



Manual

MSB-RS485

Version 7.0.2

www.iftools.com

Inhaltsverzeichnis

1	Analysis of RS422/485 Bus systems	1
1.1	Special serial driver software	3
1.2	Bus-tap 2-Wire Bus	3
1.3	Double bus-tap 4-Wire bus	4
1.4	Sampling	4
2	MSB-RS485 Analyzer	5
2.1	Advantages of a hardware solution	5
2.2	Innovative software concept	6
2.3	Application fields	7
3	Features & Benefits	9
4	Specifications	11
5	Program Installation	15
5.1	Installation under Windows	15
5.2	Installation under Linux	16
5.2.1	Manual installation under Linux	16
5.2.2	Installation for all users	17
5.3	Program Updates	18
6	Connection of the Analyzer	19
6.1	Definition of the Signal lines	19
6.2	Internal Signal Processing	20
6.3	Digital In/Outputs	21
6.4	Bus Termination and Tapping	21
6.5	Tapping 2-wire system	22
6.6	Segment Analysis 2-wire system	22
6.7	Tapping 4-wire system	23
6.8	Segment Analyse 4-wire system	24
6.9	Signal assignment	24
6.10	Lightment elements LEDs	25
6.10.1	Green LEDs	26
6.10.2	Red LEDs	26
7	Program start	27
7.1	User Interface	28
7.2	Configure a record	29
7.2.1	Transmission setup	29
7.2.2	Bus wiring	31
7.2.3	Signals	32
7.2.4	Record mode	33
7.2.5	Autosave	34
7.2.6	General	35
7.3	Start a record	35
7.4	Status display	35
7.4.1	Display I	36

INHALTSVERZEICHNIS

7.4.2	Display II	36
7.4.3	Display III	36
7.5	The analysis tools	37
7.6	Save a recording	37
7.7	Save a session as a project	38
7.8	Open an earlier recording	38
7.9	Open an earlier session (project)	39
7.10	Last opened recordings and projects	39
7.11	Drag and drop	39
7.12	Connecting multiple analysers	39
7.13	Automatic start after computer boot	40
7.13.1	Activate the autostart feature under Windows	41
7.13.2	Activate the autostart feature under Linux	41
7.14	Short commands	41
7.15	Additional program arguments	42
7.16	Special program parameters	43
8	The MultiView design	45
8.1	Synchronization	45
8.1.1	Follow (autoscroll)	46
8.1.2	Locked (fixed)	46
8.1.3	Linked	46
8.2	Views (displays)	46
8.2.1	Virtual Ledtester	47
8.2.2	DataView - Data Monitor	47
8.2.3	EventView - Event Monitor	47
8.2.4	ProtocolView - Protocol Monitor	47
8.2.5	SignalView - Signal Monitor	47
8.2.6	Regions	48
8.3	Copy Views	48
8.4	Default settings for Views	48
9	Session management	49
9.1	Projects	49
9.2	Store and reload projects	50
9.3	Automatic storing of a session	50
10	The virtual Ledtester	53
10.1	The toolbar	54
11	The Data View	55
11.1	User Interface	55
11.1.1	Display of data errors	56
11.1.2	Synchronizing	57
11.1.3	Data channel selection	58
11.1.4	Addressing the window content	58
11.2	Data selection	58
11.2.1	Copy and Paste	59
11.2.2	Save data selection	59
11.2.3	Export a data selection	59

INHALTSVERZEICHNIS

11.3	Settings	60
11.3.1	Columns and data format	61
11.3.2	Coloring data	61
11.3.3	Change the font	61
11.4	The data inspector	61
11.5	Searching the record	62
11.5.1	Pattern search	62
11.5.2	Search for time distances	64
11.5.3	Search for transmission errors	65
11.6	Integrated Lua	65
11.6.1	How does it work?	66
11.6.2	Sorted results	68
11.6.3	Select and run a Lua script	69
11.6.4	Script errors	69
11.6.5	Debugging	70
11.6.6	Template file location	71
11.6.7	Import a template	71
11.6.8	How can I remove waste scripts	71
11.6.9	Limitations	71
11.7	DataView specific Lua extensions	72
11.7.1	The data module	72
11.7.2	The debug module	74
11.8	The toolbar	76
11.9	Short commands	77
12	The Event View	79
12.1	User Interface	79
12.1.1	Each line is one event	80
12.1.2	All event types at a glance	81
12.1.3	Signal alterations	81
12.2	Navigation through the event list	81
12.3	Event search with the LevelFinder	82
12.3.1	Enter a search pattern	82
12.3.1.1	Formulate a level condition	83
12.3.1.2	Formulate a data error	83
12.3.1.3	Formulate a data value	83
12.3.2	Search input and search	84
12.3.3	Search for signal changes	85
12.3.4	Searching with time specification	86
12.4	Mark a selection	86
12.4.1	Save a selection as a region	87
12.4.2	Export a selection as CSV file	88
12.5	Measure time distances	90
12.6	The toolbar	90
12.7	Short commands	91

INHALTSVERZEICHNIS

13 The Protocol View	93
13.1 User Interface	94
13.1.1 Telegram window	94
13.1.2 Synchronizing	95
13.1.3 Data direction	95
13.1.4 Open an identical view	95
13.1.5 Pin your settings	95
13.1.6 Goto a given telegram number	96
13.1.7 Filter control	96
13.1.8 Choosing a range	96
13.2 Protocol templates	96
13.2.1 Select a protocol template	97
13.2.2 Modify a protocol template	97
13.2.3 Individual protocol setup	98
13.2.4 Write a new template	98
13.2.5 Template file location	99
13.2.6 Import a template	99
13.3 Template language syntax	99
13.3.1 Splitting the data stream into telegrams	100
13.3.2 Individual displaying of the datagrams	106
13.4 Filtering	123
13.4.1 Show and hide (filter) complete telegrams	123
13.4.2 Choose between different telegram display formats	127
13.5 New filter mechanism	128
13.6 Individual Filter dialogs	131
13.7 Export Telegrams	133
13.7.1 How the program determines the export fields	133
13.7.2 The export dialog	134
13.7.3 Export as CSV file	135
13.7.4 Export as HTML	135
13.7.5 Export as text	135
13.7.6 Export as Latex	135
13.7.7 Special notes about the caption labeling	136
13.8 ProtocolView specific Lua extensions	137
13.8.1 The box module	137
13.8.2 The debug module	139
13.8.3 The event module	142
13.8.4 The linestates module	144
13.8.5 The sequences module	146
13.8.6 The shared module	147
13.8.7 The telegram type	148
13.8.8 The telegrams module	154
13.9 Settings	156
13.9.1 Show additional telegram information	156
13.9.2 Change the font	156
13.9.3 Set an individual background	157
13.9.4 Lua compatibility	157
13.10 The Toolbar	157
13.11 Short commands	159
13.12 Alterations to former versions	159

13.12.1 Incompatible changes	159
13.12.2 Obsolete functions and modules	159
14 The Signal View	163
14.1 Signal representation	164
14.2 Navigation	165
14.2.1 Navigation and zooming by mouse wheel	166
14.2.2 Shift with the hand cursor	166
14.3 The time base	166
14.4 Undo and Redo	166
14.5 Signal control field	167
14.5.1 Remove or hide a signal	167
14.5.2 Signal colour	167
14.5.3 Data overlay	167
14.5.4 Invert signal	168
14.5.5 Rearrange signal order	168
14.6 Settings dialog	168
14.6.1 Common settings	168
14.6.2 Graphical effects	168
14.7 Cursor operating	169
14.7.1 Signal selection	169
14.7.2 Regions	170
14.8 Measure data frames with the frame ruler	170
14.8.1 Adjust the data frame ruler	171
14.9 Synchronizing	172
14.10 The toolbar	172
14.11 Short keys	173
15 Regions	175
15.1 Switch regions on/off	175
15.2 Remove a region	176
15.3 Rename a region	176
15.4 Move regions into view	176
15.5 Region storage	176
15.6 Region properties	177
16 The Editor	179
16.1 Open the editor	180
16.2 Start with a new script	180
16.3 Interactive coding	180
16.3.1 Lua script errors	181
16.4 Highlight individual keywords	181
16.5 Find	181
16.6 Find and replace	182
16.7 Code folding	182
16.8 Editor settings	182
16.9 Colour wizard	182
16.10 Script files location	183
16.11 Editor short keys	183

INHALTSVERZEICHNIS

17 An introduction to Lua	185
17.1 Getting started	185
17.1.1 Using functions	186
17.1.2 Function with multiple results	187
17.1.3 Processing and manipulating strings	187
17.1.4 Data structures in Lua	189
17.1.5 Reuse code with Lua modules	191
17.2 The Lua language	192
17.2.1 Lua is case-sensitive	193
17.2.2 Whitespaces and line ends	193
17.2.3 Comments	193
17.2.4 Types and values	194
17.2.4.1 Numbers	194
17.2.4.2 Integer versus floating point	195
17.2.4.3 Hexadecimal constants	196
17.2.4.4 Floating point constants	196
17.2.4.5 Booleans	196
17.2.4.6 Strings	196
17.2.4.7 Escape sequences in strings	197
17.2.4.8 nil	197
17.2.5 Tables	197
17.2.5.1 Discontinuous tables with holes	198
17.2.5.2 Iterate through tables	199
17.2.5.3 Sorting tables	201
17.2.6 Identifiers	202
17.2.7 Keywords	202
17.2.8 Variables	203
17.2.8.1 Assignment	203
17.2.8.2 Global and local variables	203
17.2.9 Operators	204
17.2.9.1 Arithmetic operators	204
17.2.10 Bitwise operators	204
17.2.10.1 Conditional operators	204
17.2.10.2 Logical operators	205
17.2.10.3 String concatenation operator	205
17.2.10.4 The length operator	205
17.2.10.5 Precedence	205
17.2.11 Control structures	206
17.2.11.1 if then else	206
17.2.11.2 while	206
17.2.11.3 repeat	206
17.2.11.4 Numeric for	207
17.2.11.5 break	207
17.2.12 Functions	207
17.2.12.1 Function call	207
17.2.12.2 Function definition	207
17.2.12.3 Recursive function calls	208
17.2.13 Modules	209
17.2.13.1 Standard modules	209
17.3 Lua restrictions	210

17.4	Lua References	211
18	Lua analyzer extensions	213
18.1	Modules overview	213
18.2	Common extensions for all Views	214
18.2.1	The base16 module	214
18.2.1.1	base16.decode	215
18.2.1.2	base16.encode	215
18.2.2	The bit32 module	215
18.2.3	The functions bpack and bunpack	216
18.2.4	string.pack and string.unpack	218
18.2.5	The checksum module	219
18.2.5.1	checksum.crc8_bacnet	219
18.2.5.2	checksum.crc16_bacnet	220
18.2.5.3	checksum.crc16_ccitt_kermit	221
18.2.5.4	checksum.crc16_df1	221
18.2.5.5	checksum.crc16_dnp3	221
18.2.5.6	checksum.lrc	222
18.2.5.7	checksum.crc16_modbus	222
18.2.6	The config module	223
18.2.7	The record module	224
18.2.7.1	record.analyzer	224
18.2.7.2	record.buswiring	224
18.2.7.3	record.signalnames	225
18.2.7.4	record.starttime	225
18.2.8	The string dump extension	225
18.2.8.1	string.dump	225
18.2.9	The transmission module	226
18.2.9.1	transmission.baudrate	226
18.2.9.2	transmission.bitpause	227
18.2.9.3	transmission.bytepause	227
18.2.9.4	transmission.databits	227
18.2.9.5	transmission.parity	228
18.3	Lua modules for individual views	228
19	Lua Protocol dialogs	229
19.1	How does it work?	230
19.2	The dialog framework	230
19.3	Add a template dialog	231
19.3.1	Add widgets elements to your dialog	232
19.3.2	Apply the user settings	234
19.3.3	Passing data between dialog and script	236
19.3.4	Refresh or reload	237
19.3.5	Defining element action handlers	238
19.3.6	Initialize dialog variables	239
19.3.7	Dialog settings	240
19.3.8	Save dialog settings between sessions	241
19.4	More positioning and interaction	242
19.4.1	Advanced callbacks	243
19.5	Update existing widgets	244

INHALTSVERZEICHNIS

19.6	Further examples	244
19.7	Supported Dialog elements or widgets	245
19.7.1	Named parameters	245
19.7.2	Common widget parameters	245
19.7.3	Button	246
19.7.4	CheckBox	247
19.7.5	Choice	247
19.7.6	Label	248
19.7.7	Line	249
19.7.8	RadioBox	249
19.7.9	Spacer	250
19.7.10	SpinCtrl	250
19.7.11	Table	251
19.7.12	TextCtrl	252
19.8	Functions dealing with widget elements	253
19.8.1	Clear	253
19.8.2	Enable	255
19.8.3	GetPosition	255
19.8.4	GetValue	256
19.8.5	IsEnabled	257
19.8.6	SetValue	257
19.8.7	SetDialogSize	257
19.8.8	SetTitle	258
20	Lua modules	261
20.1	Writing a module	262
20.2	Module path	264
21	Synchronize two analyzers	267
21.1	Technical requirements	267
21.2	Master Slave operation	268
21.3	Establish a synchronous record	269
21.4	Analyse a synchronous record	270
21.5	Synchronize more than two analyzers	271
21.6	Conclusion	271
21.6.1	Synchronous recording	272
21.6.2	Synchronous analysis	272
22	Commandline API	273
22.1	Combine the programs as a tool chain	274
22.1.1	Data source	274
22.1.2	Manipulators	274
22.1.3	Data sink	274
22.1.4	Some examples	274
22.2	Record data with <code>msb_record</code>	275
22.2.1	Connection settings and events	276
22.2.2	Usage in your own application	276
22.2.3	Remote control	277
22.2.4	Synchronous recording with two or more analyzers	277
22.2.5	Remote control a synchronous record	279

INHALTSVERZEICHNIS

22.2.6	msb_record program parameters	280
22.2.6.1	Digital IO setup parameter	283
22.2.6.2	Transmission parameters	284
22.3	Formatted output with msb_format	284
22.3.1	Output of any character	285
22.3.2	File output	286
22.3.3	Format parameters	286
22.3.4	User defined date and time	288
22.3.5	msb_format program parameters	290
22.4	Filtering data output with msb_filter	290
22.4.1	Filter data	291
22.4.2	Filter certain signal events	291
22.4.3	Filter a given record part	291
22.4.4	msb_filter program parameter	292
22.5	Split records with msb_split	292
22.5.1	Split existing record files	293
22.5.2	Splitting the current recording from msb_record	294
22.5.3	Keep only a given number of records	294
22.5.4	msb_split Program Parameter	294
22.6	Trigger a record with msb_trigger	295
22.6.1	Edit a trigger script	296
22.6.2	Define a trigger condition	296
22.6.3	Conditional start of a record with pre and post-trigger	297
22.6.4	Conditional output of an existing record file	298
22.6.5	Scan a record file for certain events	298
22.6.6	One script for scan and trigger	299
22.6.7	Multiple triggering	300
22.6.8	Provided Lua modules	302
22.6.9	msb_trigger Program Parameter	302
22.7	One config file for all	303
A	ASCII character table	305
B	Baudrate measuring	307
C	Colors	309
C.1	RGB short form	309
C.2	RGB long form	309
C.3	Predefined color names	309
C.3.1	Grey colors	310
C.3.2	Basic colors	310
C.3.3	Extended colors	310
D	Windows Trouble-Shooting	313
D.1	Check analyzer connection	313
D.2	Check analyzer bus connections	314
D.3	(Re)Install driver	314
D.3.1	Remove driver	314
D.3.2	Install driver	315
D.4	Helpful program arguments	316

INHALTSVERZEICHNIS

D.4.1	Analyzer not found	316
D.4.2	Firmware transfer error	317
D.5	Please help us with conflicting devices	317
D.6	Disable USB power management	317
D.7	Windows Device Manager	318
D.8	Other problem(s)	318
E	Linux Trouble-Shooting	319
E.1	Check analyzer connection	319
E.2	Check analyzer bus connections	320
E.3	Check your permission	320
E.4	Install udev rule	321
E.5	Remove Braille driver	322
E.6	Helpful program arguments	322
E.6.1	Analyzer not found	323
E.6.2	Firmware transfer error	323
E.7	Please help us with conflicting devices	323
E.8	Check system log with dmesg	323
E.9	Other problem(s)	324

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 more¹) participants. Data can be transmitted optionally over a single 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.

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

KAPITEL 1. ANALYSIS OF RS422/485 BUS SYSTEMS

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-485² 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**

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

1.1. SPECIAL SERIAL DRIVER SOFTWARE

- 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 OS. Equipped with a multitude of visualization tools it allows a very detailed view into every RS485/422 communication and detects conditions which can be recognized by a true 'hardware solution' only.

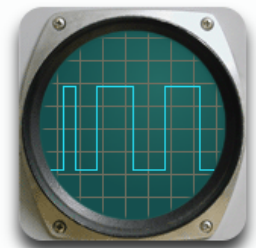
The Analyzer MSB-RS485 samples all four channels and two auxiliary inputs simultaneously with a sampling rate of maximum 16 MHz. Thereby all events, that means level changes on the line, are stamped with the exact time in a micro second precise resolution.

All changes of the line levels are regarded as events, 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 analyzer features real time analyzing and simultaneous access/display of different record parts even during an active recording. The maximum record file size on PC is 4 GByte and unlimited (resp. limited by the free hard disk space) when using the special command line API for long time recordings. The record time depends only on the selected kind of events and data rate of the connection.

2.1 Advantages of a hardware solution

The MSB-RS485 Analyzer offers the capabilities of a logic analyzer, combined with a 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. Especially if you are using PC COM ports. And with typical USB to RS422/485 converters it even gets worse since USB uses a polling mechanism to fetch the data from the converters and the timing becomes more or less useless.



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

KAPITEL 2. MSB-RS485 ANALYZER

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)	✓	—
Any particular baud rate 1 Baud to 1MBps	✓	—
Real time stamps	✓	—
Time resolution	1 μ s	>10ms
Display of the real level changes	✓	—
Automatic detection of baudrate and protocol	✓	—
Supports protocols with 9 Bit data word length	✓	—
Correct time relationships between data and control lines	yes	no
Detects baudrate jitter and wrong bit times	✓	—

2.2 Innovative software concept

The software of the analyzer is designed as a Multi-Process Architecture mapping the OSI model and runs under all modern Microsoft Windows OS (XP, Vista, Windows 8/8.1 and Windows 10) or Linux.

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.

While the control program controls the recording the transferred data can al-

2.3. APPLICATION FIELDS

ready 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.

Application specific protocols and telegrams can be displayed with the help of the integrated script language Lua. Already provided field-bus protocols are:

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

More are in preparation and provided by free updates in future versions.

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



For Windows...
XP, Vista, 8, 8.1, 10



...and all Linux
32 und 64 Bit



Individual protocols
programmable in Lua

KAPITEL 2. MSB-RS485 ANALYZER

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 1 μ sec and a maximum sampling rate of 16 MHz, independent from 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-Analysis** : 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.

KAPITEL 3. FEATURES & BENEFITS

- **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.
- **Logic mode** : Every RS232 input can be switched to a logic input with a trigger level of 1.3V at 5kOhm to sample logic signals which are not » RS232 compatible «.
- **Synchronize of two analysers with mikrosecond precision**: The internal Link jack provides the user with a time synchronous 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 full duplex) by parallel sampling with a maximum of 16 MHz. Precise measurement of all data and bus signals with a resolution of $1\mu\text{s}$.

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 ($-0.7\text{V} < \text{In} < +0.7\text{V}$)

- **Time resolution**

All lines are exactly sampled and marked with $1\mu\text{s}$ time stamps, independent of the operating system of the PC.

EIA-422/485 connectors

- **Signal levels**

Standard EIA-422/485 level $\pm 0.2\text{V}$ to $\pm 12\text{V}$, ESD protected inputs $12\text{k}\Omega$, Common Mode $\pm 7\text{V}$. Detection of the tri-state level of differential signals below $\pm 0.7\text{V}$

- **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.

KAPITEL 4. SPECIFICATIONS

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 XP, Vista, Windows 7, Windows 8/8.1, Windows 10 (all 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**
200 MByte empty space for the software installation plus additional space for the recording files.
- **Memory**
256 Mbyte or more.
- **USB connector**
One free USB 2.0 connector.

KAPITEL 4. SPECIFICATIONS

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, functional for the analyzer (`ftdi_sio`).

The software is bilingual (German and English) and can be installed even without an 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.

5.1 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 7.0.2)`. 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.

KAPITEL 5. PROGRAM INSTALLATION

5.2 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 7.0.2).

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.

5.2.1 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-7.0.2-linux-installer.run ~/Desktop
```

3 Make the installation file executable

by setting the executable flag with:

```
chmod +x ~/Desktop/msb-7.0.2-linux-installer.run
```

4 Start the installation file

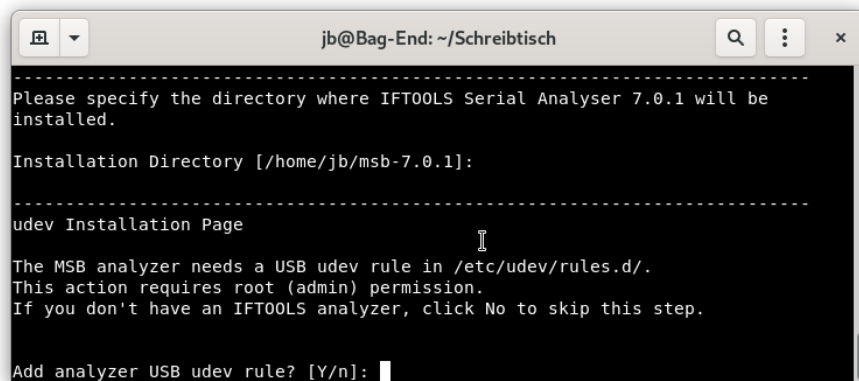
via mouse click (double click). Alternatively you can also run the installer in the text mode in case of the graphical installer did not start:

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

Please note!

For the operation of the analyzer device you need a special `udev` rule. The installer therefore will ask you for your user password to perform the `udev` rule installation as `sudoer` during the installation process. (The normal graphical installation procedure does the same). In the text mode it looks like:

5.2. INSTALLATION UNDER LINUX



```
jib@Bag-End: ~/Schreibtisch
-----
Please specify the directory where IFT00LS Serial Analyser 7.0.1 will be
installed.

Installation Directory [/home/jib/msb-7.0.1]:
-----
udev Installation Page

The MSB analyzer needs a USB udev rule in /etc/udev/rules.d/.
This action requires root (admin) permission.
If you don't have an IFT00LS analyzer, click No to skip this step.

Add analyzer USB udev rule? [Y/n]:
```

It is mandatory to input 'Y' otherwise a connected analyzer will not be recognized by the Linux system!

5 Install the udev rule manually

In case something went wrong with the udev rule, you can install the rule by yourself. Since it is a write access to the `/etc/udev/rules.d` directory, you need root rights (or become a sudoer).

For this execute the analyzer installer as described above but input 'n' to skip the udev installation. When the installer is finished, change directory to the new analyzer software folder (usually it's a folder `msb-VERSION` in your home directory).

In this directory execute the following command best as sudoer or alternatively as root:

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

You can always remove the rule by executing the uninstall script via:

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

But then again the analyzer is not recognized by your system anymore.

5.2.2 Installation for all users

A system wide installation is as easy as to install the software for a single user. In this case just run the installer as a sudoer, for instance:

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

The installer will suggest you `/opt/opt-VERSION` as installation path when it detects running with root/sudo permissions. Alternatively you can run the installer in the text mode. This makes sense if the graphical installer did not start or you only have access by a text terminal:

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

KAPITEL 5. PROGRAM INSTALLATION

5.3 Program Updates

IFTOOLS issues software updates in irregular intervals with new features and improvements. These updates are free of charge and can be loaded from the following address <https://iftools.com/download>

The updates 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 users only have to execute the update (installer) file.

Under Linux it is necessary to make the file executable and then to start it like described above.

6

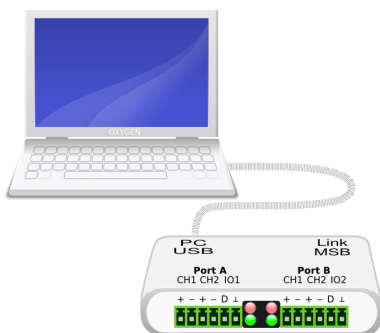
Connection of the Analyzer

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

KAPITEL 6. CONNECTION OF THE ANALYZER

avoid more confusion the terminals of the analyzer MSB-RS485 are simply marked with '+' and '-', which corresponds 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 : $\boxed{\text{CH1}} \rightarrow \boxed{\text{CH2}}$ 1 : $\boxed{\text{CH2}} \rightarrow \boxed{\text{CH1}}$
Output	Activity of the bus through $\boxed{\text{CH1}}$ and $\boxed{\text{CH2}}$ (segment analysis): 0: inactive (Tri-state), 1: active
Output	Bus activity of $\boxed{\text{CH1}}$ 1: active, 0: inactive (tri-state)
Output	Bus activity of $\boxed{\text{CH2}}$ 1: active, 0: inactive (tri-state)
Output	Bus activity of $\boxed{\text{CH3}}$ 1: active, 0: inactive (tri-state)
Output	Bus activity of $\boxed{\text{CH4}}$ 1: active, 0: inactive (tri-state)
Output	Error output, 1-bit width pulse when detecting frame, parity or break

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.

KAPITEL 6. CONNECTION OF THE ANALYZER

To make the connection and assignment as simple as possible the analyzer offers a selection of various combinations for bus system and tapping (the connection mode or short wiring), which are in accordance with the following variants.

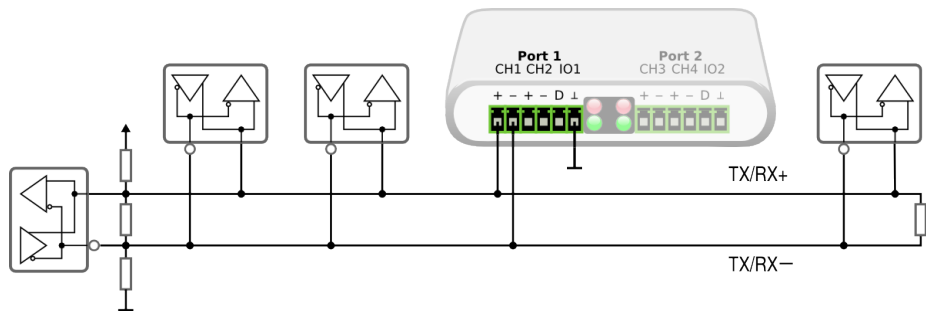
6.5 Tapping 2-wire system

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-)

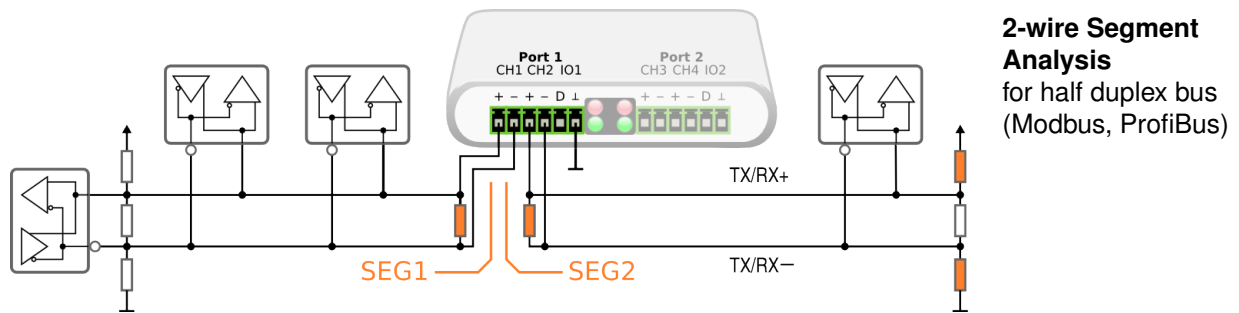
The second bus segment is connected to the second analyzer channel Port1, CH2 (CH2+,CH2-).

The bus is now split into two segments. Please note that you possibly have to

6.7. TAPPING 4-WIRE SYSTEM

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)

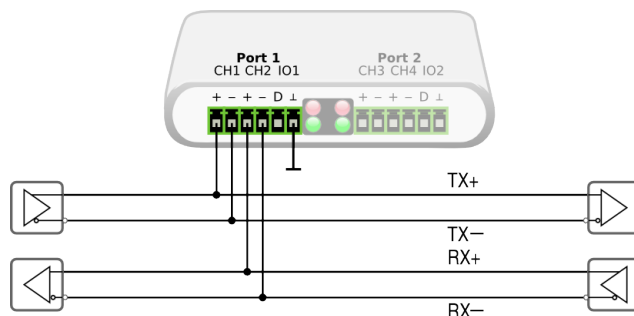
The analyzer records the data of both segments independently while the data of both directions are passed fully transparent. Data of the first segment are marked as coming from channel 1 with the internal name 'A', data of the second segment are marked as coming from channel 2 with the internal name '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)

KAPITEL 6. CONNECTION OF THE ANALYZER

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.

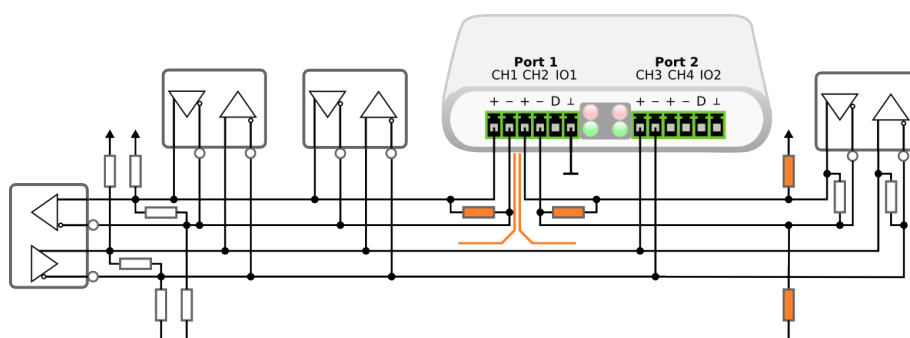
In contrast the slaves share one channel to send their data back to the master. With the help of the segment analysis a singular device can intentionally be separated and its communication with the master monitored without regarding the other devices.

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 diffe-

6.10. LIGHTMENT ELEMENTS LEDS

rential 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

^aThe 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

KAPITEL 6. CONNECTION OF THE ANALYZER

6.10.1 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, corresponds 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.

6.10.2 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

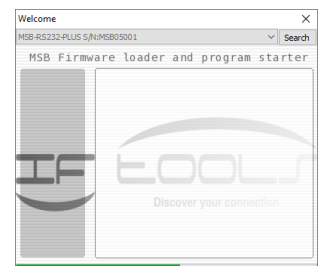
Program start

The control program is the cockpit of your analyzer. It is the pivotal point to prepare and record bus transmissions. With it you start a recording, save or load records or projects and open different analysis tools to examine the transmission on several OSI layers in real-time.

The **Firmware** of the MSB-RS485 is 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.

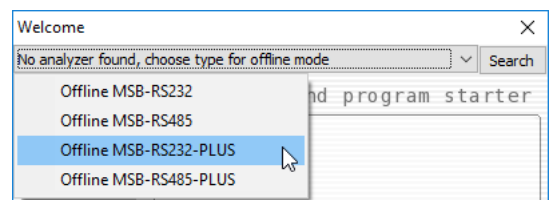
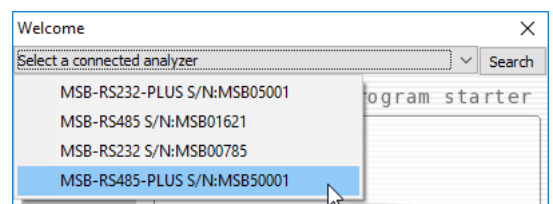


The first start loads the firmware into the device

If no MSB-RS485 was found, even though it is connected to your PC, read the hints for Trouble shooting (Windows **D**, Linux **E**) 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 (PLUS) or MSB-RS485 (PLUS) - 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.

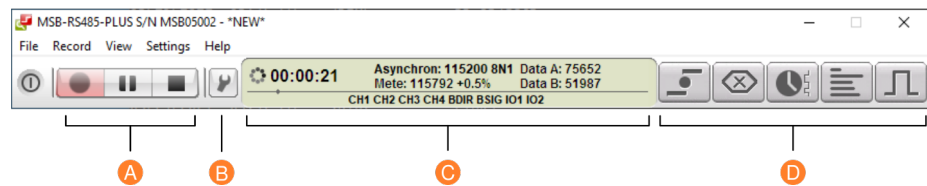


KAPITEL 7. PROGRAM START

The MSB-RS485 control program is designed for an easy understandable operation experience and offers you a set of easy to operate tools which are appropriate for your application.

7.1 User Interface

The control program appears after the firmware transfer was completed. Depending on your last session or if you started the program by clicking a project further program windows representing different views into the record will appear. But here we talk about the control program and it looks like this:



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

The control program is divided in four parts. On the left side are the record controls (A). They are inspired from an audio or video recorder and contains of a record, pause and stop button. The meaning should be easy-to-read. On click, and your record starts...

The button with the wrench symbol (B) gives you access to all necessary record settings, see 7.2. You have to set-up the analyzer according to the examined bus - but only once as long as you don't have a change in your application. The central part of the control program is the status display (C). It gives you all vital information about an active recording. A short description of the selected transmission type, the record time, metered bit rate and received data and events and more. We explain it in detail in section 7.4.

The remaining right part (D) contains a launcher for all so called Views. These are independent programs and every type gives you a special OSI level insight into the active transmission or record, see section 7.5.

Remember! The analyzer software has a multi-process design. You can launch as many Views to examine the transmission as you want. The control program is completely independent from it and won't be affect even if one of the View crashed! This gives you maximum safety for your recording especially in case when where wouldn't be a second chance to fetch an error condition.


But let's begin how to setup or configure a record.

7.2. CONFIGURE A RECORD

7.2 Configure a record

To stick with our picture of an audio recorder you must first select the input source and adjust the record level. The same applies to the analyzer. Before you press the record button you have to configure the device! The software memorizes these settings so that you do not have to enter them any more.

The MSB-RS485 analyzer is able to record and analyse a lot of different bus systems and transmission protocols. But to do so you must first tell the analyzer what kind of bus you want to record, how you have connected the bus with the analyzer device and which are the transmission parameters (bit rate, data format,...).

The analyzer program leads you to all necessary steps. Just click the setup button with the wrench symbol  on the left side of the control display.

The appearing dialog is divided in different setting options arranged by the setup order importance from left to right.



Mandatory for a record are the first two options because these specify what the analyzer samples (in our picture the input source) and how the data are evaluated from the transmitted bit stream. The other four dialog tabs (Signals, Record, Autosave and General) are preset with default values which covers almost all applications in the beginning. You can left them unchanged in the beginning.

Some settings directly influence the logging. Therefore they are deactivated while a logging session is running. The respective setup menus are then displayed in gray. The same is valid if you work offline (because without a connected analyzer device there is nothing to set).

7.2.1 Transmission setup

The settings in the first tab tell the analyzer all it has to know about the physical transmission layer! In the OSI model these involves the physical and data link layer. The MSB-RS485 allows you to record almost all kind of asynchronous field-bus transmissions. But to do so it is absolutely necessary to set the correct parameters otherwise the analyzer cannot sample the accurate data from the transmission signal!

Note! The analyzer is able to detect the parameters for you (see section [Automatical protocol scan](#)), but you nevertheless have to apply it BEFORE you start a record!

The reason: An automatical detection and setup during an active sampling inevitable leads to data losses caused by the fact that the hardware needs a certain period of time to determine and apply the right settings.



Record setup
Configure a record



Transmission setup

KAPITEL 7. PROGRAM START

An asynchronous transmission is specified by the bit or baud rate and the data format. The latter includes the number of data bits, an optional parity and the number of stop bits. You may have seen this specified e.g. as 38400 8N1 or 9600 7E2. As mentioned before: You can let the analyzer detect the right settings or input it manually in this settings form. The necessary parameters are:

- 1 Bit rate** - or baud rate. Specifies the transmission speed of the bus. Additional to the standard bit (baud) rates, the MSB-RS485 analyser 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. 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.
- 2 Data bits** - the number of bits transmitted as one byte. 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.
- 3 Parity** - indicates the number of set bits for error detection. The parity setting does not influence the data evaluation, but you should use the same parity setup (none, odd, even, mark or space) as used by the transmission to avoid false parity errors.
- 4 Stop bits** - automatically handled by the analyzer. People often wonder why the analyzer doesn't need the number of stop bits. The answer is: For the analyzer the number of stop bits does not matter. Like the start bit is necessary to indicate the start of a frame with the transition from idle to active the stop bit finishes the data frame with a safe return to idle. So the stop bit is logical '0'. Only one stop bit is necessary for correct decoding, the following bit can be the start bit of the next frame. To give the receiving decoder more time to transfer the detected byte to its internal output buffer a further stop bit can be added before the next start bit. Generally every further stop bit extends the gap to the start of the next frame, so these additional stop bits are simply idle conditions. That means that a data receiver like the MSB-RS485 do not care about the number of stop bits. One stop bit is sufficient to end the frame and to transport the data to the internal analyzer buffer for further processing. All further idle bits are ignored, waiting for the edge of the start bit of the next data frame.



Stop bits
are handled automatically

Auto-detect the asynchronous parameters

Don't worry if you don't know the asynchronous protocol settings of your connected bus. 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 bit rate but also the detection of the used data format. The only thing you have to do is to click the [Autodetect](#) button below the paramter controls to open the protocol scanner dialog.

7.2. CONFIGURE A RECORD

A correct detection of the connection parameters implies an appropriate transmission. For an exact result the scanner needs longer and especially different data sequences. A transmission with always the same data (byte) leads to wrong or inaccurate results. 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 bit 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 parameters are correctly detected you can adopt the found settings with the button 'Use scan'. Afterwards you are lead back to the settings dialog.

Bit rate measurement

By default the display of the control program shows the measured bit rate of both data channels A and B whereas alternatively the detected data at CH1 or CH2 are used for the evaluation. The displayed bit rate is always a good indication of any bus activity and works constantly without starting a record. You can select a single data channel (detecting source) if you want to check a certain direction (segment mode) or bus. Or you disable the metering completely (some synchronous bus systems does not use a constant bit rate).

7.2.2 Bus wiring

The MSB-RS485 Analyzer has four differential inputs which are used as data or signal loads, according to the selected 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. This applies for all kind of transmissions!

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 (section 7.2.3) you can change this behavior and input own names. The signal assignment is separated into:

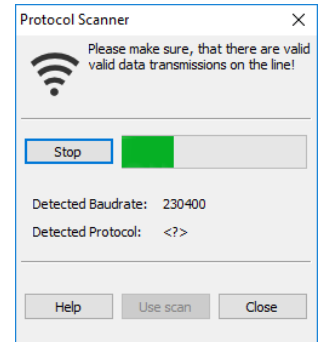
■ 2 Data channels

These channels contain the decoded data of both USARTs. 9 bit data are supported and displayed as well as occurred transmission errors like parity and framing. Data frames with more bits like in synchronous transmissions are divided in consecutive bytes.

Depending on the selected wiring a data channel can also contain the data of several differential inputs (asynchronous transmissions only). 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.



Autodetect dialog
for asynchronous
transmissions parameters



Choose a Bus wiring

KAPITEL 7. PROGRAM START

The signal channels 5 to 8 have a special status depending on the transmission type. When analysing an asynchronous transmissions via segment mode they display additional internally generated logic signals. These are the bus-direction between CH1 ↔ 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 channel.

With a selected synchronous bus type, channel 5 to 8 provide a valid frame/error signal and a data/clock combination for both data channels.

- **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

7.2.3 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 program provides you with reasonable signal names as a result of the chosen transmission type (transmission page) and optional the specified mode of connection. 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 check mark 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 USARTs are switched on.

Please check, 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 USARTs 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.

7.2. CONFIGURE A RECORD

7.2.4 Record mode

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 occurrence 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:



Choose a record mode
synchronous, continuous
or loop recording

1 Continuous recording:

In the continuous mode all occurring 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.

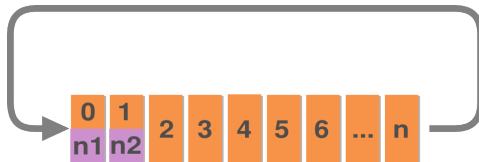


Events

2 Time loop with Fifo mode:

In the Fifo Mode a certain amount of data of the last (before stop of recording) occurred 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 corresponds 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.



Events

Events after Fifo end

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 receives so called synchron impulses on its MSB-Link jack from a connected 'Master'.

KAPITEL 7. PROGRAM START

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. We describe this feature in detail in section 21.

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 examine 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.



Save a record
automatically after
each stop

7.2.5 Autosave

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 accessible.

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.

7.3. START A RECORD


7.2.6 General

This page is intended for general settings. Here you can enable the security question for not yet stored data (default is on) and also save the current session as the default session so it will be restored during the next program start (default is on too).

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 both records interact as it used to be in a single recording.

For this case you have to allow the external synchronisation first.

7.3 Start a record

As soon as you have specified the communication parameters one click onto the record key starts the recording. The record key begins to glow red (see margin picture) 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 occurring 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.

All further tools to display the transferred data or their physical level can be optionally opened or closed without influencing the recording.

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.

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 7.15 Additional program arguments).

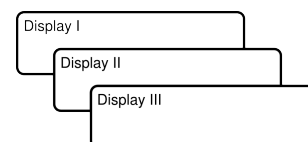
The speed of the moving marker is depending on the quantity of the occurring



Common settings
like warnings, taskbar behaviour and external views synchronisation



Record Pause Stop
Recording control



Toggle the display
with a right mouse click

KAPITEL 7. PROGRAM START

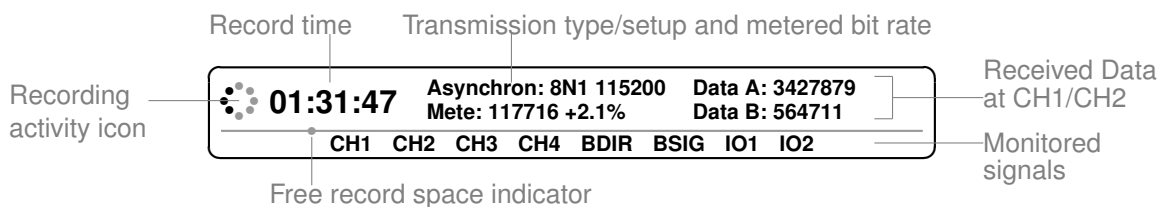
(and selected) events as well as on the size of the empty space in the used temporary directory.

7.4.1 Display I

This is the default display when you start the program. It informs you about the elapsed record time, the set transmission type and transmission parameters (here Asynchron, bit rate, data format). And by comparison the metered baud or bit rate with deviation in percent to the set value (Mete:).

The part below the line of the record space indicator shows the activated input channels resp. signals you have activated for the recording.

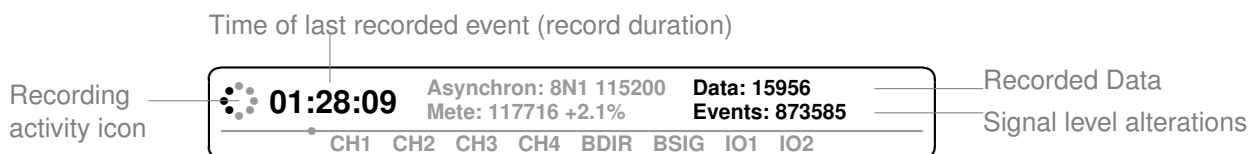
The number of received data bytes (depending on the direction, bus or bus segment) are shown on the right side (Data A and B).



7.4.2 Display II

The second display informs you about the total sum of transmitted data bytes and about the quantity of the remaining events. These are in general all signal alternations including the signals generated by the analyzer itself like bus direction, frame valid signal a.s.o.

Note! In this display mode the time indication is related to the last occurred event and not the running recording time. So the time remains zero until the first event happens and only changes with a new occurring event, either a data byte or a signal alteration.



7.4.3 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 and uses a direct USB link for the full 480 MBit transfer rate.

7.6. SAVE A RECORDING



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 internal buffer of the device driver could not handle the data rate (overflow, lost data). In this case you should reduce the number of recorded events.

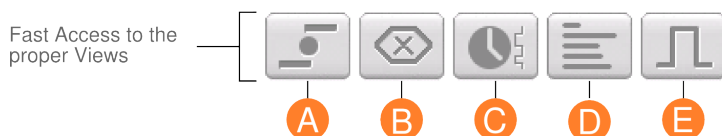
All signal and data lines can be individually enabled and disabled for logging. Inactive are shown as lines.

In the examples above the level changes of the input channels CH3 and CH4 are not logged.

7.5 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.53) Virtual Ledtester:** Das virtual counterpart of a real ledtester.
- B (P.55) Data View:** Data dump of the transmitted data with special search features.
- C (P.79) Event View:** All line changes in a clear look, search for line modifications.
- D (P.93) Protocol View:** Display any protocol with your own definition.
- E (P.163) Signal View:** Digital Scope like view of all lines.

7.6 Save a recording

Independent 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 recor-

KAPITEL 7. PROGRAM START

ding. In the opening dialog you can enter a new file name (The extension .msblog is automatically 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 chosen 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 transmitted data bytes only or a part of it use the data monitor and mark the interesting range.

7.7 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 extension.

Separate files for project and record

Project files always have the extension *.msbprj, the recorded data files the extension *.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.

7.8 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

7.9. OPEN AN EARLIER SESSION (PROJECT)

into your current session. Please note that this can be done only if no recording is running and that your recorded data are overwritten.

7.9 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.

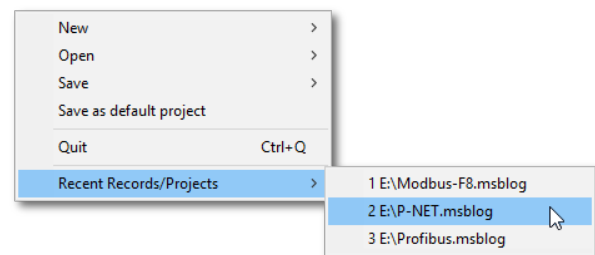
7.10 Last opened recordings and projects

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 chosen 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.



7.11 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.

7.12 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.

KAPITEL 7. PROGRAM START

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:\Program Files (x86)\msb-7.0.2\msb_serv.exe -nMSB#####
```

