IF TOOLS
Discover your connection

CleverTerm

Version 2.5.6

www.iftools.com

Contents

1 Introduction

CleverTerm is a program for communication through serial ports like RS232, RS422, RS485. It is especially designed for testing of field-bus applications in the automation technology.

Even though CleverTerm is called a terminal program (it sends inputs to a connected device and displays the answers) it should not be mixed up with programs like HyperTerm or minicom.

The usage for the CleverTerm is more in the field of automation and field-bus applications. Wherever devices exchange data over an asynchronous serial connection/bus - the success of the communication depends on the correct building of the sent telegrams and data frames. This requires an exact knowledge about the used communication protocol (for instance Modbus ASCII/RTU or others). But dealing with the right protocol sequence can give you a lot of trouble.
Sometimes the operating manuals of the devices regarding the provided interface commands are very short or even incorrect. Also the understanding of the underlying protocol often leaves room for misunderstanding simply caused by a lack of examples. At this point there is nothing else left than to test the single commands manually.

Everyone who ever tried to send valid telegrams for a certain field-bus - or in general any communication protocol - knows the difficulties in doing so. These are among others: the input of binary data, the calculation of the correct check sum and above all: The reproduction of a valid telegram content.
The usual terminal programs only offer rudimentary possibilities to edit, correct and repeat such extensive command sequences.

That's where CleverTerm comes into play. CleverTerm provides you with:

- No loss of data by a thread-based, GUI independent device access
- Various input modes to handle also binary data sequences
- Comfortable editing of data sequences before sending

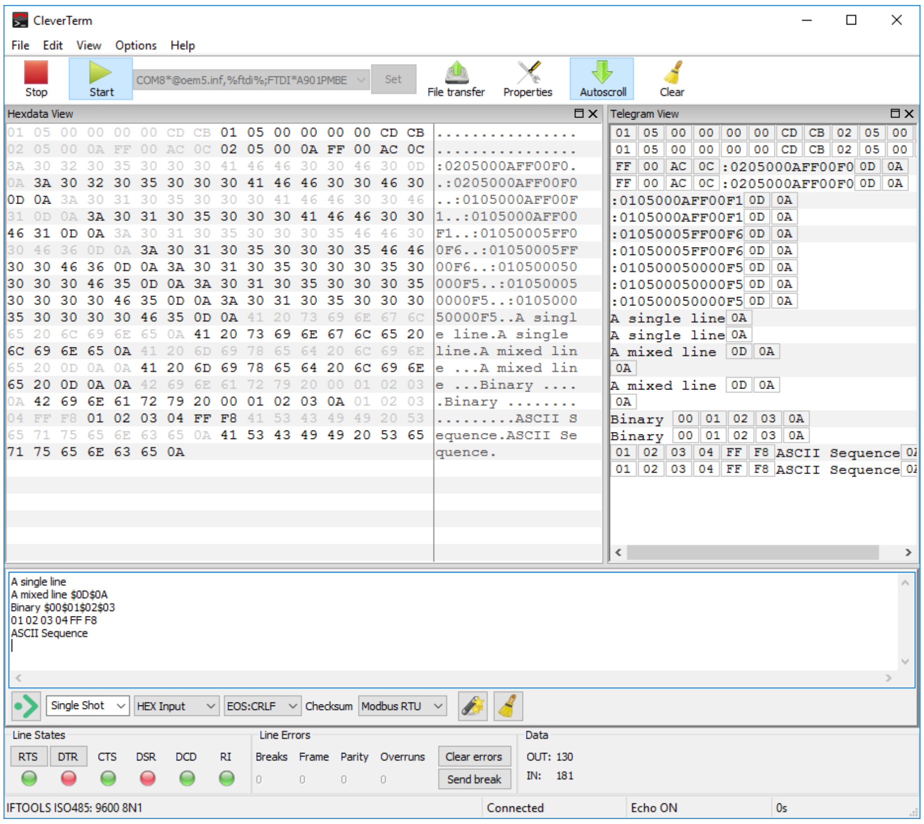
- Input history and easy repeat/modify of former sequences
- Simultaneous data display in hex and telegram view
- Auto-repeat of data sequences in adjustable times
- Built-in check sum generation for Modbus ASCII and RTU
- A port/device selector let you choose your devices by name
- Extendable with individual send/telegram apps scripted in Lua
- A full-featured Lua script editor for interactive scripting
- Reporting frame, parity, break and overrun errors
- Display of all RS232 control line states
- Toggle of RTS/CTS, sending breaks
- Support of non-usual baud rates

CleverTerm is available for Windows® and Linux.

2 Operating

Easy sending of data and commands to the connected device and a visualization of the received answer at the same time allows a clear focus on the essential.

Unlike other terminal programs CleverTerm keeps the sent and received data strictly apart in separated windows. This gives you the advantage that your input sequences are not always interrupted by incoming data and both transmissions are not mixed up in a complete mess.



