and also the numbers of the databits and the parity setting. The count of stopbits doesn't matter. The analyser takes care about the stopbits automatically.
If you are unsure about the connection parameters just let the integrated protocol scanner detecting the right settings for you. .

Connection setup
baudrate, word length, parity

Additional to the standard baudrates, the MSB-RS485 analyser also supports any rate in a wide range of 1 Baud to 1 MBaud.
To use an more special rate like 123456, just input it in the baudrate field. Or select a standard baudrate by click on the ▾ button. Valid entries are 1 Baud to 1 MBaud.

Besides the common data lengths the MSB-Analyzer also allows you the record 9 bit data transmissions, which are often used for address decoding or to mark a telegram start.

Please note that a 9 bit data length excludes any other parity except for none.

By default, the display of the control program shows the measured baudrate were alternatively the detected data at CH1 or CH2 are used for the evaluation.

Choose a Bus wiring

Bus wiring

It is decisive for the signal assignment how the MSB-RS485 analyzer is connected to the examined bus. Since the connection can not be automatically determined you have to inform the analyzer about the chosen wiring.
Depending on the connection setting only a part of the differential inputs are used. The unused inputs and the digital auxiliary inputs can be used freely in your application. Not available signals are marked with 'disabled'.
The setup menu shows a respective graphic and a table for the signal assignments. The signal names are automatically assigned. In the setup menu of the signal names (32) you can change this behavior and assign own names. The signal assignment is separated into:

- **2 Data channels**
  These channels contain the decoded data of both UARTs. 9 bit data are supported and displayed as well as occurred transmission errors like parity and framing.
  Depending on the selected wiring a data channel can also contain the data of several differential inputs. In this case the inputs are explicitly listed with a '+' For example CH1+CH2.
- **8 Logical signal channels**
  All differential inputs CH1 to CH4 are additionally displayed as a 'plain' logic signal independent of their data decoding. The four inputs are directly shown in the four signal channels 1 to 4.
  The signal channels 5 and 6 have a special status. For the selected segment analysis they display additional internally generated logic signals. These are the bus-direction between CH1 $\leftrightarrow$ CH2 (signal channel 5) and the combination of the logic signals from both inputs CH1 and CH2 (Signal channel 6).
  The signal channels 7 and 8 are assigned to the two auxiliary channe
- **Digital auxiliary inputs/outputs**
  As the name implies both channels can be individually operated as input or output. That allows the recording of additional signals or the output of the current bus state (bus validity, activity or direction). Reasonable if you need a signal to trigger external measuring equipment.
  By default both terminals are set to open inputs with pull down resistor. A detailed description of the possible settings can be found in chapter 6.3.

Signal und Name
settings for the record

Signals

You can select and rename each of the sampled input channels or signals separately. The latter one can be done even with a running recording.

The chosen bus connection mode specifies reasonable signal names (default bus connection).
It is up to you which one you choose or if you define own names as 'User defined'. In this case set the signal naming to 'User defined' and enter the new

names for each signal.
The names Sig1 to Sig8 are used as place holder. Every name can consist of a maximum of 7 characters, allowed are: all digits and letters, the underscore, colon, and full stop.
The modified signal names are automatically adopted by the control program and all analysis windows.

You can individually enable or disable the line events (signal alternations) to be monitored by the Analyser by setting or removing the checkmark beside the relating signal. As default all level changes detected on the input channels CH1 to CH4, both auxiliary digital inputs and the decoded results of the two UARTs are switched on.
Please note, that the selected signal names are shown. If you have chosen 'User defined' but haven't entered names then here appear - no names (blank)!

The decoding of data by the UARTs is done independently of the level change recording. If you need the data only but not the logical signal you can deactivate the recording of the channels CH1 to CH4 because each level change is stored as an additional event and strikingly increases the quantity of the recorded data.

Please keep in mind that the more unnecessary events you admit the more (needless?) data is stored onto your hard disk.

Record mode

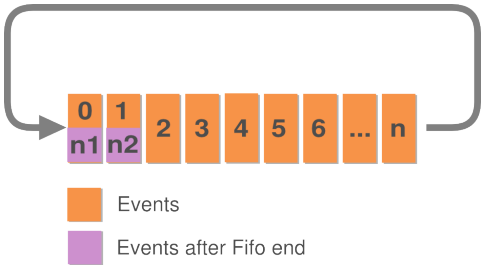
Choose a record mode
synchroneous, continous or loop recording

For troubleshooting of serial connections you often get the problem that you do not know when the error occurs but you need a sufficient big quantity of data to get a statement about potential reasons for the fault.
Of course you can run the logging up to the occurence of the fault. But this can cause a rapidly increasing amount of data. With 115200 Baud and recording of all level changes this can mean 2MBytes data per second!
Therefore the analyzer supports 2 modes of logging:

1 **Continuous recording:**
In the continous mode all occuring events are stored until the recording is stopped. This mode is appropriate if you want to watch and analyze the data stream already while recording.

2 **Time loop with Fifo mode:**
In the Fifo Mode a certain amount of data of the last (before stop of recording) occured events is stored. The amount can be defined by setting the maximum size of the recorded events (1000...1000000 events) or by setting a time limit (10...600 seconds).
This behaviour quasi correspondends to an analogue endless tape (used with

observation cameras). With this tape always the last time, defined by the tape length, is recorded. In this case you can define the 'tape length' in a given range.



Please note that in the Fifo mode no analysis tool can be used while recording. The reason is that the tools need a random access to the recorded data which is not possible in the Fifo mode. In this mode data is always overwritten from the beginning of the buffers. As the Fifo mode is normally used for recording with later analysis this behaviour is not necessarily a disadvantage.
As soon as you stop the recording all recorded data are normalized. That means that they are sorted according to their time stamps and can then be analyzed as usual.

Time synchron recording

In the program defaults each MSB-Analyzer works autonomously unless it received so called synchron impules on its MSB-Link jack from a connected 'Master'. If you like to record two independent connections at the same time, for instance a RS232 and RS485 port of a level converter, you have to choose one of the analysers as the record 'Master'.
You can see the current analyser status in the display. A 'Master' above the running record time indicates that the device works as the master, a 'Slave' means that the analyser is linked as the slave. In the latter case all settings are disabled since there is only one Master allowed.

You can use the 'Flash connected analyser' if you are in doubt which one you are currently setting.

Adapt the record date and time

Sometimes you want to adjust the date and start time of an analysing record perhaps if you examinate the record in another time zone or need to adapt it to the date and time of a second comparing record.
To do so, just click the 'Adapt now' button and enter the new date and start time.
Close the dialog with 'Ok'. The Views automatically refresh their display.
You can always switch back to the original record date in the same dialog. The new date/time doesn't change the original record. The software only uses the new value as an offset added to the initial record time and date.

Save new record date
The modification of the record date and time doesn't alter the record file. The new date/time is only temporary for the current session. If you want to store the modifications permanently you have to save the record.

Autosave

Save a record
automatically after each stop

The amount of data can rapidly increase during a record. Therefore the program only stores the data if the user explicitly want to save it in a file.
But there are conditions which require the storage of the record. For instance: If you like to make sequent records for a later analysis or if an analyser in slave mode isn't accessable.
For both cases you can preset an automatical storage of the record. The storage always takes place:

1 After stop of a synchronous record
2 After each stop of a record

The place and folder of the saved files are freely selectable. The program creates a unique file name according to the serial number and the record start date-time which prevents to overwrite existing files. But you can add an additional prefix for a better identification or classification.

General

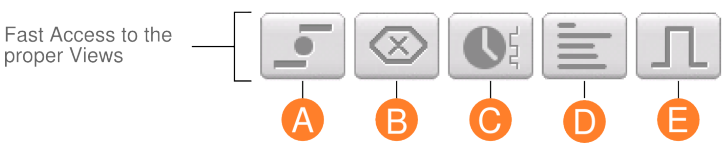
Common settings
like warnings, taskbar behaviour and external views synchronisation

This page is intended for general settings. Among other things you can switch of the security question for not yet stored data and/or suppress the display of single views in the task bar or the task changer. The latter is possible for Windows only. Linux desktops usually group the program views within one taskbar entry.

As a special feature the software offers you to synchronize the Views of two running MSB-Analyzer programs with each other. This comes in handy if you like to compare and analyse two synchronous records. All views of boths records interact as it used to be in a single recording.
For this case you have to allow the external synchronisation first.

7.6 The analysis tools

The control program solely makes the collected data available. The actual Display and analysis of the data is done by analysis tools - separate program modules to visualize the data at different time points and in different display modes.
You can open any number of analysis windows by either using the below mentioned short commands or by clicking on one of the quick start buttons at the right side of the display.



A	(P.49)	**Virtual Ledtester:** Das virtual counterpart of a real ledtester.
B	(P.51)	**Data View:** Data dump of the transmitted data with special search features.
C	(P.65)	**Event View:** All line changes in a clear look, search for line modifications.
D	(P.77)	**Protocol View:** Display any protocol with your own definition.
E	(P.145)	**Signal View:** Digital Scope like view of all lines.

7.7 Save a recording

Independend of the status of the recording (aktiv, paused or stoped) you can save the data, collected so far, in a file.
Press the keys Ctrl+S or select in the file menu the entry Save→Save recording. In the opening dialog you can enter a new file name (The extention .msblog is automaically added) or you can overwrite an already existing recording. This file contains all information about the selected and recorded events and data bytes. Settings of the control program and opened analysis windows are separately stored as a project file.

Every time you save a recording the choosen file is stored in the list of last opened recording files and can be loaded at any time. More information can be found in chapter 'Last opened Recordings and Projects'.

Save a special section

To save any section of the recorded data use the event monitor and mark the interesting range. To save the tranmitted data bytes only or a part of it use the data monitor and mark the interesting range.

7.8 Save a session as a project

A session contains the current state of the opened analyser program. This includes all current settings and views which represent the program on the screen. That means that beside the connection parameters also position, size and content of all opened analysis tools and all the marked regions are combined into the session.
You can save the session at any time by pressing: Ctrl+Shift+S or by selecting Save→Save project in the file menu.
When a session is saved also the data, recorded until this time, are saved in a separate file with the same project name, but with a different extention.

Separate files for project and record

Project files always have the extention $\star$.msbprj, the recorded data files the extention $\star$.msblog.

Accordingly a record data file is also loaded (if available) when the project file is opened.
With it you have all informations you need to resume an analysis of recorded data at exactly that point where you paused or finished the examination before.
Saved projects are also managed in the list of last opened project files, see 'Last opened Recordings and Projects'.
Each session can saved as a independent project template.
For this purpose clear all recorded data by New→New record in the file menu or press Ctrl+N. Afterwards save the session under a name of your choice.

7.9 Open an earlier recording

A devision between project files and record files was intentionally made. The reason is that you can load an earlier data recording into your current project without loosing your current settings.
Press Ctrl+O or click on Open→Open Record in the file menu to load the data into your current session. Please note that tThis can be done only if no recording is running and that your recorded data are overwritten.

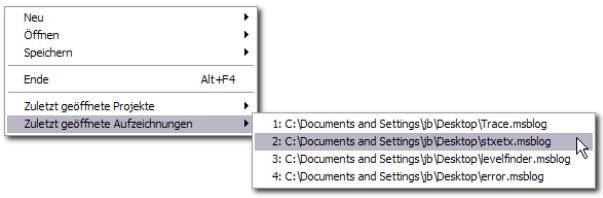
7.10 Open an earlier session (project)

Press Ctrl+Shift+O or click on Open→Open project in the file menu to open a saved session.
The control program loads the associated recording, places the analysis tools and makes the corresponding settings. In short it restores the program state as it was at the moment of saving the session.

7.11 Last opened recordings and projects

As mentioned earlier all saved recordings and sessions (Projects) are listed in two separate lists. You get quick access to the files you used last.
The lists contain the names in sequential order so that the newest files are on top. All files are listed with full path to clearly identify them.

Click on 'Last opened Recordings' in the file menu and select the appropriate one. This is the same like open a recording with the open dialog but is much faster because you do not have to move through the different menus and directory trees.
If the choosen file is no more available, i.e. because you deleted it, you are asked by the program if the file shall be removed from the list. If the file really does no longer exist you can answer with 'yes'. But if the file is on a data medium that is only temporarily unavailable you can answer with 'no' and the entry is kept.



7.12 Drag and drop

You can load any record or project file by simply drag and drop it into the application. Just drag the wanted file from your file browser or desktop into the control program.

This will replace the current session with the data of the new dragged file. In case of a project file also all stored session settings are restored including all Views.
Please note that drag and drop isn't possible during an active recording.

7.13 Connecting multiple analysers

You can use multiple analyzer at one PC at the same. Furthermore it is possible to compare the data and events of one analyzer to the data of another one or to data recorded earlier. Every control program acts independent to the others. To explicitly connect the control program to a certain MSB-RS485 you have to start the program with declaration of the serial number of the MSB-RS485. The serial number is attached to the bottom of the instrument and is displayed in each window frame of the running program. It has the following format: `MSB#####`.
If you do not add the serial number the program connects to the first instrument it finds on the USB. This is the default behaviour.
Depending on the PC and the sequence of search the analyzer found in each case may vary.
To select a certain analyzer by double click on the start icon proceed as follows:

1. Right click onto the MSB-RS485 Icon and select the entry *Copy*.
2. Right click onto an empty space of your desktop and select Insert to add a copy of the start icon.
3. Rename the Copy to e.g. `MSB#####` (`#####` is the serial number).
4. Right click on the renamed icon and select the entry *Properties*.
5. Add in the field *Target*: to the call of the control program the parameter `-nMSB#####`. I.e. something like this:
   `C:\Programme\msb-4.6.0\msb_serv.exe -nMSB#####`
   resp. for Linux: `/home/USER/msb-4.6.0msb_serv -nMSB######`
6. Click on OK to apply the addition.

Take care that the added serial number of your analyzer is the same as the one on the instrument. Otherwise the analyzer will not be found and an error message will be issued.
Following this procedure you can define an own start icon for each analyzer.

7.14 Automatical start after computer boot

The analyzer can be started automatically and set into the logging mode after booting of the computer. That means in detail:

1. As soon as the windows boot process is finished an analyzer is searched and load with the firmware.
2. Subsequently the analyzer is set to the recording state and starts logging the connection. For this application the last connection settings are used.
3. Every new recording is stored in an own logging file. Its name is combined from the serial number of the analyzer and the start date and time of the recording. For example: `MSB00237-20120702093107.msblog` means a record taken on 2th July 2012 at 09:31:07.
4. The analyzer software closes the record file as soon as the computer is shut down.

Activate the autostart feature

Windows automatically starts all programs, which are located in the autostart folder. An additional parameter for the analyzer program is necessary to start logging after searching and loading the analyzer. At the same time this parameter takes care, that the logging and its file is correctly closed before the system is shut down.
Open the autostart folder of the active user with the file explorer. Commonly it is the folder:

```
C:\Documents and Settings\Username\Start Menu\Programs\Startup
```

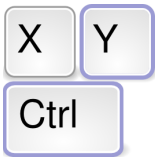
Copy the MSB-RS485 start icon from your desktop into the autostart folder (copy, not move). Then right click the new icon in the autostart folder and select *Properties*.
Add to the target entry the parameter -a, i.e.:

```
C:\Program Files\msb-4.6.0\msb_serv.exe -a
```

Click on Apply and OK to save the change. When the computer is rebootet the analyzer program is executed and a new recording is performed.
Please note, that the last events of the recording are possibly not stored when the computer is incorrectly switched off (power off without shut down command).

Autostart with high data increase
Please not! The time to store the data depends on the amount of data and can take several minutes with very large volumens of data.
An alternative would be to use the command line tools from chapter 19 and put an according script or batch file in the autostart folder.

7.15 Short commands



Short commands
of the most important functions

Action	Short command
Online help for the control program	F1
New recording	Ctrl + N
New project	Ctrl + Shift + N
Open recording	Ctrl + O
Open procekt	Ctrl + Shift + O
Save recording as...	Ctrl + S
Save project as...	Ctrl + Shift + S
Start recording	R
Pause recording	P

Stop recording	S
Open a virtual Ledtester	Ctrl + Alt + L
Open a Data View	Ctrl + Alt + D
Open a Event View	Ctrl + Alt + E
Open a Protocl View	Ctrl + Alt + P
Open s Signal View	Ctrl + Alt + S
Save settings and close program	Alt + F4

7.16 Additional program arguments

The MSB control program can be called with a series of additional parameters to set explicite defaults like language, offline mode or the typ of the connected analyser.
In most cases the default setting (automatical search an initializing of the analyzer) is sufficient. If the analyzer is not found (this can happen if Bluetooth converters are used because they reserve some COM ports) or if you want to set another directory for storing your temporary logging data you can change this with the following program parameter.
You can add additional program arguments to your desktop start icon like described in chapter 7.13.

Parameter	Description
-a	Starts the analyzer in autostart mode. That means that after loading the firmware into the connected analyzer the device is immediately switched into logging mode and all recording files have serial numbered names.
-D *directory*	Set the working directory.
-e	Starts the control program with the default settings. All stored program and session settings will be ignored.
--force	Forces the program to use the given port name and serial number in cases the automatical detection of the analyzer fails. Please note! This parameter only works with a given port name and serial number, i.e. `msb_serv -pCOM12 -nMSB01234 --force`
-i	Forces the loading of the firmware even when the MSB-RS485 is already loaded.
-j	Forces the program windows to appear on the current screen. Use this parameter, if you want to open a project file, which was saved on a workstation with more than one monitor. (And therefor the windows doesn't appear, because they are saved on a non visible screen).

-l *language*	Select the language. Values for *language* are: 0: System default, depending on your operating system, 1: english, 2: german Syntax: `msb_serv -l1`
-n *serno*	Select a analyzer by it's serial number *serno*. Important, if you are connecting more than one analyzer at the same time. Syntax: `msb_serv -n MSB12345`
-o *type*	Starts the control program offline using the given analyzer type (and suppress the selection dialog). A connected analyser is not searched for. Recordings are not possible but saved data can be examined. Syntax: `msb_serv -o typ` Valid types are: MSB-RS232 or MSB-RS485, for instance: `msb_serv -o MSB-RS232`
-p *port*	Virtual com port to be used. For example COM1 (Windows) or `/dev/ttyUSB0` (Linux). Please note! You have to pass the serial number of the connected analyzer with parameter -n (see above) otherwise the program will use the default MSB00000 for setting storage. Syntax: `msb_serv -p COM1 -n MSB12345` (Windows) respectively `msb_serv -p /dev/ttyUSB0 -n MSB12345` (Linux)
-r *number*	Reduces the firmware transfer speed by the given number. Default is 0 (full speed), maximum value 100.
-T *directory*	Presetting of the directory where the temporary logging data is stored. By default this is `C:\Documents and Settings\Username\Local Settings\Temp` (Windows) respectively `/tmp` (Linux).
--verbose	Stores a report file (AnalyzerScan.txt) about the analyzer detection process on the desktop. Send this file to support@iftools.com when the software fails to recognize the device correctly.

7.17 Special program parameters

Beside the 'normal' program arguments the control program also offers a few parameters to affect the program in some special cases.
The relating parameters are listed below and are not stored after the program end. That is you have to give it to the program each time you start it again.

Parameter	Description
--ignore-unsaved-data	Disables the warning about recorded but not saved data. This may useful if you running some tests without a need to store the data always afterwards.

--socket=*portnumber*	Specifies another socket port for the communication with the SwitchEditor. The default ports are in the range 50000...50100, but sometimes other applications have already reserved these. A validate port number starts with 1024, the max. number is 65535. A zero port number disables the socket completely, the use of the SwitchOption isn't possible then.

8

The MultiView design

Already while recording the data can be displayed at different points in time in different formats with different time resolution. We call this concept MultiView, the actors Views or Analysis Tools.

The MSB-RS485 analyzer software uses a multi-process architecture to guarantee a high maximum in stability and scalability. The Recording of data from the via USB connected analyzer and their display and evaluation are done by separated and independent programs and processes which communicate with each other. That has a lot of advantages:

- A recording can be examined at the same time at different segments of the data stream and in different representations with different analysis tools.
- Visualization in real time already while recording.
- The number of views only depends on the computing and system power (scalability).
- Application errors in the displaying programs do not have effect on the recording.

By the capability of the single programs (Views) to communicate with each other a number of new possibilities to make the analysis of EIA-422/485 connections easy are opened.
So different views of the recorded data can be *linked*. What does that mean?
Every display program can be selected as *master*. All other data views automatically follow this master view and synchronize their displays to it. For instance: The graph of the physical data signal (scope view) follows the cursor of the data monitor and vice versa.
The search for a defined level change or a specific data delay fades in the respective data sequence. A click onto the recorded parity error shows the respective signal, a.s.o

8.1 Synchronization

This way of communicating is called synchronization, the handling is identical for all Views.
Each display program may alternately follow the current recording and display the last occurred events (data byte or level change). Or it can lock the current view to compare it with another sector or recording.

Synchronize the displays
individual for each View

If the display program is switched to interlocked operation it reacts on all synchronizing requests which are triggered from other Views and fades in the the respective section of the recorded data in its own display mode. Thereby the program, which is just operated by the user, is automatically seen as the *master*.
With this simple concept any views can be synchronized, completely independent of the running recording.

Symbol	Action	Description
	Follow (autoscroll)	The display follows the recording and always fades in the last recorded data.
	Locked	If locked the content of the View is *frozen*, e.g. to compare it with other views from other parts of the recording.
	Linked	If linked the View is synchronized with the content of the *master* View.

Follow (autoscroll)

If your interest is in the last events of the examined data connection, for instance if you like to see the current data flow or you want to control the current bus direction and/or bus validation you have to activate the *Follow* button in the tool bar.
The analysis window is switched to the autoscroll mode and shifted its window content always so that the last event is visible.
Please note that in this autoscroll mode no synchronization with other analysis windows is performed. An active autoscroll is limited to the respective window and has no effect on other analysis windows.

Locked (fixed)

In case the opened windows shall represent different data sections a synchronizing or following of the display is not wanted. You just want to intendedly watch the different data sections. An update by synchronization would delete the window content. Therefore set the display mode to locked.

Linked

As soon as you activate this button the content of the window follows the cursor movements of the active input window. That means it synchronizes with the analysis window which currently has the input focus and is operated by you (master window).
If more than one analysis window is opened at the same time automatically the window which has the input focus is the master. All cursor movements or shifts are also transferred to all those windows, which are set to the *linked* mode.

8.2 Views (displays)

Views are autonomous programs which link into a current running recording and visualize data in a certain format. The MSB-RS485 analyzer software follows the concept to offer a specially optimized display tool for each kind of examination.

Each view provides functions which represents its kind of data interpretation. Thereby the handling stays easy and clear, multiline toolbars and overload menus are avoided.
You are searching in a data View for data sequences, while you watch out for level changes in the event monitor? Each View provides just the search dialog you would assume to find there.
Simply close data views which you do not need or do not open them. Since they all are independent programs you can place them on your desktop as you like and vary their size and position.
The session management saves all settings. Views are automatically shown with their last adjustments and can be copied with a single click.
The following Views are available:

Virtual Ledtester

The current line level displaying LED tester is a standard tool for checking RS232 communications. We modified its virtual EIA-232 counterpart for the operation on EIA-422/485 connections. In this way a fast check of the bus states (inactive /active), data direction, handshake conditions and digital auxiliary inputs is possible.

DataView - Data Monitor

The data monitor represents the transferred data as a series of data bytes in different formats (ASCII, decimal or hexadecimal). As a special feature the data monitor allows the search for defined pattern by the use of regular expressions, which exceeds the normal search for words or sequences by far. In addition you can search for pauses between sent and received data and in general between any data.
With the help of the integrated script language the displayed data can be computed and colored in any way. Protocols can be visualized, checksums tested in real time and data transformed into other forms.

EventView - Event Monitor

Every line change is an event and is logged. Be it the change of a control line or the change of a single bit of a transferred data byte. The event monitor lists them all and allows a simple navigating between all or certain event types, the measuring of times between events and the search for defined conditions or condition changes. E.g. Changing of the bus state (tri-state) or of a handshake signal during a data sequence.

ProtocolView - Protocol Monitor

The protocol view enables you to display the recorded data according to special rules.
Define your own protocol so that every data sequence is displayed in an own line. Also color any section of the sequence to make them more readable.

SignalView - Signal Monitor

The MSB-RS485 analyzer samples the logical state of all signals with a maximum of 16 MHz. The result can be watched in the signal monitor. Analogous to a digital scope you can move to any section and examine in different resolutions.

By synchronizing to other views you immediately see the basic signal behavior of every data byte and therewith the real world of your EIA-422/485 connection.

Regions

Regions are definable sections of the recording. They can be compared with bookmarks and define time ranges in the recording file. Regions can be named, they also can send out *synchronization requests* to other views.
A click onto the start or end of the region is sufficient to let them be faded in into other windows. In this way it is easy to compare recorded sectors in different representations.

Copy View
to compare its content with another sector

8.3 Copy Views

The *Clone* symbol in the toolbar starts an exact copy of the current analysis window with all its features, settings and position within the recorded data.
By this you can fix a current view while you go on working with the copy (or the original). This makes sense when you want to compare various data regions.

Save settings
and close the window

8.4 Saving the state of the Views

To make the working with views easy the current settings of a view like size and position are saved as default when the window is closed by click onto the ⓘ symbol. The view is restored when you reopen the view. That avoids that you have to enter all settings when you start a new analysis tool.

In case you do not want to save the settings, e.g. because you experimented with the settings and want to close it without 'consequences', simply click onto the close symbol in the window frame instead of the quit symbol in the toolbar.

Discard settings
and close window

The saved defaults stay active even after the end of a session. They are part of the session settings and are also regarded in project files.
Project files contain a complete description of the current session. They are the subject of the following chapter.

9 Session management

A program session contains a variety of opened windows in different views. The session management cares that at program start you will find everything again like you left when closing the program.

The session management takes care for the correct storing of all relevant settings for a session. All recording parameters, window properties (position, size) and content (colors, text size, formats) of the open Views are saved as configuration when closing the window and restored at next start.
The storage of the current program settings are completely transparent. You do not have to trigger this process, but you also can save the complete session including the recorded data as a project. In this case you can proceed with the examination of the data at a later time simply by opening the project file.

9.1 Projects

Projects are used for saving of your current work (analysis) with the MSB-RS485 software, that means the recorded data is also stored. Therefore a project always consists of two files:

1. **Project file:** Project file: Describes the condition and properties of all open Views. Project files do have the extension `*.msbprj`.
2. **Record file:** Record file: Contains the actual data and all information, relevant for the data recording which are: data rate, protocol, defined regions, which event types are recorded and time of their recording. Record files do have the extension `*.msblog`.

Why this splitting into two files?

Stored sessions (projects) corresponds to the user request to configure the program individually for his own needs. These are mostly independent of the recorded data. Perhaps he wants to adapt the placement and display of the views to the screen resolution or use other fonts than the default ones.
On the other hand record files contain information which are independent of the session settings. These information about the protocol, time stamps, regions, used signal names and (de)activated event types. Furthermore recordings should be analyzed by different persons with different ideas about the configuration.

By this splitting some advantages are added:

- The storage of the recording is done independent of the current session.
- A recording can be loaded into an existing session without disturbing it.
- Other users can examine the data with their individual configuration.
- Project files can be purposefully defined for certain analysis and forwarded.

By the clear separation between project and record file you always can examine a recording with your own program settings or you can use another predefined program configuration for the analysis.
Project and record files have their own icon to make the distinction easier. They are linked to the MSB-RS485 software while installation and can be opened by a double click

A project file
stores all session settings and has the extension *.msbprj

A record file
contains the data and also all transmission parameters. It has the extension *.msblog

9.2 Store and reload projects

Storing and reloading of projects are executed from the control program. The seperation into a session and recording file is done automatically like described before.
Likewise a recording file (if existing) is loaded when you open a project.
The same applies when you start the MSB-RS485 software with double-click onto a project file `*.msbprj` from the Windows file explorer. Opening of a project file automatically loads the associated recording file msblog too.
Please note that certain settings like the baudrate are stored as default in the session file as well as in the recording file as mandatory part.

As soon as a recording is loaded by the software this information is fetched from the recording file and (over)written into the session configuration. This applies for the protocol settings (baudrate, parity, stopbit) and for definition of the signal names and activated events. These settings are inseparably linked to the recorded data.

Create a pure project file without data recording

To save a current session as configuration for later examinations you have to save it as project without data or you can delete the record file to get the pure project file.

9.3 Automatic storing of a session

This process is done transparently in the background as soon as you close the current session by closing of the control program. The MSB-RS485 software stores all necessary settings in a configuration file with the name `MSB#####.msbprj` in your home directory whereat `#####` is the serial number of your MSB device. Under Microsoft Windows this is:

```
C:\Documents and settings\User name\MSB######.msbprj
```

Under Linux:

```
/home/User name/MSB#####.msbprj
```

If you did not connect an analyzer the settings are stored in the file `MSB00000.msbprj`.

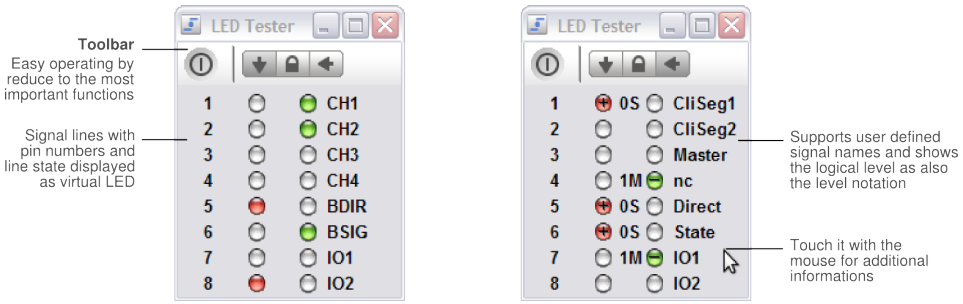
10 The virtual Ledtester

The the current line level indicating LED tine testers are standard tools for checking the levels at the RS232 lines. The for EIA-422/485 connections adapted virtual counterpart allows a fast overview about the bus state, bus data direction and bus activity.

The virtual LED tester is modeled on a customary serial line state tester and shows the state of all differential inputs CH1 to CH4 and of both auxiliary inputs Additionally the bus signal and the bus direction between CH1 and Ch2 (segment analysis) is visualized.

For better clarity the LED tester (or line monitor) has two separate diode columns for every active line state, consisting of red and green LEDs in each case. The red LEDs on the right side signal a positive line level, the green LEDs on the left side a negative level.

In the range from ±0.7V both LEDs are off. This correspondends to the inactive Bus/line condition. The trigger level of the EIA-485 receivers is about ±200mV. By the higher level of the MSB-RS485 analyzer inactive bus levels are still recognized, even if the bus rest level is drawn by pull up resistors to more than ±200mV.



By default the current state is displayed independent o a running recording. This corresponds to the synchronization to the last recorded event. Therefore the scroll button in the toolbar is activated. So the monitor is comparable to a real tester.
You also can use the monitor for watching the status of earlier data in the re-

corded data stream.
Click the 'Sync' symbol in the toolbar. With this the line state monitor is synchronized with the active display window, e.g. a data monitor. Or you freeze the current state by clicking the 'Lock' Button.

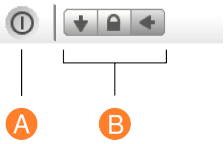
The active levels of a EIA-422/485 connection are alternatively described with logical 0/1 as space/mark or as a physical positive or negative voltage. Most time this is more confusing than helpful. To make it a bit easier the Ledtester fades in additional information about the line conditions.
Simply move the cursor over the Tester to make this information visible.

Level	Description
1M	A logical 1, Mark refers to a negativ voltage level (–0.7V...-7V) on the difference signal input (green LED with a minus Symbol)
0S	A logical 0, space refers to a positive voltage level (+0.7V...+12V) on the difference signal input (red LED with a plus Symbol)

As all other Views the virtual Ledtester also updates the signal names as soon as these are changed in the control program. This is also true when you change the bus wiring.

10.1 The toolbar

The toolbar offers a quick access to the most needed functions.



A	**End:** Saves all settings and closes the window.
B	**Display mode:** According to the mode the ledtester either shows always the current (last recorded) line states or locked or actualizes its content synchronous to the other windows.

11 The Data View

You are searching for certain data sequences? For communication breaks of a certain length? The data monitor shows the data in their real time sequence, alternatively in decimal, hexadecimal or ASCII and additionally contains parity, framing or break information. Regular expressions allow the search for any data pattern and much more...

The data view displays all transmitted and by the MSB-RS485 recorded data bytes in their sequence. Changes in the control lines are fade out, so that only the pure user data is shown.
The data can be displayed separated for each data channel A/B (the assignment input signal / Data channel depends on the connection mode) or together (see signal selection). The latter makes sense if the reaction on sent data shall be inspected.
If you want to examine the data separate for data channel A and B without mixing them simply start two data monitors.
You also can watch different sections of the same data stream. You can open as many windows as you need. The PC-Resources are the only limitation.

11.1 User Interface

The data monitor shows the transmitted data bytes like a hex editor. Default are 8 characters or bytes[1] per line, displayed in hexadecimal notation and in the ASCII representation. Every line starts with the current address or position as the offset from the beginning of the data stream. Non printable bytes e.g. the carriage return sign are displayed as a dot.
Use the arrow keys to move the Cursor while additional information is displayed in the Statusline like the exact time, position and quantity in relation to the complete data stream.
In case of a communication error (framing or parity) the data monitor fades in the error into the associated data byte. This is also done for the break condition, which could be misinterpreted in the data stream as a null byte.

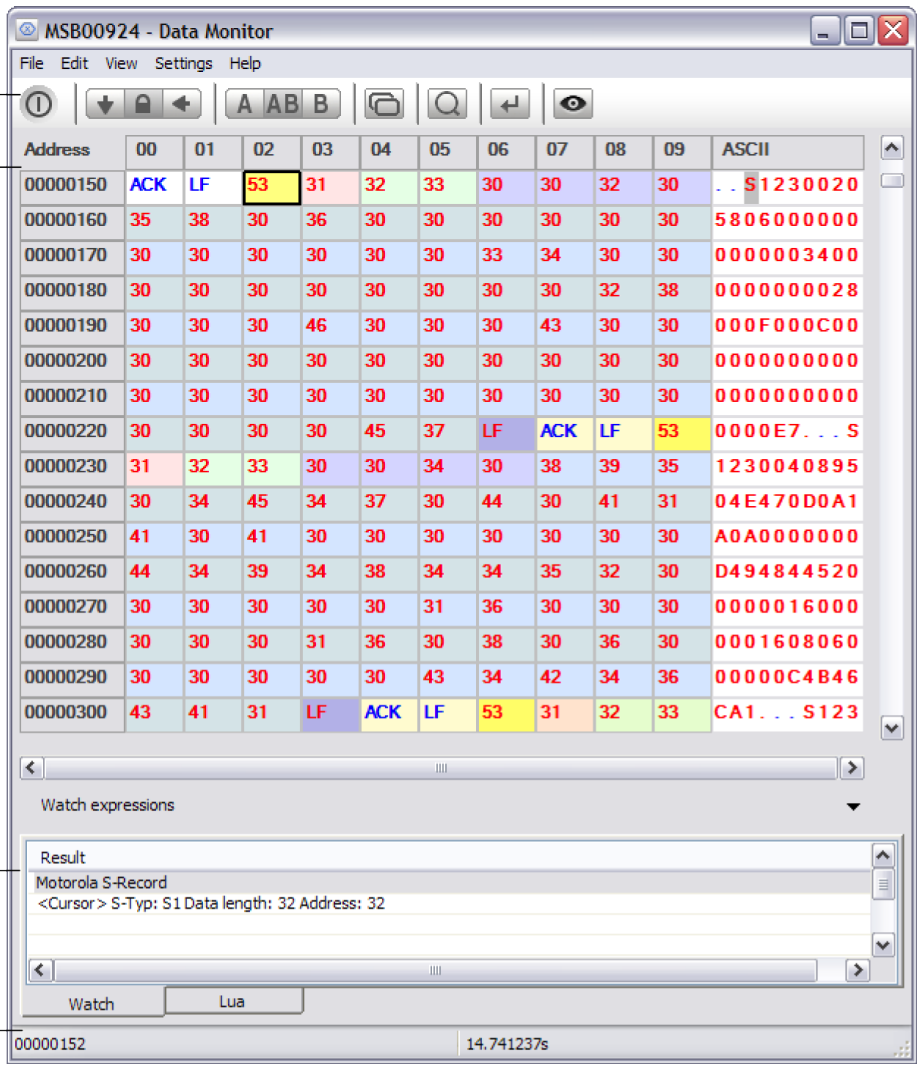
[1] Strictly spoken 9 bit values because the MSB-RS485 analyzer supports transmissions with 9 bit data length.

Toolbar
Simple operation by reduce to the most important functions

Data display
Visualising your data depending on special protocols in various formats Fade in transmission errors and display 9 bit values too

Folding Watch Window
Validate, convert and mark any data sequence with the integrated Lua Script engine

Statusbar
Let you spezify your own information you like to see in the statusbar



With the integrated Lua script interpreter you can calculate the displayed data in any form, convert a sequence of data in another format and output the result in the watch window (see section 11.6.

Integrated Lua

Even more - you can colorize and mark the displayed data in real time controlled by a Lua script. For instance if you like to emphasize a curtain protocol or some special sequences of interest. And if you wish to validate an additional checksum too - no problem. A little Lua script will do the job and marks the according bytes in the display as correct or wrong.

The information in the status line is always related to the current cursor position. In the default settings the left field contains the exact position inside the data stream. Only data bytes are counted, other events like level changes are ignored.
The right field contains the exact time when the data byte has occurred.

You can modify the output of each statusbar field anytime by an own Lua script. For instance, if you like to show the time in another format, or if you are interest in the distance to the previous and/or next byte. The chapter 16.1 will give you an introduction how you can realize this.

Data channel selection

The data monitor optionally displays both data channels (sources) A and B or a single one. According to the bus connection (tapping 2/4-wire system or segemnt analysis) the display of both channels makes more or less sense. You can switch between the several directions anytime just by click the channel selection in the toolbar.

The data channel selection alse defines which data are stored. If you choose both data channels A+B both are stored, otherwise only the data of the selected one. In this way it is possible to save the recorded data or parts of it depending on the data source in one file.

Data channel selection displays both or single data sources

Synchronizing

Each analysis window can synchronize its current view with other windows (see Synchronizing the data view). Is the data monitor the active window, that means the one which gets your inputs, than every move of the cursor sends a sync signal to other opened windows.

This includes the cursor movement as a result of a search or positioning. In this way you can watch the signals in the signal monitor, remotely controlled by the search for certain data sequences.

Leftclick the designated data byte to fade in its representation in other views.

Likewise the data monitor reacts on a synchronization from other views and fades in the respective data section, where the cursor is positioned onto the data byte nearest to the original event.

How the data monitor acts when it receives the sync signal from another active window determine the sync-buttons in the toolbar.

By default the data view is locked, the window does not react on changes. Please note that the data monitor always generates a sync signal, independent of the choosen display lock/unlock function. Windows which shall not react on sync signals have to be locked.

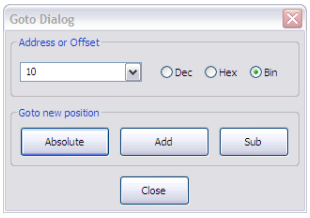
Scroll, Lock or Update data display by other views

Addressing the window content

Besides the navigation by cursor or the left scroll bar the data monitor offers an absolute positioning and shift relative to the current position..

Click the  symbol in the toolbar or open the dialog via View→Goto. Or simply just press Ctrl+G.

Simply enter the absolute address or the wanted offset and click one of the following keys.



Absolute or relative positioning with Ctrl+G

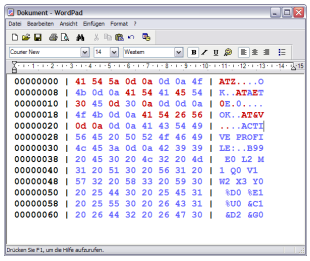
- **Absolut:** Moves the data sector to the entered address, shortkey Alt+A.
- **Plus:** Adds the entered value to the current position and moves the data view towards data end, shortkey Alt+P.
- **Minus:** Subtracts the entered value from the current position and moves the data view towards data start, shortkey Alt+N.