resp. for Linux:

```
~/msb-7.0.2/msb_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.

7.13 Automatical start after computer boot

The analyzer can be started automatically and set into the logging mode after booting of the computer. An additional parameter ensures that the analyzer switches into the record mode immediately after the firmware transfer was completed. At the same time this parameter takes care, that the logging stops and the resulting record file is correctly closed when the system is shut down.

That means in detail:

- 1 As soon as the boot process is finished an analyzer is searched and loaded 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.

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


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 22 and put an according script or batch file in the autostart folder.

7.14. SHORT COMMANDS

7.13.1 Activate the autostart feature under Windows

Windows automatically starts all programs after login which are located in the Startup folder of the logged-in user (since Windows 7). That is also valid for the MSB-Analyzer software.

- 1 Press the keys  + R and input the following command: `shell:startup`
This opens the Startup folder in the file explorer.
- 2 Copy and paste the analyser desktop icon into the Startup folder.
- 3 Right click the COPIED icon un the startup folder and select `>> Properties <<`.
- 4 Add to the target entry the autostart parameter `-a`, i.e.:
`C:\Program Files (x86)\msb-7.0.2\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.

7.13.2 Activate the autostart feature under Linux

The Linux approach is similar to windows but there are two autostart (startup) folders to consider. The system wide folder is located under:

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

but you need root rights or must use `sudo` for write permissions. We recommend to use the user depending autostart folder. Especially if you installed the software as a normal user. The location (and also the mechanism) might depend on the desktop environment. But for most desktops the folder path is:

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

Just open the autostart folder with your file browser and copy and paste the analyzer desktop start icon into this folder.

Afterwards right-click the start icon (in the autostart folder!) and select `>> Properties <<`. In the command field add the analyzer autostart parameter `-a` as described in the autostart Windows section above. The command field should look like:

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

Close the property settings dialog and login again to check if the analyzer software starts automatically. You don't have to rebbot your system!

7.14 Short commands

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



Short commands
of the most important
functions

KAPITEL 7. PROGRAM START

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 Protocol View	Ctrl + Alt + P
Open a Signal View	Ctrl + Alt + S
Open the bookmarks (regions)	Ctrl + Alt + R
Save settings and close program	Alt + F4

7.15 Additional program arguments

The MSB control program can be called with a series of additional parameters to set explicit defaults like language, offline mode or the type of the connected analyser.

In most cases the default setting (automatic search and 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 7.12.

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 <i>directory</i>	Set the working directory.
-e	Starts the control program with the default settings. All stored program and session settings will be ignored.
-f <i>firmware</i>	Load an alternative firmware (firmware file). USE WITH CAUTION! An invalid firmware may damage the device!
-i	Forces the loading of the firmware even when the MSB-RS485 is already loaded.

7.16. SPECIAL PROGRAM PARAMETERS

-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 <i>language</i>	Select the language. Values for <i>language</i> are: 0: System default, depending on your operating system, 1: english, 2: german Syntax: <code>msb_serv -l1</code>
-n <i>serno</i>	Select a analyzer by it's serial number <i>serno</i> . Important, if you are connecting more than one analyzer at the same time. Syntax: <code>msb_serv -n MSB12345</code>
-o <i>type</i>	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: <code>msb_serv -o typ</code> Valid types are: MSB-RS232 MSB-RS232-PLUS MSB-RS485 MSB-RS485-PLUS For instance: <code>msb_serv -o MSB-RS232</code>
-r <i>number</i>	Reduces the firmware transfer speed by the given number. Default is 0 (full speed), maximum value 100.
-T <i>directory</i>	Presetting of the directory where the temporary logging data is stored. By default this is for Windows: <code>C:\Users\USERNAME\AppData\Local\Temp\</code> and for Linux <code>/tmp/</code>
--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.

7.16 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.

KAPITEL 7. PROGRAM START

Parameter	Description
<code>--exit-without-saving</code>	Close the program without any warning about unsaved data or settings.
<code>--fifo-size</code>	Specifies the size of the data fifo between the analyzer and the program. The default size is 10MB. You can increase the size when recording a bus with very high bit rates and data load. Please note that the maximum fifo size depends on the free system memory. The following example set the fifo size to 700MB: <code>msb_serv --fifo-size=700000000</code>
<code>--ignore-unsaved-data</code>	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.
<code>--max-used-memory=size</code>	Tells the program to use a smaller size of shared memory. Default is 4 GByte. For example: <code>msb_serv --max-used-memory=1000000000</code>
<code>--socket=portnumber</code>	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.

KAPITEL 8. THE MULTIVIEW DESIGN



Synchronize 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 <i>frozen</i> , 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 <i>master</i> View.

8.1.1 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.

8.1.2 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.

8.1.3 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.

8.2. VIEWS (DISPLAYS)

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:

8.2.1 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.

8.2.2 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.

8.2.3 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.

8.2.4 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.

8.2.5 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.

KAPITEL 8. THE MULTIVIEW DESIGN

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.

8.2.6 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



Save settings

for this view type

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.

8.4 Default settings for Views

Position, size and individual settings of every open view are stored by default in the default project file¹ when the application is closed (the user quits the control program).

Every view launched by the control program starts with its own individual setup which may be different from the reopened views after program start.

You can specify the standard setup (how a certain view appears) by 'pin' the current settings of every view with clicking the pushpin icon in its toolbar. For instance:

You want to start the SignalView always with the dark theme (black signal background) and only 4 signal channels. Let's say only RxD, TxD, RTS and CTS or CH1...CH4 when running a MSB-RS485 (PLUS).

Just configure the SignalView according your wishes and click the pushpin in the toolbar. The control program shows a short message confirming the new setup. If you start a new SignalView afterwards, it uses the new setup. This setup is also stored as the default session when you close the application and available again in your next session.

¹Project 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 default configuration when ending the session by closing the analyzer control program. The session is automatically 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.

KAPITEL 9. SESSION MANAGEMENT

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 separation 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 on to 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 following naming:

ANALYZERTYPE-SERIALNUMBER.msbprj

Typical names are:

9.3. AUTOMATIC STORING OF A SESSION

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

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

The location of the default project files depends on your OS. Windows User will find them under:

[C:\User\USER\AppData\Roaming\IFTOOLS\SerialAnalyzer\VERSION\Defaults](#)

except for Windows XP where they are be stored under:

[C:\Documents and Settings\USER\Application Data\IFTOOLS\SerialAnalyzer\VERSION\Defaults](#)

Under Linux the default project files directory is:

[/home/USER/.IFTOOLS/SerialAnalyzer/VERSION/Defaults](#)

10

The virtual Ledtester

The current line level indicating LED testers are standard tools for checking the signal levels at RS232 lines. The adapted and virtual counterpart for EIA-422/485 connections 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 $\pm 0.7V$ both LEDs are off. This corresponds to the inactive Bus/line condition. The trigger level of the EIA-485 receivers is about $\pm 200mV$. 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 $\pm 200mV$.

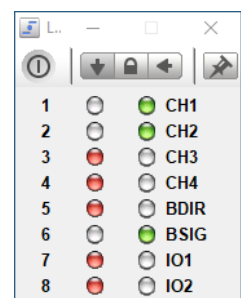
By default the current state is displayed independent of a running recording. This corresponds to the synchronization to the last recorded event. Therefore the scroll button in the toolbar is activated and the virtual LED tester acts like a real tester.

You also can use the LED tester for watching the status of earlier data in the recorded data stream.

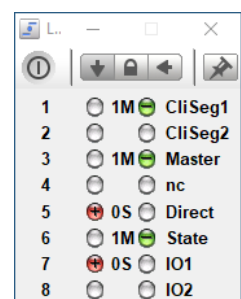
Click the 'Sync' symbol in the toolbar. With this the line state monitor is synchronized with the active display window, e.g. a data monitor. Or you freeze the current state by clicking the 'Lock' Button.

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.



Virtual Ledtester
with standard names...



Blended levels
and individual names...

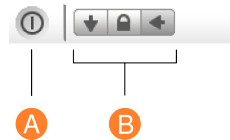
KAPITEL 10. THE VIRTUAL LEDTESTER

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 analyser 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

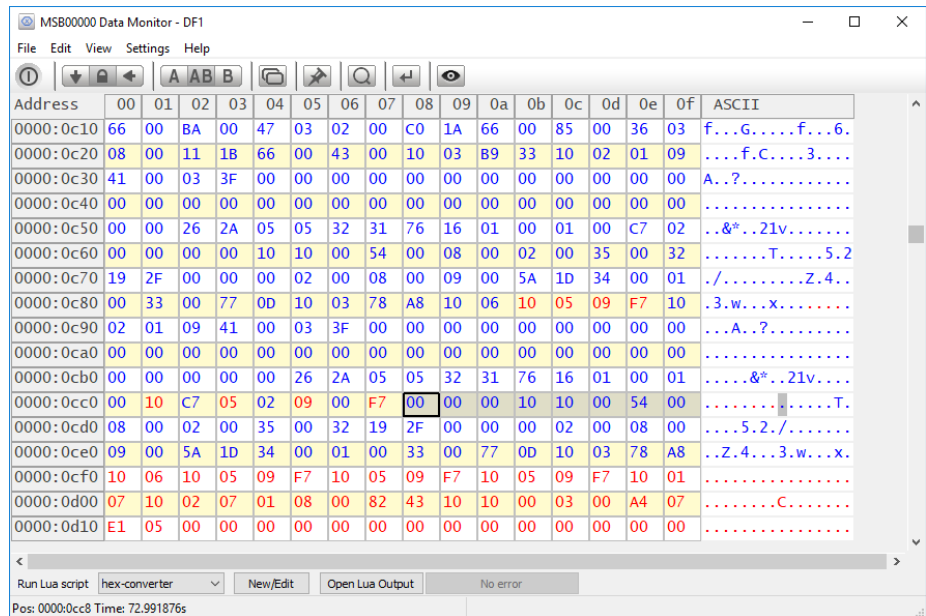
The data monitor shows the transmitted data bytes like a hex editor. Default are 8 characters or bytes¹ per line, displayed in hexadecimal notation and in the ASCII representation. Every line starts with the current address or position as the offset from the beginning of the data stream. Non printable 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, see the following section [11.1.1](#).

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

KAPITEL 11. THE DATA VIEW



With the integrated Lua script interpreter you can calculate the displayed data in any form, convert a sequence of data in another format and display the result in the Lua output.

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 certain 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.

All Lua related functions are grouped below the main window and can be called quickly if needed. For more information see section 11.6.

The information in the status line is always related to the current cursor position. The first left value Pos: states the exact position inside the data stream. Only data bytes are counted, other events like level changes are ignored. The second value Time: shows the exact time when the data byte has occurred.

11.1.1 Display of data errors

In an asynchronous serial data connection a data byte is transmitted as a frame of bits. Every byte starts with a start bit, followed by a number of bits (5 to 9), an optional parity bit and an ending stop bit. The whole sequence is called a data frame. The start bit (or better the falling edge of the start bit) is especially important since it not only marks the beginning of the frame but also triggers and resynchronize the data sampling of the recipients. The signal picture below gives you an impression how a data frame looks like on the physical (logical) level as displayed in the signal monitor.

If the analyzer recognize a wrong frame, for example there are more or less data bits as set in the connection settings, these data bytes are marked with a

00	DA	34	00	^B 33	^P
19	^F	^B 00	^P 33	^P 43	^P D1
C8	EB	FB	00	00	
FC	5E	6A	34	32	
36	34	32	C8	EB	

Displayed data errors
Frame, Parity, Break

11.1. USER INTERFACE

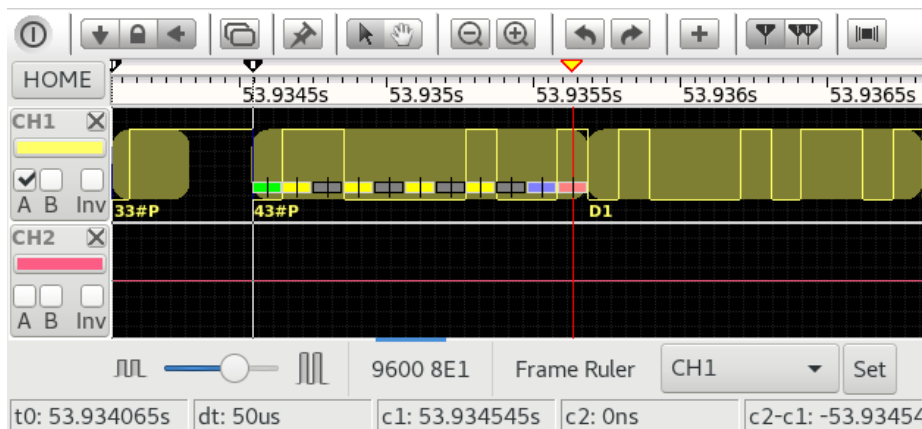
'F' (frame error).

A 'P' (parity) error is shown when the parity bit does not fit with the record settings. This happens when the record is made for instance with an even (E) parity (something like 38400 8E1) and the data is transmitted with an odd (O) parity (38400 8O1).

Both - frame and/or parity errors can be also caused by a wrong or jittering baud rate. They are always and indication, that something with your transmission (or at least the recording) goes wrong.

A break is indicated as 'B' and defined as a specific time which is significant longer than one character transmission where the signal level is low. The according data byte therefore shows a null value but must not be confused with a real 0 byte. The analyzer distinguishes between real breaks and null bytes. Breaks are sometimes used as telegram delimiters or to reset the recipient.

You can open a signal monitor and set its synchronization settings to follow other views, see the following section 11.1.2. Afterwards just click the erroneous data byte in the data monitor and it will be displayed in the signal view for further evaluation.




Signal in Sync Mode
shows clicked data byte


In the example picture above the signal view shows clearly the reason for the parity error. The record was adjusted to 8E1, but the data was transmitted with 8O1 (odd parity). The analyzer marks the data byte because it expected a even number of bits in the frame (including the parity bit) and therefore a high level of the parity bit (displayed as blue in the frame ruler).

Sometimes only a few data errors occurs. If you want to be sure, that there is definitely no error in the data, you can search for them as described in section 11.5.3.

11.1.2 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.


Scroll, Lock or Update
display by other views

KAPITEL 11. THE DATA VIEW

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 chosen display lock/unlock function. Windows which shall not react on sync signals have to be locked.


11.1.3 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 segment 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 also defines which data are stored as a binary data file. 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.

11.1.4 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.

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

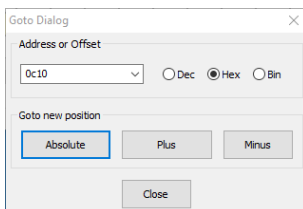
If you press the left mouse key while the cursor is on one of the displayed data byte a context menu opens. In this menu any section of the recorded data can be selected. You can mark the beginning or the end of the selection.

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 Microsoft Windows Explorer. The selection is marked with a light blue color.



Data selection

Single channels or both



Absolute or relative
positioning with Goto

11.2. DATA SELECTION

If you want to select all data, just press Ctrl+A.

The data selection can be stored separately 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.

11.2.1 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 blue².

Pure text editors (like Notepad) doesn't provide any text formatting. 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.
```

11.2.2 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.

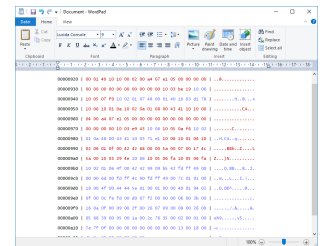
For example if you know the result of the sent data from a bus participant or the original data and you simply want to check if these data have been correctly transmitted. Simply select the wanted range or all data with Ctrl+A and click the menu item File→Save as....

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.

11.2.3 Export a data selection

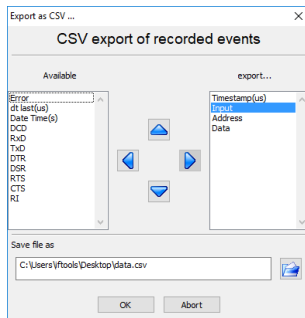
To analyze a section or all recorded data with spread sheet analysis you can export these as a CSV (Comma Separated Values) File. Spreadsheet programs offer extensive statistical tools to evaluate the data. For example the frequency distribution of single data or minimum and maximum times between the bytes.

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



Copy and Paste
in a word processor

KAPITEL 11. THE DATA VIEW



**Data export
as a CSV file**

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 apostrophes. The same applies for the addresses.

(The address is the position of the data in the data stream). An example for the hexadecimal address and data format:

```
"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 Settings

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.

11.4. THE DATA INSPECTOR

11.3.1 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.

11.3.2 Coloring data

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

With version 5.0 the number of colour rules is limited to 4 for a better usability and what we experienced as sufficient in most cases.

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 4 (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.

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, see chapter 11.6.

11.3.3 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.

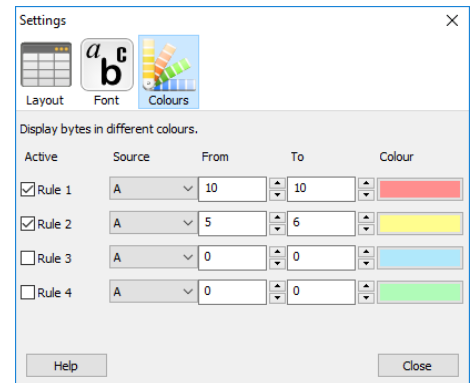
Click on `Settings`→`Configure Data View` to open the settings dialog. The chosen font type is automatically stored.

11.4 The data inspector

Watching the transferred data is one thing. To find out the reasons for communication problems sometimes it is necessary to analyse the exact timing for the transferred 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 'In-

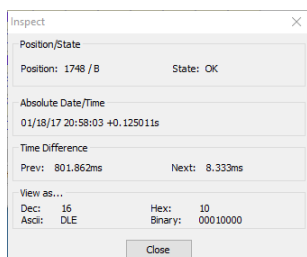
Adresse	00	01	02	03	04	05	06	07
00000000	41	54	5A	0D	0A	0D	0A	4F
00000010	30	45	0D	30	0A	0D	0D	0A
00000020	0D	0A	0D	0A	41	43	54	49
00000030	4C	45	3A	0D	0A	42	39	39
00000040	31	20	51	30	20	56	31	20

Coloring data with colour rules...

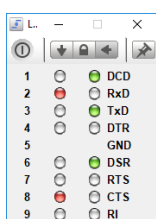


Choose another font in the settings dialog

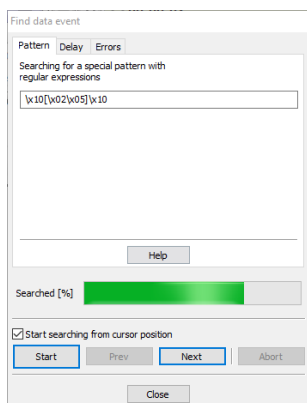
KAPITEL 11. THE DATA VIEW



The data inspector shows time distances and converts data byte values



The virtual Ledtester shows the current line levels



Find any string with regular expressions

spect' 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.

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.

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.

11.5.1 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 which have the extension 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

11.5. SEARCHING THE RECORD

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.

```
[ L l ][ O o ][ G g ][ I i ][ N n ]
```

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

```
Password$ \ 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 'PASSWORD' but not single 'LOGIN' Sequences without following 'PASSWORD' Sequence:

```
LOGIN*PASSWORD
```

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 table lists the available expressions.

Expression	Meaning
?	any character

Search for 9-Bit sequences

KAPITEL 11. THE DATA VIEW

*	any character string
[abc]	a char out of the set <i>abc</i>
[!abc]	a char not member of the set <i>abc</i> 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, carriage 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.

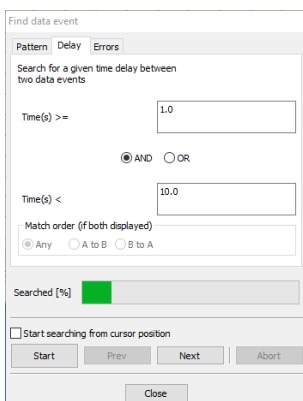
11.5.2 Search for time distances

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 1 μ s (MSB PLUS series 10ns). 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 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.



Find time distances and transmission intermissions

11.6. INTEGRATED LUA

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.

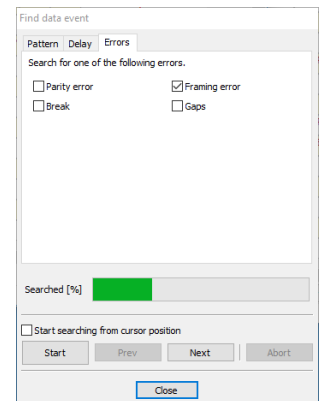
The sequence which shall be obeyed for the search process can be set explicitly. Default is Any, i.e. the sequence is irrelevant.

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.

11.5.3 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.



Find errors
just with a click

11.6 Integrated Lua

The DataView was the first View with integrated Lua support. It was completely reworked in version 5.0 and comes with a lot new features making the processing of the data with Lua more easier than ever.

First at all: The former design of a line based watch expressions folded out below the main window was replaced by a general and freely positionable output window. The new output window let you display single line results but also complex tables as shown in the left picture.

And: You do not longer stick on the limited editor as provided by the earlier watch window! The DataView now uses the new full featured stand-alone Lua script editor as all other Views with Lua support.

The new editor not only helps you with predefined code frames it also let you test selected Lua code lines and complete scripts as well.

The editor is described in detail in chapter 16.

With Lua you are able to answer questions like:

Conversion	Value
Hex Sequence	00 00 00 10 10 00 54 00
Double Big Endian (ABCDEFGH)	3.4084568016767e-313
Float Big Endian (ABCD)	2.2420775429197e-044
uint32 Big Endian (ABCD)	16
int32 Big Endian (ABCD)	16
uint16 Big Endian (AB)	0
int16 Big Endian (AB)	0
Double Little Endian (HGFEDCBA)	4.4502022523062e-307
Float Little Endian (DCBA)	2.5243548967072e-029
uint32 Little Endian (DCBA)	268435456
int32 Little Endian (DCBA)	268435456
uint16 Little Endian (BA)	0
int16 Little Endian (BA)	0

Number Converter
written in Lua

KAPITEL 11. THE DATA VIEW

- What is the CRC16 checksum of a given sequence?
- What is the 32 bit unsigned integer value in little endian of the selected four bytes?
- How long does it take to transmit the selected number of bytes?

There is almost no limit what you can do with Lua as long as your code is restricted to the DataView internals.

11.6.1 How does it work?

Exchanging data between the DataView and Lua is provided by a very simple mechanism. The DataView passes all necessary information like cursor position and a possibly selection to Lua. In Lua you have access to all recorded data. You can process them as desired. The result must be passed back to the DataView as a simple Lua value or as a table. In detail:

The DataView executes a certain Lua function every time the cursor is moved (or the script is updated). This function `onchange` is called with the current cursor position and the start and end of a given selection. The function is defined as:

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

In the function body you can access every data byte received during a recording either absolutely or relatively to the passed cursor position. For instance:

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

The example simply returns the value of the received data byte at the given cursor position. The module function `data.at` let you access any recorded data byte by its position as shown in the DataView address column. The result is a data object covering not only the data byte value but also the source and time when it occurred. Here we asked the object for its value with `val()`.

In line 3 we simply return it as a single Lua value and the DataView shows it in the Lua output window.

Results returned by `onchange` are always interpreted as a table of key/value pairs. In case of a single result (as in our example above) the key shows the result, the value is kept blank which gives you something like this:

A8	
----	--

Now let us extend our example to return all information of the data byte at the current cursor position:

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

11.6. INTEGRATED LUA

```
8     }
9 end
```

In line 4 we create a table with `{...}`, add the data object information as key/value pairs (see 17.2.5) and return it. The result in the Lua output window is displayed as:

Source	1
Time	354.84122s
Value	3F

Since the result is already a table (or array) of key/value pairs it is easy to display it as a two-column grid with the left column showing the key and the right column the according values.

You may notice that the output is displayed in a different order. The DataView always arranges the results in the alphabetic order of the key names. Therefore Source, Time, Value.

We will stick to this for now and give you a solution in the following section.

Our next examples demonstrates how to use the passed selection parameters to calculate a simple module 256 checksum.

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

Line 1...7 covers a simple checksum function adding up all bytes from start index `from` to `to` and returning the lower 8 bits (by the remainder or module operator `%`).

The DataView passes a selected data sequence with its start and end address as parameter `selbeg` and `selend`. If they are nil, (no selection at all) the script returns 'No selection!' in line 16.

You can select a data sequence forwards and backwards. The latter may lead to a selection end (index) lower than the start. We test this in line 11 and use a little Lua syntax sugar to swap the start/end values if necessary³.

Afterwards we call the checksum function and return the result as a decimal value.

³Lua provides multiple assignments. In this case Lua first evaluates all values and only then executes the assignments. Pretty nice to realize a value swapping in a single line.

KAPITEL 11. THE DATA VIEW

In our last example we will show you how you can mark (colourize) certain data bytes or sequences relative to the cursor position.

This is especially of interest if you synchronize the DataView by the ProtocolView. Clicking a telegram in the ProtocolView passes the first AND last telegram byte as a selection to the listening DataView. You can use this information in `onchange` to mark the according data bytes in the DataView. For instance to highlight parts of the telegram in the DataView.

You will find an appropriate example (`Modbus-RTU`) in the DataView scripts. We will start with the basics and mark the byte on the previous cursor position red, the cursor byte itself green and the byte on the next cursor position blue.

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

We explain the details in section 11.7.1 but the meaning should be clear enough. The table `cursorcolours` is just a Lua table of key/value pairs. The key is the position relative to the cursor (0 means the cursor itself), the assigned value a RGB colour in hex notation⁴.

In the code above we assign three colours red, green and blue. The function must not return a result if you don't need any output in the Lua output window. Every time you move the cursor (and `onchange` is called), the main window updates its content using the pairs in these tables.

11.6.2 Sorted results

At least back to our first example. We mentioned, that a Lua table does not keep the key/value pairs in a specific order⁵. The DataView output window arranges the result in an alphabetic order but sometimes you expect the output exactly in the same order as you have coded it.

Source	3
Time	2
Value	1

```
1 return {
2   ["Value"] = 1,
3   ["Time"]  = 2,
4   ["Source"] = 3
5 }
```

You can try to force the output order by using some sort pattern. For instance you can add a prefix number separated by a tab "`\t`" to get the desired order with:

Value	1
Time	2
Source	3

```
1 return {
2   ["1\tValue"] = 1,
3   ["2\tTime"]  = 2,
4   ["3\tSource"] = 3
5 }
```

The DataView ignores all text until the first tab (and including the tab) before it displays the key/value pairs in the Lua output window. Nevertheless this is still

⁴0xRRGGBB whereas RR is the hex 8-bit red value, GG the hex 8-bit green value and BB the hex 8-bit blue value.

⁵There is an order but it depends on the internal hash number!

11.6. INTEGRATED LUA

annoying. Especially if you have more than 3 lines and must adapt the numbers every time you rearrange the order in your code.

The next and final solution therefore use a little sort function to add the desired prefix. Here is the complete example code:

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

Line 3 defines a local counter variable (used for the ordering). Line 5...8 is the sort function and returns an increasing prefix "01\t"... "99\t" with every new call⁶.

Now we just have to add the `sort()` function to our key names as shown in line 11...13.

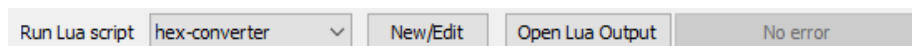
11.6.3 Select and run a Lua script

The analyzer software already comes with some Lua script example for the DataView. One is a number converter which transforms the data sequence at cursor position in different numbers formats and especially helpful if you want to 'read' transmitted number values.

Another script shows you how to calculate a checksum of a data selection.

The DataView groups all Lua related controls below the main window.

There you can select and run a Lua script, open it in the editor for modifications or write a new one.



The button 'Open Lua Output' opens the output window to show the results of the selected script. The output window itself is displayed until you close it again.

Please note!

There is no Lua on/off switch! The integrated Lua doesn't need much resources and therefore can stay active in the background even if you do not open the Lua output window. If you definitely want to suppress any output or marked data bytes just select the **idle** script.

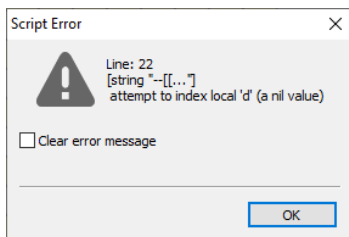
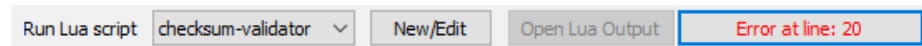
11.6.4 Script errors

Nobody is perfect! Even coding a small script may leads to syntax errors or typos. The editor itself cannot handle the errors (even if you can test code snip-

⁶Note that 11 is ordered between 1 and 2, therefore the explicit 2 digits with a leading zero.

KAPITEL 11. THE DATA VIEW

pets in the editor) because the script is executed by the DataView. If your script contains an error, the DataView informs you about it in the Lua error button:



Script error dialog

This button is inactive as long as no error occurs. In case of an error you can click the button to get more information about the error cause.

The appearing script error dialog indicates the kind of error and the line in your script file where you can fix it in the editor.

The red error message in the error button disappears automatically when you save the corrected script. Or when you enable the clear error message checkbox in the dialog before closing it. But then the error may occur again because you did not solve the cause.

11.6.5 Debugging

Knowing that there is a bug in your script in one thing, solving it quite another one. An appropriate way in script languages (because of the short edit/test cycles) is to insert output messages at specific positions. This allows you to test if the script executes a certain part of code or to check the value behind a variable.

The DataView offers its own debug output window. You can open it by clicking the according entry in the 'View' menu or press Ctrl + Alt + O.

We describe the `modul` later. Here in short. You can output any text or value or combination with the function:

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

You can delegate the debug output to a central function to enable/disable all debug outputs in your script with a single flag.

```
1 DEBUG=true
2
3 function print(...)
4     if DEBUG then
5         local s = ""
6         for i,v in ipairs( arg ) do
7             s = s..v.."\\t"
8         end
9         debug.print( s )
10    end
11 end
12
13 function onchange( cursor, selbeg, selend )
14     local d = data.at( cursor )
15     if d then
16         print( "Cursor:", cursor )
17     end
18 end
```

The function `debug.print(...)` accepts a variable number of arguments, thus we iterate over all parameters in line 6 and build the result string in line 7. Provided that the global variable `DEBUG` is true (line 1) the function outputs the result in line 9.

11.6. INTEGRATED LUA

11.6.6 Template file location

The location of the Lua templates/scripts for the DataView has changed with version 5.0.0.

Linux user find them under:

```
~/IFTTOOLS/SerialAnalyzer/7.0.2/Templates/DataView
```

Under Windows the directory is located under:

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

But you must not worry about this. If you start a new template or save a modified script under a different name, the editor keeps the right path for you. And there is a good reason for this:

The DataView scans this path for new scripts always when you click the template/script selection button. All found template scripts are afterwards listened alphabetically in the opened script list.

You can - of course - save a script under a different location. But we only recommend it when you want to pass a template to a colleague, another computer or for other reasons.

As long as you want to use a template within the analyzer application, it has to stay in the according folder above!

11.6.7 Import a template

As mentioned before: If you want to use a template provided from a colleague or another source you have to store it in the right location, otherwise the DataView will not find it.

Importing or adding a new template to the list of already existing DataView scripts is easy. Just drag and drop the new template file (extension msbtml) into the open DataView window. That's all! The program automatically stores the new template in the according Template/DataView directory and applies it to the current record.

If there is an error, you will get an according note in the status bar error button. The program will also warn you when a template with the same name already exists.

11.6.8 How can I remove waste scripts

The normal script file operation (load, modify, save) is in the responsibility of the editor. Hence the DataView does not provide any file handling except for loading/updating a selected script.

But you can easily start the editor, click the 'Open file' icon in the toolbar and delete or move the not longer wanted file from within the system file dialog or explorer.

11.6.9 Limitations

You can run any operation in Lua, write complex functions and perform extensive evaluations. But the DataView 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

KAPITEL 11. THE DATA VIEW

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 DataView specific Lua extensions

As like the ProtocolView the DataView extends Lua by several modules common to all views providing an integrated Lua. They all are described in detail in chapter 18. The one module exclusively available in the DataView is the `data` module.

A second Lua extension which you perhaps already know from the ProtocolView is the `debug` module. Even if this one is no special part of the DataView, we describe it here too because - as the name suggests - it's purpose is to help you debug your code in the context of the DataView.

11.7.1 The data module

This module provides you random access to all received data bytes by passing the data byte address (or number) and is indispensable when running Lua scripts by the DataView.

Please note that the received data bytes are counted from zero and that the number depends of the displayed data sources A and/or B.

The module let you also colorize data relative to the cursor position. The `data` module in detail:

Function	Description
<code>at</code>	Let you access the value (including 9 bit values) with <code>val()</code> , the direction (or source A, B) with <code>dir()</code> and time stamp of any received data byte with <code>time()</code> .
<code>cursorcolours</code>	an internal table holding pairs of relative position and colour which are used to colourize the data relative to the cursor position.

`data.at`

Returns a data object covering all information about the data byte at the given index/position. The data object allows you to query the data byte value, the direction and time when it was received by the analyzer.

`data.at(index)`

- **index:** The index or address of the data byte in the range of the available data. An invalid index/address (outside the data range or negative) results in a **nil** value.

You can query the data byte value, source and time directly with:

Example

```
1 function onchange( cursor, selbeg, selend )
2   — input your code here
```

11.7. DATAVIEW SPECIFIC LUA EXTENSIONS

```
3     return {
4         ["Value"] = string.format("%02X", data.at( cursor ):val() ),
5         ["Time"] = data.at( cursor ):time() .. "s",
6         ["Source"] = data.at( cursor ):dir()
7     }
8 end
```

But this always starts a new random access with `data.at` for every data object property. A better approach is to store the data object returned by `data.at` in a local variable and use it later.

It is also always a good strategy to check if the result of the `data.at(...)` call returns a valid object. This is not the case, if you click a data cell in the window outside the record range (grey marked as XX). We do this in line 4.

Example

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

data.cursorcolours

This is an internal table of key/value pairs used to assign a certain hex colour value (for instance 0xFF0000 means red) to a relative cursor position (something like 0, 1, 2 but also -1, -2).

Please note! In contrary to the standard Lua indexing the relative position counts from zero! A zero index means the cursor position itself, a negative index counts backwards and a positive index addresses the data bytes after the cursor.

The following example marks the data byte before the cursor red, the cursor itself green and the byte after the cursor blue.

Examples

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

KAPITEL 11. THE DATA VIEW

11.7.2 The debug module

The DataView 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:

Ctrl + **Alt** + **O**

Function	Description
clear	Clears the content of the debug window.
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.
resume	resumes a former suspended output.
summarize	if activated the debug output collects all identical messages and show it only once with the repeating number.
suspend	stops (suspends) the debug output. All further print calls will be suppressed.
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.clear

Clears the content of the debug output window.

debug.clear()

Examples

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

debug.print

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

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

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

Examples

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

11.7. DATAVIEW SPECIFIC LUA EXTENSIONS

The code above outputs the cursor position and the value of the data byte at the cursor position.

debug.resume

The `resume` function continues the previously suspended debug output. It is very unlikely that you will use this function and its counterpart `suspend`. Since they are part of the `debug` module, we nevertheless dedicate them a short description.

debug.resume()

Examples

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

In the example above we stop all debug outputs when the cursor hits an invalid data, for instance when the cursor is clicked on a data field outside the recorded data. As soon as the cursor is back on valid ground, we resume the output. The example is a little bit constructed but should be enough to understand its meaning.

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 `onchange()` 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()

Examples

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

KAPITEL 11. THE DATA VIEW

As long as you move the cursor of data bytes with the same value, the summarize mechanism just collects the message but suppress any output until the cursor hits a different byte. Then it outputs the number of equal bytes and the new one.

`debug.suspend`

This function suppresses 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:

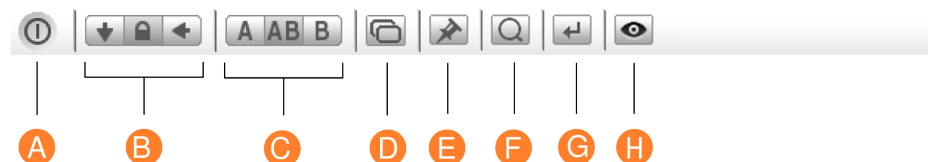
```
12:24:48: My debug message
```

Examples

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

11.8 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.

D New View: Opens a new window with the same sector and settings.

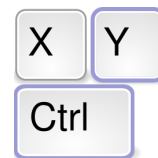
E Pin settings: Applies (pins) the current window settings as default setup when open again.

11.9. SHORT COMMANDS

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

11.9 Short commands

Aktion	Kurzbehl
Online Help for the data monitor	F1
Save selection as region	F4
Save selection as data frame region	Shift+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
Open the debug output window	Ctrl + Alt + O
Close window	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 $1\mu s$. 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 event monitor is the right tool.

Beside this you can use the event monitor also to extract a certain part of a record and save it as a new one, see section [12.4.1](#).

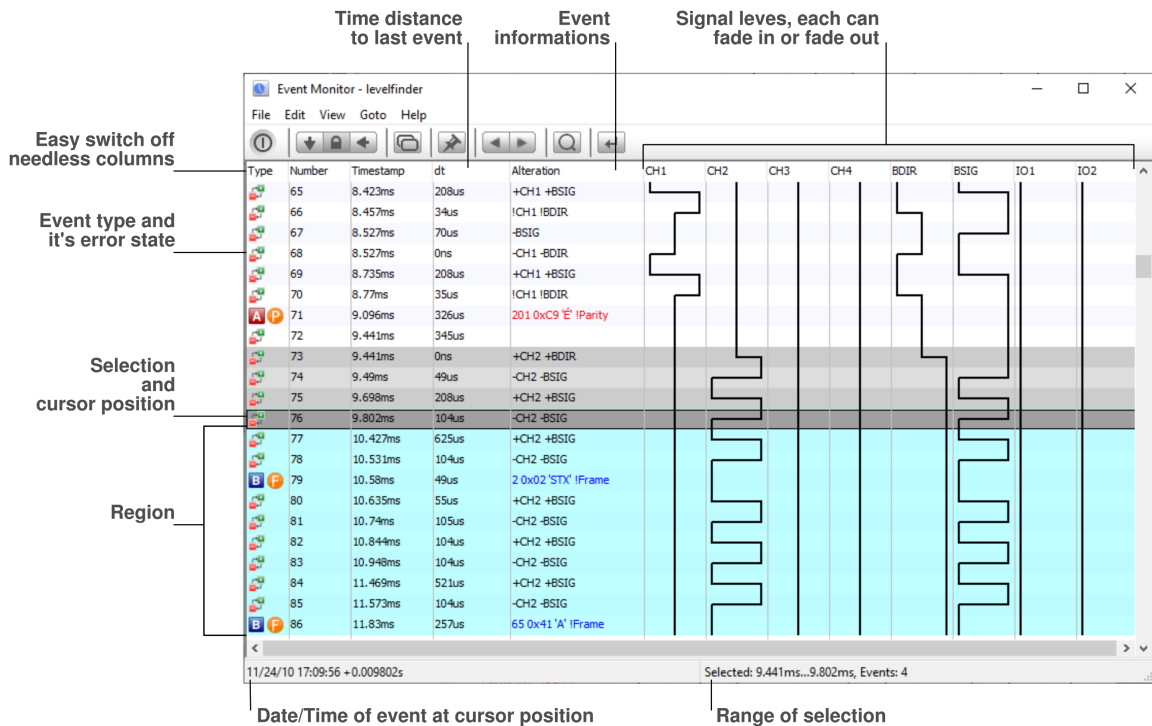
With the LevelFinder not only any static level can be found but also sequence of events and level changes. The single search parameters can be combined optionally with AND or OR and can additionally be combined with a time duration to search for events within a defined time frame or to exclude them.

The search options are described in detail in section [12.3](#).

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.

KAPITEL 12. THE EVENT VIEW

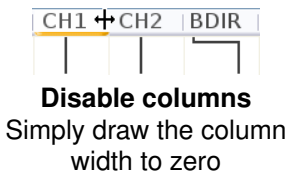


12.1.1 Each line is one event

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.

The individual columns have the following meaning:









- Type**
Type of the event, either a data byte or a signal level alteration.
- Number:**
Number of the event, counted from 1.
- Timestamp:**
The time when the event occurred in $1\mu\text{s}$ relative to the beginning of the record.
- dt:**
Time distance to the former Event.
- Alteration:**
Describes the alteration causing the event, see 12.1.3.
- Signals:**
Signal levels alterations of all analyzer input channels.

12.2. NAVIGATION THROUGH THE EVENT LIST

12.1.2 All event types at a glance

The event monitor distinguishes between the following event types:

Symbol	Event type	Description
	Data byte	Data byte received at data channel A.
	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.
	Framing	Data byte received with a framing error.
	Parity	Data byte received with a parity error.
	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.

12.1.3 Signal alterations

A signal alteration event is recorded every time a signal level of one of the 8 analyzer inputs has changed. The MSB-RS485 distinguishes 3 (tri-state) signal levels: High, low and invalid (idle). The column 'Alteration' gives a description which signal modification(s) caused the event. A prefixed plus (+) sign indicates a switch to the high level, the minus (–) a switch to low and the exclamation mark (!) that the given signal goes back to an idle or invalid level. For instance:

–CH1 +CH2 !CH3

The CH1 signal went low and at the same time (within the scope of the sampling rate) the CH2 signal to high and the CH3 input to an invalid/idle (or undriven) signal level.

–CH1 –CH2

In this example both lines, CH1 and CH2 switched to a low signal level.

Level changes of the data lines

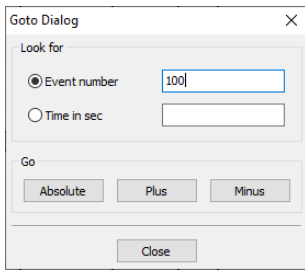
If you do not see level changes of the difference signal inputs CH1...CH4 you have to activate them in the control program. If you are not interested in these changes deactivate them to save computing power and disk space since these events come very often.

12.2 Navigation through the event list

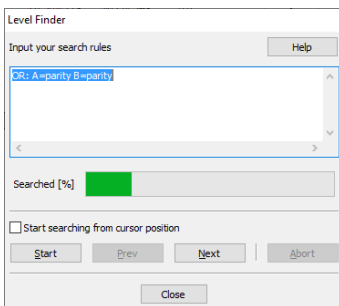
The event monitor offers beside the usual scrolling possibilities by mouse, mouse wheel, scroll bar or arrow and page up/down keys also the directed jump to the next or last event of the same type.

Click onto the desired event and then click the ctrl key together with the down

KAPITEL 12. THE EVENT VIEW



Go to event
absolute, step-wise or
time dependent



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

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

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.

12.3.1 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 corresponds 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.

¹This 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

12.3. EVENT SEARCH WITH THE LEVELFINDER

Correct definition of a condition

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

12.3.1.1 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 descriptions 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 descriptions 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.

12.3.1.2 Formulate a data error

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

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.

12.3.1.3 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 be 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:

KAPITEL 12. THE EVENT VIEW

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='?'
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=~xxxxxxx, B=~xxxxxxx	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.



Example project
in the folder
examples/RS485-
Analyzer

12.3.2 Search input and search

Before you start open in the control program the sample project

`Scan-for-signal-levels.msbrj`

in the `examples\RS485-Analyzer` 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.

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 grey line.

12.3. EVENT SEARCH WITH THE LEVELFINDER

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 CH1 (data A) with CH2 idle

- Click text field and enter `>>AND: A=break CH2=none<<` eingeben

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

Search for an inactive CH1 while CH2 is low and BDIR is high

- Now enter into the text field:
`>>AND: CH1=none CH2=low BDIR=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.

12.3.3 Search for signal changes

Changes are defined by 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:

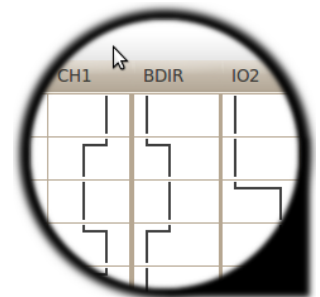
- 1 CH1=high BDIR=low IO2=low
- 2 CH1=none BDIR=none IO2=low
- 3 CH1=none BDIR=none IO2=high

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

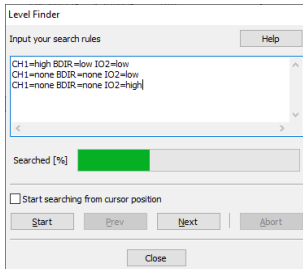
If you made a syntactical error, e.g. a wrong name or an invalid character, the level finder answers with a yellow text field as soon as you start the search.



Looking for a level change?

No problem with the LevelFinder!

KAPITEL 12. THE EVENT VIEW



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



Level duration?
Search with the stop watch

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.

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*.

12.3.4 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.

- 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

12.4. MARK A SELECTION

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 (extension `*.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.