COM8*@oem5.inf,%ftdi%;FTDI*A901PMBE

Hexdata View

```
01 05 00 00 00 00 CD CB 01 05 00 00 00 00 CD CB  ................
02 05 00 0A FF 00 AC 0C 02 05 00 0A FF 00 AC 0C  ................
3A 30 32 30 35 30 30 30 41 46 46 30 30 46 30 0D  :0205000AFF00F0.
0A 3A 30 32 30 35 30 30 30 41 46 46 30 30 46 30  .:0205000AFF00F0
0D 0A 3A 30 31 30 35 30 30 30 41 46 46 30 30 46  ..:0105000AFF00F
31 0D 0A 3A 30 31 30 35 30 30 30 41 46 46 30 30  1..:0105000AFF00
46 31 0D 0A 3A 30 31 30 35 30 30 30 35 46 46 30  F1..:01050005FF0
30 46 36 0D 0A 3A 30 31 30 35 30 30 30 35 46 46  0F6..:01050005FF
30 30 46 36 0D 0A 3A 30 31 30 35 30 30 30 35 30  00F6..:010500050
30 30 30 46 35 0D 0A 3A 30 31 30 35 30 30 30 35  000F5..:01050005
30 30 30 30 46 35 0D 0A 3A 30 31 30 35 30 30 30  0000F5..:0105000
35 30 30 30 30 46 35 0D 0A 41 20 73 69 6E 67 6C  50000F5..A singl
65 20 6C 69 6E 65 0A 41 20 73 69 6E 67 6C 65 20  e line.A single
6C 69 6E 65 0A 41 20 6D 69 78 65 64 20 6C 69 6E  line.A mixed lin
65 20 0D 0A 0A 41 20 6D 69 78 65 64 20 6C 69 6E  e ...A mixed lin
65 20 0D 0A 0A 42 69 6E 61 72 79 20 00 01 02 03  e ...Binary ....
0A 42 69 6E 61 72 79 20 00 01 02 03 0A 01 02 03  .Binary ........
04 FF F8 01 02 03 04 FF F8 41 53 43 49 49 20 53  .........ASCII S
65 71 75 65 6E 63 65 0A 41 53 43 49 49 20 53 65  equence.ASCII Se
71 75 65 6E 63 65 0A                             quence.
```

Telegram View

```
01 05 00 00 00 00 CD CB 02 05 00
01 05 00 00 00 00 CD CB 02 05 00
FF 00 AC 0C :0205000AFF00F0 0D 0A
FF 00 AC 0C :0205000AFF00F0 0D 0A
:0105000AFF00F1 0D 0A
:0105000AFF00F1 0D 0A
:01050005FF00F6 0D 0A
:01050005FF00F6 0D 0A
:010500050000F5 0D 0A
:010500050000F5 0D 0A
:010500050000F5 0D 0A
A single line 0A
A single line 0A
A mixed line  0D 0A
0A
A mixed line  0D 0A
0A
Binary 00 01 02 03 0A
Binary 00 01 02 03 0A
01 02 03 04 FF F8 ASCII Sequence 0
01 02 03 04 FF F8 ASCII Sequence 0
```

```
A single line
A mixed line $0D$0A
Binary $00$01$02$03
01 02 03 04 FF F8
ASCII Sequence
```

Single Shot | HEX Input | EOS:CRLF | Checksum Modbus RTU

Line States					
RTS	DTR	CTS	DSR	DCD	RI
🟢	🔴	🟢	🔴	🟢	🟢

Line Errors			
Breaks	Frame	Parity	Overruns
0	0	0	0

Clear errors | Send break

Data
OUT: 130
IN: 181

IFTOOLS ISO485: 9600 8N1 | Connected | Echo ON | 0s

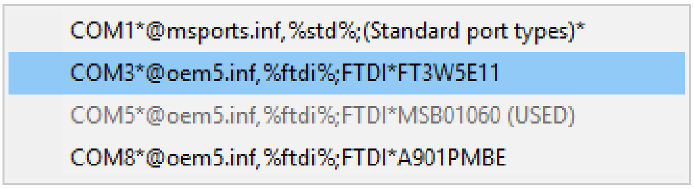
For this the CleverTerm program window is clearly divided in three main parts:

- The receive window shows the data received from the connected device. It is split into a variable hex and telegram view.
- In the transmission (send) window the user enters the data he wants to send to the device. It includes additional controls to change the input format and data composition.
- The device (line) status window with line toggle controls and error display.

The toolbar is - as usual - on top of the program window. But you will notice that there is a port selector unlike all you may have seen so far.

2.1 Start a communication

The first thing you have to do when starting a communication with a connected serial device is to select the right serial port. Sounds not particularly difficult, but imagine you have several serial device connected with your PC (e.g. several USB to RS232/RS485 converters).



```
COM1*@msports.inf,%std%;(Standard port types)*
COM3*@oem5.inf,%ftdi%;FTDI*FT3W5E11
COM5*@oem5.inf,%ftdi%;FTDI*MSB01060 (USED)
COM8*@oem5.inf,%ftdi%;FTDI*A901PMBE
```

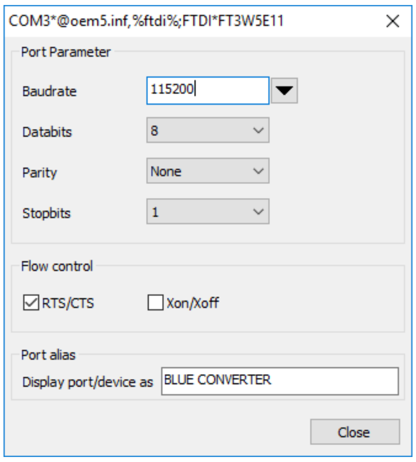
Normally you will see a list of `/dev/ttyUSB...` (Linux) or `COM...` (Windows). Every item represents a certain device, but which is which? A displayed `COM19` or `/dev/ttyUSB7` is not very informative - isn't it?
The CleverTerm port selector takes a new approach. He gathers all available information about the existing serial ports and checks whether each one is occupied by other programs or free. Used ports are shown (it's good to know) but are not selectable.
Every port is displayed with it's port name and (in case of an USB converter) also with product name, vendor and serial number.
The port information are updated whenever you click the port selector. In case you add a new USB to serial converter to your PC, the device will appear automatically in the selection list. If you remove a converter from your PC it will disappear from the list.
Since CleverTerm recognizes removable ports (USB to serial converters) by their serial number, even a converter plugged into another USB socket will remain the same regardless if it becomes another port number.