The input can be made in decimal, hexadecimal or binary format. Simply click on the used number format. Like most other dialogs you can leave this dialog open as long you need it.

11.2 Data selection

If you press the left mouse key while the cursor is on one of the displyed data byte a context menu opens. In this menu any section of the recorded data can be selected. You can mark the beginning or the end of the selection.

You can also mark the beginning of a region with Ctrl + left mouse key and the end with Shift + left Mouse key. This corresponds to the file selection in Mircosoft Windows Explorer. The selection is marked with a light blue color.
If you want to select all data, just press Ctrl+A.



Copy and Paste
in a word processor

The data selection can be stored seperately or assigned to a Region with F4. By storing special data sequences you can examine these data for transmission errors or compare to other data sequences.
With the export or copy and paste mechanism you are allowed to evaluate any desired section of data in other applications.

Copy and Paste

Copy and paste copies the selected range as text into the clipboard and paste it into another program. If the target application supports RTF like the most word processing software (for example WordPad®, Microsoft Word®or OpenOffice Writer®), the copied data will be inserted with the origin color information, i.e. the data of port A are shown as red, data of port B as blue[2].
Pure text editors (like Notepad) doesn't provide any text formating. Therefore the information of the data direction has to be visualized in a different way. Data bytes received via the first Data channel (A) precedes a dot, data from the second data channel (B) a colon.
The text display is generally in the hexadecimal format to avoid problems with different character fonts.

```
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.
```

Save data selection

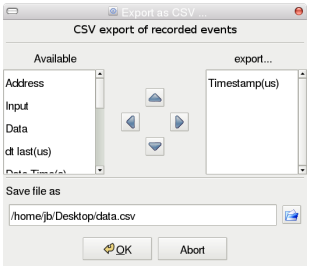
The data monitor allows to save any selected range (see Data selection) as binary data into a file. This file herewith contains an accurate series of the marked data. You will appreciate this when you want to compare the recorded data sequences to others, available as data files.

[2]Coloured Copy and Paste is supported only in the analyser software for Microsoft Windows.

For example if you know the result of the sent data from a bus participant or the original data and you simply want to check if these data have been correctly transmitted. Simply select the wanted range or all data with Ctrl+A and click the menu item File→Save as....
If you selected both data channels for display (A+B) both are stored as well. For a comparison it makes sense to choose just one channel.

Export a data selection

To analyze a section or all recorded data with spread sheet analysis you can export these as a CSV (Comma Separated Values) File. Spreadsheet programs offer extensive statistical tools to evaluate the data. For example the frequency distribution of single data or minimum and maximum times between the bytes. The export capabilities concern the data only. If you are interested in an analysis of other events read chapter Export selection in the event monitor.
Select the wanted range and click on the entry export as CVS in the file menu. In the opening export dialog you can select from the list of the available values any value by clicking on it and moving it with the right arrow to the list of the export values. Repeat this for all interesting values.
To change the sequence of the export values click on the value to shift and move it up or down with the up or down arrow.
Likewise you can remove a value from the export list with the left arrow.
Then enter a name for the export file and click on 'OK' to start the export.
For exporting the current view of the data monitor is regarded. The data is exported as hexadecimal values with prefix 0x, or as decimal value or as ASCII character included in apostophes. The same applies for the addresses.
(The address is the position of the data byte in th data stream). An example for the hexadecimal address and data format:



Data export
as comma separate value list

```
"Timestamp(us)","Address","Input","Data"
3547,0x000050,A,0x20
3547,0x000051,B,0x20
3634,0x000052,A,0x21
3634,0x000053,B,0x21
3720,0x000054,A,0x22
3720,0x000055,B,0x22
...
```

The same selection showing a decimal displaying address and the data as ASCII.

```
"Timestamp(us)","Address","Input","Data"
3547,00000080,A,' '
3547,00000081,B,' '
3634,00000082,A,'!'
3634,00000083,B,'!'
3720,00000084,A,'"'
3720,00000085,B,'"'
...
```

Note! The timestamp resolution is in micro seconds (us). Because we have record this samples with a loop back jack, always two data events on data channel A and B have the same timestamp.

11.3 Data displaying

The data display can be adapted to your own requirements. Just open the setup dialog in the menu `Settings→Configure Data Monitor...`
Every view offers only those setup possibilities which are relevant for this view. In case of the data monitor it is:

- **Display:** Number and form of columns and data.
- **Colours:** Coloring rules for representation and marking of certain data.
- **Font:** Font type and size.

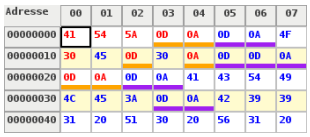
All settings can be tested with the apply button before they are finally accepted with the OK button.

Columns and data format

In this part of the settings dialog `Settings→Configure Data Monitor...` you can individually select the number of columns as well as the kind of display (hexa, decimal, ASCII). The number of lines are changed by extending or reducing the display window.
In addition you can fade in the generally defined names for the first 32 characters of the ASCII character set (control characters), e.g. to display 'LF' for linefeed instead of Hex 0A.
Not printable characters can be displayed alternatively as a dot or as the original character according to the chosen font type.

Coloring data

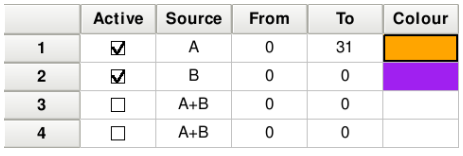
The data monitor allows to color any data depending on the data source. That is an important feature if you want to hi-light certain data bytes or sequences. For example the EOS character like carriage Return and/or Line Feed. Or characters wit a set 8th or 9th bit as often used in bus protocols to separate data from address commands.



Adresse	00	01	02	03	04	05	06	07
00000000	41	54	5A	0D	0A	0D	0A	4F
00000010	30	45	0D	30	0A	0D	0D	0A
00000020	0D	0A	0D	0A	41	43	54	49
00000030	4C	45	3A	0D	0A	42	39	39
00000040	31	20	51	30	20	56	31	20

Coloring data
with colour rules...

To activate this feature you can define any number of *coloring rules* which are applied on the display of the transmitted data.



	Active	Source	From	To	Colour
1	☑	A	0	31	
2	☑	B	0	0	
3	☐	A+B	0	0	
4	☐	A+B	0	0	

Each rule contains the data source or data channel (A or B), a range for the data value and the color to dye these data bytes. You can switch every rule on or off individually by enabling or disabling it.
The input of the data values from/to is done in decimal where the range is 0 to 511. Values above 255 makes sense only if you analyze transmissions 511. with 9 data bits.
The rules are processed in the sequence from 1 to n (or from top to bottom). Rules can overlap. In this case the last rule is regarded. With this it is possible to overwrite rules in parts to recolor single bytes of a rule defined before.
The presetting has four rules. By clicking onto the [+] button you can add new rules at any time or remove them by using the [−] button.
New rules are attached either at the end or are directly inserted over the selected rule. To remove a rule you first have to select it by clicking onto it.
All entered rules are automatically stored and are at the same time available for all later opened data monitors.

Color schemes are intended for simple applications. If you want to mark the data with complex rules use the integrated Lua script editor in the watch window, see chapter 11.6

Change the font

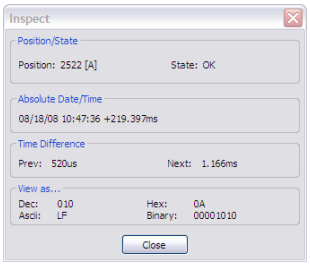
Besides the number of columns and the representation of the data bytes you also can alter the font type, e.g. to use a font with letters of equal width instead of the default proportional font or to adapt the letter type and height.
Click on `Settings→configure data monitor` to open the settings dialog.
The chosen font type is automatically stored.

Choose another font
in the settings dialog

11.4 The data inspector

Watching the transfered data is one thing. To find out the reasons for communication problems sometimes it is necessary to analyse the exact timing for the transfered data. What is the time difference between two bytes? Or how long does it take to receive the answer for a sent data string? Click on the 'Inspect' symbol in the toolbar or press Ctrl+I to open the data inspector. The data inspector offers some informations related to the current byte:

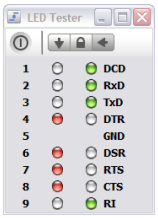
- **Position:** The absolute position of the byte and it's source (Port A or Port B).
- **State:** Error state of the byte, i.e. Parity, Framing or Break.
- **Absolute Date/Time:** Shows the absolute date and time of the received byte in your locale time format.
- **Time difference:** Displays the distance between the previous and next byte.
- **View as...:** Converts the data byte value in different formats.



The data inspector
shows time distances and converts data byte values

Display the line states

To watch the current line state in parallel to the data byte simply open the 'virtual LedTester' in the control program and switch it to synchronizing operation.



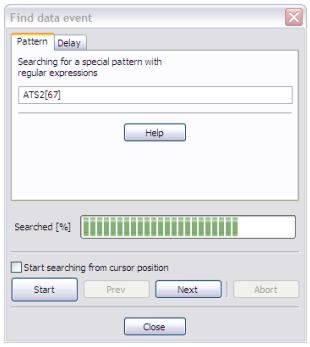
The virtual Ledtester
shows the current line levels

11.5 Searching the record

The data monitor contains some functions which are optimal adapted to the search for data and data sequences. So different searches are possible. The search for a certain series of data bytes, for too short or too long times between request and answer, or simply for transmission errors like parity, framing or break. Since every search starts from the beginning or the current cursor position all search functions can be combined in any way.

Pattern search

One of the outstanding attributes of the data monitor is the search function for special data sequences. The search input is not limited to simple comparisons of data strings. In fact the Search dialog allows the input of so called regular expressions.
Regular expressions are extended by the wild card characters '*' and '?', known from the MSDOS DIR Command. So the command DIR *.HTM lists all files



Find any string
with regular expressions

which have the extention HTM.
DIR FILE?.TXT lists the files (when available) FILE1.TXT, FILE2.TXT etc. Similar mechanisms for searching special data sequences are offered by the Search dialog of the data monitor. It is opened by the search symbol in the toolbar or simply with Ctrl + F.
Generally search starts at cursor position! By default the search is starts with the begin of the data recording. You also can start the pattern search beginning from any other time stamp within the data stream. For this purpose position the cursor of the data monitor at the desired start position and activate the button 'Search from cursor position'.
To find a special sequence you first have to describe this sequence in the input box. It can be a simple string, e.g. LOGIN in a modem connection. Click on the Search button to start the search in the recorded data stream.
The search is always restricted to the displayed data channel. If you have selected channel A only the bytes assigned to this channels are searched. The same applies for channel B. If both channels are displayed all recorded data are regarded. Often the searched data can not be described by a simple data series. For example the recorded data stream can contain the word 'LOGIN' in the following combinations: LOGIN, Login or login. The latter ones can be described like in MSDOS with ?ogin for search. To find all three variants you have to describe the search pattern as a series of the characters L,O,G,I,N, where each character can be a capital letter or not.
For the conversion a regular expression is used. The expressions are listed in the Table below.

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

Every character is described by a Set which exactly corresponds to the searched letters.
Imagine you inspect a data connection where from time to time the CR of a CR-LF(carriage return line feed) sequence is missing. That means you search for a single LF WITHOUT a CR directly before. The appropriate regular expression is:

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

and means: all characters except Hex 0D(means CR), followed by a Hex 0A (LF).
A regular expression is a series of any charcter, where certain characters can have a special function. They are listed in the following Table . If you want to use one of the special characters as a normal one, e.g. if you search for Password? and '?' is NOT any charcter but really the question mark, you have to quote it. This is done by a preceding \ char. For instance

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

With the '*' char in a search pattern any data sequence is marked. That makes sense only if this character is framed by other search patterns, otherwise everything will be found.
The following expression finds all names found between 'LOGIN' and 'PASSWORT' but not single 'LOGIN' Sequences without following 'PASSWORT' Sequence:

```
LOGIN*PASSWORT
```

Search for 9-Bit sequences

As a special case the search mechanism in the DataView also supports 9-bit values. A 9-bit value cannot input as a normal character. The DataView therefore extends the definition of any hex value as described above by a 'special' 3-digit hex input. Such a value (or 9-bit character) is initiate with a upper \X followed by three hex digits.
The following example looks for a sequence starting with a Hex 10B or Hex 133 followed by Hex 33:

[\X10B\X133]\X033

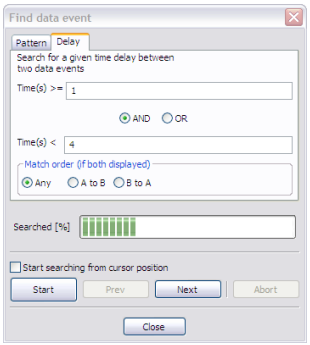
The following tablelists the available expressions.

Expressic	Meaning
?	any character
*	any character string
[abc]	a char out of the set *abc*
[!abc]	a char not member of the set *abc* angehört
\xHL	a char in hexadecimal notation, H is the upper half byte, L the lower half byte
\X1HL	Same like above, but supporting 9-bit values. The first digit must always 0 or 1. Valid range is from 000 to 1FF. Please note, that a upper 'X' always requires a 9-bit hex digit!
\?	the character ?
*	the character *
\[	the character [
\]	the character]
\d	any decimal character 0...9
\n	the linefeed control character (Hex. 0x0A)
\s	any whitespace character (blank, linefeed, carridge return, horizontal tab)
\\	the character \

A misentry will be showed as a selected marked input, so you just can retype a corrected version of your matching rule.

Search for time distances

Beside the pattern search facilities the data view also supports the search for defined time distances between two data events. Click onto the search symbol in the toolbar or press Ctrl+F and select the slider Delay.
The time specification is always done in seconds, e.g. 0.0015 for 15 milliseconds. The smallest time unit correspondends to the resolution of the analyzer and is 0.000001 or 1μs. Time distances can be defined as limits for over or under stepping or as a range. The button with the link symbol decides if both

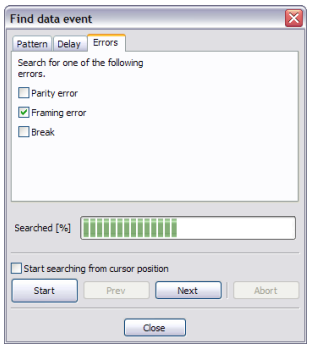


Find time distances and transmission intermissions

times have to be valid for the search result (AND-relation) or only one of them, which is the default.
Please take a look to the following table:

Time(s) >=	Logic	Time(s) <	Result
1.000000s	OR	0	Finds all distances which are longer than 1s OR shorter than 0s. Negative times are not valid, so that the search is for times longer than the entered 1s.
1.000000s	AND	2.000000s	Finds all times which are longer than 1s and shorter than 2s, i.e. Times between 1 and 2 seconds.
10000000s	OR	0.001s	Finds all times which are greater than 100000s or smaller than 1ms. Since such long times will never happen only times smaller than 1 ms will be found.

Besides the time specifications also the sequence plays a role. That means whether the timely distance between two data bytes of one source, e.g. data channel A, is measured. Or a data byte from channel A, followed by a data byte from channel B (a possible answer). Depending on the wiring you can intentionally search for answering times of a certain bus device and check the answering behavior.
The sequence which shall be obeyed for the search process can be set explicitely. Default is Any, i.e. the sequence is irrelevant.
Please note, that the sequence can be set only if both data sources A and B are activated in the data monitor, otherwise it is disabled.



Find transmission errors
just with a click

Search for transmission errors

The search for error conditions refers to errors in the data transmission. These are framing and parity errors. Breaks are usually no errors, but because they must not be mixed with the nul byte and are sometimes used for initializing or resetting of communication partners they are also integrated into the search mechanism.
The search for errors is easy. Mark one or more error conditions and start the search with a click onto the start button.

11.6 The Watch window

The Watch Window serves as the input window for Lua scripts and as the display of their program outputs. If you open up the Watch window for the first time you see an empty list of eight lines or entries. Each entry a Lua script is assigned. As long as you do not enter a script or this script does not produce any output the entry stays empty.
You can display any information in these lines. Numbers, text or a combination of both but no line breaks. Every output is limited to one single but unlimited

line.
Double click on any entry you want to write a script for or whose script you want to change. Or click on the Lua tab. Both opens the script editor of the selected entry.

The script editor

The integrated script editor offers all you await from a user-friendly editor! Syntax highlighting unlimited Undo/Redo, Copy and Paste and import and storing of scripts for exchange with other users.
The entries for storing or loading of script are activated in the file menu as soon as you open the script editor. The file operations always refer to the active editor and therefore to the selected entry in the watch window.
The MSB-RS485 software comes with a number of examples which you can find in the examples/Lua directory of the installation directory.
You can load one of these examples into the editor (see section 11.6) or simply enter the following two lines to get you in the mood (presumed some recorded data):

```
1  dv.watch( "Cursor at ", dv.cursor() )
2  dv.mark( dv.cursor(), 1, #ff8080 )
```

This small script displays the current cursor position in relationship to the current segment and colors the corresponding data cell in a bright red.
To run the script click on the 'Run' button. Alternatively you can use the key combination Alt+R. The script output is shown as an entry in the Watch List and also in the text line of the script editor because you do not see the watch list entry when the editor is opened.

Run script

In case of a wrong input or syntax error an error message is displayed at this place. In our example the current cursor position- if you input the text correctly. With the start of the script it is transferred into the internal byte code of the Lua interpreter. This code is run each time the cursor is moved or the window content is changed. The same happens when you close the editor and return to the watch list. Move the cursor or click onto another data cell to see effects of the script.
A storage of the editor content is not necessary. The data monitor automatically saves its current status and restores it with the next start.
Additionally you can save the editor content as a text file to reload it into the editor at any time.

Example scripts

In the folder examples/DataView of the installation directory you will find some examples to show the possibilities of Lua. Start the analyzer software by double clicking (Windows) or open the wanted project file `*.msbprj` from the control program `msb_serv`. Alternatively you can directly load the project from the control program.
Every example project contains a recording and loads the data monitor with the corresponding Lua script and settings so that you can immediately start.

Lua examples
in the installations folder

- `9bit.msbprj`
  Analysis of a 9 Bit data protocol including check sum test. Lua coded LRC (Logitudinal Redundancy Check) function tests for the correctness of the check sum when the cursor is positioned on the start of the protocol sequence.

At the same time the sequence is visualized with different colors for Start (Address) byte, length, data and checksum byte.

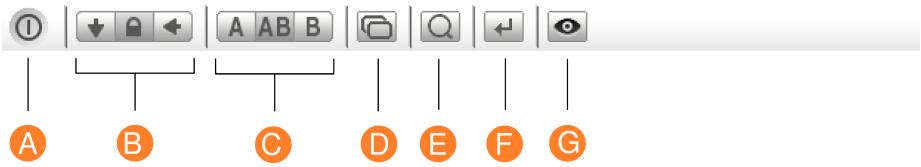
- `errors.msbprj`
  Coloring of the data bytes for frame-, parity errors or breaks. Shows the information output in the status line, the iteration over the data shown in the data monitor and the indication depending on the error status.
- `modbus-rtu.msbprj`
  Shows the 2-wire segment analysis of a RS485 Modbus connection with one Master and two slaves. The single sequences are colored differently for address, function code, data and check sum. By clicking onto an address byte automatically the CRC16 check sum of the sequence is computed and displayed in the status line for comparison with the recorded check sum.
- `srecord.msbprj`
  Coloring of all protocol sequences in the data area. Using the recorded data of a Motorola S-Record transfers the data display is shown in terms of color where two following bytes in the BCD representation are converted into the underlying 8 bit value.

Limitations

You can run any operation in Lua, write complex functions and perform extensive evaluations. But the data monitor allows each Lua script only a certain number of computing operations (recursions) or time period for the execution. As soon as your entered script exceeds this limit you get an error message. And that is of a good reason.
If you have programmed an endless loop for whatever reason the data monitor will kindly notify you instead of wordlessly stop further co-operation.

11.7 The toolbar

The tool bar is used for a quick access to the most needed functions. Some are identical to other views, some are specific for the data view..

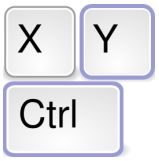


A	**End:** Saves all settings and closes the window.
B	**Display mode:** According to the mode the window either shows always the current (last recorded) event or locked or actualizes its content synchronous to the other windows.
C	**Data direction:** The protocol monitor can display both data directions (port A and B) combined or separately to display them in different windows.
C	**New View:** Opens a new window with the same sector and settings.

E	**Search dialog:** Opens the dialog for pattern search and transmission interceptions.
F	**Goto...:** Opens the Goto dialog to select the visible section by a absolut address or offset.
G	**Data inspector:** Starts the data inspector.

11.8 Short commands

Aktion	Kurzbefehl
Runs the current script in the script editor	Alt+R
Online Help for the data monitor	F1
Save selection as region	F4
Start selection	Ctrl+Left mouse key
End of selection	Shift+Left mouse key
Select all	Ctrl+A
Clear selection	Shift+Ctrl+A
Copy selection into clipboard	Ctrl+C
Export selection	Ctrl+E
Open search dialog	Ctrl+F
Open goto dialog	Ctrl+G
Show data inspector	Ctrl+I
Open View in a new window	Shift+Ctrl+N
Save settings and close data view	Ctrl+Q
Save selection as binary file	Ctrl+S



Key commands
of the most important functions

12 The Event View

When did an event occur? Did a certain level change happen while data was transferred? Or was an error condition (break, parity, framing) recognized? How the status of the bus lines was at a specific time.
The event monitor lists all occurred events, searches for event sequences or level conditions and exports events as CSV file.

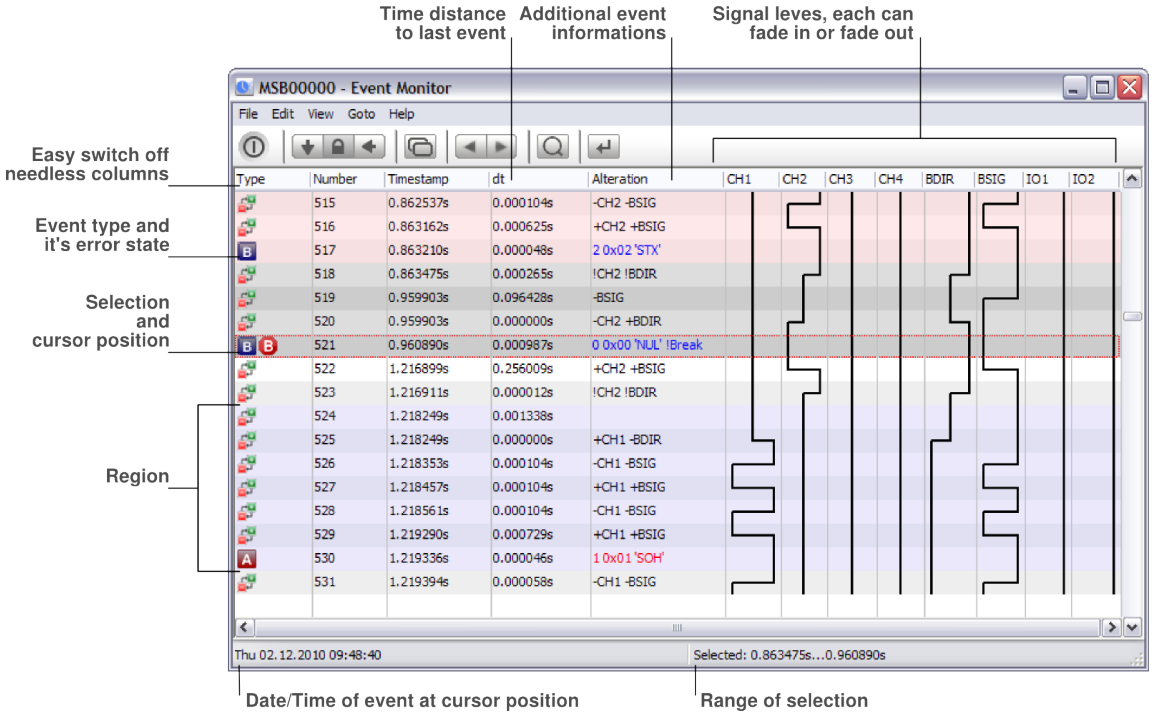
Contrary to the data monitor the event monitor displays all occurred events (data and level changes) with their time relationship. While the data monitor offers a lot of mechanisms to investigate data streams and to represent the data view of the recording, the event monitor is optimized for the display and search of level changes.
That concerns to changes in the level of the control signals as well as asynchronous events like framing or parity errors or breaks.
Each event gets equipped with a time stamp which represents the exact time of its occurrence with a resolution of 1μs. The time distance between has no influence on the display. So signal conditions and changes before and after the recording are easy to identify.

The search for certain data sequences is the task of the data monitor. As soon as a search contains a level condition, a change or an error condition the even monitor is the right tool.

With the LevelFinder not only any static level can be found but also sequence of events and level changes. The single search parameters can be combined optionally with AND or OR and can additionally be combined with a time duration to search for events within a defined time frame or to exclude them.
In the chapter event search the search options are described in detail.

12.1 User Interface

The window of the event monitor at any time offers a quick overview over the level conditions. From here you start the search for event sequences, the export of any section or compare different sector of the recorded data stream.



Each line is one event

The event monitor displays the recorder events in list form where every line represents the current event and its changes compared to the preceding line status. The list display can be freely configured. Except for the first entry (the type of event) you can fade each column out by drawing its width to zero with the mouse. (In the view menu you can reactivate the faded out columns).
The description of the columns automatically adapt to the defined names. These names can be set globally in the control program.

Disable columns
Simply draw the column width to zero.

All event types at a glance

The event monitor distinguishes between the following event types:

Symbol	Event type	Description
A	Data byte	Data byte received at data channel A.
B	Data byte	Data byte received at data channel B.
	Level change	Any change of the level of any signal, including level changes of the data lines.
F	Framing	Data byte received with a framing error.
P	Parity	Data byte received with a parity error.
B	Break	Break detected.

The indented error symbols do never occur singular, but always together with (followed by) a data byte event because it signals an error in the data transfer.

Changes in the levels of a data line (TxD or RxD) likewise trigger level events, followed by a data event as soon as the data bits are completely received.

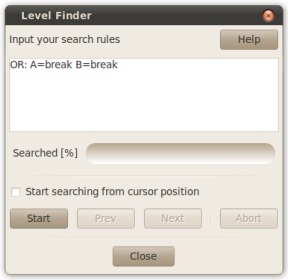
Level changes of the data lines
If you do not see level changes of the difference signal inputs CH1...CH4 you have to activate them in the control program. If you are not interested in these changes deactivate them to save computing power and disk space since these events come very often.

12.2 Navigation through the event list

Go to event
absolute, stepwise or time-dependend

The event monitor offers beside the usual scrolling possibilities by mouse, mouse wheel, scroll bar or arrow and page up/down keys also the directed jump to the next or last event of the same type.
Click onto the desired event and than click the ctrl key together with the down arrow resp. up arrow. In this way you easily navigate from break to break or from data byte to data byte.
For longer records you can purposefully fade in ranges from a determined event or time stamp. The latter one awaits the input of a time offset from the start of the recording.
The specification of an event number allows to jump to a determined event or to move in determined steps from event to event. For example all 1000 events forth and back.

12.3 Event search with the LevelFinder



LevelFinder
searches for level changes, errors and time relationships between events

The detection of definite event sequences is one of the unique features of the event monitor. In contrast to the data monitor the search is not restricted to certain data sequences (for which the data monitor is the best solution) but intentionally adapted to changes in the physical level of the single lines. What does that mean?
You can set up the event monitor to search for the level change and/or the condition of any line. You can combine it with the occurrence of a certain data byte or a number of set bits within a data byte or an asynchronous event like break, framing or parity error. Click onto the magnifying glass symbol in the toolbar or press Ctrl+F to open the search dialog.

Enter a search pattern

The search pattern may be complicated. The integrated level finder accepts the search input in form of logical expressions where every expression can exist of one or more conditions which can be combined with AND or OR.

Expression: condition1 condition2 ... conditionN

For example:

AND: CH1=high BDIR=high

The formulation of the single conditions correspondents to the intuitive question for searching of defined conditions. In the preceding example: Search for the position in the recording where at the same time CH1 is `high` AND the bus

direction signal BDIR is `high` too. [1].
Each condition consists of a target (which the condition shall be applied to) and a description of this condition. So:

Target=Condition

Target is either a single signal line, described by its (also user defined) name or a data (channel) source A or B.
Condition defines the status which the target has to have for the search. In case of one signal line this could be one of the three possible level conditions on, off or invalid (alternative names are mark, space, high, low). If the target is a data source the possible conditions are an exact data value, a bit pattern or an error.

Correct definiton of a condition

Conditions must not contain blanks. Please note, that the names 'A' and 'B' are deserved and must not be used for signal names.

Formulate a level condition

Level conditions can be defined for each of the four difference signal inputs CH1...CH4, both of the additional auxiliary inputs IO1, IO2 and the internal by the analyzer generated bus signals BDIR and BSIG. Not defined lines are simply ignored and not regarded for the search.

Input	Description
1, on, high, mark, -V	Signal level is logical one which corresponds to a physical level of –0.7V...–7V (measured at a difference signal input). All listed descritions are equal. CH1=on is the same as CH1=1 or CH1=high.
0, off, low, space, +V	Signal level is a logical Zero which corresponds to a physical level of +0.7V...+12V (measured at a difference signal input). All listed descritions are equal. CH1=off is the same as CH1=0 or CH1=low.
none, invalid, 0V	Signal level is invalid. That corresponds to a physical level of -0.7V...+0.7V (according to the EIA-422/485 definition the trigger level of the receivers is about $\pm$200mV. By the higher level of the MSB-RS485 analyzer rest levels which are set by pull-up and pull-down resistors are clearly detected as rest levels. The definition of an invalid level is as follows: CH1=none, CH1=invalid, CH1=inactive or CH1=0V.

Formulate a data error

Data errors occur only in connection with a data (channel) source. Therefore the target has to be A or B.

[1] This sample describes a bus conflict. A |high| level of the BDIR signal means an active transmission at CH2 and a bus participant at CH1 must not send at the same time

Input	Description
break, frame, parity	The data received by the data channel A or B shows a break or a framing or parity error. For example a parity error can be found by A=parity or B=parity.

Formulate a data value

Each of the data channel A and B (the data bytes received at the according sources) can be checked for equality or for set bits. The latter one is meaningful when certain bit combinations may are not allowed in the bytes or have a special meaning, defined by the used protocol.
Any data are described by with the * symbol.
The level finder offers four types of entry possibilities:

Input	Description
A=*, B=*	Every data event (A or B) delivers a hit, the data value is not regarded. Makes sense if you search for any data event.
A='x', B='x'	Checks the target (A or B) for equality with the character, enclosed by the apostrophes. The search for a question mark, received at port A is written as : A='?'.
A=$xx, B=$xx	Checks the target (A or B) for equality with the hexadecimal value. A search for a question mark, received at port A is written as: A=$3f or A=$3F.
A=~xxxxxxxx, B=~xxxxxxxx	Checks the target (A or B) for the set bits in xxxxxxxx. For this check the bit pattern is logical AND combined with the data and then checked for equality. To find a data byte at port B with a set 7th bit enter: B=~10000000.

Search input and search

Before you start open in the control program the sample project `levelfinder495.msbprj` in the `examples\EventView` folder. It is about a 2-wire segment analysis with the additional recording of a digital output of a Modbus device with the help of the second digital IO terminal of the analyzer. The recording contains a number of data errors and some combinations of level changes which we will search for in the following.

Example project
in the folder
examples/EventView

Search for a break in data channel A

- Open LevelFinder dialog with Ctrl+F
- Click onto the text field and enter ≫`AND: A=break`≪
- Click the start button
- Click the button 'More' to search for the next break
- Go back to the last hit by click on 'Back'

The visible segment of the monitor changes its position with every hit and displays the found event as a black line.

Search for a break in data channel A or B

- Click onto the text field and enter ≫`OR: A=break B=break`≪

Search Break at Port B with DTR high

- Click text field and enter ≫`AND: B=break DTR=high`≪ eingeben

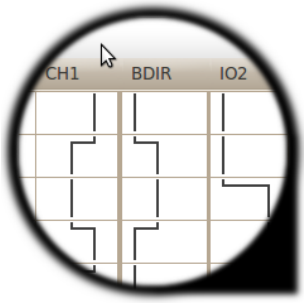
That was easy. We make it a bit more complicated and search for:

Search for an inactive CH2 and BSIG as well as BDIR, IO1, IO2 all high

- Now enter into the text field:
  » `AND: CH2=none BSIG=none BDIR=high IO1=high IO2=high` «

- Click the button 'more' to find the next hits.

You can combine all level conditions, data and data errors combine in any way. It is not possible to combine different logical operations within one search expression, that means mixing of AND and OR expressions. But we will see, that AND and OR are allowed in succeeding expressions. Sequences with different search expressions are used for searching of signal changes. For instance the search for a change in the bus direction BDIR with an active IO2 line at the same time.



Looking for a level change?
No problem with the LevelFinder!

Search for signal changes

Changes are definedby two or more sequenced search expressions which describe the line state before and after the signal change. In this respect we expand the search inputs for the possibility to enter more than one expression in different lines. It is possible to vary the logical operators.
Watch the picture at the side. The interest is on the lines CH1, BDIR and IO2. We want to find the signal change displayed in the ideal zoomed display.
All together there are three changes that have to happen one after another:

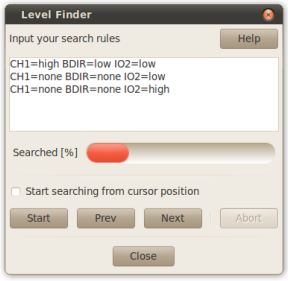
1. CH1=high BDIR=low IO2=low
2. CH1=none BDIR=none IO2=low
3. CH1=none BDIR=none IO2=high

In each state all three signal levels have to be fulfilled, that means that for every single search expression the AND condition has to be true.
Enter each expression in an own line in the text field of the level finder Since the AND operation is the default you can omit it and write the lines exactly as noted above. (Instead of high or low you also can write 1 or 0).

Wrong input, what happens

If you made a syntactical error, e.g. a wrong name or an invalid character, the level finder answers with a yellow text field as soon as you start the search.

You can write any search combinations, for instance the search for a certain data byte together with an active IO2 line, or any data byte from data channel A, followed by a change of the bus direction signal BDIR.
The number of search expressions is unlimited but you have to keep in mind that every expression costs additional computing time and slows down the search process.
The search process runs in parallel to the application and can be aborted at any time by clicking the abort button. Even during a longer search you can operate the event monitor in a normal way and speed.
The LevelFinder saves the current search expression automatically. That also is done when you close the monitor or the complete session.



The LevelFinder
finds data bytes, errors and level changes incl. time measurement.

Start a search beginning with a defined position

Click onto the line where the search shall start and activate *Start search from the cursor position*.

Searching with time specification

As a special feature the LevelFinder offers an integrated stop watch which can be started in every search expression and can be read out in the following expressions. Thus it is possible to search for level changes which exist a certain time only. For instance an active bus direction signal (BDIR) with a duration in the range of 0.1 to 0.3 seconds.

Level duration?
Search with the stop watch

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

Please note that all conditions within an expression are combined per default with the AND operator. line 2 defines the change of the bus direction from an inactive level `none` (line 1) to `high` and at the same time the watch is started. Strictly spoken the watch is reset and loaded with the time of the occurred event, the change of the BDIR signal to high.
In the third line the change of the BDIR level back to an inactive state is AND-combined with a duration greater than 0.1 sec and smaller than 0.1 sec. Positive signal edges which occur later than after 0.3 sec. are ignored.

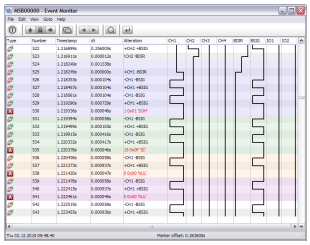
Indication of the hits in the signal monitor

The sample contains exactly two positions which fulfill the conditions above, nicely to be seen in the signal display. Open the signal monitor and set it to 'synchronization' with other views. With every search result the marker jumps to the corresponding signal position.

12.4 Mark a selection

Certain functions of the event monitor like the saving of events as an independent record file or the export in CSV format for later evaluation or as a region always refer to a before defined selection.
The selection of any event sequences is done like in other programs. Left click

in the list representation onto the line, which shall be the first event of the selection. Then scroll by mouse wheel or scroll bar through the recording until the desired last event of the selection. This one click with the left mouse key while pressing the Shift key.
To jump to the last event of your selection you also can use the level finder or the Go-To dialog. It is only important that you click the last event together with the Shift key.
Please note that you can select only ranges and not single arbitrary events.
With `File→save as...` you can save the selection as an own record file `*.msblog` for later evaluation with the analyzer tools. In this way the interesting parts of a recording can be extracted and the necessary storage capacity can be reduced.



Colored regions
highlight special sections

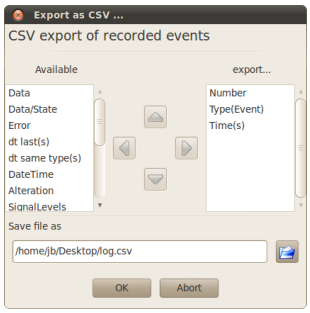
Save a selection as a region

A region serves for marking of certain sections of the recorded data which are of special interest. Contrary to a selection regions are valid for all analysis tools. As soon as a region is added it is visible for all analysis windows.
Regions are displayed in different colored ranges and exist independently from the event monitor until they are explicitly deleted.
To save a selection as region press the F4 key or click onto `Edit→copy to region`. A maximum of eight regions are available. Under `View→show region dialog` the available regions can be fade in or out or removed.
Regions are part of the record and are stored together with the data in the record file `*.msblog`.

Data export as CSV
for external evoluation

Export a selection as CSV file

The even monitor allows to save any selection of recorded events as a Comma Separated List (Values) (CSV). You will use this when you want to import these values into a spread sheet program like Microsoft Excel, Gnumeric or Open Office.
At this time you will probably ask why you should want to import the recorded events into a spread sheet program.
Assume you want to sort the recorded events for the longest pauses between two sent data bytes. Or you want to create a statistic of the events or data. Requirements of this type are the domain of spread sheet calculation programs. The event monitor offers you the possibility to benefit from them.
At first mark the selection of the events to be exported. With Ctrl+A you can select all recorded events at one time.
Click in the menu `File→export as CVS`.



An export dialog opens where you can select a value from the list of available values by clicking and moving with the right arrow button into the list of exported values. Repeat this for all desired values.
To change the sequence of values click onto the value to be moved and shift it in the desired direction with the up or down arrow.

Likewise you can remove a marked entry back to the selection list with the left arrow. Finally you have to enter a name for the export file and click the OK button to start the export.

The list of available events contains the following values:

Value	Description
Number	Event number, starting with 0.
Type(Event)	The following types are defined: A=Data at port A, B=Data at port B, L=Line level change.
Time(s)	Time stamp of the event as offset to the start of the recording in seconds with microsecond precision.
Data	The data (9bit) as an decimal value (0...511).
Data/State	contains either the data up to 9 bit (event type A/B) or the tri-state status of all lines, see line ⇒1.
Error	in case of a transmission fault it contains the kind of error like B (Break), F (Frame) or P (Parity).
dt same type	Time difference to the last event of the same type in seconds with microsecond precision like 0.251518.
dt last	Time difference to the last occurred event, i.e. the last recorded change in the line. The result is in seconds like 0.000217.
Date/Time(s)	Absolute date and time with microseconds of the event like 2014-08-20 09:49:31+148762s.
Alteration	Shows only the changes since the last event as displayed in the alteration column.
SignalLevels	Shows the state/alteration of all lines as displayed in the signal line columns. The graphical display is changed to a respective text string ⇒2.
*CH1,*CH2,*CH3,...	Exports the state of the given signal as a number. The leading '*' differs the signal name from any other field. `-1` represents -12V or mark or 'logical 1' `0` is an invalid level resp. an inactive line state `+1` represents +12V or space or 'logical 0' ⇒3.