12.4.1 Save a selection as a region

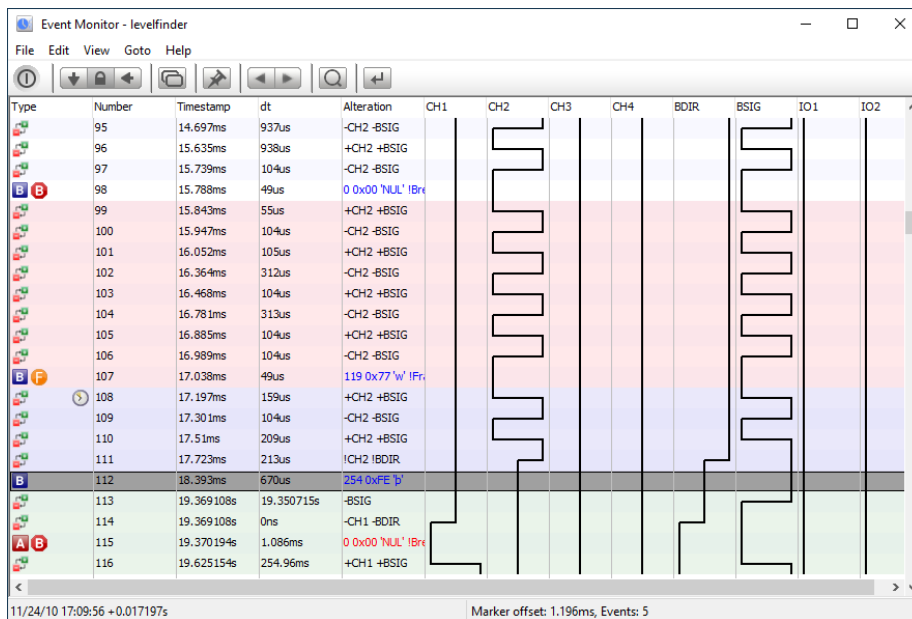
A region serves for marking of certain sections of the recorded data which are of special interest. Contrary to a selection regions are valid for all analysis tools. As soon as a region is added it is visible for all analysis windows.

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 (extension `*.msblog`). You will find further information about regions in chapter 15.

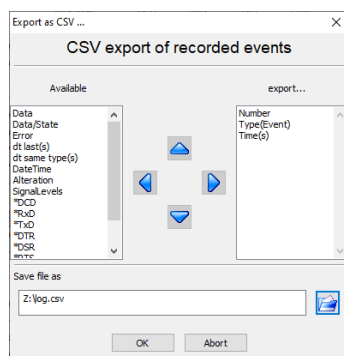
The following picture shows a section in the event list with three visible regions.



KAPITEL 12. THE EVENT VIEW



Data export as CSV for external evaluation



Export dialog

12.4.2 Export a selection as CSV file

The event 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 items:

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.

12.4. MARK A SELECTION

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	Line State							Bit	Bit	Valid State							Bit
15							8	7									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",
```

KAPITEL 12. THE EVENT VIEW

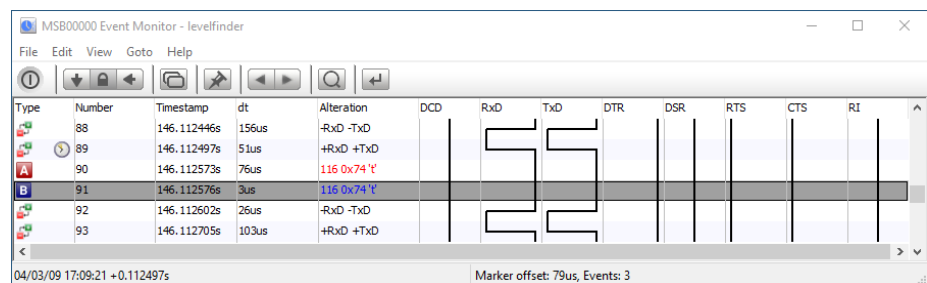
[3] Signal names during exporting

Please note, that the lines do not 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

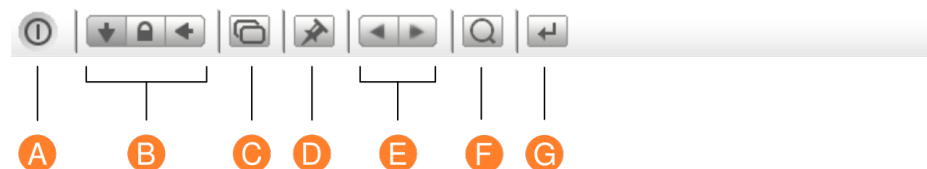
Every event can be marked by right click the event line (item). Next to the right side of the type column a clock symbol is faded in and in the status line the time difference from the marked event to the current event at the cursor position is displayed.

A second right click onto the marked event removes the clock mark again.



12.6 The toolbar

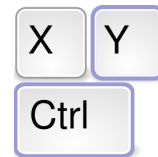
The tool bar is used for a quick access to the most needed functions. Some are identical to other views, some are specific for the event monitor.



- 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 Pin settings:** Applies (pins) the current window settings as default setup when open again.
- E Event dependent scrolling:** Jumps to the last or next event of the same type like the one at cursor position.
- F Event search:** Opens the level finder dialog for event search.
- G 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 (LeviFinder)	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.

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.

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



...and more...



Lua Version 5.3

KAPITEL 13. THE PROTOCOL VIEW

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

The ProtocolView is designed for an optimal usage when examine telegrams which are part of the upper layers in the OSI model.

To avoid redundant accessories the main part of the ProtocolView window simply serves to list the transmitted telegram sequences as shown in the picture with a Modbus RTU transmission.

No	Time	Transmission	Device	Function	Address	Quantity	Byte Count	Reg:1	Reg:2	Reg:3	Cks		
225	231.920139s	Query =>	3	16 Write Multiple Registers	216	3	6	55296	774	1	0339		
226	231.942017s	Response <=	3	16	4	4	03EC						
227	231.948993s	Query =>	3	16 Write Multiple Registers	216	3	6	55296	774	1	0339		
228	231.9705s	Response <=	3	16	4	4	03EC						
229	231.988732s	Query =>	2	03 Read Holding Register	1999	2	73F5						
230	232.094453s	Response <=	2	03 Read Holding Register	4	5	9	3419					
231	232.10732s	Query =>	2	03 Read Holding Register	2099	8	5086						
232	232.213823s	Response <=	2	03 Read Holding Register	16	2	232	58	55	720	720	720	9269
233	232.227891s	Query =>	2	03 Read Holding Register	2108	1	5546						
234	235.248407s	Query =>	2	03 Read Holding Register	2108	1	5546						
235	235.354687s	Response <=	2	03 Read Holding Register	2	64	84FD						
236	235.36929s	Query =>	2	03 Read Holding Register	2110	4	9627						
237	235.48003s	Response <=	2	03 Read Holding Register	8	3615	2969	6211	2679	4859			
238	235.493399s	Query =>	2	03 Read Holding Register	2116	6	8E87						
239	235.600194s	Response <=	2	03 Read Holding Register	12	0	169	0	0	0	0	0	2642
240	235.613863s	Query =>	2	03 Read Holding Register	3259	4	4F37						
241	238.63235s	Query =>	2	03 Read Holding Register	3259	4	4F37						
242	238.738605s	Response <=	2	03 Read Holding Register	8	115	116	105	120	105	120	6188	
243	238.753334s	Query =>	2	03 Read Holding Register	3999	1	0387						
244	238.859423s	Response <=	2	03 Read Holding Register	2	2	857D						

13.1.1 Telegram window

The ProtocolView displays every sequence (or telegram) in a single line. Where a telegram in the data stream starts and where it ends is determined by the selected protocol template script. The same applies - as mentioned before - how the telegram data is displayed.

Each telegram can optionally be 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 se-

quence. 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 ProtocolView...

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. 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 in this field!

13.1.2 Synchronizing

All Views have in common that they can synchronize or lock their displays or always show the last recorded data. That also applies for the ProtocolView. You can choose an autoscrolling behavior to see always the last detected telegram, lock the window to study telegrams at your leisure or synchronize the telegram display with other views.

A left click on the desired telegram let other Views update their display to show their content on the clicked telegram position.

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.



Display modes

13.1.3 Data direction

The IFTOOLS analyzers provide two independent data channels which are assigned to the source or direction of the transmitted data. In full-duplex bus systems like RS422 (or RS232) the meaning is clear. Other field-bus systems use only on connection which is shared by all other bus participants like Modbus.

The analysers are able even to detect the direction in the latter case but it is not always necessary. Anyway, with the data source selector you can display the telegrams of only one source A, B or both A+B.



Direction modes

13.1.4 Open an identical view

Sometimes you want to stay at a current position because there is a telegram you are very interested in. And you want to check it against other telegrams.

You cannot do it in the same window! But you can 'clone' your current ProtocolView window by clicking the 'clone' icon.

Now you have two active ProtocolViews and you can compare different telegrams or sections in your record.



Open a copy

13.1.5 Pin your settings

The analyzer application stores the size, position and views settings by default when you close the main control program.

KAPITEL 13. THE PROTOCOL VIEW

But you can also determine which settings are applied to a new ProtocolView window when you execute it from the control program. In case of our ProtocolView you can specify the used protocol and all other settings like the font, display mode, direction and so on.

By clicking the pin icon in the toolbar, the current settings are stored in the control program as default setup for every new ProtocolView instance.

If you open a ProtocolView afterwards it shows up with your desired settings.

13.1.6 Goto a given telegram number

The ProtocolView numbering all telegrams consecutively. If you want to see for instance the telegram with the number 10204, click the 'goto' icon in the toolbar or press **STRG** + **G**. Input the number and the main telegram windows moves its content to display the desired telegram in the center line.

Inputs greater as the number of available telegrams forces a jump to the current transmission end, negative numbers to the beginning.

Note: Set the display mode to lock or sync. In autoscroll mode the telegram window always jumps to the last received telegram and will not stay on the given telegram number.

13.1.7 Filter control

The filter control is very special and we will explain it in detail in section 13.4. For now only one short comment: Filtering telegrams depends mainly on the used protocol specification. You cannot filter telegrams for command functions (like Modbus) if the protocol specification doesn't include such things.

So the actual filter code is made by the template itself. Not every template offers a filter, but if, you can select/input a filter criterion here.

In our Modbus example you can hide diagnostic telegrams or show only requests/response for one device address. You can even filter for a given address/function combination.

13.1.8 Choosing a range

With the export 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.

13.2 Protocol templates

The analyzer software already provides you with many templates for the most used protocols. More will be added in future software releases. The currently provided field-bus protocols are:

- 3964(R)
- BACnet
- DF1
- DNP3

13.2. PROTOCOL TEMPLATES

- Executive (Vending machines)
- IEC60870-5-101
- IEC60870-5-103
- MDB/IPC
- Modbus ASCII & RTU
- MOVILINK
- NMEA
- P-Net
- Profibus
- SAE-J1587
- SAE-J1922
- SMA-NET
- SSI (synchronous, only PLUS analyzers)
- USS

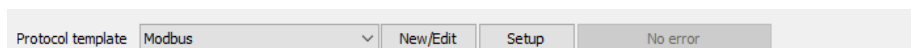
Beside these there are several basic templates serving as a start for protocols using 9-bit values, a certain start and/or end delimiter, a break (like LIN Bus) and more.

You can - of course - also modify all of the standard templates for your own purpose.

13.2.1 Select a protocol template

You can easily adapt another protocol just by clicking the protocol selector below the main window. The adaption to the telegrams in the main window is directly done after selecting a list item.

Thus you can test an transmission with different protocol specifications in an uncomplicated way and - without stopping or restarting the record!



Remember! Regardless of the chosen template the record is never affected! So even if a template gives you not the desired results, just try another one. If no template fits your needs, you can contact the IFTTOOLS support or adapt a template to your protocol by yourself.

13.2.2 Modify a protocol template

The 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.

The predefined templates offer an easy start for own concepts. New templates are automatically added to the protocol list and can be selected as easily as the ones coming with the software.

For this click the [New/Edit](#) button on the right side of the protocol template selector and the ProtocolView opens the according Lua template script in the editor.

KAPITEL 13. THE PROTOCOL VIEW

Since version 5.0 the ProtocolView not longer provides its own integrated editor but use the new and lot better suitable Lua script editor coming as an separate program with the analyzer software.

The editor offers all features and comfort you are expecting from a really good editor. But what makes the editor really good is:

It let you test any Lua code snippets in a sketch buffer and also selected lines when you edit a protocol template. Beside this it offers code frame works for new templates. You will find a detailed description of the editor in chapter 16.

At first you probably will change only small things: the color of the data, the definition of a line end character or the idle time between the telegrams (as in Modbus RTU). As soon as you save your changes in the editor the ProtocolView instantly applies your changes onto the displayed telegrams in the main window (even in an active recording).

If the template is erroneous a respective message is fade in into the status line of the ProtocolView.

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.

You can also copy a predefined template for modification according to your application just by open it in the editor and save it with another name.

13.2.3 Individual protocol setup

When analyzing a field-bus transmission you are often confronted with 'variable' protocol specifications. For example: An IEC60601-5-101 application can require one or two address bytes for the bus participants. Another example is Modbus where you may face a different setup for the interframe idle time.

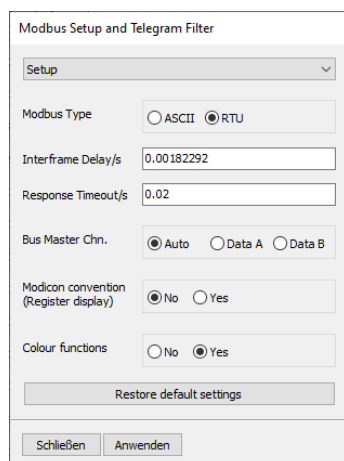
It's clear that you don't want to adapt the template code every time your bus requirement changes. The protocol template API therefore offers you to write your own setup dialog - individual for every template. A existing dialog provided, you can open it every time you need to change some protocol settings simply by clicking the **Setup** button. The image on the left shows the setup for Modbus coded in the Modbus template.

A setup dialog is not mandatory. If a template doesn't contains a dialog code, you get an adequate information. The template dialog feature is covered in all details in chapter 19.

13.2.4 Write a new template

The new editor is the pivotal point to handle all Lua code for the analyzer application. This will also apply when you want to add your own protocol template (therefore the button named **New/Edit**).

In the editor toolbar click the 'new icon' or press **CTRL** + **N** to start a new Lua script file. The editor will ask you what kind of script you want to write.



A setup dialog
here fore Modbus

13.3. TEMPLATE LANGUAGE SYNTAX

Choose 'ProtocolView' and the editor presents you a frame code especially for a protocol template. (You will find more details in the editor chapter 16). We introduce the template language itself in following chapter 13.3.

13.2.5 Template file location

The location of the protocol templates (and all other scripts) has changed with version 5.0.0.

Linux user find them under:

```
~/IFTTOOLS/SerialAnalyzer/7.0.2/Templates/ProtocolView
```

Under Windows the directory is located under:

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

But you must not worry about this. If you start a new template or save a modified script under a different name, the editor keeps the right path for you. And there is a good reason for this:

The ProtocolView scans this path for new scripts always when you click the protocol template selection button. All found template scripts are afterwards listed alphabetically in the opened template list.

You can - of course - save a script under a different location. But we only recommend it when you want to pass a template to a colleague, another computer or for other reasons.

As long as you want to use a template within the analyzer application, it has to stay in the according folder above!

13.2.6 Import a template

Sometimes you may want to use a template provided from a colleague or which you got from another source. As mentioned before, the template must be stored in the right location, otherwise the Protocol View will not find it.

Importing or adding a new template to the list of already existing files is easy. Just drag and drop the new template file (extension msbtml) into the open Protocol View telegram window. That's all! The program automatically stores the new template in the according Template/ProtocolView directory and applies it to the current record.

If there is an error, you will get an according note in the status bar error button. The program will also warn you when a template with the same name already exists.

13.3 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

KAPITEL 13. THE PROTOCOL VIEW

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**

13.3.1 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

REMOVED obsolete?

Filtering telegrams by returning **REMOVED** is not longer recommended because it has a lot of disadvantages, especially for more complicated filter scenarios. See [13.5](#) for a far better solution!

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

13.3. TEMPLATE LANGUAGE SYNTAX

- 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 ASCII coded data as 0-9 and A-F	EOS 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`.

KAPITEL 13. THE PROTOCOL VIEW

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 - $1\mu\text{s}$. 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 > transmission.bytepause( 3.5 ) then return STARTED end
3     return MODIFIED
4 end
```

The code should be clear enough except for the `transmission.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 `transmission` module on page [226](#).

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).

13.3. TEMPLATE LANGUAGE SYNTAX

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 ProfiBus¹, 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 bytes 0x05 and 0x64 (in hexadecimal notation) as a start and synchronisation header. A simple DNP3 frame looks like:

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

KAPITEL 13. THE PROTOCOL VIEW

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 - marks 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 [17.2.8.2](#)).

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 (triggered 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

```

KAPITEL 13. THE PROTOCOL VIEW

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 exist and are 'visible' only in the loop body.

You see: It is always a good idea to declare all variables as **local** in the first place. In case of a need for a globally 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 informations 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 an EOS, the telegrams from port B (CH2) were finished with a LF.

```
1 function split( data, interval, 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 adaptation 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 17.1 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.

13.3.2 Individual displaying of the datagrams

All the formatting 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 formatting instructions
3 end
```

The principle of the new output mechanism is based on a set of individual rectangular boxes in a row, whereby each row represents a single telegram.

Every box contains a caption line and a text specified by the user. Foreground

13.3. TEMPLATE LANGUAGE SYNTAX

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.

Caption →	Time	Addr	Func	Startaddr	Quantity	Cks OK
Content →	0.080646s	1	Read Holding Register	0	114	efc5

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:

```

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

```

Time
0.137345

A simple box

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

KAPITEL 13. THE PROTOCOL VIEW

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 response².

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()` function³.

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 classical 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:

²The Modbus RTU template makes use of this.

³You are not longer limited to the current and previous telegram as in former program versions.

13.3. TEMPLATE LANGUAGE SYNTAX

```
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 haven't explain it yet and you may still ask yourself what's the difference in using a dot or a colon.

KAPITEL 13. THE PROTOCOL VIEW

The dot in `telegrams.this()` accesses the function `this` of the `telegrams` module. A module is - simply spoken - organized as a table and the function `this` is one of several table entries. The dot here refers to the `this` 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 initiated with the current telegram (line 1), then 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 13.8.7. 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()
```

13.3. TEMPLATE LANGUAGE SYNTAX

```
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,
8         width=2 } }
9     box.text{ caption="EOS", text=tg:dump{ first=-2, last=-1, width=2 }
10    }
11 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

KAPITEL 13. THE PROTOCOL VIEW

`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.msbrj` 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.msbrj

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 `0xFFFFFFFF`.

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
```

13.3. TEMPLATE LANGUAGE SYNTAX

```
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
16  source
17  local fc = textcolors[ tg:dir() ]
18  local bc = backcolors[ tg:dir() ]
19
20  — display time
21  box.text{ caption="Time", text=tg:time(), fg=fc, bg=bc }
22
23  — display all data as hex
24  box.text{ caption="Data (hex)", text=tg:dump(), fg=fc, bg=bc }
25 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 be 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
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
```

KAPITEL 13. THE PROTOCOL VIEW

```
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 2.339189	SOF :	Data (hex) 030 032 030 032 043 034	EOS 00d 00a
Time 2.351468	SOF :	Data (hex) 030 032 030 032 030 046 031 035 036 041 034 032 037 044	EOS 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 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 2.339189	SOF :	Data (hex) 30 32 30 32 43 34	EOS 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.

⁴The string module is part of the Lua language.

13.3. TEMPLATE LANGUAGE SYNTAX

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 214.

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 }
```

KAPITEL 13. THE PROTOCOL VIEW

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
23  — the function number field
24  box.text{ caption="Function", text=bindata:byte(2), fg=fc, bg=bc }
25
26  — is it a response?
27  if #bindata >= 7 then
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",
```


13.3. TEMPLATE LANGUAGE SYNTAX

```
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,
42             bg=fc }
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 like⁵:

```
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 somewhere 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
```

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

KAPITEL 13. THE PROTOCOL VIEW

```
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    ...
12
13
14
15
16
17
18
19
20
21
22
23    — the function number field
24    box.text{caption="Function", text=GetFunctionName( bindata:byte(2) ),
25              fg=fc ,bg=bc}
26    ...
27 end
```

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 transferred 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.

13.3. TEMPLATE LANGUAGE SYNTAX

The function `string.unpack` 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. The function `unpack` replaces the older `bunpack` and is part of the Lua `string` module since the analyzer software uses Lua 5.3.

```
val1, ..., pos = string.unpack( format, sequence, position )
```

The function returns at least two values. The first value(s) are always the result of the transformation regarding the format parameter. This parameter also specifies how many values are returned in total.

The last result indicates the string position where the next conversion should take place.

Before we go back to our tutorial, here are some examples which may give you an idea how the `string.unpack` works.

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

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 Little-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 `string.unpack` can provide us with the real numbers behind this sequence. We start with the 32-bit integer:

```
2 i, pos = string.unpack( "<i", seq )
```

The first argument of the call `string.unpack` is always the format string. It specifies the kind of data, which we expect at the given (optional) position. Since the default position is equal to the string start, we can leave it out.

The second parameter is the byte sequence we want to transform. The magic behind the `string.unpack` 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 Lua 5.3 online documentation here:

<http://www.lua.org/manual/5.3/> or in the according section [18.2.3](#).

In our example above the format string "<i" indicates a 32 bit 'signed integer' in little endian. `string.unpack` returns two values: The decoded integer

KAPITEL 13. THE PROTOCOL VIEW

(the result is 75000) and the new position in the given byte sequence after the decoding took place. Here `pos` is 5 because the next value (the floating point number) starts after 4 bytes are consumed by the "`<i`" specification. Remember: "`<i`" stands for a 32-bit signed integer which means 4 bytes in the passed sequence.

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 f, pos = string.unpack( "<f", seq, 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.

And if we don't have a need for the `pos` result, we can simply omit it. It is the last value in the result and Lua doesn't complain when you skip it.

```
i, f = string.unpack( seq, "<i<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
12    — access the current telegram
13    local tg = telegrams.this()
14    local bindata = base16.decode( tg:string():sub( 2, -3 ) )
15
16    — select the text and background color depending on the data
17    source
```

13.3. TEMPLATE LANGUAGE SYNTAX

```
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,
24          fg=bc }
25
26  — the device address
27  box.text{ caption="Address", text=bindata:byte(1), fg=fc, bg=bc }
28
29  — the function number
30  box.text{ caption="Function", text=GetFunctionName(bindata:byte(2)),
31          fg=fc, bg=bc }
32
33  if #bindata >= 7 then
34      — it is a response
35      local value = string.unpack( "<f", bindata, 3 )
36      box.text{ caption="Value", text=value, fg=fgColor, bg=bgColor
37              }
38
39  end
40
41  — the checksum byte is always the last byte in bindata
42  box.text{ caption="Chksum",
43          text=string.format("%02X", bindata:byte(-1)),
44          fg=fc, bg=bc }
45
46  — end of frame CRLF
47  box.text{ caption="EOF", text=tg:dump{ first=-2 }, fg=bc, bg=fc }
48
49  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

KAPITEL 13. THE PROTOCOL VIEW

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%%", "%.2fmBar"}
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 value = string.unpack( bindata, "<f", 3 )
11    box.text{ caption="Value", text=GetFunctionValue( fnc, value ), fg=fc, bg=
12         bc }
```

The code should be self-explanatory maybe except for the "%.2f%%" 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=0xFFFF, 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.

13.4 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.

13.4.1 Show and hide (filter) complete telegrams

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
```

KAPITEL 13. THE PROTOCOL VIEW

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.

Nevertheless accessing the content (input) of the filter control is of use in a different manner. We come back to this special case in the next section. Here we concentrate on filtering complete telegrams in the way of hide/show only certain ones.

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 `DataGridView`:

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 inputted 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 hidden 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
```

KAPITEL 13. THE PROTOCOL VIEW

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 "Show all , 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: Show all, 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.mshtml` 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
```

13.4.2 Choose between different telegram display formats

As mentioned above: It doesn't make any sense to hide unwanted telegram types in the `out()` function because this will not simply remove the specified telegram. It will present you with an empty space instead.

But there are other applications which make it preferable to modify the display of the telegram depending on some user input. For example: You have an embedded protocol and want to switch on/off the outer protocol layer. Or something more trivial which lead us back to our example. You want to display the pressure and temperature in a metric or Anglo-American measurement systems, means: You want to display the pressure in **psi** or **bar** and the temperature in **°C** or **°F**.

In the filter control you will let the user select one of the two available systems of measurement. Here we go: We define two filter items to make the selection clear from the beginning.

```
1 function filters()
2     return "Metric , Anglo-American"
3 end
```

The filter input is passed as a `filter` parameter to the `out()` function like it already works with the `split()` function. In the `out()` function you can check the input against different strings and use the result to apply a conversion function to the temperature or pressure.

The following code snippet may give you an idea how it works. All displayed values in the tutorial are handled in one function: `GetFunctionValue()`. This function is called with two parameters, a number indicating the value type and the value itself.

In line 4 we check the filter control input. A 'Metric' selection doesn't need any calculation because the exchanged values are already in °C and mbar. The new modifications are covered in the following `else` block at line 8.

First we change the unit formats (line 9) and add two fractional digits for the psi values. Line 10 is pure Lua magic and shows once again how easy you can solve problems which use a lot of code in other languages. Here our goal is to have different conversion functions depending on the input variable (temperature, moisture or pressure).

Line 10 use a hash table (or associated array) with three anonymous functions. Every function is called by the key 1, 2 or 3 and equal with the type temperature, moisture or pressure. In line 18,19 we simply call the right conversion by passing the number parameter. Compare this with line 6 for a better understanding.

```
1 function out( filter )
2     — skip unchanged code here
3     function GetFunctionValue( number, value )
4         if filter == "Metric" then
5             local formats = { "%.2fC", "%.2f%%", "%.2fmBar" }
6             return string.format( formats[ number ], value )
7         else
8             — conversion factor for Temperature, Moisture and Pressure
9             local formats = { "%.2fF", "%.2f%%", "%.2fpsi" }
10            convs = {
11                — function to convert celsius to fahrenheit
12                [1] = function(c) return c * 1.8 + 32 end,
13                — the mositure is always in percent
14                [2] = function(m) return m end,
```

KAPITEL 13. THE PROTOCOL VIEW

```
15         — function to convert mbar to psi
16         [3] = function(p) return p * 0.01450377 end
17     }
18     return string.format( formats[ number ],
19                           convs[number](value) )
20 end
21 end
```

Just replace the `GetFunctionValue()` with this new version and add the parameter filter in the `out()` function. With the already modified `filters()` function the template now lets you toggle between an Anglo-American and metric value output.

13.5 New filter mechanism

Until the published release of the beta 6.1.0 filtering certain telegrams out of the received data stream has to be done in the `split()` function. But this approach has one major disadvantage:

You can only hide a telegram sequence (by returning `REMOVED`) as long as the actual data byte in the `split(data, ...)` function belongs to the telegram you want to hide! As soon as the passed data byte is part of a new telegram sequence, the former telegram is out of reach and you cannot remove it any longer! A short example should clarify this:

Modbus RTU telegrams are split by the idle time between two consecutive bytes. If the pause between them is greater as the specified idle time, the current byte in the `split` function is indicated as the start of a new Modbus RTU telegram. Here the basic split rule:

```
1 function split( data, intval, alter, str )
2     if intval > transmission.bytepause(3.5) then
3         — data is the first byte of the next telegram
4         return STARTED
5     end
6     return MODIFIED
7 end
```

Imagine you are looking for Modbus telegrams with an invalid checksum (which is often an indicator for deeper errors in a Modbus transmission). To do so you want to hide all correct telegrams. Modbus specifies a CRC16 checksum (Cyclic Redundancy Check of all bytes with a 16 bit result) which is added to the telegram body as two subsequent bytes. The details are unimportant. Important is only the fact, that you cannot validate the checksum as long as the telegram is not completely passed to the `split()` function. And that's the crux of the matter!

The only way to detect a complete Modbus RTU is when the first byte of the next following telegram is passed to the `split()` function⁶. Since you cannot return `REMOVED` for all data save the current one and at the same time return `STARTED` for the actual data byte, the situation is intractable.

There are other scenarios which too showing the current filter approach as not longer contemporary. For example: you want to show only Modbus requests without responses. In doing so you have to hide all correctly received

⁶You can of course estimate the telegram length by evaluating the content. But this is not only very time consuming, it especially does not work for invalid or damaged telegrams.

13.5. NEW FILTER MECHANISM

request/response pairs. To achieve this you must wait for the response before you can decide to remove the response AND the FORMER received associated request. And this requires to remove telegrams earlier handled by the `split()` function - which does not work.

At least another (also Modbus) example before we start describing the new mechanism:

Imagine you are looking for Modbus exceptions. Exceptions are responses by a Modbus slave indicating an error condition. Something like an invalid register or a not supported function. In such a case it would be nice to see only the request/exception pairs and hide all other telegrams in the ProtocolView window. Also here you have to wait for the completeness of the response before you can decide to remove both - the former request and the current response - when it's not an exception!

Crucial of the new filter mechanism are two features.

- 1 You are always working with complete telegrams and must not longer care about dividing the data stream into single telegrams. This is still the task of the `split()` function.
- 2 You can access any already received telegram which means: You can examine (and remove) not only the last (complete) telegram but also all former telegrams.

The new mechanism takes place in the single function `split_complete(no)` and does not affect older template code:

```
1 table function split_complete( no )
2   — query direction , data sequence and time of the last telegram
3   local dir , seq , time = sequences.get( no )
4   — examine the telegram content
5   ...
6 end
```

The function `split_complete` is called every time a new telegram is completed by the `split` function. It releases you from doing this by yourself (which is sometimes really hard as noted before) and at the same time reduces the split code to its core task.

The passed value `no` contains the number of the last completed telegram and serves as a functional parameter to query not only the current but also former telegram information. This includes the telegram direction (source), the telegram data (as a Lua string) and the telegram time (timestamp of the first telegram byte). Accessing the last telegram is easily done with:

```
1 local dir , seq , time = sequences.get( no )
```

The value of `no` doesn't matter. Important is, that `no` always represents the last completed telegram number and `no - 1` the second last and so on. It is - of course - the responsibility of the template author to check that the expression $no - n \geq 0$ is never invalid!

So far we didn't say anything about how to remove unwanted telegrams in the `split_complete()` function. Since the new mechanism was designed to filter not only single telegrams but also a range of telegrams the result of the

KAPITEL 13. THE PROTOCOL VIEW

function is specified as a table (a pair) with two values `{from,to}`.

This is explained best by some some real examples.

First we want to hide all telegrams in a Modbus transmission except for a given slave. In Modbus the master accesses a slave by its device address. This is the first byte in a Modbus sequence and luckily the slave responses with the same (its) address in the first byte too. So to hide all unwanted telegrams we just have to check the first byte of each telegram and remove it, when it's unequal to the address we want to display. For a better understanding, here a short description of Modbus (RTU) telegrams:

ADDR	FUNC	DATA PAYLOAD	CRC16
1 byte	1 byte	0 to 252 bytes	2 bytes

The data payload is of no interest. Important is only the first address byte.

```
1 local addr = 5
2 function split_complete( no )
3     — query direction, data sequence and time of the last telegram
4     local dir, seq, time = sequences.get( no )
5     — check the address byte
6     if seq:byte(1) ~= addr then
7         return {no,no}
8     end
9 end
```

The filter code is simple. We query the template content in line 4 and assign it (beside other informations) to the local Lua string `seq`.

In line 6 we check if the first byte of the string is unequal to the (assumed address value 5 in variable `addr` (see line 1)). If the condition is true we return a table with two values specifying the range of the telegrams we want to remove (line 7). You must not return a table when the removing condition is false. But every time you want to remove a telegram you must return a table with the first and last telegram number (`no`). In case of a single telegram like here it is just the same number for first and last, which is `return{no,no}`⁷.

Please note that a table in Lua is defined in curly brackets!

Our next example makes use of this feature. This time we want to examine a Multi-Drop-Bus (in short MDB) transmission. This kind of protocol is widely used in vending machines. Every time you put a coin in such a machine to get a hot cup of coffee or a chocolate bar the internal components in the vending machine communicates with each other via this protocol. The details are of no further interest. For us one speciality of the protocol will serve as a good example to remove two consecutive telegrams.

The MDB protocol is a master/slave protocol like Modbus where the master always initiate the communication with a request or command. Not all master requests must be answered by the slaves (the several vending machine components). And sometimes the answers are only simple acknowledgements.

One master request is the POLL command. In simple terms it asks the addressed component if there are some status changes. Because the state is

⁷If you want to remove the second last telegram, the return expression is `return{no-1, no-1}`

13.6. INDIVIDUAL FILTER DIALOGS

unchanged most of the times, all these POLL commands are mostly of no interest for the analysis and it would be nice to hide them because they make up a large part of the record. An unchanged status is answered by a slave (periphery device) with a single ACK. The ACK is specified as `00h` with a set mode bit (MDB is a 9-bit protocol using the parity bit as mode bit) but we can ignore the mode bit here.

Our goal is therefore to hide all telegram pairs with a POLL request and a single `00h` byte ACK response. The following code does exactly that:

```
1  — Master/Peripheral direction
2  MASTER = 1  — Master on CH1
3  PERI = 2    — Peripheral on CH2
4
5  function split_complete( no )
6      local dir, seq = sequences.get( no )
7      if no > 0 and dir == PERI and #seq == 1 and seq == "\x00" then
8          — device answered with ACK, if the former telegram
9          — is a POLL, remove both
10         local lastDir, lastSeq = sequences.get( no - 1 )
11         if #lastSeq == 2 and isPoll( lastSeq:byte(2) ) then
12             return {no-1,no}
13         end
14     end
15 end
```

Line 2 and 3 are just other names for the direction or source/origin of a telegram. In our application the Master is connected with CH1, all the peripherals with CH2.

We require all information about the current (last completed) telegram sequence and assign it to the variables `dir` and `seq`, see line 6.

In the next line we make sure, that we have at least 2 completed telegrams ($n > 0$) because we want to access $no - 1$. We discussed this in the former sections. We also check, if the telegram is a slave ACK (a single `00h` byte)⁸.

If so, we query the direction and content of the telegram before (the assumed request, line 10). A POLL command is always two byte long. The device address and command are coded in the first byte, a simple checksum in the second byte. In MDB the raw POLL command value depends on the slave type. For a better understanding we outsourced the comparison in the `isPoll(...)` function. But you will find the complete code in the MDB template.

Here the important thing is only, that in case of a matching telegram pair (POLL request and single ACK response) we have to return a range of two telegrams. The former $no - 1$ request and the current no response. See line 12.

13.6 Individual Filter dialogs

The `split_complete()` function provides you with a lot of opportunities to adapt the template to your special analysis situations. But it would be a poor solution if you have to edit the template every time you want to change the filter parameters or to add new filters at all.

⁸We compare the telegram `seq` with a null byte string, but you can reach the same with `seq:byte(1) == 0`

KAPITEL 13. THE PROTOCOL VIEW

Luckily the `split_complete()` function fits very well with the GUI dialog features of the ProtocolView. We discuss the dialog feature in detail in chapter 19. Let us go back to our first example where we filtered a Modbus transmission for telegrams accessing a certain bus participant by using the device address in each telegram sentence.

```
1 local addr = 5
2 function split_complete( no )
3     — query direction, data sequence and time of the last telegram
4     local dir, seq, time = sequences.get( no )
5     — check the address byte
6     if seq:byte(1) ~= addr then
7         return {no,no}
8     end
9 end
```

To edit line 1 every time you want to change the address is annoying. A better way is to provide a small dialog where the user can edit the wanted address by itself. The `split_complete()` function is called by the same Lua interpreter which is responsible for the splitting of the data stream. Since the interpreter which executes the dialog interface is a different one, we must 'share' variables like the address value in the `widgets` environment or name space.

Sounds complicated - but is not. To share a variable between two Lua interpreters is like to share global variables between two functions. You just have to add a `widgets.` in front of the variable name like `widgets.VARIABLE_NAME`. Let us take a look of the following code to make it clearer:

```
1 — preset the used GUI variables.
2 if not widgets.FILTER_DEVICE_ADDRESS then
3     widgets.FILTER_DEVICE_ADDRESS = 1
4 end
5 if not widgets.FILTER_DEVICE_ADDRESS_ENABLE then
6     widgets.FILTER_DEVICE_ADDRESS_ENABLE = 0
7 end
8
9 function split_complete( no )
10     — apply address filter only when enabled via checkbox
11     if widgets.FILTER_DEVICE_ADDRESS_ENABLE == 1 then
12         — query direction, data sequence and time of the last telegram
13         local dir, seq, time = sequences.get( no )
14         — check the address byte
15         if seq:byte(1) ~= widgets.FILTER_DEVICE_ADDRESS then
16             return {no,no}
17         end
18     end
19 end
20
21 function dialog()
22     widgets.SetTitle( "Modbus Device Filter " )
23     — switch on/off the address filter
24     widgets.CheckBox{ name="wxFilterDeviceAddressEnable",
25                     label="Device Address", row=1, col=1,
26                     value = widgets.FILTER_DEVICE_ADDRESS_ENABLE == 1
27                     }
28     — input the device address
29     widgets.SpinCtrl{ name="wxFilterDeviceAddress",
30                     min=1, max=255, row=1, col=2,
31                     value = widgets.FILTER_DEVICE_ADDRESS }
32 end
```

13.7. EXPORT TELEGRAMS

```
33 function apply()  
34     — filter device address (enable checkbox and address field)  
35     widgets.FILTER_DEVICE_ADDRESS = widgets.GetValue(  
36         "wxFilterDeviceAddress")  
37     widgets.FILTER_DEVICE_ADDRESS_ENABLE = widgets.GetValue(  
38         "wxFilterDeviceAddressEnable")  
39     return "Reload"  
40 end
```

In the lines 2-7 we define the global and shared variables to hold the device address and the filter enable flag.

The following `split_complete()` function is the same as before but now uses the shared address variable for the comparison (line 15) and is only executed, when the enable flag is true (line 11).

Function `dialog()` is responsible for the user interface. Here we simply use a checkbox to switch on/off the address filter (line 24) and a `SpinCtrl` to let the user input an address between 1...255 (line 28).

As mentioned before: All details for writing a dialog are covered in chapter 19. In this chapter we also address how you can store the dialog settings in a persistent way.

Last but not least the `apply()` function. This function is called every time the user clicks the dialog apply button. Here we assign the user input to the according 'global' variables (line 35 and 37) and return 'RELOAD' to instruct the `ProtocolView` to parse and filter all telegrams again by reloading the transmission.

13.7 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 `ProtocolView` 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.

13.7.1 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

KAPITEL 13. THE PROTOCOL VIEW

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. Then 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.

13.7.2 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

13.7. EXPORT TELEGRAMS

side of the progress gauge in the status bar. And you can continue examining the record while a longer running export is in progress.

13.7.3 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.

13.7.4 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).

13.7.5 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:

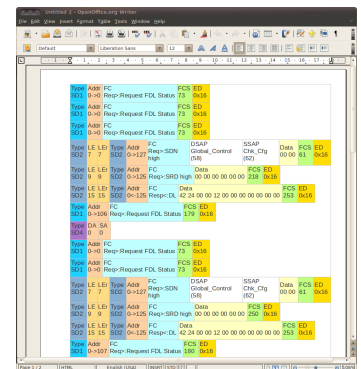
```
Src(Master) Dest(1) Fnc(Read Coils) Desc(Addr=0, Quantity=10) Cks(0DBC)
Src(1) Dest(Master) Fnc(Read Coils) Desc(Byte count=2) Data(00 00) Cks(FCB9)
```

13.7.6 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 }
```



Open Office Writer with html telegram export

KAPITEL 13. THE PROTOCOL VIEW

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.

13.7.7 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!

13.8 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.

- **box module** : The box module is responsible to display the data of each telegram.
- **debug module** : The debug module let you output ProtocolView related 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.
- **sequences module** : The sequences module is only available in the function `split_complete()` and let you access all already completed telegrams. It is especially important for telegram filter mechanisms.
- **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.
- **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 occurring 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.
- **widgets module** : Provides you with individual user interface elements to write your own template setup and/or filter dialogs. This also includes a special variable sharing method between the dialog and the other template functions and an automatism to permanently store/restore dialog settings. The module is described in its own chapter 19.

The several types, functions and modules in alphabetic order:

13.8.1 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.

KAPITEL 13. THE PROTOCOL VIEW

Function	Description
setup	New! Let you add label/colour pairs for a general box coloring and increase the number of lines in a box.
space	The space box inserts just an empty space with a width given in characters or pixel.
text	A common box with free definable caption, text content, foreground color (text and outline) and background.

box.setup

With `box.setup` you can assign certain box properties as default settings to all boxes in your script at a central location. The settings in the `setup` overrules all other individual box parameter, especially the background colour. By this you can for instance preset all telegram fields with the caption 'Checksum' with a given colour and must not do it in several code lines again and again.

Example

```
1 box.setup{
2   lines=3,
3   colours = {
4     ["Checksum"] = 0x00FF00,
5     ["Source"] = 0x80FFFF
6   }
7 }
```

And! With version 5.0.0 the number of lines in a box are not limited to 2 anymore (a caption and a text). You can chose between 1 to 4 lines specifying the height of all boxes as text lines.

One line only shows the caption, but with a box height 3 or even 4 you can pass two or three lines to the internal box text parameter.

Example

```
1 box.setup{ lines=4 }
2 box.text{ caption="TEST", text="line1\nline2\nline3" }
```

Please note the 2 line feeds `\n` between the passed lines in line 2 above.

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 'M' characters.
- **px**: the width defined as pixel.

Example

13.8. PROTOCOLVIEW SPECIFIC LUA EXTENSIONS

```
1 function out()
2   — insert a small space with the width of two space (blank)
3     characters
4   box.space{ em=2 }
5   — insert an empty space with the width of 50 pixel
6   box.space{ px=50 }
7 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, em=0, px=0*] }

- **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.
- **em**: Optional width defined as count of 'M' characters.
- **px**: Optional box width in pixel. The default width is the box text width.

Example

```
1 function out()
2   box.text{ caption="Caption", text="Some text", px=100,
3           fg=0xFF0000, bg=0xAADDFF }
4 end
```

13.8.2 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:

Ctrl + **Alt** + **D**

Function	Description
clear	Clears the content of the debug window.
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.
resume	resumes a former suspended output.
summarize	if activated the debug output collects all identical messages and show it only once with the repeating number.

KAPITEL 13. THE PROTOCOL VIEW

suspend	stops (suspends) the debug output. All further print calls will be suppressed.
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.clear

Clears the content of the debug output window.

debug.clear()

Example

```
1  — a global counter holding the number of error responses
2  errorCounter = 0
3
4  function split( data, intval, alter, str, filter )
5      if alter or intval > transmission.bytepause(3.5) then
6          — a function byte > 0x80 means an error response in Modbus
7          if #str > 1 and str:byte(2) >= 0x80 then
8              errorCounter = errorCounter + 1
9              debug.clear()
10             debug.print( "Error Counter: "..errorCounter )
11         end
12         return STARTED
13     end
14     — all other bytes extend the current telegram
15     return MODIFIED
16 end
```

The example above counts all error responses in a Modbus RTU transmission. A error response or message is indicated by a function value (second byte in a Modbus RTU telegram) with a set MSB (most significant bit). Since only the `split()` function 'sees' all data, the debug output must be placed in the `split()` function! Line 9 clears the former debug output before replacing it with the current counter.

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() )
```

13.8. PROTOCOLVIEW SPECIFIC LUA EXTENSIONS

```
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 )
```

KAPITEL 13. THE PROTOCOL VIEW

```
3
4 function split( data, intval )
5     if intval > transmission.bitpause( 33 ) then return STARTED end
6     return MODIFIED
7 end
8
9 function out()
10     — your output code...
11 end
```

debug.suspend

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.

13.8.3 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 be used in function `out()`.

Function	Description
<code>data</code>	returns the current data as 9 bit value.
<code>dir</code>	returns the source or direction of the current data.
<code>isbreak</code>	returns true if the current byte is a break.
<code>level</code>	returns the current signal level of the given line when the data event occurred.
<code>number</code>	returns the event position in the record counted from zero.
<code>time</code>	returns the time stamp of data event in seconds since the start of the record.

event.data

Returns the data of the current data event as a 9 bit value. It is the same as the passed `data` parameter and is listed here in the sake of completeness.

event.data()

Example

13.8. PROTOCOLVIEW SPECIFIC LUA EXTENSIONS

```
1 function split( data, intval, alter, str )
2   — test for LF
3   if event.data() == 0x0A then
4     return COMPLETED
5   end
6   return MODIFIED
7 end
```

event.dir

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

KAPITEL 13. THE PROTOCOL VIEW

```
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.number

Returns the event position or number in the record stream. The number is counted from zero. Normally you don't need this function but it could be useful in case you want to know, if some other events (line state changes) were recorded. The latter depends on your record settings.

event.number()

Example

```
1 lastNumber = 0
2 function split( data, intval, alter, str )
3   if event.number() ~= lastNumber + 1 then
4     lastNumber = event.number()
5     — a line state event has occurred since the last call
6     return COMPLETED
7   end
8   lastNumber = event.number()
9   return MODIFIED
10 end
```

event.time

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

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

13.8.4 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

13.8. PROTOCOLVIEW SPECIFIC LUA EXTENSIONS

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
<code>changed(signo)</code>	returns true if the given line (signo 1...8) has changed since the last call.
<code>count(signo)</code>	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

```
1 function split( data, intval, alter, str )
2     local RTS = 6
3     local CTS = 7
4     if event.dir() == 1 then
5         if event.level( RTS ) == 1 and linestates.changed( RTS ) then
6             return STARTED
7         end
8     else
9         if event.level( CTS ) == 1 and linestates.changed( CTS ) then
10            return STARTED
11        end
12    end
13    return MODIFIED
14 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)`

KAPITEL 13. THE PROTOCOL VIEW

- **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
```

13.8.5 The sequences module

This module provides only one function and serves to access information about former received telegram sequences in the `split_complete()` function. These are the telegram direction (source), the telegram data as a Lua string and the telegram timestamp (of the first telegram byte).

Differences between `sequences.get` and `telegrams.at`

There is another method to access the received telegrams which you have become accustomed to use in the `out()` function. So why a different new one?

First: `telegrams.at(index)` returns not only completed but also telegrams actually still in progress.