Setup a serial port

CleverTerm treats the port settings for every serial device separated from each other. That means: You can set a baud rate of 115200 for your first COM port and a different setup for that special USB to Serial converter. Every adjustment is bind to the according port and will be restored automatically when choosing that port again.
Furthermore: You can give every device or port an individual name, like 'My Device' or 'Blue Converter'.
All your settings are stored in a default configuration file and can be saved as an individual CleverTerm project file for later use.

After you have selected the wanted port, click the ’Set’ button on the right to adjust the necessary serial port parameters.

Serial port parameters


COM3*@oem5.inf,%ftdi%;FTDI*FT3W5E11

Port Parameter

Baudrate	115200
Databits	8
Parity	None
Stopbits	1

Flow control

[✓] RTS/CTS [ ] Xon/Xoff

Port alias

Display port/device as: BLUE CONVERTER

- **Baudrate :** CleverTerm supports all baud rates in the range from 1 Baud to 1000000 Baud and the more special rates of 1.5 Mbps, 2 Mbps and 3 Mbps (often used by Profibus). The standard baud rates are selectable from a pre-defined list by clicking the down arrow on the right. Non-standard rates can be input directly by clicking into the baud rate field and using the backspace and numeric keys to overwrite the value with a special one.
  Please note, that not all baud rates are supported by every hardware. This concerns in particular non-standard rates. USB converters with support of non-standard rates are the IFTOOLS USB232CONV, ISO232, ISO485 and ISO485-BOX. Latter comes with support of the higher rates of 1.5 Mbps, 2 Mbps and 3 Mbps.
- **Databits :** Number of the bits which are used for one character. CleverTerm supports word length from 5 to 8 bits.
- **Parity :** Beside the common parity settings none, even, odd parity CleverTerm also allows the selection of Mark and Space which clears or sets the parity permanently.
- **Stopbits :** You can allow 1 or 2 stop bits. The default is one stop bit.
- **RTS/CTS :** Activates the RTS/CTS protocol. This has to be supported by both communication partners and prevents from data losses by buffer overflow in the UART (serial interface chip).
- **Xon/Xoff :** This is a matter of a software protocol where the data flow is controlled by two special data bytes Xon (decimal 17) and Xoff (decimal 19). Because the controls are part of the data stream this protocol cannot prevent from possible data losses. Therefore some manufacturers implement this protocol directly into the hardware like FTDI based converters (e.g. USB232CONV or ISO232/ISO485 from IFTOOLS).
- **Port alias :** Sometimes it would be nice to distinguish several serial ports by giving each one an individual name. Especially if the ports have the same vendor and product naming. Here you can input a name for the current port under which the port is shown in the statusbar afterwards.
  Allowed characters for an alias name are: A-Z, a-z, 0-9, ’–’, ’_’ and the space. Use the backspace to delete a former input.

Start and stop the connection

Here start means to open the connection/port and stop to disconnect it again. After you have finished the port setup, close the port settings dialog and click the green button in the toolbar to activate the connection between the CleverTerm and the serial port.
During an active (opened) connection you are not allowed to change the port setup (for good reasons). The ’Set’ button therefore is disabled.

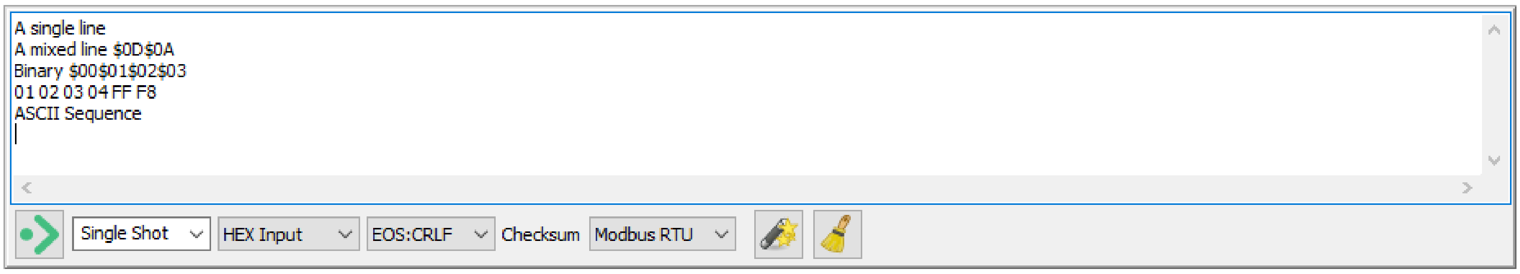
Now - an active connection assumed - you are ready to input some text or data you want to send throughout the given serial port.

2.2 Send data sequences

Sending sequences can be either typed in manually in the transmission window or automated by an individual Lua script. The latter let you extend the CleverTerm by own sending apps (dialogs) using the full power of the Lua script language.
Writing a customized dialog is covered in an separate chapter. Here we focus on the 'normal' way to input any desired data sequence and send it throughout the connected port.

Enter sending data

The input format for the to be transferred data depends on the used protocol. Mostly it is a question about how single bytes are transferred. Simple protocols often limit the allowable data bytes to the range of 'printable' ASCII characters and use the so called control characters (range 0...31) to mark the start and end of a valid data sequence. For example: STX/ETX protocols or the modem command mode specified by Hayes, where each command has to be finished with an <CR> (Carriage Return).
Other protocols are using a pair of the characters 0-9 and A-F to transmit any byte value. The disadvantage: You need always two characters to send one byte which obviously blows the whole data volume up by a factor 2. Examples therefore are the Motorola SRecord (SREC) format and Modbus ASCII.
More sophisticated protocols allow the full range of bytes for their data payload - which means: Sending such a sequence includes characters you cannot simply input via your keyboard, (e.g. the NULL byte or all other ASCII codes above hex 7F).
CleverTerm offers you an intelligent and simple input mode for every case. Beside the trivial ASCII input it also allows you to mix ASCII and binary data, input a raw hex string and operate in a terminal like mode (send each hit key immediately to the device as shown in the following image.



```
A single line
A mixed line $0D$0A
Binary $00$01$02$03
01 02 03 04 FF F8
ASCII Sequence
```

The modes are:

1 ASCII
2 Mixed

3 Hex

4 Terminal

and selected via the Input selector below the input window.