[1] Data status

The content of this field is depending on the data type. If it is a transferred data byte it contains the data value in the lower 9 bits, the upper 7 bits are 0.

Bit	Unused							Data Byte							Bit
15							8	7							0

In case of a level change the upper 8 bits contain the logical state of the lines. The lower 8 bits contain the valid states. If it is '0' the line is in an invalid condition, that is a level of -0.7V to +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] Text symbolism of the level conditions

Sent data and line states is different information and consist of a different number of fields. Data are represented as hex value with a respective ASCII or control name, while the lines are listed as eight status and transition sequences.
To reach the same number of columns for the CSV export the data as well as the conditions of the lines are embraced by " ... ".
The conditions and transitions of all lines are described by the following names:

- `^` : High level
- `-` : Invalid level
- `v` : Low level

A sequence of `-v` describes a change from invalid to low level, while a level change from high to low is described by `^v`. The following extraction shall clarify this:

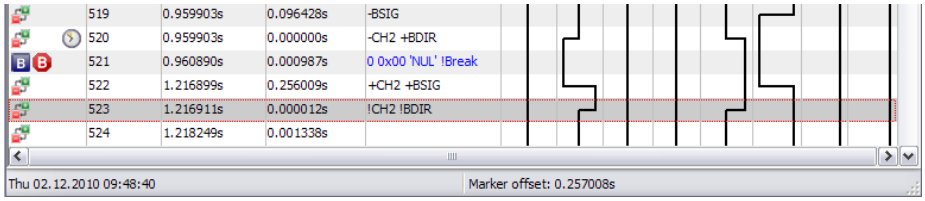
```
"Number","Type(Event)","Time(s)","SignalLevels"
0,L,17.359117,"-vCH1,-^CH2,-^CH3,-vCH4,-vBDIR,-^BSIG,-^IO1,--IO2",
1,L,18.408774,"vvCH1,^^CH2,^^CH3,vvCH4,vvBDIR,^vBSIG,^^IO1,--IO2",
2,L,18.408911,"vvCH1,^^CH2,^vCH3,vvCH4,vvBDIR,vvBSIG,^^IO1,--IO2",
3,L,18.409014,"vvCH1,^^CH2,v^CH3,vvCH4,vvBDIR,vvBSIG,^^IO1,--IO2",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

Please note, that the lines do not have to have the default a names since you can rename them according to your application. The new names appear instead of the standard names. Compare the chapter signal names in the control program.

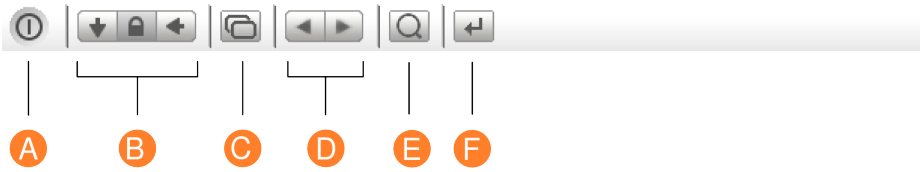
12.5 Measure time distances

Every event can be marked by right click the event line (item). Next to the right side of the type column a clock symbol is fade in and in the status line the time difference from the marked event to the current event at the cursor position is displayed.
A second right click onto the marked event removes the clock mark again.



12.6 The toolbar

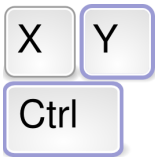
The tool bar is used for a quick access to the most needed functions. Some are identical to other views, some are specific for the event monitor.



A	**End:** Saves all settings and closes the window.
B	**Display mode:** According to the mode the window either shows always the current (last recorded) event or locked or actualizes its content synchronous to the other windows.
C	**New View:** Opens a new window with the same sector and settings.
D	**Event dependent scrolling:** Jumps to the last or next event of the same type like the one at cursor position.
E	**Event search:** Opens the level finder dialog for event search.
F	**Goto...:** Opens the Goto dialog to select the visible section by event number or time specification.

12.7 Short commands

Action	Short command
Online Help for the event monitor	F1
Opens the search dialog (LevlFinder)	Ctrl + F
Open Goto dialog	Ctrl + G
Jump to the time marker	Ctrl + T
Select all Events	Ctrl + A
Clear selection	Ctrl + Shift + A
Save selection as region	F4
Jump to last event of the same type	Ctrl + Up arrow
Jump to next event of the same type	Ctrl + Down arrow



Key commands of the most important functions

13 The Protocol View

With the analysis of protocols you enter the next level of communication. The seemingly arbitrarily occurring data are sorted and grouped according to your rules. Output functions allow you to format and color data sequences individually.

PLEASE NOTE - Template language has changed!
The Protocol View now uses the scripting language Lua for its template definitions and isn't compatible with templates written in version 3.2 or older any longer.

New template API

If you still use an older version of the Analyzer software (3.2.x) forget all about `in={...}`, `aout={...}` and `bout={...}`. The new template mechanism seems to look a little bit more complicate compared with the previous one, but you will soon appreciate and like the chances Lua will offer you. Especially when your telegrams show you the correctness of checksums, function names instead raw numbers or device names in place of address bytes. So here we go...

The exchanging of data between two or more communication partners generally happens depending on a protocol, which defines the format of the transferred data together with their content and meaning. The smallest data unit is called a telegram or datagram. While the data monitor displays the transferred data in the sequence of their occurrence without any interpretation (which sometimes has advantages) now the analysis of protocols and datagrams is the next level for understanding the communication.

For this, the data stream, captured by the analyzer, has to be split into single data sequences or telegrams before displaying them on screen. Since there are no defined rules (resp. many of different standards) for the definition of datagrams, a lot of different practical realizations are known. They vary from simple end-of-string characters (EOS), start (STX) and end (ETX) marks to the usage of certain pauses between single data packets (Modbus RTU, Profibus), run time length codes and other definitions.

Further more: Every telegram should be shown with certain information: number, address (bus participant), function code, data (in various formats), checksum, telegram delimiter and other things which will become needful when you

have to interpret or analyze a communication.

It's obvious that even a wide range of predefined protocol styles cannot meet all requirements. Especially when the analyzer program has to face individual protocol definitions or preferences. The Protocol View therefore handles both, the splitting of the continuous data stream into separate telegrams and the individual displaying of the telegram contents by an integrated script language.
Lua has already proved its suitability in the Data View, so it is more than logical to use it again as the protocol template scripting language.

The previous implementation of the Protocol View (until version 3.2.x) used a completely different design and couldn't offer conditions, which are essential if you want to handle data in the telegram depending on their position or telegram length. And: The former mechanism also lacks of a more flexible design to transform number of data bytes in other formats. For instance: if you like to show two sequent bytes as a decimal value or validate a checksum.

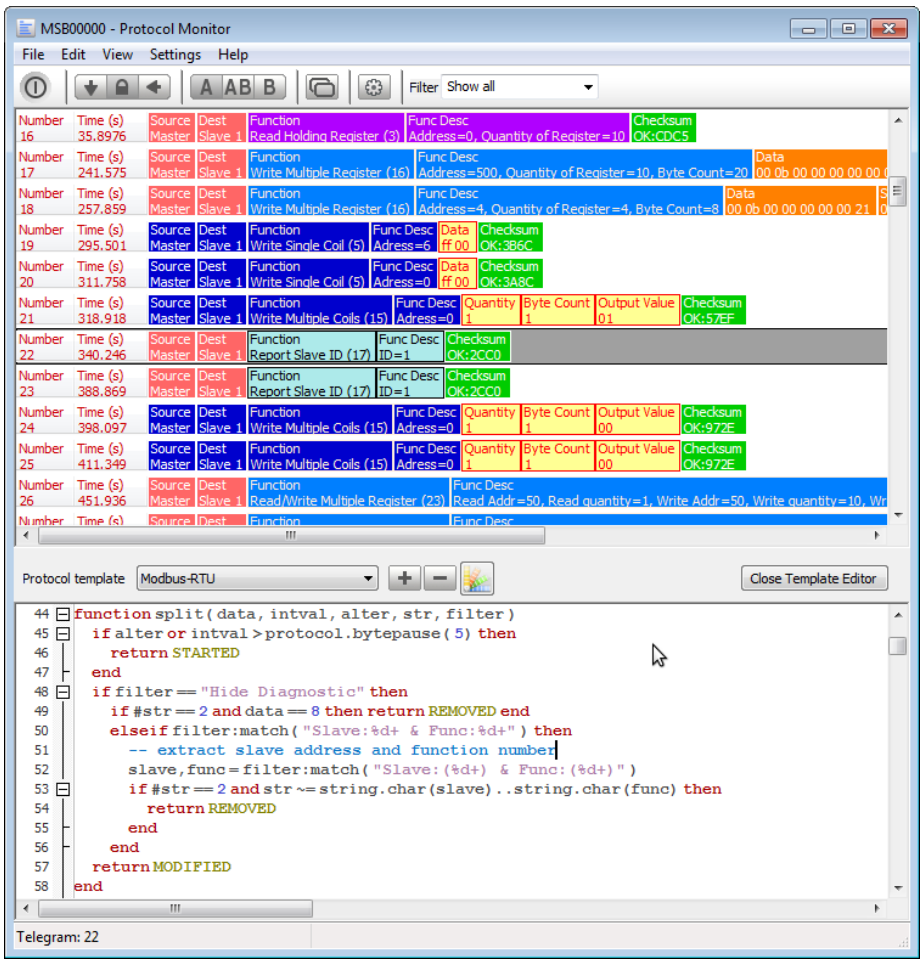
Lua provides you to create protocol templates with its full language strength. You can write your own functions for checksum validation (beside an already integrated checksum module which offers you some standard checksum algorithms), you can replace certain telegram function numbers with more readable names and hide telegrams you don't like to see.
And you can do this interactively and already during an active recording. The recording itself isn't disturbed. Your modifications are directly applied on your recorded data so you can see the changes immediately.
We will discuss the whole template scripting later. At the beginning let us show you the Protocol View in action.

13.1 User Interface

To make the protocol template adaption for your application as easy as possible the Protocol View integrates a powerful editor with full Lua syntax highlighting and all comfort you are expecting from a really good editor. The input of templates is interactive and directly effects the display of the recorded data, even while the logging is still running.

This capability of the Protocol View to define sequences using Lua as a template language exceeds -of course - the normal use of a fixed selection list. Admittedly this demands a certain learning process concerning the syntax and may sometimes be a bit hindering for trivial problems.

Therefore the user interface of the Protocol View offers both. Often used templates, e.g. standard protocols like Modbus RTU, Modbus ASCII, telegrams with 9-bit addresses, different combinations of carriage return and linefeed characters or changes in the data direction are simply to recall from a selection list. These templates are unchangeable (they are read-only, so don't worry to lose them) but can be easily added as a copy and then edited by the user. The following picture shows a typical application.



In the upper part of the monitor the recorded data are displayed according to the definition of the chosen template. In this case a Modbus RTU communication was selected. Each telegram is prefixed by an optional telegram number and telegram time (here relative to the record start time). But you can also define your own date and time format and disable the prefixed information at all.

The lower expanded window contains the editor with the relating template written in Lua. As mentioned before: the Protocol View comes with a lot of predefined templates for common use. The selection of a given protocol is always possible, also during an active session.

You can copy a predefined template for modification according to your application just by a simple click and without losing the original.
At first you probably will change only small things: the color of the data or the definition of a line end character. With F5 or a click onto the cogwheel symbol you directly apply your changes onto the logged data.
The predefined templates offer an easy start for own concepts. All new templates are automatically added to the list and saved (see chapter 13.3).

13.2 Protocol Display

The Protocol monitor displays every sequence (or telegram) in a single line. Each telegram can optional prefixed with one of the following information: the telegram number, the telegram time (absolute and relative to the record start), the duration of the telegram and the time distance to the former sequence. All this is easily changeable in the settings dialog.

Telegram time and index
You can add optional information for every telegram (independent of the used template) in the Settings menu under Settings→Configure Protocol Monitor...

The prefixed information also indicate the source (or direction) of the telegram. I.e. whether the telegram was recorded on Data channel A (red text) or Data channel B (blue text). In a bus application you will - of course - find several devices speaking on Data channel A, and others on Data channel B. Since all sequences do not always occur alternately every sequence is counted separately for each data direction. Uncompleted sequences, that means datagrams whose end condition is not yet reached (e.g. no end character received), are marked with a punctuation behind the telegram number.
Please note: The punctuation is only shown when the telegram number is displayed.

Synchronizing the display

All MultiView programs have in common that they can synchronize or lock their displays or always show the last recorded data (auto scrolling). That also applies for the protocol monitor. Leftclick onto the desired datagram to switch all the other views to the display of the marked datagram. The current datagram is framed.
Likewise the protocol display is sensitive to synchronization from other views. The sequence which is part of the synchronization is marked with the current line selection.

Choosing a range

With the export or copy and paste function you can process further any segment of the protocol in other applications. For that you first have to mark the desired area.
The selection is done like the file selection in your operating system. Place the cursor onto the first cell of the desired sector and click the left mouse key. After that shift the visible section to the end of the range and mark the end of selection with a left-click together with a pressed shift key. The field will become gray. If you want to mark all lines press Ctrl+A.

13.3 Protocol Templates

The protocol monitor contains a list of predefined templates for the representation of known protocols. By clicking onto the selection button of the template toolbar you can chose another template at any time.

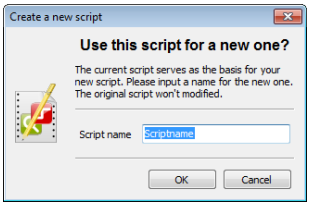
Template Toolbar
Editor inactive

The opening list shows you all available protocol templates. The adaption of the protocol view is directly done after selecting a list item.
In the beginning the list will only contain the predefined templates which come with the program. As they are samples for your own template definitions they can neither be deleted nor be altered.

Define your own templates

For the handling of the templates the protocol monitor follows a double strategy. On the one hand it offers a list of ready-made templates (as mentioned above), on the other hand their adaption for own applications shall be as easy as possible.
To change a protocol template or to create a new one first select a template from the list which should be used as a basic for your own modifications.
Afterwards open the template editor by clicking on the 'Open Template Editor' button or simply press Ctrl+T.

As a next step you can copy the current template with the now active '+' button in the template toolbar. The appearing dialog asks you for a meaningful name which will be used in the selection later. If you input an always existing name the program will warn you that you are going to overwrite a template.
The copied template can be modified in the editor window in any way. With F5 or click onto the wheel symbol in the toolbar the template definition is applied onto the data.
If the template is erroneous a respective message is fade in into the status line.



New template
by simple copy or altering of an already available template.

Apply template
with key F5.

Modify an available template

As already mentioned the predefined templates are secured against deletion and modification. This is allowed for your own templates only.
Simply select the favored template from the list and modify it according to your application. All changes are automatically saved every time the template is applied to the data.

Apply the template

Please note, that a change in the splitting rules requires a new formatting of the recorded data and may take a while depending on the size of the recording. The modification of the datagram representation affects the display only and is directly visible.

Template files and where you can find them

The ProtocolView uses two predetermined places for the storage of template files. All predefined templates are located in the `Scripts/ProtocolView` folder in your installation directory. Every template file in this folder has read-only permissions.
Own defined templates (as other program and session information) are individually saved for every user in a special application directory. Linux user will find that directory as usual in their home path under:

```
/home/username/.IFTOOLS/SerialAnalyzer/Scripts/ProtocolView
```

Under Windows the directory is located under:

```
C:\Users\username\AppData\Local\IFTOOLS\SerialAnalyzer\Scripts\ProtocolView
```

Every time a ProtocolView is opened it scans at first the script folder in the installation directory and then searches for existing template files `*.msbtml` in the IFTOOLS application data path. All found template scripts are afterwards listened alphabetically in the selection control.
You can also add and remove template files manually in the second folder, but we recommend to use the [+] and [-] buttons in the template toolbar in this case. Nevertheless there is sometimes another reason to handle single template files directly. This comes in handy when you like to share templates with other users or want to transfer them to another computer.

The template file manager

The protocol monitor offers you a simple file dialog to manage your own templates. With it you can import and export template files, rename or delete them. The template file manager only allows the operation with template files and protect you from other things.
You can open the file manager from within the file menu or simply by press Ctrl+M.

Import and export templates

To import a new template, just drag and drop it into the open template file manager. Likewise you can export or store a template by dragging it from the file manager to another location.
The imported files appear automatically in the template selection list afterwards.

Rename or delete a template file

You can edit every displayed template file in the template file manager. Just select the desired file with a normal left click. Afterwards left click the file name to start the editing. Change the name and finish the renaming with Enter (or left click a point outside the editor field). The file manager will give you a warning, if the new name already exists. It also checks the new file name for a correct extension.
To delete a file, just select it and press the Del key on your keyboard.
Please note! Renaming or deleting the currently used template isn't allowed for obvious reasons! The file manager indicates this with a little message every time you try to do this.
As soon as the file manager was closed, the template selector represents the changes of your own template portfolio in the selection list.

Open a template file directly

Alternatively you can simply drag and drop a template file into the open template editor. The ProtocolView replaces the current template with the new one and applies it immediately to the recorded data. The program will also warn you, in case a template with the same name already exists.

13.4 Template language syntax

Every template (file or script) has to provide at least two functions. The first one splits the incoming data into single datagrams (or telegrams). It contains the code which is necessary to decide when a new telegram starts and when it ends.
The second function let you control the appearance of every telegram in an enormous field. Here you can specify how the telegram content (or parts of it) are shown. For instance: You can convert a sequence of bytes into other numeric formats, validate a checksum or label data sections with your own description. And you can color various sections of the telegram in your own colors.
There is another - third function - to filter specific telegrams interactively using the filter control in the toolbar. But at first we will concentrate us to the two essential things, a template has to do:

1. **Splitting the data stream into telegrams**
2. **Individual displaying of the telegrams**

Splitting the data stream into telegrams

For the definition of a protocol template the first question has always to be: when does a telegram start and when does it end? Sometimes an end condition is sufficient, i.e. a Carriage Return and/or Linefeed, or a alternation of the data direction. But often the world is more tricky. You may thinking of binary protocols with certain pauses between every telegram like Modbus RTU, Profibus or similar.

The ProtocolView covers the complete splitting functionality in the function `split` as shown in the following.

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

This function is called every time a new byte arrived in the record and must return one of the following states:

1. STARTED → a new telegram begins
2. MODIFIED → the data doesn't do anything but increases the telegram length
3. COMPLETED → the telegram is complete
4. REMOVED → remove the current telegram for filter reasons
5. MARKED → mark the current bytes as telegram start and continues

It quickly becomes apparent that the current byte isn't enough to detect a valid start or end condition. For instance: An EOS (End Of String) condition consist of more than one byte. Or: A telegram is specified with a certain start AND a certain end.
That's why the `split` function is called with additional parameters. They are:

1. data → the current data byte (up to 9 bits)
2. interval → (short intval), the time distance to the former byte in seconds (with microsecond resolution)

3 alternation → (short alter), true when the direction has changed
4 string → (short str), all received data since the last telegram as a byte string
5 filter → the current selection of the filter tool passed as a string

You can rename the parameter for your own purpose but don't change the order of the parameter! It's also allowed to skip unused parameter from the right.

Ok, it seems more complicated as it is. Just let us make some little examples. Imagine a simple protocol where every telegram ends with a linefeed.

```
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
```

We don't need the filter parameter and therefore skipped it (line 1).
Our example lacks of a specified start condition. In other words: A new telegram starts with the first byte after the former telegram was finished with the linefeed character.
The parameter `str` is a Lua string and contains the whole data of the current (yet partly) telegram. In case of the first byte, the length of the string is 1 and we have to return STARTED. Lua offers you a special length operator '#' to query the count of bytes within a string (line 2). But you can also code it as

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

Line 3 specifies the end condition. The telegram is complete when a linefeed occurs. The parameter data contains always the current byte. We compare it with the linefeed (character value is 10) and return COMPLETED in case the condition is true.
In all other cases (the current telegram length is greater as 1 and the current data byte isn't a linefeed) the function returns MODIFIED.

Now let us adapt our little example to a protocol with a defined start and end string. For instance something like the Modbus ASCII protocol.

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

A telegram starts with a colon (character value is decimal 58) followed by the data (the data field only allows the characters 0-9 and A-F). The end of the telegram is marked with a Carriage Return and Linefeed (CRLF). The according `split` function is then:

```
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
```

Line 2 compares all occurring bytes with the colon and returns STARTED as soon as a colon is detected.
Line 3 searches for the CRLF in all currently received bytes (`"\r\n"` is the

Lua equivalent to CRLF). A found CRLF means the end of the telegram and will return COMPLETED.
All other data bytes are assumed as the telegram data and therefore only MODIFIED the telegram.

We have covered the parameters `data` and `str`. But what about the remaining `intval` and `alter`?
Imagine a protocol with alternately sent telegrams. Every telegram start is defined as a change in the direction and you won't bother with any further details. Here is a fitting `split` function:

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

The parameter `alter` is always true when the source of the current data byte isn't equal to the former byte. All we have to do is returning a STARTED then.

Some protocols use a defined pause (specified as a idle time of the transmission line) as a delimiter between the telegrams, i.e. the Modbus RTU protocol. The advantage of such a design is: The telegram doesn't depend on 'special' start and/or end characters and therefore can use a binary format for the data. (There isn't any data byte which must interpreted otherwise).

Telegrams with time gaps for framing

The Modbus RTU delimiter for instance is defined as a sending pause of 3.5 byte. Or generally speaking: The time which is needed to send 3.5 bytes with the current baud rate.
And that's where the last parameter `intval` comes into picture. `intval` is the time distance to the former byte in seconds. The resolution is - as usual - 1μs.
The transmission time for a byte depends on the baud rate. Luckily the ProtocolView provides you with some helpful functions. But first the `split` function code:

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

The code should be clear enough except for the `protocol.bytepause`.
The ProtocolView extends the Lua language with its own module functions. `bytepause` returns the sending time of the passed count of bytes for the current baud rate in seconds. So we just have to compare the time since the last byte with the calculated pause to detect a start condition. You will find a detailed description of the `protocol` module on page 127.

But hold on! What about the COMPLETED state?
The sending pause is both! The current telegram is COMPLETED by detecting the pause AND a new telegram is STARTED at the same time. In this case the ProtocolView marks the current telegram automatically as COMPLETED when a new telegram starts.

Uncompleted telegrams are shown with a series of dots in the prefixed number or index field (you can enable or disable the number field in the settings dialog).

Special case: Telegrams consisting of only one byte

Telegrams with only one byte are a particular case because the single byte represent both: A STARTED and also a COMPLETED state. Exceptions to this are only protocols with a idle time as a telegram delimiter like Modbus RTU or ProfiBus[1], because the telegram delimiter is independent of a certain byte.
But all other protocols with a predefined EOS (End of String) must consider the specific nature of a single byte message. For instance:
Your protocol terminates every telegram with a linefeed (LF). Beside this a single LF is used as an short acknowledge.
Without a special handling of a single linefeed the ProtocolView will display the first occurring LF as an incomplete telegram until another one arrives. Here is the split function of the 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
```

Since the first byte is marked as the beginning of a new telegram also a single LF will be handled in this way. You may consider to exchange the lines 2 and 3, but this isn't a solution. Then a linefeed will only be shown when it was the last byte of a sequence with different bytes.
The split function processes always one byte after another. In case of a single byte telegram the function therefore has to return both: STARTED and COMPLETED. Fortunately you can solve this just by combine both states in a single return statement. See below:

```
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
```

Line 4 returns STARTED and COMPLETED only when a linefeed arrives (line 3) and if it was the first byte in the telegram (line 2). Both conditions make sure that it is really a single LF.

Telegrams with a start sequence of more than one byte

There are certain protocols in which a telegram has to start with a byte sequence instead of a single character. For instance the DNP3 protocol uses the

[1] ProfiBus use a single byte message (SC) as a short acknowledge frame. The message consists only of the byte E5h.

bytes 0x05 and 0x64 (in hexadecimal notation) as a start and synchronisation header. A simple DNP3 frame looks like:

Start 0x05	Start 0x64	Length	Control	Destination	Source	CRC

Apart from the fact, that such a protocol has to make clear that the start sequence must not occur in the data payload or in general in the remaining part of the telegram frame, the `split` function nevertheless faces a problem here.

Consider the following situation: The `split` is called with an received byte 0x05 which - maybe - markes the start of a new DNP3 telegram. You can return the result STARTED - but hold on!
What about a following byte unequal to 0x64. In such a case the formerly returned result is simply wrong. You may suggest to wait until the next byte arrived before checking for the 0x05 0x64 sequence. Even so the returning STARTED will still be invalid because the telegram than won't start with the 0x05 but with the 0x64 and the very first byte of the start sequence will just ended as the last byte of the former telegram.
What we need is a mechanism which 'marks' a byte as a possible start and continue checking the next arriving characters before deciding on the final result.

Here the MARKED result comes into play. Returning MARKED doesn't start a new telegram. On the contrary, the parsing of the incoming bytes continues as if nothing had happened. First when the `split` function returns STARTED (and only STARTED), the formerly marked byte (or byte position in the incoming data stream) was used as a real new telegram start. An example:

```
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
```

How does it work?
The trick is to use a global variable `last` to hold the former data byte. By default Lua variables are always global and initiated with nil. Normally we recommend to avoid global variables and to use `local` in front of each variable definition, but there are sometimes exceptions. This is one of them.
The first line in the function (line 2) checks the present byte passed as parameter `data` with 0x64 and the former byte associated to `last` with 0x05. In the beginning `last` is not defined. Lua creates it on the fly with a `nil` content (see next section 13.4).
A matching comparison means the start of a new telegram and we simply has to return STARTED (line 3).
Afterwards we update `last` for the next call of `split` (line 5).
The following condition in line 6 makes sure that a new telegram start (trigge-

red by returning STARTED in line 3) is always associated with 0x05 (or in other words: MARKED).
If `data` is neither 0x64 nor 0x05 `split` ends up by returning MODIFIED to attach the current byte to the internal telegram sequence.

There is only one internal MARKED position!
A returning MARKED value in split always overwrites a former MARKED position and therefore the final STARTED returns the position of the last MARKED.

Global and local variables

Lua variables are global by default. They are accessible from all over the script after their first occurrence. But this leads sometimes to strong results in the script execution when equal named variables which are supposed to be independent share in fact the same content. Consider the following lines:

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

In line 12 we assign the actual telegram length to the variable `n`. Afterwards we calculate the checksum of the telegram by passing the telegram content as a Lua string to the function `chksum`. And here lurks the problem!
The function `chksum` internally also uses a variable `n` to summarize the several data bytes. But from the Lua interpreters point of view `n` ALREADY exists (declared by the assignment of the telegram length). Therefore `n` is first set to 0, then summed up with the telegram bytes.
When querying the variable `n` in line 16 we don't get the telegram length but the sum of the telegram bytes! That's not exactly what we want - isn't it?
You can - of course - just rename the `n` in the chksum function. However in huge templates this maybe means a lot of work and will not be as easy as originally thought.
A more simple solution is: Declare every variable used only in a function as 'local'. The appropriate Lua keyword is **local**. Applied to our example the shown modification of line 2 is completely sufficient:

```
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
```

The local variable `n` now only exists in the function `chksum` and hasn't any relation with the telegram length `n` in line 12.
But what about the variable `i`?
The counting (or control) variables within a `for` loop are local in general. They exists and are 'visible' only in the loop body.

You see: It is always a good idea to declare all variable as **local** in the first place. In case of a need for a global accessible variable we suggest to add a prefix to its name, i.e. `g_n` (global n).

Gain more information about the current data event

The arguments passed to the `split` should be completely sufficient in most cases. Nevertheless there are situations when you may need additional information for a correct telegram extraction. Such like the data direction (and not only the alternation state), the time stamp (and not only the distance).
Because every new parameter perhaps breaks the compatibility with older templates, the `split` function therefore supports access to these information with the `event` module. The module is described in the module section. Here just a simple example how you can determine the receiving source or direction of the current data event. The example presumes a protocol with two different EOS characters depending on the direction. All telegrams (and therefore data) received at port A (CH1) use a CR as a EOS, the telegrams from port B (CH2) were finished with a LF.

```
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 conclusion

The `split` function provides an adaption for almost all kinds of telegrams. Although the split code consists of a few lines, it became clear, that a little knowledge about Lua is necessary when writing your own templates. See chapter 16.1 for a complete introduction into this amazing script language.

Splitting the data stream in single telegrams was the first thing. In the next section you will learn how to format a telegram in your very own way.

Individual displaying of the datagrams

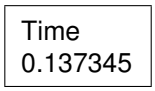
All the formating is done in a single `out` function. The function is always called when a telegram is drawn in the telegram window. This provides a greater

performance because only the code for visible telegrams is executed.

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

The principle of the new output mechanism based on a set of individual rectangular boxes in a row, whereby each row represent a single telegram.
Every box contains a caption line and a text specified by the user. Foreground and background color are free definable too. The caption and the text are Lua strings and therefore can be the result of any operation with the data in the relating telegram. The same applies for the colors. For instance:
The following picture shows a single Modbus RTU telegram realized with the new mechanism. The various elements of the telegram like the address, function number, etc. are shown as individual boxes. The last box displays the validated CRC16 checksum, here green for a correct value.

The size (width) of a box is calculated from its content and every new box is attached automatically on the right border of the previous box. This means: the box positions in the row depend on the order of their calls inside the Lua `out()` function. The first call of a box function displays a box leftmost, the second shows a box right to the first and so on...
A simple box will be defined like this:



A simple box

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

Please note! We don't show the split code here and focus on the output function. Later we will discuss the box model with an example, including a functional split method.
The code above will create the simple box as shown on the left. The caption or headline text is 'Time' (because we want to display the time stamp of the telegram), the content is the result of the `telegram:time()` call which we will explain in the next section.

The box will use black for the text and the outline and white as the background color as long as no color is specified.
Let us now extend the output with the data in the telegram.

```
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
```

For this we add a second box in line 4. Instead of iterating through all data and build the 'text' by ourself, the `telegram` luckily offers a much simpler way.
The telegram function `dump` returns any desired section of its data as a hex

data (dump) string. By default, without parameters, the full data sequence is used.
One word on the curly brackets of the `dump` call. Like the `box.text` they indicate that the function expects named parameters.

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

The box with the hex data appears behind the time box because it was called AFTER the time relating box. All data is shown as a 3 digit hex value per default (remember that the MSB-Analyzer supports 9 bit data values).
`dump` is one of the mostly used functions, just because it give you a first glimpse into the telegram without pondering about the size or content.

Telegram data access

As mentioned above: The `out()` function is called for every telegram (line) be displayed in the ProtocolView window. For instance: If the window shows the first ten telegrams, `out()` is called ten times, starting with the very first recorded telegram as created by the `split()` code and ending with the tenth.
When you scroll through the record and the window displays a section somewhere in between, lets say the telegrams from 1201 to 1217, then the `out()` function is called for the telegram 1201, 1202 until it reached number 1217.
The telegram relating to the actually handled line is accessible via the `telegrams` module with:

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

For instance: A box like the following

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

will display the telegram number of the currently in the function `out` processed telegram. But the `telegrams` module offers not only access to the actual telegram.
Imagine the displaying of the telegram structure depends on information of previous telegrams. Or you have to know the elapsed time since receiving the prior telegram, to decide if the actual telegram is a request or a response[2].

The `telegrams` module gives you random access to ALL telegrams, from the very first one until that one which is currently handled in the `out()` function[3].
You can simply indexing any desired telegram with:

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

The parameter `index` addresses the telegram in two different ways.
A positive index (absolute addressing) gives you the telegram of the passed index (or number). An index of 1 returns the first recorded telegram, an index of 100 the hundredth one. Indexing a not existing telegram will give you a clas-

[2] The Modbus RTU template makes use of this.
[3] You are not longer limited to the current and previous telegram as in former program versions.

sical Lua `nil` result.

By far more interesting are 'negative' indexes. Negative indexes stand for 'relative addressing' and are counted backwards (from the current telegram in the `out` function.
So means an index of -1 the actual telegram, and `telegrams.this()` is just an alias for it. An index of -2 accesses the prior telegram. And also here exists an alias: `telegrams.prev()`. Persons with Lua experience will surely not be surprised by this as Lua use negative indexes in several string functions too.
The following code demonstrates how you can calculate the response time between the actual and previous telegram:

```
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
```

Since `telegrams.this()` or `telegrams.at(-1)` always returns a valid telegram, this doesn't happen when one query the precursor of the very first telegram. Without the precaution in line 4 `tprev` will become nil when scrolling to the top. And `nil` means a lot of white emptiness in the telegram window.

In most cases using the `telegrams.this()` is sufficient. But the world of protocols is not always easy and sometimes a bus device reaction depends on an earlier received telegram type. If you like to mirror such a behavior in the telegram window, you have to iterate through the past telegrams.
The access time of `telegrams.at(index)` is linear, nevertheless to iterate through an undefined amount of telegrams means literally nothing good. There is always a risk for endless loops which the Lua interpreter punishes with an 'Overrun of allowed executions'. To avoid it, limit the iteration to an responsible number.

We have spoken a lot about telegram accessing. Now it's time to look after the resulting object - the returned type `telegram` itself.
The `telegram` type represents a single telegram as it is returned from a `telegrams` module function. It's like an container (or object) and covers all telegram relevant information like the telegram time, the size (count of bytes or data), the direction respectively source and so on.
In the example above `tcurr` and `tprev` are of the type `telegram`.
Don't confuse the type `telegram` with a module. It's rather like a number or a string and only exists as a result of a preceding call of a `telegrams` module function. You can assign the result (the type `telegram`) to a variable (as shown above) or process it directly. Therefore the following lines provide the same outcome. At first an approach without any intermediate step.

```
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() }
```

That's quite feasible, but leads to three identical and therefore unnecessary calls of `telegrams.this()`. A better way to achieve this is:

```
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() }
```

Differences between . and : in Lua

You may have noticed that the examples above contain a lot of dots '.' and colons ':'. We havn't explain it yet and you may still ask yourself what's the difference in using a dot or a colon.

The dot in `telegrams.this()` accesses the function `at` of the `telegrams` module. A module is - simply spoken - organized as a table and the function `at` is one of several table entries. The dot here refers to the `at` entry in the `telegrams` table (or module). For this you can regard a module also as a collection of functions.

But what about the `tg:number()`? It seems like the same kind of expression, namely to call the function `number()` of the `tg` 'object'.
I say 'object' deliberately! `tg` is a telegram variable or telegram object but it ISN'T a module. By using a colon ':' Lua is instructed to access the function (passed after the colon) which belongs to a specific variable/object (named before the colon). `tg:number()` therefore returns the number of the associated telegram `tg`.

```
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 the example above `tg` was first initiate with the current telegram (line 1), than asked for its number (line 3). Afterwards we assigned `tg` to the previous telegram. `tg` is now identical with the previous telegram. Asking its number again returns a different number, namely the number of the previous one.

The following rule may serve as a little mnemonic:

Use a colon ':' every time you can say: ≫ Variable, please do this for me ≪

Examine a telegram content

As said before: You can request several telegram information by calling the relating function. They are all listed in section 13.7. One of this functions I consider particular important because it will give you a quick view of the telegram data when you struggle with unknown content. The functions name is `dump{}` and you learned about it earlier in this chapter.
`dump` returns the content of the associated telegram as a Lua string, listing all data bytes as hexadecimal or decimal values. A `dump` call accepts the following named parameters, here with their default settings:

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

Without any given parameter, `dump` returns the whole content (first=1, last=-1) as 3-digit (width=3) hex values (base=16), separated by a space (sep=' '). The parameter max limits the maximal count of shown bytes and outputs only the first and last half n bytes, assigned to max.
Let's assume a telegram with the byte sequence:

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

And a simple `out()` function:

```
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
```

This will give you an output like this:

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

The argument *base* let you specify the number base. Default is hexadecimal (base=16), but you can also choose a decimal output with base=10.

```
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

Next we will limit the hex values to two digits, since the telegram contains only 8-bit data.

```
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

Ok, that was easy. Now imagine you want do display the last two bytes (the CRLF) in an individual End-Of-String box. `dump` offers you the two position parameter *first* and *last* to select any range of the content. You can pass the byte position as an absolute value, i.e. first=1 means starting with the first byte of the telegram. Or you count backwards with negative positions.

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

The last byte is indexed as -1. To select the last two bytes, we indicate a range of *first=-2* and *last=-1*. Accordingly we stop the dump of the former bytes at position -3 which means the byte before the CR. This is what we get:

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

As you can see: Using negative indexes is a very comfortable way to avoid querying the length of the telegram for absolute positioning.

The parameter *sep* is easy to understand. It just replaces the space or blank between the values with any other single character/string. And - of course - you can also remove the separator completely with:

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

Time	Data (hex)
2.351468	3A30323032304631353641343237440D0A

The ProtocolView handles telegrams without limit in size. Nevertheless it is sometimes annoying to scroll horizontally through a lot of data output by a `dump{}` call. Here comes the last parameter *max* into play. *max* specifies the maximum count of displayed data/bytes, one half at the beginning, the other half at the end. The remaining data in between were shown as byte count surrounded by ellipsis points. For instance:

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

gives you:

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

Create your own template step by step

For the next steps we recommend you to load or double-click the project file `Tutorial.msbprj` in the `Examples\ProtocolView` directory. The example also works without a connected analyzer and will show you every ongoing step in the further template adaption at first hand.

Tutorial
Tutorial.msbprj

The sample lesson contains a record of a simple protocol where each telegram starts with a colon ':' and ends with CRLF as an End-Of-Frame delimiter. You may already have discovered it in the pictured EOS box above (the data 013 010).
For this we can use the `split` function from the last section. Here is a remainder:

```
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
```

The protocol also specifies the device address of the receiver, a function number, some data and a simple checksum. If it reminds you a little bit of the Modbus ASCII you are right.
The project template is read-only, so you have to copy the template as a new one by clicking the + button (open the editor first) and input a script name, for

example 'MyTutorial' or something else. Otherwise you cannot edit it.

At first we will add some color in our current telegram display for we want to see the direction or source of every telegram. As usual we will show telegrams received at Port A (CH1) in red and the data at Port B (CH2) in blue.
We already mentioned that a box has a foreground and background color parameter. Colors are passed as a RGB hex value like `0xAABBCC`. The first byte (AA) specifies the red part (between 0...255), the second byte (here BB) the green part and the lowest byte (here CC) the blue part. For instance: black is 0x000000, white is 0xFFFFFF.
We will display all telegrams received at Port A with a red text on a light red background. And the data on Port B as a blue text on a lightblue background. Ok, here we go:

```
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
```

Line 9 defines a Lua table (or array) with two items for the text color, line 10 the same for the background color.
In line 13 we query the telegram to display and assign it to the local telegram variable `tg`.
The function `tg:dir()` returns the direction of the current telegram (1 or 2) and the result is used to select the relating text and background from the two color tables (line 16 and 17).
At last we just pass the two color values to all box calls (the box parameter *fg* specifies the foreground, *bg* the background colour) and - voila - after pressing F5 to execute the modification, the telegrams appear in two different colors.