Second: `sequences.get(no)` is highly optimized to make as little performance impact as possible since if used it is executed with every completed telegram. Therefore it only returns three results and not an object.

Function	Description
<code>get</code>	returns direction, data and time of the telegram with the given index.

`sequences.get`

Returns direction, data and time of an earlier received and completed telegram with the given index relative to the passed telegram number (`no`) in the `split_complete(no)` function.

`sequences.get(index)`

- **index**: Telegram number.

Example

13.8. PROTOCOLVIEW SPECIFIC LUA EXTENSIONS

```
1 local addr = 5
2 function split_complete( no )
3     — query direction, data, time of the last received telegram
4     local dir, seq, time = sequences.get( no )
5     — check the address byte
6     if seq:byte(1) ~= addr then
7         return {no,no}
8     end
9 end
```

13.8.6 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
<code>get</code>	returns the content of the global variable with the given name.
<code>set</code>	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'.

KAPITEL 13. THE PROTOCOL VIEW

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 then 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.

13.8.7 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).

13.8. PROTOCOLVIEW SPECIFIC LUA EXTENSIONS

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).
datetime	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. <code>dump</code> 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.
geterror	Returns the error state (frame, parity, break) of the data byte at the given telegram position. The results are: 0:no error, 1:frame, 2:parity, 4:break.
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.

KAPITEL 13. THE PROTOCOL VIEW

Example

```
1 function out()
2   — access the current telegram
3   local tg = telegrams.this()
4   — shows the first byte in the telegram as decimal value
5   box.text{ caption="First", text=tg:data( 1 ) }
6   — shows the last byte in the telegram as decimal value
7   box.text{ caption="Last", text=tg:data( -1 ) }
8 end
```

telegram:datetime

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:datetime(*INDEX*)

- **INDEX** index of the requested byte.

Example

```
1 function out()
2   — access the current telegram
3   local tg = telegrams.this()
4   — shows the pause between the stop bit of the first byte and the
5   — start bit of the second byte (subtract sending time)
6   local delay = tg:datetime( 2 ) - tg:datetime( 1 ) - transmission.
7   bytepause( 1 )
8   box.text{ caption="Pause 1-2", delay }
9 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

```
1 function out()
2   — access the current telegram
3   local tg = telegrams.this()
4
5   if tg:dir() == 1 then
6     — do something with data form port A
7   else
8     — telegram received at port B
9   end
10 end
```

13.8. PROTOCOLVIEW SPECIFIC LUA EXTENSIONS

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

```
1 function out()
2   — access the current telegram
3   local tg = telegrams.this()
4   — show the complete telegram content as hex dump
5   box.text{ caption="Data (hex)", text=tg:dump{} }
6   — shows the last two bytes as hex without separator and 2 digit
7   box.text{ caption="EOS", text=tg:dump{ first=-2, width=2, sep='' }
8   — shows the second byte as a decimal value
9   box.text{ caption="Second", text=tg:dump{ first=2, last=2, base=10
   }
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

KAPITEL 13. THE PROTOCOL VIEW

```
1 function out()
2   — access the current telegram
3   local tg = telegrams.this()
4   — display the duration of the telegram
5   box.text{ caption="Length (s)", text=tg:duration() }
6 end
```

telegram:geterror

Returns a value unequal to zero if the byte at the indexed position was marked with an error. Possible results are:

0: No error, 1: Frame error, 2: parity error, 4: break

telegram:geterror(*INDEX*)

- **INDEX** index of the requested byte.

Example

```
1 function out()
2   — access the current telegram
3   local tg = telegrams.this()
4   local err = tg:geterror(1)
5   if err then
6     local t = {
7       [1] = "Frame",
8       [2] = "Parity",
9       [4] = "Break"
10    }
11    box.text{ caption="Error", text = t[err] or "UNKNOWN: "..err }
12  end
13 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

```
1 function out()
2   — access the current telegram
3   local tg = telegrams.this()
4   — distinguish between a normal null byte and a break
5   if tg:data( 1 ) == 0 then
6     if tg:isbreak( 1 ) then
7       box.text{ caption="BREAK", text=tg:data( 1 ) }
8     else
9       box.text{ caption="NULL", text=tg:data( 1 ) }
```

13.8. PROTOCOLVIEW SPECIFIC LUA EXTENSIONS

```
10     end
11   end
12 end
```

telegram:number

Queries the number of the telegram. The telegram numbers are counted from 1 (the very first received telegram).

telegram:number()

Example

```
1 function out()
2   — access the current telegram
3   local tg = telegrams.this()
4   — show the current telegram number
5   box.text{ caption="Number", text=tg:number() }
6 end
```

telegram:size

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

```
1 function out()
2   — access the current telegram
3   local tg = telegrams.this()
4   — show the size of a telegram
5   box.text{ caption="Length", text=tg:size() }
6 end
```

telegram:string

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 discarded.

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

KAPITEL 13. THE PROTOCOL VIEW

```
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
```

13.8.8 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 before⁹.

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 13.8.7). This type acts as an interface and provides the same

⁹Up to now the `out()` function could only handle the current and previous telegrams by using the modules `tg` and `tgprev`.

13.8. PROTOCOLVIEW SPECIFIC LUA EXTENSIONS

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
<code>at</code>	returns the telegram at the given index/position.
<code>this</code>	returns the current telegram handled by the <code>out</code> function. It is an alias for <code>telegrams.at(-1)</code> .
<code>prev</code>	returns the previous telegram handled by the <code>out</code> function. The same like <code>telegrams.at(-2)</code> .

`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:

KAPITEL 13. THE PROTOCOL VIEW

```
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
```

13.9 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 ProtocolView...` to open the settings dialog.

13.9.1 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.

13.9.2 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.



Telegram font

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.

13.9.3 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.

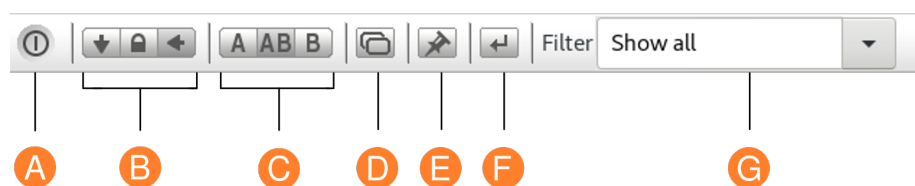
**Background color****13.9.4 Lua compatibility**

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 13.12.2 for a detail overview about the now obsolete functions and modules.

**Lua compatibility****13.10 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 ProtocolView.



- 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 ProtocolView 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 new window with the same sector and settings.

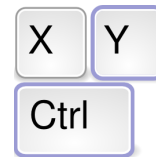
KAPITEL 13. THE PROTOCOL VIEW

- E Pin settings:** Applies (pins) the current window settings as default setup when open again.
 - F Goto Dialog:** Opens the goto telegram number dialog.
 - G Filter control:** Select and pass any filter parameter to the current template script.
-

13.12. ALTERATIONS TO FORMER VERSIONS

13.11 Short commands

Action	Short command
Online help for the Protocol view	F1
Show telegrams in a new window	Ctrl + N
Select all telegrams lines	Ctrl + A
Reverse selection	Shift + Ctrl + A
Export selected lines	Ctrl + E
Open goto telegram number dialog	Ctrl + G
Increase the current telegram font (zoom in)	Ctrl + <input data-bbox="922 757 959 786" type="text" value="+"/>
Decrease the current telegram font (zoom out)	Ctrl + <input data-bbox="922 801 959 831" type="text" value="-"/>
Switch back to the default telegram font size	Ctrl + 0
Open the debug output window	Ctrl + Alt + O
Close window	Ctrl + Q



Short commands
for the most important
functions

13.12 Alterations to former versions

We always try to keep the Lua template extensions as compatible as best with existing versions. Nevertheless sometimes an old design just doesn't fit anymore or only can be supported with great effort. This section list all changes and how you can adapt them to your own code if necessary.

13.12.1 Incompatible changes

With version 5.0.0 the following module names have been changed (mainly for a better understanding or to correspondent with the Lua 5.3 documentation).

- bit → bit32
- cfg → config
- protocol → transmission

You can either replace the occurrence of every call in your script with the new name (which is simple by using the find/replace feature of the editor). Or you assign the new module name to the old at top of your script(s) with:

```
1 bit = bit32
2 cfg = config
3 protocol = transmission
```

13.12.2 Obsolete functions and modules

The following functions are marked as 'obsolete' long ago. They are still there but will be removed finally in one of our the next releases. This section will be a guidance how to update your templates by replacing obsolete code with the more powerful new functions and modules¹⁰.

In the beginning the list of obsolete modules:

¹⁰The protocol templates and examples are already adapted and may serve as a first instance.

KAPITEL 13. THE PROTOCOL VIEW

- `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.

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

13.12. ALTERATIONS TO FORMER VERSIONS

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 `string.unpack`¹¹ 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 adr,fnc,reg,val,lrc,pos = string.unpack( "bb>H>Hb", bindata )
```

`string.unpack` is instructed to 'unpack' the given sequence or string according to the passed format string `"bb>H>Hb"`. The translated meaning is:

- 1 return the first character as byte (b) (`adr`)
- 2 return the second character as byte (b) (`fnc`)
- 3 return the 3th and 4th byte as an unsigned 16 bit value (H) with most significant byte first (>) (`reg`)
- 4 return the 5th and 6th byte as an unsigned 16 bit value (H) with most significant byte first (>) (`val`)
- 5 return the 7th byte as byte (b) (`lrc`)
- 6 return ALWAYS the end of the parsing (`pos`)

At least six results are returned. The last one is always the position where the unpacking stopped. This is equate to the position where you perhaps want to continue with another `unpack` 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 adr,fnc,reg,val,lrc,pos = string.unpack( "bb>H>Hb", bindata )
4 box.text{ caption="Addr", text=adr }
5 box.text{ caption="Func", text=fnc }
6 box.text{ caption="Register", text=reg }
7 box.text{ caption="Value", text=val }
8 box.text{ caption="LRC", text=string.format("%02X",lrc) }
```

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.

¹¹Please note! `string.unpack` replaced the former `bunpack`

KAPITEL 13. THE PROTOCOL VIEW

```
1 local tele = telegrams.this()
2 local bindata = base16.decode(tele:string():sub(2,-3))
3 local adr,fnc,reg,val,lrc,pos = string.unpack( "bb>H>Hb", bindata )
4 box.text| caption)"Start", text=string.char( tele:data(1) )
5 box.text{ caption="Addr", text=adr }
6 box.text{ caption="Func", text=fnc }
7 box.text{ caption="Register", text=reg }
8 box.text{ caption="Value", text=val }
9 box.text{ caption="LRC", text=string.format("%02X",lrc) }
10 box.text{ caption="End", text=tele:dump{ first=-2, width=2 } }
```

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 part of the record and examine it 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.

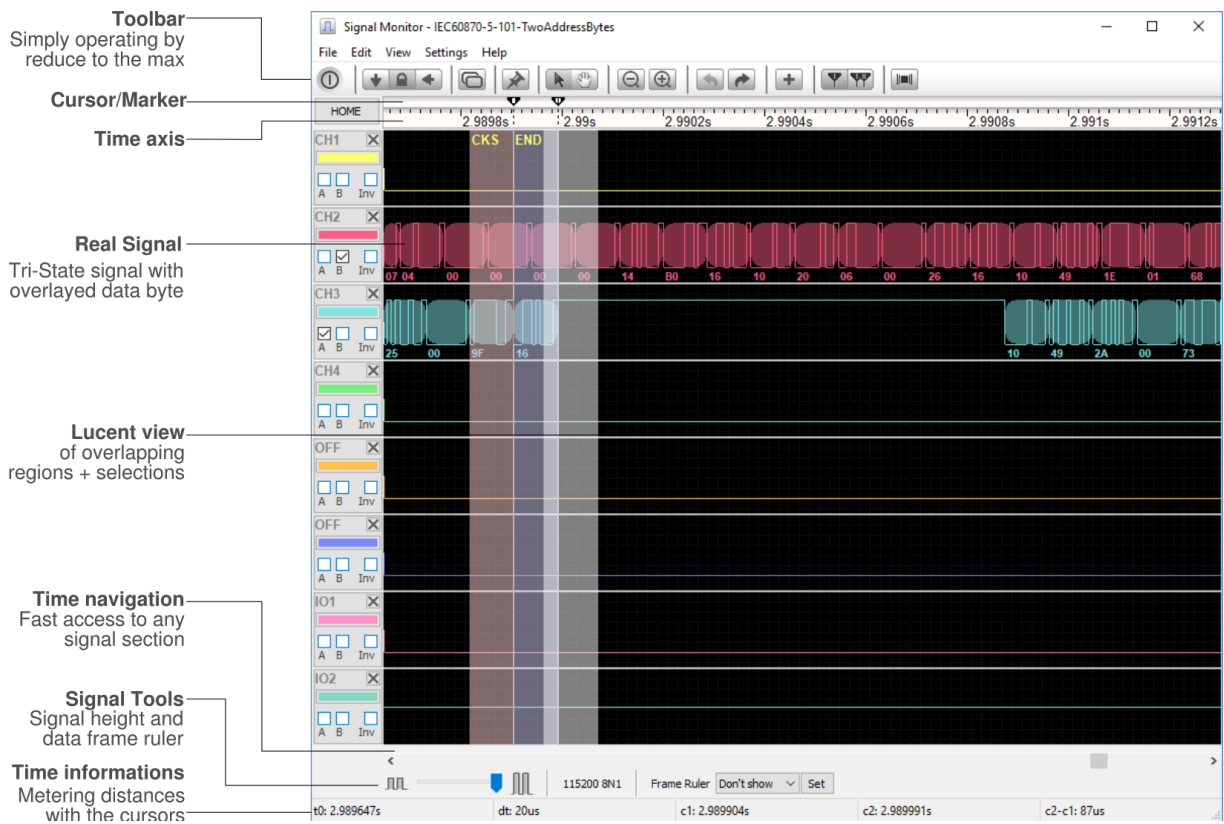
By wrong settings or a malfunctioning hardware invalid data can be set onto the bus.

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.

KAPITEL 14. THE SIGNAL VIEW



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 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 want.

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

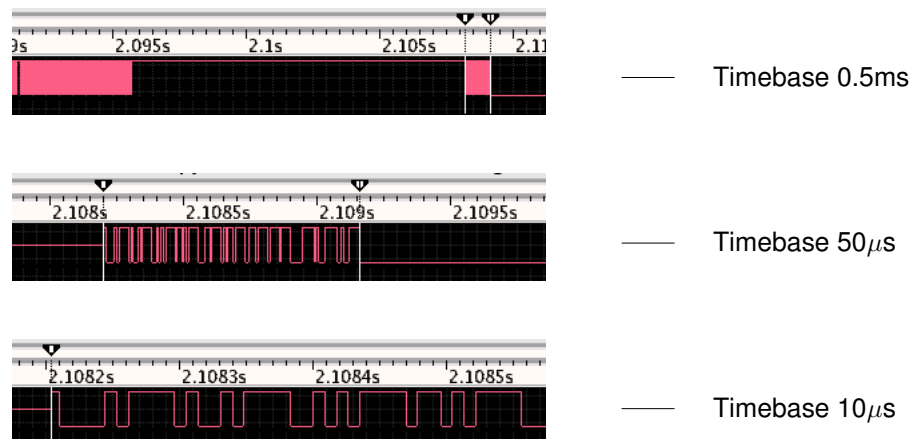
of each other. Unimportant signals can be faded out. All signal specific settings (including the signal order) are directly accessible via the according signal control field on the left side, see section 14.5.

Individual signal settings

Signal sequence, visibility, inverting, data overlay and colour are settable in the according signal control field!

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 click the **HOME** button in the top left corner or press Ctrl+Pos1. The **Timebase** is automatically chosen 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:



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.

KAPITEL 14. THE SIGNAL VIEW

Below the scroll bar you see a slider, with which you can vary the signal height 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 height.

14.2.1 Navigation and zooming by mouse wheel

You can zoom the signal (changing the time base) around the mouse position by turning the mouse wheel with a pressed Ctrl key.

Turning the wheel without a pressed key shifts the signal one grid to the left or right - depending on the turning direction. Pressing the Shift key while turning the wheel moves the signal in 10 grid steps.

14.2.2 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 gripping hand.

14.3 The time base

The time base corresponds to the magnifying for the represented signal. The smallest time base is $10\mu\text{s}$, that means 10 microsecs per raster and means $1\mu\text{s}$ 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 millisecond or second range you will choose 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 by 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. SIGNAL CONTROL FIELD


14.5 Signal control field

Every displayed signal comes with its own control field on the left side. With version 5.0 this field replaces the former signal settings and gives you a fast and easy access to all signal relevant properties. Here you can remove a signal (you can add it again in the settings dialog), chose a different colour, add a data overlay and invert the signal. All this just by a simple click.

You can even rearrange the top-down signal order by drag and drop a certain control field above or below the adjoining fields.

If the height of the signal channel is too less to show all signal settings properly (in case you reduced the window height), a click on the signal color button opens an according little dialog where you can do all the settings which are otherwise provided by the bigger control field. The dialog is shown on the right margin side.

14.5.1 Remove or hide a signal

You can remove a signal from displaying - for instance if the sampling of this signal was disabled during the record or if you just not interested in the signal - simply by clicking the close button  in the according control field. The record is not influenced and you can retrieve the signal display again in the settings dialog, see 14.6.

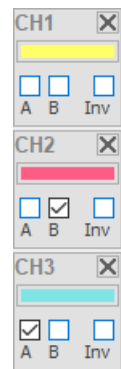
14.5.2 Signal colour

The SignalView defines eight different colours for every signal which fit best with the used colour theme. But you can chose your own colours at any time by left-click the colour bar in the control field.

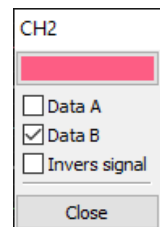
The opening dialog presents the familiar OS colour dialog where you can select a predefined colour or create a new one.

14.5.3 Data overlay

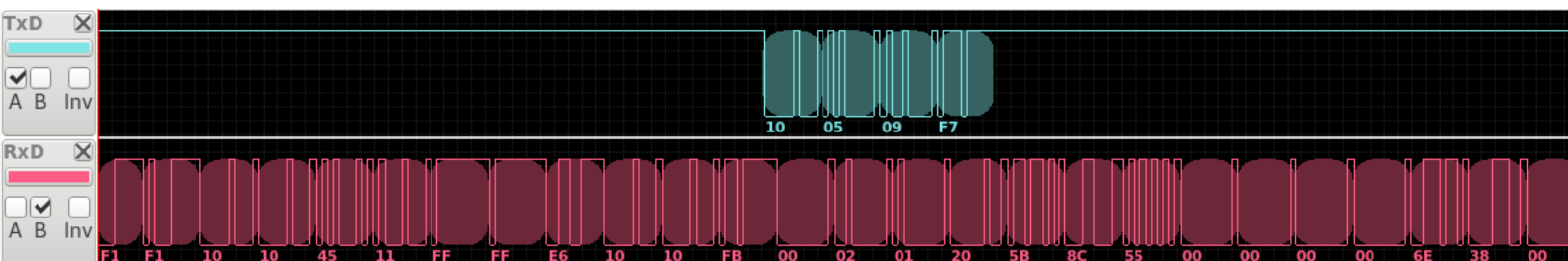
You can overlay the signal with the data from channel A or B. The SignalView calculates the frame of the data (the time between the start and stop bit) and shows every data frame including its hex value in a translucent rounded box.



Signal control field



Signal control dialog



The rounded box corners serve as an orientation where the data frame from the analyzers internal timing measurement begins and ends. If it is different to the tri-state signal lines it indicates a problem in your (application) transmission setup.

Data overlay

KAPITEL 14. THE SIGNAL VIEW

14.5.4 Invert signal

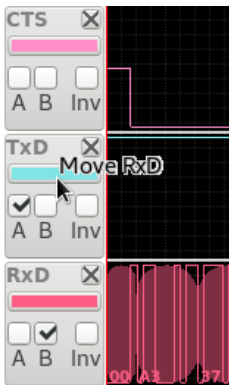
You can invert every shown signal individually by activate the **Inv** checkbox in the control field. This only reverse the signal display and does not affect the recorded data. But it is nevertheless sometimes useful if you are unsure about your bus connection and want to check it for instance with the integrated transmission frame ruler, see 14.8.

14.5.5 Rearrange signal order

Rearranging the signals to your preferred order is easy and intuitive once you have seen (or read) it. Just left-click and hold the name in the control field you want to move. Then drag the name to one of the control fields above or below and release the mouse button.

Dragging a signal control field to a upper field puts the dragged signal above that signal. If you release the mouse on a field below, the signal is put directly under this.

The name of the dragged signal is attached to the mouse pointer as long as you keep the left mouse button down.



Rearrange signals
via drag and drop

14.6 Settings dialog

You will most probably use the settings dialog to enable a hidden signals again. But the dialog also serves for some other common settings which affects all signals. For instance the general colour scheme and several transparency effects. Both setting groups are accessible in two separate tabs.

Please note! There is no 'Apply' button. All you changes have a direct effect to the signal display.

14.6.1 Common settings

The common settings cover - first at all - the on/off switches for all signals. Here you can enable a former maybe unintentional removed signal again.

The remaining controls of the dialog are used to adapt the colour settings common to all signals. These are the background colour, the grid colour and also if the grid is visible or not.

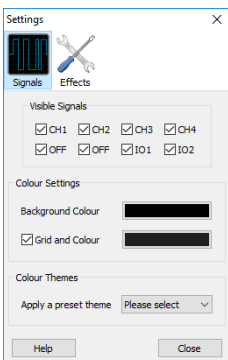
And we have the colour themes selector!

The SignalView offers two colour themes. A 'Classic theme' which dyes background and grid similar to the older software releases.

And the new 'Dark theme' which gives the SignalView a modern digital scope like appearance with best fitting signal colours and discreet grid. But at least it is your choice.

If you are not happy with the themes, you can use an individual background and grid colour and adapt the signal colours as you like.

Future versions will come with more themes and - perhaps - let you create your own themes.



Common Settings
and colour themes

14.6.2 Graphical effects

The SignalView uses transparency effects for a better user experience. But the grade of transparency is as usual of individual taste. Thus this second setting tab. There are three different transparency parameter:

- **Selection:** Specifies the transparency of a mouse selection. A low transparency makes the selection more brighter but the signal behind less discernible.

14.7. CURSOR OPERATING

- **Regions:** The translucency of the regions. The same applies here. A medium transparency is a good adjustment.
- **Data Overlay:** Let you fade in and out the data overlay. A low transparency conceals the underlying signal, a high value fades out the data overlay at all.

14.7 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 position!

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 possible, 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.

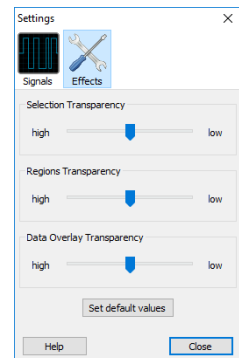
As long as this key is activated both cursors are moved simultaneously, all the same which one you move.

14.7.1 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 between 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.

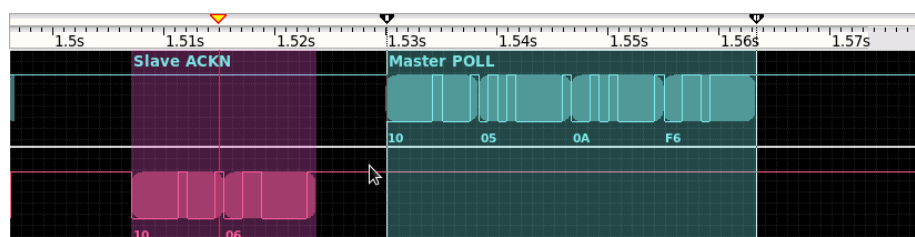


Graphical Effects for translucency

KAPITEL 14. THE SIGNAL VIEW

14.7.2 Regions

The SignalView displays regions as transparent overlays in different colours. The following image shows two regions, where the right 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. It indicates an synchronizing with the two byte transmission in the left region.

Every region may have its own name. Here the left one is states as 'Slave ACKN' and the right as 'Master POLL'. The example is picked from a DF1 transmission. You can find more information at [P.175](#).

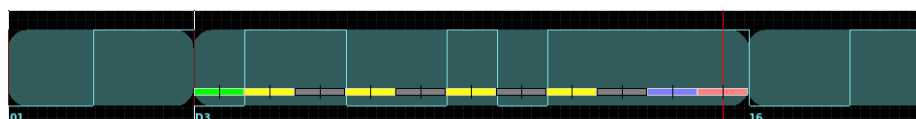
14.8 Measure data frames with the frame ruler

When examining asynchronous transmissions, you sometimes need a closer look into single data frames. For instance if you want to check the transmitted parity bit, the correctness of the data frame between start and stop bit or if you suspect a diverging clock by the sender.

In asynchronous transmissions the bus participants synchronizing their internal data clock for the data sampling with the falling edge of the start bit. The data clock divergence between sender and recipient must not exceed 2% for both otherwise you will see unexpected bytes. Depending on the parity settings such an error must not automatically lead to a parity or frame error and the reason is difficult to find without an examination of the data frame itself.

To make life easier, the SignalView comes with an integrated and so called data frame ruler, see picture below. You can imagine this ruler like a measuring tape with a start mark, marks for every data bit (as adjusted), a mark for the parity (if configured in the transmission) and at least a mark for the stop bit.

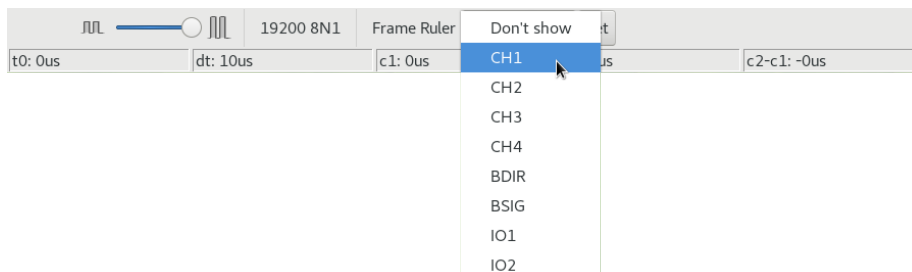
By default the ruler takes the record settings for it's appearance.



The example picture shows a ruler for a data frame format of 8E1 which means: 1 start bit (green), 8 data bits (alternating yellow and dark grey), an even parity (blue) and 1 stop bit (red). The ruler also indicates the sample position for every bit as a vertical line in the center of every field.

14.8. MEASURE DATA FRAMES WITH THE FRAME RULER

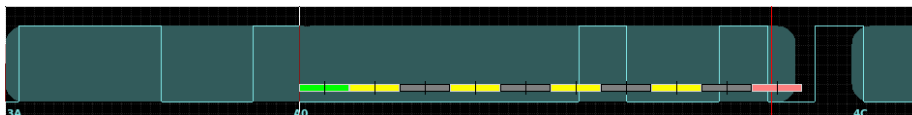
You can assign the ruler to a signal by selecting the appropriate signal name in the frame ruler selector (below the horizontal scroll bar).



Afterwards you can move the ruler to the falling edge of a data frame by dragging cursor 1 as described earlier. The falling edge is indicated by the overlaid data value (here D3) or the beginning of the displayed data frame (the dark cyan area in the picture).

The lowest bit starts immediately after the green start bit. Here we see a bit sequence of 11001011 which is hex D3 (low bit first). The number of high bits is 5 and odd, thus the parity bit is 1 to get an even number of high bits as shown. The stop bit must always 1 which is also correct. Even the clock wide is precisely maintained by the sender.

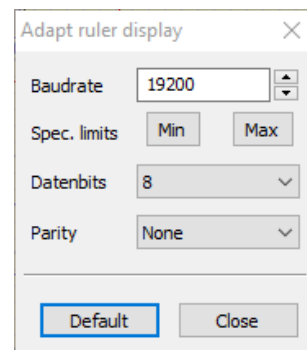
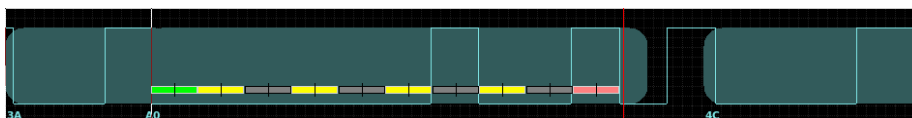
In case of a different data clock, every bit in a frame (after the start bit) will be start slightly earlier or later which - depending of the difference - added up to an intolerable shift. The recipient will assign the signal level to the wrong bit and the resulting data will be calculated wrong. In the picture below a transmitted byte expected as hex 20 becomes hex A0 and the stop bit is false.



14.8.1 Adjust the data frame ruler

You can adapt the frame ruler to other data frame parameters or a different baud rate in the frame ruler dialog. The dialog appears when you click the **Set** button on the right of the frame ruler signal selector.

The ruler dialog (see picture in the margin) let you choose a different baud rate or set the ruler clock/bit width to the tolerable minimum or maximum rate. You can also select a different number of data bits or parity. The made adjustments are applied immediately to the ruler in the signal. With it you can easily determine if a certain byte frame is within the specifications or not. And what bit rate is really used in the data frame without measuring single bit widths. In our example we find a fitting baud rate at about 20600.



Ruler dialog
adjust the frame ruler

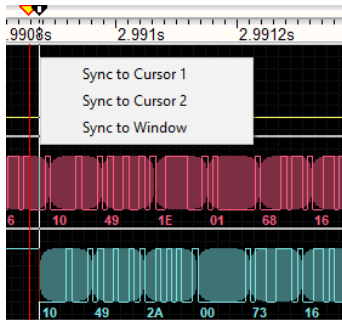
14.9 Synchronizing

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.



Synchronizing
other views with cursor

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

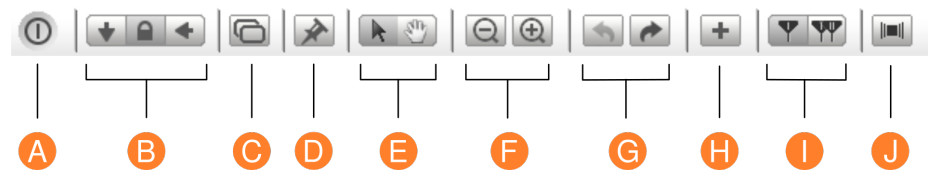
As the synchronizing event the current signal sector is used. That means the first level change seen from the left border.

Why is the first event after the cursor position used and not the cursor position itself?

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.

14.10 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.

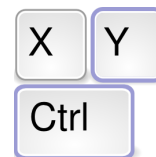
C New view: Opens (clones) a new window with the same signal section and settings.

14.11. SHORT KEYS

- D Pin settings:** Applies (pins) the current window settings as default setup when open again.
- E Mouse control:** Optionally the mouse can be used to zoom the selection or to move the signal (hand symbol).
- F Signal zooming:** Magnifies or demagnifies the visible section in 1, 2, 5 multiplicative factors by choosing the next lower or higher time basis.
- G Undo/Redo:** Undoes the last change of the visible section or restores it respectively.
- H Add region:** Saves the range between both cursers as a new region.
- I Interlock Cursor:** The cursors can be selectively moved singly or together (combined).
- J Show region dialog:** Opens the region dialog, e.g. to fade in or off regions, to delete them or to name them.

14.11 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 grid step towards signal end	Right arrow
Move view 1 grid step towards signal start	Left arrow
Move view 10 grid steps towards signal end	Shift + Right arrow
Move view 10 grid steps towards signal start	Shift + Left arrow
Move signal horizontal 1 grid step	Mouse wheel
Move signal horizontal 10 grid steps	Shift + Mouse wheel
Zoom in signal	Ctrl + <input data-bbox="938 1563 975 1592" type="text" value="+"/>
Zoom out signal	Ctrl + <input data-bbox="938 1615 975 1644" type="text" value="-"/>
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



Short commands
for the most important
functions

KAPITEL 14. THE SIGNAL VIEW

Close signal view

Ctrl + Q

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 SignalView 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 a range between both cursors, in the data monitor as a certain data sequence or as the occurrence of single characters and in the protocol monitor simply by adding a single telegram or a selection of telegrams as a new region.

Regions are acting like bookmarks of interested sections in your record. As soon as a region is listed in the region dialog, you can jump to the start or end of that region just by clicking the start or end property in the region.

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).

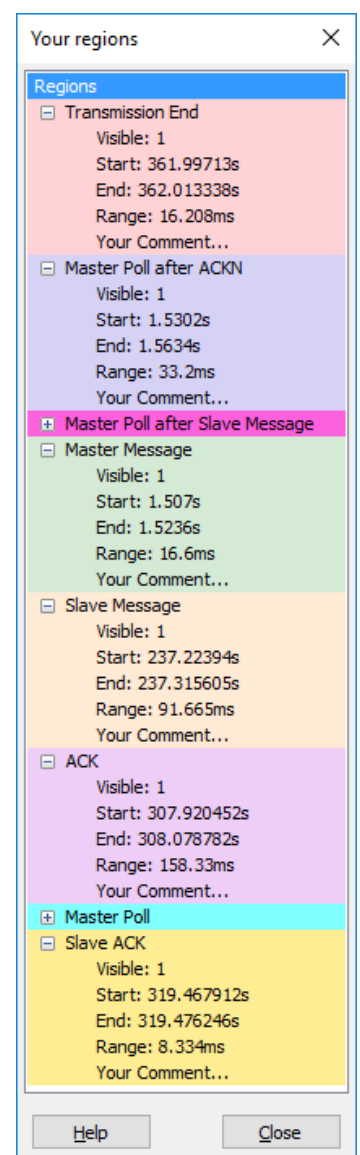
The MSB software allows the definition of 8 regions. Every region can be individually named and optionally switched on and off. With exception of the line state monitor you open the region dialog with View→Region dialog in every analysis window. An always opened region window will be put in front of all windows automatically.

The right picture shows a region dialog with all together 8 regions where the regions 3 (magenta) and 7 (cyan) are folded in. All regions are visible (the `Visible` property is set to 1).

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 double click the according `Visible` property.

A shown region is indicated by a value of 1 whereas a hidden region has a 0 value.



Regions Dialog

KAPITEL 15. REGIONS

15.2 Remove a region

To delete a region from the region dialog, just click/select the region name and press the **Del** button on your keyboard. You must confirm the deletion before the region is finally removed from the list.

To delete a region does not alter the record. It only removes the region information. It is like to remove a bookmark from a book. This too does not remove any pages.

15.3 Rename a region

Regions are by default named in the order sequence of their creation as Region1 to Region8. But you can choose a more self-explanatory name by double click the name and input a text of your own.

A Region name can contain any character but they are limited to a maximum length of 200 characters. Of course labels of such a length don't make sense. A better place for additional information or remarks for a region is the comment property¹.

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 analysis 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 activated 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 enabled the synchronization by other views 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. The display mode of the according program window or view must be set to synchronous mode!

15.5 Region storage

Regions are part of a record (`msblog` file) and will be automatically stored when you are saving a record or project. The software will warn you about unsaved regions on program exit or when you start/load a new record to avoid data losses.

¹Remarks will be supported in one of the future versions

15.6 Region properties

Below is a summarized list of the several region properties. All properties are clickable except for the range property which has only an informal usage.

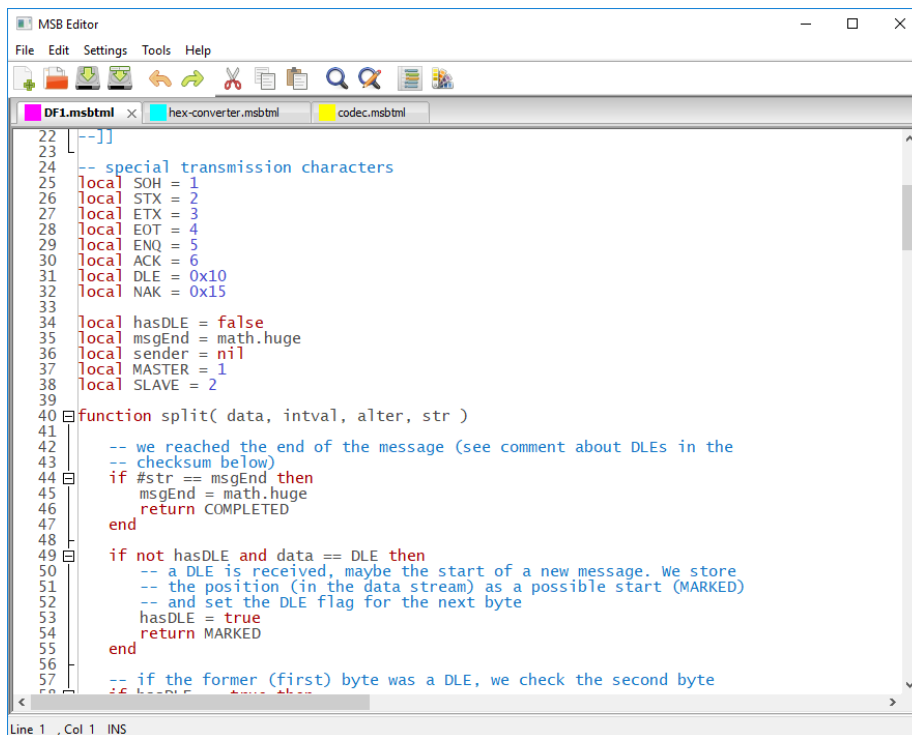
Property	Meaning
Region name	Editable region name
Visible	0: Region is hidden in views, 1: Region is displayed by views.
Start	The region start relative to the record start in seconds. A click causes synchronized views to display the beginning of that region.
End	The region end relative to the record start in seconds. A click causes synchronized views to display the end of that region.
Range	The region range in seconds (not clickable).
Comment	region description, not yet supported.

16

The Editor

Starting with version 5.0.0 Lua becomes a more and more important part of the whole Serial Analyzer software. The benefits of a very adaptable script language for protocol templates has been fully transferred to the DataView, allowing the processing of the raw data by Lua already accustomed in the ProtocolView.

An external, full equipped editor was the logical consequence.



```
MSB Editor
File Edit Settings Tools Help
DF1.msbtml hex-converter.msbtml codec.msbtml
22 [--]]
23
24 -- special transmission characters
25 local SOH = 1
26 local STX = 2
27 local ETX = 3
28 local EOT = 4
29 local ENQ = 5
30 local ACK = 6
31 local DLE = 0x10
32 local NAK = 0x15
33
34 local hasDLE = false
35 local msgEnd = math.huge
36 local sender = nil
37 local MASTER = 1
38 local SLAVE = 2
39
40 function split( data, intval, alter, str )
41
42 -- we reached the end of the message (see comment about DLEs in the
43 -- checksum below)
44 if #str == msgEnd then
45     msgEnd = math.huge
46     return COMPLETED
47 end
48
49 if not hasDLE and data == DLE then
50     -- a DLE is received, maybe the start of a new message. We store
51     -- the position (in the data stream) as a possible start (MARKED)
52     -- and set the DLE flag for the next byte
53     hasDLE = true
54     return MARKED
55 end
56
57 -- if the former (first) byte was a DLE, we check the second byte
58 if hasDLE and data == DLE then
```

The editor integrated into the Serial Analyzer software is not only especially designed for writing Lua code, it also features all kind of qualities you expect from a good editor like code folding, syntax highlighting, multi-doc interface, unlimited undo/redo and more.

KAPITEL 16. THE EDITOR

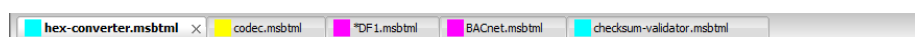
To make scripting as easy as possible the editor further scores with automatic code frameworks depending whether you write a protocol template, a script for the DataView or a module used by both of them.

16.1 Open the editor

The editor is invoked when pressing the **New/Edit** in the DataView or ProtocolView. It pops up either with the currently selected dialog script or shows the script in an additional document tab.

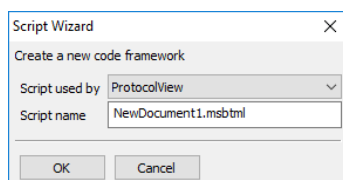
The latter let you open several scripts at the same time, e.g. to compare parts of different scripts or copy and paste certain code sections between them.

To distinguish between scripts for the Data-, ProtocolView or a module, the editor script tabs are showing different colours. This helps also to differ scripts with the same name but used in different kind of views.



DataView scripts are displayed with a cyan box, protocol templates with a magenta box and a yellow box indicates a module script file.

Script files with unsaved modifications are marked with a little '*' in the tab. You can close a file by click on the **X** in its tab. If the file is modified, you will get a warning. The editor (or the MSB-Analyzer program) will never ends without informing you about open changes and asking you how you will proceed.



Script Wizard

16.2 Start with a new script

In contrary to former versions, new scripts are now solely created in the editor itself. This avoids doubled edited scripts for instance in two concurrently running ProtocolViews.

In case of a new script the editor asks you what kind of script you want to edit and provides you with an according code framework.

The new script then is shared with all Views which are using the same template or DataView watch script.

Select a new created script

Please note! A new created script file is not selectable by a view until you have saved it in the editor.

16.3 Interactive coding

As an firmly integrated part of the MSB-Analyzer program the editor is intended to interact with the linked views automatically. In particular to trigger the updating or redrawing of a telegram display in the ProtocolView or the watch dialog in the DataView when saving the according Lua script. As simple as it is, it makes the coding of protocol templates or the processing of data in the DataView an amazing experience.

As soon as you save an edited script in the editor which is also selected in a

16.4. HIGHLIGHT INDIVIDUAL KEYWORDS

DataView or ProtocolView the views are updating automatically their window content - applying you last script modifications. Just click the save button in the toolbar or press CTRL+S and - voila - the views refresh the display.

16.3.1 Lua script errors

We need to differentiate between common Lua errors like wrong keywords (for instance input `functns` instead of `functions`), module errors (calling a module function not available for the given View) and run-time errors. The latter only occurs during execution time like dividing by zero or accessing a nil (invalid) variable content.

The editor itself does not check or evaluate the edited scripts. This is in the responsibility of the View which executes the script. Therefore errors are not shown in the editor but in the view which display an error with detail information about the cause and script line.

16.4 Highlight individual keywords

Lua scripts, especially protocol templates could become quite extensive and the programmer will be thankful for every help to make code reading easier.

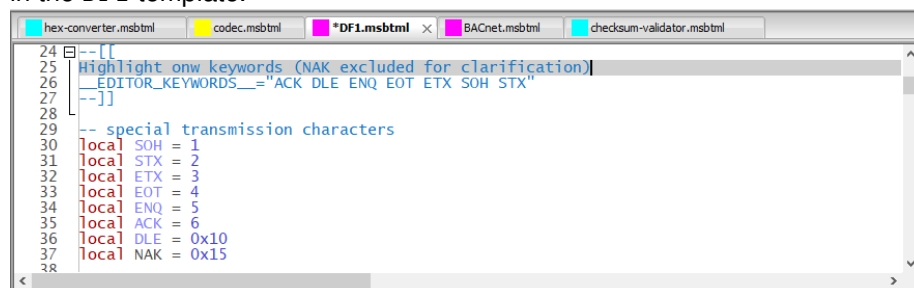
One proven method is the syntax highlighting. The editor extends its ability of highlighting Lua keywords by let the user assign individual words as a space separated string to the variable `__EDITOR_KEYWORDS__`. The new keywords then are displayed too in a different colour.

You can also cover this assignment in a comment because the editor only parse for a: `__EDITOR_KEYWORDS__="keyword1 keyword2 ..."` pattern which we strictly recommend to avoid side effects.

The following picture shows the result of

```
__EDITOR_KEYWORDS__="ACK DLE ENQ EOT ETX NAK SOH STX"
```

in the DF1 template.



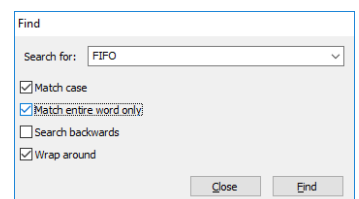
```
24 --[[
25 Highlight onw keywords (NAK excluded for clarification)
26 __EDITOR_KEYWORDS__="ACK DLE ENQ EOT ETX SOH STX"
27 --]]
28
29 -- special transmission characters
30 local SOH = 1
31 local STX = 2
32 local ETX = 3
33 local EOT = 4
34 local ENQ = 5
35 local ACK = 6
36 local DLE = 0x10
37 local NAK = 0x15
38
```

16.5 Find

Looking for a certain piece of code or text often depends on other facts. You want to find only matches for entire words or that the results are equal in upper and lower case letters. You like to search backwards and wrap around.

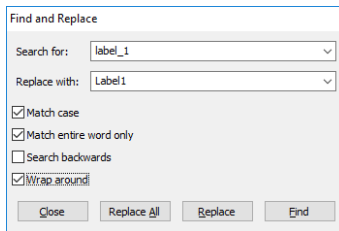
The MSB-Analyzer editor provides you with a simple but nevertheless powerful search functionality. Just click the find icon in the toolbar or press CTRL+F to start the find dialog.

The dialog keeps your settings during a session.



Find dialog

16.6 Find and replace



Find & replace dialog


It's the usual business of a programmer to rename a certain variable after rethinking the meaning of it. The MSB-Analyzer editor offers you a powerful find and replace dialog which let you replace a given text step by step as well as in one go. A step by step approach is especially helpful if you like to check the replacement first before you want to apply it. Here you can jump from text passage to text passage and switch the text by your choice.

The dialog supports all settings of the find mechanism and remembers all your inputs during the program session. This makes it easy to repeat a former replacement later.

You can open the find and replace dialog by clicking the according toolbar icon or press CTRL+H.

16.7 Code folding

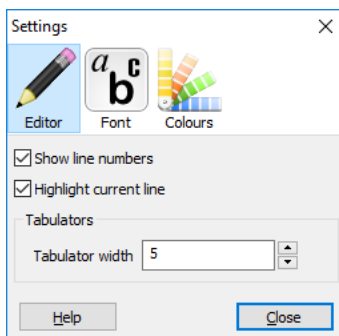
Code folding is a nice feature when your script consists of a lot of functions or other code blocks like tables. Activated in the toolbar it collapse every function into their very first code line. In case of a function, it's the function definition or name. Tables collapse into the first line of the table code.

Every folded code block is headed by a  on the left editor margin. You can fold or unfold only certain functions/blocks or apply the folding to every block in your script by clicking the icon in the toolbar.

Find and replace with folded code

Folded code is not 'visible' for the find dialog, but replacing ALL occurrences of a given text with another one affects also collapse code lines!

16.8 Editor settings



Settings dialog

The editor is adaptable to your own preferences via the settings dialog. You can change the behaviour of the editor (show/hide line numbers, preset the tabular size, highlight the current line), change the font (default is the system fixed width font) and the colour scheme.

The editor comes with two colour themes. A classic theme with a white background and a dark theme for users preferring an editor with a light typeface on a dark background.

All changed settings are applied automatically and saved for the current session when closing the settings dialog.

16.9 Colour wizard

Especially the ProtocolView uses different colours to illustrate telegrams. But also the DataView allows colours to mark certain data sequences. Colours are specified by a RGB (red, green, blue) value. In Lua as an integer value like: `0xRRGGBB` where RR, GG and BB representing the red, green and blue part as a hex value of 00...FF. A pure red colour for instance is defined as `0xFF0000`. Choosing the right colour only by vary the number is boring. The editor therefore provides a colour wizard (or colour chooser) which let you select the colour

16.10. SCRIPT FILES LOCATION

and insert the right Lua value at the cursor position afterwards.
You can open the colour wizard in the toolbar or by pressing CTRL+Alt+C.

16.10 Script files location

All script files are located in the users application data folder by default. Under Linux this is:

```
~/IFTTOOLS/SerialAnalyzer/7.0.2/Templates/
```

Under Windows:

```
C:\Users\USERNAME\AppData\Roaming\IFTTOOLS\SerialAnalyzer\7.0.2\Templates\
```

The Templates are organized in different directories according to their view belonging. So there is a ProtocolView, a DataView, a Trigger and a Module directory. You will also see a SwitchEditor folder, but this one only contains drawings and no scripts.

You can save a script on a different location, for instance if you want to share it with other peoples or use it in another application. The best way is to use 'Save as copy' in the file menu. But note! Views can only 'process' a script in their default location because it is the only place where they looking for new or modified scripts.

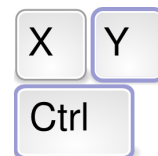
Script location

The DataView and ProtocolView require the scripts in their default location otherwise they wont see new files or react to modified scripts!

16.11 Editor short keys



The usage of the editor is as simple as possible. All editor functions are accessible from the toolbar or via a right mouse click (selection, copy, paste, ...). A few short keys are nevertheless worth to remember, since it spares you some additional mouse clicks.

Action	Short key
Copy the selected text into the clipboard	Ctrl + C
Opens the find dialog	Ctrl + F
Opens the find and replace dialog	Ctrl + H
Toggle the folding of all code blocks	ALT + T
Create a new script/document	Ctrl + N
Load a script file into the editor	Ctrl + O
Save the current script/document and trigger a view update	Ctrl + S



Short keys
of the most important
functions

KAPITEL 16. THE EDITOR

Save the current script/document under a new name	Shift + Ctrl + S
Paste the text in the clipboard at the current cursor position	Ctrl + V
Cut the selected text and copy it into the clipboard	Ctrl + X
Redo last undo	Ctrl + Y
Undo last modification	Ctrl + Z
Increase the current editor font (zoom in)	Ctrl + 
Decrease the current editor font (zoom out)	Ctrl + 
Increase or decrease the editor font via the mousewheel	Ctrl+Wheel

17

An introduction to Lua

Lua is one of the fastest scripting languages in the world. Because of its small and simple design it's also easy to learn. Lua contains a few but also very 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.

17.1 Getting started

Lua is a programming language that offers a very impressive set of features while keeping everything fast, small and simple. In the analyser software we are using Lua version 5.3. So lets go to learn a little bit more about this amazing scripting language.

You can test all the following examples by yourself. Just start the analyzer application and open the Lua script editor in the control program by clicking the 'Open Lua Script Editor' in the 'Views' menu. You do not need a connected analyzer device.

The Lua script editor allows you to execute little code pieces just by select the desired line and press **SHIFT** + **F5**. You can even execute a complete script buffer simply with **F5**. Code evaluation works in all opened files. But the editor provides you with a special ***SKETCH*** buffer which let you play with code snippets without modifying existing scripts.

The content of this buffer is automatically saved when closing the buffer or editor and restored when you open the sketch buffer again.

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. So open the sketch buffer in the editor Lua menu and input the following line:

```
1 print("Hello World")
```

Press **F5**. The editor opens the evaluation output window and gives you a friendly `Hello world` greeting.

The Lua `print` function is intended as a quick way to show a value, typically for debugging or testing. It receives any number of arguments separated by a



Lua Version 5.3

KAPITEL 17. AN INTRODUCTION TO LUA

comma and prints their values to the Lua output window whereas every result is separated by a tab character. For example:

```
1 print("Result of 1/3", 1/3 ) → Result of 1/3 0.33333333333
```

Here we pass two arguments to the `print` function. The first is a string enclosed in double quotes, the second is the result of an arithmetic statement. Lua handles all numbers as floating point numbers by default (see page 194). So the result is a real number and not an integer.

The `print` function can also serve as a simple one line calculator. Since Lua allows the input of hexadecimal values it also works as a number converter. Consider the following lines:

```
1 print( 0xFFFF * 2 ) → 131070
2 print( string.format( "%x", 123456 ) ) → 1E240
3 print( string.format( 0xFFFF ~ 0x3000 ) ) → CFFF
```

Lua accepts hexadecimal numbers by adding a leading `0x` to the number as shown in Line 1. The output is always decimal but you can use the string format function to manipulate the output format. This function works as similar as the `printf` in C/C++.

Line 2 uses the unsigned hexadecimal notation `"%x"` to display the output with radix 16.

If you ask yourself: What means the point between `string` and `format`?

Lua provides all generic functions for string manipulations in a separate library or module. To access a library function you must call it with the leading library/module name separated by a dot before the function name.

Line 3 introduces the 'exclusive or' bit operator which in Lua is the tilde symbol. Lua supports all bitwise operators you may know from other languages since version 5.3. For more information about the built-in bitwise operators, please see page 204.

17.1.1 Using functions

`print` is a built-in Lua function (beside many others). You can add your own functions at any time and call them afterwards in the script or sketch buffer. For more details about functions see page 207. Now let us extend our 'hello world' example with a greeting function.