With exception of the Terminal mode all inputs are editable before they will be sent by hit the Enter key.

ASCII Mode

All entered characters are send out only after hitting the Enter key. Until then the input can be corrected in any way.

Hex Mode

The hex mode is a special mode to input raw binary data in an easy way. This becomes very helpful, if you have to communicate via a protocol like Modbus RTU. In Hex Mode you only can input valid hexdigits like 0-9, A-F (upper and lower case) and the space as an optional separator character between each byte. The space is NOT necessary and will be ignored during the transmission. It is just for you to make your hex sentence more readable. The following example shows the input of eight bytes in the range of 0 to 7.
`00 01 02 03 04 05 06 07` In case of a selected checksum (Modbus LTU or Modbus RTU), it will be calculated from the binary data and not from the inputed line. Means: The checksum input are the binary values 0...7.

Mixed Mode

Corresponds to the ASCII Mode but the character '$' gets a special functionality. The two characters directly behind the '$' are interpreted as hexadecimal values. Of course only if they are from the range '0' to '9' or 'A' to 'F'. This is useful to insert any control character or characters which can not be found on the keyboard into an ASCII send string.
As an example the Input of `$FFHello World$00` leads to the sending of a byte with the value 255, followed by the string 'Hello World' and a closing null byte.
To transmit the '$' itself in this mode enter `$$` or its hex value `$24`.

Terminal Mode

The Raw mode is used to transmit every input character directly. This includes also correcting keys like backspace or the arrow keys.

Select an EOS

Whether or which EOS (End of String) character is attached to the entered characters is also decided by the user. You can select the EOS from a list. CleverTerm will add it automatically at the end of the string to be send.

Line repetition

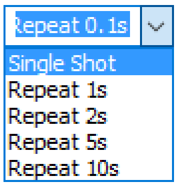
While working with long command strings it is very annoying to enter them again and again for repeating the command in original or slightly changed.
CleverTerm prevents you from this nerving action. Just hit Enter to send the current line (which includes the cursor) again. Of course you can edit each line (except for the terminal mode) before you repeat the sending.
A linefeed only occurs if you edit/send the last line in the input window. If you like to insert a new empty line between two data sequences, just press Shift+Enter.
It doesn't matter where the cursor stays, but if you don't want to wrap the line, place the cursor on the line end or line start.
CleverTerm stores all your input lines in a history file and restore it when you start the program the next time.
The history content is automatically removed when clearing the input window with Ctrl+L or the 'Clear Input button' below.

Checksums

In the current version CleverTerm offers two checksum generators for the LTU and CRC16. Both are mainly used in the communication with Modbus systems. The checksum will automatically appended to the data sequence to be send.
Please note! If you choosed an EOS too, the checksum will be add BEFORE the EOS character(s).
If you need a special checksum and/or EOS, you can use the Lua dialog extension to build your very own telegrams. More on this topic later.

Cyclic transmissions

The default behaviour for a sending sequence is 'Single Shot' which means, after confirming your input with Enter, the data will be transmitted once. But sometimes you want to repeat a transmission sequence automatically in certain intervals. For instance a polling request or a telegram to check periodically for new bus devices.
In such a case just click the down arrow on the right of the 'Single Shot' control. The opening list predefines four options displayed as 'Repeat 1s, 2s, 5s, 10s'. You may think that this is a lean choice. But then every list is always limited.
CleverTerm therefore use a different approach and let you specify your very own interval. For this just click INTO the selected item and type the desired interval in seconds. For not whole-numbered intervals use the dot '.' as a decimal point (not the comma). The four entries are serving simply as examples. CleverTerm understands the following inputs:

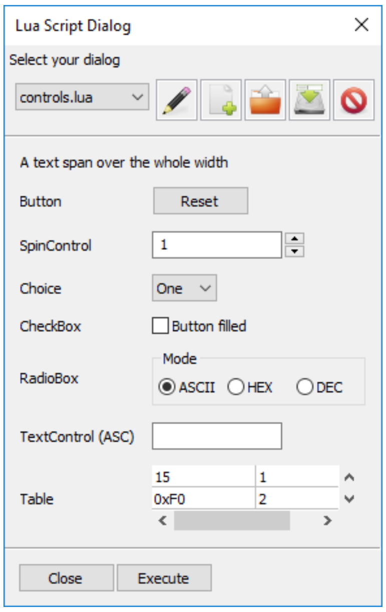


```
Repeat 3s
Repeat 1.25s
Repeat 0.1
0.5
```

The word repeat is optional as well as the unit for seconds. The interval starts when you input/send the next sequence.
To stop the sending loop choose 'Single Shot' again. (The default entries will stay in the list independent of your own repeat intervals).
The repeat/interval mechanism works also for self-written Lua extensions.

Send data via individual dialogs

How you can write your own sending dialogs (or CleverTerm apps) is part of an independent chapter. Here we will just give you an idea what you are able to achieve with this powerful feature.
If you don't have already clicked the button with the magic wand (on the right side of the Single Shot/Repeat control), click it now.



Lua Script Dialog

Select your dialog: controls.lua

A text span over the whole width

Button	Reset
SpinControl	1
Choice	One
CheckBox	Button filled
RadioBox	Mode: (●) ASCII ( ) HEX ( ) DEC
TextControl (ASC)	
Table	15 / 1; 0xF0 / 2

Close | Execute

The Lua Script dialog 'controls.lua' pops up, displaying all available graphical elements (widgets or controls).
The dialog window is divided in three parts. On top is the dialog selector (which shows you all your existing dialogs) and an 'Edit' button to open the according script in an editor.
On the bottom are the 'Close' and 'Execute' buttons. The 'Close' button closes the Lua dialog. The 'Execute' button executes the dialog script and send the result of the script evaluation throughout an active connection.