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

The modifications will stored automatically every time you execute the template

with F5.
Next we will highlight the starting colon ':' (hex 3A) and the End-Of-Frame sequence (CRLF). This will help us to see any variation in the telegram itself caused by a telegram error.

```
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  function out()
7      -- telegram colors
8      local textcolors = {0xFF0000,0x0000FF}
9      local backcolors = {0xFFEEDD,0xDDEEFF}
10
11     -- access the current telegram
12     local tg = telegrams.this()
13
14     -- select the text and background color depending on the data source
15     local fc = textcolors[tg:dir()]
16     local bc = backcolors[tg:dir()]
17
18     -- display time
19     box.text{caption="Time",text=tg:time(),fg=fc,bg=bc}
20
21     -- start colon
22     box.text{caption="SOF",text=string.char(tg:data(1)),bg=fc,fg=bc}
23
24     -- display all data as hex
25     box.text{caption="Data (hex)",text=tg:dump{first=2,last=-3},fg=fc,bg=bc}
26
27     -- end of frame CRLF
28     box.text{caption="EOF",text=tg:dump{first=-2},fg=bc,bg=fc}
29 end
```

Line 22 calls a normal text box for the colon display. The caption or headline is SOF (Start Of Frame). The text parameter is the first byte in the telegram queried with `tg:data(1)`. Instead of showing the value of the colon (hex 3A) we output the value as a character with `string.char(tg:data(1))`[4].
The end of frame sequence don't need any transforming. We simply show the CRLF as a separate box with an inverse coloring (line 28).
At last we have to adapt the position and size of the hex display in line 25. The remaining data starts now at position 2 (the first byte is the colon), and ends with the last byte before the CRLF, the third last byte or -3. The result is shown in the following picture:

Time	SOF	Data (hex)	EOS
2.339189	:	030 032 030 032 043 034	00d 00a

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

The data is displayed with always three digits. This is the default since the MSB-Analyzer supports 9 bit data words.
In our example with don't have 9 bit data, so we can reduce the data representation to two digits. With the parameter `width` we can pass another count of

[4]The string module is part of the Lua language.

digits to the telegram `dump` function. Here the relating line 25:

```
25  box.text{caption="Data (hex)",text=tg:dump{first=2,last=-3,width=2},fg=fc,bg=bc}
```

and the result on the example of the 'red' telegram:

Time	SOF	Data (hex)	EOS
2.339189	:	30 32 30 32 43 34	00d 00a

Handle a base16 encoding

And now let us introduce some new things which go far beyond the abilities of the former protocol view.
Our example protocol simulates some kind of a bus communication. The bus consists of a sender and two devices which continuously read the temperature, air pressure and humidity.
The sender (or bus master) queries each device randomly for one of these information. The value in the answer is coded as a floating point number. Except for the starting colon and the ending CRLF all data bytes are sent as two ASCII characters (hex ASCII or base16 format). For instance: The byte hex 0x5B is encoded as two characters: 0x35 and 0x42 (0x35 = '5', 0x42 = 'B' in ASCII).
Last but not least a simple checksum provides the integrity of the telegrams. A single telegram looks like:

Start	Address	Function	Data	Checksum	End
:	2 chars	2 chars	0 or 8 chars	2 chars	CRLF

Please note: Queries have an empty data field!

In a first step we will convert the actual data from the base16 encoding back into its origin binary sequence. This will ease the later handling of the information packed in the telegram itself.
You can write a little Lua function to do this job, but the ProtocolView already can offer you a helpful `base16` module to handle such a coding in a flexible way. The module is described in detail on page 114.
To get the representation of a base16 coded string, just pass the according string to `base16.decode(string)` like this:

```
1  local tg = telegrams.this()
2  local bindata = base16.decode( tg:string():sub( 2, -3 ) )
```

`tg:string()` returns all bytes of the telegram as a Lua string.
The colon start character ':' and the end-of-string CRLF are not part of the encoding. Therefore we must only decode the substring from the second byte (position 2) to the third-last (position -3). To extract a certain sequence from a string is a frequent application and the Lua string module offers an appropriate function: `sub(first,last)`.
Calling the `sub` function directly from the string variable (or object) via:
`tg:string():sub(2,-3)` is all we have to do.
Please note! Since Lua strings can only consist of normal bytes, any 9-bit information is discarded. If you have to deal with 9-bit data, then you must access the single data with the telegram function `telegram:data(index)`.

The result is assigned to the local variable `bindata` which we will use for extracting all further information. The content of the binary data representation is thus:

Address	Function	Data	Checksum
1 byte	1 byte	0 or 4 bytes	1 byte

You can query any byte of a Lua string with `string:byte(index)`. The function works similar to `telegram:data(index)` and simply returns the byte value on the given string position (index). To display the address and function is easy to realize:

```
1  box.text{ caption="Address", text=bindata:byte(1), fg=fc, bg=bc }
2  box.text{ caption="Function", text=bindata:byte(2), fg=fc, bg=bc }
```

The checksum is the last byte in the binary sequence and it would be nice to see the checksum in a hexadecimal notation. To do this, we pass the value to the string belonging format function.

```
box.text{caption="Chksum",
         text=string.format("%02X",bindata:byte(-1)),fg=fc,bg=bc}
```

As mentioned above: The example protocol distinguishes between a request and a response. Only response telegrams contain an additional data field, encoded as a 4-byte floating point number.
Response telegrams are characterized by a length of totally 17 bytes (the original telegram with base16 encoding), or by a binary sequence of 7 bytes (address=1 byte, function=1 byte, data=4 bytes, checksum=1 byte). To distinguish it from a request, querying the size of the telegram or bindata length would be enough. For instance:

```
1  if tg:size() == 17 then ... end
2  if #bindata == 7 then ... end
```

It doesn't matter which one you are choosing. But since we refer to the binary data later in the response block, we use the second form. Here we go:

```
1  function out()
2      -- telegram colors
3      local textcolors = { 0xFF0000, 0x0000FF }
4      local backcolors = { 0xFFEEDD, 0xDDEEFF }
5
6      -- access the current telegram
7      local tg = telegrams.this()
8      local bindata = base16.decode( tg:string():sub( 2, -3 ) )
9
10     -- select the text and background color depending on the data source
11     local fc = textcolors[ tg:dir() ]
12     local bc = backcolors[ tg:dir() ]
13
14     -- display time
15     box.text{ caption="Time", text=tg:time(), fg=fc, bg=bc }
16
17     -- start colon
18     box.text{ caption="SOF", text=string.char( tg:data( 1 ) ), bg=fc, fg=bc }
19
20     -- the address field
21     box.text{ caption="Address", text=bindata:byte(1), fg=fc, bg=bc }

22     -- the function number field
23     box.text{ caption="Function", text=bindata:byte(2), fg=fc, bg=bc }
24
25     -- is it a response?
26     if #bindata >= 7 then
27
28         -- display the response data as hex
29         box.text{caption="Data (hex)",
30                  text=tg:dump{ first=3,last=6, width=2},
31                  fg=fc,bg=bc}
32
33     end
34
35     -- the checksum byte is always the last byte in bindata
36     box.text{caption="Chksum",
37              text=string.format("%02X",bindata:byte(-1)),
38              fg=fc,bg=bc}
39
40     -- end of frame CRLF
41     box.text{ caption="EOF", text=tg:dump{ first=-2, width=2 }, fg=bc, bg=fc }
42
43 end
```

Line 26 checks if the telegram is a response. If so (the `bindata` sequence contains at least 7 bytes), an additional data block is appended. An according telegram would look like:

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

Displaying the function as a number may be sufficient in most cases. But wouldn't it be not more convenient to replace the function number right with a function call description, so you can easily understand the meaning of the telegram? In our example protocol the functions are numerated as:

1 Temperature

2 Moisture
3 Pressure

A Lua function to return a text string relating to the function number may look like[5]:

```
1  function GetFunctionName( number )
2      local names = { "Moisture", "Humidity", "Pressure" }
3      return names[ number ]
4  end
```

Line 2 creates a Lua table (or array) with the three function descriptions. Because `names` is declared as **local** no other part of your code can access the table except for the code within `GetFunctionName`. This avoids conflicts when you have further `names` variables in other functions and therefore this should be always the recommended procedure!
Lua indicates tables starting with 1. Line 3 returns the table entry according to the given number parameter.

Lua allows you to put a function definition into another function. The function above can reside inside of `out()` but you can also place it somethere else. As a rule use an inside definition only in case of small functions (like the `GetFunctionName`) and write your additional functions outside of `out()` otherwise.

```
1  function out()
2      function inside()
3          -- do something
4      end
5      -- call the function
6      inside()
7  end
```

Ok, let us put the pieces together. The listing below shows the significant modifications. We add the function GetFunctionName() just inside of `out` and call the function with passing the function number in line 24.

```
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
```

[5]We assume that the number parameter is always in a valid range.

And that's the result for the first two telegrams:

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

The displaying of the function meaning is just an example to show you how to convert several parts of a telegram in a more human readable information. You can just as easy replace the address with a certain device name, but we are now going to focus our attention to the data itself.

In the protocol specification we said before, that the devices respond with a floating point value according to the received function number. These are values for the temperature, the moisture or pressure.
The floating-point format consist of four bytes in the `bindata` string (or eight in the original telegram sequence). The picture above shows the four response bytes in the Data (hex) box.
Our next task is to convert a sequence of bytes into a specific number.

Convert byte sequences into numbers

In dealing with protocols you will often have to transform a byte sequence to a certain number; and there are various forms of numbers: Integer values can be transmitted in two or four bytes, floating point numbers sent as four or eight bytes (double precision). And: Even the order of the transfered bytes matters. Some protocols put the most significant byte first on the line (Big Endian), others the lowest (Low Endian).
Luckily the Lua interpreter comes with a mighty function to handle all the different types.

The function `bunpack` expects two mandatory parameters: The byte sequence as a Lua string, and how to transform it as a second format string. An optional third parameter specifies the position within the string, when the conversion doesn't have to start with the first character.

```
pos,val1,... = bunpack( sequence, format, position )
```

The function returns at least two values. The first one (here pos) indicates the string position where the next conversion should take place. Then one or more results are following, depending on the format parameter.
Before we go back to our tutorial, here are some examples which may give you an idea how the `bunpack` works.

```
1  seq = "\248\036\001\000\154\153\045\065"
2  pos, i = bunpack( seq, "<L" )
3  pos, f = bunpack( seq, "<f", pos )
```

Line 1 creates a byte sequence coded as single values in decimal notation. For instance: A "\255" gives you a single hex FF byte, a "\104\101\108\108\111" is the same as the string "hello". The decimal notation allows us to form any sequence of bytes which we otherwise couldn't input with an usual keyboard.

The sequence above isn't been chosen by chance.
The first four bytes represent the 32-bit integer value 75000 with the least significant byte first (Little-Endian). The second four bytes are the binary imprint of the number 10.85, also in Litte-Endian format (LE). The following table shows the string in hexadecimal notation:

LE Integer 75000				LE Float 10.85			
F8	24	01	00	9A	99	2D	41
1	Byte position						8

Now let's see how the `bunpack` can provide us with the real numbers behind this sequence. We start with the 32-bit integer:

```
2  pos, i = bunpack( seq, "<L" )
```

The first argument of the call `bunpack` is always a string. The second (format) parameter `"<L"` specifies the kind of data, which we expect at the given position - passed as a third argument. Since the default position is equal to the string start, we can leave it out.
The magic behind the `bunpack` resides in the format string. One certain letter is assigned to one data type. A '<' or '>' in front of them defines the byte order. A '<' stands for little endian, '>' means a big endian interpretation.
There are a lot of different types understandable by the format parameter. They are listed in detail in the according section 13.7.

In our example above the format string `"<L"` indicates a 'long signed integer' in little endian. `bunpack` returns the position of the byte after the decoded long value. Here pos is 5 because the next value (the floating point number) starts at position 5. The second result `i` is the long value 75000 itself.

The conversion in line 3 starts at the position returned in the former call. Since the expected value is a floating-point number in little-endian order, we passed a `"<f"` as format directive.

```
3  pos, f = bunpack( seq, "<f", pos )
```

The outcome is again a pair of values. pos points to the ninth byte (the byte following the floating point sequence) and `f` is the floating point value 10.85.
In our example the two data (long integer and float) directly following each other. In such a case you can extract the data in one go and avoid passing the position parameter again and again.

```
pos, i, f = bunpack( seq, "<L<f" )
```

Fantastic - isn't it?
And since the ProtocolView allows you, to 'play around' with the format parameter it becomes easy to check whether a certain sequence exists in a little or big endian order, or if it contains an integer or floating point number. This is particular in unknown or undocumented protocols a great advantage.

Ok, after this little excursion into data conversion let's return to our tutorial. We

already transformed the hex data of the telegram in a binary representation. Remember, that a response telegram contains the requested value (passed as the function number) as a floating point number. Here the response telegram structure again:

Address	Function	Float Number	Checksum
1 byte	1 byte	4 bytes	1 byte

The following lines summarize all modifications to the `out()` function so far:

```
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     -- access the current telegram
12     local tg = telegrams.this()
13     local bindata = base16.decode( tg:string():sub( 2, -3 ) )
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     -- the device address
26     box.text{ caption="Address", text=bindata:byte(1), fg=fc, bg=bc }
27
28     -- the function number
29     box.text{caption="Function",text=GetFunctionName(bindata:byte(2)),
30              fg=fc,bg=bc}
31
32     if #bindata >= 7 then
33         -- it is a response
34         local next, value = bunpack( bindata, "<f", 3 )
35         box.text{ caption="Value", text=value ), fg=fgColor, bg=bgColor }
36     end
37
38     -- the checksum byte is always the last byte in bindata
39     box.text{caption="Chksum",
40              text=string.format("%02X",bindata:byte(-1)),
41              fg=fc,bg=bc}
42
43     -- end of frame CRLF
44     box.text{ caption="EOF", text=tg:dump{ first=-2 }, fg=bc, bg=fc }
45 end
```

The according line 34 should be readily comprehensible. In case of a response telegram we extract the floating point number and display it in an additional text box as value.

The checksum validation isn't yet part of our telegram display. We will look into it later. You may also notice that we 'dump' the EOS with a relative (negative) index. Thus it's easy to access the EOS bytes without passing the telegram length. The output for the very first two telegrams is:

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

Compared with the first steps it's a lot more understandable, isn't it?
But we have not yet reached the finish line. There is still place for some improvements. For instance: The floating point values are shown with a lot too much digits. And it would be nice to validate the checksum.
Lua comes with an internal `string` module which offers you, beside other string operations like search, replace and regular expressions, a string format function similar to the printf in C.

```
35  box.text{ caption="Value", text=string.format( "%.2f", value ) }
```

`string.format` understands a lot of types and options, for more information please refer to one of the Lua online manuals as listened at the end of the chapter. Here we use the format string `"%.2f"` which formats the given floating point value ('f') with 2 fractional digits.
You can add an individual format string with the physical unit for each kind of value as a little exercise. The solution is shown partially below:

```
1  function GetFunctionValue( number, value )
2      local formats = { "%.2f Deg", "%.2f%%", "%imBar"}
3      return string.format( formats[number], value )
4  end
5
6  if #bindata >= 7 then
7      -- it is a response
8      local fnc = bindata:byte(2)
9
10     local next, value = bunpack( bindata, "<f", 3 )
11     box.text{caption="Value",text=GetFunctionValue(fnc,value),fg=fc,bg=bc}
12 end
```

The code should be self-explanatory maybe accept for the `"%.2%%"` in line 2. The percent sign is used as a placeholder for the given value. If you like to use it as part of the output string, you have to quote it with a leading percent sign or simply spoken use two of them.

Checksum validation

Last but not least we will finish the introduction of the template mechanism with a checksum validation. Our goal is to display valid checksums in green and invalid ones in a warning orange.
The protocol checksum is calculated by add up all bytes starting with the address and ending with the last data byte. The colon and the CRLF are not part of it. Carries have to be discard.
The checksum validation function looks like:

```
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
```

The checksum function is called with the byte sequence (string) we want to summarize and returns the lower 8 bits of the sum.

```
1  local chksum = Checksum( tg:string():sub( 2, -5 ) )
```

The colon isn't part of the checksum as mentioned above. So we passed the substring start position 2. Also the checksum itself (2 bytes) and the CRLF (also 2 bytes) has to be excluded. Which means an ending substring position 4 bytes lesser as the telegram size or the fifth byte counted backwards.
Finally we compare the calculated and the read checksum and - depending on the result - add a good or a bad checksum box.

```
1  local chksum = Checksum( tg:string():sub( 2, -5 ) )
2
3  if chksum == bindata:byte( -1 ) then
4      box.text{ caption="Checksum", text=string.format("%02X", chksum),
5          fg=0xFFFFFF, bg=0x00cc00 }
6  else
7      box.text{ caption="CHKS ERR!", text=string.format("%02x", chksum),
8          fg=0xcc0000, bg=0xffff88 }
9  end
```

You will find the complete template in the `examples\ProtocolView` folder as `complete-sample.msbtml`. The sample contains also one invalid checksum in the third answer telegram to demonstrate the correctness of our checksum validation.

Named parameters

A few additional words regarding the parameter passing. You may here noticed that a lot function parameter follow the conversation:

```
parametername = value
```

This is not Lua typical but we decided that so called named parameters are more convenient and even more understandable. And: you don't have to worry about the parameter order. For instance:

```
1  box.text( "Func", "Command", 0xFFAAAA, 0x0000FF )
```

Without a look in the manual it's hard to get the meaning - isn't it? What's the caption, what's the text color?
On the other hand the same code with named parameters:

```
1  box.text{ caption="Func", text="Command", fg=0xFFAAAA, bg=0x0000FF }
```

The meaning is obvious (although you have to remember that fg stands for foreground color and bg means background color).
Please note: Named parameters are always included between an opening and closing brace `{...}` because Lua sees the parameter as a table. Normally you would have to write: `function({...})` but the outer brackets are optional here and you can forego them.

13.5 Filtering

Filtering of the data or telegrams is an often demanded feature. But how to filter unknown types of telegrams? From the viewpoint of the program a predefined list of filters doesn't make many sense since every telegram has a very special need of what you want to show (filter) and hide in the output.
For instance: You want to see only telegrams of bus participants with a certain address or certain function. In this case the filter mechanism must be able to extract and compare the address with the given filter parameter.
It is obviously that the filtering therefore has to be part of the template.

It's now the appropriate time to introduce the last parameter in the `split` function. In case you don't remember the call of the function. Here it is again:

```
1  function split(data,inval,alter,str,filter)
2      -- you split code
3      return STATE
4  end
```

The filter parameter is just a text string containing the current selected item of the filter control in the toolbar. But with it you can pass any desired data (as a string) to the `split` function. The real filtering has to be done in the `split` function itself.
You may comment, that the filtering is surely better in the `out` function. But the `out` function only displays the visible telegrams. It cannot remove (filter out) a single telegram without creating a discrepancy between the visible and available telegrams. I.e. you can hide all telegrams but nevertheless the count of telegrams is unchanged and the scrollbars will tell you that by scrolling through an empty list.
As usual it is the best way to explain the filter mechanism with the aid of an example. Load the tutorial project again and select the Tutorial-Complete template. Copy the template with the [+] button so that you can modify it by yourself.

The tutorial record shows the communication with two devices. The first one has the address 1, the second the address 2.
And now imagine you can simply list only the communication between the master and the first device. Or showing the telegrams relating to a certain query, for instance all requests and answers for the temperature.

To filter certain telegrams means to 'remove' all of them you don't want to see. For this the `split` function can return the state REMOVED. First we will only list the telegrams with the first device (address 1). To do this we have to suppress all telegrams to and from the second device. The address is coded as two hex ASCII characters in the second and third byte of the telegram. Here - for instance - the very first telegram as it is shown in the DataView:

3A	30	32	30	32	43	43	0D	0A	:0202C4..

To detect a telegram with address 2 we just have to look for the string "02" on the second position. In Lua formulated it is:

```
1  if str:find("02") == 2 then ...
```

Add the line in our `split` function and return the state REMOVED when the condition is true (see line 3 in the following code).

```
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
```

Now hit the F5 key (or click the execution button in the toolbar) - voila - all remaining telegrams show only the communication with the first device.
When you replace the string "02" with "01" the display lists only the telegrams of the second device.
Nevertheless it is bothering to change the template all the time when you just want to look for telegrams with the other address. Here's where the filter parameter comes into place.
The filter control in the toolbar passes any inputed text string to the `split` function as the filter argument. With this it becomes easy to set the address of the not wanted devices. We only have to replace the address string "02" in line 3 with the filter parameter as shown in the following code.

```
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
```

Without a given input or in case of an unfitting address the REMOVED condition is always ignored and all telegrams are displayed in the telegram list. But as soon as the filter control text matches the address of the current telegram at position 2, `split` returns a REMOVED and the telegrams with the given address will be hide in the display.

You can check your code afterwards by input a 01 in the filter control and press Enter. All telegrams relating to the first device should disappear. Then change the string in the filter control to 02 and hit the Enter key again. The remaining telegrams show the address 01.
Now let us extend our little example and implement a filtering for the three function codes:

```
1 Temperature
2 Moisture
3 Pressure
```

The function number is transmitted in the 4th and 5th data byte of the telegram. The user should be allowed to select a certain function in the filter control. Only telegrams containing that function should be displayed in the telegram window. For instance:
In case of an input function number 1 (temperature) all telegrams according to the moisture and pressure have to be REMOVED.

```
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
```

The code above filters all other telegrams except for that which was entered as a number in the filter control. But sometimes it's hard to remember the correct function number. Especially when a more readable function name is provided by the protocol.

In the next step we will improve the handling by adding predefined selection strings in the filter control. We will introduce a special `filters` function to you, which automatically fills the filter control with a specified list of entries. The definition of that function is simple:

```
1  function filters()
2      return "Temperature,Moisture,Pressure"
3  end
```

The `filters` function must return a single text string whereas each item for the filter control is separated by a comma. The first item appears at the top of the selection list, the last at the bottom.
You can place the `filters` function at every point of the template script but not within another function. We recommend to insert the function on the top of the script.
As soon as a `filters` function is detected by the internal script engine it fills the filter control with the items in the returning string. Here we get the items: Temperature, Moisture and Pressure.

At least we change the REMOVED conditions in `split` and use the items above instead of the simple function numbers. (You will find the complete template as `tutorial-complete-with-filtering.msbtml` in the examples folder).

With the changes in the template code below (see line 3, 7 and 11) the user is able to select one kind of the telegrams in the filter control independent of some function numbers. And since the filter mechanism is part of the template you can provide every protocol template with an exactly matching filter handling.

```
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
```

13.6 Export Telegrams

The MSB-RS485 program is well equipped for most analyzing intentions. Nevertheless there are situations when you have a need for processing the recorded data - here the telegrams - with additional tools or external applications which are specialized in certain things the analyzer software cannot handle.

The Protocol View supports an unlimited number of protocols thanks to the ability to let the users write their own templates for displaying the telegrams. But this also means that the exported data directly depends on the individual templates.
The ProtocolView therefore uses an intelligent mechanism to share the telegram information with other applications. In most cases the mechanism will work out of the box. But it's still good to know how the program determines what information are suitable for the export. In particular when you intend to use the telegram information in a spreadsheet application.

How the program determines the export fields

In the prior sections you learned all about the basic box model. Each box consists of a indicating caption and the relating information. By labeling a certain information you already assigned this data with a name. For instance:
You have a telegram field (or box) which shows the device address. It's obviously that you name the field as 'Address' or likewise. And it is only logical that you want to export the information (all telegram address fields) under the same name.

The assignment `caption="Field Name"` in the template script therefore becomes the elementary part of the export mechanism. Every time you open the export dialog, the program extracts all caption labels and handles them in an internal list. The prefixes chosen in the settings dialog are put in front of the list. Than the list is shown to the user who can select or deselect single or multiple items.
Except for the prefixes (which were displayed always at the beginning) all extracted fields are listed alphabetically. This is due to the fact that the order of

the `caption="..."` assignments in the template script doesn't say anything about the relating field order in a telegram.
The actual export process is similar to the displaying of the individual telegrams and only depends on the chosen export format.

1. **Export as CSV**
2. **Export as HTML**
3. **Export as Text**
4. **Export as Latex**

All selected telegrams are fed through the Lua interpreter. The script engine assigns the data according to the caption name into the right column of the CSV file or creates a HTML table cell with the same text and background color as shown in the telegram itself. Not selected telegram fields are suppressed and don't show up in the export file.
Please note: Each export format serves a different purpose. In a CSV file every possible field represents a column. But not every telegram is composed of all fields. For instance: Some telegrams may consist of additional data fields, other short telegrams are hardly resembling more than an acknowledge. All fields (column items) which are not part of the current telegram are left with an empty string "".
A HTML export on the other side is used to represent every telegram as shown in the program window for documentation. For that reason the HTML export creates a single HTML table for every telegram, whereas every table consists of only that fields, which are part of the current telegram and were selected formerly in the export dialog.

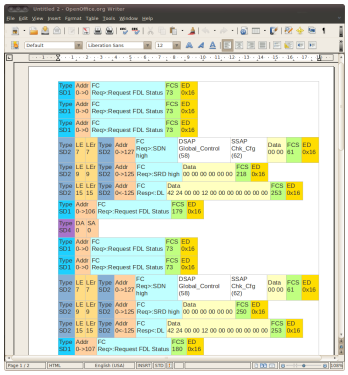
The export dialog

Before you start an export you have to select the wanted telegrams. Without a chosen range of telegrams, the one marked by the cursor is used. The export dialog is opened with Ctrl+E or by click on the export item in the file menu.
The dialog window presents you a list of all available telegram fields as well as enabled prefixes and preselect all of them. You can disable respectively enable each item singly. Or you select or deselect all in a single rush with click of one of the two buttons below.
The default export format is CSV (comma separated values), but you can likewise use HTML as an output format.
After input a valid file name (the program default is the file `telegrams` with the according extension on your desktop), the export starts as a parallel process. You can always stop the export by clicking on the 'Cancel' button on the left side of the progress gauge in the status bar. And you can continue examining the record while a longer running export is in progress.

Export as CSV file

Imagine you want to find the maximum time between a request and response. Or you are interested in some statistic about the frequency of telegrams with a certain type. There are a lot of questions which are better handled by a spreadsheet programs like Microsoft Excel®, Open Office Calc or similar tools. The ProtocolView therefore offers you an easy way to export all displayed information as a CSV file with comma separated values.

All column values are quoted with quotations marks and can be easy imported by most of all spreadsheet applications. The headline of the CSV file consists of the column names as extracted from the caption assignments in the template script.
If you are interested only in a few data deselect all unnecessary fields to quicken the export.



Open Office Writer
with html telegram export

Export as HTML

The HTML export is mainly intended for documentation purposes. The program outputs the selected telegrams as a valid html document whereas every single telegram is rendered as a html table, representing the telegram as shown in the Protocol View window. This includes besides the data information also the text and background color.
Most text processing applications are able to import such a html file. Open Office user for instance can simply drag and drop the file into their documents (see the example picture on the left side).

Export as text

This kind of export outputs the content of the selected telegrams as a sequence of `Label(VALUE)` expressions. It comes into play when you want to use the telegram information in a raw text environment or when you documentation/report tool isn't able to handle graphical objects. In contrary to the (also raw text) CSV output, the text export of a telegram only contents the existing telegram fields which makes it - under some circumstances - a lot easier to parse the information for further processing. Here a short excerpt from a text export:

```
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)
```

Export as Latex

This format is mainly intended for users who prefer LaTeX as their text documentation system. Each selected telegram is exported as a `tabular` environment whereas every box is represented by a table cell. The several cells are colored like the boxes in the telegram window, using the additional LaTeX packages `color`, `colortbl` and `xcolor`.
Before inserting a exported telegram, make sure that you add these LaTeX packages with the following command at the beginning of your LaTeX file:

```
\usepackage{ color , colortbl , xcolor }
```

You can switch on/off certain telegram fields before exporting them. But it is also easy to remove unnecessary entries in the table later in the `tabular` code.

Special notes about the caption labeling

We mentioned above: The export fields (columns in CSV) are named by the caption assignment in the template box functions. As long as you are using plain text and unique names the export results will look as expected.
But considering the following box with a composed caption label including the current function number. (A more readable description of the function is displayed in the text variable).

```
1 box.text{caption="Fnc (".. tg:data(2)..")", text=GetFuncDesc(tg:data(2))}
```

The displayed output may be something like this:

Func (8) This is function 8

The export mechanism unfortunately cannot assemble the complete caption label. For this it has to execute the template with all telegrams BEFORE it can start the export itself. And: A lot of different Fnc(...) labels leads to a confusing amount of different export field names when only one field (the function number) will be needed.
Remember: The export mechanism searches for assignments in the form of `caption="NAME"`. In the example above the extraction of the caption name will give you `"Fnc ("` and discards the remaining expression. When your caption name starts with an evaluation, i.e. `caption=tg:data(1).."-Typ"` the search for `caption="` fails completely and therefore neither won't be listed in the export dialog nor exported at all.
The same occurs when you are using a variable for the caption. I.e.

```
1 label = "Chksum OK"
2 if ChecksumTest() == false then
3     label = "Chksum fails"
4 end
5 box.text{caption=label, text=GetChecksumByte()}
```

Also here the search for a pattern like `caption="..."` fails and both possible caption labels won't be added to the list of exportable fields.
Other effects may be less significant. Nevertheless it's good to keep them in mind. The extraction mechanism cannot understand whether the caption assignment is part of a out-commented section or in between a function which happens to be never executed. In both cases the export dialog will show the according field names. But this will give you at worst only empty records in a CSV column.
So as a conclusion: Just avoid compounded expressions for the caption and only use plain text for it!

13.7 ProtocolView specific Lua extensions

The following section covers all Lua modules, functions, extensions and data types which are not necessarily part of the Lua language but especially implemented or added for the ProtocolView. Lua offers - naturally - a lot more data types, modules and functions which we - alas - cannot handle here.

- **base16 module :** Encoding and decoding functions for base16 sequences (i.e. used in Modbus ASCII and Intel SRecord).
- **box module :** The box module is responsible to display the data of each telegram.
- **bunpack function :** Converts a given byte sequence in a certain number. The bunpack function is part of the official Lua lpack library and firmly integrated into the ProtocolView Lua interpreter.

- **checksum module :** Contains checksum algorithms for Modbus RTU (CRC16), Modbus ASCII (LRC), BACNet (CRC8 and CRC16), DNP3 and CRC16 CCITT (Kermit).
- **datetime module :** The datetime module offers functions to output the telegram time in a user specific format.
- **debug module :** The debug module let you output debug information in the debug window.
- **event module :** The event module is only available in the split function and gives you access to additional information of the current data event.
- **linestates module :** With the linestates module you can check if a certain line signal has changed or query the number of a given signal alternation.
- **protocol module :** Returns information about the current baudrate, data bits, parity and stopbits.
- **record module :** Provides information about the record. For now only the start time of the record and the used bus-wiring (MSB-RS485) are implemented.
- **shared module :** The protocol template mechanism uses an independent Lua interpreter for every data direction. The shared module let you share data between both of them.
- **string.dump :** Extents the original Lua string module with a hex/dec dump functionality like the telegram relating dump function.
- **telegram type :** A ProtocolView specific data type. The telegram type covers all information about a single telegram.
- **telegrams module :** The telegrams module gives you access to all occuring telegrams up to the present time. The result is always a variable of the type `telegram`. It replaces the tg and tgprev module which were limited to the current and previous telegram. The `telegrams` module it is only accessible in the `out()` function.

The several types, functions and modules in alphabetic order:

The base16 module

The `base16` module provides you with two helpful encoding/decoding functions when you have to deal with telegram data transmitted in base16 (hex ASCII) format. This concerns in particular the Modbus ASCII protocol or SRecord transmissions.

Function	Description
decode	Deciphers a base16 encoded data string and returns its binary (original) content.
encode	Converts a given Lua string to its base16 representation and returns it as another string.

base16.decode

Converts the given base16 sequence back into its original binary representation and returns it as a string. The decoding will stop automatically when it reaches the end of the passed string or when it encounters an invalid character.

base16.decode(*string*)

- **string:** A base16 encoded data sequence.

Example

```
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
```

base16.encode

You wont normally make use of this function, since it converts a Lua string in its base16 representation. Nevertheless there may exist situations in which you like to see a binary sequence in a base16 encoding. I.e. if you like to compare a given known string with a result received by the analyzer.

base16.encode(*string*)

- **string:** A Lua string which has to be converted into base16.

Example

```
1  function out()
2      local seq = "hello world"
3      box.text{ caption="Base16", text=base16.encode( seq ) }
4  end
```

Function bunpack

The function `bunpack` comes originally from the Lua lpack library and is a close integrated part of the Lua interpreter in the ProtocolView. It provides you with all necessary kind of transformations you need to convert any byte sequence into a certain number type.
`bunpack` works like the scanf function in C. A given string or byte sequence is translated in one or more numbers specified by an additional format string. Since Lua functions are not limited to a single returning value, the conversion results can be assigned to several variables in one step.
A third position parameter let you start a conversion from a different position instead of the default first sequence byte.

```
pos,val1,... = bunpack( sequence, format, position )
```

The following list shows the most important format/transform specifiers defined by the `bunpack` function.

Format	Description
`b`	Interpret the next byte as a single unsigned byte (8-bit) value.
`c`	Interpret the next byte into a single signed byte (8-bit) value.
`d`	Convert the next 8 bytes into a double floating-point number (a floating-point value with double precision or 64 bit).
`f`	Interpret a series of 4 bytes as a floating-point number (32 bit).
`H`	Convert the next 2 bytes into an unsigned short number (16 bit).
`h`	Convert the next 2 bytes into a signed short number (16 bit).
`L`	Convert a series of 4 bytes into an unsigned integer number (32 bit).
`l`	Convert a series of 4 bytes into a signed integer number (32 bit).
`>`	Interpret the sequence with the most significant byte first (big-endian order).
`<`	Interpret the sequence with the lowest significant byte first (little-endian order).

bunpack(*sequence, format, position=1*)

- **sequence:** A Lua string which has to be extracted (unpacked) to one or more numbers.
- **format:** The conversion format applied to the given sequence.
- **position:** The byte position where the conversion has to start. Default is the first byte of the given sequence.

Example

Imagine a Modbus-RTU 'Write Single Register' command. The structure of the telegram is thus (byte sequence):

Dev	Fnc	Reg HI	Reg LO	Value HI	Value LO	CRC HI	CRC LO

This Modbus telegram commands a device to write a 16 bit number into a given register specified by its 16 bit address. The register address is in the 3th and 4th byte, the register value in the 5th and 6th. The last two bytes contain the CRC16 checksum. The bytes are arranged in big-endian order.
With `bunpack` you are able to extract the register address, register value and CRC16 in one single step:

```
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
```

The conversion starts with the first byte in the sequence (position 1) and interprets the following bytes according to the instructions in the format specifier. Finally the function returns the position of the byte next to the last conversion and fills the remaining variables on the left side with the results.

Dev	Fnc	Reg HI	Reg LO	Value HI	Value LO	CRC HI	CRC LO
B	B	>H		>H		>H	

Don't worry about too few variables on the left side. Lua takes care and only assigns results to the existing variables. So the following code is also correct but lacks of the CRC16 checksum value.

```
local pos,dev,fnc,reg,val = bunpack(tg:string(),"bb>H>H>H",1)
```

The box module

This module provides you with the necessary 'boxes' you need when displaying any content in the telegram window. The basic box is the 'text' box. It allows you to display any information (numbers, hex sequence, text) with a certain label (caption) in a rectangular shape.
Each box can have it's own individual text and background colors, passed with the named parameters `fg` and `bg`. The default is a black text (and outline) on a white background.
But you may find also the 'space' box sometimes helpful in case you want to set several boxes apart.

Function	Description
box.space	The space box inserts just an empty space with a width given in characters or pixel.
box.text	A common box with free definable caption, text content, foreground color (text and outline) and background.

box.space

Inserts an empty space with a given width. Without a parameter a space of 10 pixel is used. You can specify the width as pixels or as a number of characters. The latter respects the current font size (zooming) which means: The 'space' grows with the font magnification.

box.space{ *em=0, px=10* }

- **em:** the width defined as count of space characters the max count is 100.
- **px:** the width defined as pixel. The max width is 100.

Example

```
function out()
    -- insert a small space with the width of two space (blank) characters
    box.space{ em=2 }
    -- insert an empty space with the width of 50 pixel
    box.space{ px=50 }
end
```

box.text

Display a common box with individual colors, caption and content string. The size (width) of the box is automatically adapted to its content.

box.text{ *caption=STRING, text=STRING [, fg=RGB, bg=RGB]* }

- **caption:** a string displayed as caption.
- **text:** a string displayed as the data content.
- **fg:** Optional RGB color for the text and outline, default is black.
- **bg:** Optional RGB color of the box background, default is white.

Example

```
function out()
    box.text{ caption="Caption", text="Some text", fg=0xFF0000, bg=0xAADDFF }
end
```

The checksum module

The checksum module always comes in handy when your protocol uses one of the following checksum algorithms listed below (more will be added in the next future).
All functions of the checksum module expect a Lua string as parameter and generate the checksum by iterating over all data in the string relating to the selected algorithm. The checksum is returned as a integer number.
Please note that some applications use a different order of the 16 bit value. Modbus RTU telegrams for instance transmit first the low byte of the crc16 checksum, then the high byte.
See also section 13.4 in case your checksum isn't listed here and you have to write it by yourself.

Function	Description
checksum.crc8_bacnet	the 8 bit checksum as used in the BACNet (header) telegrams.
checksum.crc16_bacnet	the 16 bit checksum as used in the BACNet protocol.
checksum.crc16_ccitt_kermit	calculates the crc16 checksum of the given data string using another start value as used in CCITT kermit.

checksum.crc16_dnp3	calculates the crc16 checksum of the given data string as used in the DNP3 protocol. The returning result is a 16 Bit value.
checksum.lrc	returns the checksum of the given data string calculated as Longitudinal redundancy check (used in Modbus ASCII.
checksum.crc16_modbus	calculates the Modbus RTU (CRC16) checksum of the given data string and return it as a 16 bit integer.

checksum.crc8_bacnet

A checksum algorithm for BACNet (header) telegrams. The result is a single byte (8 bit value).

checksum.crc8_bacnet(*String*)

- **String:** the data as a string

Example

```
1 function out()
2     cks = checksum.crc8_bacnet( telegrams.this():string():sub(1,-2) )
3     box.text{ caption="Checksum", cks }
4 end
```

checksum.crc16_bacnet

The crc16 checksum algorithm for BACNet telegrams. The result is a 16 bit integer value.

checksum.crc16_bacnet(*String*)

- **String:** the data as a string

Example

```
1 function out()
2     cks = checksum.crc16_bacnet( telegrams.this():string():sub(1,-3) )
3     box.text{ caption="Checksum", cks }
4 end
```

checksum.crc16_ccitt_kermit

Returns the CRC16 CCITT (Kermit) checksum of the given data string as an integer.

checksum.crc16_ccitt_kermit(*String*)

- **String:** the data as a string

Example

```
function out()
    -- the following code checks the content of the entire message except
    -- for the last two byte (which are the checksum itself)
    cks = checksum.crc16_ccitt_kermit(telegrams.this():string():sub(1,-3))
    box.text{ caption="Checksum", cks }
end
```

checksum.crc16_dnp3

Returns the CRC16 checksum according to the DNP3 specification of the given data string as an integer.

checksum.crc16_dnp3(*String*)

- **String:** the data as a string

Example

```
function out()
    -- the following code checks the content of the entire message except
    -- for the last two byte (which are the checksum itself)
    cks = checksum.crc16_dnp3(telegrams.this():string():sub(1,-3))
    box.text{ caption="Checksum", cks }
end
```

checksum.lrc

A checksum mechanism based on a Longitudinal Redundancy Checking as used in Modbus ASCII transmissions. The result is a single byte (8 bit value).

checksum.lrc(*String*)

- **String:** the data as a string

Example

```
function out()
    -- in Modbus ASCII each byte is sent as a two ASCII characters but
    -- the checksum is calculated before encoding the message. So we
    -- must decode it first with base16.decode
    local bindata = base16.decode( telegrams.this():string():sub(2,-3) )
    cks = checksum.lrc( bindata )
    box.text{ caption="Checksum", cks }
end
```

checksum.crc16_modbus

Returns the Modbus RTU checksum of the given data string as an integer.

checksum.crc16_modbus(*String*)

- **String:** the data as a string

Example

```
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      cks = checksum.crc16_modbus(telegrams.this():string():sub(1,-3))
5      box.text{ caption="Checksum", cks }
6  end
```

The datetime module

The `datetime` provides you with a function to display the date and time of each telegram in a very user special format.

datetime module functions	
datetime.date Format specifier	Displays the date and time of the telegram according to the given format specifiers. The following format specifiers are defined (each one starting with the '%' character):
%a	abbreviated weekday name (e.g. Wed)
%A	full weekday name (e.g. Wednesday)
%b	abbreviated month name (e.g. Sep)
%B	full month name (e.g. September)
%c	date and time (e.g. 18/04/13 11:05:22)
%d	day of the month (01-31)
%H	24-hour clock hour (00-23)
%I	12-hour clock (01-12)
%M	minute (00-59)
%m	month (01-12)
%p	either 'am' or 'pm'
%S	second (00-61, considers leap second)
%w	weekday (0-6 = Sunday-Saturday)
%x	date (e.g. 18/04/13)
%X	time (e.g. 11:05:22)
%Y	full year (e.g. 2013)
%y	two digit year (00-99)
%%	the character '%'

The date function based on the Lua os.time function and uses the same format specifiers.

datetime.date

Returns the date and time of the given (telegram) time in a user specified format.

datetime.date(*format, time*)

- **format:** a string specifing the format.
- **time:** the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds.