```
1 function greeting( text, name )
2   print( text.." " ..name )
3 end
4
5 greeting( "Hello", "Bob" )
```

Input the function and the calling line 5 in the editor sketch buffer and press **F5** to execute the buffer.

The function `greeting` gets two arguments, the greeting text and a name. In the function body both arguments are put together with the Lua string concatenation operator `..` (see page 205). Here we take the greeting text, add a whitespace and the name which gives us:

```
Hello Bob
```

17.1.2 Function with multiple results

The `greetings` function does not return a result. It just prints out the jointed arguments. But Lua functions can return any number of results, see page 207. Imagine a function which divides two integer and returns both, the quotient and the remainder¹.

```
1 function divide( dividend, divisor )
2   local remainder = dividend % divisor
3   local quotient = ( dividend - remainder ) / divisor
4   return quotient, remainder
5 end
6
7 print( divide( 10, 3 ) ) —> 3 1
```

We first calculate the remainder using Lua's module operator `%` in line 2. Since Lua handles all numbers as floating point numbers we cannot simply divide the dividend by the divisor which will give us 3,3333333333333.

Instead we first subtract the remainder from the dividend in line 3 before the division to get an integer result for the quotient.

The keyword **local** in line 2 and 3 limits the scope of the variables `remainder` and `quotient` to the surrounding code block, here the function `divide`, see page 203.

Line 4 returns both results separated simply by a comma.

17.1.3 Processing and manipulating strings

Lua strings can contain any byte value. The length of a Lua string is not specified by a certain character. You don't have to struggle with null bytes as in C/C++ for instance. And: With Lua you can mix strings and numbers in a very easy and intuitive way. This makes Lua strings the ideal data type to build and represent protocol telegrams.

Take a look at the following string example:

```
print( string.dump( "\000\001\255\128" ) ) —> 00 01 FF 80
```

You can add any byte value to a Lua string by input it's decimal value with a leading backslash. In the example we start with a null byte followed by a binary 1, 255 and 128 value. The `dump` function is an extension to the Lua string library provided by the analyzer software. It outputs the content of a string in hexadecimal notation (amongst others) and is very helpful when validate the result of own created telegram strings.

In our next example we consider a little checksum function.

Most field-bus protocols specify a checksum to validate the correctness of a transmitted byte sequence. A simple checksum algorithm is the modulo 256 sum of a given data sequence². The function (line 1...7) looks like:

```
1 function Checksum( data )
2   local chksum = 0
3   for i=1,#data do
4     chksum = chksum + string.byte( data, i )
5   end
6   return chksum % 256
```

¹This is just an example and works for positive integer arguments only!

²As optionally used in IEC60870-5-103

KAPITEL 17. AN INTRODUCTION TO LUA

```
7 end
8
9 print( Checksum( "hello world" ) ) —> 92
```

Input the lines above in the editor sketch buffer and press **F5**. This calls the checksum function with the data sequence or string "Hello world" in line 9, the result is 92.

How does it work?

In the function body we create a local variable `chksum` and initiate it with 0. In line 3 we iterate over all bytes in the passed data (string) argument.

Please note! In contrary to other programming languages Lua indexes in tables (arrays) and strings starts always with 1.

The `#` character is defined as the length operator. It returns the length of a string or array. For example:

```
print( #"hello world" ) —> 11
```

returns 11.

Line 3 in our example therefore counts `i` from 1 (the first byte in the given data sequence) to the last byte.

You can access the numeric code or value of every byte/character in a string with the string related function `string.byte(i)`³.

In line 4 we add up the numeric values of all bytes in the passed string. An alternative way to call the `sub` function is:

```
chksum = chksum + data:byte( i )
```

Lua uses the colon operator to provide a special syntax for object oriented calls. The colon in the expression `data:byte(i)` tells Lua to call the `byte(i)` method of the string variable or object on the left side of the colon⁴. Therefore it must not be passed as an argument to the `sub` function. If you are familiar with object oriented languages like C++ or JavaScript consider the colon in Lua as the normal dot in these programming languages when accessing an object method.

Line 6 at least returns the lowest 8 bits by returning the modulo 256 of the `chksum` variable.

A checksum is added to the transmitted data sequence usually at the end. If you want to verify the checksum of a given telegram you therefore must calculate the checksum from all (or a part) of the data sequence except for the checksum itself. This means: You must pass a certain section of the data sequence to the checksum function.

Lua provides you with a mighty sub string mechanism which makes it especially easy to extract any part of a string. Because it uses negative indexes for backwards counting you can even query the last 2 bytes without knowing the string length. The following examples will give you an idea how it works.

³This is a simplification. The `byte` method accepts a second index too and returns an array (table) of the specified range. The second index is set to the first by default.

⁴The variable type of the left side of the colon must be a string, otherwise you will get an error.

17.1. GETTING STARTED

```
1 seq = "Hello world!"
2 print( string.sub( seq, 7 ) )    —> world!
3 print( string.sub( seq, 1, 5 ) ) —> Hello
4 print( string.sub( seq, -6 ) )  —> world!
5 print( seq:sub( -6, -2 ) )      —> world
```

The string method (or function) `sub` is called with three arguments:

```
string.sub( STRING, FROM, TO )
```

whereas `TO` is optional and set to the string end by default.

Line 1 returns the sub string from position 7 (the 'w') to the end (by default).

Remember that Lua counts indexes strings starting with 1.

Line 2 returns the first 5 characters as a string (from 1 to 5).

In line 3 we select the sub string by counting from the string end. -1 means the last character, -6 the 'w' again.

At least line 4. Here we use the object oriented syntax and limit the end to the second last character by passing the end position as -2.

The function `sub` is part of the string module. Lua strings can represent all data or telegram content as long as the data width is 8 bit or less.

17.1.4 Data structures in Lua

Protocol specifications often define numeric constants with a certain meaning. These are status or error states or command values like the function numbers in a Modbus transmission. Basically spoken these are key/value pairs. The key is the numeric value of an error, state or command and the value is a string describing its meaning. Imagine a digital input returning 0 or 1 but you want to see an OFF or ON.

To get the according string (the description) you can write a Lua function like this:

```
1 function GetDigitStateText( state )
2   if state == 0 then
3     return "OFF"
4   elseif state == 1 then
5     return "ON"
6   end
7   return ""
8 end
```

This works well for small numbers of states. But consider you have a lot of error codes. Growing `if...elseif` constructs quickly become illegible.

In computing, key/value pairs are stored in so called associated arrays whereas a key works as an index in a list of values. The key may be an unique number (like in normal arrays) but also any string. The latter calculates a unique hash value to access the associated value.

Lua comes with its own kind of associated array called **table**. Tables are the main and only data structuring mechanism in Lua and a very powerful one. You can use tables to build ordinary arrays (even with a counting from zero), stacks, queues, function and symbol tables and more. Lua modules are organized by tables. When we called the `string.dump` function for Lua it means call the function with the index `dump` in the string module table.

KAPITEL 17. AN INTRODUCTION TO LUA

You will find more information about tables on page [197](#). Here we will concentrate on the basics and consider tables as a list of key/value pairs.

You can create a very simple table with:

```
1 days = { "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun" }
2 for i=1,#days do
3     print( days[ i ] )
4 end
```

Lua uses a normal numeric index counting from 1 when not specified otherwise. The table index in the example above starts with 1. In line 2..4 we iterate over all table items (the length operator # returns the number of items in the table) and output the abbreviated week day names.

We can also change the indexing to start from zero by using the associate array feature in Lua tables. Take a look:

```
1 days = {
2     [ 0 ] = "Mon",
3     [ 1 ] = "Tue",
4     [ 2 ] = "Wed",
5     [ 3 ] = "Thu",
6     [ 4 ] = "Fri",
7     [ 5 ] = "Sat",
8     [ 6 ] = "Sun"
9 }
10 for i=0,#days do
11     print( days[ i ] )
12 end
```

Here we add an individual key for every week day. The keys are simply numbers starting with 0. The `for` loop in line 10 begins with 0 instead of 1, the length operator # now returns 6 (the last key in the table).

Please note! If the table has no continuous index values, the result of the # operator is different. We will discuss this special case on page [197](#).

Now back to our digital input state. We create a table and 'associate' every digital state 0 or 1 with the corresponding string 'OFF' or 'ON'.

```
IOState = {
    [ 0 ] = "OFF",
    [ 1 ] = "ON"
}
```

Our `GetDigitStateText` function becomes:

```
1 function GetDigitStateText( state )
2     local tbl = {
3         [ 0 ] = "OFF",
4         [ 1 ] = "ON"
5     }
6     return tbl[ state ]
7 end
```

If you pass an wrong state which doesn't exist as a key in the table, for example 2, the function returns `nil` which is the Lua way to say 'this is an invalid value'. The `print` function ignores a `nil` argument but the Lua interpreter will give you

17.1. GETTING STARTED

an error message as soon as you try to process a **nil** value like in a mathematical function⁵:

```
print( nil ) —> nothing
print( math.sin( nil ) ) —> bad argument #1 to 'sin'
                             —> (number expected, got nil)
```

Returning a nil value when a string result is expected is not a good idea. In most cases the result is processed by other functions or it is part of a string operation. For example: The Lua interpreter stops the execution of the script in the following line with an error *attempt to concatenate a nil value* because you cannot add a nil value to a string.

```
print( "IO1"..GetDigitStateText( 2 ) )
```

But you can easily intercept the returning of an invalid key (or index) in a table because Lua handles a **nil** value also as logical **false**. See here:

```
1 function GetDigitStateText( state )
2   local tbl = {
3     [ 0 ] = "OFF",
4     [ 1 ] = "ON"
5   }
6   return tbl[ state ] or "INVALID STATE"
7 end
```

We simply add a logical **or** condition in line 6. If the table item with the index/key state is valid (exists and not nil), we return the indexed value otherwise we return any fitting (maybe empty) string.

17.1.5 Reuse code with Lua modules

You can copy and paste individual functions wherever you need them. But it makes more sense to group functions serving a similar goal in a module and store them in an independent file.

A checksum module is a good example. The Lua interpreter used by the analyzer software comes with its own native checksum implementation and provides you with the most used checksum algorithms. Nevertheless there may be others you have written by yourself as like our example above.

To start a new Lua module file, click the 'New file' icon in the editor toolbar or press **STRG** + **N**.

The editor asks you what kind of new file you want to create. Choose Script for 'Modules' and replace the default file name with `mychecksums.msbtml`. Click 'OK' and the editor opens a new buffer with an already filled module code framework.

Replace all occurrences of the default module placeholder name `ModuleName` with `MyChecksums`. Add our checksum function from above but rename it to: `MyChecksums.Mod256Sum`. Your module file now should look like:

```
1 local MyChecksums = {}
2
3 function MyChecksums.Mod256Sum( data )
4   local chksum = 0
5   for i=1,#data do
```

⁵Lua groups all mathematical functions in the math module.

KAPITEL 17. AN INTRODUCTION TO LUA

```
6     chksum = chksum + data:byte( i )
7     end
8     return chksum % 256
9 end
10
11 function MyChecksums.version()
12     return "1.0.0"
13 end
14
15 return MyChecksums
```

Lua handles modules via tables (another application for the mighty table concept). In line 1 an empty table is created whereas the name is of no significant importance. Afterwards we add our functions into the table. We do so by add the leading table name to the function name separated with a dot. Lua uses the function name as an index for the later access. In our example there are two functions: `Mod256Sum` and `Version`.

Line 15 at least returns the table with all our functions.

Save the new created module, than switch to the SKETCH buffer again. Input:

```
1 checksums = require "MyChecksums"
2 print( checksums.Mod256Sum( "Hello world" ) )
```

The Lua keyword **require** in line 1 loads our module file and assigns it to the given variable, here `checksums`.

`checksums` is indeed a table as you can see with:

```
print( type( checksums ) ) → table
```

(The Lua `type` function returns the type of its only argument as a string.)

You can add further checksum functions to your checksum module file. Just remember to add the module name (in our example `MyChecksums` with a dot to the function name. The module name followed by the dot makes the function part of the internal module table where each function name is a table entry.

```
1 function MODULENAME.FUNCTIONNAME(...)
2     — function code ...
3 end
```

Functions without the leading module name behave like local variables. They are only accessible from within the module file but not from the outside. You will find more information about Modules on page 209.

Test limitations

The editor built-in Lua interpreter cannot execute Lua modules which depend on a special view environment like the `ProtocolView` box module or others!

17.2 The Lua language

Each programming 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 programming language syntax. The language syntax declares, how a program has to been written correctly.

17.2. THE LUA LANGUAGE

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.

17.2.1 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.

17.2.2 Whitespaces and line ends

Lua ignores any whitespaces (like the space or tab characters) if they aren't part of a string constant (see 17.2.4.6). 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 interpreter 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
```

17.2.3 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 print( 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 print( x ) —> 10
```

KAPITEL 17. AN INTRODUCTION TO LUA

17.2.4 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 like 100, 2.5, 5.2E-03 (integer and floating point numbers, whereat Lua doesn't distinguish between them), strings like "Hello world" and the boolean values **false** and **true**.

17.2.4.1 Numbers

Lua simplifies the usage of different numbers like integer, single float, double float by using only one kind of type for each numbers. Numbers in Lua are internally always handled as double precision floating point numbers and were converted automatically when necessary.

At least this was true before Lua introduces the bitwise operators which demanded a second number representation, namely integer number types.

However, this is a special case and you will mostly not effected by it. The rare situations when you have to pay attention are discussed in section 17.2.4.2.

At the moment we consider numbers as floating point types exclusively.

```
1 print( 1 )    → 1
2 print( -12 ) → -12
3 print( 10000000000 ) → 10000000000
```

Notice that the numbers are never rounded into integers to. Hence:

```
1 print( 10 / 3 ) → 3.3333333333333
```

If you need only the integer part of a number, you can use the `math.modf` function. This function returns both, the integral and fractional part of a given floating number.

```
print( math.modf( 10 / 3 ) ) → 3    0.3333333333333
```

You can use the function to build your own floating to integer conversion:

```
function integer( number )
    int, dec = math.modf( number )
    return int
end
print( integer( 10 / 3 ) ) → 3
```

As said before: Lua automatically makes a type conversion if needed. In the code snippets above mostly from a floating point (double) representation to a string. But it works in both directions. For example:

```
s = "1.25"
print( 2.0 * s + 2.5 ) → 5.0
```

Here the string "1.25" is first converted into a floating point number, multiplied with 2.0 and added to 2.5 before the result (still a floating point number) is converted back to a string by the `print` function.

17.2.4.2 Integer versus floating point

We mentioned it before: Since bitwise operations don't make sense with floating point numbers, Lua tries to convert - if necessary - all number operands to integer values when they are part of a bitwise operation. The result of any bitwise operation is also an integer value. This is easy to understand. And to handle integer operations more convenient Lua 5.3 introduced an own integer type as a second number representation. But with the price of some little pitfalls where the automatic conversion doesn't work any longer.

Here are some examples you may consider when your Lua code doesn't work anymore after updating the analyzer program:

```
— Lua 5.2 using the bit32 library
print( bit.rshift( 1.5, 1 ) ) → 1
— Lua 5.3 with integrated bitwise operators
print( 1.5 >> 1 ) → error, number has no integer representation
— Lua 5.2
print( string.format( "%d\n", 10 / 3 ) ) → 3
print( string.format( "%04x", 1.25 ) ) → 0001
— Lua 5.3
print( string.format( "%d\n", 10 / 3 ) ) → bad argument #2 to 'format'
      (number has no integer representation)
print( string.format( "%04x", 1.25 ) ) → bad argument #2 to 'format'
      (number has no integer representation)
```

Every time you experience such a case it would be nice to query the number type of a variable or operation result. Luckily Lua have two. The first one (`type()`) returns the common type, i.e. a string, function, table, number.

```
x = "1.25"
print( type( x ) ) → string
print( type( x + 1.0 ) ) → number
print( type( type ) ) → function
```

But here we are more interested in the number type itself. Is a number of type integer or floating point. The Lua solution is the `math.type()` function.

```
print( math.type( 1.25 ) ) → float
print( math.type( 1 ) ) → integer
```

Please note that `math.type()` returns `nil` if its parameter is of no number type!

```
print( math.type( "1.2" ) ) → nil
```

But what can you do if you need an explicit type conversion? For instance if you must make sure, that the result of a calculation or operation definitely is an integer like in the `string.format(...)` example line above.

At this point you need to know that Lua makes an automatic floating point to integer conversion as long as the floating point exists only of an integer part. Numbers like 1.0 or 2E03.

```
print( string.format( "%04X", 1.0 ) ) → 0001
print( string.format( "%04X", 1E02 ) ) → 0064
```

KAPITEL 17. AN INTRODUCTION TO LUA

But:

```
x = 1E-02
print( string.format( "%04X", x ) ) —> bad argument #2 to 'format' (
    number has no integer representation)
```

You may already suspect it. The solution is simply to get rid of the decimal places. You can do this either with the `math.floor()` function or even easier with the Lua floor division operator `//`.

```
x = 100.5
print( string.format( "%04X", x // 1 ) ) —> 100
```

17.2.4.3 Hexadecimal constants

Despite the fact, that Lua compute exclusively with floating points you sometimes want to use other number bases like hex.

```
1 print( 0x1234 ) —> 4660
```

Lua allows to input hexadecimal values with a leading `0x`.

17.2.4.4 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 print( -0.05 ) —> -0.05
2 print( 10E-2 ) —> 0.1
3 print( 1.25E+6 ) —> 1250000
```

17.2.4.5 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 print( 2 > 1 ) —> true
2 x = 2 < 4
3 print( x ) —> false
```

17.2.4.6 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 print( "It's your code" ) —> It's your code
2 print( 'He says:"Hi"' ) —> He says: "Hi"
3 print( [[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.

17.2.4.7 Escape sequences in strings

Lua strings can contain the following escape sequences:

Escape sequence	Description
<code>\a</code>	bell
<code>\b</code>	backspace
<code>\f</code>	formfeed
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\t</code>	horizontal tab
<code>\v</code>	vertical tab
<code>\\</code>	backslash
<code>\"</code>	double quote
<code>\'</code>	single quote
<code>\ddd</code>	character with its numeric value ddd

The following examples show their use:

```

1 print( 'It\'s your code' )    → It's your code
2 print( "He says:\\"Hi\\"" )  → He says: "Hi"
3 print( "Tab1\\tTab2" )      → Tab1  Tab2
4 print( "Two backslashes \\\\" ) → Two backslashes\\
5 print( "Hello\\nworld" )    → Hello
6                               world
7 print( "Hello world\\033" )  → Hello world!
```

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"`.

17.2.4.8 nil

nil is a special type and indicates a non-value. Each variable has a **nil** value before its first assignment by default.

```
1 print( 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.

17.2.5 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 fruits as strings:

KAPITEL 17. AN INTRODUCTION TO LUA

```
1 fruits = { "apple", "banana", "orange" }
2 print( fruits[2] ) → banana
```

This statement in line 1 will initialize the first entry `fruits[1]` in the table with "apple", the second `fruits[2]` with "banana" and the third with "orange".

Please note again!

We mentioned before that Lua indexes always start with 1 and not with 0 (like in C/C++ and other languages). The table here behaves like a simple list. You can append a new element to the fruits with:

```
fruits[ #t+1 ] = "pear"
```

The expression above uses Lua's internal length operator `#` (see page 205) to get the next free table entry.

As an alternative you can use 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 print( "Count: ", #t ) → Count: 6
```

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 print( "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 print( "Count: ", #t ) → Count: 4
```

You can always replace one element with another one just simply by overwriting it. Each element in the list is accessible 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. We call such tables as numeric arrays or tables. 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 print( point.x, point.y ) → 5 10
```

A table storing data with a key/value relationship is sometimes called a dictionary.

17.2.5.1 Discontinuous tables with holes

Consider the following example:

```

1 errorCodes = {
2   [ 1 ] = "General error",
3   [ 2 ] = "Invalid command",
4   [ 3 ] = "Invalid function",
5   [ 7 ] = "Wrong checksum",
6   [ 11 ] = "Timeout"
7 }
8 print( #errorCodes ) --> 3 !!!

```

What's that? Lua initialize every not used index in a table with **nil**. When the table or array has nil elements (or gaps), the length operator treats the first occurring nil value as the end of the table. In our example the index 4 is nil just like the following table entries with index 5...10. So # stops at index 4.

A similar situation appears when the table is used as a dictionary, containing strings as keys for the key/value pairs. For instance:

```

1 translations = {
2   [ "apple" ] = "Apfel",
3   [ "banana" ] = "Banane",
4   [ "orange" ] = "Orange",
5   [ "pear" ] = "Birne"
6 }
7 print( #translations ) --> 0 !!!

```

Lua calculates a special hash number for every key which then is used as a table index where to store the value⁶. A perfect hash algorithm creates one unique number for every possible key (string). The hash number can be any value in the range of possible keys. In our example none of the hash numbers is 1. The very first table entry therefore is nil and the length operator already stops on the first index giving us 0.

If you really need the number of items in a not continuous table you cannot use the # operator but have to iterate over all entries. We will discuss this in the next section. For now remember:

Use the # operator on tables with care

The # operator only works on continuous one-dimensional arrays (tables without a key). Avoid using the length operator on arrays that may contain holes!

Luckily this is not a great disadvantage because you rarely need the number of entries in a not continuous table. You rather access the values by their keys. The size of the table also doesn't matter because Lua takes care about it.

17.2.5.2 Iterate through tables

You can iterate over a numeric (one-dimensional and continuous) array or table as we did before when iterate over a string.

```

1 fruits = {"apple","banana","orange","pear"}
2 for i=1,#fruits do
3   print( fruits[i] ) --> apple banana orange pear
4 end

```

⁶This is a simplification, but the principle is the same!

KAPITEL 17. AN INTRODUCTION TO LUA

This also works when you are using an index starting with 0. And even if you add table entries in an unsorted order.

```
1 fruits = {
2     [1] = "banana",
3     [2] = "orange",
4     [3] = "pear",
5     [0] = "apple",
6 }
7 for i=0,#fruits do
8     print( fruits[i] ) —> apple banana orange pear
9 end
```

It is only important, that the index numbers are continuous and without holes. The following example does not work:

```
1 fruits = {
2     [2] = "apple",
3     [3] = "banana",
4     [4] = "orange",
5     [5] = "pear",
6 }
7 for i=0,#fruits do
8     print( fruits[i] ) —> !
9 end
```

The first index in an one-dimension (numeric) table is 1 by default unless you start with 0 as shown above. Here we start with index 2 which left the first table entry (index 1) as **nil** and the length operator stops already here.

Lua provides you with two iterators `ipairs` and `pairs`. Both are called with a table as argument and return a key/value pair (thus the name).

`ipairs` is mostly used for numeric tables and returns index/value pairs. It stops on the first nil or non numeric key. You can rewrite our first example with `ipairs` as:

```
1 fruits = {"apple","banana","orange","pear"}
2 for i,v in ipairs(fruits) do
3     print( i, v )
4 end
```

Line 2 calls the iterator and stores the results in the variables `i` (key or index) and `v` (value). The iterator stops automatically at the the first nil entry or table end. Line 3 outputs the pair as:

```
1 apple
2 banana
3 orange
4 pear
```

More interesting is the second iterator `pairs`. It is specially used for associative tables and therefore particular suitable for tables with holes. But it also works for numeric tables. See again:

```
1 fruits = {
2     [2] = "apple",
3     [3] = "banana",
4     [4] = "orange",
5     [5] = "pear",
6 }
7 for i,v = pairs( fruits ) do
```

```
8 print( i, v )
9 end
```

The result is:

```
3 banana
2 apple
4 orange
5 pear
```

You may notice that the output is in an unsorted order. Even if the keys look like a sorted sequence from 2...5 they are internally used as input for the hash number algorithm. And since the resulting hash value is different from the original key, so is the order of the table entries.

17.2.5.3 Sorting tables

Associated arrays in Lua cannot be sorted by the keys as explained before. But you can create an additional numeric table to store and order the keys and use this one to iterate over the table you want to sort.

```
1 fruits = {
2     [2] = "apple",
3     [3] = "banana",
4     [4] = "orange",
5     [5] = "pear",
6 }
7 a = {}
8 for i,v in pairs( fruits ) do a[#a+1] = i end
9 table.sort(a)
10 for i,v in ipairs(a) do print( i, fruits[v] ) end
```

Now the result becomes:

```
2 apple
3 banana
4 orange
5 pear
```

Explanation: Line 7 creates a new table which we use to store the indexes (keys) of our `fruits` table.

In line 8 we iterate over the `fruits` and add every index (key) at the end of the table `a`. Table `a` becomes `{3, 2, 4, 5}` (see output before).

Table `a` is a numeric table⁷.! So we can sort it with the Lua `table.sort` function in line 9. The table `a` content is now `{2, 3, 4, 5}`.

In line 10 at least we iterate over this table using every item as index for our original `fruits`.

We can even put this code in our own iterator and use it instead of the standard `pairs()` and `ipairs()`. The iterator function must only return a new key/value pair every time it is called.

```
1 function sortedPairs( t )
2     local a = {}
3     for i in pairs(t) do a[#a + 1] = i end
4     table.sort( a )
```

⁷Strictly speaking: Numeric tables in Lua don't have a key, they consist only of a continuous sequence of values forming a one-dimensional array.

KAPITEL 17. AN INTRODUCTION TO LUA

```
5     local i = 0
6     return function()
7         i = i + 1
8         return a[i], t[a[i]]
9     end
10 end
11
12 for i,v in sortedPairs( fruits ) do
13     print( i, v )
14 end
```

The crucial part in the code above between line 5...9.

`sortedPair` does not return a key/value pair itself. Rather it delegates this to an anonymous function⁸ in line 6 which returns the pair.

Such kind of function like `sortedPair` is also called a factory function.

When we call `sortedPair` in the **for** loop in line 12, the function sorts the indexes of the passed table and stores the result internally in a new table `a`. It also initiates an index counter `i = 0` which is also kept in memory (but nevertheless not accessible from outside because it is local). Then it returns the actual iterating function which accesses the sorted indexes in line 6.

In other words: It creates a new sorting iterator function and initializes all the necessary stuff therefore the name factory function.

In the **for** loop every call of `sortedPair` calls the anonymous function in fact. And only the first call initiates all the internal variables (line 2...5).

17.2.6 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 digit⁹. Valid names are:

```
x          y          ABC          t1          _nm
aVeryLongVariableName          the_last_result
```

Invalid names throw an error

```
1 print( 2n )      —> malformed number near '2a'
```

17.2.7 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-sensitive, **and** is a keyword, whereas `And` and `and` AND are just two other and different identifiers!

⁸An anonymous function is a function without a name.

⁹Because the analyzer software and Lua itself too reserves the starting `_` for some language supplements we recommend to start a variable name without an underscore.

17.2.8 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 create it automatically.

17.2.8.1 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 print( 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 print( 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 little 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 print( x, y )  —> 10 5
```

17.2.8.2 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 accessible 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 print( 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.

KAPITEL 17. AN INTRODUCTION TO LUA

17.2.9 Operators

Operators are symbols, which activate calculation, when using them in combination with variables, values or results from expressions. Lua supports arithmetic, bitwise, conditional and logical operators. In addition a very helpful string concatenation operator.

17.2.9.1 Arithmetic operators

Lua supports the usual arithmetic operators: the binary + (addition), - (subtraction), * (multiplication), / (division), % (modulo), ^ (exponentiation) and unary - (negation).

+ - * / % ^ //

The last one // is a special divisor operator and returns a rounded quotient (towards minus infinity, which is the same as the floor of the division of its operands). Please note: In Lua Numbers are always represented as real (double-precision floating-point) numbers.

17.2.10 Bitwise operators

The Lua interpreter embedded in the analyzer program provides the following bitwise operators:

& bitwise AND
| bitwise OR
~ bitwise exclusive OR
>> right shift
<< left shift
~ unary bitwise NOT

Please not: All bitwise operands convert it operands to integers and the result of the bitwise operation is also always an integer.

17.2.10.1 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 print("abc" < "def" ) → true
2 print( 0 or true)   → 0
3 print( false or true) → true
```


17.2.10.2 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 print( 4 and 5 )      —> 5
2 print( 4 or 5 )      —> 4 (short-cut evaluation)
3 print( false and true ) —> false (short-cut evaluation)
4 print( a and 1 )     —> nil, because a wasn't specified
5 print( not false )   —> true
```

17.2.10.3 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 misinterprets the first dot as a decimal point and throws an error. Hence:

```
1 print( "Hello " .. "World" ) —> Hello World
2 print( "100 " .. "sec" )     —> 100sec
```

The concatenation operator always creates a new string and leaves the operands behind without modifications.

17.2.10.4 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 print( #"Hello World" ) —> 11
```

17.2.10.5 Precedence

The following is a list of all Lua operators and their order of precedence. The operators are listed from lowest to highest priority:

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

If in doubt, use explicit parentheses. It makes your code more readable and prevents you from an any additional look in this manual.

KAPITEL 17. AN INTRODUCTION TO LUA

17.2.11 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**.

17.2.11.1 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
```

17.2.11.2 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
```

17.2.11.3 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
```

17.2.11.4 Numeric for

Lua provides two **for** statements but we confine ourselves 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=from ,to ,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 print( m ) → 5.5
```

17.2.11.5 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 print( m ) → 2.5
```

17.2.12 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 an empty list specified by `()`.

17.2.12.1 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 `print(...)` or the `math.sqrt(x)`. Most of these functions are part of some module, others are defined by the analyzer.

17.2.12.2 Function definition

Functions are conventionally defined with the keyword **function**.

```
1 function fnc( arg1 , arg2 , ... )
2   → the function body
3 end
```

KAPITEL 17. AN INTRODUCTION TO LUA

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 print( 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.

17.2.12.3 Recursive function calls

In the following we will write a function to call the faculty of a given number (the mathematical equivalent to $n!$).

```
1 function faculty( n )
2   if n == 0 then
3     return 1
4   else
5     return n * faculty( n - 1 )
6   end
7 end
8
9 print( faculty( 5 ) ) → 120
```

Recursive functions often provide elegant and small solutions. But they have a price. Considering the code above. You see that the recursion depth grows with each increasing number. For a programming language this means: Every time a new function is called without leaving the calling scope the usage of stack memory¹⁰ rises until no further stack memory is left. Then the program crashes!

You can limit the maximum number of recursions. In our example just by limit the passed parameter n . But this requires a disciplined approach by the programmers - and sometimes they simply forget.

Luckily the analyzers Lua implements a built-in guard. Change the call to:

¹⁰The stack is part of the normal memory but reserved for functions to store their local variables.

17.2. THE LUA LANGUAGE

```
print( faculty( 30 ) ) —> [string "—[[...]:11: stack overflow
```

The analyzer software limits the recursion depth to specific number which we experienced as a good compromise between memory consumption and processor usage.

This is especially important if you write some protocol template code which is executed in real-time!

We can write the most recursive functions also in a non-recursive way. In our example an alternative faculty function looks like:

```
1 function faculty_non_recursive( n )
2     local sum = 1
3     if n <= 1 then
4         return 1
5     else
6         for i=2,n do
7             sum = sum * i
8         end
9     end
10    return sum
11 end
12
13 print( faculty_non_recursive( 100 ) ) —> 9.3326215443944e+157
```

This one consumes only processor power. The stack usage is very small, the local variables `sum` and iteration counter `i`.

17.2.13 Modules

A module is a package of functions for a special purpose. You already know the modules `string`, `math` and `table`. There are also modules belonging to the analyzer software which are not part of the standard Lua language. We explain these separately in chapter 18.

And: Don't forget your own modules written in Lua itself like our checksum example before.

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.

How you can extend the Lua language with your own modules was part of the Lua introduction before (see section 191). Here we focus on the standard modules coming with Lua 5.3.

17.2.13.1 Standard modules

The following standard modules are supported by Lua in the analyzer environment. They are working within all Views if not stated otherwise.

Module	Description
coroutine	Coroutines provide an independent thread of execution in a kind of cooperative multitasking. They are part of the basic Lua but we recommend not to use it unless you know what you are doing.
math	The mathematical library. It provides access to the mathematical functions defined by the C standard.

KAPITEL 17. AN INTRODUCTION TO LUA

os	Partly supported! Gives access to some OS specific functions like date, time and locale settings. Other functions for executing programs, read environment variables or mode/rename files which are part of the standard Lua <code>os</code> library are excluded!
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).

Lua modules NOT supported in the analyzer Lua implementation are:

```
debug (replaced by own debug module)
io
```

Provided functions in the partly supported `os` module are:

```
os.clock
os.date
os.difftime,
os.setlocale
os.time
```

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 analyser software. Take a look in the `doc` directory of your installation folder.

17.3 Lua restrictions

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 outran 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 in the Editor SKETCH buffer:

```
1 local x = 0
2 while true do
3     x = x + 1
4 end
```

```
--> [string "local x = 0..."]:-1: overrun of allowed executions
```

You can adapt the allowed executions in a specific range (which we experienced as a good compromise between CPU lasting execution and maximum blocking time of the interpreter) with the `config` module. It is described in detail in section [18.2.6](#).

17.4 Lua References

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 Serial Analyzer software. It also gives you a small outlook of all the language features Lua comes with.

Lua is free available and well documented. You will find a lot of 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>

You will find a very good tutorial too at:

<http://lua-users.org/wiki/TutorialDirectory>

A direct link to the original Lua manual for 5.3 as used in the analyser software is here:

<http://www.lua.org/manual/5.3/>

18

Lua analyzer extensions

The Serial Analyzer software offers some additional modules and data structures to link the capabilities of the analyzer with the Lua language. Some of them are fitted as best to the according view. Others can be used in all Views without limitations.

18.1 Modules overview

The table below lists all provided modules, the View availability and a short description in an alphabetical order. The modules are explained in detail afterwards with example code either for a given View or in a more general usage. Modules exclusively usable in certain Views like the ProtocolView or DataView are described in detail in the according View chapter. They are only listed here for completeness.

Name	Usage	Description
base16	Common	Encoding and decoding functions for base16 sequences as used in Modbus ASCII transmissions
bit32	OBSOLETE!	This library provided bitwise operations for 32 bit values but with the new bitwise operators in Lua you don't need it anymore! The module will be removed in one of the next versions. See 18.2.2 for the details.
<i>box</i>	<i>ProtocolView</i>	<i>Only ProtocolView! The box module is responsible to display the data of each telegram in the ProtocolView.</i>
bpack	OBSOLETE!	This function was replaced by the now native Lua <code>string.pack</code> function. See 18.2.3 for the details.
bunpack	OBSOLETE!	This function too was replaced by the now native Lua <code>string.unpack</code> function. See 18.2.3 for the details.
checksum	Common	Contains checksum algorithms for Modbus RTU (CRC16), Modbus ASCII (LRC), BAC-Net (CRC8 and CRC16), DNP3 and CRC16 CCITT (Kermit) and more.



Obsolete modules
bit32 and b(un)pack

KAPITEL 18. LUA ANALYZER EXTENSIONS

config	Common	The config module is intended to adjust some internal Lua interpreter settings. Replaces the former <code>cfg</code> Module.
data	<i>DataView</i>	<i>Only DataView! Provides random access to all recorded data bytes in the DataView and highlighting of single bytes relative to the cursor position.</i>
debug	<i>DataView, ProtocolView</i>	<i>Let you output any text or variable for debug purpose. The provided functions are the same for both views but the context may differ, so they are described in the DataView respectively ProtocolView chapter.</i>
record	Common	Provides information about the current or loaded record. For example the record start time, the bus wiring, the signal names and the analyzer type of the record.
string.dump	Common	Extends the original Lua string module with a hex/dec dump functionality like the telegram relating dump function.
telegrams	<i>ProtocolView</i>	<i>Only ProtocolView! The telegrams module gives you access to all recorded telegrams in the ProtocolView up to the present time.</i>
transmission	Common	Returns information about the underlying transmission protocol like baudrate, data bits, parity and stopbits. Replaces the former <code>protocol</code> module!

18.2 Common extensions for all Views

The following modules or functions are working in all Views and also in the editor sketch buffer (Sketch Example).

Please note! Some examples are nevertheless written for a specific View for a better understanding. If so, the example comes with an according head line.

18.2.1 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.

18.2. COMMON EXTENSIONS FOR ALL VIEWS

18.2.1.1 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.