The important part is between them. The whole content is coded in a Lua script. You can open the script in the editor and change every aspect of the displayed dialog elements interactively. Every modification is applied automatically to the shown dialog as soon as you save your changes in the editor. We will discuss this - as mentioned before - in all details later.
At the moment just play with the shown graphic elements. The 'Reset' button performs a simple preset of a certain number of controls and toggles the accessibility of the 'Choice' control.
The radio box let you switch the input mode of the text control below. ASCII allows you to enter all characters, whereas HEX limit the allowed characters to the hex digits and DEC further more to the decimal numbers.
A click on the 'Execute' button collects a selection of your modifications and send it to the active serial port.

The control.lua is intended in particular to demonstrate the usage of the provided graphic controls (GUI elements or widgets). Beside this script are other dialogs scripts, e.g. to simulate Modbus server requests (modbus.lua), a dialog to send a sequence of random bytes and an example for the table element. You can easily switch between the available dialogs by selecting another one with the top selector.

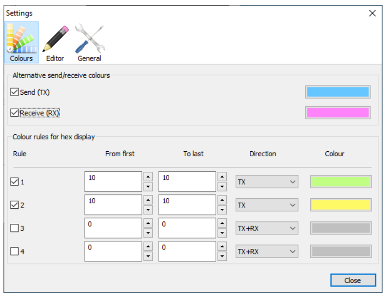
You can easily watch the output of every dialog by starting a connection (press the green Start button in the toolbar) and switch on the internal 'Local Echo Mode'. A local echo means that every sent byte is also 'echoed' in the receiving channel to display the transmitted data in the receive window too. You can enable/disable the local echo in the 'View' menu or simply by pressing Ctrl+E.

2.3 The data reception window

The reception window is split into a Hexdata and Telegram view and displays all data received form the connected device. Since CleverTerm delegates the handling of the transmitted data to an autonomous thread, the data is displayed immediately and will not be disturbed or interrupted by any user interaction.
Every view let you scroll through the received data independent of each other. If you press the 'Autoscroll' button in the toolbar, both views will always show the recently received data.

The Hexdata View

As already indicated in it's name, this view (or window) shows the incoming (received) data bytes in a typical hex dump presentation. Each byte is displayed with it's hex value and - if printable - in it's ASCII representation.
The number of hex values per line depends on the width and the font size of the view. You can change the width by dragging the sash divider to the left or right - and/or by increasing/decreasing the font by turning the mouse wheel while holding the Ctrl key.
You can - of course - hide the complete Hexdata view by clicking the closing cross in it's frame or make it the main window by clicking the maximize frame button. A hidden view can be restored from within the program's 'View' menu.



Alternative send/receive colours

- ✓ Send (TX) 🟦
- ✓ Receive (RX) 🟪

Colour rules for hex display

Rule	From first	To last	Direction	Colour
✓ 1	10	10	TX	🟩
✓ 2	10	10	TX	🟨
☐ 3	0	0	TX+RX	⬜
☐ 4	0	0	TX+RX	⬜

Mark data bytes in different colours

As a special feature the Hexdata view let you mark certain bytes or a range of bytes in a different colour. For this, just click the 'Properties' button in the toolbar and select the 'Colours' tab. Each colour rule is defined by a start and end value and can be assign to both data directions or only TX (send) or RX (receive). If start and end are the same only this byte is displayed in the selected color.
Additional you can select an individual colour for generally all transmitted and received data. These colours are used as text colours whereas the rule colours are for a coloured background.
The colour settings are stored automatically at program end.

The Telegram View

The main purpose of the telegram view is to split the received data in individual byte sequences. It achieves this by using the chosen EOS as a delimiter or end marking. Incoming bytes are displayed as normal characters or - in case the byte isn't printable - as a little box with it's hex value.
As soon as the Telegram View detects an EOS, it will start the next byte in a new line. This kind of data display is mainly addressed to protocols with an telegram end consisting of a CR, LF or a combination of both. Modbus ASCII

is a typical example.
As like the Hexdata view also the font size of the Telegram view can easily be changed by turning the mouse wheel while holding the Ctrl key.

Display the telegram time stamp

A pure software solution will never be able to give you the 'real' time when a data byte arrives. This is an often misunderstood fact and a source of speculations and discussions. Here are only the main facts:

1 Incoming bytes are buffered by the hardware (the UART of a serial port or USB to RS232 converter) and handled by the OS on a later point in time. The time depends on things like interrupt priorities (the serial port as a low priority in comparison with hard disk or network interrupts).

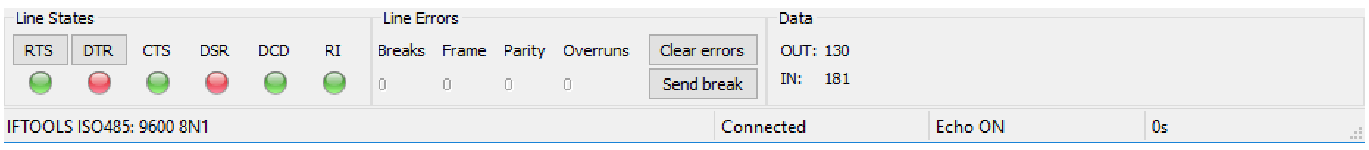
2 The USB to serial converter are mostly polled by the USB sub system which add another not predictable time offset to the byte time stamp.

3 Only a hardware which generates the time stamp immediately and stores both - byte and time - will provide you with time critical information.