Example

```
1  function out()
2      local tg = telegrams.this()
3      box.text{ caption="Date",
4                text=datetime.date("%X %x",record.starttime() + tg:time())}
5      -- returns something like 08:50:44 16.04.2013
6  end
```

The debug module

The Protocol View contains a built in debug window which you can use to show special information about the state of your script or the results of certain operations. The debug module covers all functions to output any text or value. You can also suspend, resume or summarize the output in case of repeating messages. To open the output window for debug messages, just press Alt + D.

Debug window
with Alt + D

Function	Description
debug.print	Outputs the given arguments in the debug window. You can pass as many arguments as you want. Each argument (text or value) must separated with a comma.
debug.resume	resumes a former suspended output.
debug.summarize	if activated the debug output collects all identical messages and show it only once with the repeating number.
debug.suspend	stops (suspends) the debug output. All further debug.print calls will be suppressed.
debug.timeprompt	put the current time (hh:mm:ss) in front of each debug output. You can enable or disable it by passing true or false to the function.

debug.print

Output the given, comma separated, arguments in the debug window.

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

- **param:** comma separated list of parameters.

Example

```
1  function out()
2      local tg = telegrams.this()
3      if tg:size() > 10 then
4          -- output the time and size of the current telegram
5          debug.print( "Time:"..tg:time(), "Size:"..tg:size() )
6      end
7  end
```

Avoid heavy usage of the debug.print
Every output via debug.print takes some time and will slow down the execution of your template script a little bit.

debug.resume

Continues the previously suspended debug output. Here we stop all debugging after receiving a telegram with no hex 10 as a first byte and resume the debugging when another telegram starts with hex 10 again.

debug.resume()

Example

```
1  function out()
2      local tg = telegrams.this()
3      if tg:data(1) == 0x10 then
4          -- first output the debug message
5          debug.print( "Data:"..tg:data(1),"Size:"..tg:size() )
6          -- then suppress any other output
7          debug.resume()
8      else
9          -- enable the debug output again
10         debug.suspense()
11     end
12 end
```

debug.summarize

Collects all identical debug messages and output them when the first different one occurs. The repeated messages are shown like this:

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

`n` means the number of repetitions.
Usually you put a statement like `debug.summarize(true)` at the beginning of your script, that is outside of the split() or out() function because there isn't any need to execute the command more than one a time. (See line 1 in the example below).

debug.summarize()

Example

```
1  debug.summarize( true )
2  debug.timeprompt( true )
3
4  function split( data, intval )
5      if intval > protocol.bitpause( 33 ) then return STARTED end
6      return MODIFIED
7  end
8
9  function out()
10     -- your output code...
11 end
```

debug.suspend

Suppress all debug output via `debug.print` till another call of `debug.resume` is executed. See the resume example above for usage.

debug.timeprompt

Enable or disable an additional prefix with the current time for every debug message when the output is done. The default is an output without any prefix. If activated, each output is headed by the current time in the format hh:mm:ss. For instance:

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

See line 2 in the example above.

The event module

The `event` module provides you with additional data event information which are not passed to the `split` function as parameter.
Please note! The event module is only accessible in between the `split` function body and cannot used in `out()`.

Function	Description
event.dir	returns the source or direction of the current data.
event.isbreak	returns true if the current byte is a break.
event.level	returns the current signal level of the given line when the data event occurred.
event.time	returns the time stamp of data event in seconds since the start of the record.

event.dir

Returns the direction or source of the current data event as an integer value with the following result: 1: Port A (CH1), 2: Port B (CH2).

event.dir()

Example

```
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

Returns true if the current byte is a break. A break is also received as a NULL byte. With `event.isbreak()` you are able to distinguish between a normal NULL byte and a real break. This function comes in handy i.e. when your protocol specifies a break as a delimiter.

event.isbreak()

Example

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

event.level

Returns the current signal level of the given line when the data event occurred. The line is passed as a number from 1...8 according to the display in the control program. Possible results are: 1: high level, -1: low level, 0: invalid/inactive.

event.level(*signal=NUMBER*)

- **signal:** signal or line number (1...8)

Example

```
1  function split( data, intval, alter, str )
2      -- the RI signal marks a special one byte broadcast telegram
3      if event.level(8) == 1 then return STARTED+COMPLETED end
4      if #str == 1 then return STARTED end
5      if data == eos then return COMPLETED end
6      return MODIFIED
7  end
```

event.time

Returns the time stamp of the current data event in seconds since start of the record. The result is a floating point number with microsecond resolution.

event.time()

Example

```
function split( data, intval, alter, str )
    -- remove all telegrams in the first 5s of the record
    if event.time() < 5.0 then return REMOVED end
    if #str == 1 then return STARTED end
    if data == eos then return COMPLETED end
    return MODIFIED
end
```

The linestates module

By its design the `split` function doesn't 'see' events except for transmitted data. If you need to know if a certain line like RTS or CTS has changed to initiate a new telegram frame (e.g. to enable carrier modulation like some Radio RTU producer do), you not only have to query the current line states when the data byte arrives. You also have to check, if the specified line has changed before. The `linestates` module comes to fill this gap.
Please note! The linestates module is only accessible in between the `split` function body and cannot used in `out()`.

Function	Description
linestates.changed(signo)	returns true if the given line (signo 1...8) has changed since the last call.
linestates.count(signo)	returns the number of changes of the given line signo (1...8).

linestates.changed

Returns true if the given signal number (line) has changed since the last call. The signal number is counted from 1 to 8 and meets the sequence as shown in the control program display.
A signal alternation is always detected when a signal changes its tri-state level. This includes not only changes from high to low but also changes from valid to invalid and visa versa.

linestates.changed(*signo*)

- **signo** line (signal) number.

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

Returns the number of line changes of the given line or signal number since start of the record. The signal number is counted from 1 to 8 and meets the sequence as shown in the control program display.
A signal alternation is always detected when a signal changes its tri-state level. This includes not only changes from high to low but also changes from valid to invalid and visa versa.

linestates.count(*signo*)

- **signo** line (signal) number.

Example

```
1  rtsChanges = 0
2  function split( data, intval, alter, str )
3      local RTS = 6
4       if linestates.count( RTS ) > 5 then
5           rtsChanges = 0
6           return STARTED
7      end
8      return MODIFIED
9  end
```

The protocol module

Every time you need information about the currently used protocol, for instance the baud rate, the number of data bits, the parity settings, the protocol module will come in handy.

Function	Description
protocol.baudrate	Returns the baud rate used in the current recording.
protocol.bitpause(bits)	This function returns the time which is needed to send the given number of bits. Profibus for instance uses a pause of 33 bits as a telegram delimiter.
protocol.bytepause(bytes)	This function returns the time which is needed to send the given number of bytes. Modbus RTU for instance uses a byte pause of 3.5 byte as a telegram delimiter.
protocol.databits	Queries the used number of data bits. The result is a value in the range 5...9.
protocol.parity	Returns the parity setting of the current recording as following: None = 0, Odd = 1, Even = 2, Mark = 3, Space = 4.

protocol.baudrate

Returns the baudrate as used in the current record.

protocol.baudrate()

Example

```
function split( data, intval, alter, str )
    -- start a new telegram after a pause of 33 bits
    if intval > 33 / protocol.baudrate() then
        return STARTED
    end
    return MODIFIED
end
```

protocol.bitpause

Returns the necessary time to send the number of the given bits.

protocol.bitpause(*bits*)

- **bits** number of paused bits.

Example

```
function split( data, intval, alter, str )
    -- Profibus specifies a pause of 33 bits as a telegram delimiter
    if intval > protocol.bitpause( 33 ) then
        return STARTED
    end
    return MODIFIED
end
```

protocol.bytepause

Returns the necessary time to send the number of the given bytes.

protocol.bytepause(*bytes*)

- **bytes** number of paused bytes.

Example

```
function split( data, intval, alter, str )
    -- Modbus RTU specifies a pause of 3.5 bytes as a telegram delimiter
    if intval > protocol.bytepause( 3.5 ) then
        return STARTED
    end
    return MODIFIED
end
```

protocol.databits

Returns the number of data bits (word length) as used in the current record.

protocol.databits()

Example

```
function output()
    local tg = telegrams.this()
    if protocol.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
```

protocol.parity

Please note! This function returns the specified parity settings of the record, not the parity bit of an individual byte.

protocol.parity()

Example

```
function out()
    if protocol.parity() ~= 2 then
        -- do something when parity is not even
        box.text{ caption="Warning", text="We need an even parity" }
        return
    end
end
```

The record module

The `record` let you query information about the record, for now only the record start time and the bus-wiring are accessable.

record module functions	
record.buswiring	returns the selected bus wiring. 0 : 2-Wire-Tap, 1 : 2-Wire-Segment, 2 : 4-Wire-Tap, 3 : 4-Wire-Segment
record.starttime	returns the start of the record in seconds since the Epoch (as used in the datetime module).

record.buswiring

Returns the current bus-wiring as set in the bus wiring dialog (only MSB-RS485) or as it was stored in a reloaded record.

record.buswiring()

Example

```
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
```

record.starttime

Returns the seconds since the Epoch (00:00:00 UTC, January 1, 1970).

record.starttime()

Example

```
1  function out()
2      local tg = telegrams.this()
3      local datetime=datetime.date("%X %x",record.starttime() + tg:time())}
4      -- returns something like 08:50:44 16.04.2013
5  end
```

The shared module

The protocol mechanism uses two Lua interpreter to split the incoming data stream into individual telegrams. One for every data direction.
Therefore you never have to worry about the right source of the passed `data` parameter. Also the internal representation of the already received bytes (given as `str`) always relates to one data source.
Using two independent Lua interpreters makes the operation of the template function much easier. But it has one pitfall:
Despite the fact, that you can create variables outside of the `split` function, you nevertheless cannot use them to share an information between the two Lua interpreters since they work totally independent of each other.
If you have a need to share data between the `split` function called by data source A and the second one called when a byte of source B arrives, than the `shared` module comes into play.
All variables put into the `shared` module are accessible from both interpreters. You can create such a variable for instance when a certain byte on port A arrives and query its content when handling data on port B.

Function	Description
shared.get	returns the content of the global variable with the given name.
shared.set	store the variable with the given name.

shared.get

Returns the global variable with the given name or nil if no variable with this name exists

shared.get(*name*)

- **name:** The name of the variable as a Lua string.

shared.set

Create a new variable with the given name and assign the value to it. If the variable already exists, its content will be overwritten.

shared.set(*name, value*)

- **name:** The name of the variable as a Lua string.
- **value:** Any Lua value (number, boolean, string).

The following code uses a imaginary protocol. Every telegram starts with a colon ':' and ends with LF. Consider a special telegram used as a 'life ping'. In our example we like to hide every 'life ping' AND the relating response. To make the whole matter a little bit more complicated, the response to a 'life ping' has to be the same as to other requests.
A 'life ping' is specified as an empty telegram, which means a colon ':' followed by a LF.
To distinguish a 'life ping' response from other identical responses we have to memorize a 'life ping' telegram, for instance received on port A.
In case of a later response we than check the memorized state to show or hide the according telegram. Here is the 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
```

The example above seems a little bit constructed, but it serves our purpose how to use the `shared` module to exchange information between the two Lua interpreters.
Line 2 just triggers the start of a new telegram (58 is the decimal ASCII value of the colon) . A received LF (decimal 10) marks the end of the telegram (line 3). We then have to check for a 'life ping' telegram, which means the shared variable `LifePing` was set to true (line 4). Returning REMOVED in this case hides the telegram from being displayed.
In line 7 we test for every 'life ping' telegram (length is 2 bytes) and set the global `LifePing` to true or false. In case of a 'life ping' the telegram has to be

REMOVED (line 9). Otherwise the telegram state is COMPLETED (line 13). All other data bytes are added to the internal telegram representation by returning MODIFIED.

Please note! The code above will not work by using a normal Lua global value instead of the `shared` module since every interpreter of the according data direction uses his 'own' global values. The `shared` module is the only possibility to shared data between both interpreters.

The string dump extension

This function let you 'hex dump' the content of any Lua string. The function works similar to the `telegram:dump`, but since it is not assigned to a certain telegram, it allows you to hex dump for instance also the results of a base16 conversion.

string.dump

Creates a string summarizing (hex dump) of the string data as 2-digit hex or 3-digit decimal values separated by a specific character. The default number base is hex (16) and the default separator is a space.

Please note! `string.dump` isn't part of the common Lua language and only works within the ProtocolView.

string.dump(*str, base, sep*)

- **str:** The Lua string you want to hex dump.
- **base:** The used number base, default is hex (base 16).
- **sep:** Replaces the default space separator with any character or string. An empty string suppresses the separator completely.

Example

```
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
```

The telegram type

The data type `telegram` maps the telegram properties of a very particular telegram in the record. It is always the result of accessing one of them via the `telegrams` module (notice the plural in the module name).
You can query every property by calling the telegram's related function in an object oriented manner. An additional dump method provides a quick overview of the whole telegram content.

Function	Description
data	Returns the data as a 9 bit value at the given position. You can address a certain data from the beginning with positive indexes (1 means the first data) or with negative indexes backwards (-1 accesses the last data).
datatime	Returns the timestamp of the given telegram data byte in seconds with microsecond precision. The indexing of the data bytes is the same as with function data above.
dir	Queries the direction or source of the telegram. Returns 1 when the telegram was received at Port 1, or 2 otherwise (Port 2).
dump	Returns a Lua string with a hexadecimal or decimal list of all or a given range of the telegram data. `dump` comes in handy when you need a quick insight of the telegram content or in case you just want to display a certain range of data.
duration	Provides the time length or duration of the telegram in seconds. This means the time between the first start bit and the last stop bit.
isbreak	Returns true, if the data byte at the given position is a break.
number	Queries the telegram number, the result is counting from 1.
size	Returns the data size or length of the telegram. Please note: The data may consist of 9-bit values which are also counted as one data byte.
string	Returns the telegram content as a Lua string. Since a Lua string cannot cover 9-bit values, possible existing 9-bit values are reduced to 8-bit.
time	Returns the time when the telegram was received in seconds with micro second precision. For instance: A value of 25.034198 means a telegram received 25.034198 seconds after starting the record.

telegram:data

Returns the data value at the indexed position of the telegram. Indexes starts from 1 as usual. Negative index values address the data from behind. Because the MSB-RS485 also supports 9 bit value, the return value is in the range of 0...511.

telegram:data(*INDEX)*

- **INDEX** index of the requested byte.

Example

```
function out()
    -- access the current telegram
    local tg = telegrams.this()
    -- shows the first byte in the telegram as decimal value
    box.text{ caption="First", text=tg:data( 1 ) }
    -- shows the last byte in the telegram as decimal value
    box.text{ caption="Last", text=tg:data( -1 ) }
end
```

telegram:datatime

Returns the data timestamp of the indexed telegram data byte. Indexes starts from 1 as usual. Negative index values address the data from behind.

telegram:datatime(*INDEX*)

- **INDEX** index of the requested byte.

Example

```
function out()
    -- access the current telegram
    local tg = telegrams.this()
    -- shows the pause between the stop bit of the first byte and the
    -- start bit of the second byte (substract sending time)
    local delay = tg:datatime( 2 ) - tg:datatime( 1 ) - protocol.bytepause( 1 )
    box.text{ caption="Pause 1-2", delay }
end
```

telegram:dir

Queries the telegram direction or source. A value of 1 means the telegram was received at Port 1, a value of 2 marks a telegram from Port 2.

telegram:dir

Example

```
function out()
    -- access the current telegram
    local tg = telegrams.this()

    if tg:dir() == 1 then
        -- do something with data form port A
    else
        -- telegram received at port B
    end
end
```

telegram:dump

Creates a string summarizing (hex dump) of a given range of telegram data as 3-digit hex or decimal values, separated by a specific character. Without any argument, the whole telegram content is used. The default number base is hex (16) and the default separator is a space.

telegram:dump{ *first=1, last=-1, base=16, width=3, sep=' ', max=LEN* }

- **first:** Specifies the first data used in the hex dump, default is the first data in the telegram (1).
- **last:** Specifies the last data used in the hex dump, default is the final data of the telegram (-1 or `telegram:size()`).
- **base:** The used number base, default is hex (base 16).
- **width:** The number of digits used for the data output, default is 3 digits (to support also 9 bit values). In most case you will pass `width=2` when using the hexadecimal notation.
- **sep:** Replaces the default space separator with any character or sting. An empty string suppresses the separator completely.
- **max:** Limits the maximum count of data in the hex dump. A given values of `max=4` outputs only the first two and last two data values and displays the remaining data as a quantum value. The default value is equal to the telegram length (LEN).

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

Returns the telegrams time length or duration in seconds. The duration time is defined as the difference between the start bit of the first and the stop bit of the last transmitted byte. The result is a double precision floating point number with the usual resolution of one micro second.

telegram:duration()

Example

```
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:isbreak

Returns true, if the data (null) byte at the indexed position of the telegram is a break.

telegram:isbreak(*INDEX)*

- **INDEX** index of the requested byte.

Example

```
function out()
    -- access the current telegram
    local tg = telegrams.this()
    -- distinguish between a normal null byte and a break
    if tg:data( 1 ) == 0 then
        if tg:isbreak( 1 ) then
            box.text{ caption="BREAK", text=tg:data( 1 ) }
        else
            box.text{ caption="NULL", text=tg:data( 1 ) }
        end
    end
end
```

telegram:number

Queries the number of the telegram. The telegram numbers are counted from 1 (the very first received telegram).

telegram:number()

Example

```
function out()
    -- access the current telegram
    local tg = telegrams.this()
    -- show the current telegram number
    box.text{ caption="Number", text=tg:number() }
end
```

telegram:size

Queries the size of the telegram. Please note that also a 9 bit value in a telegram is counted as one item.

telegram:size()

Example

```
function out()
    -- access the current telegram
    local tg = telegrams.this()
    -- show the size of a telegram
    box.text{ caption="Length", text=tg:size() }
end
```

telegram:string

Returns the complete telegram data as a Lua string.
A Lua string can contain any byte in the range 0...255 but only 8 bit values. If

the telegram consists of 9 bit values, the ninth bit will be discard.
In contrary to the earlier `tg` and `tgprev` modules, `telegram.string()` simply returns the whole telegram data as a Lua string without accepting any substring defining parameters.
Since it's easier to leave the relating substring functionality to the Lua string module, there isn't any reason to implement it again. And: Since Lua allows the indexing of substrings from the end, it has additional advantages.

telegram:string()

Example

```
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

Returns the time stamp of the telegram which is the measured time of the first received byte in seconds since starting the record. The result is a double precision floating point number with the usual resolution of one micro second.

telegram:time()

Example

```
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
7          tp = tc
8      end
9      box.text{ caption="Response time", text=tc:time() - tp:time() }
10 end
```

The telegrams module

The `telegrams` module provides you with an easy method to access any telegram recorded up to the current time. The big advance: In contrary to the obsolete `tg` and `tgprev` modules the access isn't only limited to current and former telegram. By using the `telegrams` module your are now able to handle the active telegram in the `out()` function depending on the data/state of any telegram occurring before[6].

[6] Up to now the out() function could only handle the current and previous telegrams by using the modules tg and tgprev.

For instance: You have to treat a telegram in a different way when one of the former telegrams was of a certain type. Since it often isn't just the former or penultimate telegram, you need a way to 'iterate' through the prior telegrams and looking for the according telegram.

Quering any recorded telegram is simply done by calling the module function `telegrams.at(index)` whereas index refers to the telegram you want to access.
The value (or object) returned by the function `telegrams.at` is always of type `telegram` (see 13.7). This type acts as an interface and provides the same functions as you are accustomed from the former `tg` or `tgprev` modules.
The `telegrams.at(index)` function is the only one you ever need. But since the access to the current and previous telegram is the most widely-used operation, the module offers two alias functions for these. The following table lists all module functions:

Function	Description
telegrams.at	returns the telegram at the given index/position.
telegrams.this	returns the current telegram handled by the `out` function. It is an alias for `telegrams.at(-1)`.
telegrams.prev	returns the previous telegram handled by the `out` function. The same like `telegrams.at(-2)`.

telegrams.at

The `telegrams.at(index)` function accepts absolute and relative indexes and returns the relating telegram - or nil if you pass an invalid index.
The effort is always linear and it makes no difference to query the actual or the very first telegram of the record.
Absolute addresses starts with an index of 1 (first telegram) up to the current telegram number. A relative address can be -1 (the last or current telegram as used in the obsolete `tg` module) or any other negative value. An index of -2 returns the previous telegram (and makes the `tgprev` obsolete), an index of -3 accesses the telegram before the previous one and so on.
Since the `out()` function always handles ONE telegram (one telegram line in the telegram window) every call, a relative indexing is more convenient because you don't have to worry about the right 'absolute' index number.

telegrams:at(*index*)

- **index:** The index of the requested telegram. A positive index counts from the beginning of the record, a negative counts backwards from the current handled telegram in `out()`.

Examples

```
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
```

The next piece of code calculates the time distance of the current telegram (index -1) relating to the previous one (index -2). Instead of storing the returned `telegram` value as a local variable, we use them directly to access the wanted information. Here is the code:

```
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
```

13.8 Settings

The settings dialog provides you with several options, i.e. a list of predefined telegram prefixes (number, date/time, etc.), an individual telegram font, another background color for the telegram window and a Lua compatibility switch.
Click on `Settings→Configure Protocol monitor...` to open the settings dialog.

Show additional telegram information

Although you can put any desirable information in front of a telegram by yourself it's easier to simply enable or disable the wanted facts just by some clicks.
The prefix settings dialog let you select one or more of the following information which then are displayed in front of every telegram.

1. **Telegram number**
   This is the actual number of the telegram count from 1 and independent of the telegram source.
2. **Telegram time**
   The time stamp of the first byte of the telegram relative to the record start in seconds.
3. **Telegram date and time**
   The absolute time and date of the telegram occurrence. Time and date are shown in your local time format (depending on your PC settings) with an additional microsecond part.
4. **Telegram duration**
   This is the length of the telegram in seconds (with microsecond resolution), measured from the first start bit to the last stop bit.
5. **Time distance to the former telegram**
   The 'pause time' between the last and current telegram. It is the time between the last stop bit bit of the former telegram and the first start bit of the current telegram.

Every selection acts directly on the telegram display.

Telegram font

Change the font

Altering the font effects the display of the boxes. You can choose a smaller font when you want to see more data in a line or a bigger one for a more comfortable viewing. Open the settings dialog and click the font icon in the head line.
The font dialog offers you to select an individual typeface, font style and size. All changes are applied immediately to the telegram window and are stored automatically.

Change font via mouse and keyboard

There exists a more directly way to adapt the font size to your preference without using the settings dialog. Press the Ctrl key and scroll the mouse wheel. Or just hit the Ctrl+ + or Ctrl+ − to increase or decrease the font size. Ctrl+ 0 switches back to the default size.

Background color

Set an individual background

The template script let you only influence the color settings of the telegram via the box model. When you like to adapt the background of the whole telegram window, click the color button and select a color of your choice. This color becomes the new background.

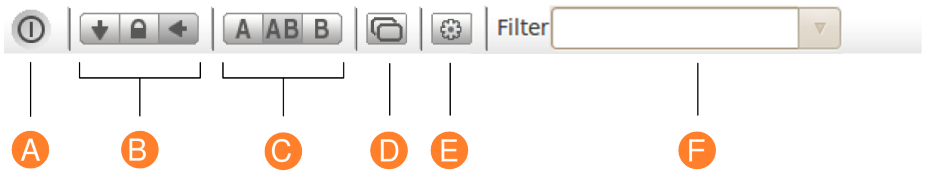
Lua compatibility

Lua compatibility

To provide the best possible protocol handling it's sometimes inevitable to change Lua functions and names at the expense of downward compatibility. We don't do this flippantly and we only break with former versions when the benefits are outstanding. In such a case we will give you the chance to adapt your own templates as painless as possible.
The Lua interpreter accepts obsolete functions by default for a while. When you are willing to update your templates, just disable the compatibility switch in this dialog. The ProtocolView then points you to all lines in your code that it cannot accept any longer. See section 13.11 for a detail overview about the now obsolete functions and modules.

13.9 The Toolbar

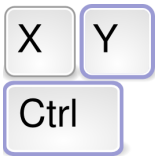
The toolbar serves for a fast access to the most used functions. Some are identical in all monitor windows, some others are only specific for the protocol monitor.



A	**End:** Saves all settings and closes the window.
B	**Display mode:** According to the mode the window either shows always the current (last recorded) event or locked or actualizes its content synchronous to the other windows.

C	**Data direction:** The protocol monitor can display both data directions (Data channel A and Data channel B) combined or separately to display them in different windows.
D	**New view:** Opens a new window with the same sector and settings.
E	**Apply template:** Applies the current template on the available data.
F	**Filter control:** Select and pass any text to the split filter parameter.

13.10 Short commands



Short commands for the most important functions

Action	Short command
Online help for the Protocol view	F1
Apply template	F5
Open the template manager	Ctrl + M
Show telegrams in a new window	Ctrl + N
Save current template	Ctrl + S
Load template from file and apply	Ctrl + O
Open and close template editor	Ctrl + T
Select all protocol or editor lines	Ctrl + A
Reverse selection	Shift + Ctrl + A
Export selected lines	Ctrl + E
Search text in the editor	Ctrl + F
Search and replace text in the editor	Ctrl + H
Fold/unfold functions in the editor	Ctrl + L
Increase the current telegram font (zoom in)	Ctrl + +
Decrease the current telegram font (zoom out)	Ctrl + −
Switch back to the default telegram font size	Ctrl + 0
Open the colour chooser dialog	Ctrl + Alt + C
Open the debug output window	Ctrl + Alt + O
Save settings and close protocol view	Ctrl + Q

13.11 Obsolete functions and modules

With the beta-release 4.1.9 several functions and modules become obsolete and they will be removed finally in the next release 4.2.0. This section will be a guidance how to update your templates by replacing obsolete code with the

more powerful new functions and modules[7].
In the beginning the list of obsolete modules:

- `tg` - Access the current telegram in function `out`
- `tgprev` - Access the previous last telegram in function `out`
- `hex` - Provides a hex ascii to binary/numbers conversions from the current telegram
- `box.hexdata` - Display part of the current telegram data as hex dump

You may ask what's wrong with them?
The weakness in the design is the mixup of pure output or conversion modules (box and hex module) with an fixed telegram access, emphasized as current.
The `hex` module as well as the `box.hexdata` can ONLY process the current telegram. You cannot hex dump a previous telegram and you won't able to convert hex ascii coded data from other telegrams except for the current one. In both cases you have to write your own Lua code to achieve a similar functionality when not handling the current telegram.
Furthermore: `tg` and `tgprev` limit the processing in the `out` function to the current and previous telegram. As soon as you have to examine more preceding telegrams you are lost.

The new `telegrams` module put an end to this limitation and provides you with a random access to all telegrams currently received while executing the `out` function. As a logical step the successor of the `box.hexdata` and `hex` module throw off the `tg` dependency and they are now also capable to handle arbitrary telegrams.

In the following we will explain how to update the display code of the Modbus ASCII telegram 'Write Single Register'. The telegram consists of the parts:

:	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

Except for the starting colon ':' and the ending mark CRLF all telegram data are transmitted in hexadecimal 0-9, A-F (hex ASCII coded). Here an example byte sequence as shown in the DataView:

3A	30	31	30	36	30	30	31	39	30	33	33	45	39	46	0D	0A

A former solution using the obsolete modules would look like:

```
1 box.text{caption="Start",text=string.char(tg.data(1))}
2 box.text{caption="Addr",text=hex.byte{ pos=2 }}
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}
```

Our first objective is to replace the `tg` module and to convert the hex ascii characters (the green and yellow sections) into their binary representation.

[7] The protocol templates and examples are already adapted and may serve as a first instance.

```
1  local tele = telegrams.this()
2  local bindata = base16.decode(tele:string():sub(2,-3))
```

Line 1 queries the current telegram and assign it to the variable `tele` which now contains the same information as when accessing the `tg` module. But in contrary to `tg` the variable could also refer to any other telegram.
In line 2 we pass the section (substring) of the green and yellow bytes starting with the second byte (index 2) and ending with the third last (index -3) to the `base16` decode function. The result is a binary sequence of the green and yellow bytes.
Please note! You cannot convert the whole telegram into binary because the colon start byte as well as the CRLF are no valid hex ascii characters!

With the binary data at hand there isn't any further need to convert the different parts of the telegram like address, function, register, value or LRC checksum from hex ascii. The function `bunpack` in line 3 provides a much easier way to extract the information in one step.

```
1  local tele = telegrams.this()
2  local bindata = base16.decode(tele:string():sub(2,-3))
3  local pos,adr,fnc,reg,val,lrc = bunpack(bindata,"bb>H>Hb")
```

`bunpack` is instructed to 'unpack' the given sequence or string according to the passed format string `"bb>H>Hb"`. The translated meaning is:

1. return ALWAYS the end of the parsing (pos)
2. return the first character as byte (b) (adr)
3. return the second character as byte (b) (fnc)
4. return the 3th and 4th byte as an unsigned 16 bit value (H) with most significant byte first (>) (reg)
5. return the 5th and 6th byte as an unsigned 16 bit value (H) with most significant byte first (>) (val)
6. return the 7th byte as byte (b) (lrc)

At least six results are returned. The first one is always the position where the unpacking stopped. This is equate to the position where you perhaps want to continue with another bunpack call.
Line 3 simply collects the results in the according variables and we can display them without any further processing.

```
1  local tele = telegrams.this()
2  local bindata = base16.decode(tele:string():sub(2,-3))
3  local pos,adr,fnc,reg,val,lrc = bunpack(bindata,"bb>H>Hb")
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) }
```

The remaining parts are the start ':' character and the CRLF end sequence. The colon is the first byte in the original telegram `tele` and we can handle it similar to the obsolete coding. See line 4 in the listing below.
The new `telegram:dump` function replaces the restricted `box.hexdata` in

line 10. `dump` belongs always to a prior assigned telegram and doesn't access the `tg` internally.

```
1  local tele = telegrams.this()
2  local bindata = base16.decode(tele:string():sub(2,-3))
3  local pos,adr,fnc,reg,val,lrc = bunpack(bindata,"bb>H>Hb")
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 } }
```

13.12 Lua References

Lua is free available and well documented. You will find a lot or sources, examples and documentations in the world wide web. A good (if not to say the best one) is the Lua website at:

http://www.lua.org

A direct link to the original Lua manual for version 5.1 (including the C API) is here:

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

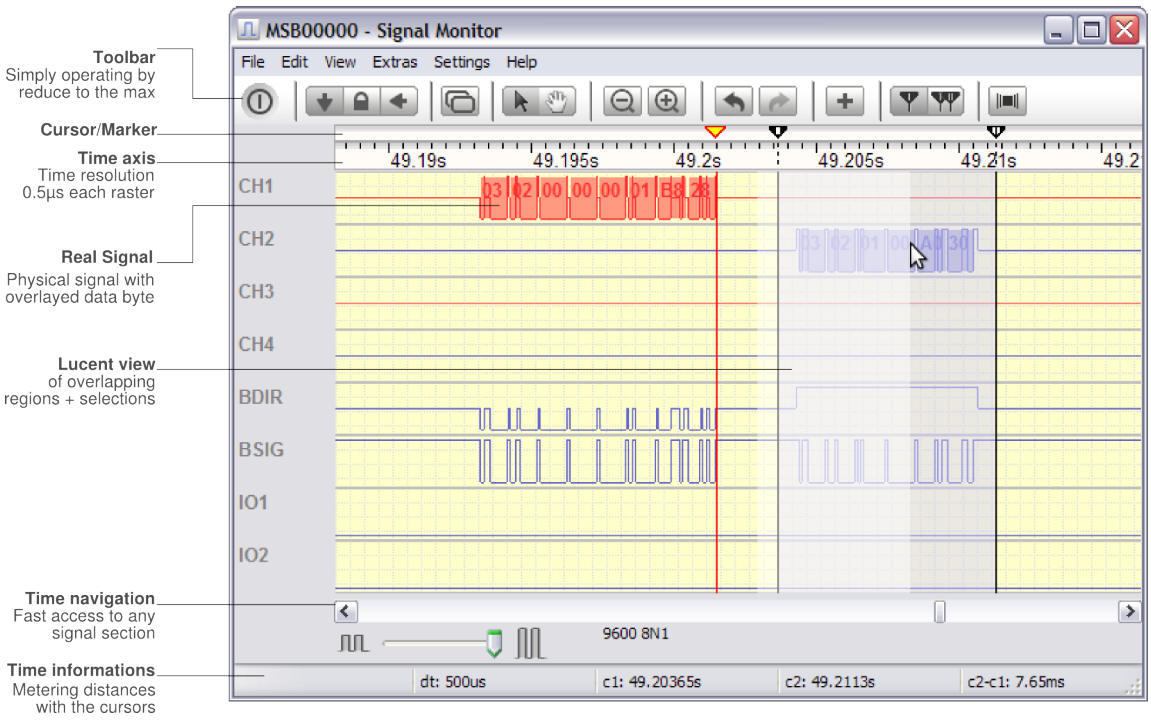
14
The Signal View

The MSB-RS485 Analyzer samples all signals with up to 16 MHz. The result displays the signal monitor. Analogous to a digital scope you can select any sectorand examine in different magnification levels.

For analyzing of serial data streams it is sometimes not sufficient to watch the transmitted data bytes.
Especially bus systems need a smooth interaction of all components. This requires a correct parameterization and strict observance of the protocol specifications by all bus participants. The possible error reasons are manifold.

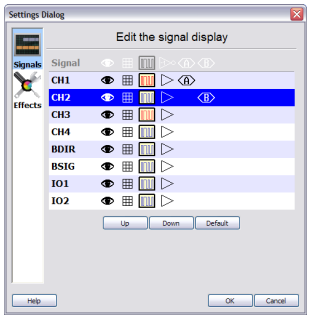
By wrong settings or a malfunctioning hardware invalid data can be set onto the bus.
Occurring data collisions by multiple simultaneously active senders (Bus blocking /unblocking) or invalid data sequences (frames), caused by wrong timing, can not be judged only by recording the data. The same applies for the hardware handshakes.

All by the MSB-RS485 recorded lines are displayed in parallel, whereby each line signal can be individually switched on and off and the sequence of their display can be varied. The function of the signal client is that of a 8 channel digital scope. In the opposition to a scope the recording depth (Record depth) and the duration of the recording is limited only by the disc capacity and computing power of your PC.
By opening of multiple signal clients you can check the recorded signals at different places with different time resolution. Aside that the signal client is also well suited to judge the response time of sent data bytes. In the easiest case the signal client shows level changes of an active data connection and provides important hints for the function or malfunction of the connection.



14.1 Signal representation

The signal display is divided into 3 sectors. Directly below the toolbar the cursor bar is located (look Cursor) and the timeline. The timeline provides you with the exact position and resolution of the visible signal section. To ease the the readout all times displays are shortened by removing unnecessary prefixes. So 0.012570s changes to 12.57ms.
Below the timeline the signals are displayed. For all signals the same sector and the same resolution applies (time base). To examine a signal at different positions simply start a new signal client. You can duplicate the actual client by pressing the 'Clone' button in the toolbar. By this a new signal client will be started which has exactly the same settings like the actual one. Or you start a signal client with default settings from the control program.



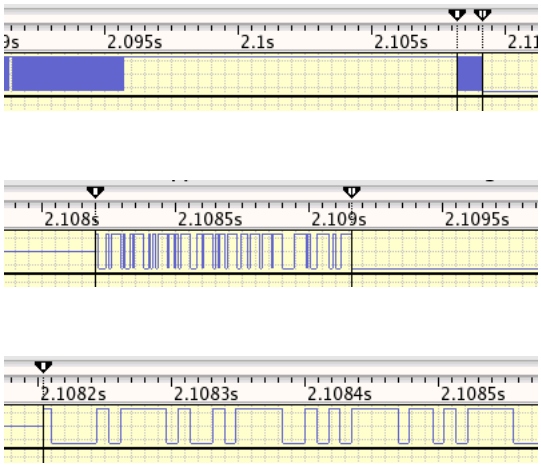
Compare different signal sections

Different signal regions can be examined through multiple signal clients. Just open as many as you wanted.

Each visible signal is described by its name at the left border. The signal name is set before the recording in the preference dialog of the control program. The sequence in which the signals are displayed can be set in each signal client individually. That makes it possible to arrange important signals directly on top of each other. Unimportant signals can be fade out. All signal specific settings are done in the Settings dialog, which is opened from the `Options→Preferences` menu.

Individual signal settings
Signal sequence, visibility, resolution and color are set in the signal dialog!

The visible signal region is defined by its position as time difference from the beginning of the recording and its visible sector. The visible sector is derived from the size of the window counted in screen pixel and the time base, that means how many microseconds are displayed per Pixel. The greater the time base the bigger is the time window of the signal sector.
To get a complete overview over the recorded events select 'View total' from the 'View' menu or press Ctrl+Pos1. The Timebase is automatically choosen so that all events can be seen at the same time. Depending on the selected time base multiple events could have occurred in one screen pixel. The signal client draws a vertical line instead of a single pixel to mark this time. The following image shall make this behaviour clear:



Timebase 0.5ms

Timebase 50μs

Timebase 10μs

All three pictures apply to the same signal, but displayed with decreasing time resolution.

14.2 Navigation

The scroll bar below the signal windows represents an overview over the position and size of the displayed sector in comparison to the complete signal. The slider size of the scroll bar represents the size, the slider position the offset of the displayed sector.
Beside that the scroll bar allows to navigate through the complete signal. The arrows at the left and right border scroll the signal sector in grid or in 10 grid steps. Or the sector can be moved with the slider. Also the signal can be moved with the arrow keys, look Shortcuts.
Position and zooming (Time base) are displayed in the two left status boxes. Below the scroll bar you see a slider, with which you can vary the signal hight of all signals. Normally you will not need this function. It is useful if you can not read the displayed name of a Region when it is hidden by the signal. In this case simply decrease the signal hight.

Navigation by mouse wheel

The navigation by mouse wheel offers a way to shift the signal. Keep the Ctrl key pressed while turning the wheel. Depending on the turning direction the signal is shifted to the right or to the left in 10 grid steps. If the ctrl key is not pressed the signal is zoomed or unzoomed.

Shift with the hand cursor

The hand cursor allows the pixelwise shifting of the signals. click on the hand symbol in tool bar. The cursor changes to a hand symbol. To move the signal to the right or to the left simply grip the signal by pressing the left mouse key and drag the signal in the desired direction. Keep the mouse key pressed. while gripping the signal the cursor has the look of a griping hand.

14.3 The time base

The time base correspondends to the magnifying for the represented signal. The smallest time base is 10μs, that means 10 microsecs per raster and means 1μs per displayed pixel. One raster grid is 10 pixel wide. The signal is magnified, for a screen resolution of 1024 it is about 1 millisecond (if you have maximized the signal monitor window).
If the level changes are in the milisec or second range you will chosse a higher timebase to watch a larger section of the signal. By clicking of the two magnifying glass symbols in the toolbar the time base is set to the next higher or lower value. The same can be done by using the key combination Ctrl+Up Arrow and Ctrl+Down Arrow.
You also can magnify a certain sector of the signal by selecting the sector with a pressed left mouse key. Move the mouse cursor to the beginning of the sector and press the left mouse key. Hold the key pressed and move the cursor to the end of the section. A rectangle marks the current selection. As soon as you release the mouse key the section is displayed magnified.