ProtocolView 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
```

18.2.1.2 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.

Sketch Example

```
1 local seq = "hello world"
2 print( base16.encode( seq ) ) —> 40 20 00 00 00 00 03 E8
3 print( base16.decode( base16.encode( seq ) ) ) —> Hello world
```

18.2.2 The bit32 module

With the integration of Lua 5.3 the usage of the bit32 module becomes obsolete. We still provide it in the analyzer software for backward compatibility. Therefore here the description of it.

But keep in mind, that we will remove the module sooner or later since the Lua native bitwise operators are a lot easier to use and especially easier to read¹.

A remark to the bit width: The bit32 module was limited to 32 bit (therefore the name). Greater values are normalized to the remainder of its division by 2^{32} . This is also valid for the new bitwise operators on Windows systems except for

¹A bit32 module was introduced to Lua in version 5.2. Since Lua 5.3 bitwise operators are part of the language itself.



bit32 is obsolete

KAPITEL 18. LUA ANALYZER EXTENSIONS

the Linux 64 bit program version, where integer operations are handled on a 64 bit base. And now, the description of the obsolete `bit32` module:

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 `bit32` module expand the integrated Lua interpreter with the following functions:

Function	Description
<code>band</code>	returns the bitwise AND of its arguments <code>x1</code> and <code>x2</code> , for instance <code>bit32.band(0xFF, 0x01)</code>
<code>bor</code>	returns the bitwise OR of its arguments <code>x1</code> and <code>x2</code> , for instance <code>bit32.bor(0xFF, 0x01)</code>
<code>bxor</code>	returns the bitwise exclusive or (XOR) of its arguments <code>x1</code> and <code>x2</code> , for instance <code>bit32.bxor(0xFF, 0x0F)</code>
<code>bnot</code>	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 <code>bit32.bnot(0x55)</code>
<code>lshift</code>	returns bitwise logical left-shift of its first argument <code>x</code> by the number of bits given by the second argument <code>n</code> . For instance <code>bit32.lshift(0x100, 2)</code>
<code>rshift</code>	returns bitwise logical right-shift of its first argument <code>x</code> by the number of bits given by the second argument <code>n</code> . For instance <code>bit32.rshift(0x1FF, 1)</code>

Sketch Example

```
1 print( string.format( "%X", bit32.band( 0xFF, 0x03 ) ) ) --> 3
2 print( string.format( "%X", bit32.bor( 0x03, 0x10 ) ) ) --> 13
3 print( string.format( "%X", bit32.bxor( 0x1001, 0x1000 ) ) ) --> 1
4 print( string.format( "%X", bit32.bnot( 0x1001 ) ) ) --> FFFFFFFE
5 print( string.format( "%X", bit32.lshift( 0x1000, 2 ) ) ) --> 4000
6 print( string.format( "%X", bit32.rshift( 0x4000, 2 ) ) ) --> 1000
```

18.2.3 The functions `bpack` and `bunpack`

Both functions came originally from the Lua `lpack` library. They are still an integrated part of the Lua interpreter in the analyzer software for backward compatibility. But we will remove it sooner or later since Lua now gives you the same functionality via its own `string` module.

The following description as long as section 18.2.4 about the necessary code adaptations may help you to update your templates for future software versions.

The functions `bpack` and `bunpack` provide you with all necessary kind of transformations you need to convert any byte sequence into a certain number type and visa versa.

`bunpack` is likely the function you need most. It works like the `scanf` function in C. A given string or byte sequence is translated in one or more numbers



`bpack` and `bunpack` are obsolete!

18.2. COMMON EXTENSIONS FOR ALL VIEWS

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. But the same conversion rules apply to `bpack`.

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).
I	Convert a series of 4 bytes into an unsigned integer number (32 bit).
i	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.

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:

KAPITEL 18. LUA ANALYZER EXTENSIONS

ProtocolView Example

```
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)
```

bpack(format, value1, values2, ...)

- **format:** The conversion format applied to the given value(s).
- **value1, values2, ...:** A list of Lua numbers separated by a comma.

Example Sketch

```
1 seq = string.dump( bpack( ">f>l", 2.5, 1000 ) )
2 print( seq ) —> 40 20 00 00 00 00 03 E8
```

Granted: You will need the `bpack` (or the new `string.pack`) only in rare occasions - if at all. But consider you want to check the string representation of a floating point number or a 64 bit long integer value. Then the `string.pack` will prove itself as a very helpful and mighty function.

18.2.4 string.pack and string.unpack

As mentioned before: With the integration of Lua version 5.3 scripts/templates now can use Lua's own pack/unpack function pair. Both are part of the native Lua `string` module. And when you look more closely, this makes sense since the operation takes a string or results in a string.

The new `string` module functions are working nearly identical to `bpack` and `bunpack`. Even the format specifiers are identical. You will find all details here: <http://www.lua.org/manual/5.3/manual>.

The only difference you have to consider with regards the order of the parameters and results between the old `bunpack` and new `string.unpack` functions. Here in comparison:

18.2. COMMON EXTENSIONS FOR ALL VIEWS

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

The `position` parameter is optional in both cases!

The resulting `pos` is now the last result. And the `format` parameter is now the first as it is already in the `bpack` and also `string.pack` function. Which is just consistent. Adapted to our former ProtocolView example we get:

```
1 function out()
2   — extract the binary data of a Modbus-ASCII telegram
3   local tg = telegrams.this()
4   — assume its a Write Single Register telegram
5   local dev, fnc, reg, val, crc, pos = string.unpack("bb>H>H>H", tg : string())
6   end
7 end
```

18.2.5 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 13.3.2 in case your checksum isn't listed here and you have to write it by yourself.

Function	Description
<code>crc8_bacnet</code>	the 8 bit checksum as used in the BACNet (header) telegrams.
<code>crc16_bacnet</code>	the 16 bit checksum as used in the BACNet protocol.
<code>crc16_ccitt_kermit</code>	calculates the crc16 checksum of the given data string using another start value as used in CCITT kermit.
<code>crc16_df1</code>	calculates the crc16 checksum of the given data string as used in the Allen-Bradley DF1 protocol. The returning result is a 16 Bit value.
<code>crc16_dnp3</code>	calculates the crc16 checksum of the given data string as used in the DNP3 protocol. The returning result is a 16 Bit value.
<code>lrc</code>	returns the checksum of the given data string calculated as Longitudinal redundancy check (used in Modbus ASCII).
<code>crc16_modbus</code>	calculates the Modbus RTU (CRC16) checksum of the given data string and return it as a 16 bit integer.

18.2.5.1 checksum.crc8_bacnet

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

KAPITEL 18. LUA ANALYZER EXTENSIONS

`checksum.crc8_bacnet(String)`

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

Sketch Example

```
1 print( checksum.crc8_bacnet( "Hello world" ) ) —> 157
```

Protocol Example

```
1 function out()
2   local tg = telegrams.this()
3   — checksum header crc
4   local header_crc = checksum.crc8_bacnet(tg:string():sub(3,8) )
5   ...
6   — checksum data crc
7   local datalength = tg:data(6) * 256 + tg:data(7)
8   local data_crc = checksum.crc16_bacnet(tg:string():sub(9, 9+
9     datalength+1) )
9 end
```

18.2.5.2 `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

Sketch Example

```
1 print( checksum.crc16_bacnet( "Hello world" ) ) —> 20985
```

ProtocolView Example

```
1 function out()
2   local tg = telegrams.this()
3   — checksum header crc
4   local header_crc = checksum.crc8_bacnet(tg:string():sub(3,8) )
5   ...
6   — checksum data crc
7   local datalength = tg:data(6) * 256 + tg:data(7)
8   local data_crc = checksum.crc16_bacnet(tg:string():sub(9, 9+
9     datalength+1) )
9 end
```

18.2. COMMON EXTENSIONS FOR ALL VIEWS

18.2.5.3 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

Sketch Example

```
1 print( checksum.crc16_ccitt_kermit( "Hello world" ) ) --> 41426
```

ProtocolView Example

```
1 function out()
2   — the following code checks the content of the entire message
3   — except for the last two byte (which are the checksum itself)
4   cks = checksum.crc16_ccitt_kermit(telegrams.this():string():sub
5     (1,-3))
6   box.text{ caption="Checksum", cks }
7 end
```

18.2.5.4 checksum.crc16_df1

The crc16 checksum algorithm for Allen-Bradley DF1 telegrams. The result is a 16 bit integer value.

checksum.crc16_df1(String)

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

ProtocolView Example

```
1 function out()
2   local tg = telegrams.this()
3   — extract the application data and substitute DLE DLE
4   local data = tg:string():sub(6,-5):gsub('\016\016','\016')
5   — checksum validation, the CRC16 is calculated from the STN (3th
6     byte),
7   — STX (5th byte), the application data AND the final ETX
8   local data = tg:string():sub(3,3)..tg:string():sub(5,5)..data..
9     tg:string():sub(-3,-3)
10  local cks = checksum.crc16_df1(data)
11  box.text{ caption="Checksum", cks }
12 end
```

18.2.5.5 checksum.crc16_dnp3

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

checksum.crc16_dnp3(String)

KAPITEL 18. LUA ANALYZER EXTENSIONS

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

ProtocolView Example

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

18.2.5.6 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

ProtocolView Example

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

18.2.5.7 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

The next example is for the editor sketch buffer again. We construct a Modbus RTU master command in line 2 (Lua allows the input of any binary value as decimal values via `\ddd`. Line 3 just serves to confirm our sequence in hex. The checksum is calculated from the Modbus command except for the CRC16 checksum itself (the last two bytes). We use the Lua `string.sub` function to extract all bytes less the last two. The checksum result is in a different byte order because Modbus transmits the low byte first.

Sketch Example

```
1 — Modbus RTU Read Holding Register, address = 2, count = 5
2 seq = "\001\003\000\002\000\005\036\009"
3 print( seq:dump() ) —> 01 03 00 02 00 05 24 09
4 cks = checksum.crc16_modbus( seq:sub( 1, -3 ) )
5 print( string.format( "%04x", cks ) ) —> 0924
```

18.2. COMMON EXTENSIONS FOR ALL VIEWS

ProtocolView 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
```

18.2.6 The config module

The config module is rarely required and mainly intended to change internal settings of the Lua interpreter. Since this internal settings are chosen with purpose modifying them can leads to unexpected behaviours.

Nevertheless there are reasons to apply your own settings to the interpreter. One - and actually the only available - parameter limits the number of operations for a script. Without limitation the interpreter may run in endless loops leaving the application inoperable.

Note! cfg module renamed to config

In the analyzer version 5.0 the former cfg module is renamed to config for a better understanding.



Consider the following code snippet:

```
1 while true do end
```

The line will never ends and blocking the whole program window. Thanks to the MultiView concept, an active recording is not affected and all other open Views will remain operable. Still it is annoying and forces you to end the given window with the task manager.

To avoid this the internal Lua interpreter will stop the execution after a certain amount of internal operations independent of the script.

The number of operations or command executions before the application becomes inoperable depends on the processor power. Therefore we have chosen a conservative value before the script execution is stopped.

It rarely happens, but in case of a very CPU extensive script, perhaps with some data decryption and a complex Lua checksum code you may see an error message like this:

```
Overrun of allowed executions!
```

You can produce this error by input the endless loop code above in the editor sketch buffer!

By default the Lua interpreter aborts a script after 10000 internal execution units which is sufficient for also more comprehensive template scripts. If your script still needs more, increase the allowed executions by add the following line at the beginning of your script:

```
1 config.setmaxop(1000000)
```

The allowed range of execution numbers is 10000...1000000.

KAPITEL 18. LUA ANALYZER EXTENSIONS

18.2.7 The record module

The `record` let you query several information about the current or loaded record which may be necessary when your Lua code depends - for instance - on the bus wiring or the analyzer type. It also provides you the record start time and signal names.

Function	Description
<code>analyzer</code>	returns the used analyzer type. 0 : MSB-RS232, 1 : MSB-RS485, 2 : MSB-RS232-PLUS, 3 : MSB-RS485-PLUS
<code>buswiring</code>	returns the selected bus wiring. 0 : 2-Wire-Tap, 1 : 2-Wire-Segment, 2 : 4-Wire-Tap, 3 : 4-Wire-Segment
<code>signalnames</code>	returns all eight signalnames as a list from Signal1 to Signal8. For instance: <code>s1,s2,s3,s4,s5,s6,s7,s8 = record.signalnames()</code>
<code>starttime</code>	supplies the start time (date) of the current record as the so called Unix time (represents the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC)).

18.2.7.1 `record.analyzer`

Returns the analyzer type currently used either in the loaded record or in the active recording. This function is mainly intended to handle difference type of analyzers (especially the new PLUS series) in your Lua scripts.

`record.analyzer()`

SKETCH Example

```
1 analyzers = {
2   [ 0 ] = "MSB-RS232",
3   [ 1 ] = "MSB-RS485",
4   [ 2 ] = "MSB-RS232-PLUS",
5   [ 3 ] = "MSB-RS485-PLUS"
6 }
7 print( analyzers[ record.analyzer() ] )
```

18.2.7.2 `record.buswiring`

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

`record.buswiring()`

ProtocolView Example

```
1 function out()
2   local tg = telegrams.this()
3   if record.buswiring() == 1 or record.buswiring() == 3 then
```

18.2. COMMON EXTENSIONS FOR ALL VIEWS

```
4      — we can use the tg:dir() to distinguish between request and
5      response
6  end
end
```

18.2.7.3 record.signalnames

Returns the used signal names, either set in the loaded record or in the signal settings of the control program.

record.signalnames()

SKETCH Example

```
1 local signames = {record.signalnames()}
2 print( signames[3] ) —> TxD
```

18.2.7.4 record.starttime

Returns the seconds since the Epoch (00:00:00 UTC, January 1, 1970). You can format and display the result with the `os.date` function.

record.starttime()

ProtocolView Example

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

18.2.8 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.

18.2.8.1 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 analyzer software!

string.dump(str, base, sep)

- **str**: The Lua string you want to hex dump.
- **base**: The used number base, default is hex (base 16).

KAPITEL 18. LUA ANALYZER EXTENSIONS

- **sep**: Replaces the default space separator with any character or string. An empty string suppresses the separator completely.

ProtocolView 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 (dec)",
10            text=bindata:dump( bindata, 10, ":" ) }
11 end
```

18.2.9 The transmission module

Replaces the former `protocol` module!

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



protocol module renamed to transmission

In the analyzer version 5.0 the protocol module is renamed to transmission to avoid misconceptions with the protocol layer used in the ProtocolView.

Function	Description
baudrate	Returns the baud rate used in the current recording or settings.
bitpause	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.
bytepause	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.
databits	Queries the used number of data bits. The result is a value in the range 5...9.
parity	Returns the parity setting of the current recording as following: None = 0, Odd = 1, Even = 2, Mark = 3, Space = 4.

18.2.9.1 transmission.baudrate

Returns the baudrate as used in the current record or settings.

transmission.baudrate()

18.2. COMMON EXTENSIONS FOR ALL VIEWS

ProtocolView Example

```
1 function split( data, intval, alter, str )
2   — start a new telegram after a pause of 33 bits
3   if intval > 33 / transmission.baudrate() then
4     return STARTED
5   end
6   return MODIFIED
7 end
```

18.2.9.2 transmission.bitpause

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

transmission.bitpause(bits)

- **bits** number of paused bits.

ProtocolView Example

```
1 function split( data, intval, alter, str )
2   — Profibus specifies a pause of 33 bits as a telegram delimiter
3   if intval > transmission.bitpause( 33 ) then
4     return STARTED
5   end
6   return MODIFIED
7 end
```

18.2.9.3 transmission.bytepause

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

transmission.bytepause(bytes)

- **bytes** number of paused bytes.

ProtocolView Example

```
1 function split( data, intval, alter, str )
2   — Modbus RTU specifies a pause of 3.5 bytes as a telegram
   delimiter
3   if intval > transmission.bytepause( 3.5 ) then
4     return STARTED
5   end
6   return MODIFIED
7 end
```

18.2.9.4 transmission.databits

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

transmission.databits()

KAPITEL 18. LUA ANALYZER EXTENSIONS

ProtocolView Example

```
1 function output()
2     local tg = telegrams.this()
3     if transmission.databits() > 8 then
4         — discards the 9th bit and uses a warning red background
5         box.text{ caption="9 Bit", text=tg:data % 256, bg=0xFF0000, fg=0
6             }
7     else
8         box.text{ caption="8 Bit", text=tg:data }
9     end
end
```

18.2.9.5 transmission.parity

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

transmission.parity()

ProtocolView Example

```
1 function out()
2     if transmission.parity() ~= 2 then
3         — do something when parity is not even
4         box.text{ caption="Warning", text="We need an even parity" }
5         return
6     end
7 end
```

18.3 Lua modules for individual views

There are several Lua modules which only work in a specific view (the cur-sive listed module in the overview table at the beginning of this chapter). For instance: The `box` and `telegrams` modules need the environment of the `ProtocolView` to access and display the transmitted telegrams.

The same applies for the `databytes` module. It is intended to provide you a random access to every recorded byte and cannot be executed outside the `DataView`.

All of them are closely linked with the according view. We therefore explain them in detail in the according View chapters.

`box` module see [13.8.1](#)

`telegrams` module see [13.8.8](#)

`databytes` module see [11.7](#)

`debug` module see `ProtocolView` [13.8.2](#) and `DataView` [11.7.2](#)

19

Lua Protocol dialogs

The design of the Lua protocol template provides an uttermost degree of flexibility. But it lacks of an interactive way to change the behaviour of the applied template. For instance to adapt certain protocol parameters or search/filter rules. An easy to use Lua GUI framework fills this gap.

Imagine you have a Modbus RTU transmission with a different interframe delay time. This is not uncommon but forces you to adapt the Modbus template code every time you have to deal with such one. This becomes especially annoying when you analyse several Modbus transmission with different specifications. Another example is the IEC60870-5-101 protocol which exists in at least two variants. One with a single address byte, a second one using two address bytes instead.

Every time you need a different protocol setup you have to open the template editor, search for the according code line and modify it to your needs. You can - of course - maintain various versions of the same template which may vary only by a constant value, but this only a poor solution in the absence of a better way.

What we need is a flexible way to define protocol specific parameters without touching a single code line, in other words a graphical user interface.

It is clear, that every user interaction depends on the currently selected protocol template. A Modbus parameter setup is obviously different from - let's say - Profibus settings. Thus the implementation of any graphical user interface (GUI) must be part of the according protocol template. In particular because every user interaction affects the current telegram display.

The ProtocolView allows you to write/program your own setup dialog in the template itself and open/start it by clicking the 'Setup' button below the telegram window.

A simple dialog contains only one or a few elements. But you can also write dialogs with as many elements you want. Some elements may be even linked together, which means: The appearance of a element changes depending on other elements. For example: Disable an element if another one has a certain value.

And you will have to choose if an user interaction only affects the visible telegrams (e.g. switch between a hex or decimal checksum display) or if it is

KAPITEL 19. LUA PROTOCOL DIALOGS

necessary to parse the whole data stream again because the split condition has changed.

19.1 How does it work?

Before we start to explain the writing of dialogs in detail, here an short overview how a typical dialog looks like and how you can use it to pass parameters to your protocol template.

Every scripted Lua dialog is encapsulated in a special dialog frame, providing you with the dialog content and - the most important thing - an 'Apply' button. The latter invokes that part of the Lua code which passes the input parameters to the template and triggers a new evaluation and/or refresh of the protocol window. According to this every dialog consists of two main parts:

- 1 The Lua function `dialog` is holding the code for the graphical user interface.
- 2 The Lua function `apply` is called by the 'Apply button and updates the protocol window.

A dialog can - of course - consist of a lot more functions. But these two above are essential for every functional ProtocolView dialog.

The ProtocolView executes the `dialog` function with an independent Lua interpreter every time you click the  button. This separates the GUI from the data stream evaluation.

In the `apply` function you ask the dialog elements for their current entry (made by the user) and pass the result to the protocol script afterwards.

19.2 The dialog framework

Wherever GUIs (Graphical User Interface) are concerned, one of the first questions is always: How to arrange the several control elements?

There are various approaches to put controls together. One is to position them absolutely by specifying position and size. But this would be cumbersome and laborious.

The analyzer software uses a more elegant way to align your controls more or less in an automatic way. For this it covers the dialog area with an invisible grid. The grid is not limited in width and height and the columns width and row height are automatically adapted to the controls size.

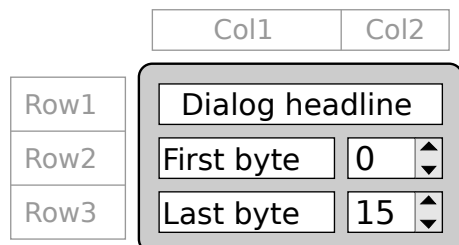
You can imagine the grid like a letter (or type) case. Every box in it can contain a single control. Whereby a single control also can span across several horizontal grid cells. By arranging your controls in columns and rows you are able to produce nice and user-friendly graphical interfaces.

To place a control in a certain box, simply pass the column and row index. The invisible grid expands as necessary and adapts the width of the column (or height of the row) to the needed size of the placing control.

And more: If you want to replace an element in a box (for instance to adapt the element after an user interaction), just overwrite the existing one by putting the

19.3. ADD A TEMPLATE DIALOG

new element into this box (or cell). Consider the following figure:



This little dialog consists of two columns and three rows which gives you a grid of 2 x 3. It serves as an input of a range of numbers defined by a first and last byte. The first row is completely occupied by a headline (using an additional `span=2` parameter).

The second row has a text label 'First Byte' in the left (first) column and a so called Spin Control to pass a number value by increment or decrement the input on the second column.

The third row is to specify the last byte by again a text label 'Last byte' and another Spin Control to pass the last byte value.

Not used grid cells between elements stay empty. This gives you an easy way to group controls together by keep them spatially away from others.

And that is the corresponding Lua code:

```
1 function dialog()
2   — the headline label spanned over two columns
3   widgets.Label{ name="headline", text="Dialog headline",
4                 row=1, col=1, span=2 }
5   — label and first byte input control
6   widgets.Label{ name="label1", text="First byte", row=2, col=1 }
7   widgets.SpinCtrl{ name="first", row=2, col=2,
8                   min=0, max=255, value=0 }
9   — label and last byte input control
10  widgets.Label{ name="label2", text="Last byte", row=3, col=1 }
11  widgets.SpinCtrl{ name="last", row=3, col=2,
12                  min=0, max=255, value=15 }
13 end
```

Don't worry, if you have some difficulties to understand this script. We will explain all details later. Here it should give you only a first impression how easily you can implement a dialog with a few code lines.

19.3 Add a template dialog

All the coding for your dialog has to be done in the function `dialog`. This is the only function which the Lua dialog interpreter executes when it evaluates the GUI (graphical user interface).

In case of more complicated user interfaces, you can outsource part of the code into other functions. But each of them has to be called at least from within the `dialog` context.

The supported graphical elements are pooled all in one module named `widgets`. A module in Lua is like a library in other programming languages. You can ima-

KAPITEL 19. LUA PROTOCOL DIALOGS

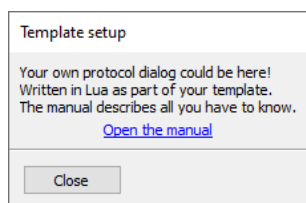
gine it as a collection of functions which deal especially with widgets.

To call a certain module function, Lua expects the module name, followed by a dot and the function name. For a button it's like this:

```
1 widgets.Button{ PARAMETER... }
```

New language features are best explained with an example. Open the dialog tutorial project in the examples folder (`Tutorial-Dialogs`). You can either double click it with your file explorer or first start the analyzer program and click 'Examples' in the file menu of the control program. Choose the Tutorial folder and open the project file. Depending on the selected analyzer on start the program may ask you to restart with the according type.

I assume that you know the example from our template lesson in the Protocol-View chapter (see 13.3.2). If not, please read it first.



Empty dialog

Now let's start with an empty dialog. At first this is nothing else than an empty or even not existing `dialog` function.

```
1 function dialog ()
2 end
```

If you click `Setup` in the ProtocolView you will get a dialog as shown on the left. The dialog is still empty and therefore displays a default title and a note with a link to the according manual chapter (this one).

You can add the `dialog` function at any point in the template except for inside another function (which is allowed by Lua). But we recommend to put all GUI code at the end. The content of the function block is not limited to widget functions alone. You can do all kind of Lua coding here, but avoid time-consuming stuff, otherwise your dialog will become inoperable.

19.3.1 Add widgets elements to your dialog

In our example transmission a master requests temperature, moisture and pressure from three sensors. The telegram rules are simple and similar to Modbus ASCII. Every telegram starts with a colon ':' and ends with a carriage return linefeed (CRLF). It looks like:

Time	SOF	Address	Function	Checksum	EOS	
2.339189	:	2	Moisture	C4	0D 0A	
Time	SOF	Address	Function	Value	Checksum	EOS
2.351468	:	2	Moisture	58.52%	7D	0D 0A

We already discussed how to display the data optionally in metric or Anglo-American format in chapter 13.4.2 and solved it by using a filter.

This time we will use a real dialog to apply some individual settings to our protocol template. We will start with a switch between metric and Anglo-American unit output.

A single selection between two or more options is usually realized by using a so called radio box. Add the following code at the end of the Tutorial template:

```
1 function dialog ()
2     widgets.RadioButton{ name="unit", col=1, row=1,
3                         label="Unit"
4                         choices={"Metric", "Anglo-American"}}
5 end
```

19.3. ADD A TEMPLATE DIALOG

Every widget needs at least an unique name and a position specified by a column and row value. For the moment you only have to know that the name must be singular because we need it for accessing the widget data later.

Save your modifications and click the `Setup` to test the dialog.

Test your dialog

You can always test you dialog by clicking the `Setup` button after saving your code modifications.

The dialog shows the radio box and you can switch between the two physical units. At the moment it doesn't make any effects to the telegram output because the script itself has no idea what you did in the dialog.

Passing the result of the user interaction to the script is subject in the next session. We will stay a little longer and discuss some aspects regarding the arrangement and usability of widget objects.

To do so, we first add an additional text (description) above the radio box and corrected the row parameter of the radio box which is now displayed in the second row (the first row now belongs to the label widget).

```
1 function dialog()
2     widgets.Label{ text="Individual tutorial protocol settings",
3                   col=1, row=1 }
4     widgets.RadioBox{ name="unit",
5                      label="Unit",
6                      col=1, row=2,
7                      choices={"Metric", "Anglo-American"}}
8 end
```

The result is shown on in the picture 'Dialog with radio box'.

The dialog mechanism tries to use the smallest space for every widget by default. As you can see in the picture, the radio box consumes only a part of the available width. This not only looks ugly, it is no good usability style too!

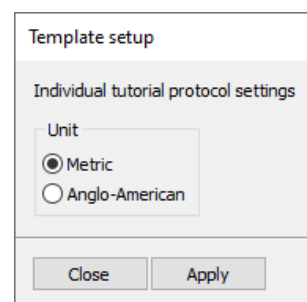
You can instruct the 'arrangement' mechanism known as sizer to use all space with the widgets parameter `fill=true`.

```
1 function dialog()
2     widgets.Label{ text="Individual tutorial protocol settings",
3                   col=1, row=1 }
4     widgets.RadioBox{ name="unit",
5                      label="Unit",
6                      col=1, row=2, fill=true,
7                      choices={"Metric", "Anglo-American"}}
8 end
```

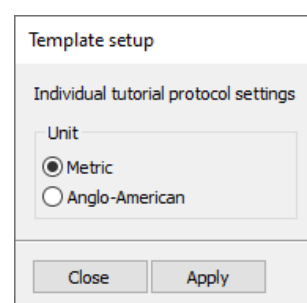
That looks better! The radio box now uses (fills out) the whole width of the dialog, here specified by the length of the text in the `Label` widget.

The current dialog arrangement is a grid with one column and two rows. But how do we 'span' a widget over several columns? The `fill` parameter is limited to a single 'cell' in the grid.

We indicated an according `span` parameter in the framework introduction 19.2. For a better understanding let us add a further widget to choose between several EOS settings. (Just in case, our telegram specification allows different



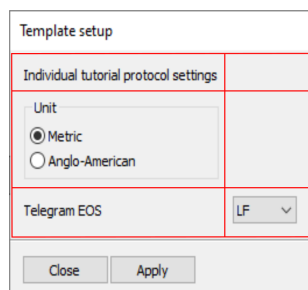
Dialog with radio box



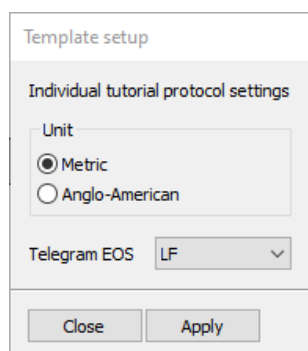
Filled radio box

KAPITEL 19. LUA PROTOCOL DIALOGS

EOS like CRLF, LF, CR or LFCR). The appropriate widget for such a thing is a `Choice` element.



Dialog grid 2 x 3



Dialog improved

```
1 function dialog()
2     widgets.Label{ text="Individual tutorial protocol settings",
3                   col=1, row=1 }
4     widgets.RadioButton{ name="unit",
5                          label="Unit",
6                          col=1, row=2, fill=true,
7                          choices={"Metric", "Anglo-American"}}
8     widgets.Label{ text="Telegram EOS",
9                   col=1, row=3 }
10    widgets.Choice{ name="eos",
11                  col=2, row=3, fill=true,
12                  choices={"LF", "CR", "LFCR", "CRLF" } }
13 end
```

Since the `Choice` element does not provide us with a label, we put a `Label` widget in column 1 and assign the `Choice` to column 2 (line 9 and 11).

The framework works exactly as instructed. All elements except for the EOS selection (`Choice`) are arranged in the left (first) column and the EOS chooser in the right column (two) - which looks not so good (see the picture on the left). Here comes the `span` parameter into play. We can force the top label and the radio box to cover more than one column by adding the argument `span=columns` to the widget call. The following listing shows the improved code (line 3 and 6) and the resulting dialog appearance:

```
1 function dialog()
2     widgets.Label{ text="Individual tutorial protocol settings",
3                   col=1, row=1, span=2 }
4     widgets.RadioButton{ name="unit",
5                          label="Unit",
6                          col=1, row=2, fill=true, span=2,
7                          choices={"Metric", "Anglo-American" } }
8     widgets.Label{ text="Telegram EOS",
9                   col=1, row=3 }
10    widgets.Choice{ name="eos",
11                  col=2, row=3, fill=true,
12                  choices={"LF", "CR", "LFCR", "CRLF" } }
13 end
```

The underlying framework resizes all elements to fit best in the available space. This accomplished, we can turn our attention to the question, how to pass the settings in the dialog to the part of the code which splits the data stream in single telegrams and displays the telegram content. First we must query the user inputs from the dialog elements.

19.3.2 Apply the user settings

We already mentioned that each widget element needs an unique name. This is not entirely correct because all `Label` elements in our example lacks of any individual name. A widget only needs a unique name in case you want to access the object later. Labels are in most case not a subject of user interaction. In contrary we must be able to read the internal state of a radio box to fetch its current value.

In our example the user can change the unit and/or the EOS sequence. We wrote in the introduction, that the `dialog` function is usually escorted by the `apply` function which is triggered every time the user clicks the dialog [Apply](#)

19.3. ADD A TEMPLATE DIALOG

button. This is the first place to query the user input and pass the values to the code responsible for the telegram evaluation and output.

We begin with the code:

```
1 function apply()
2     unit = widgets.GetValue( "unit" )
3     eos = widgets.GetValue( "eos" )
4     debug.clear()
5     debug.print( "Unit="..unit , "EOS="..eos )
6 end
```

The function queries the current selection (value) of the radio box and the selected choice item and assign them to the global values `unit` and `eos`. Note! All Lua variables are global if not declared otherwise with the keyword `local`! In line 5 we output both values in the debug output window after we cleared the debug window in line 4. Just open the debug window in the 'View' menu or with **Ctrl** + **Alt** + **O** to see the result. Every time you click the apply button, the script updates the values in the debug window. This makes the debug window a very valuable tool during the process of code writing.

The sensor number value is formatted in the function `GetFunctionValue()`. For a switching between the two units 'Metric' and 'Anglo-American' we must first extend the function to handle Anglo-American units too.

```
1 function out( filter )
2     — skip unchanged code here
3     function GetFunctionValue( number, value )
4         if unit == "Metric" then
5             local formats = { "%.2fC", "%.2f%%", "%imBar" }
6             return string.format( formats[ number ], value )
7         else
8             — conversion factor for Temperature, Moisture and Pressure
9             local formats = { "%.2fF", "%.2f%%", "%.2fpsi" }
10            convs = {
11                — function to convert celsius to fahrenheit
12                [1] = function(c) return c * 1.8 + 32 end,
13                — the moisture is always in percent
14                [2] = function(m) return m end,
15                — function to convert mbar to psi
16                [3] = function(p) return p * 0.01450377 end
17            }
18            return string.format( formats[ number ],
19                                convs[number](value) )
20        end
21    end
```

We explain this code in detail in section [13.4.2](#).

In line 4 we test the value of the global `unit` variable. If it is 'Metric' we format it as before. Otherwise we convert the temperature to Fahrenheit and the pressure to psi. The moisture is always in percent.

Now click **Setup** and switch between the two unit systems. Don't forget to click the 'Apply' button in the dialog to trigger the `apply` function.

You will observe - nothing! With the exception that the units are now Anglo-American nothing happens! What's wrong?

We can assure you, that the code is correct. But we have omitted a small,

KAPITEL 19. LUA PROTOCOL DIALOGS

nevertheless important detail. (We did it on purpose!) The template script is executed by different Lua interpreters each with its own global variables. One interpreter is responsible for the GUI (executing the `dialog` and `apply` functions). This one owns the global variables `unit` and `eos`. The interpreter handling the telegram display by running the `output` function does not know the variable `unit`. From it's point of view `unit` is `nil` which leads to the execution of the `else` block.

Interesting, isn't it? But it surely is not the answer you may expected. So let us explain the reasons for this implementation and the details behind before we come back.

19.3.3 Passing data between dialog and script

Since the parsing of the data stream (in the `split` and `output` function) is handled by different Lua interpreters, you cannot simply use a global Lua variable to share data between the dialog functions and the rest of the template script. This initially seems rather unusual but it really make sense if you consider, that the evaluation of the incoming data must not under any circumstances be stopped when operating the dialog!

It also provides a clear separation between the actual protocol code and the user interface and keeps the interpreter running the `split` function lean and thus also very fast.

So even if you use a global variable `x` in the `split`, `output` and `apply` function, they are - in reality - four different variables!

Realize that every protocol template is executed by four!! Lua interpreters. The first two run the `split` function to divide the data stream into telegrams. One interpreter for each direction. The third calls the `output` function to display the telegrams in the telegram window. And the fourth interpreter runs the GUI by executing the `display` function. All of them are running independent of each other.

But even if there are the independent interpreters, they all share the same code which makes it easy to reuse special functions in all four environments.

This design makes the ProtocolView not only very robust against code failures, it also avoids the unintentional mixing when using global variables (which is a common problem with globals in other languages too).

But how can we shared data between two (or more) Lua instances if global variables are not an option? Or to be more precise: How can we pass the data from the GUI code to the rest of the template?

The `widgets` module provides a simple mechanism to share any kind of Lua data types (number, string, boolean) with the other interpreters. It is realized as an anonymous table belonging to the `widgets` module. As soon as you create a variable as part of `widgets` module name space the variable is automatically accessible by the other interpreters. For instance:

```
1 function apply()
2     widgets.unit = widgets.GetValue( "unit" )
3     widgets.eos = widgets.GetValue( "eos" )
4 end
```

19.3. ADD A TEMPLATE DIALOG

If we make the same modification to the `GetFunctionValue` the number format will change every time we select a different unit and click `Apply`.

```
1 function out()
2     ...
3     function GetFunctionValue( number, value )
4         if widgets.unit == "Metric" then
5             local formats = { "%.2fC", "%.2f%%", "%imBar" }
6             return string.format( formats[ number ], value )
7         else
8             ...
9     end
```

Since the user input is hold in a table, it is also easy to provide two simple functions to load or save the current dialog state (the settings made by the user) with a single function call. More in section 19.3.7.

19.3.4 Refresh or reload

Changing the system unit only effects the display of the telegrams. The telegrams considered as a certain sequence of data bytes do not change. All telegrams start and end at the same position in the data stream independent of the unit settings. This means:

The `apply` function only triggers a refresh of the visible telegrams in the telegram window. This is the default behaviour.

It's now time to attend to the remaining widget element in our dialog. The EOS selection.

The four items in the `Choice` widget stand for: LF, CR, LFCR and CRLF (which is used by the tutorial) and must be pass to the `split` function. But before we can use them, we must replace the selection text with the proper EOS sequence. In Lua LF is coded as `"\n"`, CR as `"\r"`. C(++) programmer knows them.

```
1 function apply()
2     local t = {
3         ["LF"] = "\n",
4         ["CR"] = "\r",
5         ["LFCR"] = "\n\r",
6         ["CRLF"] = "\r\n"
7     }
8     widgets.unit = widgets.GetValue( "unit" )
9     widgets.eos = t[widgets.GetValue( "eos" )]
10    return "RELOAD"
11 end
```

The local table `t` maps the choice selection into the right character sequence as shown in line 9.

Line 10 is new. So far we did not return any value which leads to the default behaviour, namely to refresh (redraw) the visible telegrams. Changing the EOS is different because it affects start and end of every telegram - and not only the visible ones. With other words: The `ProtocolView` must reload and parse again the whole data stream! Returning "RELOAD" exactly does what it says.

In a last step we must apply the selected EOS string to the `split` function - which is easy. Just replace the line:

```
1 if str:find( "\r\n") then return COMPLETED end
```

with:

KAPITEL 19. LUA PROTOCOL DIALOGS

```
1 if str:find( widgets.eos ) then return COMPLETED end
```

The script reacts now as expected. You can switch between the two unit systems and select different EOS strings. There is only one small downer: Whether we change only the units or the EOS too, the ProtocolView always reload the whole data. This can be very annoying if you are analysing real huge records of several GBytes and the reload takes some time.

A much better approach would be to limit the reload for cases where the underlying telegram data has changed. For instance a different EOS. And simply refresh the display when the user action only effects the telegram output.

In our example it would be nice to see the selected unit immediately when the user clicks the unit radio box. How we can achieve this is subject of the next section.

19.3.5 Defining element action handlers

The dialog framework offers an easy to understand mechanism to bind an action handler to a certain widget. An action handler is a function which is called every time a widget element is clicked, selected, modified or similar. It is also known as a 'callback' function.

They are particular useful when a user interaction demands an immediate reaction. Examples are the click of a button or the adaption of other elements depending on an user input. Or simply to update the representation of the visible telegrams which takes us back to our example.

How can we add an action handler or callback function to the radio box in our dialog? The dialog framework defines a callback as follows:

```
function callback_NAME( value )
    — do something
end
```

The important detail here is NAME. It reflects the name of the widget you want to have an action handler for¹. As soon as you have added a callback function for a given widget, it will be executed every time the user interacts with it.

In our example the necessary functions are named as `callback_unit`. See the listing below:

```
function callback_unit( state )
    debug.print( state )
    widgets.unit = state
end
```

The internal callback mechanism always passes the current widget state or selection as a Lua string. The string type was choose to cover all different widget inputs. Imagine a callback for a text input, or a list control.

Here it is more than convenient. We just have to assign the passed radio box state ('Metric' or 'Anglo-American') directly to the global `widgets.unit` variable and - voila - the shown unit in the telegram output changes with every click of the radio box²

The debug output is only for testing purpose and to give you an idea, how to check the parameter of the callback function.

Just remember, that this parameter is always a Lua string!

¹And another reason to choose the widget name carefully!

²The framework triggers a refresh of the display after every callback.

19.3. ADD A TEMPLATE DIALOG

Although Lua makes a lot of type conversion automatically, there is sometimes no alternative than to convert a string into a number by ourselves. In particular if Lua doesn't know (and cannot predict) which kind of variable we want it to act on. In such cases use the following string to number conversion:

```
x = tonumber( string )
```

19.3.6 Initialize dialog variables

So far we didn't waste a thought about the initial values of the dialog variables. But our script will remind us as soon as we clone the current ProtocolView. Just press **Ctrl**+**Shift**+**N**.

The new ProtocolView gives you an error `bad argument #1 to find...` in your script. What happens here?

Let us summarize how Lua interprets your script.

When you open a new ProtocolView or select a new template, the very first code which is executed is the `split` function because further action depends on a correct separation of the recorded data stream in single telegrams. In our example:

```
function split( data, intval, alter, str )
    if #str == 1 then return STARTED end
    if str:find( widgets.eos ) then return COMPLETED end
    return MODIFIED
end
```

In the `split` function we call `find` with the value of the `widgets.eos` variable to search for an equal EOS. The module `widgets` is accessible by the interpreter which runs the `split` code. But until you apply the dialog settings the anonymous table in the `widgets` module does not contain a variable with the name `eos`! This variable will be created first when execution the `apply` function.

As a result `str.find` is called with a `nil` value and produces consequently the given error message.

You can convince yourself by open the dialog and click 'Apply'. The ProtocolView now shows the telegrams as expected.

In order to prevent the Lua interpreter from accessing a not existing or invalid variable you must first declare and initialize it! In our case it's enough to assign an initial value to every dialog value in the global name space (outside of every function).

```
widgets.eos = "\r\n"
```

```
function split( data, intval, alter, str )
    if #str == 1 then return STARTED end
    if str:find( widgets.eos ) then return COMPLETED end
    return MODIFIED
end
```

With it the ProtocolView splits all telegram after it received an CRLF - at least until the user decides otherwise.

But a correct initialization does not only affects the splitting of the data stream. The `output` function too accesses a dialog variable, here the `widgets.unit`

KAPITEL 19. LUA PROTOCOL DIALOGS

which is `nil` as long as you do not initialize it.

Lua does not issue an error because comparing a `nil` value is allowed. In our case the line:

```
if widgets.unit == "Metric" then
```

returns false and therefore the telegram output shows Anglo-American units. So it is always a good practice to initiate all dialog values at the beginning of your template.

```
widgets.eos = "\r\n"
widgets.unit = "Metric"
```

Initialize all dialog variables

To avoid errors caused by not existing dialog variables declare and initialize them first at the beginning of your script!

You will observe, that your dialog always starts with the same settings independent of your last selection. This becomes understandable when you consider that there is no code which preset the dialog elements according your last input. When you click the `Setup` the ProtocolView starts a new Lua interpreter executing the `dialog` function. This interpreter exists only for the time you interact with your dialog and ends when you close it.

It is your responsibility as a programmer to preset the dialog elements (dialog settings). Luckily this is very easy as we will explain in the next section.

19.3.7 Dialog settings

With dialog settings we mean all changeable parameters the user can influence in the dialog. We already worked out how the user input has to apply to the protocol processing. Now we will focus on the other way round. To preset the dialog elements to their last state.

Even though the Lua interpreter running the dialog cease to exist when the dialog is closed, the state of the user input was transferred to the `widgets.eos` and `widgets.unit` before. We did so in the `apply` function and the unit callback `callback_unit`.

The first thing we have to do is to initialize all changeable elements in the dialog with these values in the `dialog` function.

```
1 function dialog()
2     widgets.Label{ text="Individual tutorial protocol settings",
3                   col=1, row=1, span=2 }
4     widgets.RadioButton{ name="unit",
5                          label="Unit",
6                          col=1, row=2, fill=true, span=2,
7                          choices={"Metric","Anglo-American"} }
8     widgets.Label{ text="Telegram EOS",
9                   col=1, row=3 }
10    widgets.Choice{ name="eos",
11                  col=2, row=3, fill=true,
12                  choices={"LF", "CR", "LFCR", "CRLF"} }
13    — initialize dialog elements with their last settings
14    widgets.SetValue( "unit", widgets.unit )
15    local t = {
16        ["\n"] = "LF",
17        ["\r"] = "CR",
```

19.3. ADD A TEMPLATE DIALOG

```
18         ["\n\r"] = "LF\r",
19         ["\r\n"] = "CRLF",
20     }
21     widgets.SetValue( "eos", t[widgets.eos] )
22 end
```

Please note! We must convert the current EOS sequence back to the items shown in the eos selector as we did the other way when assign the selection to `widgets.eos`.

If you save and try the script it nevertheless shows always the initialized values in the dialog which are for the unit 'Metric' and for the EOS the CRLF. This is not really a surprise because we assigned exactly these values when initializing the `widgets.eos` and `widgets.unit` variables at the beginning of our script.

When Lua runs the `dialog` function it also evaluates all other 'reachable' code in the script - amongst others the assignment of the `widgets` dialog values. This happens even if the variables already exist.

The second thing is therefore to prevent an initialization of the variables if Lua already knows them.

```
if not widgets.eos then
    widgets.eos = "\r\n"
end
if not widgets.unit then
    widgets.unit = "Metric"
end
```

With those changes the dialog now acts like expected. You can change the settings and as soon as you reopen the dialog it reflects the current template parameter.

19.3.8 Save dialog settings between sessions

You can instruct the dialog framework to store all `widgets` variables (containing the dialog settings) in a file so you can restore the settings when reopen the ProtocolView later. The file name is built from the current record name with the extension `msbini` and then stored in the same folder as your template.

All you have to do is to add the following line at the end of your `apply` and `callback_unit` function³

```
widgets.SaveSettings()
```

To restore the settings call the counterpart after you have initialized the `widgets` variables:

```
if not widgets.eos then
    widgets.eos = "\r\n"
end
if not widgets.unit then
    widgets.unit = "Metric"
end
widgets.LoadSettings()
```

³Just everywhere you apply the settings!

KAPITEL 19. LUA PROTOCOL DIALOGS

Please note the order! You should always initialize the widgets variables first before you call `widgets.LoadSettings()`. This makes sure, that the variables are valid even if the settings file does not exist⁴.

msbini file naming

You may wonder why the `msbini` file name is derived from the record file and not from the template name. The answer is simple: Consider for example two different tutorial records. The first one with CR as EOS, the second with LF as EOS. It is clear, that we need two independent `msbini` files, otherwise we must always correct our settings when we switch between the two record files.

19.4 More positioning and interaction

As mentioned before, all widgets in the dialog are organized by an invisible grid. You can pass the position (specified by the column and row) directly (as seen in the example code in the previous section) or as a result of a former computation. This also applies to all other widget parameters.

Imagine you want a 3x3 grid of buttons. A first approach will lead you probably to something like this:

```
1 function dialog ()
2     widgets.Button{ name="B1/1", label="B1/1", col=1, row=1 }
3     widgets.Button{ name="B2/1", label="B2/1", col=2, row=1 }
4     widgets.Button{ name="B3/1", label="B3/1", col=3, row=1 }
5     widgets.Button{ name="B1/2", label="B1/2", col=1, row=2 }
6     ...
7 end
```

But instead of writing nine lines of `widgets.Button{...}` code, you can achieve this more easier by creating the buttons in a loop.