Nevertheless the Telegram View offers you to display the time when the first byte of a telegram was received by the program (and NOT when it first and really occurred in the hardware!).
The time stamp is shown with 0.1s resolution. On modern PCs this is an acceptable value since the delay between the first arose of the byte and the processing by the OS are mostly less than 0.1s. But keep in mind, that special circumstances can cause an increasing delay. The time stamp in the Telegram View therefore are above all rough guide values, not more.
You can enable/disable the time stamps in the program's 'view' menu.

2.4 The device status window

In addition to the receipt and transmission windows CleverTerm provides you with a special device status section. This area below the sending input control not only shows the current line states, it also displays transmission flaws like frame and parity error, real breaks and data overruns reported by the connected device.



Line States

RTS	DTR	CTS	DSR	DCD	RI
🟢	🔴	🟢	🔴	🟢	🟢

Line Errors

Breaks	Frame	Parity	Overruns
0	0	0	0

Clear errors
Send break

Data
OUT: 130
IN: 181

IFTOOLS ISO485: 9600 8N1 | Connected | Echo ON | 0s

RTS and DTR are outputs whose level can be switched by the respective key, e.g. to test hardware protocols. The current line states are symbolized like a Led tester. A green Led means a negative signal level (-12V), a red one a positive signal level (+12V). Off Leds indicate a closed or inactive port.
Please note that the line states are only sampled during an active connection.

The number of occurring frame, parity errors and breaks is also an approximate value because most UARTs (and USB converters) only report that such an error arose in general but not for single bytes.
Apart from that it is still a great benefit to know about these errors in a transmission.
Finally the status window shows you the number of received and sent bytes. You can control the counting in the 'Properties' dialog. Especially when the counters have to be reset to zero. Possible settings are:

- Never during an active connection (the default)
- Every time a new file is transmitted
- Every time a new data sequence is sent

2.5 File transfer

CleverTerm offers you two ways to transfer a file content to the connected device. Both require an active (open) connection. Otherwise the according toolbar button and menu entry are disabled.

- Click the 'File transfer' button in the toolbar or press Ctrl+T and select a file from the file dialog
- Drag and drop a file into the program window

The second alternative is especially interesting if you have several files you want to transfer. Instead of selecting a file again and again within a file dialog you can simply pick it up from your desktop or an open file browser.
The file transfer is done by an autonomous running thread and the data is transmitted with the maximum speed provided by the connected device - independent of any other user interaction.
During the file transfer a progress bar with a stop/cancel buttons appears in the right corner of the status bar. The gauge there informs you about the transfer progress. Here you can also stop the transfer by clicking on the red stop button.

2.6 Save your settings

You can save your current settings (connection parameter, window partitioning, program size and position) at any time as an individual configuration file. CleverTerm uses the file extension `cterm` for this files. The extension is registered as a CleverTerm file association with an own file icon during the program installation.
To start the CleverTerm with a saved setup, just double-click a saved `cterm` file or right-click the file icon and select 'Open with CleverTerm'.

2.7 Save received data

CleverTerm records all transmitted data in the background. You can save the received data at any time by pressing Ctrl+S. CleverTerm writes the read

data (IN-coming data) as a binary file. All sent data (displayed when ECHO is on) are discarded and will not appear in the binary file.
The resulting file ONLY contains incoming data!
Please note that a clearing of the received data via the clear button in the toolbar also empties the background data buffer and the data cannot be saved furthermore!

2.8 Short keys

Short keys
of the most important functions

Action	Short key
Online Help to CleverTerm	F1
Opens the currently selected interface	F2
Closes the currently selected interface	F3
Input window: Send the current line to the connected device	Enter
Input window: Insert a new empty line	Shift + Enter
Sends a Break	Ctrl + B
Toogle the DTR line	Ctrl + D
Switch the echo mode in the output window on or off	Ctrl + E
Switches to hex display	Ctrl + H
Clears the output window	Ctrl + L
Opens a file for output over the active interface	Ctrl + O
Toggle the RTS line	Ctrl + R
Saves the received data into a file	Ctrl + S
Switches to ASCII display	Ctrl + T
Saves all settings and closes the program	Ctrl + X
Activates or deactivates the scrolling of the output window	Ctrl + Shift + S

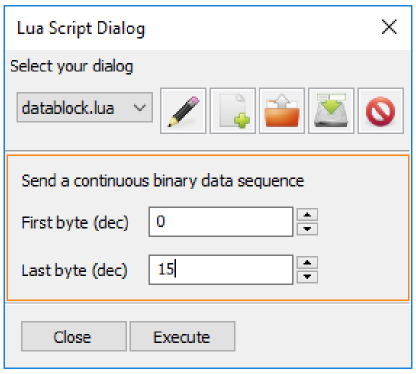
3 Lua dialogs

A unique feature of the CleverTerm program is the option to extend its functionality by own sending dialogs. These range from simple inputs like to generate telegrams with individual checksums to a complete simulation of Modbus master requests. The dialogs are written in Lua, here we cover the necessary details.

Lua is not only one of the fastest scripting languages in the world - because of its nice and simple design it is also very easily to learn. Beside this Lua contains some special concepts which make it the first choice to add the benefits of a scripting language to the CleverTerm program.
It's obvious that we cannot give you a complete introduction in Lua. There are better sources to learn the language in the web. First of all the ⇒ Lua web site and a nice ⇒ Lua tutorial.
Here we will focus on the CleverTerm's Lua extension and how you can use it to write interactive dialogs for your very own application.

3.1 How it works

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 simulate specific transmissions.