14.4 Undo and Redo

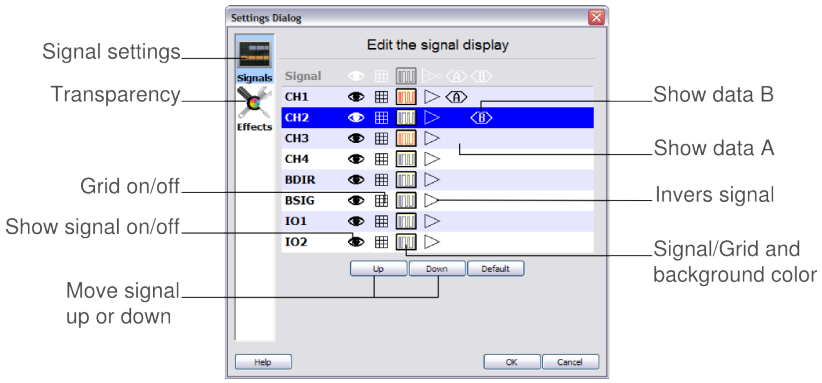
All magnifications can be taken back (undo) or redo after an undo.
By that you can magnify an interesting signal section, for example to place a cursor exactly, and go back to the original view, simply ba clicking on the undo symbol in the toolbar or entering Ctrl+Z.
The original view before an undo is recalled by redo. Both symbols are marked as inactive if no further undo/redo steps are possible. Undo and redo are used only for section magnifying. A normal increasing/decreasing of the signal is possible at every time so that no undo redo is necessary.

14.5 Settings dialog

In the setup dialog you can adapt the signal display to your needs in broad range. This applies to the number of displayed signals as well as to the sequence and color. Aside that the performance of the display can be influenced.
From Version 2.1.1 you can additionally fade the recorded data bytes (receive and send data) in into any signal to compare them with the physical signals.

The signal dialog

In the signal dialog all available signals are listed and their actual settings are displayed. The signal with blue background is the actual selected one. First you have to select a signal with your mouse then you can change its attributes.



Visibility and grid can be swtiched on or off by clicking the respective symbol. To change the colors of a signal click the signal symbol on the right side. In the opening color dialog you can select signal color, background color and grid color from 72 colors.
Choose your favored color and clock on the corresponding button for paper, signal and grid to actualize the color. The changes are directly actualized in the signal client so that you can see immediately the changes you have made.
If the changes are ok accept with 'Ok' while 'cancel' restore the old settings.

Choose your favorite colors

Save your color settings

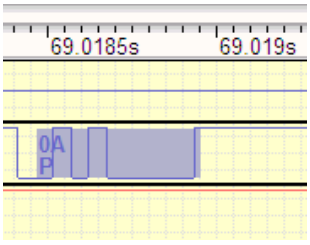
Don't worry about your color settings. All properties will be saved automatically after close the dialog and will be present also for future sessions.

Signal inverting

Each signal can individually be inverted. By default the representation is as recorded by the analyzer, symbolized by a buffer symbol. Click with your mouse onto this symbol to invert the display. The signal is now mirrored horizontally. In the table now an inverted driver symbol appears.
Please note, that this inverting applies to the display only.

Signal sequence

The Signal sequence is freely definable. At first select the signal whose position you want to change. Then move the signal with the keys 'Up' and 'Down' of the signal dialog. The setting directly apply to the signal client so that you can check the settings immediately.
The key 'Defauls' cancels your changes and resets the colors to default values.



Fade in data byte (Data channel B) with Parity error

Fade in the transfered data

Independent of the recording of the physical signals you can fade in the transfered data bytes into any signal. You only have to activate the recording of

RxData and TxData in the control program. The display is done as a data block with the hex value of the data byte.

Grafical effects

The display of the selected range and fade in region was made as background color. This had some disadvantages.

Depending on the selected display colors the selected range or region was hardly to see. Additionally the start and end of overlapping region could not be seen.
With introduction of transparent ranges for selection and regions the display is now independent of the colors for signal, grid and background. Selection and regions now appear transparent and allow a more user friendly inspection and marking of signals.
However, transparent display has a disadvantage. They are not for free and cost additional performance. That should not be an issue on modern PCs and is limited to the display of regions only. If the performance degree is too obvious you can switch the transparent display of in the setup dialog (effects), separated for selection and regions. You also can adjust the transmittance of the tranparent ranges.

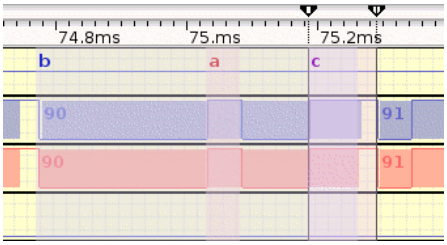


Abbildung 14.1: Overlapped regions, (a) is on the bottom, followed by (b) and (c) on the top. See also the section about regions without any transparency.

14.6 Cursor operating

Every signal client owns 2 Cursors I and II which can be moved arbitrarily over the signal inside the visible range. To move a cursor click on the respective cursor symbol, an upside down triangle above the timeline, and draw the cursor line to the wanted position. If both cursors are on the same position you can see only the last activated one because it overlays the other cursor. But in this case always Cursor I will be activated. To move the second cursor keep the SHIFT button pressed while clicking the cursors. Now you can move cursor II while Cursor I stays at its place.

> **Cursor selection**
> With pressed SHIFT key the second cursor is activated if both cursors are at the same postion!

Placed cursors keep their signal specific position even if you choose another signal view. Cursor outside the visible signal window are displayed at the left or right border. Their actual position can be read in the status line. c1 means cursor I and c2 cursor II.
In addition to the position of each cursor their time difference is fade in in the status line. So a time difference measurement is easily posible, e.g. the

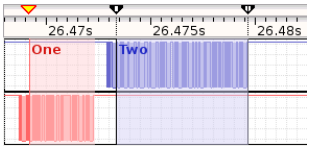
duration of an active line.
To compare multiple sectors you can assign the marked signal sectors to a region. Click the 'Add Region' Button in the toolbar. A maximum of 8 regions can be defined. The range between both cursors gets colored. Read more about regions in chapter Regions.
You can also move both cursors at the same time, for instance to compare the duration of two signal changes which did not occur at the same time. Both cursors are connected by pressing 'c1+c2' in the toolbar.
As long as this key is activated both cursers are moved simultanously, all the same which one you move.

Signal selection

The sector between both cursors represent the current selection at any time. You do not have to make another selection. Since both cursors do not change their position in relation to the signal, the position and distance beween them stays the same, even if the signal view is changed. Both are displayed in the status line.
All operations, which are related to the current selection, always concern the signal range between both cursors.
To define a region click on the '+' Symbol in the toolbar or press Key F4.

Regions

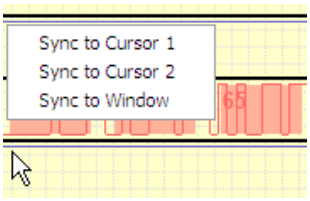
Analogous to the other monitors regions have a colored background.
The picture shows two regions, where the light blue one is framed by the two cursors and assumedly selected by them. Because regions are superordinated and valid for all analysis windows, selected signal sectors can be marked and examined with different tools at the same time.
The red-yellow triangle in the cursor bar is the current Synchronizing Event, received from another analysis window.
You can find more information at P.155.



Signal regions
marks interested sections

14.7 Synchronizing

Each analysis window can synchronize its view with others.
How the signal monitor acts on receiving of the sync signal from other analysis tools depends on the sync selector in the toolbar, identical for every analysis tool.
By default, the display of the signal monitor is locked, it does not react on change commands from other tools. Click on the 'Sync' symbol and a red-yellow triangle appears in the cursor bar which marks the position of the current synchronizing event.
If you switch on the 'Scroll' Symbol the signal monitor always shows the last event resp. level change.
The signal monitor not only reacts on sync-changes from other tools but can also trigger a sync event itself... For that click in the signal view the right mouse key (context menu) to open the sync. menu. The entries are more or less self explaining.



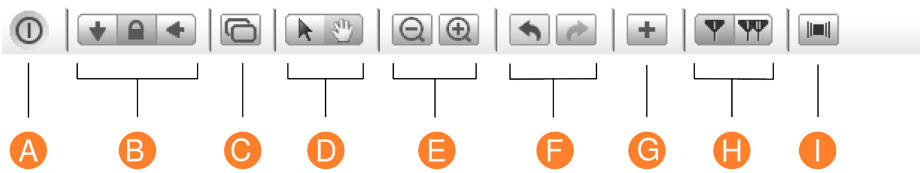
1 **Synchronizing on Cursor 1**
Synchronized is on the first event after the cursor 1 position.

2 **Synchronizing on Cursor 2**
Synchronized is on the first event after the cursor 2 position

3 **Synchronizing the display**
As the synchronizing event the current signal sector is used. That means the first level change seen from the left border.
Why is the first event after the cursor position used and not the cursor position itsself?
Synchronization is only on events not on a certain time in the signal. As the cursor can be between two events (in contrary to other tools) the next following event has to be taken for synchronization.

14.8 The toolbar

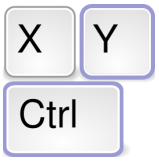
The toolbar serves for a fast access to the most used functions. Some are identical in all monitor windows, some are specific for the protocol monitor.



A	**End:** Save all settings and close the signal monitor window.
B	**Display mode:** According to the mode the window either shows always the current (last recorded) event or locked or actualizes its content synchronous to the other windows.
C	**New view:** Opens a new window with the same sector and settings.
D	**Mouse control:** Optionally the mouse can be used to zoom the selection or to move the signal (hand symbol).
E	**Signal zooming:** Magnifies or demagnifies the visible section in 1, 2, 5 multiplicative factors by choosing the next lower or higher time basis.
F	**Undo/Redo:** Undoes the last change of the visible section or restores it respectively.
G	**Add region:** Saves the range between both cursers as a new region.
H	**Interlock Cursor:** The cursors can be selectively moved singly or together (combined).
I	**Show region dialog:** Opens the region dialog, e.g. to fade in or off regions, to delete them or to name them.

14.9 Short keys

Action	Short command
Online help for the Signal View	F1
Undo last selection/zooming operation	Ctrl + Z
Redo last selection/zooming operation	Ctrl + Y
Add range between cursors as new region	F4
Move view 1 raster towards signal end	Right arrow
Move view 1 raster towards signal start	Left arrow
Move view 10 raster towards signal end	Shift + Right arrow
Move view 10 raster towards signal start	Shift + Left arrow
Move signal horizontal	Shift + Mouse wheel
Zoom in signal	Ctrl + +
Zoom out signal	Ctrl + −
Zooming in/out at mouse position	Ctrl + Mouse wheel
Signal total view	Ctrl + Home
Jump to first event	Home
Jump to last event	End
Open in a new window	Ctrl + Shift + N
Save settings and close signal view	Ctrl + Q



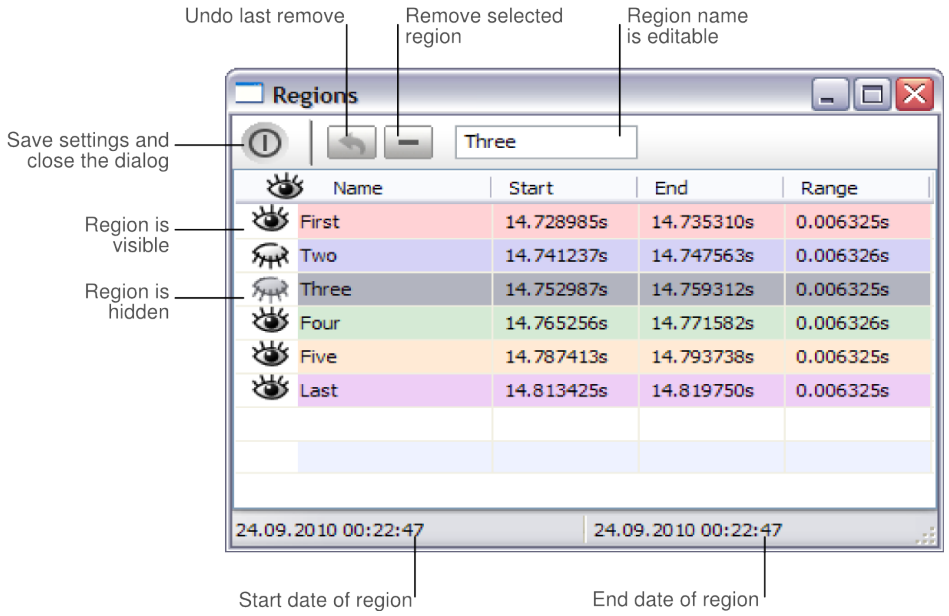
Short commands for the most important functions

15 Regions

To save and quickly recover interesting areas in the recorded data these areas can be marked as regions. Regions are present in all views so that a region, marked in the data monitor is shown in signalmonitor too. Regions can directly accessed.

Regions are selected Ranges which are shown in all analysis windows. Each window can define a selected range as a region and make it available to other windows. Since the different analysis tools represent different kinds of data views each region can be defined in a different way. A region can be defined in the signal monitor as the selection of a certain signal sector, in the data monitor as a certain data sequence or as the occurence of single characters.
This interaction is interesting if you want to examine a defined section in a different view, for example the physical signal state (signal monitor) of a data sequence (data monitor).
As soon as you add a selected range as a region this one is attached to the list of available regions.
Regions are part of each recording and therefor are saved with them self-acting.

The MSB software allows the definition of 8 regions. Every region can be individually named and optionally switched on and off. With exception of the line state monitor you open the region dialog with View→Region dialog in every analysis window. An always opened region window will be put in front of all windows automatically.
The picture above shows a region dialog with all together 6 regions where region nr 2 with the name first request is switched off and therefore not shown in the other analysis tools (indicated by a closed eye).



15.1 Switch regions on/off

Each region can be individually fade in or out. This is reasonable if some regions overlap in the display and make an assignment difficult. To alter the visibility condition of a region simply click onto the eye symbol at the very left side. An open eye symbolizes a visible region, a closed eye a fade out region respectively.

15.2 Remove a region

Delete deletes the current in the region dialog selected region. A further enquiry is not made but you can undo the deletion each time as long as the region dialog is open. Just click the Undo button in the toolbar.

15.3 Rename a region

Rename regions
with a remarkable name

By default regions have no name, but you can choose a remarkable name for each one. Mark the wished region with the left mouse key and click on the edit name field in the toolbar.
The name of a region allows any character except for the colon ':' and is limited to 64 characters. Enter adopt the name to the selected region.

15.4 Move regions into view

Certain segments of the recording are marked as region because they are important parts. Of course you want to bring them fast and easy into the visible part of your analysis windows, e.g. the signal or data monitor.
Possibly you want to compare two regions.
Therefore the region dialog supports the same mechanism for synchronization like the other analyis windows with the exception that it can initiate the synchronization only. Select the appropriate region and click on the start value to bring

up the start of the regions in all analysis windows with acivated synchronization. Or click on the end value to bring up the right limits of the region.
Please note: Only those analysis windows will react which have the synchronization active.

Fetch regions into views

Regions can easily be brought into the visible range of an analysis window by a click onto the start or end value with the left mouse key.

16 A quick start with Lua

Lua is one of the fastest scripting languages in the world.
Because of it's small and simple design it's also easy to learn.
Lua contains a few but also more powerful concepts which
makes it the first choice to add the benefits of a scripting
language to the analyzer software.
This chapter will give you a first glimpse of the language and how
it fits with your analysis.

16.1 Getting started

The most descriptions of a new computer language just begins with the traditional "Hello World". To keep the tradition, our first script will do the same.

Open the Data Monitor and expand the *Watch expressions* on the bottom of the program view. The Watch window contains eight (at first empty) entries. Each one shows the result of the according Lua script.
To start with our little "Hello World" example, double click the first line in the watch list which opens the integrated editor. All you have to do is to input the following two lines:

```
1  require "dv"
2  dv.watch("Hello World")
```

Now let's go and see what happens. The internal Lua engine will translate this expression into it's byte code and execute it after you switch to the watch list again by click on the *Watch* tab.
Later you will see, that each expression will also be executed at every cursor movement, receiving synchronize trigger or after update a find request. But for this time we keep it as simple as possible.
If you don't have made any typo the first watch items should display:

```
--> Hello World
```

Don't worry, if you made some mistake. In this case, the watch window will show you some informational message about the error.

So what's going on here?
The origin Lua doesn't know anything about the analyzer or the watch window of the Data Monitor. Therefore we need some 'glue' to put both things together. One of this glue is the `dv` (Data View) library or module. It provides all necessary output functions to print the results of your scripts in the watch list or to mark the data in the grid window.

To load a module, you simply call `require"`*`modname`*`"`. After that you can access each function provided by the module by prefix the module name with a dot like `dv.watch`.

In normal case you always have to load a module before you can use it. But for all analyzer specific modules and also the basic libraries coming with Lua we decide to 'preload' them into the Lua engine for an easier handling. So the `require"`*`dv`*`"` is obsolete and we will dispense with it in the following examples.

You can add as much arguments to the `watch` function as you like. Because the arguments will be displayed one after another you should also add some separator between them. So let us modify the example above to:

```
1  dv.watch( 1+1, " is less than ", 10/3 )


--> 2 is less than 3.3333333333333
```

As you can see, we are now calling the `dv.watch` function with three complete different arguments separated by a comma. The first is the result of the addition of two integer numbers. The second is a text string identified by its enclosing double quotes. And the last one is the real number 3.333...
But you don't have to bother about the 'type' of each variable. Lua is a dynamically typed language and handles each type automatically[1].
That's nice, but you may ask what's the benefit for the Data Monitor? Read on!

16.2 Accessing the Data Monitor

With Lua you can compute every data displayed by the Data Monitor. It is the aim of the Data Monitor module to provide you with functions to query the cursor position and also the information about each cell in the grid. The next little script shows the current position of the cursor:

```
1  n = dv.cursor()
2  dv.watch( n )
```

The function `dv.cursor()` returns the current cursor position starting with 1 (the upper left cell in the grid) and goes up to the bottom right cell in the data grid. As soon as you move the cursor or click into any cell, the watch entry is updated with the new position.

[1] Lua supports numbers (double-precision floating point), strings like our "Hello World" and so called tables (which are a very mighty concept to realize almost all typical data structures such as arrays, sets, lists, and records. We will discuss the tables later)

After this little introduction it's time for a really important function:
`dv.cell(index)` returns all necessary informations about the given grid entry like the recorded data byte, the timestamp, the direction and something more in a table.
Don't worry about the item 'table'. For a first explanation consider a table as a container with several cases. Each entry is accessable via its name.
The following example shows what we mean:

```
1  c = dv.cursor()
2  dv.watch( "Data: ", dv.cell(c).Data, "Time: ", dv.cell(c).Time )
```

For a better output, we have to put an additional description in front of each information.
Now move the cursor in the data grid with the arrow keys of your keyboard or click on any data cell. If you havn't closed the watch window you will note that every time you change the cursor position, the watch item is updated with the new information computed by this little script.
Line breaks play no role in Lua's syntax. Therefore we can arrange the both arguments of the `dv.watch` function in separate lines to make it more readable.

```
1  c = dv.cursor()
2  dv.watch(
3      "Data:", dv.cell(c).Data,
4      "Time:", dv.cell(c).Time
5  )
```

As mentioned above the function `dv.cell` returns all informations in one table. The entry 'Data' contains the recorded data byte at the given cursor position, the entry 'Time' the timestamp of this data event in seconds.
The syntax for accessing table entries is similar to the modules, i.e. a dot following by the name of the item.
You will find a more detailed description about the data table and all its fields in the following chapter 17.12. For now we continue with another very useful function of the `dv` module.
Take your colour pencils, we are going on and colorize the grid.

16.3 Mark sequences in the data grid

Imagine you can visualize one or more specific data sequences. For instance to highlight a protocol frame or a range of data starting by the cursor position. All you need to know is the function: `dv.mark(index,length,colour)`

For the first we want to show the next four data beginning by the cursor as a 32 bit integer (big endian) and also mark them in the grid.

```
1  c = dv.cursor()
2  v = 0
3  for i=0,3 do
4      v = v * 256
5      v = v + dv.cell( c + i ).Data
6  end
7  dv.watch( "32bit: ", v )
8  -- mark the grid
9  dv.mark( c, 4, 0xCCFFCC )
```

Voila - that's all. As you can see the next four data cells directed by the cursor are highlighted and the result of our little computing is shown in the watch window.

The `dv.mark` function is independent of the current cursor position. Therefore we have to call the function explicitly with the index of the cell where the cursor was located. This will give you the big advance to mark also cells without pending on the cursor movement. The following few lines mark all cells with an error in a warning red background.

```
1  for i=1,dv.size() do
2      err = dv.cell( i ).Error
3      if err == 1 then
4          mark( i, 1, 0xFF0000 )
5      elseif err == 2 then
6          mark( i, 1, 0xFF4040 )
7      elseif err == 3 then
8          mark( i, 1, 0xFF8080 )
9      end
10 end
```

17 Lua beginners guide

In the last chapter we gave you a quick overview what's possible with Lua without any deeper details about the Lua language syntax and all its mighty qualities. Lua is a programming language that offers a very impressive set of features while keeping everything fast, small and simple. So lets go to learn a little bit more about this amazing scripting language.

Each programing language comes with its own ingredients like operators, keywords, functions and last but not least some rules how you put this things together. This is called the programing language syntax. The language syntax declares, how a program has to been written correctly.
In this chapter we will give you a short overview about the Lua language, the supported operators, keywords and some helpful additional modules (libraries) we have integrated in the embedded Lua by default.

Please note! Each time we need an output of some Lua script computing we are using the Data Monitor and its `dv` (Data View) Module.
If you like to give each example a trial just start the MSB-RS485 program, open a Data Monitor and enter the code in a watch entry.

17.1 Lua is case-sensitive

First of all: Lua is a case sensitive language. **while** is a reserved word (a so called keyword), but `WHILE` or `While` are two other identifiers denote a variable or function. Because this is the common use in the most modern languages it shouldn't bewilder you much.

17.2 Whitespaces and line ends

Lua ignores any whitespaces (like the space or tab characters) if they aren't part of a string constant (see 17.4). It also doesn't worry about the indention like Python, therefore you can format your code for your own purpose (or just make it more readable).
Lua doesn't use any special line end and line breaks play no rule in the Lua syntax. The Lua intepreter detects the end of a statement automatically therefore a line can contain more than one statement and a statement can also be split into several lines.

If you write several statements in one line, you can use the semicolon as a separator.

```
1  x = 1 y = 2  --> not very readable but ok
2  x = 1; y = 2 --> better
3  z = x
4  +
5  Y            --> z = 3
```

17.3 Comments

A comment in Lua starts anywhere with a double hyphen `--` and runs until the end of the line. It's also helpful if you want to exclude some lines from execution.
More than this. Lua provides also a block comment which starts with `--[[` and runs until the corresponding `--]]`. It makes it very easy to comment or uncomment several lines as we will show in the following:

```
1  x = 1
2  --[[
3  x = 10
4  --]]
5  dv.watch( x ) --> 1
```

To uncomment the block, just add a single hyphen to the beginning comment. The starting and closing comment identifiers are now just like other commented lines and the statement between them will be executed as normal.

```
1  x = 1
2  ---[[
3  x = 10
4  --]]
5  dv.watch( x ) --> 10
```

17.4 Types and values

Lua is a dynamically typed language. You don't have to specify the type of a value, because each value carries its own type. Lua supports eight basic types but we contemplate only the following ones:

- number
- boolean
- string
- nil
- table
- function

It is common use to define most of the types also as a 'constant' value. A constant is a 'hard coded' value in your program which isn't a result of any computing. Constants are numbers (integer and floating point numbers, whereat Lua doesn't distinguish between them), strings and the boolean values **false** and **true**.

Numbers

Lua simplify the use of different numbers like integer, single float, double float by using only one kind of type for each numbers. Numbers in Lua are always double precision floating point numbers and were converted automatically.

```
1  dv.watch( 1 )      --> 1
2  dv.watch( -12 )    --> -12
3  dv.watch( 100000000000 )  --> 100000000000
```

Notice that the numbers are never rounded into integers to. Hence:

```
1  dv.watch( 10 / 3 )  --> 3.3333333333333
```

Hexadecimal constants

Despite the fact, that Lua compute exclusively with floating points you sometimes want to use other number bases like hex.

```
1  dv.watch( 0x1234 )     --> 4660
```

Floating point constants

Lua can understand also exponent types for expressing numbers. Therefore you can write numeric constants with an optional decimal part and an optional decimal exponent like:

```
1  dv.watch( -0.05 )    --> -0.05
2  dv.watch( 10E-2 )    --> 0.1
3  dv.watch( 1.25E+6 ) --> 1250000
```

Booleans

A boolean data type according to the classical logical state and is either **true** or **false**. If a boolean value isn't true, it has to be false and reversely. Boolean values are used to represent the result of logical or conditional operations.

```
1  dv.watch( 2 > 1 ) --> true
2  x = 2 < 4
3  dv.watch( x ) --> false
```

Strings

Strings in Lua has the common meaning, a sequence of characters. But Lua is, in opposition to other languages, eight-bit clean which has the great advantage: Strings can contain characters with any numeric code, also a null byte (in C the string terminator). With other words: You can store any binary data in a string without an exception.
Strings can be defined using single quotes, double quotes, or double square brackets.

```
1  dv.watch( "It's your code" ) --> It's your code
2  dv.watch( 'He says:"Hi"' )   --> He says: "Hi"
3  dv.watch( [[Hello\nWorld]] ) --> Hello\nWorld
```

Why so different ways to specify a string? It allows you to enclose one type of quotes in the other. And: Double brackets have a few other properties like to suppress escape sequences as seen above.

Escape sequences in strings

Lua strings can contain the following escape sequences:

Escape sequence	Description
\a	bell
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\"	double quote
\'	single quote
\ddd	character with its numeric value ddd

The following examples show their use:

```
1 dv.watch( 'It\'s your code' )      —> It's your code
2 dv.watch( "He says:\"Hi\"" )       —> He says: "Hi"
3 dv.watch( "Tab1\tTab2" )           —> Tab1     Tab2
4 dv.watch( "Two backslashes \\\\" ) —> Two \\
5 dv.watch( "Hello\nworld'' )        —> Hello
6                                       world
7 dv.watch( [[Hello\nworld]] )       —> Hello\nworld
```

You can also specify each character in a string by its numeric decimal value through the escape sequence \ddd as mentioned above. For instance the binary sequence of the bytes 0...3 comes as: "\000\001\002\003".

nil

nil is a special type and indicates a non-value. Each variable has a **nil** value before its first assignment by default.

```
1 dv.watch( x ) —> nil
```

More than: Lua uses **nil** to specify the absence of a useful value (it doesn't exist anymore). By setting a variable to nil you can delete a variable.

Tables

One of Lua's mightiest built-in datatypes is an associate array, which defines one-to-one relationships between keys and values. Key and values can be of each type. And more than this: Because functions are also just some kind of value, you are able to realize some object orientated behaviour with tables too, but this go beyond the scope of this chapter.

Tables has no fixed size and grow up as necessary. If you havn't use of a table anymore, you can throw it away with assigning **nil** to it.
Ok, that's enough for the first. Let's go on with a few examples to bring more light in this matter. At first we will create a simple list containing three line states as strings:

```
1  linestates = { "mark", "space", "invalid" }
2  dv.watch( linestate[2] ) --> space
```

This statement in line 1 will initialize the first entry in the table `linestates[1]` with "mark", the second `linestates[2]` with "space" and the third with "invalid". Please note, that in Lua indexes start with 1 and not with 0 (like in C). The table here behaves like a simple list. You can append a new element to a table using the `table.insert(`*`table,value`*`)` function.

```
1  t = { 1, 2, 3, 4, 5 }
2  table.insert( t, 6 ) --> 1, 2, 3, 4, 5, 6
3  dv.watch( "Count: ", #t )  --> Count: 6
```

The statement in line 3 uses Lua's internal length operator #, see also section 17.8. If you like to insert a new item somewhere in between the list without having to shuffle the other elements around you can use the same function with an additional position parameter `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  dv.watch( "Count: ", #t )  --> Count: 6
```

To remove an element from the table (or list) use the call:
`table.remove(`*`table,position`*`)`.

```
1  t = { 1, 2, 3, 4, 5 }
2  table.remove( t, 4 ) --> 1, 2, 3, 5
3  dv.watch( "Count: ", #t )  --> Count: 4
```

You can always replace one element with another one just simply by overwriting it. Each element in the list is accessable with the index operator `[]`. To rewrite the second element with 200 `t[2]=200` will do the job. You can also query the value at a given position conversely with: `v=t[2]`. If there doesn't exist a value at the index position, a **nil** will return. On the other hand: If you try to overwrite a value at an invalid position it will append to the list.

In the examples above the keys of the associated array are set by default (as numeric) and only the values are given. But you can choose any desired index or key value too. Imagine a data type representing a point:

```
1  point = { x = 5, y = 10 }
2  dv.watch( point.x, point.y ) --> 5    10
```

A table storing data with a key/value relationship is sometimes called a dictionary.

Functions

We will discuss functions in this paragraph only according to their role as values. For detailed information about functions and their definition please take a look at section 17.10.
In Lua, functions are assigned to variables, just like numbers and strings. If you are bothering with the long term `dv.watch` assign it to the internal Lua `print` (the latter is without any use in the analyzer enviroment and can therefore be overwritten).

```
1  print = dv.watch
2  print( "Hello World" ) --> output in the watch window
```

Furthermore a function is a variable referencing the code of the according function and you can overwrite it with any other value.

17.5 Identifiers

In computer languages identifiers are names referencing some kind of variable or a function. Some identifiers are reserved by the language itself as so called keywords. (We already know the boolean keywords **true** and **false**). Others are built in functions like `print`.
Names (or identifier) in Lua can be any string of letters, digits and underscores, not beginning with a digit[1]. Valid names are:

```
x          y          ABC        t1         _nm
aVeryLongVariableName          the_last_result
```

Invalid names throw an error

```
1  dv.watch( 2n )      --> malformed number near '2a'
```

17.6 Keywords

The following keywords are reserved and cannot be used as names:

```
end        false      for        function   if
in         local      nil        not        or
repeat     return     then       true       until      while
```

Please remember: Because Lua is case-sensitiv, **and** is a keyword, whereas `And` and `AND` are just two other and different identifiers!

17.7 Variables

Variables are like a named box that can store any kind of value. In Lua variables can cover a single number as also a million characters or a container of key-value pairs. The name of the variable has to be a valid identifier (see above).
You don't have to declare a variable before the first use. As soon as the Lua interpreter finds a new variable it will created it automatically.

[1]Because the analyzer software and Lua itself too reserves the starting _ for some language supplements we recommend to start a variable name without an underscore.

Assignment

Note! Before the first assignment to a variable, its value is nil. Assignment is the general procedure to set or change the value of a variable (or a table field).

```
1  if x == nil then
2      x = 1
3  end
4  dv.watch( x )        --> 1
```

As mentioned before: Lua is a dynamically typed language. You don't have to define the type of a variable because each value carries its own type.
And: The type of a variable is an object of change. Every time you assign a new kind of value to a variable it change its type again.

```
1  x = 1
2  x = "Hello World"
3  dv.watch( x )        --> Hello World
```

Lua also supports multiple assignment which means: A count of values is assigned to a count of variables in one step. We will discuss this very nice feature in a later section in the context of functions with multiple results. For the curious reader here a litte code example exchanging the values of two variables without any additional temporary variable:

```
1  x = 5
2  y = 10
3  x,y = y,x
4  dv.watch( x, y ) --> 10    5
```

Global and local variables

There are three kinds of variables in Lua: Global variables, local variables and table fields (we discuss tables later).
By default each variable is a global one which means: It is accessable during the complete runtime. Global values resides in a 'global' space (in detail in a global table).
Beside this local variables is only valid in the context or block where they are declared.

```
1  y = 10
2  if x == nil then
3      local y = 5
4      x = 1
5  end
6  dv.watch( x, y )       --> 1     10
```

It's a common strategy to use local variables wherever you don't like to access a global one. For instance if you need some variables only in a function, declare them as local.

17.8 Operators

Operators are symbols, which activate calculation, when using them in combination with variables, values or results from expressions. Lua supports arithmetic, conditional and logical operators. In addition a very helpful string concatenation operator.

Arithmetic operators

Lua supports the usual arithmetic operators: the binary + (addition), - (subtraction), * (multiplication), / (division), % (modulo), ^ (exponentiation) and unary - (negation).

```
+     -     *     /     %     ^
```

Please note: In Lua Numbers are always represented as real (double-precision floating-point) numbers.

Conditional operators

Conditional operators always result in **true** or **false**. Lua provides the following conditional (or relational) operators:

```
<     >     <=    >=    ==    ~=
```

The == operator tests for equality, the operator ~= is the opposite of equality. You can apply all operators to any two values, numbers and strings (all condition operators also dealing with strings). If the both values have different types, Lua handles them as not equal.
Please note: The value 0 isn't a false test condition as you may suspect from other languages.

```
1 dv.watch("abc" < "def" ) --> true
2 dv.watch( 0 or true)     --> 0
3 dv.watch( false or true) --> true
```

Logical operators

Lua provides logical operators for use in statements. They are: **and**, **or** and **not**. The logical operators behave in a common way. They always evaluate to either **true** or **false**. In a special case the value **nil** will be considered as **false**. **and** and **or** use a short-cut evaluation, means: They evaluate their second operand only when necessary. For instance:

```
1 dv.watch( 4 and 5 )         --> 5
2 dv.watch( 4 or 5 )          --> 4 (short-cut evaluation)
3 dv.watch( false and true )  --> false (short-cut evaluation)
4 dv.watch( a and 1 )         --> nil, because a wasn't specified
5 dv.watch( not false )       --> true
```

String concatenation operator

The two dots `..` denote the concatenation operator in Lua. The operator takes two strings (numbers will convert by Lua in strings) and combines them in one. Please note! If the first operand is a number you have to insert a space between the number and the `..` operator. Otherwise Lua missinterprets the first dot as a decimal point and throws an error. Hence:

```
1 dv.watch( "Hello " .. "World" ) --> Hello World
2 dv.watch( "100 " .. "sec" )     --> 100sec
```

The concatenation operator always creates a new string and leaves the operands behind without modifications.

The length operator

The length operator is denoted by #. The length operator returns the count of bytes in a string or the items in a table if the table doesn't have any gaps.

```
1 dv.watch( #"Hello World" ) --> 11
```

Precedence

The following is a list of all Lua operators and their order of predence. The operators are listed highest to lowest.

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

If in doubt, use explicit parentheses. It makes your code more readable and prevents you from an any additional look in this manual.

17.9 Control structures

Control structures tell the program which way to proceed in the code (or script). They are integrated part of each language and something like the traffic police in Lua scripts.
Lua provides the following set of control structures, the **if** for conditional executions, **for**, **repeat** and **while** for iteration. All of them, except **repeat**, needs the explicit **end** terminator. **repeat** has to be closed with **until**.

if then else

The **if** statement tests a condition and executes depending to its result the **then** section or the **else** section. The later one is optional.

```
1 if x < 0 then
2     x = 0
3 else
4     x = math.sqrt( x )
5 end
```

You can put small condition tests in a single line like:

```
1 function max( a, b )
2     if a > b then return a else return b end
3 end
```

Lua doesn't have any switch statement. Therefore the following **if**, **elseif** chains are common.

```
1  if a >= 100 then
2      exp = 2
3  elseif a >= 10 then
4      exp = 1
5  else
6      exp = 0
7  end
```

while

The **while** statement executes a block as soon as the **while** condition is true. As usual the condition is tested first. The block will never execute if the first test results in false.

```
1  local x = 0
2  while x < 10 do
3      x = x + 1
4  end
```

repeat

On the contrary the **repeat** statement repeats its body until the condition is true. Because the test is done after the block, the block is always executed at least once. Please note the different terminator **until**.

```
1  local x = 0
2  do
3      x = x + 1
4  until x < 10
```

Numeric for

Lua provides two **for** statements but we confine ourself to describe only the first and more comprehensible numeric **for**. The numeric **for** has a variable with a starting assignment, an end value and an optional step value. The latter one is 1 by default.

```
1  for var=start,end,step do
2      --> do something
3  end
```

For instance a

```
1  local m = 0
2  for n=0, 9, 0.1 do
3      m = m + 1
4  end
5  dv.watch( m )  --> 5.5
```

break

A **break** cancels a **for**, **repeat** or **while** loop and continues with the instructions after the loop block.

```
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  dv.watch( m )  --> 2.5
```

17.10 Functions

Every computer language has functions, even the simple ones. Lua is no exception. A function can perform a specific task and/or compute and return values. If you notice the plural values you are right - Lua functions are able to return multiple results.
In both cases you have to give the function a list of arguments enclosed in parentheses. If the function doesn't need any argument, you still give it a empty list specified by `()`.

Function call

Simply said a function is called just by its name and an optional count of arguments as a list. You can invoke a function with more than the specified arguments whereas only the first are handled but if you try to call a function with fewer parameters you get an error.
You learned about some already defined functions like the `dv.watch(...)` or the `math.sqrt(x)`. Most of this functions are part of some module, others are defined by the analyzer.

Function definition

Function are conventionally defined with the keyword **function**.

```
1  function fnc( arg1, arg2, ... )
2      --> the function body
3  end
```

For example a maximum function returning the greater value of two given numbers is defined as:

```
1  function max( n1, n2 )
2      if n1 > n2 then return n1 else return n2 end
3  end
```

Function doesn't have to return a value. In this case you can just omit the **return** statement or leave the **return** without any following value(s).
As mentioned above, a function in Lua can also return multiple results. This is a big advantage, because you don't have to collect the results in some container and don't run the risk of side effects by set up a global (outstanding) variable.
Several predefined functions in Lua return multiple values like the `math.modf(x)` (one result is the integral part of x and the other one the fractional part of x). For instance:

```
1  dv.watch( math.modf( 5.125 ) ) --> 5      0.125
```

The definition of a function with multiple results is as easy as of each other function. An example shows the differences. Imagine some function to convert coordinates of a plane polar system (radius and angle) into the cartesian system (x and y).