```
1 function dialog ()
2     for row=1,3 do
3         for col=1,3 do
4             local s = "B"..col.."/"..row
5             widgets.Button{ name=s, label=s, row=row, col=col }
6         end
7     end
8 end
```

An interesting part of this little code snippet is line 4. Since Lua handles different kinds of variables like numbers or string almost automatically, it's very easy to build a string from different values. Here we are using the Lua string concatenation operator `..` to create a unique name `s` from the buttons column and row position. Every name starts with a uppercase 'B' (a string), the column (Lua converts the column number to a string for you), followed by the separator `'/'` and the row.

The resulting variable `s` is then assigned to the button name and label in line 5.

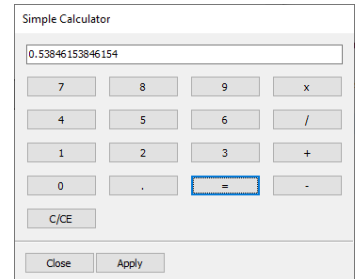
An example of how to use this technique to create nice looking dialogs is the `Tutorial-Calculator`. The template script contains a small but nevertheless full functional calculator. There is no further usage. It's only purpose is to demonstrate some advanced techniques with the Lua framework.

Here is the code for the dialog.

⁴Which is the case until the user applies your dialog.

19.4. MORE POSITIONING AND INTERACTION

```
1 function dialog()
2     widgets.TextCtrl{ name="display", col=1, row=1, span=4, fill=true,
3                       datatype=DEC_NUMBER }
4     local labels={ "7", "8", "9", "x",
5                   "4", "5", "6", "/",
6                   "1", "2", "3", "+",
7                   "0", ".", "=", "-" }
8     local i = 1
9     for y=2,5 do
10        for x=1,4 do
11            widgets.Button{ name=labels[i], label=labels[i], col=x, row=
12                           y }
13            i = i + 1
14        end
15    end
16    widgets.Button{ name="Clear", label="C/CE", col=1, row=6 }
end
```



Simple calculator

In the calculator example we combine a hard-coded position for the calculator display (line 2...3) and the 'Erase' key C/CE (line 15). The display itself is spanned over all four columns (`span=4`).

The number and operator keys are arranged in a 4x4 grid and will be created in a loop (line 9...14). Since the `TextCtrl` occupies the first row, we start with row 2 and iterate until row 5 was finished.

Each row has four columns and the inner loop in line 10 reflects this by counting from 1 to 4.

Line 4...7 specify the name and labels for the button which we assign in line 11. All in all, the whole user interface of the calculator needs just 16 lines of code.

19.4.1 Advanced callbacks

With one exception (the `TextCtrl`) the calculator GUI consists of only buttons. When the user clicks on one of the number buttons, the according digit should be inserted or attached to the value in the display field.

By clicking an arithmetic key the operation has to be stored and applied to the next input value when the user clicks the `=`.

At least the `C/CE`. If clicked all inputs should be cleared as usual.

We have learned so far, that we can use a callback function for every individual widget. This could be a proven method here too, but with 17 callbacks it would also be a rather unmanageable approach. Especially since some buttons, although different, require the same action. (For instance the digit buttons 0...9 simply have to add their number to the displayed value).

A single callback for all buttons would serve us a lot more. Every time the user clicks a button, a common button callback is invoked and performs an action depending on the pressed button.

The Lua framework acts exactly like this.

First it looks for an individual callback `callback_NAME` and performs the code there. Then it calls a function `callback_all_buttons`. If such a function exists, the code is executed too. The function body looks like

```
1 function callback_all_buttons( name )
2 end
```

KAPITEL 19. LUA PROTOCOL DIALOGS

and the parameter contains the name of the clicked button.

With a single function answering to the click of every button the reacting and computing code remains relative easy. Just open the `Tutorial-Calculator` in the editor and take look. The code is well documented.

19.5 Update existing widgets

So far we didn't mention how to change or update an already existing widget. We used the widgets function `widgets.SetValue(name)` to preset a given widget and we can surely use this function to update a value represented by a widget. But what, if you - for instance - want to modify the min/max value of a `SpinCtrl`?⁵

A `SetValue(name, value)` call will only affect the internal `SpinCtrl` value, but not the range. To make it a little bit clearer, here again the code how to add a `SpinCtrl` to a dialog:

```
widgets.SpinCtrl{ name="char", row=2, col=2, min=0, max=255, value=0 }
```

The control is intended to let the user input a character as a value between 0 and 255. But now consider that you must limit the input for printable characters only - forced by another widget interaction. In this case you must change the valid range from 30 (the space) to 127 (the last character in the normal ASCII table).

It would be rather complicated to provide a special change or update function for every kind of widget. A far better approach is to replace the control with simply the same type but with now different specifications. In our case the min and max parameters.

Since all dialog widgets are organised in a grid, this means nothing else than to overwrite the existing widget on a certain position with a new one:

```
1 local col, row = widgets.GetPosition( "char" );
2 widgets.SpinCtrl{ name="char", row=row, col=col,
3                 min=30, max=127, value=0 }
```

You can - of course - hard code the row and col values, but it makes your script only error prone. Better to query the position (grid column and row) as shown in line 1 with the widgets function `GetPosition(name)`. In line 2 we replace the widget on this position (the former `SpinCtrl` with the range 0...255) with a new `SpinCtrl` and the new range 30...127.

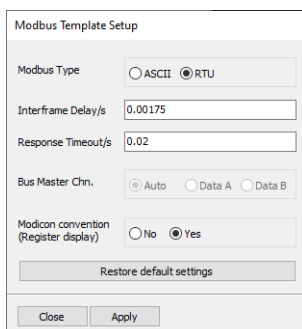
We can extend our little example and restore the actual value of the former widget if it is still in the new range by doing as following:

```
1 local col, row = widgets.GetPosition( "char" );
2 local val = widgets.GetValue( "char" )
3 widgets.SpinCtrl{ name="char", row=row, col=col,
4                 min=30, max=127, value=val }
```

19.6 Further examples

This tutorial showed you only a theoretical example. If you are interested in real protocol templates and what you can achieve with the dialog framework, take a look in the Modbus project. Or just select Modbus in the `ProtocolView` and click

⁵We used a `SpinCtrl` in our very first example explaining the dialog grid sizing mechanism, remember?



Modbus dialog

19.7. SUPPORTED DIALOG ELEMENTS OR WIDGETS

the `Setup` button.

Then open the template script with the editor. The code is well documented.

19.7 Supported Dialog elements or widgets

All the elements listed here are part of the analyzer software and will not work in a pure Lua environment.

Every widget supports the parameters listed below. Please note that although all parameters are optional in a sense of that you can omit them without producing an error, some are nevertheless mandatory for a correct operating of your code.

For a better reading we mark every parameter with the following symbols:

- ⊗ A mandatory parameter
- ⊙ An optional parameter

19.7.1 Named parameters

A few additional words regarding the parameter passing. You may have noticed that all widget element parameters 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 widgets.Button( "MyButton", "Press", 1, 3, true )
```

Without a look in the manual it's hard to get the meaning - isn't it? What is the widget name, what the button label, does 1 mean the row or the column and what is true?

On the other hand the same code with named parameters:

```
1 widgets.Button{ name="MyButton", label="Press", col=1, row=3, fill=true }
```

The meaning should be obvious.

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: `({ . . . })` but the outer brackets are optional here and you can forego them.

19.7.2 Common widget parameters

Every widget understands at least the following so called named parameters. Named parameter means: You pass a parameter by assign a parameter value to the parameter name like this:

```
name="MyName"  
col=1  
fill=true
```

KAPITEL 19. LUA PROTOCOL DIALOGS

The parameters are listed mandatory first.

- ⊗ **name** : Every widget needs an individual name with which you can later access the control, e.g. to query the input value or a selection.
- ⊗ **col** : Specifies the column index where the widget should be placed. The index starts from 1. Default is the first column.
- ⊗ **row** : Specifies the row number where the widget should be placed. The index starts from 1. Default is the first row.
- ⊙ **fill** : Set this parameter to true if you want to fill the whole available cell space with the widget element.
- ⊙ **span** : You can 'span' a widget over several columns by assign the number of columns to this parameter.

Beside the parameters above some widgets understand additional parameters, which we will explain in the according widget section.

19.7.3 Button

A simple button with a text label.

```
widgets.Button{ name=STRING, label=STRING, col=NUM, row=NUM,
                fill=BOOL, span=NUM }
```

- ⊗ **name** : the button name as a Lua string.
- ⊗ **col** : the column index as an integer
- ⊗ **row** : the row index as an integer
- ⊙ **label** : the button label
- ⊙ **fill** : fill the cell completely, true or false
- ⊙ **span** : the number of spanned columns as an integer

Example

```
1 function dialog()
2     — a button on top left2
3     widgets.Button{ name="MyButton", label="Press", col=1, row=1 }
4 end

5 — this callback is executed every time the button is clicked
6 function callback_MyButton()
7     — do something...
8 end
```

19.7. SUPPORTED DIALOG ELEMENTS OR WIDGETS

19.7.4 CheckBox

A checkbox is a labeled box which can be either true (checkmark is visible) or false (checkmark is absence).

```
widgets.CheckBox{ name=STRING, label=STRING, col=NUM, row=NUM,
                 fill=BOOL, span=NUM }
```

- ⊗ **name** : the checkbox name as a Lua string.
- ⊗ **col** : the column index as an integer
- ⊗ **row** : the row index as an integer
- **label** : an optional label shown on the right side of the checkbox
- **fill** : fill the grid cell completely, true or false
- **span** : the number of spanned columns as an integer
- **value** : the initiate (checked) state of the checkbox passed, true or false

Example

```
1 function dialog()
2     — a button we want to stretch or shrink
3     widgets.Button{ name="MyButton", label="Press", col=1, row=1 }
4     — a checkbox to toggle a fill parameter
5     widgets.CheckBox{ name="MyCheckBox", label="Stretch the button",
6         col=1, row=2, value=false }
7 end

8 — this callback is executed every time the checkbox is clicked
9 function callback_MyCheckBox( selection )
10    — query the position of the button widget
11    row,col = widgets.GetPosition( "MyButton" )
12    — recreate the button with the new fill parameter
13    widgets.Button{ name="MyButton", label="Press",
14        col=col, row=row, fill=(selection=="true") }
15 end
```

19.7.5 Choice

A choice widget let you select one of a list of strings. Only the current selection is displayed. The list of available strings is only shown when the user pull downs the menu of choices.

```
widgets.Choices{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
                choices={STRING1, STRING2 [ ,... ]}}
```

- ⊗ **name** : the choice name as a Lua string.
- ⊗ **col** : the column index as an integer
- ⊗ **row** : the row index as an integer
- ⊗ **choices** : a Lua table with at least one string
- **fill** : fill the grid cell completely, true or false
- **span** : the number of spanned columns as an integer
- **value** : the default selection for the choice passed as string

KAPITEL 19. LUA PROTOCOL DIALOGS

Example

```
1 function dialog()
2   — a button we want to stretch or shrink
3   widgets.Button{ name="MyButton", label="Press", col=1, row=1 }
4   — a checkbox to toggle a fill parameter
5   widgets.Choice{ name="MyChoice", col=1, row=2,
6                   choices={ "Shrink button", "Stretch button"},
7                   value = "Stretch button" }
8 end

9 — this callback is executed every time a choice is made
10 function callback_MyChoice( selection )
11   — query the position of the button widget
12   row,col = widgets.GetPosition( "MyButton" )
13   — recreate the button with the new fill parameter
14   widgets.Button{ name="MyButton", label="Press",
15                  col=col, row=row,
16                  fill=(selection=="Stretch button") }
17 end
```

19.7.6 Label

The label widget is used to show any static (not clickable) text. For instance an explaining label for another widget.

```
widgets.Label{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
               text=STRING}
```

- ⊗ **name** : the choice name as a Lua string.
- ⊗ **col** : the column index as an integer
- ⊗ **row** : the row index as an integer
- ⊗ **text** : a Lua string uses as the label
- ⊙ **fill** : fill the grid cell completely, true or false
- ⊙ **span** : the number of spanned columns as an integer

Example

```
1 function dialog()
2   — a explaining text for the button
3   widgets.Label{ name="MyLabel", text="You can click me",
4                 col=1, row=1 }
5   — the button
6   widgets.Button{ name="MyButton", label="Disable me",
7                  col=1, row=2 }
8 end

9 — this callback is executed every time the button is clicked
10 function callback_MyButton()
11   — disable the button
12   widgets.Enable( "MyButton", false )
13   — and adapt the label text
```

19.7. SUPPORTED DIALOG ELEMENTS OR WIDGETS

```
14     local col, row = widgets.GetPosition( "MyLabel" )
15     widgets.Label{ name="MyLabel", text="You cannot click me anymore",
16                   col=col, row=row }
17 end
```

19.7.7 Line

This is just a line which is commonly used to divide several groups of controls. A line always fills the complete space of a grid cell. With the span parameter you can stretch the line over a number of columns.

```
widgets.Line{ name=STRING, col=NUM, row=NUM, span=NUM }
```

- ⊗ **col** : the column index as an integer
- ⊗ **row** : the row index as an integer
- ⊙ **span** : the number of spanned columns as an integer
- ⊙ **name** : the line name as a Lua string. Can be omitted if you don't want to access the line later.

Example

```
1 function dialog()
2     widgets.Label{ name="MyLabel", text="A label", col=1, row=1 }
3     widgets.Button{ name="Button1", label="Press me", col=2, row=1 }
4     widgets.Button{ name="Button2", label="Or me", col=3, row=1 }
5     — draw a line over all columns (span=3)
6     widgets.Line{ span=3, col=1, row=2 }
7 end
```

19.7.8 RadioBox

A radio box is used to select one of a number of mutually exclusive choices. It is displayed as a vertical column or horizontal row of labeled and clickable boxes.

```
widgets.RadioBox{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
                  choices={STRING1, STRING2 [ ,... ]}}
```

- ⊗ **name** : the radio box name as a Lua string.
- ⊗ **col** : the column index as an integer
- ⊗ **row** : the row index as an integer
- ⊗ **choices** : a Lua table with a string for each choice (at least one)
- ⊙ **fill** : fill the grid cell completely, true or false
- ⊙ **span** : the number of spanned columns as an integer
- ⊙ **orientation** : The RadioBox orientation can be 'vertical' (default) or 'horizontal'.
- ⊙ **value** : the default selection for the RadioBox passed as Lua string

Example

KAPITEL 19. LUA PROTOCOL DIALOGS

```
1 function dialog()
2     widgets.RadioButton{ name="MyRadioBox", col=1, row=1,
3                           choices={ "vertical", "horizontal" },
4                           value="horizontal" }
5 end

6 — each click changes the orientation of MyRadioBox
7 function callback_MyRadioBox( selection )
8     widgets.RadioButton{ name="MyRadioBox", col=1, row=1,
9                           choices={ "vertical", "horizontal" },
10                          orientation=selection,
11                          value=selection }
12 end
```

19.7.9 Spacer

A spacer is just an empty widget. It is especially used when you want to remove a certain widget or - simply spoken - overwrite an existing widget with 'nothing'.

```
widgets.Spacer{ name=STRING, col=NUM, row=NUM, span=NUM }
```

- ⊗ **name** : the spacer name as a Lua string.
- ⊗ **col** : the column index as an integer
- ⊗ **row** : the row index as an integer
- ⊙ **span** : the number of spanned columns as an integer

Example

```
1 function dialog()
2     widgets.Button{ name="MyButton", col=1, row=1,
3                     label="Click me", fill=true }
4     widgets.RadioButton{ name="MyRadioBox", col=1, row=2,
5                           choices={ "Show button", "Hide button" } }
6 end

7 — Show or hide the button according to the RadioBox selection
8 function callback_MyRadioBox( selection )
9     if selection == "Show button" then
10        widgets.Button{ name="MyButton", col=1, row=1,
11                        label="Click me", fill=true }
12     else
13        widgets.Spacer{ col=1, row=1 }
14     end
15 end
```

19.7.10 SpinCtrl

A SpinCtrl is a combined number input field with two increment and decrement buttons. This widget is used to adjust a integer value between a minimum and maximum value either by input the value directly (it will corrected automatically if the given limit is exceeded) or by increasing or decreasing it with the buttons.

```
widgets.SpinCtrl{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
                  min=NUM, max=NUM, value=NUM }
```

19.7. SUPPORTED DIALOG ELEMENTS OR WIDGETS

- ⊗ **name** : the SpinCtrl name as a Lua string.
- ⊗ **col** : the column index as an integer
- ⊗ **row** : the row index as an integer
- ⊙ **fill** : fill the grid cell completely, true or false
- ⊙ **min** : the minimum value, default is 1
- ⊙ **max** : the maximum value, default is 100
- ⊙ **value** : the initial value, default is the minimum value
- ⊙ **span** : the number of spanned columns as an integer

Example

```
1 function dialog()
2     widgets.Label{ name="MyLabel", text="Valid numbers are 1...100",
3                   col=1, row=1 }
4     widgets.SpinCtrl{ name="MySpinCtrl", min=1, max=100,
5                      col=1, row=2, value=10 }
6 end

7 — always called when the value in the SpinCtrl was changed
8 function callback_MySpinCtrl( value )
9     local num = tonumber( value )
10    if num >= 100 then
11        widgets.Label{ name="MyLabel", text="Maximum number reached",
12                      col=1, row=1 }
13    elseif num <= 1 then
14        widgets.Label{ name="MyLabel", text="Minimum number reached",
15                      col=1, row=1 }
16    else
17        widgets.Label{ name="MyLabel", text="Valid numbers are
18                      1...100",
19                      col=1, row=1 }
20    end
end
```

19.7.11 Table

The Table widget is as its name revealed, an text input control organized as a table. This means: You can create a table with two columns and four rows. Every table cell serves as an text input for various strings or numbers. You can even preset every table cell or pass a Lua array (table) to it.

A table is particular interesting for field bus systems where the participants structure their values in register tables like Modbus.

```
widgets.Table{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
               cols=NUM, rows=NUM, preset=STRING,
               choices={STRING1, STRING2 [,...]} }
```

- ⊗ **name** : the Table name as a Lua string.
- ⊗ **col** : the column index as an integer
- ⊗ **row** : the row index as an integer
- ⊙ **fill** : fill the grid cell completely, true or false
- ⊙ **span** : the number of spanned columns as an integer

KAPITEL 19. LUA PROTOCOL DIALOGS

- ⦿ **cols** : the number of columns, default is 1 is 1
- ⦿ **rows** : the number of rows, default is 1
- ⦿ **preset** : the initial value for every table cell, default is an empty string.
- ⦿ **content** : a Lua array or table which contains a certain number of strings or numbers. The assignment starts with the first item in the passed content and stops either when reaching the last Table cell or last content value.

Example

```
1 function dialog()
2   — the number of columns
3   local tcols = 3
4   — the number of rows
5   local trows = 20
6   — an empty table holding the default values
7   local tvalues = {}
8
9   — fill the table with incrementing numbers starting with 1
10  for i=1,tcols*trows do
11    tvalues[i] = i
12  end
13
14  — special mark of the first and last table entry
15  tvalues[ 1 ] = "FIRST"
16  tvalues[ #tvalues ] = "LAST"
17  — create a table widget and pass the tvalues table as initial
18  values
19  — to initiate all table cells
20  widgets.Table{ name="table", col=1, row=1, cols=tcols, rows=trows,
                preset="FFFF", content=tvalues }
end
```

19.7.12 TextCtrl

A TextCtrl comes in handy every time you need an input field for numbers or text strings. The TextCtrl furthermore filters the key strokes according to its input type. You can create a TextCtrl for plain decimal, numbers with a decimal point, hexadecimal or binary values and - of course - for normal strings.

```
widgets.TextCtrl{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
                 datatype=TYPE, datalen=NUM, value=STRING }
```

- ⊗ **name** : the Table name as a Lua string.
- ⊗ **col** : the column index as an integer
- ⊗ **row** : the row index as an integer
- ⦿ **fill** : fill the grid cell completely, true or false
- ⦿ **span** : the number of spanned columns as an integer
- ⦿ **datatype** : specifies the kind of input data. TextCtrl offers you the number types BIN_NUMBER (binary), DEC_NUMBER (decimal), FLOAT_NUMBER (floating point numbers to input numbers with a decimal point) and HEX_NUMBER (hexadecimal). Additional ASCII_STRING (normal text) and HEX_STRING (which is a sequence of hex characters). Depending on the passed datatype the input filter is set. Default is ASCII_STRING.

19.8. FUNCTIONS DEALING WITH WIDGET ELEMENTS

- ⦿ **datalen** : Specifies the valid length of the input data. For instance: How many characters of an input should be used during the value request.
- ⦿ **value** : the initial value, default is an empty string

Example

```
1 function dialog()
2     widgets.RadioButton{ name="MyRadioBox", col=1, row=1, label="Mode",
3                         orientation="horizontal",
4                         choices={ "ASCII", "HEX", "DEC", "BIN" } }
5     widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
6                      datatype=ASCII_STRING, fill=true }
7 end
8
9 function callback_MyRadioBox( mode)
10    local filter = ASCII_STRING
11    if mode== "DEC" then filter = DEC_NUMBER
12    elseif mode== "HEX" then filter = HEX_NUMBER
13    elseif mode== "BIN" then filter = BIN_NUMBER
14    end
15    widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
16                    datatype=filter , fill=true }
17 end
```

19.8 Functions dealing with widget elements

You have already seen some of these functions in the examples when we were querying an input from a widget or modifying its value.

The following functions are provided by the widgets module. All these functions expect a unique widget name.

Please note! Since the functions need not more than two parameters, the arguments are passed directly and not as a NAME=VALUE pair. The only exception is the `SetDialogSize` function.

A function with named parameters (NAME=VALUE pairs) expects the arguments between two `{}`. Here the arguments are enclosed between two normal round brackets `()`.

19.8.1 Clear

A call of `widgets.Clear()` removes (deletes) all existing widgets in the dialog. This function becomes extremely useful if you want to realize a dialog with different 'pages'. Since the current dialog implementation doesn't provide 'nested' dialog elements (assigned to different dialog pages), you must remove all the widgets of a not visible dialog page before adding the new ones - which could be very cumbersome. The `Clear()` function provides you with a new 'clear' dialog where you can add the necessary elements of a given dialog page. When switching to another page, just do the same. First clear the dialog, then add the elements for the now visible page.

The Modbus template is a good example. It is divided into a dialog page for the general Modbus protocol setup and a second page providing typical Modbus telegram filter elements.

KAPITEL 19. LUA PROTOCOL DIALOGS

The following (smaller) code example will give you a first impression how it works.

```
1  if not widgets.DIALOG_PAGE then
2      widgets.DIALOG_PAGE = " Filter "
3  end
4
5  function dialog()
6      callback_page( widgets.DIALOG_PAGE )
7  end
8
9  function dialog_filter()
10     widgets.Clear()
11     widgets.RadioButton{ name="page", choices={"Setup"," Filter "},
12                          row=1, col=1, fill=true, span=2,
13                          orientation="horizontal", value="Filter" }
14     widgets.Label{ text="This is the FILTER page", row=2, col=1,
15                   span=2, fill=true }
16     widgets.Label{ text="Device Address", row=3, col=1, fill=true }
17     widgets.SpinCtrl{ name="wxFilterAddr", row=3, col=2, fill=true,
18                      min=1,max=16 }
19 end
20
21 function dialog_setup()
22     widgets.Clear()
23     widgets.RadioButton{ name="page", choices={"Setup"," Filter "},
24                          row=1, col=1, fill=true,
25                          orientation="horizontal", value="Setup" }
26     widgets.Label{ text="This is the SETUP page", row=2, col=1,
27                   fill=true }
28     widgets.CheckBox{ name="test", label="Test mode",
29                      row=3, col=1, fill=true }
30 end
31
32 function callback_page( page )
33     if page == "Setup" then
34         dialog_setup()
35     else
36         dialog_filter()
37     end
38     — store selected dialog page
39     widgets.DIALOG_PAGE = page
40     widgets.SaveSettings()
41 end
42
43 function apply()
44     local page = widgets.GetValue( "page" )
45     if page == "Setup" then
46         — do apply only setup settings
47     else
48         — do apply only filter settings
49     end
50 end
```

Line 1..3 defines a non volatile variable to store and restore the last selected dialog page, see 19.8.8. The dialog itself consists of two different pages. Each one is coded in its own function `dialog_filter()` and `dialog_setup()`. The selection is done by a `RadioButton` provided in every page (line 11 and 23). The original `dialog` calls the right code depending on `widgets.DIALOG_PAGE` in line 6. The same happens when the user clicks the `RadioButton` via the callback `callback_page`.

19.8. FUNCTIONS DEALING WITH WIDGET ELEMENTS

Every dialog page starts with a call `widgets.Clear()`. Remember, that you cannot delete or remove a certain element. You can replace an element with a `Spacer` widget, but this is only a poor solution (if at all). Especially when the dialog pages have different rows and/or columns. Without clearing the page, every call of `dialog_filter` or `dialog_setup` will just overwrite widgets at the given column and row index. If the former dialog has more columns or rows the remaining elements remain in place.

Since `Clear()` removes all elements you have to add the page selector (here the `RadioBox`) in every page separately. You can - of course - use other elements for the page selection. For instance a `Choice` or a row of buttons. It is up to you to decide.

One word to the `apply` mechanism. Since `Clear()` removes all elements of the 'other' dialog page, this also means, that you only can apply values of the current - existing - dialog widgets. Or to put it in other words:

`widgets.GetValue(...)` returns `nil` if you access an not existing widget element. A good approach is to ask for the current page (see line 44 and 45) before query and apply the user input.

19.8.2 Enable

This function enables a widget for user interaction (which is the default state of a widget element) or disables it. A disabled widget appears greyed out and is not accessible by the user.

```
widgets.Enable( NAME, STATUS )
```

- ⊗ **NAME** : The name of the widget as a Lua string.
- ⊗ **STATUS** : The new enable state of the widget, true or false

Example

```
1 function dialog()
2     widgets.RadioBox{ name="MyRadioBox", col=1, row=1, label="Mode",
3                       orientation="horizontal",
4                       choices={ "ENABLED", "DISABLED" } }
5     widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
6                       datatype=ASCII_STRING, fill=true }
7 end
8
9 function callback_MyRadioBox( mode)
10    local enable = true
11    if mode== "DISABLED" then enable = false
12    else enable = true
13    end
14    widgets.Enable( "MyTextInput", enable )
15 end
```

19.8.3 GetPosition

Asks a widget with the given name for its position in the grid. This function is especially useful if you favorite a dynamic column and row assignment.

KAPITEL 19. LUA PROTOCOL DIALOGS

POSITION = widgets.GetPosition(NAME)

- ⊗ **NAME** : The name of the widget as a Lua string.
- = **POSITION** : The position as a value pair column, row.

Example

```
1 function dialog()
2     for row=1,4 do
3         for col=1,4 do
4             local name = "B"..col.."x"..row
5             if col == 3 and row == 2 then name="PRESS" end
6             widgets.Button{ name=name, label=name, col=col, row=row }
7         end
8     end
9 end

10 function callback_PRESS( control)
11     local col,row = widgets.GetPosition( "PRESS" )
12     widgets.Label{ name="PRESS", text="Ready", col=col, row=row }
13 end
```

Note the returning of two variables. This is a special feature of Lua.

19.8.4 GetValue

Queries the value of the given widget. The type of the result depends on the asked widget. It can be a number (SpinCtrl), a boolean (CheckBox), a string (TextCtrl, Choice, RadioBox) or an array of strings (Table).

VALUE = widgets.GetValue(NAME)

- ⊗ **STRING** : The name of the widget.
- = **VALUE** : The value of the widget. The result type depends on the widget.

Example

```
1 function dialog()
2     widgets.Label{ name="MyLabel", text="Input a hex number", col=1,
3         row=1 }
4     widgets.TextCtrl{ name="MyInput", col=2, row=1, datatype=HEX_NUMBER
5         }
6 end

7 function apply()
8     — pass the input to the send mechanism
9     return widgets.GetValue( "MyInput" )
10 end
```

19.8. FUNCTIONS DEALING WITH WIDGET ELEMENTS

19.8.5 IsEnabled

Checks if the given widget is enabled for user inputs or disabled.

```
RESULT = widgets.IsEnabled( NAME )
```

- ⊗ **NAME** : The name of the widget.
- = **RESULT** : Returns true when the widget is enabled, false otherwise.

Example

```
1 function dialog()
2     widgets.Button{ name="MyButton", label="Toggle inout field", col=1,
3         row=1 }
4     widgets.TextCtrl{ name="MyInput", col=1, row=2, fill=true}
5 end
6
7 function callback_MyButton()
8     widgets.Enable( "MyInput", widgets.IsEnabled( "MyInput" ) == false
9     )
10 end
```

19.8.6 SetValue

Sets the internal value of the given widget.

```
widgets.SetValue( NAME, VALUE )
```

- ⊗ **NAME** : The name of the widget.
- ⊗ **VALUE** : The new value displayed by the given widget.

Example

```
1 function dialog()
2     widgets.Button{ name="MyButton", label="Default value", col=1, row
3         =1 }
4     widgets.SpinCtrl{ name="MySpinCtrl", col=1, row=2, fill=true}
5 end
6
7 function callback_MyButton()
8     widgets.SetValue( "MySpinCtrl", 50 )
9 end
```

19.8.7 SetDialogSize

In some circumstances it may be necessary to set the dimension of the dialog explicitly. This function let you specify the width and height of the dialog independent of internal grid mechanism.

```
widgets.SetDialogSize{ width=400, height=500 }
```

- ⊗ **width** : The new width of the dialog in pixel.

KAPITEL 19. LUA PROTOCOL DIALOGS

⊗ **height** : The new height of the dialog in pixel.

Example

```
1 function dialog ()
2     widgets.SetDialogSize{ width="600", height="400" }
3     widgets.RadioBox{ name="MyRadioBox", col=1, row=1, label="Mode",
4         orientation="horizontal",
5         choices={ "ENABLED", "DISABLED" } }
6     widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
7         datatype=ASCII_STRING, fill=true }
8 end
```

19.8.8 SetTitle

If you like to give the dialog an individual title in the window frame, this function comes into play.

```
widgets.SetTitle( title )
```

⊗ **title** : The title of your dialog as a string.

Example

```
1 function dialog ()
2     widgets.SetTitle(" Tutorial Protocol Settings")
3     widgets.RadioBox{ name="unit",
4         label="Unit",
5         col=1, row=2, fill=true, span=2,
6         choices={" Metric ", "Anglo-American" } }
7     widgets.Label{ text="Telegram EOS",
8         col=1, row=3 }
9     widgets.Choice{ name="eos",
10        col=2, row=3, fill=true,
11        choices={"LF", "CR", "LFCR", "CRLF" } }
12 end
```

LoadSettings

The counterpart of `SaveSettings()`. This function restores the previously saved dialog variables (settings) of the `widgets` module (name space).

```
RESULT = widgets.LoadSettings ()
```

= **RESULT** : Returns true when the settings were loaded without failure, false otherwise.

Example

19.8. FUNCTIONS DEALING WITH WIDGET ELEMENTS

```
1 if not widgets.LoadSettings() then
2     debug.print("OOPS")
3 end
4 ...
5 function split(...)
6 end
```

SaveSettings

Saves all variables of the `widgets` name space in a file. The file name is derived from the current record name. The file extension is `msbini` and created in the `templates` folder. The variables are stored as key value pairs like:

```
eos="%013%010"
unit="Metric"
timeout=0.01
```

Control characters or characters outside the normal ASCII range are 'escaped' by the `%` character followed by its 3-digit ASCII value. In the example above `%013` means the CR, `%010` means the LF. The `%` itself is stored as `%037`.

```
RESULT = widgets.LoadSettings()
```

RESULT : Returns true when the settings were saved without failure, false otherwise.

Example

```
1 function apply()
2     local t = {
3         ["LF"] = "\n",
4         ["CR"] = "\r",
5         ["LF CR"] = "\n\r",
6         ["CRLF"] = "\r\n"
7     }
8     widgets.unit = widgets.GetValue( "unit" )
9     widgets.eos = t[widgets.GetValue( "eos" )]
10    if not widgets.SaveSettings() then
11        debug.print( "OOPS" )
12    end
13    return "RELOAD"
14 end
```


20

Lua modules

Lua modules are like libraries in other languages. You already know some modules like `math`, `string` or `widgets`. The first two are fixed part of the original Lua interpreter. The latter was added as a fixed built-in by the MSB-RS485 software. Here you learn how you can write your own modules which you afterwards can reuse and share between your scripts.

As a rule a module is a collection of functions which are serving a common purpose. For example the `widgets` module contains all necessary functions for building a graphical user interface.

In Lua these functions are stored in a table. Calling a module function is nothing else like accessing a table element/variable. The table name provides the module name, making the functions in it distinguishable from functions coincidentally with the same name¹.

Let's take a look to the following code snippet:

```
1 function Sleep( t )
2   — will not executed!
3 end
4 time.Sleep( 0.5 )
```

When executing this line Lua looks for a function `Sleep` in the already loaded table `time`. It does not call the global `Sleep` function in line 1.

But hold on a minute! What means 'already loaded table' in this context?

In case of built-in modules like `string` or `widgets` the according module tables are already loaded by the interpreter so to offer their functions without any additional code. Individual modules (written by yourself) cannot pre-loaded since the interpreter does know nothing about them. It is your duty to do that. The Lua command or function to load a module is:

```
require "modname"
```

In simple terms `require` looks in specified paths² for a Lua file with the given name and executes the code. The result is cached so if you require the same module again you get the cached result/instance from the first time.

A module code can - of course - solely exists of a single instruction like:

¹In this case it works like a name space in other programming languages, e.g. C++

²Where Lua looks for modules is part of a later section

KAPITEL 20. LUA MODULES

```
— minimal.lua
debug.print( "I'm a module" )
```

In this case you get the debug message once you require the given module. But only once, even if you repeat the require statement. Since the module is already loaded Lua does not reload and therefore does not execute it again!

```
require "minimal"  —> outputs "I'm a module"
require "minimal"  —> nothing!
```

Such modules with directly executable code are suitable to do some initialisation or to provide your code with special program constants like field-bus protocol specifiers, name aliases and so on. But they represent not really a module with the meaning of a collection of functions as mentioned before.

20.1 Writing a module

In a module a collection of functions is organized in a table. Imagine a small module providing just the two functions `min` and `max`. Both functions are called with two number parameters and return either the minimum or maximum value. We will put these functions in a module called `algorithm`. Here at first the module code:

```
1 — algorithm.lua
2 return {
3     min = function(a,b) if a > b then return b else return a end end,
4     max = function(a,b) if a < b then return b else return a end end
5 }
```

Remember, Lua does not differ between numbers, strings or functions. These are all first-class values³ and you can store a function as a table item as easily as any other type. Here we simply assign the table values or entries `min` and `max` with the according functions.

Loading the module with `require` returns a table which Lua stores in its own module table. But to access the table (or module) functions you need a reference for it. As said before `require` returns the result of the loaded module code, here the module table which you just can bind to any variable.

```
local algorithm = require "algorithm"
```

Afterwards you can access every function in the module as easily as any other table value. To get the min or max value of two number pairs is then performed with:

```
local algorithm = require "algorithm"
debug.print( algorithm.min(1,2) ) —> outputs 1
debug.print( algorithm.max(1,2) ) —> outputs 2
```

Note! We named the module reference as like the module name. This is common practice but you can decide for yourself.

Coding a module like above (directly inserted in the table) may work for very short function codes but it is surely not a good advice for more complicated functions. A better approach is to define an empty table and assign the function later. Here we go:

³First-class value in Lua means a function is a value with the same rights as strings or numbers and therefore can be stored in variables and tables equally.

20.1. WRITING A MODULE

```
1 local m = {}
2
3 function m.min( a, b )
4     if a > b then return b else return a end
5 end
6
7 function m.max( a, b )
8     if a > b then return a else return b end
9 end
10
11 return m
```

Line 1 creates an empty (module) table. We can put every function (and also every variable we like) into the table just by add the table name in front of the function name separated by a dot.

Alternatively you can write:

```
1 local m = {}
2
3 m.min = function( a, b )
4     if a > b then return b else return a end
5 end
6
7 m.max = function( a, b )
8     if a > b then return a else return b end
9 end
10
11 return m
```

But the first approach looks more elegant.

Functions or variables which are not part of the module table are not accessible from the outside and behaves like private members in a C++ class. To make that clear let us extend the `algorithm` module with a function to calculate the factorial of a given number.

```
1 function m.fact( n )
2     if n < 0 then return nil
3     elseif n == 1 then return 1
4     else return n * m.fact( n - 1 )
5     end
6 end
```

This is a recursive function and calls itself n times (see line 4). And how it is with recursive functions you always risk a stack overflow. Here it won't happen. Lua reaches the most possible floating point number before the stack overruns with $n=170$ and returns infinity (inf).

But let us assume our stack size is limited. In this case we must prevent `fact` to exceed a given number of recursive calls. We do so by specifying an internal (private) module variable `MAX_STACK=100`.

```
1 local MAX_STACK = 100
2
3 function m.fact( n )
4     if n < 0 or n >= MAX_STACK then return nil
5     elseif n == 1 then return 1
6     else return n * m.fact( n - 1 )
7     end
8 end
```

Finally we add a function to control this limit from the outside of the module.

KAPITEL 20. LUA MODULES

```
1 function m.set_stack_limit( n )
2     if n > 2 and n <= 100 then
3         MAX_STACK = n
4     end
5 end
6
7 function m.fact( n )
8     if n < 0 or n >= MAX_STACK then return nil
9     elseif n == 1 then return 1
10    else return n * m.fact( n - 1 )
11    end
12 end
```

The function `set_stack_limit` is the only way to control the stack limit outside the module. And since it also does a range check, we are sure that the limit is always in a valid range.

Here we hide a normal variable. But it is always a good idea to do the same with functions which are only called within the module. You will find the complete `algorithm` module in the `modules` folder.

20.2 Module path

This is the path where MSB-RS485 looks for modules loaded with `require`. Under Linux it is:

```
~/IFTTOOLS/SerialAnalyzer/7.0.2/Templates/modules
```

Under Windows the module path is:

```
C:\Users\USERNAME\AppData\Roaming\IFTTOOLS\SerialAnalyzer\7.0.2\Templates\modules
```

You can check it by yourself. Just use an invalid module name in the `require` command like:

```
algorithm = require "algoorithm"
```

and Lua shows you an according error in the editor error window:

```
[string "--[...]:8: module 'algoorithm' not found:
no field package.preload['algoorithm']
no file '~/IFTTOOLS/SerialAnalyzer/6.0.2/Templates/Modules/algoorithm.lua'
```

The last line indicates the place where Lua looks for a module.

You can also organize your modules in different folders of the module path. For instance: You have a module collection serving only for a specific project or you consider to make a new version and want to have both for a certain time. In all these cases just add the sub folders to the `require` argument. Here an example:

You have two versions of the `algorithm` module, version 1.0 and 1.1. In the module folder they are stored as `modules/1.0/algorithm.lua` and `modules/1.1/algorithm.lua`. The load command then looks like:

```
algorithm = require "1.0/algorithm"
```

You can even use a variable to switch between the two versions with:

20.2. MODULE PATH

```
local modpath = "1.0/algorithm"  
algorithm = require( modpath )
```

Please note!

To pass the module path as a variable you have to put it in brackets to make clear, that it is an argument. And: Lua converts the module path internally to its OS. It doesn't matter if you run Windows or Linux, always prefer a '/' in the path. This is not only platform independent it also looks better than:

```
— in a string a backslash must be input as double backslashes  
local modpath = "1.0\\algorithm"  
algorithm = require( modpath )
```


21

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.

21.1 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-

KAPITEL 21. SYNCHRONIZE TWO ANALYZERS

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 devices¹. 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.

21.2 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).

¹Tested was a CAT6 network cable with 100m length.

21.3. ESTABLISH A SYNCHRONOUS RECORD

21.3 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.

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.

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



Choose a storage place
for the automatically
stored records



Master and Slave
Specify the record master

KAPITEL 21. SYNCHRONIZE TWO ANALYZERS

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 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.

21.4 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 dialogs².

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

²In the end a running MSB analyzer program application corresponds to ONE recording. This is the same as for Audio or Video applications.

21.5. SYNCHRONIZE MORE THAN TWO ANALYZERS

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'.

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.

21.5 Synchronize more than two analyzers

In rare cases synchronizing two analyzers is not enough. What, if you have to make a synchronous record with three or more devices?

As described above: The idea behind the synchronization is, that the master adapts the clock of the linked slave to its own by sending a special pulse/signal over the MSB Link connection.

If we split this signal into two, the master is able to synchronize two slaves. And we every additional split we can synchronize another one.

IFTOOLS offers such a split device as MSB-Link-Port-Doubler. Inserting the doubler cable into the MSB-Link port of the master provides you with now two link sockets where you can connect two slave devices or another doubler in case you need more than three synchronized devices.

You will find the doubler and the detailed description as an optional analyzer accessory in our web shop.

21.6 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.



Ext. Synchronisation
is enabled in the record settings



Synchronous display
of all Views in both records



MSB-Link-Doubler
doubles sync pulse

KAPITEL 21. SYNCHRONIZE TWO ANALYZERS

21.6.1 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.

21.6.2 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.

22

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 be saved as several sequenced files or splitted afterwards.

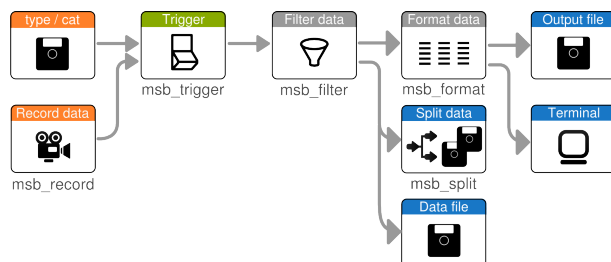
You like to control the analyser from within your application.

The MSB-Analyser 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.

KAPITEL 22. COMMANDLINE API

- **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.

22.1 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
```

22.1.1 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
```

22.1.2 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`.

22.1.3 Data sink

A data sink specifies the end of the processing chain. A typical 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 several files on your hard disk.

22.1.4 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

22.2. RECORD DATA WITH MSB_RECORD

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.

22.2 Record data with `msb_record`

As the name indicated this program controls the operating and recording of a connected MSB-Analyser. 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, transfers 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' "
```

¹You 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.

KAPITEL 22. COMMANDLINE API

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
```

22.2.1 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 be 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.

22.2.2 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, Lab-View offers a System Exec VI. You can call an external command mostly in two ways:

22.2. RECORD DATA WITH MSB_RECORD

First: The caller (your application) waits until completion of the command. We don't recommend 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`.

22.2.3 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.

22.2.4 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 [21](#).

KAPITEL 22. COMMANDLINE API

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 slave².
- 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.

²Remember that the master and slave must not run on the same computer.

22.2. RECORD DATA WITH MSB_RECORD

22.2.5 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 well³.

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

³timeout 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

KAPITEL 22. COMMANDLINE API

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.

22.2.6 msb_record program parameters

Call the program with:

```
msb_record [OPTION]...
```

[OPTION] can contain one or 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 written 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
--baudrate= <i>rate</i>	OBSOLETE! Please use instead parameter -b or --bitrate, see next!
-b <i>rate</i> --bitrate= <i>rate</i>	Bitrate of the recorded connection, default is 115200.
--bit-order	Bit order, MSB (most significant bit) first=1 (default), MSB last=0. Only valid for Manchester.
--clock-delay	Add an additional $\frac{1}{2}$ clock delay for clock signals on their limit. SSI only! Default is off.
-c --config-file= <i>file</i>	Uses the settings specified in the given config file.
-C --create-config-file	Creates a new config file <code>msb_tools.config</code> in the current directory.
--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
--doubled-frame	Special mode to check data integrity (rarely used). SSI only. Default is off.
--frame-bits= <i>bits</i>	Number of data/clock bits in a data frame. Valid bits are 5..63, mandatory for synchronous bus systems.
-h --help	Help. Output of all program parameters.

22.2. RECORD DATA WITH MSB_RECORD

<code>-i</code> <code>--initiate</code>	Transfers the firmware to the analyzer, even it is already loaded.
<code>--io1=operation</code>	Use digital auxiliary channel IO1 (only MSB-RS485 and MSB-RS485-PLUS). The possible settings depend on the chosen transmission mode. For valid settings see section 22.2.6.1 below. Example (output bus direction): <code>msb_record --io1=3</code>
<code>--io2=operation</code>	Use digital auxiliary channel IO2 (only MSB-RS485 and MSB-RS485-PLUS). Valid values see section 22.2.6.1 below. Example (output bus validity): <code>msb_record --io2=4</code>
<code>-l list</code> <code>--log-signals=list</code>	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: <code>-l 2,3</code> oder <code>--log-signals=2,3,6,7</code> .
<code>-L</code> <code>--logic-mode</code>	Switches the inputs to logic signal levels (only MSB-RS232). Default are RS232 signal levels.
<code>--memory-test</code>	Forces the analyser to execute an internal memory test.
<code>--nice=niceness</code>	The nice parameter controls the <code>msb_record</code> 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. <code>msb_record --nice=0</code>
<code>-n serno</code> <code>--serno</code>	Use the analyzer with the given serial number <code>serno</code> .
<code>--output-buffering</code>	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 buffering the recorded events doesn't occur 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 <code>msb_format</code> tool.
<code>-o file</code> <code>--output=file</code>	Output file. Default is the standard output (console).
<code>--paused</code>	Starts the analyser in paused state. The record begins only after the program receives a remote start command.

KAPITEL 22. COMMANDLINE API

<code>-p protocol</code> <code>--protocol=protocol</code>	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.
<code>-r Command</code> <code>--remote=command</code>	Remote. Sends the following command to an already running program. The following commands are supported: <code>quit</code> quits and removes the background process <code>start</code> starts or resumes a recording <code>stop</code> stops or pauses the recording
<code>--show-analyzers</code>	Shows all available (connected) MSB-Analyzer.
<code>--sync-mode=mode</code>	Set the synchronization mode (autonom, master, slave) when using two or more analyzers for synchronous recordings. Default is autonom.
<code>-t num</code> <code>--time-delay=num</code>	Transfer delay. Slows the firmware transfer down by the indicated number num. 0 is no delay (default), maximum is 100.
<code>--transmission=mode</code>	Selects the transmission mode of the connected bus. Asynchronous, synchronous SSI or Manchester systems. Default is async. The possible values depend on the used analyzer, see section 22.2.6.2 below. Example: <code>msb_record --transmission=sync-ssi</code>
<code>--trigger-source=src</code>	Selects the trigger source for the data frame and data error output when enabled in the auxiliary digital IO settings, see parameter <code>--io1</code> and <code>--io2</code> . Possible sources are: A, B and A+B (default).
<code>-u</code> <code>--unique-file</code>	Stores the recorded data in the current working directory in a record file with an unique name like <code>YYYYMMDD-HHmMMmSSs.msblog</code> , for instance <code>20110324-03h04m41s.msblog</code> . This parameter is especially interesting in those applications where the record should start automatically after a (re)boot of the PC.
<code>-v</code> <code>--verbose</code>	Verbose, output of additional Information. .
<code>-V</code> <code>--verbose</code>	Output of the program version.

22.2. RECORD DATA WITH MSB_RECORD

<p><code>-w wiring</code> <code>--wiring=wiring</code></p>	<p>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)</p>
---	---

22.2.6.1 Digital IO setup parameter

The available values for the digital IO depends on the chosen transmission mode and analyzer typ. The following tables shows all possible variations. The first column represents the number assigned to the `--io[1,2]` parameter.