Send a continuous binary data sequence
First byte (dec) 0
Last byte (dec) 15

Every scripted Lua dialog (the orange area on the right image) is encapsulated in a special CleverTerm dialog frame, providing you with a script/dialog selector, an edit button (to modify the actually shown dialog) and - regarding the purpose of the CleverTerm - the most important thing - an 'Execute' button.
The latter invokes that part of the Lua code which build the wanted transmission sequence and passes it to the CleverTerm sending mechanism.
According to this every dialog consists of two main parts:

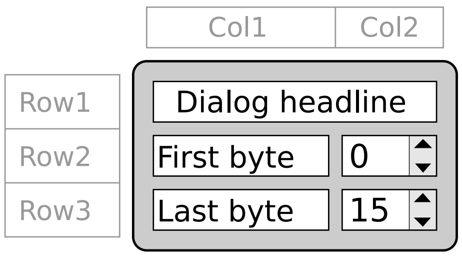
1 The Lua function `dialog` holding the code for the graphical user interface responsible for the orange area in the right image.

2 The Lua function `apply` which is called by the 'Execute' button and returns the transmission sequence as a Lua string.

A dialog can - of course - consist of a lot more functions. But these two above are essential for every functional CleverTerm dialog.

3.2 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 positioning them absolutely by specifying position and size. But this would be cumbersome and laborious.
CleverTerm uses a more elegant way to align your controls more or less in an automatic way.
For this CleverTerm covers the dialog area with an invisible grid. The grid is not limited in width and height and the columns width and rows 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. 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 is expanded as necessary and the width of the column (or height of the row) is adapted 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 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.
The first row is completely occupied by a headline (using an additional col span 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's is the corresponding Lua code:

```lua
function dialog()
    -- the headline label spanned over two columns
    widgets.Label{ name="headline", text="Dialog headline",
                   row=1, col=1, span=2 }
    -- label and first byte input control
    widgets.Label{ name="label1", text="First byte", row=2, col=1 }
    widgets.SpinCtrl{ name="first", row=2, col=2,
                      min=0, max=255, value=0 }
   -- label and last byte input control
   widgets.Label{ name="label2", text="Last byte", row=3, col=1 }
   widgets.SpinCtrl{ name="last", row=3, col=2,
                     min=0, max=255, value=15 }
end
```

Don't worry, if you have some difficulties to understand this script. We will explain all details later. Here it only should give you a first impression how easily a dialog is implemented with a few code lines.

3.3 Add widgets elements to your dialog

All the coding for your dialog has to be done in the function `dialog`. This is the only function which the CleverTerm Lua interpreter executes when it evaluates the GUI (graphical user interface).
For more complicated user interfaces like the Modbus dialog, you can out-source part of the code into other functions. But each of them has to be called at least from within `dialog`.
The supported graphical elements are pooled all in one module. A module in Lua is like a library in other programming languages. You can imagine 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:

```lua
widgets.Button{ PARAMETER... }
```

Now let's start with an empty dialog as described in section 3.6. At first this is nothing else than an empty or even not existing `dialog` function.

```lua
function dialog()
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.

Add a new widget element to the dialog is easy. Just insert the according widgets module function and pass the necessary parameters to it. 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 later. We will cover this in a greater

extend in the next section.
The first widget in our dialog is a label serving as a headline. For this it spans over all - in our case two columns (span=2).

```lua
function dialog()
    widgets.Label{ name="headline", text="Dialog headline",
                   row=1, col=1, span=2 }
end
```

The label position starts in the first column (1) and row (1). The parameter `text` defines the label text, `name` is the name of the widget, here `headline`.
Next we want to add a SpinCtrl to input the first value of the sending byte sequence. For a better usability the SpinCtrl should have a brief text explaining the meaning of the control.

```lua
function dialog()
    widgets.Label{ name="headline", text="Dialog headline",
                   row=1, col=1, span=2 }
    widgets.Label{ name="label1", text="First byte", row=2, col=1 }
    widgets.SpinCtrl{ name="first", row=2, col=2,
                      min=0, max=255, value=0 }
end
```

The briefing label gets the text 'First byte' and is positioned in the second row and left (first) column.
To the right, same row, second column, is the new Spin Control located. The range of valid input values starts with 0 and ends with 255. It's initial value is 0.
You can increase the row or col and see how the widget is new positioned after saving your modifications.
By repeating the last steps for the second SpinCtrl we have finished our little example. Without any comments the code now should look like:

```lua
function dialog()
    widgets.Label{ name="headline", text="Dialog headline",
                   row=1, col=1, span=2 }
    widgets.Label{ name="label1", text="First byte", row=2, col=1 }
    widgets.SpinCtrl{ name="first", row=2, col=2,
                      min=0, max=255, value=0 }
    widgets.Label{ name="label2", text="First byte", row=3, col=1 }
    widgets.SpinCtrl{ name="last", row=3, col=2,
                      min=0, max=255, value=0 }
end
```

All interactive widgets of the dialog are behaving like expected. This means: You can increase or decrease the value of the both Spin Controls in the given range. If you restart the dialog by re-chose it again in the dialog selector, the initial values reappears.
That looks promising - but without getting some output of the dialog, it keeps nevertheless meaningless. The next section explains how you query any input from a certain control or widget and how you handle user interactions like clicks directly.

3.4 Dialog element interaction

Building a dialog by placing the necessary elements is merely the first thing. You can play around with the controls - but how do you query the values and states of each control?
And how do you pass the user inputs to the connected device?
To achieve this you must access every widget individually.

Considering again the `datablock.lua` dialog from last section. When the user clicks the Execute button, a data sequence has to be built starting with the byte value of the first SpinCtrl and ending with the byte value specified in the second SpinCtrl.
To do so we must query the counter value of both spin controls in the `apply` function. (Remember, the `apply` function is automatically called when the user clicks the Execute button).

Accessing individual elements by name

To distinguish the two spin controls - and in general every graphical widget you are using in a dialog - each widget element must have an individual and unique name. The CleverTerm Lua dialog extension uses the name to provide you with a simple method for accessing the properties of each element. This includes getting and setting the input value and/or enable/disable the widget.

Back to our example. Let's say the first spin control is named 'first' and the second as 'last'. The according function to ask a widget for it's value is:

```lua
  value = widgets.GetValue( "NAME" )
```

One of Lua's special features is that you don't have to worry about the resulting type. In most cases Lua handles the requested value (number or string) automatically. Here we expect a number from both spin controls.
Retrieving the first and last byte value is done in two lines (2 and 3 in the following listing). The rest is a little bit of Lua coding. We just make sure that we iterate from the lower to the higher value since the user may have input a 'last' value lower than specified in the first spin control.
The remaining thing is to build the data sequence by adding byte values form first to last and returning the result.