```
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
```

Instead of build some container for both results we just return them as a multiple result.

17.11 Modules

A module is a package of functions for a special purpose. You already know the modules `math` or the `dv` module belonging to the analyzer software. From the Lua point of view each module is a table which contains functions (functions are a special kind of value as we mentioned before), global module values, module constants etc. Therefore each module function is called like a table element with the prefixed table (module) name and a dot.

Standard Modules

The following standard modules are supported by Lua in the analyzer enviroment, see section Limitations 17.13.
In normal case you have to load a module first with the `require("modname")` statement before you can access any module function. But for the analyzer enviroment we decide to preload the following standard modules automatically for an easier use.

Module	Description
math	The mathematical library. It provides access to the mathematical functions defined by the C standard.
string	The string library provides a lot of generic function for string manipulations, searching and extracting substrings and pattern matching with regular expressions.
table	The table library contains special functions for array or list tables (with a numeric indexing).

You will find a short (but very good) reference paper about Lua and its supported modules as one PDF file at: http://lua-users.org/wiki/LuaShortReference
The paper is also contributed by the MSB-RS485 software. Take a look in the `doc` directory of your installation folder.

Analyzer Modules

The MSB-RS485 software offers some additional modules to connect the capabilities of the analyzer with the Lua language. Most of them are fitted as best to the according view.

- bit Module
- dv Module
- record Module

The `bit` module provides you with binary bit operations which are not implemented in Lua.
The second module you have get to know is the Lua support for the Data Monitor.
With the `record` module you can query imformations about the current record like the used protocol, the start time of the record and the names for each signal.

Bit Module

Lua does not know different number formats but processes all numbers as floating point values with double precision. Therefore bit operations are not provided in the standard implementation of Lua.
On protocol or data level you will sometimes face the task to evaluate single bits or to modify data bytes bit-wise (e.g. in the context with check sum evaluation).
The `bit` module expand the integrated Lua interpreter with the following functions:

The Bit Module (bit)

`band(x1,x2)`	returns the bitwise and of its arguments x1 and x2, for instance `bit.band(0xFF,0x01)`
`bor(x1,x2)`	returns the bitwise or of its arguments x1 and x2, for instance `bit.bor(0xFF,0x01)`
`bxor(x1,x2)`	returns the bitwise xor of its arguments x1 and x2, for instance `bit.bxor(0xFF,0x0F)`
`bnot(x)`	The result is the logical negation of the single bits (also ones complement). Each 1 is replaced by a 0 and vice versa. For instance `bit.bnot(0x55)`
`lshift(x,n)`	returns bitwise logical left-shift of its first argument x by the number of bits given by the second argument n. For instance `bit.lshift(0x100,2)`
`rshift(x,n)`	returns bitwise logical right-shift of its first argument x by the number of bits given by the second argument n. For instance `bit.rshift(0x1FF,1)`

9bit.msbprj
Example of a LRC calculation

The sample project `9bit.msbprj` shows the use of the bit module based on the evaluation of a LRC check sum. You will find it in the `examples/DataView` sub-directory of the installation directory.

Data View Module

The Data Monitor (or Data View) module `dv` provides you with functions to query all information about the recorded data as displayed in the grid. For instance you can access each cell via its index, the current cursor position and the amount of the cells in the grid.

Leave it to Lua to compute the data from specific cells and output it in up to eight independent entries in the Watch window. For example validate a checksum or convert the data of different cells into floating point numbers etc.

Beside this you are able to mark a specified cell or a sequence of cells with a given colour. And you can dye the complete data field or just a small number of cells relative to the cursor position.

Last but not least: If you are unhappy with the information shown in both fields of the statusbar, don't worry. The Data Monitor modules provides you with a function to put in any information you like to see.

Further examples...
Error displaying,
S-Record visualisation

Examples for the use of the Data View Module can also be found in the subdirectory `examples/DataView` of the installation directory.
`errors.msbprj` colors all data bytes containing a frame or parity error or occurred breaks. `srecord.msbprj` visualizes a recorded S-Record transmission.

The Data View (dv) module

`cell(`*`index`*`)`	returns the log event table from the cell with the given index. The first cell of the Data Monitor is indexed by 1, the last with `dv.size()`. See also 17.12
`cursor()`	returns the current cursor position in the grid as index. The index starts with 1, the last possible index is `dv.size()`.
`mark(`*`pos,n,col`*`)`	marks the next n data cells from position pos (starting with the first cell as 1) with the given colour col
`size()`	returns the index of the last visible cell in the Data Monitor grid.
`statusbar(`*`f,args`*`)`	displays each of the given arguments args in the left field (f=1) or right field (f=2) of the statusbar
`watch(`*`args`*`)`	displays each of the passed args to the according watch entry in the Data Monitor

Record Module

The `record` module offers you some additional functions to query important informations about the current record. For instance: the used baudrate, wordlen, parity setting, stopbits, all signal names and the start time (date) of the recording.

The functions in detail:

Das Record Modul (record)	
`protocol()`	returns the baudrate, count of databits, parity setting **N**one, **O**dd, **E**ven and used stopbits as a value list. For example: `baud,databits,parity,stopbits = record.protocol()`
`signalnames()`	returns all eight signalnames as a list from Signal1 to Signal8. For instance: `s1,s2,s3,s4,s5,s6,s7,s8 = record.signalnames()`
`starttime()`	supplies the start time (date) of the current record as the so called Unixtime (represents the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC)).

17.12 Analyzer specific data types

The analyzer stores each detected change as a special record. This record contains all the associated information like the time in microsecs, the kind of alteration, the state of each line and other. With a
`dv.cell(dv.cursor())`
you are able to request the event from the current cursor position (or relative to it) as a table and compute the information for your own purpose.
Each field in the table represents one spezific information about the event. Valid table fields are:

The LogEvent table

`dv.cell(i).Data`	returns the data (up to 9 bit) of the given cell `i`.
`dv.cell(i).CountAll`	the current data event number (all data channels A+B).
`dv.cell(i).CountThis`	the current data event number (only for this direction).
`dv.cell(i).Error`	returns the error (if exist) of the given cell `i` or 0. Errors are: 1=Frame, 2=Parity, 3=Break.
`dv.cell(i).Position`	the event number of the given cell `i`. The event counting starts with 0 and includes all selected events.
`dv.cell(i).Sig?`	query the logical signal state (line level) at the time of the given data event (cell). ? denotes the signal number 1...8. The result is: +1, –1 or 0 (inactive).
`dv.cell(i).Source`	returns the the data source of the given cell `i`. 1: Data channel A, 2: Data channel B.
`dv.cell(i).Time`	the time stamp of the given cell `i` in seconds as a floating point number.
`dv.cell(i).Valid`	returns the valid state of the given cell `i`. Empty cells (marked as XX) returns false, otherwise true.

17.13 Limitations

The original Lua comes with a lot of additional modules. Not all of them are for any use in the analyzer enviroment and therefore are excluded from the embedded Lua implementation. Not available modules are:

- I/O Library
- OS Library

To avoid a slow down of the analyzer software by busy or overloaded computing scripts, for instance a long running or endless loop, the internal Lua VM (virtual machine) doesn't allow to out ran a specified quantum of operations. In this case, the VM aborts the execution of the script and throws out an informational message.
Just try the following:

```
1  local x = 0
2  while true do
3      x = x + 1
4  end
```

```
--> [string "local x = 0..."]:-1: overrun of allowed executions
```

17.14 Further information

This chapter can't replace any good introduction to Lua. It only covers the necessary information you need to undertake the first steps with Lua in the MSB-RS485 software.
It also gives you a small outlook of all the language features Lua comes with.
For more information about Lua please visit the Lua website at http://www.lua.org.
You will find a very good tutorial at http://lua-users.org/wiki/TutorialDirectory too.

18 Synchronize two analyzers

You have two connections (RS232 and/or RS422/485) which you want to watch or examine in parallel, for instance IN and OUT data of a protocol converter, different bus segments or generally interdependent data transmissions.
How you have to proceed and what is to be regarded is described in this chapter.

For a simultaneous recording of two separated connections you need two MSB analyzers. But this is only one requirement. To compare two recorded data files the data have to be in a precise time relationship. Without this relationship you can neither decide about the chronological sequence nor check the synchronicity of certain events.
For example when was a data byte or data sequence sent in relationship to the data of another connection. What happened in both connections at a defined point in time.

18.1 Technical requirements

One of the outstanding features of the MSB-Analyzer is the exact time measurement and visualizing of the time behavior in microsecond resolution. This precision is necessary to deliver correct results even for higher baud rates and is also valid for the common analysis of two connections. What does this mean?

Imagine two agents which shall enter a secured building and have to watch and record the way of the guards at different positions. Before they begin they compare their watches. This is done within a difference of a typical one second divergence. Both agents have watches which differ not more than one second per day. It doesn't bear contemplating if both watches would disperse after some minutes. Now each agent proceeds to his position.
Each one notes the point in time (seconds precision) of the change of guards. As the watching takes some days both agents synchronize their watches repeatedly at midnight by a short radio pulse from one of the agents.

For the successful execution of their plan they have to know each step of the guards within a difference of one second.
The same procedure but with a far higher precision must be performed for the simultaneous recording of two connections. The time comparison at the begin-

ning is done by the exact simultaneous start of the recording, where simultaneous means a precision of one microsecond.

Of course the clocks of both MSB-Analyzer are more precise than the watches of the agents, but nevertheless they also differ from each other because of small natural differences of the crystal oscillators. They have to be synchronized in regular intervals. The MSB-Analyzer uses for both, the synchronous start and the regular timing of the clocks an additional synchronizing connection.

For this purpose each MSB-Analyzer offers a so called 'MSB-Link' jack in RJ45 design. To synchronize two analyzers simply connect them with a standard network 1:1 cable. Please regard that although standard network cables are used you must not connect the analyzer to another data network. The signals are not compatible and the analyzer could be damaged.

It doesn't matter if both analyzers are connected to the same or different PCs. The PCS also do not have to share a common network. The only restriction is the length of the synchronizing cable between both devices[1]. The synchronizing affects only the start of the recording and the precise keeping of the common time basis. That means that the time stamps of both analyzers are comparable on millionth second.

Furthermore both analyzers work fully independent. That means that you can record completely different protocols and events (baud rate, data format a.s.o.). Moreover you can connect a MSB-RS232 and a MSB-RS485 analyzer to simultaneously analyzer RS232 and RS485/422 connections, for example in interface converters.

18.2 Master Slave operation

It makes no sense to start the recording of the synchronized analyzers separately. Especially if both devices run at different PCs which may be at different locations. Start, Pause, Stop of the common recording are operated from a before as 'Master' defined analyzer. This one is freely selectable. The second analyzer, connected via the synchronizing cable, is automatically set as 'Slave' and controls its own recording synchronous to the Master with microsecond precision.

Both synchronized analyzers can be configured in the way that the recorded data are automatically stored to a predefined storage location when the record is stopped. This can be a local drive of the PC where the respective analyzer is connected to. It can also be any other drive, for instance a network drive. So both analyzers can store their data on the same drive but in different files.

The recorded data files are named with the serial number of the analyzer and the date/time of the start of recording. Additionally you can place any character string at the beginning (prefix).

[1] Tested was a CAT6 network cable with 100m length.

18.3 Establish a synchronous record

Now you have a rough idea of how the synchronous recording works. Let's come to the practical part. Imagine you have two RS232 connections you want to record commonly. To simplify matters the recording is done at only one PC, where both analyzers are connected to.
At first connect both devices via a standard network cable. We recommend cable of category CAT-6, but for the most applications cables of category CAT 5 are sufficient.

> Warning!
> Please note that the analyzer must NOT be connected to a PC network through the MSB Link jack. This will probably result in a damage of the MSB-Analyzer.

Start the Analyzer software with the desktop icon. If multiple analyzers are connected to the same PC you have to select the wanted analyzer from a list (select connected analyzer). Repeat this step for the second analyzer. The same procedure applies even if the analyzers are connected to different PCs. Place both control programs (each connected to a different analyzer) on the screen.

Still both devices work independently of each other. You can start, stop or pause the analyzers individually. They also work on different time bases.
Since both analyzers can record different connection protocols or types (RS232 or RS485) you first have to configure each analyzer according to the requirements of the examined connections. This is done just like the recording of a not simultaneous recording. Set all the required parameters in the settings dialogs of the analyzers.

Choose a storage place
for the automatically stored records

By default both analyzers store their data on the desktop as soon as the recording is stopped by the Master (by you). You can also set the place to any other location by selecting another directory in the settings dialog 'Auto store'.

The file name is set by the analyzer program itself to avoid errors for repeated storing by already available files. It also makes it possible to assign the file to the analyzer exclusively.
The file name consists of the following parts, here a sample of the analyzer with the serial number MSB01060, started on 16th of April 2014 at 15:32.17.

```
MSB01060-20140416153217.msblog
```

Additionally you can place any character string in front of the name as a prefix, e.g. MASTER or SLAVE.

Master and Slave
Specify the record master

After having configured both devices you only have to set the Master for synchronous recording (assumed both analyzers are connected via network cable).

Activate the device which shall be the Master device. This is done in the settings dialog under 'Analyzer is Master'. In the program display the word Master

is shown above the running recording time.
At the same time the analyzer, which is connected to the Master, displays the word 'Slave' in its program display and the buttons and menu entries to control the recording are deactivated.

As soon as you uncheck the Master entry both analyzers are autonomous devices again. The same applies if you disconnect the link cable.

Close the settings dialog and click in the control program of the master on the start button. Both devices change to the record mode, indicated by the respective button display and the red LED at the analyzers themselves.
Click the Pause button of the master to hold the recording.
After clicking the stop button of the master the recording of both analyzers are finished. They automatically store their data on your desktop or any other specified directory.

You can repeat this procedure any time. As soon as you click the start button both analyzers will start a new recording and and after clicking on stop they will store them as two new data files.
This way of operation does not differ if both analyzers are connected to different PCs which may also be placed in different rooms. The only requirement is the connection via the link cable.

18.4 Analyse a synchronous record

The MSB-Analyzer software is optimized to visualize a single recording by multiple different views. The loading of multiple record files is not possible because two or more records with different settings makes no sense within the application. For instance a RS232 and a RS485 recording need different displays and dialogs[2].
But how can two records be analyzed at the same time?

The Analyser software consequently extends the already available communication between the views of a single application to multiple parallel running applications. That means that like the the signal monitor follows the cursor of the data view now all views of the separately running analyzer programs are synchronized to the cursor That has a number of crucial advantages:

- Comparing analysis of differing recordings (baudrate, protocol, type of communication, ..).
- Synchronous moving and parallel display of certain ranges in both records (e.g. search for events in record A and showing the respective signal sequence in record B).
- No new operating scheme, no new menus.

Therefor the analysis of two synchronized recordings is not different to the analysis of a single recording. Instead of starting only one analyser application you

[2] In the end a running MSB analyzer program application correspondents to ONE recording. This is the same as for Audio or Video applications.

now start two different programs for the Master and the Slave recording.

For the evaluation in conjunction you do not need a connected analyzer. The examination can be done as is usual in the offline mode.
Click on the master and slave recordings one after another. Both applications make the accustomed access to the respective data. The views of each application synchronize their windows if the synch. Mode is activated in their tool bar.

To synchronize the views between BOTH running applications you first have to enable this feature. By default the synchronization from external sources is disabled.
The enabling is done for all Views of an application centrally in the control program at 'common settings'. Activate 'allow external synchronization'.

Ext. Synchronisation is enabled in the record settings

By enabling the external synchronization the control program receives the mouse clicks or events (search results, region selection, etc.) from a parallel running analyzer application and passes them to its opened views. Each view with active synch. setting reacts on these events and actualizes its display.
In this way you can watch the slave recording at any time point in the master recording and vice versa. Both applications keep their views synchronous to one defined time stamp.

Synchronous display of all Views in both records

18.5 Conclusion

The comparing recording or analyzing of two separate connections requires a high precise reference to set the recorded data and events in relationship to each other.
These chapters showed you why this is necessary, which technical requirements have to be fulfilled and how such a recording has do be done with two MSB analyzers.
Here come the necessary steps again without ballast.

Synchronous recording

1. Connect both analyzers which shall be synchronized via a standard network cable.
2. Connect both analyzers to one or two PCs.
3. Start a separate MSB analyzer program for both devices.
4. Set up individual connection parameters for both analyzers.
5. Check if the automatic storage after record stop is activated and specify a storing location if necessary.
6. Define one of the devices as Master in the set up dialog of the appropriate application program at 'Record'.
7. Start the synchronous recording at the master control program.
8. The recording is also stopped by the master whereas both records are automatically stored separately.

Synchronous analysis

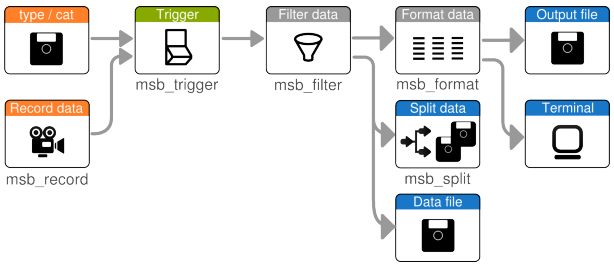
For the evaluation of two synchronously logged records you do not need a connected analyzer, but both MSB analyzer programs have to run on the same computer because a synchronization of the views is not possible through the network cable in opposite to the synchronization of the recordings themselves.

1. Double click on both (master and slave) recordings resp. start two MSB-Analyzer programs. Load the files into the control program.
2. Activate in both programs under 'General' the entry 'allow external synchronization'
3. Place both control programs and the wanted views on the screen.
4. Navigate as used through both recordings. The views in synchronous mode will automatically align their content to the examined time period.

19 Commandline API

You want to automatize the recording of a data connection and process the recorded data in your own application, or to store respectively output them?
A long recording should saved as several sequenced files or splitted afterwards.
You like to control the analyser from within your application.
The MSB-Analyzer software offers a series of powerful tools which we will describe in this chapter.

After installation of the analyzer software you will find some helpful other tools in the installation directory beside the programs for operating the MSB-RS485 and visualizing the recorded data.
All these programs are based on command lines and might be used as part of batch files or shell scripts. According to the Unix philosophy *'Do only one thing but do it well'* each of these programs has only one function. By their capability to read from the standard input and send their results back to the standard output these programs may be combined in any way (program tool chain).
Further more: You can combine them with a lot of other programs which are able to handle data via standard input/output.



The command line programs in overview:

- **msb_record**
  This tool controls the analyser and writes all received data to the standard output or in a given file.
- **msb_format**
  Output the analyser data read from standard input in a user specified format.

- **msb_filter**
  Filters the analyser data passed from standard input to output by user defined rules.
- **msb_split**
  `msb_split` reads data or a record file from the standard input and splits the output into smaller record files.
- **msb_trigger**
  Checks the data from the standard input against some given trigger conditions and start or stop passing the data to standard out according to the result.

19.1 Combine the programs as a tool chain

You can simply put the tools mentioned above together to work as a processing chain. Thereby each program processes data of the former program and forwards it to the next tool in the queue.
The processing (tool) chain always consists of a data source and a data sink. The single programs can be linked with the '|' operator which is identical for Windows and Linux.

```
DATASOURCE | MANIPULATOR1 | MANIPULATOR2 | ... | DATASINK
```

Data source

Each tool chain starts with a data source. The data source provides the following programs with the necessary input, here most of all the tool `msb_record`. But the output of a already existing analyser record file via `type` (Windows) respectively `cat` (Linux) works just as well. For instance:
`type recordfile.msblog` or `cat recordfile.msblog`

Manipulators

A program which modifies the data during the forwarding is called a manipulator. A typical manipulator might remove special parts of the read data before it pass it on the next link in the chain or change the read data in another format specified by the user.
Therewith you can extend or complete the processing of the data simply by inserting any number of manipulators in the processing chain. One manipulator might remove unwanted data before the next tool converts the remaining data into another format, for instance with the `msb_format`.

Data sink

A data sink specifies the end of the processing chain. A typicall sink is the screen output (of a command line window) or a file storing the data.
But a data sink might be also your own application which read the resulting data and processes it for your own purpose, for instance a LabView application.
The `msb_split` tool is a representative data sink. The program doesn't forward the data to other tools but stores it as serveral files on your hard disk.

Some examples

The program folder of the MSB-Analyzer software will be added automatically to the search path for executables during installation. For a first try, you just have to open a command shell (console). We will use the example records coming with the software package as a data source. Therefore you don't need

a connected analyser.
Go to the example directory and pipe a record into the `msb_format` program like:

```
type DataView\9bit.msblog | msb_format
```

Linux users have to use the `cat` command instead of `type`. The command `type record` works as a data source like an active recording with `msb_record`. `msb_format` is the manipulator tool in the processing chain and forwards the result to the command line window which stands for the data sink and just displays the result on your screen.

Now use the `msb_split` tool to split the same record file in several little pieces.

```
type DataView\9bit.msblog | msb_split -n1000
```

Without any arguments `msb_split` simply creates the following two files in the current directory named as xaa.msblog and xab.msblog.
You will find more detailed information about these tools in the program relating sections below.

19.2 Record data with msb_record

As the name indicated this program controls the operating and recording of a connected MSB-Analyzer. At the same time `msb_record` functions as a data source for all other tools.
Called without any further arguments `msb_record` searches for a connected analyser, transferes the firmware if needed and starts a new record of all transmitted data bytes with 115200 baud and 8N1 protocol as default.
If the tool doesn't find any device or detects more than one analyser, it will give you appropriate message. In the last case you can select the proper analyser by passing the serial number of the analyser to the program.

`msb_record` writes the recorded data directly to the standard output to make them available for the other tools. We will illustrate this with the following example, inputed directly on the command line window:

```
msb_record | msb_format
```

All recorded data are forwarded with the pipe operator '|' to the next program in the tool chain, here the `msb_format`[1]. The latter reads all data received from it's standard input, makes some transformation and put the result to the standard output again. In this case - unless there are more programs in the queue, it writes it just as a simple informal list.

```
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' "
```

[1] You can of course execute the `msb_record` tool standalone. But because the outputed data is in a binary format this doesn't make any real sense.

Press 'Ctrl+C' to abort the program.

In general you will use `msb_record` either in a combination with other command line tools as shown above or to write the output directly into a file. There are two ways to save the recorded data as a file. You can redirect the output like:

```
msb_record > output.msblog
```

Or you pass a file name as an additional argument:

```
msb_record -o output.msblog
```

Connection settings and events

We didn't bother about the connection properties so far and called the record tool implicitly with its default settings. We restricted ourself also to record only the transmitted data bytes and ignored the change of the line states.
Imagine you have a serial connection working with 38400 baud, 7 databits and an even parity. The MSB-Analyzer doesn't worry about the number of stop bits, but nevertheless we will assume 2 stop bits here.
Beside the raw data bytes we like to analyse the line level change of both data transmission lines (here RxD, TxD) as well as the handshake control lines RTS and CTS. The call of the `msb_record` tool is described as follows:

```
msb_record --baudrate=38400 --protocol=7E2 --logsignals=2,3,6,7
```

or in a short form:

```
msb_record -b 38400 -p 7E2 -l 2,3,6,7
```

We will show another way to pass the parameters via a configuration file later. For now and most important:
Don't separate the arguments from the associate program! In the command line above all `msb_record` parameters have to specified before the pipe operator.

```
msb_record -b 38400 -p 7E2 -l 2,3,6,7 | msb_format
```

This applies for all tools, programs belonging to the MSB-Analyzer and also other ones.

Usage in your own application

Maybe you thinking that's all pretty interesting, but how can I use these tools within my own application?
Your application only has to fulfil the following requirements:

1. Execution of any command from within your application.
2. Read of a file opened/written by another process.

That sounds worse than it is.
The most programming languages come with a special function to execute an external command. For instance: C has the `system` and `popen` function, LabView offers a System Exec VI. You can call an external command mostly in two ways:

First: The caller (your application) waits until completition of the command. We don't recomment this, because it would block your application.
Second: The command chain is executed as a so called detached process. In this case the function (with the command chain) returns immediately to the caller and the command chains works parallel to your application.

So far as good, but how can you get the results of the command chain?
Your tool chain can write the results in a file where you read them back from within your application. Or: You fetch the results directly from the tool chain output. Which of one fits you best depends on your application language.

The detached tool chain process will be stopped and closed automatically at the end of your application. But there is another option to control the data collection of a parallel running `msb_record`.

Remote control

The `msb_record` tool contains a simple and easy to access inter process communication method which works for both platforms (Linux and Windows) equally.
Sending a command to a running background process is done by calling the `msb_record` program with the parameter '-r command' or executing the command from your application via system call.

Just open two command shells and start a record in one of them with:

```
msb_record | msb_format
```

The connected MSB-Analyzer is initialized and the recording is started, indicated by a permanent lighted red LED1. It doesn't matter if there aren't any data available.

Now switch to the second console (command window) and stop the recording with:

```
msb_record -r stop
```

The execution of the stop command can be checked by watching the analyzer red LEDs. They are blinking alternatively again.
To start or resume the recording repeat the call but now with the command 'start':

```
msb_record -r start
```

The command `msb_record -r quit` ends the process respectively the tool chain and closes the output channel/file.

Synchronous recording with two or more analyzers

Every analyzer has a MSB Link connector to synchronize the recordings of two or more devices (using an MSB Port-Link Doubler from IFTOOLS) with one microsecond resolution. We described this special operation and its benefits in detail in the chapter 18.

But synchronized recordings are also possible with the command line tools. Here we therefore will explain the handling of two linked analyzers via the API tools.

Let us assume that you have two analyzers. Both are connected via the MSB Link sockets. When using the graphical software you first start a program application for each analyzer. In a second step you have to select one of them as 'Master'. The remaining analyzer becomes a 'Slave' automatically. A proper setup provided, you then just have to click the record button in the Master application to launch the recording.

Command line tools are different by nature. The program `msb_record` has neither a dialog to choose 'Master' or 'Slave', nor a button to start the record when both analyzers are connected and ready. So you have to tell the analyzer which one is a 'Master' and which is the 'Slave' by passing an according program argument. And since there are at least two analyzers connected with your computer, you have to pass the serial number of the master or slave too.
Both commands - for the 'Master' and 'Slave' - must be started in their own shell (or DOS command window).
At first we input the record command for the 'Master' with the serial number MSB01000. We use the default settings and pipe the output directly through the formater tool. Please adapt the serial number to your own analyzer.

```
msb_record -nMSB01000 --sync-mode=master | msb_format
```

As soon as you hit the Enter key, the command prompts you with a request to start FIRST the 'Slave', THEN press Enter to begin with the record. What's that?
A synchronized recording requires the exchange of certain information between both analyzers before they could continue. For instance and most of all time relevant data.
So let's open a second command shell (or DOS box) and start the slave with:

```
msb_record -nMSB02000 --sync-mode=slave | msb_format
```

Again: The serial number is just a placeholder. Don't forget to change it to the number on your second analyzer!
Both analyzer are now in a 'ready for record' state, their red LEDs flashing alternately. And more important! The slave device is active and able to process the timing data the master will broadcast with the beginning of the record.
If anything is arranged, hit the Enter key in the master shell.
Two things are happening in the following:

1. The master passes the correct record start time to the slave[2].
2. The master gives the start command and provides the slave periodically with synchronous impulse over the link cable.

Afterwards both commands (in both shells) are interacting independently and behave as when executing a normal (not synchronous) record. You can extend the command with additional parameters or pipe constructs and write/split the output of the master and/or slave in different files.
Press Ctrl+C in each command shell to finish the according process.

[2] Remember that the master and slave must not run on the same computer.

Remote control a synchronous record

Starting two master/slave processes via command line may be sufficient for small or rarely happening tasks. Beside this the command line tools are often used in scripts to automatic certain jobs. And here the preliminary description reveals a pitfall. How can you input the Enter key requested by the master when executing the command in a batch or script file?
You can - of course - using a process pipe and redirect an Enter to the master command process. But it isn't always trivial, and luckily there exists an easier solution too: Broadcasting a remote command:
First a little batch file for Windows user:

```
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
```

And here the Linux variant:

```
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
```

The procedure is similar for both operating systems.
First we start a background process for the master (line 3) and force him to wait till we send him an according record start command by passing the `--paused` parameter.
Windows (or the DOS shell) uses the `start` command, while Linux users put the whole command into background by attaching an ending ampersand '&'. In Linux we also redirect the stderr (2) channel to /dev/null.

Background processing means: The command line is executed and detached from the script executing shell (or DOS window). The command doesn't block and the script can proceed immediately with the next instruction.
Line 4 (and 7) gives the initialization a few seconds. The DOS command shell has no particular 'sleep' command, but the `timeout` may serve as well[3].
The slave is started in line 6 and also executed as a background process.

At this point both analyzers are ready for recording and the master 'waits' for the trigger. Instead of pressing the Enter key (which isn't possible, since the master process is detached from any keyboard) we send him a remote start

[3]timeout is not part of Windows XP. An alternative way to simulate a given pause of 2s is: ping 127.0.0.1 -n 3 >nul

command in line 9.
Thereafter both commands run independent of each other. The master stores the received data in the passed output file (`-o`) master.msblog. The slave puts its data into slave.msblog.
The internal synchronization through the MSB Link connection guarantees that the recorded events in the two record files are matching with the usual precision of one microsecond.

msb_record program parameters

Call the program with:

```
msb_record [OPTION]...
```

`[OPTION]` can contain one ore more of the following program parameters. If no parameter is set the default parameters are used. The following parameters can be send to the program at start. All recorded data will be writen to the standard output by default.
To send a command to a running background process the remote parameter indication '-r' has to be entered followed by the command (start, stop, quit).

Parameter	Description
-b *rate* --baudrate=*rate*	Baudrate of the recorded connection, default is 115200.
-c --config-file=*file*	Uses the settings specified in the given config file.
-C --create-config-file	Creates a new config file `msb_tools.config` in the current directory.
-d --device=*port*	use the analyser at the given port exclusively.
--disable-dataA	Switches off the recording of all data bytes received on Port 1. Default is on.
--disable-dataB	Switches off the recording of all data bytes received on Port 2. Default is on
-h --help	Help. Output of all program parameters.
-i --initiate	Transfers the firmware to the analyzer, even it is already loaded.

--io1=*operation*	Use digital auxiliary channel IO1 (only MSB-RS485). The following values are valid: 0 : Input with pull down 1 : Output static 0 2 : Output static 1 3 : Output of the bus direction 4 : Output of the bus validness 5 : Output CHN1 validity 6 : Output CHN2 validity 7 : Output CHN3 validity 8 : Output CHN4 validity 9 : Input with pull up
--io2=*operation*	Use digital auxiliary channel IO2 (only MSB-RS485). Valid values see IO1 above.
-l list --log-signals=*list*	Specifies the signal lines which are logged by the analyzer. The lines are numbered from 1 to 8 as they are displayed in the analyzer control program (counted from left to right). For instance: -l 2,3 oder --log-signals=2,3,6,7.
-L --logic-mode	Switches the inputs to logic signal levels (only MSB-RS232). Default are RS232 signal levels.
--memory-test	Forces the analyser to executes an internal memory test.
--nice=*niceness*	The nice parameter controls the `msb_record` idle CPU time. Valid values are 0...10. A value of 0 means a nearly 100% CPU consumption, default is 1. A niceness value of 0 is only recommended in case of very fast data rates and high data flow-rate. I.e. `msb_record --nice=0`
-n *serno* --serno	Use the analyzer with the given serial number serno.
--output-buffering	Activate the internal output buffer, which increases the performance and avoids gaps in data records with high data transfer rates. Please note: With an active puffering the recorded events doesn't occurs immediately in the following tool of the command chain, for instance if you like to see all recorded events in a console window via the msb_format tool.
-o *file* --output=*file*	Output file. Default is the standard output (console).

--paused	Starts the analyser in paused state. The record begins only after the program receives a remote start command.
-p *protocol* --protocol=*protocol*	Protocol settings of the connection as combination of number of data bits (5 to 9), parity (N)one, (E)ven, (O)dd, (0)off, (1)on and stopbits (1,2). E.g. 8N1 or 7E2. Default is 8N1.
-r *Command* --remote=*command*	Remote. Sends the following command to an already running program. The following commands are supported: `quit` quits and removes the background process `start` starts or resumes a recording `stop` stops or pauses the recording
--show-serials	Shows all available serial ports.
--show-analyzers	Shows all available (connected) MSB-Analyzer.
--sync-mode=*mode*	Set the synchronization mode (autonom, master, slave) when using two or more analyzers for synchronous recordings. Default is autonom.
-t *num* --time-delay=*num*	Transfer delay. Slows the firmware transfer down by the indicated number num. 0 is no delay (default), maximum is 100.
-u --unique-file	Stores the recorded data in the current working directory in a record file with an unique name like `YYYYMMDD-HHmMMmSSs.msblog`, for instance `20110324-03h04m41s.msblog`. This parameter is especially interesting in those applications where the record should start automatically after a (re)boot of the PC.
-v --verbose	Verbose, output of additional Information. .
-V --verbose	Output of the program version.
-w *wiring* --wiring=*wiring*	Set the bus wiring (only MSB-RS485). The following values are allowed: 0 : 2-wire tapping 1 : 2-wire segment analysis 2 : 4-wire tapping 3 : 4-wire segment analysis (masterbus)

19.3 Formatted output with msb_format

The `msb_format` tool allows you to format the recorded analyser data for your own purpose. For instance if you like to see the data as CSV (comma sepa-

rated values). Without any parameter you will get a list of the occured events, each one with informations about the time, the kind of event, the data value or line state. This complies with the format specifier 'I' which is the default setting.

To specify your own output format, call the program with the format parameter `-F` or `--format=`. All following characters are seen as format definition. A space character or all 'white space' characters like Tab or Enter end the format string. If you want to define a space character as part of the output you have to quote it. How to do this is explained in chapter 19.3.
We restrict ourselves to the simple case of displaying the data bytes together with their time stamps. In every line the time in seconds and the data byte shall be listed, separated by a comma. The appropriate format string[4] is: `T,B`

We will use an example record as data source so you can try the following samples without a connected analyzer. Please keep in mind, that it will be work the same way with the `msb_record` tool.
Open a command shell (again), go into the DataView example folder (i.a. `msb-VERSION/examples/DataView`) and input:

```
type modbus-ascii.msblog | msb_format -FT,B
```

The output looks like the following:

```
...
5633.304127,48
5633.305162,70
5633.306197,57
5633.307226,13
5633.308261,10
```

The output can be changed from ASCII to binary representation. ASCII means that the decimal binary value is coded into ASCII numbers as a string (e.g.'104' = hex binary 31,30,34) while binary means the value itself which is displayed according to the ASCII table (in this example as 'h').
Binary mode makes sense if you want to write the data into a file and read in by another application. An inconvenient conversion of the ASCII representation into native program types as double or integer might be omitted.
The format identifier % activates the binary output while @ switches back to ASCII (default). The following example displays the characters in their binary value:

```
type modbus-ascii.msblog | msb_format -FT,%B@
```

Please note that in binary mode no line feed is issued. Therefore we switch back to ASCII mode after each binary data output.

```
...
5633.304127,0
5633.305162,F
5633.306197,9
5633.307226,
5633.308261,
```

[4] A list of all format identifiers can be found in the format identifier table.

Disable line feed in ASCII Mode

To make the output in ASCII mode more readable a linefeed is automatically attached after every output line. You can disable this behavior by calling the program with the parameter –disable-linefeed or by ending the format string with % (binary mode).

Output of any character

You want to insert a non-printable character or define a line feed, independent of the operating system[5].
Use the format identifier *#ddd* to define any character which shall be output instead of this identifier. To separate the output of the data bytes by a tabulator enter the decimal value of this character. e.g.

```
type modbus-ascii.msblog | msb_format -FT#009B#009S
```

Or: Separate the values by a space char (decimal 032):

```
type modbus-ascii.msblog | msb_format -FT#032#032S
```

To generate a line line feed under windows with a single linefeed character you have to use its decimal value (010). To disable the standard system dependent line feed character sequence in ASCII mode you end the string with % to switch to binary mode:

```
type modbus-ascii.msblog | msb_format -FT#009B#009S#010%
```

The character value has to be entered with three decimal digits (0 to 9). Any other input leads to an error message.

File output

You also can redirect the output to a file. Call the program with the additional parameter '-o filename'.
Only the via format string defined outputs are send to the file, no status messages or auxiliary outputs which you might have enabled through program parameter.
A simple file output is done with:

```
type modbus-ascii.msblog | msb_format -FT#009B#009S#010% -o test.log
```

Format parameters

The following identifiers are defined as format parameters. Please note that not mentioned characters are output in the same way. Exceptions are the whitespace characters that are all blanks, tabs and enter which are used to end the format string.

[5]Under linux all lines are ended with a single linefeed, while Windows uses a combination of Carridge Return/Linefeed.

Term	Meaning	Description
%	Binary Flag	Switches to the binary output for all following parameter.
@	Ascii Flag	Switches to the ASCII output mode for all following parameters.
#ddd	Character	Output of any printable or non-printable character, specified as a 3-digit decimal value. The allowed value range is 0 up to 255. e.g. the line ending carridge return character is #013
[...]	[*format*]	User defined date and time output, see table below 19.3.
a	Alteration	Shows the alteration in the signal lines or data relating to the last event. I.e. +TxD -RTS means a rising of the TxD line and a falling of RTS.
A	All line states	Shows all signal states and/or alterations in a representive text format as shown in the EventView. For instance: `-^DCD,^^TxD,^^RxD,^^DSR,-^DTR,^^CTS,^^RTS,-^RI`
b	Data-Byte	Data byte output as 8 bit value. In ASCII mode represented as 2-digit hexadecimal number with leading zeros, e.g. '41' is the character 'A', '0A' the linefeed character.
B	Data-Byte	The same as 'b' but as decimal number output in ASCII mode, for instance: '65' means the character 'A', '10' the linefeed.
d	Date/Time	Timestamp output representation in the ISO 8601 format YYYY-MM-DD HH:MM:SS (ASCII mode). Output as 32 bit value containing the seconds since the Epoch (00:00:00 UTC, January 1, 1970) in binary mode.
D	Excel Date	Excel date as days from 1.1.1900. Output as 32 bit value (binary) or decimal number (ASCII)
e	Error	Transmission error. Errors are outputed as their leading charcaters in ASCII ('F'rame, 'P'arity', 'B'reak) or as a 8 bit integer value (0:no error, 1:frame, 2:parity, 3:break) in binary mode.
i	Info	shows all informations in a more human readable form. Only for testing purpose. Don't use this parameter with others.

l	Logic-Level	Outputs the current logic state of all 8 signal lines. A set bit correlates with a logic line level of '1'. The bit order is equal to the signal lines in the control program. Bit 0 is the first (left), bit 7 the last (right) signal. The state information is written as a 8 bit value in binary mode and as a 2-digit hex number with leading zeros in ASCII mode, e.g. '7F' means all lines except for Signal 8 have a logical '1' level.
L	Logic-Level	The same as 'l'. In ASCII mode the output is written as a decimal number, e.g. '255' means all lines have a logical '1' level.
M	Milliseconds	Time stamp of the event in milli seconds as distance to 0h00 of the current day. The output is either a decimal number (ASCII) or a 32 bit value (binary).
o	dt last event	Outputs the time since the last event in seconds as a floating point number in ASCII or as a double (8 byte) value in binary mode.
O	dt same event	Outputs the time since the last same event in seconds as a floating point number in ASCII or as a double (8 byte) value in binary mode.
P	Position	Running event counter starting with the first output. The output is either a decimal number (ASCII) or a 32 bit value (binary). The event position starts with zero.
s	Data/State	Outputs either the data up to 9 bit (event type A/B) or the line states as a combination of logical and valid state. The upper 8 bits contains the logical state of each line. See specifier 'l' and 'v' and section 12.4. Data or line state are output as a 4-digit hex number like F12E, in binary a 16 bit value.
S	Source	Source or direction of the data byte. Data channel A=1, Data channel B=2. A zero means no data event. Outputed either as decimal number (ASCII) or as 8 bit value (binary).
t	event type	Output the event type as a character in ASCII mode: A (data received at port A/channel 1), B (data received at port B/channel 2), L (logic or valid line state changed) and as a 8 bit value in binary mode, range [0...2].

T	Time stamp	Microseconds precise time stamp of the event in relationship to the start of the recording. Output in seconds as floating point number with 6 digits after the decimal point (ASCII) or as 8 byte floating point number in double precision.
u	usec part	the microseconds fraction of the timestamp. You can use it to complete the normal date/time in ASCII with the left microseconds like: `-Fd+u` results to `2012-04-11 15:57:40+184935`. In binary mode the usec are stored as a 32 bit value.
v	Valid-Level	Output the current valid state of all 8 signal lines. A set bit correlates with a valid line level. The bit order is equal to the signal lines in the control program. Bit 0 is the first (left), bit 7 the last (right) signal. The state information is written as a 8 bit value in binary mode and as a 2-digit hex number with leading zeros in ASCII mode, e.g. '7F' means all lines except for Signal 8 have a valid signal level.
V	Valid-Level	The same as 'l'. In ASCII mode the output is written as a decimal number, e.g. '255' means all lines have a valid signal level.
w	Data-Word	A 9 Bit Data byte is output as a 16 Bit binary value 0 to 511. In ASCII this value is displayed as a 3-digit hexadecimal number with leading zeros, e.g. '105' or '0FE'.
W	Data-Word	Like 'w', but writes the output in ASCII mode as a decimal number. For instance: '261' means the same as the hex value '105' from above.
x*1...8*	signal level	Output the Tri-State signal level of an individual signal. The signal number 1...8 correspondences with the numbering in the control program. The signal state is output as -1, 0 or 1 in ASCII and as signed 8 bit value in binary.

User defined date and time

A string enclosed between two square brackets is interpreted as a special date/time format. With it you can output the timestamps in your very own format, according to your application. An example:

```
type modbus-ascii.msblog | msb_format -F"[%d.%m.%Y %H:%M:%S],u#115"
```

results as:

```
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
...
```

The `msb_format` tool supports the following user defined date/time format specifiers. Every parameter must start with a leading %-character. In case of using it in a Windows batch file you must double each %[6]. Otherwise the command will throwing an 'Invalid format parameter'. The reason: The Windows command interpreter uses the % for referencing the script arguments.
To avoid this, double each % in the msb_format argument. For instance: The example above:

```
type modbus-ascii.msblog | msb_format -F"[%d.%m.%Y %H:%M:%S],u#115"
```

must be changed to:

```
type modbus-ascii.msblog | msb_format -F"[%%d.%%m.%%Y %%H:%%M:%%S],u#115"
```

Please note!
The command shell (in both, Linux and Windows) uses white space characters (here the space between date and time) as a parameter separator. Therefor you have to insert the complete format string between two quoting marks.

Parameter	Description
%a	the abbreviated weekday name
%A	the full weekday name
%b	the abbreviated month
%B	the full month name
%c	the preferred date and time representation for the current locale
%d	the day of month as a decimal number [01...31]
%e	like %d, the day of the month as a decimal number, but a leading zero is replaced by a space [' 1'...'31']
%H	the hour as a decimal number, range [00...23]
%I	the hour as decimal number, range [00...12]
%j	the day of the year as deciaml number, range [001...366]
%m	the month as decimal number, range [01...12]
%M	the minutes as decimal number, range [00...59]
%p	'am' or 'pm' (according to the given time value)

[6] This exclusively applies in Windows, not Linux

%S	the seconds as decimal number, range [00...59]
%T	the time in 24-hour notation like `%H:%M:%S`
%U	The week number of the current year as a decimal number, range [00...53], starting with the first Sunday as the first day of week 01
%w	the day of the week as decimal number [0...6], Sunday being 0
%W	The week number of the current year as a decimal number, range [00...53], starting with the first Monday as the first day of week 01
%x	The preferred date representation for the current locale without the time
%X	The preferred time representation for the current locale without the date
%y	the year as decimal number without the century [00...99]
%Y	the year as decimal number including the century
%Z	the time zone, for instance CEST
%%	the literal % character

msb_format program parameters

Call the program with:

```
msb_format [OPTION]...
```

`[OPTION]` can contain one ore more of the following program parameters. If no parameter is set the default format `-Fi` is used and the output is written to the standard output channel.

Parameter	Description
-c --config-file=*file*	Uses the settings specified in the given config file.
--disable-linefeed	Suppress linefeed in ASCII mode.
-F --format=*formatstring*	Output format definition, see format table.
-h --help	Help. Output of all program parameters.
-o *file* --output=*file*	Output file. Default is the standard output (console).

-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-dte	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, output of additional Information.
-V --verbose	Output of the program version.

19.4 Filtering data output with msb_filter

The filter tool will be your first choice when you have to extract a special part, certain events or a combination of both from a former record (*.msblog).
For example: If you want to process only the transmitted data in a record without already existing signal events.
`msb_filter` reads the data from it's standard input and write the filtered data to the standard output like each other tool. The program thereby works like a real filter between the input and output channel. You can specify the kind of data or events which are passed through the filter tool by several filter parameters.