Number	Description	Transmission	Analyzer
0	Input with pull down	Asynchron	MSB-RS485 MSB-RS485-PLUS
1	Input with pull up	Asynchron	MSB-RS485, MSB-RS485-PLUS
2	Output static 0	Asynchron	MSB-RS485, MSB-RS485-PLUS
3	Output static 1	Asynchron	MSB-RS485, MSB-RS485-PLUS
4	Output of the bus direction	Asynchron	MSB-RS485, MSB-RS485-PLUS
5	Output of the bus validness	Asynchron	MSB-RS485, MSB-RS485-PLUS
6	Output CHN1 validity	Asynchron	MSB-RS485, MSB-RS485-PLUS
7	Output CHN2 validity	Asynchron	MSB-RS485, MSB-RS485-PLUS
8	Output CHN3 validity	Asynchron	MSB-RS485, MSB-RS485-PLUS
9	Output CHN4 validity	Asynchron	MSB-RS485, MSB-RS485-PLUS
10	Output data error	Asynchron Synchron SSI Manchester	MSB-RS485, MSB-RS485-PLUS
11	Supply +5V/50mA	Asynchron Synchron SSI Manchester	MSB-RS485-PLUS
12	Output data frame	Synchron SSI Manchester	MSB-RS485-PLUS

KAPITEL 22. COMMANDLINE API

22.2.6.2 Transmission parameters

The supported field-bus transmissions depends on the used analyzer. The older MSB-RS232 and MSB-RS485 allow only the analysis of asynchronous transmissions whereas the PLUS types support synchronous and Manchester coded transmission. The following table gives you an overview about the available values for the `transmission` parameter:

Parameter	Transmission	Analyzer
<code>async</code>	Asynchronous transmissions	all
<code>sync-ssi</code>	Synchron SSI transmissions	MSB-RS232-PLUS, MSB-RS485-PLUS
<code>manchester-1</code>	Manchester I (G.E. Thomas)	MSB-RS485-PLUS
<code>manchester-2</code>	Manchester I (IEEE 802.3)	MSB-RS485-PLUS
<code>manchester-t0</code>	Differential Manchester T0	MSB-RS485-PLUS
<code>manchester-t1</code>	Differential Manchester T1	MSB-RS485-PLUS

22.3 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 separated values). Without any parameter you will get a list of the occurred events, each one with informations about the time, the kind of event, the data value or line state. This complies with the format specifier 'l' 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 22.3.1.

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 string⁴ 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
```

⁴A list of all format identifiers can be found in the format identifier table.

22.3. FORMATTED OUTPUT WITH MSB_FORMAT

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,
```

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).

22.3.1 Output of any character

You want to insert a non-printable character or define a line feed, independent of the operating system⁵.

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.

⁵Under linux all lines are ended with a single linefeed, while Windows uses a combination of Carriage Return/Linefeed.

KAPITEL 22. COMMANDLINE API

22.3.2 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
```

22.3.3 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.

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 carriage return character is #013
[...]	[format]	User defined date and time output, see table below 22.3.4 .
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 representative 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.

22.3. FORMATTED OUTPUT WITH MSB_FORMAT

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 12.4.2. 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. Outputted 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.

KAPITEL 22. COMMANDLINE API

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: <code>-Fd+u</code> results to <code>2012-04-11 15:57:40+184935</code> . 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.
x1...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.

22.3.4 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:

⁶This exclusively applies in Windows, not Linux

22.3. FORMATTED OUTPUT WITH MSB_FORMAT

```
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. Therefore 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 decimal 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)
%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

KAPITEL 22. COMMANDLINE API

%% the literal % character

22.3.5 msb_format program parameters

Call the program with:

```
msb_format [OPTION]...
```

[OPTION] can contain one or 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
<code>-c</code> <code>--config-file=file</code>	Uses the settings specified in the given config file.
<code>--disable-linefeed</code>	Suppress linefeed in ASCII mode.
<code>-F</code> <code>--format=formatstring</code>	Output format definition, see format table.
<code>-h</code> <code>--help</code>	Help. Output of all program parameters.
<code>-o file</code> <code>--output=file</code>	Output file. Default is the standard output (console).
<code>-s list</code> <code>--signal-names=list</code>	Pass a comma separated list of signal names for use in the output. For instance: <code>--signal-names=DCD,RxD,TxD,DSR,DTR,CTS,RTS,RI</code> names all signals according to the RS232 DCE standard names.
<code>--signal-rs232-dte</code>	Predefined signal list. Names all signals according to RS232 DTE.
<code>--signal-rs232-dce</code>	Predefined signal list. Names all signals according to RS232 DCE.
<code>--signal-rs485</code>	Predefined signal list for use with the RS485 analyzer. Names all signals as like: <code>--signal-names=CH1,CH2,CH3,CH4,BDIR,BSIG,IO1,IO2</code>
<code>-v</code> <code>--verbose</code>	Verbose, output of additional Information.
<code>-V</code> <code>--verbose</code>	Output of the program version.

22.4 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 its standard input and writes 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

22.4. FILTERING DATA OUTPUT WITH MSB_FILTER

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 arguments⁷. Without any filter parameter the program will block all data flow.

22.4.1 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
```

22.4.2 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.

22.4.3 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 specific 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`.

⁷Linux user use the `cat` command instead of the `type` command.

KAPITEL 22. COMMANDLINE API

22.4.4 msb_filter program parameter

Call the program with:

```
msb_filter [OPTION]...
```

[OPTION] can contain one or 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= <i>file</i>	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= <i>list</i>	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: <code>--pass-signals=2,3,6,7</code> .
-s --pass-selection= <i>list</i>	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: <code>-s 100,200</code> or <code>--pass-selection=100,200</code> .
-t --pass-time= <i>list</i>	pass all events in the given time range in seconds as comma separated list with first time, last time. For instance: <code>--pass-time=1.257,10.231</code> 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.

22.5 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.

22.5. SPLIT RECORDS WITH MSB_SPLIT

22.5.1 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.msblog, ...

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 22.6.9).

```
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:

KAPITEL 22. COMMANDLINE API

```
20110510_15h53m24s.msblog
20110510_15h58m31s.msblog
20110510_16h02m10s.msblog
...
```

22.5.2 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
```

22.5.3 Keep only a given number of records

Let's say you want to get a record every hour but only want to keep the last 10 records to save disc space. This is a typical scenario when you search an error which occurs extremely rare.

By default the `msb_split` tool keeps all generated split files. The number of files increases as long as the record runs.

With the parameter `--keep-max-files=N` you can limit the generated files to a given number. The following example creates a record every hour but keeps only the last ten:

```
msb_record -b115200 -p8N1 | msb_split -D -t3600 --keep-max-files=10
```

When the eleventh record starts, the program removes the very first automatically. This happens again with the beginning of the twelfth record. `msb_split` then removes the record of the second hour. And so on...

22.5.4 `msb_split` Program Parameter

Call the program with: `msb_split [OPTION]... [PREFIX]`

[OPTION] can contain one or 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	Description
<code>-a length</code> , <code>--suffix-length=length</code>	Number of usable characters for the enumerating suffix. Default is 2 characters.
<code>-h</code> , <code>--help</code>	Output of all program parameters.
<code>-c</code> <code>--config-file=file</code>	Uses the settings specified in the given config file.
<code>-d</code> , <code>--numeric-suffix</code> <code>--dir=directory</code>	The files are names numerically, the default is alphabetically. Use the given directory for the output (split) files.

22.6. TRIGGER A RECORD WITH MSB_TRIGGER

-D, --date-time-suffix	The files names are extended with the date and time of the first occurred event in the format <code>YYYYMMDD_HHhMMmSSs</code> .
-k, --keep-max-files= <i>N</i>	Keep only the last recorded files of the given number <i>N</i> and automatically remove the older (earlier) split record files.
-n, --number= <i>quantity</i>	Quantity of the events per file. Each event occupies 24 bytes.
-v, --verbose	Output of additional information.
-V, --version	Output of the program version.

22.6 Trigger a record with `msb_trigger`

Long-term records in conjunction with the command line tools often serves the purpose to find a rarely occurring 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` program 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.

KAPITEL 22. COMMANDLINE API

22.6.1 Edit a trigger script

You can create and/or edit a trigger script with every editor but we recommend to use the script editor provided by the analyzer software.

The analyzer editor not only offers a code frame work for trigger scripts, it also let you test little Lua code snippets in the editor buffer itself.

To open the editor, first start the analyzer software and select the 'Script Editor' in the view menu of the control program.

In the editor click the 'New file' icon in the toolbar or press **Ctrl** + **N** and select 'Trigger' in the appearing script wizard. There are a lot of examples too. You can open them by click on the 'Open file' icon or via **Ctrl** + **O**. All trigger scripts stay in the 'Trigger' folder.

22.6.2 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`.

```
1 function trigger( data, intval, dir, alter )
2   — return true if the trigger condition occurred
3 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.

```
1 lastByte=-1
2 function trigger( data )
3   if lastByte == 0x0D and data ~= 0x0A then
4     — trigger
5     return true
6   end
7   lastByte = data
8   return false
9 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 13.8.3. 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.

```
1 — the DTR signal number
2 DTR = 4
3 — here we store the last DTR line state
```

22.6. TRIGGER A RECORD WITH MSB_TRIGGER

```
4 last_dtr = -1
5 function trigger()
6     local dtr = event.level( DTR )
7     — check for a falling edge
8     if dtr == -1 and last_dtr == 1 then
9         — trigger
10        return true
11    end
12    last_dtr = dtr
13    return false
14 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
```

22.6.3 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 22.6.9). 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 2 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 > transmission.bytepause( 3.5 ) then
5         — seq represents the current telegram, test checksum
6         — read 16 bit ckecksum of current telegram
7         if #seq >= 2 then
8             local cks_is = seq:byte(-1) * 256 + seq:byte(-2)
9             local cks_must = checksum.crc16_modbus( seq:sub(1,-3) )
10            if cks_is ~= cks_must then
11                return true
12            end
13        end
14        — start a new telegram sequence
15        seq = ""
16    end
17    — add the current data byte to the actual telegram sequence
18    seq = seq..string.char( data )
19    return false;
20 end
```

KAPITEL 22. COMMANDLINE API

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-trigger=2560 ↔
--post-trigger=2560 script.lua > record.msblog
```

22.6.4 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-trigger=2560 ↔
--post-trigger=2560 script.lua > record.msblog
```

22.6.5 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 > transmission.bytepause( 3.5 ) then
8          — seq represents the current telegram, test checksum
9          — read 16 bit ckecksum of current telegram
10         if #seq >= 2 then
11             local cks_is = seq:byte(-1) * 256 + seq:byte(-2)
12             local cks_must = checksum.crc16_modbus( seq:sub(1,-3) )
13             if cks_is ~= cks_must then
14                 print( string.format( "%6f\tis:%04X, must:%04X",
15                                     ts, cks_is, cks_must ) )
16             end
17         end
18         — start a new telegram sequence
19         seq = ""
20         — store the telegram time
```

22.6. TRIGGER A RECORD WITH MSB_TRIGGER

```
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.

22.6.6 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 is easy to cover code meant either for trigger or scan. The script below again pick-ups our Modbus RTU example.

KAPITEL 22. COMMANDLINE API

```
1  — a Lua string representing the last received data of one channel
2  seq = ""
3  — contains the time stamp of the first byte of the current telegram
4  ts = 0
5  — the trigger function
6  function trigger( data, intval, dir, alter )
7      if intval > transmission.bytepause( 3.5 ) then
8          — seq represents the current telegram, test checksum
9          — read 16 bit ckecksum of current telegram
10         if #seq >= 2 then
11             local cks_is = seq:byte(-1) * 256 + seq:byte(-2)
12             local cks_must = checksum.crc16_modbus( seq:sub(1,-3) )
13             if cks_is ~= cks_must then
14                 if DEBUG then
15                     print( string.format( "%6f\tis:%04X, must:%04X",
16                                             ts, cks_is, cks_must ) )
17                 else
18                     return true
19                 end
20             end
21         end
22         — start a new telegram sequence
23         seq = ""
24         — store the telegram time
25         ts = event.time()
26     end
27     — add the current data byte to the actual telegram sequence
28     seq = seq..string.char( data )
29     return false;
30 end
```

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.

22.6.7 Multiple triggering

You can force the `msb_trigger` tool to continue parsing the input data by passing the parameter `--multi-trigger`. Given this parameter the trigger tool will output the specified number of pre- and post-trigger events and then proceeds reading the data for further trigger conditions. The resulting record file contains several trigger fragments, means: You will get one record with different blocks of pre- and post-trigger events separated by a gap of an undefined time (the time between the trigger condition less the specified pre- and post-trigger events).

The `msb_trigger` tool marks this 'gaps' with special gap events. In the DataView the gaps are displayed as two consecutive yellow fields as shown in the picture below.

KAPITEL 22. COMMANDLINE API

is true.

If you are not interested in further trigger conditions, you can return true and that's all. But consider if the script is called again with the next incoming data byte. The internal string sequence `seq` still contains the data of the former trigger conditions and the script will again return true even though the next sequence 'hello' or 'world' didn't arrive.

To avoid this, you must reset the trigger mechanism. Here: To clear the internal buffer for the pattern search. We do this in line 15.

We recommend to reset you trigger script always properly, independent of your intention to use it for a single or multiple triggering.

22.6.8 Provided Lua modules

The following Lua extensions are supported by the `msb_trigger` tool:

- **base16 module** : Encoding and decoding functions for base16 sequences (i.e. used in Modbus ASCII and Intel SRecord). See Lua Extensions, section 18.2.1.
- **bit32 module** : OBSOLETE! With the integration of Lua version 5.3 the bit32 module is not longer needed and will be removed in the next future. Please use the native Lua bitwise operators instead.
- **checksum module** : Contains checksum algorithms for Modbus RTU (CRC16), Modbus ASCII (LRC), BACNet (CRC8 and CRC16), DNP3 and CRC16 CCITT (Kermit). See Lua Extensions, section 18.2.5.
- **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, section 13.8.3.
- **transmission module** : Returns information about the current baudrate, data bits, parity and stopbits. See Lua Extension, section 18.2.9.
- **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 13.8.6.

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.

22.6.9 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	Description
-c --config-file= <i>file</i>	Uses the settings specified in the given config file.
-h, --help	Output of all program parameters.
--debug	Enables the debug mode.

22.7. ONE CONFIG FILE FOR ALL

<code>--multi-trigger</code>	The <code>msb_trigger</code> tool does not stop after the first triggering but continues to parse the incoming data for further trigger conditions.
<code>--pass-through</code>	By-pass all trigger conditions. The trigger script is executed but the program outputs all passed events independent of the script result. This is intended for scripts which don't want to change a record but collecting information in a file specified in the script itself. E.g. create a list where telegrams with invalid checksum are found in the passed record.
<code>--pre-trigger=events</code>	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).
<code>--post-trigger=events</code>	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.
<code>--trigger-source=source</code>	The kind of events passed through the trigger script. Default is <code>data</code> but you can also trigger on certain line state conditions by passing <code>signal</code> as the trigger source.
<code>-v,</code> <code>--verbose</code>	Output of additional information.
<code>-V,</code> <code>--version</code>	Output of the program version.

22.7 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 file⁸.

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

⁸This only works of course for the analyzer tools.

KAPITEL 22. COMMANDLINE API

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
```



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 definition 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. Extensions 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.

ANHANG A. ASCII CHARACTER TABLE

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:

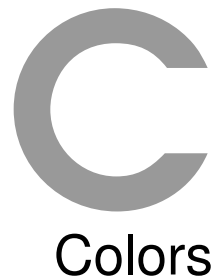
Deviations of a maximum of $\pm 3\%$ can be accepted and compensated, higher deviations should be avoided.

Baudrate tolerance

Avoid more than 3% deviation in the baudrate generation. This will result in bit errors.

ANHANG B. BAUDRATE MEASURING

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.



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 explicitly.

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 $16 \times 16 \times 16 = 4096$ 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 $256 \times 256 \times 256 = 16777216$ 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.

ANHANG C. COLORS



















C.3.1 Grey colors

Name/Value	Color	Name/Value	Color
black #000000		dim grey #696969	
dark grey #a9a9a9		grey #bebebe	
light grey #d3d3d3		white #ffffff	

C.3.2 Basic colors

Name/Value	Color	Name/Value	Color
blue #0000ff		green #00ff00	
red #ff0000		cyan #00ffff	
magenta #ff00ff		yellow #ffff00	

C.3.3 Extended colors

Name/Value	Color	Name/Value	Color
medium spring green #7fff00		forest green #228b22	
lime green #32cd32		dark green #006400	
aquamarine #70db93		spring green #00ff7f	
medium aquamarine #66cdaa		sea green #238e6b	
medium turquoise #70dbdb		dark turquoise #00ced1	
steel blue #236b8e		sky blue #3299cc	
slate blue #007fff		light steel blue #b0c4de	
cornflower blue #6495ed		navy #23238e	
medium blue #0000cd		dark slate blue #483d8b	

C.3. PREDEFINED COLOR NAMES

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	

ANHANG C. COLORS

D

Windows Trouble-Shooting

This chapter gives you some hints in case the analyzer device is not detected by the software or does not communicate properly with it afterwards. If you are running Linux, see the according chapter Linux Trouble-Shooting.

If a problem arises, first separate the analyzer from any RS422/RS485 bus and disconnect it from the PC. Then check:

- 1 Is the analyzer got hot?
- 2 Shows the analyzer any signs of damage?

If you can deny both questions following the instructions in the table below.

Should the problem persist, there may be a malfunction. In this case or if your analyzer shows other signs of damage please contact IFTOOLS and hold ready the serial number of your device. The number sticks on the bottom of the housing. The best way to reach us is via email at support@iftools.com.

D.1 Check analyzer connection

The following tests are intended to check the connectivity of your analyzer with your PC. This covers the correct driver installation and the proper USB function of the analyzer device.



Reconnect your MSB-RS485 with your PC but leave the bus ports open!

Symptom	Cause	Countermeasure
Device LEDs do not light up	No installed driver	Install driver, see D.3
	Broken USB cable or USB port	Exchange the USB cable, use a different USB port/hub
	Defective analyzer	Exchange device
Analyzer is not detected (Program cannot find device)	USB enumeration failure	Pass analyser S/N to program start, see D.4
	Wrong driver or driver mismatch	Reinstall driver, see D.3
	Driver conflict caused by other USB device	Check device detection log, see D.5

ANHANG D. WINDOWS TROUBLE-SHOOTING

Firmware transmission fails	Not reliable USB cable Transmission error Defective analyzer	Exchange USB cable Reduce transmission speed/timeout, see D.4 Exchange device
All LEDs flash red after firmware transfer	EEPROM checksum error	Contact IFTOOLS, see D.8

A correctly connected and functional MSB-RS485 (firmware loaded successfully) is indicated by flashing both red LEDs alternately.

D.2 Check analyzer bus connections

As soon as you connect your analyzer with a bus system (either a small test setup or a whole industrial field-bus) your analyzer becomes part of a greater system. This may include the flow of compensatory currents through your analyzer or voltage surges on the data lines and may affect its proper working, even can damage the analyzer. So please make sure that you made the correct bus connections including the ground line before apply the bus to your analyzer.



Connect your MSB-RS485 analyzer with your bus and start the software

Symptom	Cause	Countermeasure
The analyzer becomes unresponsive	bus short circuit	Recheck the correctness of your bus connection (signals, ground)
The analyzer does not record any data	Record not started Recording of data bytes disabled Wrong bus connection	Start record Check record settings Check connection, particular the green port LEDs
Analyzer stops recording by itself	Full disk or maximum record size reached	Reduce recorded events (signal settings), free disk space
Analyzer quit working unexpectedly	USB disabled by Windows power management	Disable power management, see D.6

D.3 (Re)Install driver

The MSB-RS485 analyzer integrates a chip from FTDI to communicate with your PC. This is a widely spread chip. If you have other USB devices connected to your PC using the same chip a driver mismatch or conflict caused by different driver versions cannot be ruled out.

D.3.1 Remove driver

To be on the safe side, first remove all existing drivers for this chip from your Windows systems before installing it again.

You will find the appropriate driver uninstaller on the IFTOOLS CDROM in section *Driver & Tools*. Click [Interactive deinstallation](#).

Optional you can visit our [download page](#) and input `CDMUninstaller` in the search

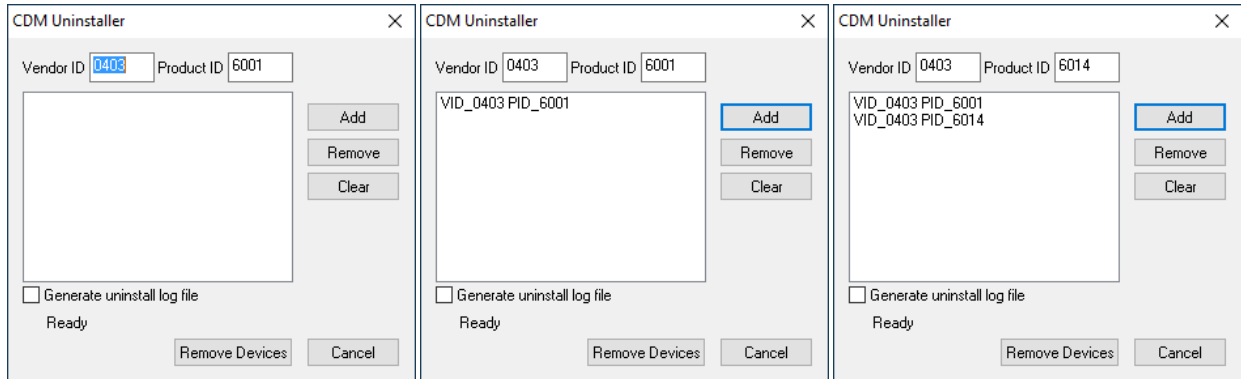
D.3. (RE)INSTALL DRIVER

field of the page.

After you have downloaded the zip archive file, unpack it and run the program `CDMuninstallerGUI.exe`, possibly as admin.



Please note! Disconnect the analyzer before starting the driver deinstallation!



The IFTOOLS analyzer are using either a chip with the USB product ID 6001 or 6014, (the Vendor ID 0403 stands for FTDI). So add both to the device list of the uninstaller as shown above in the right picture. Then click the 'Remove Devices' button.

The program displays a little message box when the action was completed. You can ignore any message like 'Failed to remove device...' because it just means, that the driver was already removed.



D.3.2 Install driver

There are several ways to install the proper device driver.

- 1 Automatic installation by Windows (needs an active Internet connection)
- 2 Installation from the IFTOOLS CDROM or an IFTOOLS USB stick
- 3 Install driver from other sources

Automatic installation by Windows

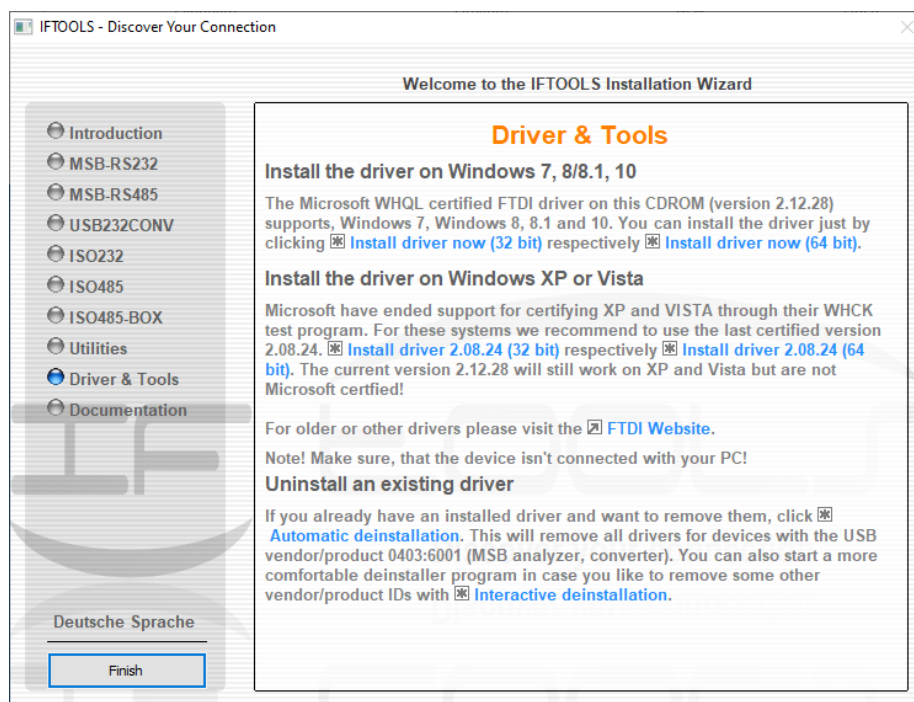
This is the easiest way. After you have removed the old driver(s), just connect your analyzer with your PC and let do Windows the rest. The driver installation may need some minutes and depending on your Windows version you might get a hint what's going on. But to be sure, open the device manager (see [D.7](#) and expand the 'Ports (COM & LPT)' list entry.

When the installation is complete (either indicated by a short message or an new entry in the 'Ports (COM & LPT)' list, you can start the analyzer software.

Installation from CDROM or USB stick

Insert the IFTOOLS CDROM or plugin the IFTOOLS USB stick. New Windows versions will ask you to allow the access to the medium instead of starting the IFTOOLS Installation Wizard automatically. In this case grant permission and run the `setup.exe` in the root directory of the CDROM or USB stick manually.

ANHANG D. WINDOWS TROUBLE-SHOOTING



Select 'Driver & Tools' in the left column as shown in the picture above. The IFTOOLS Installation Wizard provides you with relatively new and tested driver versions for all kind of Windows OS (32 and 64 bit). Just click the appropriate 'Install driver...' link to start the installation.

Install driver from other sources

FTDI provides the always newest driver on their [VCP drivers page](#). We recommend to download the setup executable because it allows you a more or less automatic driver installation just by executing this package.

Another source for the driver is the IFTOOLS [driver page](#). Here too you can chose between an executable driver package (Setup Exe) or the standard driver package for use with the Windows device manager.

After download start the installation by executing the driver setup exe or select the driver package from within the device manager.

D.4 Helpful program arguments

The analyzer software, particularly the program responsible for the communication with the analyzer device (the control program) features some special parameters to affect the device detection and firmware transfer.

In normal cases you don't need to know them. The default settings handle the device access pretty good. These parameters might come handy when a functional analyzer was not detected or the transfer of the firmware fails.

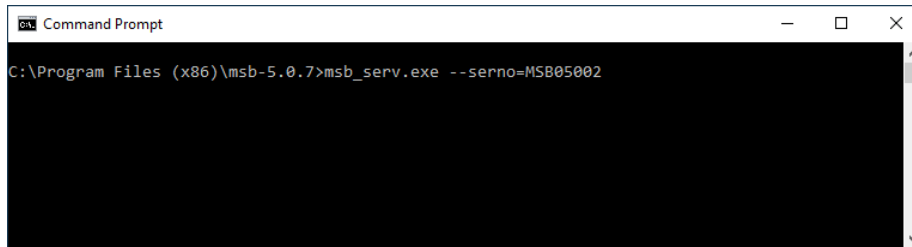
To start the analyzer software with additional parameters open a command shell (or command prompt).

D.4.1 Analyzer not found

If the analyzer was not found pass it's serial number as an additional parameter `--serno=MSBxxxxx` where MSBxxxxx is the serial number. You will find the serial number on the bottom ca-

D.5. PLEASE HELP US WITH CONFLICTING DEVICES

se:



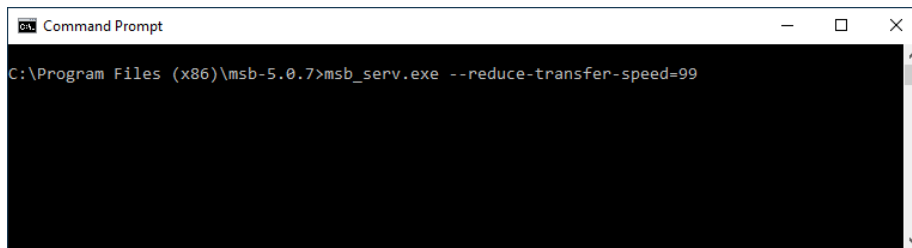
```
Command Prompt
C:\Program Files (x86)\msb-5.0.7>msb_serv.exe --serno=MSB05002
```

Analyzers of the third generation come with a bar code for the serial number. In this case replace the `xxxxxx` with the 5 last digits of the bar code number.

D.4.2 Firmware transfer error

That the software reports a firmware transfer error (the analyzer was detected but cannot be initialized) is rather rare and almost depends on the PC hardware. You can slow down the firmware transmission with the parameter `--reduce-transfer-speed=speed`. Allowed values for *speed* are 0...99.

The following example transfers the analyzer firmware with the lowest speed:



```
Command Prompt
C:\Program Files (x86)\msb-5.0.7>msb_serv.exe --reduce-transfer-speed=99
```

You will find an detailed description of all available parameters in the manual in chapter 7.15. If necessary, you can apply the critical parameters to the software start icon. This is explained in chapter 7.12.

D.5 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.6 Disable USB power management

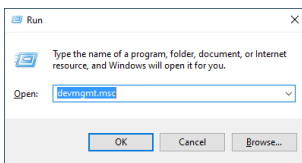
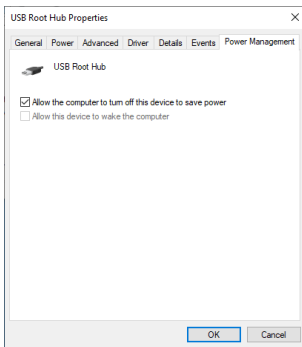
The analyzer device loses its connection and stops working unexpectedly. Most of the time the analyzer LEDs goes off too. This happens in particular when using a Notebook

ANHANG D. WINDOWS TROUBLE-SHOOTING

or Laptop.

The reason: To preserve power, the Microsoft Windows OS tries to disable USB (ports) when a device is idle. Under certain circumstances this function does not work properly and causes an USB devices to fail to respond when called. To resolve this issue, disabling power management on the USB hub is an appropriate means. Use the following steps for this:

- 1 Open the Device Manager, see [D.7](#)
- 2 Double-click the Universal Serial Bus Controllers branch to expand it.
- 3 Right-click USB Root Hub, and then click Properties.
- 4 Click Power Management.
- 5 Deselect Allow the computer to turn off this device to save power.
- 6 Repeat Steps 5 through 7 for each USB Root Hub.
- 7 Click OK, and close Device Manager.
- 8 Reboot you system



D.7 Windows Device Manager

Even if the design and the task of the device manager remains unchanged through the last 20 years every new windows version makes it a little bit harder to find it. Presumable to protect the system against incorrect hardware settings caused by the user. Truth is that in normal circumstances a normal user rarely have a need for the device manager.

Nevertheless it is the first place to look at when something - especially some device - refuses to work. Luckily the shortcut to start the device manager by a single command remains the same since Windows XP. Just press:

Windows Key + **R**

In the just opening dialog input: `devmgmt.msc` and press **Enter** or click OK.

D.8 Other problem(s)

Your problem isn't listed here!

In case of problems or questions 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.

Also don't forget to provide the serial number of your analyzer (you will find it on the bottom case).

E

Linux Trouble-Shooting

This chapter gives you some hints in case the analyzer device is not detected by the software or does not communicate properly with it afterwards. If you are running Windows, see the according chapter Windows Trouble-Shooting.

If a problem arises, first separate the analyzer from any RS422/RS485 bus and disconnect it from the PC. Then check:

- 1 Is the analyzer got hot?
- 2 Shows the analyzer any signs of damage?

If you can deny both questions following the instructions in the table below.

Should the problem persist, there may be a malfunction. In this case or if your analyzer shows other signs of damage please contact IFTOOLS and hold ready the serial number of your device. The number sticks on the bottom of the housing. The best way to reach us is via email at support@iftools.com.

E.1 Check analyzer connection

All standard Linux kernels innately contain the necessary driver module which is needed to communicate with the analyzer. Nevertheless you can be trapped by the lot of different Linux variants and their differing implementations (especially user and group permissions) which may make the correct functioning difficult.

The following tests are intended to check the connectivity of your analyzer with your PC.



Reconnect your MSB-RS485 with your PC but leave the bus ports open!

Symptom	Cause	Countermeasure
Device LEDs do not light up	Broken USB cable or USB port	Exchange the USB cable, use a different USB port/hub
	Defective analyzer	Exchange device
Analyzer is not detected (Program cannot find device)	Missing permission	Check your permission, see E.3
	Missing or invalid udev rule	Reinstall udev rule, see E.4
	Installed Braille driver	Remove Braille driver, see E.5

ANHANG E. LINUX TROUBLE-SHOOTING

	USB enumeration error	Check device detection log, see E.7
	Other reasons	Check system log, see E.8
Firmware transmission fails	Not reliable USB cable	Exchange USB cable
	Transmission error	Reduce transmission speed/timeout, see E.6
	Defective analyzer	Exchange device
All LEDs flash red after firmware transfer	EEPROM checksum error	Contact IFTOOLS, see E.9

A correctly connected and functional MSB-RS485 (firmware loaded successfully) is indicated by flashing both red LEDs alternately.

E.2 Check analyzer bus connections

As soon as you connect your analyzer with a bus system (either a small test setup or a whole industrial field-bus) your analyzer becomes part of a greater system. This may include the flow of compensatory currents through your analyzer or voltage surges on the data lines and may affect its proper working, even can damage the analyzer. So please make sure that you made the correct bus connections including the ground line before apply the bus to your analyzer.



Connect your MSB-RS485 analyzer with your bus and start the software

Symptom	Cause	Countermeasure
The analyzer becomes unresponsive	bus short circuit	Recheck the correctness of your bus connection (signals, ground)
The analyzer does not record any data	Record not started	Start record
	Recording of data bytes disabled	Check record settings
	Wrong bus connection	Check connection, particular the green port LEDs
Analyzer stops recording by itself	Full disk or maximum record size reached	Reduce recorded events (signal settings), free disk space

E.3 Check your permission

Linux handles the analyzer as a serial USB device like `/dev/ttyUSBx` and you need read/write permissions to access the device. Except for root only members of the group `dialout` (Debian based systems like Ubuntu) or `uucp` (SuSE) are allowed to do this. Open a terminal (with an connected analyzer) and type:

```
ls -l /dev/ttyUSB*
```

You will get something similar like this:

```
crw-rw---- 1 root dialout 188, 0 2020-08-26 14:47 /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 0 2020-08-26 14:47 /dev/ttyUSB1
```

Now check your group memberships by input the command:

E.4. INSTALL UDEV RULE

groups

The following result indicates, that you are member of your own group (indicated as YOUR_USER_NAME), the groups cdrom, floppy, audio, video and plugdev. But you are not member of the required group dialout (or uucp) which you need to access the analyzer device.

```
YOUR_USER_NAME cdrom floppy audio video plugdev
```

To add yourself to the group dialout (or uucp) you must execute the following command via sudo. (In case you have not sudo rights, ask your system administrator). SuSE users must replace dialout with uucp!

```
sudo adduser YOUR_USER_NAME dialout
```

Please note! You have to logout and new login first until the changes are available. A reboot is not necessary.

E.4 Install udev rule

Beginning with version 5.0 the analyzer access via a virtual serial port was replaced by a direct USB access (FTDI d2xx). This change highly increases the data transfer rate and is mandatory for the newest analyzer generation (PLUS types).

Unfortunately both kinds of device accesses are mutually exclusive, which means: You cannot perform a direct USB access via FTDI d2xx if the device is registered to the kernel as virtual serial port (/dev/ttyUSBx).

An proper udev rule is needed to exclude an analyzer device from this registration as soon as the kernel detects it. Normally the rule is installed during the software installation and you can check it with:

```
ls -l /etc/udev/rules.d
```

In a Linux system with a working analyzer you will see something like this:

```
-rw-r--r-- 1 root root 262 Jan 12 14:23 10-iftools-msb.rules
-rw-r--r-- 1 root root 620 Feb 23 2016 70-persistent-net.rules
-rw-r--r-- 1 root root 58549 Jan 17 2019 70-snap.core.rules
-rw-r--r-- 1 root root 984 Mar 4 16:10 70-snap.telegram-desktop.rules
```

Important is the file 10-iftools-msb.rules. If it does not exist, open a terminal and cd (change working directory) to the installation directory of your analyzer software.

```
cd ~/msb-5.0.9
sudo ./udev-install.sh
```

A correct udev rule for the IFTOOLS analyzers looks like:

```
1 # 10-iftools-msbc.rules
2 #
3 # Apply the new rules with: sudo udevadm trigger
4 ATTRS{idVendor}=="0403",\
5 ATTRS{idProduct}=="6001|6014",\
6 ATTRS{manufacturer}=="IFTTOOLS",\
7 MODE="0660",\
8 GROUP="dialout",\
9 RUN+="/bin/sh -c 'echo -n %k:1.0 > /sys/bus/usb/drivers/ftdi_sio/unbind '"
```

This rule is applied to all USB devices with a FTDI chip (vendor ID 0403) and a certain chip type used in the analyzers (Product ID 6001 or 6014). To make sure, that no other

ANHANG E. LINUX TROUBLE-SHOOTING

devices with the same chip configurations are affected, the rule checks the manufacturer (IFTOOLS) too.

You can test the correct operation of the rule by open a terminal, then (re)connect your analyzer and input:

```
dmesg
```



```
jb@Bag-End: ~
[ 9131.618550] usb 3-3.4: new high-speed USB device number 13 using xhci_hcd
[ 9131.734770] usb 3-3.4: New USB device found, idVendor=0403, idProduct=6014
[ 9131.734775] usb 3-3.4: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 9131.734778] usb 3-3.4: Product: MSB-RS485-PLUS
[ 9131.734780] usb 3-3.4: Manufacturer: IFTOOLS
[ 9131.734783] usb 3-3.4: SerialNumber: MSB05002
[ 9131.737142] ftdi_sio 3-3.4:1.0: FTDI USB Serial Device converter detected
[ 9131.737205] usb 3-3.4: Detected FT232H
[ 9131.737435] usb 3-3.4: FTDI USB Serial Device converter now attached to ttyUSB2
[ 9131.754944] ftdi_sio ttyUSB2: FTDI USB Serial Device converter now disconnected from ttyUSB2
[ 9131.754959] ftdi_sio 3-3.4:1.0: device disconnected
jb@Bag-End:~$
```

Here the kernel detects a new USB device (as an example our MSB-RS485-PLUS) and disconnected it from ttyUSB2 so it is free for the direct d2xx access.

E.5 Remove Braille driver

You have the correct permissions for accessing the `/dev/ttyUSBx` device and the according `udev` rule is installed and works properly. Anyhow the analyzer wasn't detected by the software.

Be sure, that you don't have an installed Braille driver (Braille is a tactile writing system used by people who are visually impaired). Disconnect and connect the analyzer again. Open a console and input:

```
dmesg
```

If the output displays something like this:

```
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 based systems (Ubuntu) for instance with:

```
sudo apt-get remove brltty
```

E.6 Helpful program arguments

The analyzer software, particularly the program responsible for the communication with the analyzer device (the control program) features some special parameters to affect the device detection and firmware transfer.

In normal cases you don't need to know them. The default settings handle the device access pretty good. These parameters might come handy when a functional analyzer was not detected or the transfer of the firmware fails.

To start the analyzer software with additional parameters open a terminal and `cd` (change working directory) into the analyzer software installation directory (mostly `$HOME/msb-7.0.2`).

E.7. PLEASE HELP US WITH CONFLICTING DEVICES

E.6.1 Analyzer not found

If the analyzer was not found pass its serial number as an additional parameter `--serno=MSBxxxxxx` where MSBxxxxxx is the serial number. You will find the serial number on the bottom case:

```
./msb_serv -nMSBxxxxxx
```

Analyzers of the third generation come with a bar code for the serial number. In this case replace the `xxxxxx` with the 5 last digits of the bar code number.

E.6.2 Firmware transfer error

That the software reports a firmware transfer error (the analyzer was detected but cannot be initialized) is rather rare and almost depends on the PC hardware. You can slow down the firmware transmission with the parameter `--reduce-transfer-speed=speed`. Allowed values for *speed* are 0...99.

The following example transfers the analyzer firmware with the lowest speed:

```
./msb_serv -r99
```

You will find a detailed description of all available parameters in the manual in chapter 7.15. If necessary, you can apply the critical parameters to the software start icon. This is explained in chapter 7.12.

E.7 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 an IFTOOLS analyzer.

Because of the manifold combinations of existing Linux distributions and USB devices 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.8 Check system log with dmesg

The Linux kernel produces a lot of useful information when detecting a new USB device. You can always check the latest (newest) information (stored in a kernel ring buffer) with the command:

```
dmesg
```

Just reconnect the analyzer and take a look into the `dmesg` output. An functional analyzer must always trigger some new lines in the log.

You can store the whole output with:

```
dmesg > log.txt
```

In doubt, just send this output to support@iftools.com

E.9 Other problem(s)

Your problem isn't listed here!

We are very much interested 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. 59
ETX	End of Text, in the ASCII character set defined as hex 0x03. ETX marks the end of a message or datagram. 93
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
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. 56
Multi-Master	Bus nodes which are allowed to initiate a data transfer with other bus nodes are denoted 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
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. 163
RTF	A document file format developed by Microsoft for cross-platform document interchange. 59

Glossar

Notation	Description
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. 93
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. 165
UART	Universal Asynchronous Receiver Transmitter. Electronic element to send or receive data over a serial data line. 2 , 20

Index

- Absolute Time, [62](#)
- Analyser
 - multiple, [39](#)
- Analysis tools
 - see Views, [37](#)
- base16
 - decode, [215](#)
 - encode, [215](#)
- box.setup, [138](#)
- box.space, [138](#)
- box.text, [139](#)
- Break
 - display, [55](#)
 - search, [65](#), [83](#)
- Break error, [56](#)
- checksum
 - crc16_bacnet, [220](#), [221](#)
 - crc8_bacnet, [219](#)
 - dnp3, [221](#)
 - kermit, [221](#)
 - lrc, [222](#)
 - modbus, [222](#)
- config.setmaxop, [223](#)
- Control display
 - active lines, [37](#)
 - PC connection, [36](#)
 - recording capacity, [35](#)
 - toggle information, [35](#)
- Control program
 - parameter, [42](#)
 - Short commands, [41](#)
 - Special parameters, [43](#)
- data
 - at, [72](#)
 - cursorcolours, [73](#)
- Data View, [55](#)
 - copy section, [59](#)
 - export section, [59](#)
 - Font, [61](#)
 - Goto address, [58](#)
 - save section, [59](#)
 - selection, [58](#)
 - Short commands, [77](#)
 - Show control chars, [61](#)
- Datagram
 - displaying, [106](#)
- Datenmonitor
 - see Data View, [55](#)
- debug
 - clear, [74](#), [140](#)
 - print, [74](#), [140](#)
 - resume, [75](#), [141](#)
 - summarize, [75](#), [141](#)
 - suspend, [76](#), [142](#)
 - timeprompt, [76](#), [142](#)
- Displays
 - see Views, [46](#)
- event
 - data, [142](#)
 - dir, [143](#)
 - isbreak, [143](#)
 - level, [143](#)
 - number, [144](#)
 - time, [144](#)
- Event View, [79](#)
 - export selection, [88](#)
 - select lines, [86](#)
 - Short commands, [91](#)
 - switch columns on/off, [80](#)
 - Tastenkürzel, [183](#)
- Firmware
 - Loading, [27](#)
- Frame error, [56](#)
- Framing
 - display, [55](#)
 - search, [65](#), [83](#)
- Ledtester, [53](#)
 - show level notation, [53](#)
- LevelFinder, [79](#)
- linestates
 - changed, [145](#)
 - count, [145](#)
- Measure time distances, [90](#)
- MultiProcess architectur, [45](#)
- overrun allowed executions, [223](#)
- Parity
 - display, [55](#)
 - search, [65](#), [83](#)
- Parity error, [56](#)
- Program settings, [50](#)
- Project
 - last opened, [39](#)
 - load, [50](#)
 - save, [38](#), [50](#)
- Projekt, [49](#)
- Protocol
 - autodetection, [31](#)
- protocol
 - bitpause, [227](#)
 - bytepause, [227](#)

INDEX

- databits, [227](#)
 - parity, [228](#)
- Protocol Templates
 - define, [98](#)
 - splitting into datagrams, [100](#)
- Protocol templates
 - Import template, [71](#), [99](#)
 - language syntax, [99](#)
- Protocol View, [93](#)
 - Font, [156](#)
 - selection, [96](#)
 - short keys, [159](#)
- Protocols, [94](#)
- ProtocolView
 - see Protocol View, [93](#)
- Record
 - open, [38](#)
 - pause, [35](#)
 - save, [37](#)
 - start, [35](#)
 - starting automatically, [40](#)
 - stop, [35](#)
- record
 - analyzer, [224](#)
 - buswiring, [224](#)
 - signalnames, [225](#)
 - starttime, [225](#)
- Record mode, [33](#)
 - continuous, [33](#)
 - Fifo, [33](#)
- Region, [175](#)
 - move in view, [176](#)
 - remove, [176](#)
 - rename, [176](#)
 - select, [59](#), [87](#), [169](#)
 - switch on/off, [175](#)
- ring buffer, [33](#)
- Scope display, [164](#)
- sequences
 - get, [146](#)
- Session
 - load, [39](#)
 - see Project, [38](#)
- session, [49](#)
- shared
 - get, [147](#)
 - set, [147](#)
- Signal level
 - displaying, [164](#)
 - search duration, [86](#)
 - search for changes, [85](#)
 - search level state, [82](#)
- Signal line
 - selection, [32](#)
- Signal name
 - rename, [32](#)
- Signal View, [163](#)
 - Cursor, [169](#)
 - Selection, [169](#)
 - short keys, [173](#)
 - size distance, [169](#)
 - undo zooming, [166](#)
 - Zooming view, [166](#)
- Signalmonitor
 - see Signal View, [163](#)
- string
 - dump, [225](#)
- String searching, [62](#)
- Telegram
 - see Protocol, [94](#)
- telegram
 - data, [149](#)
 - datetime, [150](#)
 - dir, [150](#)
 - dump, [151](#)
 - duration, [151](#)
 - geterror, [152](#)
 - isbreak, [152](#)
 - number, [153](#)
 - size, [153](#)
 - string, [153](#)
 - time, [154](#)
- telegrams
 - at, [155](#)
- Time base, [165](#)
- Time distance
 - between data bytes, [61](#)
- Time distances
 - search, [64](#)
- transmission
 - baudrate, [226](#)
- Transmission errors, [62](#)
 - search, [65](#), [83](#)
- User request
 - unsaved data dialog, [35](#)
- Views, [46](#)
 - autoscroll, [46](#)
 - copy, [48](#)
 - Default settings, [48](#)
 - locked, [46](#)
 - synchronize, [45](#)