```
type recordfile.msblog | msb_filter [Filterparameter] ...
```

Please note that you have to give at least one filter parameter because the tool only passes the data which were allowed by the program arguments[7].
Without any filter parameter the program will block all data flow.

Filter data

The following tool chain filters all data bytes (received at port A and B) from the given file `modbus-ascii.msblog` in the directory `examples/DataView` and stores the result in the new record file `data-only.msblog`.

```
type modbus-ascii.msblog | msb_filter -A -B > data-only.msblog
```

You can also pass the data of the filter tool directly to the input of the formater tool `msb_format`.

```
type modbus-ascii.msblog | msb_filter -A -B | msb_format
```

[7] Linux user use the `cat` command instead of the `type` command.

Filter certain signal events

Beside the data you can also extract the recorded signal changes of every signal line. For instance if you have a record with all line changes but you are only interested in the transmitted data and the handshake signals RTS/CTS.
The selection of the passed signals is given as a comma separated list and corresponds with the `--log-signals` parameter of the `msb_record` tool.

```
type modbus-ascii.msblog | msb_filter -A -B --pass-signals=6,7 | msb_format
```

The example above extracts the signal changes of the lines 6 and 7 (in RS232 connections the signals RTS and CTS) additional to the transmitted data and forwards the result to the output formater.

Filter a given record part

The tool `msb_split` described in the next section is able to split an existing record in several smaller record files. But imagine if you only need a 'certain' part of a record file. For instance: The first or last 100000 events? Or you want to analyze only the events in a specific time range.
The filter tool offers you two further parameters to define a specifiy record part:

1 `--pass-selection=`*pos1*,*pos2*

pos1 and pos2 specifies the event position (number) of the first and last event which are passed through the filter tool. For example:

```
type modbus-ascii.msblog | msb_filter --pass-all --pass-selection=300,310
```

2 `--pass-time=`*time1*,*time2*

time1 and time2 specifies the start and end of a record part in seconds. The time is input as a floating point number with the usual micro second precision.

```
type modbus-ascii.msblog | msb_filter --pass-all --pass-time=3.04,3.05
```

> **A Non-blocking filter**
> The filter tool 'blocks' all data by default. In case of a range selection you have to pass all allowed events as parameters. Or you disable (switch off) the filtering completely with the parameter `--pass-all`.

msb_filter program parameter

Call the program with:

```
msb_filter [OPTION]...
```

`[OPTION]` can contain one ore more of the following program parameters. You have to give at least one filter rule. Without any rule the program doesn't forward any data.

Parameter	Description
-a --pass-all	passes all data and signal events.
-A --pass-dataA	passes all data received at port A (MSB-RS232) respectively Channel 1 (MSB-RS485).
-B --pass-dataB	passes all data received at port B (MSB-RS232) respectively Channel 2 (MSB-RS485).
-c --config-file=*file*	Uses the settings specified in the given config file.
-h --help	Help. Output of all program parameters.
--pass-all-signals	passes the line change events of all lines.
--pass-signals=*list*	passes the line change events of the given lines as comma separated list. The lines are numbered from 1 to 8 as they are displayed in the analyzer control program (counted from left to right). For instance: `--pass-signals=2,3,6,7`.
-s --pass-selection=*list*	pass all events in the given range defined as comma separated event number from first to last. The following example passes all recorded events with the numbers 100 to 200: `-s 100,200` or `--pass-selection=100,200`.
-t --pass-time=*list*	pass all events in the given time range in seconds as comma separated list with first time, last time. For instance: `--pass-time=1.257,10.231` passes all recorded events in the time range 1.257s until 10.231s.
-v --verbose	Verbose, output of additional information.
-V --verbose	Output of the program version.

19.5 Split records with msb_split

When recording data with the MSB-RS485 analyzer large data quantities may arise. This happens if the searched error does not occur for days and recording in Fifo mode is not wanted for any reason.
`msb_split` reads a record file from the standard input and splits it into smaller record files. You can specify the size and name of the files by use of program parameters.

Split existing record files

You have a GByte large record file and want to split it into handy parts, especially as you are interested in the last events of the recording only.

Open a console window (Windows command window) and change to the directory which contains your record file.
enter the following command:

```
type record.msblog | msb_split -n1000000
```

With `type` the record file is sent to the standard output and fed into the standard input of the `msb_split` program by the pipe operator '|'.

Linux users use the `cat` command instead of the `type` program.

Depending on the size of the output file *record.msblog* `msb_split` divides them into multiple $1000000 * 24 + 3072$ Byte files Each file (with exception of the last one) contains 1,000,000 events, each 24 bytes long plus a header of 3072 bytes. In the current directory a number of new msblog files are generated in the form:

xaa.msblog, xab.msblog, xac.mbslog, ...

You can examine every file individually with the MSB-RS485 analyzer software by loading them into the program or double click onto it.
By default the program enumerates all files alphabetically with a preceding 'x'. You can change this behavior by adding a respective parameter. For a 3-digit, numerical enumeration use the parameters `-a` and `-d` (see 19.6).

```
type record.msblog | msb_split -a3 -d -n1000000
```

As result you get: x000.msblog, x001.msblog...

You can substitute the preceding 'x' for your prefix by appending it as last parameter to the command line.

```
type record.msblog | msb_split -a3 -d -n1000000 Test
```

The resulting files now begin with: Test000.msblog, Test001.msblog, ...

This kind of enumeration does not mention the important time range of the single files. Alternative to the alphabetical or numerical naming you can also chose date and time of the first event for the name of the split files. The parameter is `-D`.

```
type record.msblog | msb_split -D -n1000000 Projekt-
```

The generated files have the following meaning:

```
Projekt-20110510_15h53m24s.msblog
Projekt-20110510_15h58m31s.msblog
Projekt-20110510_16h02m10s.msblog
...
```

If you don't like any prefix, just append an 'empty' string as the last parameter

(PREFIX):

```
type record.msblog | msb_split -D -n1000000 ""
```

With it you will get:

```
20110510_15h53m24s.msblog
20110510_15h58m31s.msblog
20110510_16h02m10s.msblog
...
```

Splitting the current recording from msb_record

As the `msb_split` program reads its data from the standard input you can use the output of the `msb_record` tool as data source to directly divide the recorded events into small portions. This may make sense if you plan long and large recordings to examine them later.

```
msb_record -b115200 -p8N1 | msb_split -a4 -d -n1000000
```

msb_split Program Parameter

Call the program with: `msb_split [OPTION]... [PREFIX]`

`[OPTION]` can contain one ore more of the following program parameters. If no parameter is set the default parameters are used. `[PREFIX]` is an optional and freely selectable character string which precedes the file name. The default is the character 'x'.
All parameters can be used in the short form (a character with a leading '-', first row) or in the long form (second row).

Parameter	Desription
-a *length*, --suffix-length=*length*	Number of usable characters for the enumerating suffix. Default is 2 characters.
-h, --help	Output of all program parameters.
-c --config-file=*file*	Uses the settings specified in the given config file.
-d, --numeric-suffix	The files are names numerically, the default is alphabetically.
-D, --date-time-suffix	The files names are extended with the date and time of the first occurred event in the format `YYYYMMDD_HHhMMmSSs`.
-n, --number=*quantity*	Quantity of the events per file. Each event occupies 24 bytes.
-v, --verbose	Output of additional information.

-V, --version	Output of the program version.

19.6 Trigger a record with msb_trigger

Long-term records in conjunction with the command line tools often serves the purpose to find a rarely occuring event when the communication goes wrong. Such an event can be a suddenly failing device only indicated by an invalid telegram or a wrong telegram content.
It is obvious that such an event is not easily detectable by simply looking for a given data sequence. Here we have to take into account the used protocol. Just consider a bus participant in a Modbus communication which responses unexpectedly with an error (and only once in hours or days). Although the error is a two byte sequence (address byte, followed by the error function number), that sequence may occur more than once in other telegram payloads. The trigger condition is only true, when this search pattern is identical with the first two bytes of a (Modbus RTU) telegram. And this means, the trigger condition must be able to detect the start (and end) of every telegram.
Since there are a lot of different protocols out in the world, the `msb_trigger` tool uses the same approach as the `ProtocolView` and provides an integrated Lua script interpreter to let you formulate not only protocol dependent trigger conditions but also very special conditions you otherwise have no chance to find.
The `msb_trigger` program follows the rules of all other command line tools. You can use it to trigger the output of an active record with:

```
msb_record | msb_trigger script.lua > record.msblog
```

Or you can output a special part (specified by the trigger condition) of an already made record:

```
type record.msblog | msb_trigger script.lua > result.msblog
```

(Linux user use the `cat` command instead of `type`).

Please note!
To simplify the command line we forego any additional `msb_record` parameters.
You can also output the result of the `msb_trigger` to other tools like the formater (`msb_format`) or splitter (`msb_split`).
The file `script.lua` specifies the trigger condition. It works similar to the `split()` function in the `ProtocolView` and we will discuss it in detail in the following.

Define a trigger condition

By default the `msb_trigger` tool forwards all data events read from the standard input (provided by the `msb_record` or an existing record) to the Lua function `trigger`.

```
function trigger( data, intval, dir, alter )
   -- return true if the trigger condition occurred
end
```

The `trigger` function is called separately for each data direction, so you don't have to worry about the data belonging.
Beside the raw data (9-bit) the program passes the time distance to the former data byte, the direction and if a change (alternation) in the direction has occurred. Below is the list of all parameters:

1. data → the current data byte (up to 9 bits)
2. interval → (short intval), the time distance to the former byte in seconds (with microsecond resolution)
3. direction → (short dir), the direction or source of the current data event. 1=Data A, 2=Data B.
4. alternation → (short alter), true when the direction has changed

You can rename the parameter for your own purpose but don't change the order of the parameter! It's also allowed to skip unused parameter from the right. Let's take a look for a simple example. The code below triggers when in a Modbus ASCII transmission a telegram end sequence was incomplete and instead of CR LF (carriage return, line feed) only a CR occurred.

```
lastByte=-1
function trigger( data )
   if lastByte == 0x0D and data ~= 0x0A then
      -- trigger
      return true
   end
   lastByte = data
   return false
end
```

In addition to the passed parameters above exists a global event object which covers the actual event and provides additional information like the time stamp or the current signal levels.
You can access these information by using the `event` module as described in the ProtocolView chapter 13.7. The global event becomes especially useful if you need to trigger not for a data but a signal line condition. To give you an idea about this, here we trigger for a falling edge of the DTR signal.

```
-- 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
   last_dtr = dtr
   return false
end
```

Since `msb_trigger` by default only processes data events, you have to switch

the event type read by the tool from data to signal.

```
msb_record | msb_trigger --trigger-source=signal script.lua > record.msblog
```

Conditional start of a record with pre and post-trigger

As said before - this is the main purpose of the `msb_trigger` tool. Instead of examine a huge amount of recorded data for a given event you better pipe all data logged by the analyzer to the trigger program.
`msb_trigger` allows you to specify a number of pre- and post-trigger events (see section program parameters 19.6). This is especially important if you want to see the transmitted data before an event occurred and to limit the recorded data after the trigger happened. Lets say you need to know the communication around a Modbus RTU checksum failure.
The trigger script below splits the incoming data stream into single Modbus RTU telegrams by checking the idle time between the received bytes in line 4. Modbus RTU specifies an idle time (or transmission pause) of 3.5 byte for a telegram delimiter, which means the time for sending 3.5 byte. In case of a positive idle time the global variable `seq` (line 2) represents the complete telegram. The Modbus RTU protocol uses a 2 byte CRC16 checksum as the two last bytes of the telegram. The script first checks for a telegram length of at least to bytes in line 7, then compares the received checksum bytes with the calculated checksum. It returns true (trigger condition detected) if the comparison doesn't match.

```
1  -- represents the actual telegram
2  seq = ""
3  function trigger( data, intval, dir, alter )
4     if intval > protocol.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
```

You will find the trigger script in the `examples/API` folder.
Modbus RTU telegrams are limited to a maximum length of 256 byte. We want to record at least 10 telegrams before and after the reception of the telegram with the invalid checksum, which gives us a pre and post-trigger count of 2560.

```
msb_record | msb_trigger --pre-triger=2560 --post-trigger=2560 script.lua > record.msblog
```

Conditional output of an existing record file

`msb_trigger` not only serves as a trigger of an active record. With it you can also scan an already existing record for a given event and produce a new record for a later analysis with the analyzer software.
The program call of the `msb_trigger` is identical. You just replace the tool `msb_record` as the data source with the output of the given record file. For instance:
`type record.msblog` for Windows user or `cat record.msblog` for users running Linux. In case of the former example (under Windows):

```
type modbus-rtu.msblog | msb_trigger --pre-triger=2560 ↩
 --post-trigger=2560 script.lua > record.msblog
```

Scan a record file for certain events

Imagine you just want to know if there are any checksum errors (or - of course - other transmission or telegram issues). You don't like to produce a new record file with the given event. An output with the information which telegrams (time and/or number) cause the wrong checksum should be satisfied.
The `msb_trigger` tool provides you with a special parameter `--debug` which not only helps debugging your trigger script. It also serves as a switch to suppress the output of the recorded events when a trigger condition occurs. The latter is important to avoid a mixture of printed information and binary event sequences.
Since the trigger conditions are coded in Lua, it is very easy to output any information directly from within the script by the Lua `print` function.
Using our Modbus RTU checksum example again, we will now looking for telegrams with an invalid checksum and printing the time when the telegram occurred together with the transmitted (wrong) and expected (valid) CRC16.

```
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 > protocol.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
```

Please note! You don't want to stop the evaluation of the piped record by the very first trigger condition. Therefore the trigger function MUST return false (line 26). Otherwise the `msb_trigger` tool exists without any output.
You can test the script above by yourself. Just open a shell (Linux) or command window (Windows) in the `examples/API` folder of the installation directory and input:

```
type Modbus-RTU-wrong-checksum.msblog | msb_trigger ↩
--debug scan-modbus-wrong-checksum.lua
```

This will give you the following output:

```
727.232679      is:982E, must:972E
904.113558      is:50A5, must:50A6
949.962685      is:528C, must:508C
```

There are three telegrams with an invalid checksum in the record at the displayed time. The transmitted and wrong CRC16 checksum is output as 'is', the expected (calculated) checksum as 'must'. To verify, just load the record in your analyzer software.

One script for scan and trigger

Up to here you have learned the following applications:

1. How to trigger an active recording
2. How to extract the events around a trigger condition
3. How to scan a record for certain information

Especially the latter application uses the built-in debug feature of the `msb_trigger` tool. Unfortunately this code is not compatible when triggering an active record and so far you have to write two scripts: One for trigger or extract a recording and a different one to scan a record for special details.
Even if the trigger scripts are seldom very complicated it is nevertheless bothering to work with two kinds of code.

In this section we will conclude the description of the `msb_trigger` program by showing you how to bypass that distinction.

Just remember the purpose between a trigger script and a script which has to printout certain information. A script intended for trigger a record (or extract part of a record) always has to return **true** in the `trigger` function. And it never should output anything, since this would mix-up the resulting record file.
In contrast consider a script scanning a record e.g. for invalid checksums. As described earlier, the result of the `trigger` function must be **false** and you **can use** the Lua print function to output valued information.
In a nutshell, trigger scripts create new record files whereas scan scripts produce anything but NO valid analyzer records!

To solve this contradiction we need to know when a script is called for triggering a record or called with the `--debug` argument indicating a scanning purpose.

Luckily the `msb_trigger` tool defines the internal global variable `DEBUG` which reflects the `--debug` argument. With it it is easy to cover code meant either for trigger or scan. The script below again pick-ups our Modbus RTU example.

```
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 > protocol.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
```

The important line is 14. Here we use the `DEBUG` variable to check if the script is called with the `--debug` parameter indicating a scan. If so, we output the time of the telegram with the invalid checksum as well as the expected and detected CRC16 checksum values (line 15). The code proceeds and returns false in line 29.
Otherwise we just return true for a detected trigger condition in line 18.
You will find this combined script as like the other examples in the `examples/API` folder.

Provided Lua modules

The `msb_trigger` tool shares several modules with the ProtocolView. These are:

- **base16 module :** Encoding and decoding functions for base16 sequences (i.e. used in Modbus ASCII and Intel SRecord). See ProtocolView, section 13.7.
- **checksum module :** Contains checksum algorithms for Modbus RTU (CRC16), Modbus ASCII (LRC), BACNet (CRC8 and CRC16), DNP3 and CRC16 CCITT (Kermit). See ProtocolView, section 13.7.
- **event module :** The event module is only available in the trigger function and gives you access to additional information of the current data event. See ProtocolView, section13.7.

- **protocol module :** Returns information about the current baudrate, data bits, parity and stopbits. See ProtocolView, section 13.7.
- **shared module :** As like the ProtocolView also the trigger tool uses two separated Lua interpreters for both data directions. The reasons for this implementation detail and how to sometimes have to share variables between them are described in the ProtocolView chapter, section 13.7.

You can use the listed modules above in an identical way as described in the according examples of the ProtocolView chapter. All modules can be called from any location in your trigger script, except for the `event` module, which is only accessible from within the `trigger` function.

msb_trigger Program Parameter

Call the program with: `msb_trigger [OPTION]... trigger-script`

`[OPTION]` can contain one ore more of the following program parameters. If no parameter is set the default parameters are used. `trigger-script` is a Lua script specifying the trigger condition.
Some parameters can be used in the short form (a character with a leading '-', first row) or in the long form (second row).

Parameter	Desription
-c --config-file=*file*	Uses the settings specified in the given config file.
-h, --help	Output of all program parameters.
--debug	Enables the debug mode.
--pre-trigger=*events*	Specifies the number of events BEFORE the trigger point which are output when the trigger condition occurs. The default is 4096 events (data and/or line state events).
--post-trigger=*events*	Defines the number of events output AFTER the trigger condition occurred. The default is infinity, means that the output runs until the program is stopped.
--trigger-source=*source*	The kind of events passed through the trigger script. Default is `data` but you can also trigger on certain line state conditions by passing `signal` as the trigger source.
-v, --verbose	Output of additional information.
-V, --version	Output of the program version.

19.7 One config file for all

As yet we either used the default settings of the several tools or we handled our specifications to the respective tools via program parameters. Depending on the count of arguments this approach will lead to complex and - perhaps - buggy command lines.

Therefore all tools are also manageable by one configuration file which is given to the first `msb_record` program as a parameter. `msb_record` ensures that all further programs in the command (tool) chain receive the settings in this file[8].

A configuration file isn't part of the analyzer software but you can always create a new one just by the following command:

```
msb_record -C
```

respectively

```
msb_record --create-config-file
```

As a result the file `msb_tools.config` is written in the current directory. Open the file with your favorite editor (Windows user can use notepad, Linux users may choose between gedit, kate, or - of course vi, emacs, ...).
The configuration file is well documented. You can simply adapt the parameters to your application and save the file under a new name. The latter makes sense because another call of `msb_record -C` overwrites the file without a warning.

You can - of course - create several individual configuration files, one for each application. File name and file extension are of no importance.
As soon as you specify a configuration file to the `msb_record` program, all further tools in the command chain will take the settings in that file into account. For example:

```
msb_record -c meine-config-datei | msb_format
```

respectively

```
msb_record --config-file meine-config-datei | msb_format
```

Perhaps you are wondering about all the examples above which were using the output of a record file via `type` or `cat` as data source instead of the `msb_record` tool?

In this case you can specify the configuration file for each tool individually. Each analyzer tool understands the parameter `--config-file` or the simple variant `-c`. You do not need to change the configuration file. Just call the relating tool with the required file.

```
type examples\DataView\9bit.msblog | msb_format --config-file datei
```

[8] This only works of course for the analyzer tools.

A

ASCII character table

ASCII (American Standard Code for Information Interchange) is a form for the character coding, which, coming from teletype machines, now is established as the standard code for character representation.

The first 32 characters of the ASCII code (hex 00 to 1F) are non printable signs, reserved for control purposes. The main control characters are line feed or carriage return. They are used with devices which need the ASCII code for control purposes as printer or terminals. Their definiton is caused for historic reasons.

Code hex 20 is the blank and hex 7F is a special character which is used for deleting.

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

The upper table regards only 7 bits per byte, the first 128 characters. Extentions of the ASCII code use the next 128 characters for national language codings or graphical signs. They are very different in usage. So we will limit the description to the standard 7 bit version.

B

Baudrate measuring

The MSB-RS485 analyzer allows the setting and measuring of any baudrate in the wide range from 1 Baud up to 1 MBaud with the unique precision better than 0.1%

The measuring is performed eight times per second, thereby measuring and averaging the width of singular 0 or 1 bits. The more bits are available in the measuring frame of 125 ms the more precise the measuring becomes. A higher data quantity will lead to more precise and stable measuring values.
The analyzer allows three kinds of baudrate measuring.

1 Automodus **(UART A + B)**

2 CH1 **(UART A)**

3 CH2 (CH3) **(UART B)**

In the auto mode either the data of the internal UART A (CH1) or UART B (CH2 respective CH3) is used for measuring, depending on which channel delivers the first data bit at the start of a 125 ms measuring frame. Therefore the measured baudrate can vary if both channels use different clock generators with slightly different baudrates.
This mode is appropriate especially to detect different baudrates on the send and receive line.

To measure the baudrate of a certain channel the input must be explicitly set. This is done in the settings dialog of the controll program.

The status window shows 2 baudrates, the set and the measured one. The latter with its percental deviation to the set rate. Deviations over 50% are indicated by Out!.

$$dBaud = 100 * \frac{Baudmete - Baudset}{Baudset}$$

A negative value indicates a lower baudrate, positive values indicates a higher baudrate (than the set one). Many bit errors can be explained from incorrect generated baudrates. The following can be taken as a rough guide value:
Deviations of a maximum of $\pm 3\%$ can be accepted and compensated, higher deviations should be avoided.

Baudrate tolerance

Avoid more than 3% deviation in the baudrate generation. This will result in bit errors.

Because of insufficient slew rates of the EIA-422/485 sender of a examined transmission line the measuring value can be to high for high transmission rates. This could also be a hint, that the EIA-422/485 drivers are not correct for the used baud rate when the baud rate is higher than allowed baudrate for the EIA-422/485 driver.

C Colors

The MSB-RS485 analyzer software allows you to enter own color definitions at different places. A selection of predefined colors can be found here.

The input of color values can be done either in form of a color name (the following tables show an overview of the pre-defined color names) or by entering a RGB (red green blue) value as a hexadecimal number.
Please note, that the names are generally in english, even if you use a German software version. Some colors consist of compound words as 'indian red'. The blank between is part of the name and has to be entered explicitely.
In the following list you find besides the color names also the RGB value, which can be entered alternatively. RGB values can be entered in short or in long form. The number of digits (3 or 6) determine the used format.

C.1 RGB short form

The short form #RGB reduces each color part to a value between 0 and 15 decimal (0 to F hexadecimal) where R,B,G is represented by this value 0 to F. That means that each part can be defined in steps of 1/15 of 100%. For instance red is #F00 and white is #FFF.
For each part is valid that for 0 it is not contained and for F it is fully contained in the composite color. In the short form 16 x 16 x 16 = 4096 colors are possible.

C.2 RGB long form

The long form #RRGGBB extends the value range for the single color parts from 16 to 256, which simply is a higher resolution for each color. The resulting color range is 256 x 256 x 256 = 16777216 possible colors.

C.3 Predefined color names

The predefined color names are a selection from a list of standard colors used in web site displays. Besides the extended colors an input of 'green' should be much more intuitive than #0F0 of #00FF00. The basic colors like 'black', 'white', 'red' ... are easy to memorize.

Grey colors

Name/Value	Color	Name/Value	Color
black #000000		dim grey #696969	
dark grey #a9a9a9		grey #bebebe	
light grey #d3d3d3		white #fffff	

Basic colors

Name/Value	Color	Name/Value	Color
blue #0000ff		green #00ff00	
red #ff0000		cyan #00ffff	
magenta #ff00ff		yellow #ffff00	

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	
cornflower blue #6495ed		navy #23238e	
medium blue #0000cd		dark slate blue #483d8b	

Name/Value	Color	Name/Value	Color
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

The driver, necessary for the operation of the analyzer, is automatically installed and the device is detected at start of the program. If this is not the case or other problems arise you will find some solutions here.

D.1 Windows doesn't found the analyzer (Part I)

Description
The Analyzer was connected correctly with your system and both red state LEDs are on. After starting the software, the startup dialog still doesn't detect it.

Solution
At first remove all USB devices except Mouse and keyboard from your PC.
Connect The analyzer to your PC and look up the assigned COM port in the device manager. Usually a new COM Port should occur after attaching the analyzer.
If this is the case open a command shell (DOS Box) and change to the installation program. Usually this is:

```
C:\Programme\msb-4.6.0
```

Start the software manually by entering the following:

```
msb_serv.exe -pCOMxx -nMSByyyyy --force
```

COMxx is the port number as displayed in the design manager, MSByyyyy means the serial number of the analyzer. The number sticks on the bottom of the housing. For instance:

```
msb_serv.exe -pCOM12 -nMSB01234 --force
```

If the software starts with this parameter you can add this program parameter to the MSB start icon under preferences. You will find a description in the chapter 7.13 of the manual.

D.2 Windows doesn't found the analyzer (Part II)

Description
The analyzer is correctly connected to your system and both LEDs light red. But the device manager does not display a virtual COM port number. (See also D.1).

Solution
Reinstall the drivers which are necessary for the operation of the analyzer. You will find the driver on your installation CD in the directory `driver`

For the installation use the executable driver setup program (recognizable by the ending `.exe`)

Please help us with conflicting devices

The firmware loader uses the information which is collected in the USB enumeration procedure to detect all serial ports connected to a MSB-analyzer.
Because of the manifold combinations of existing USB devices and drivers we can not exclude the possibility that in rare cases the program does not detect the analyzer correctly. To regard these situations in the further program development we need your active help.
Open a command shell (DOS box) and change to the installation directory. Enter the following command:

```
msb_serv.exe --verbose
```

The `--verbose` parameter forces the program to store a report file (AnalyzerScan.txt) with information concerning the internal analyzer detection on your desktop. Just send this file afterwards to support@iftools.com.

D.3 MSB-RS485 Device quit working unexpectedly

Description
The MSB-RS485 device loses its connection and stops working unexpectedly. The device may appear to work fine when Windows first starts up but later stops working.

Solution
Use the steps (below) to resolve this issue.
Disable power management on the USB hub.
To preserve power, Microsoft Window XP tries to disable USB when a device is not used. Under certain circumstances this function does not work and may cause USB devices to fail to respond when called. To resolve this issue, disable power management on the USB hub by doing the following:

1. **Click Start, and right-click My Computer.**
2. **Click Properties, then click Hardware.**
3. **Click Device Manager.**
4. **Double-click the Universal Serial Bus Controllers branch to expand it.**
5. **Right-click USB Root Hub, and then click Properties.**

6 **Click Power Management.**

7 **Deselect Allow the computer to turn off this device to save power.**

8 **Repeat Steps 5 through 7 for each USB Root Hub.**

9 **Click OK, and close Device Manager.**

10 **Reboot you system**

D.4 Other problem

Description
Your problem isn't listed here.

Solution
In case of problems do not hesitate to send us a mail under: support@iftools.com
Please do not forget to inform us about your software and system (Windows version, Service Pack, 32/64 Bit System) as also a detail description of your problem.

E

Linux Trouble-Shooting

New Linux kernels innately contain all what is necessary for the operation of the analyzer. Nevertheless you can be trapped by the lot of different Linux variants and their differing implementations which may make the correct functioning difficult. How you can bypass the problems are explained in this chapter.

E.1 Linux doesn't found the analyzer (Part I)

Description
The Analyzer was connected correctly with your system and both red state LEDs are on. After starting the software, the startup dialog reports an error (MSB not found).

Solution
Within Linux you have to be member of the group for accessing the `/dev/ttyUSBx` device. Open a console and type:

```
~$ ls -l /dev/ttyUSB?
crw-rw---- 1 root dialout 188, 0 2009-08-26 14:47 /dev/ttyUSB0
```

On Debian the group is dialout, SuSE uses the uucp group. Make your user account member of the group and try again. Y can check your group membership with:

```
~$ groups
username dialout cdrom floppy audio video plugdev
```

Please note! You have to logout and new login first until the changes are available. A reboot is not necessary.

E.2 Linux doesn't found the analyzer (Part II)

Description
You have the correct permissions for accessing the `/dev/ttyUSBx` device. Anyhow the analyzer wasn't detected by the software.

Solution
Be sure, that you don't have an installed Braille driver. Disconnect and connect the MSB-RS485 again. Open a console and input `dmesg`.
If the output displays something like this:

```
~$ 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
```

a Braille driver is part of your system. If you have no need for a Braille device, please remove it from your system. On Debian

```
~# apt-get remove brltty
```

should be work.

E.3 Linux doesn't found the analyzer (Part III)

Description
You have disconnect all serial USB devices (except of mouse and keyboard if used) from your computer. Unfortunatelly there doesn't exist any entry like `/dev/ttyUSB0`.

Solution
Open a console window and input the following command:

```
~$ dmesg
usb 1-1: new full speed USB device using uhci_hcd and address 5
ftdi_sio 1-1:1.0: FTDI USB Serial Device converter detected
drivers/usb/serial/ftdi_sio.c: Detected FT232BM
usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB0
```

The output of the command `dmesg` shows you, if the kernel recognize the analyser as FTDI USB serial device and if the associated kernel module `ftdi_sio` was loaded by the kernel.
If not, your kernel doesn't support any USB or the FTDI support isn't part of the kernel. If you have compiled your kernel by yourself, please check, if you have enabled FTDI devices as module or part of the kernel itself. Take a look in your kernel configuration file (you will find it at `/usr/src/linux/.config`). There must be a line like this:

```
CONFIG_USB_SERIAL_FTDI_SIO=m
```

If not, you have to recompile your kernel with the line above. Also without the `ftdi_sio` module your kernel should register the connected analyser as a USB device if the USB support working correctly. You can verify this with the following command line:

```
~$ cat /proc/bus/usb/devices
T: Bus=01 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 5 Spd=12 MxCh= 0
D: Ver= 1.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs= 1
P: Vendor=0403 ProdID=6001 Rev= 4.00
S: Manufacturer=IFTOOLS
S: Product=MSB-B
S: SerialNumber=MSB00001
C:* #Ifs= 1 Cfg#= 1 Atr=01 MxPwr= 0mA
I: If#= 0 Alt= 0 #EPs= 2 Cls=ff(vend.) Sub=ff Prot=ff Driver=ftdi_sio
E: Ad=81(I) Atr=02(Bulk) MxPS= 64 Ivl=0ms
E: Ad=02(O) Atr=02(Bulk) MxPS= 64 Ivl=0ms
```

The example above shows a analyser MSB-RS485 (MSB-B) from IFTOOLS with the serial number MSB00001.
You should get this output in any case or your kernel have some trouble with USB devices in general. Can you confirm, that your Linux system works with other USB devices? Please contact us with detailed information about your system, see section E.9.

Please help us with conflicting devices

The firmware loader uses the information which is collected in the USB enumeration procedure to detect all serial ports connected to a MSB-analyzer.
Because of the manifold combinations of existing USB devices and drivers we can not exclude the possibility that in rare cases the program does not detect the analyzer correctly. To regard these situations in the further program development we need your active help.
Open a command shell (terminal) again and change to the installation directory. Enter the following command:

```
~$ ./msb_serv --verbose
```

The `--verbose` parameter forces the program to store a report file (AnalyzerScan.txt) with information concerning the internal analyzer detection on your desktop. Just send this file afterwards to support@iftools.com.

E.4 Recording doesn't work

Description
The analyzer seems to be correctly detected and the firmware was loaded. After starting of a recording session no data are recorded.

Solution
The analyzer indicates its correct firmware initialization by an alternating blinking of both red LEDs. If both LEDs are permanently on after starting the software, the firmware was not correctly transferred into the device. The reason is the high firmware transfer rate which can cause errors (very rare) in some driver implementations.
You can reduce the transfer rate for the firmware initialization via program parameter. Call the analyzer software with the following additional parameter from the installation directory:

```
~$ ./msb_serv -r 20
```

The value of 20 correspondents to the reduction of the transfer rate. Allowed values are 0 (no reduction) up to 50.
As soon as the analyzer firmware was correctly loaded you can add the used parameter to your msb-4.6.0 desktop icon.

E.5 Segmentation fault during installation

Description
The installation program crashes with a segmentation error. This happens mainly within a kde4 enviroment.

Solution
Open a console, make the installation file as executable

```
~$ chmod +x msb-2.4.0-linux-installer.bin
```

and start the program with the –mode xwindow like this:

```
~$ ./installation-file-linux.bin --mode text
```

E.6 The help menu Help→Content F1 doesn't work

Description
You pressed F1 or select the Help→Content F1, but nothing happens or you get error message. If you open the manuals directly, you can read the operation manual.

Solution
The MSB-RS485 software needs an additional PDF viewer for displaying the according manual pages. Within Linux this is the `xpdf` program, because of it's features to reference certain sections via so called named destinations.
You have to install this program if it isn't part of your system. For reading or printing the manual you can always use your favorite PDF viewer like `kpdf`, `okular` or GNOME application of course.

E.7 The software doesn't run within a 64 bit Linux (Part I)

Description
The analyzer software doesn't start via click on the desktop icon or crashed with a segfault.

Solution
The analyzer software is build as a 32 bit application. Therefore you have to install the according 32 bit shared libraries (ia32-libs) to use on amd64 and ia64 systems.On debian based systems just install the ia32-libs with:

```
~$ sudo apt-get install ia32-libs
```

You can - of course - use your package manager or software center.

E.8 The software doesn't run within a 64 bit Linux (Part II)

Description
You have installed the ia32-libs but if you start the software only a small window appears (Ubuntu 11.04 or higher) or nothing happened.

Solution
Ubuntu's method of creating the 32bit versions of these packages for the 64bit ubuntu has a bug caused by a hard coded path where the gdk-pixbuf library looks for several modules loaded at runtime. We recomment the following workaround:
Open a terminal window and input:

```
gedit ~/Desktop/msb-4.6.0.desktop
```

Add the additional line: `Path=/home/jb/msb-4.6.0`.

Replace the line: `Exec=/home/jb/msb-4.6.0/msb_serv`
with:

```
Exec=env GDK_PIXBUF_MODULE_FILE=/usr/lib32/gdk-pixbuf-2.0/2.10.0/loaders.cache msb_serv
```

Now the changed desktop file has to look like:

```
[Desktop Entry]
Type=Application
Version=0.9.4
Name=msb-4.6.0
Comment=Starts the msb program
Icon=/home/jb/msb-4.6.0/msb-48.png
Path=/home/jb/msb-4.6.0
Exec=env GDK_PIXBUF_MODULE_FILE=/usr/lib32/gdk-pixbuf-2.0/2.10.0/loaders.cache msb_serv
Terminal=false
Name[en_US]=msb-4.6.0
```

Save the file and start the software by click on the icon.

E.9 Other problem

Description
Your problem isn't listed here.

Solution
We are very much interrested that our software can be used under Linux without problems. Because of the large numbers of different Linux distributions this is not always easy. Therefore:
In case of problems do not hesitate to send us a mail under: support@iftools.com
Please do not forget to inform us about your software and kernel version, 32/64 Bit system, Linux distribution, desktop environment and a detail description of your problem.

Glossar

Notation	Description
CSV	Comma Separated Values, Comma Separated Values, text file format in which the content of single data sets are stored in independent lines, separated by commas. 55
ETX	End of Text, in the ASCII character set defined as hex 0x03. ETX marks the end of a message or datagram. 77
Firmware	Firmware describes the software contained in an electronic device which is responsible for its function. Firmware can be a fixed and unchangeable part of the hardware or can be loaded into the device before the first start. 27
FLEXUART	An IFTOOLS in-house developed UART allows high-precise setting and measuring of any, even non standard baud rates in the range from 1 Baud up to 1MBaud with 0.1% accuracy. 28
Full-Duplex	Shortened as HD or HDX. A full-duplex, or sometimes double-duplex system, allows communication in both directions, and, unlike half-duplex, allows this to happen simultaneously. 1
Half-Duplex	Shortened as HD, or HDX. A half-duplex system provides for communications in both directions, but only one direction at a time (not simultaneously). 1
Lua	Lua is a dynamically typed language intended for use as an extension or scripting language. By including only a minimum set of data types, Lua attempts to strike a balance between power and size. 52
Multi-Master	Bus nodes which are allowed to initiate a data transfer with other bus nodes are denote as active node or master (otherwise they are denoted as passive nodes or slaves). A bus with several masters is called a Multi-Master bus. 1
Multidrop	A Communication based on the Master-Slave principle whereby a master (sender) can speak to several receivers without expecting any answer (single direction). 1

Notation	Description
Record depth	The number of maximum events or samples which are contained in the signal recording is called recording or storage depth and depends on the available storage medium. 145
RTF	A document file format developed by Microsoft for cross-platform document interchange. 54
RTS/CTS Handshake	A hardware flow control implemented by correspondent levels on the RTS or CTS lines. The RTS/CTS lines of both participants are cross-connected . By setting the RTS line to logical 1 the receiver requests a stop of the data transmission. Only a few UARTS handle the flow control in hardware, so that the software driver have to react fast to recognize the state and stop the transmission. 1
STX	Start of Text, in the ASCII character set defined as hex 0x02. STX marks the start of a message or datagram. 77
Timebase	The time duration which corresponds to a grid (10 pixel). The lower the time base the higher the time resolution of the display. The lowest time base in the SignalView is 500ns, which corresponds to 50 ns per pixel. 147
UART	Universal Asynchronous Receiver Transmitter. Electronic element to send or receive data over a serial data line. 2, 20

Index