```lua
1  function apply()
2     local first = widgets.GetValue( "first" )
3     local last = widgets.GetValue( "last" )
4     local data = ''
5
6     if first > last then
7       -- the Lua way to swap a pair of values
8       first, last = last, first
9     end
10
11    -- build the resulting Lua string (byte sequence)
12    for i = first,last do data = data..string.char( i ) end
```

```lua

    -- and return it to the CleverTerm program
    return data
end
```

Defining element action handlers

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.
Let's look once again on our little example.
In the dialog the user can select a range of bytes specified by a first and last byte value. If the first value is greater than the last or vice versa, both values are interchanged to ensure an always valid boundary (line 6...9 in the code above).
This is a nice thing to demonstrate the easy swapping of two values with Lua but it doesn't provide a really good user interface. It would be much better, if the respective other value is automatically adapted if necessary.
To achieve this, we must add an action handler (callback) for both spin controls.

The CleverTerm offers a very simple way to write a callback for every widget element. A callback function is defined as:

```lua
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 for. 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_first` and `callback_last`. See the listing below:

```lua
function callback_first( first)
    local last = widgets.GetValue( "last" )
    if tonumber(first) > tonumber(last) then
        widgets.SetValue( "last", first)
    end
end

function callback_last( last)
    local first = widgets.GetValue( "first" )
    if tonumber(last)< tonumber(first) then
        widgets.SetValue( "first", last)
    end
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.
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.
That applies to this example too, since we must compare to values (the first and last) as numbers and not as strings. A comparison as strings follows completely other rules, e.g. the position of the first character in the alphabet.

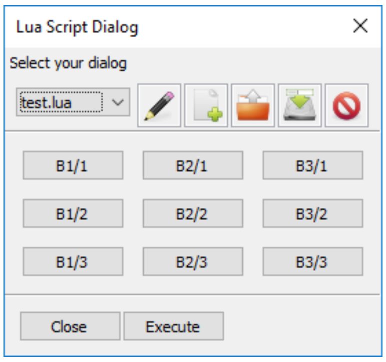
The callback function for the first spin control `callback_first` is called as described with it's current value. To check, if this value is greater than the actual 'last', line 2 queries the current input of the second spin control named 'last'. Afterwards both values are compared as numbers by tell Lua to handle both as numerical values with `tonumber`[1].
Is the first value greater, the spin control of the last value is updated in line 4.
The same approach is used for the last spin control in `callback_last` but only in reverse order.

You can test how it works by open the Lua dialog and choose `datablock.lua`.

3.5 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:

```lua
function dialog()
    widgets.Button{ name="B1/1", label="B1/1", col=1, row=1 }
    widgets.Button{ name="B2/1", label="B2/1", col=2, row=1 }
    widgets.Button{ name="B3/1", label="B3/1", col=3, row=1 }
    widgets.Button{ name="B1/2", label="B1/2", col=1, row=2 }
    ...
end
```

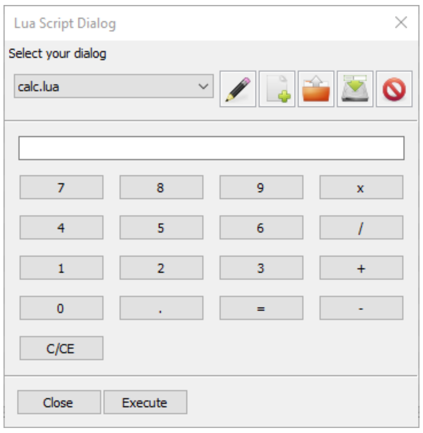


A 3x3 button grid

But instead of writing nine lines of `widgets.Button{...}` code, you can achieve this more easier by creating the buttons in a loop.

```lua
function dialog()
    for row=1,3 do
        for col=1,3 do
            local s = "B"..col.."/"..row
            widgets.Button{ name=s, label=s, row=row, col=col }
        end
    end
end
```

[1]Lua would do the conversion by itself as soon as it must perform a typical calculation like last=last+1



Elements of a calculator arranged in a grid

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 an 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 `calc.lua`. It contains a small but nevertheless full functional calculator. Since it doesn't send any data it's only purpose is to demonstrate the writing of a calculator user interface. (See the calculator picture on the left).
Here is the code for the dialog.

```lua
 1  function dialog()
 2      widgets.TextCtrl{ name="display", col=1, row=1, span=4, fill=true,
 3                        datatype=DEC_NUMBER }
 4      local labels={ "7", "8", "9", "x",
 5                     "4", "5", "6", "/",
 6                     "1", "2", "3", "+",
 7                     "0", ".", "=", "-" }
 8      local i = 1
 9      for y=2,5 do
10          for x=1,4 do
11              widgets.Button{ name=labels[i], label=labels[i], col=x, row=
                    y }
12              i = i + 1
13          end
14      end
15      widgets.Button{ name="Clear", label="C/CE", col=1, row=6 }
16  end
```

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.
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 from1 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.

Advanced callbacks

With one exception (the TextCtrl) the calculator GUI consists of only buttons. When the user clicks on 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 CleverTerm Lua extension acts exactly like this.
First it looks for an individual callback 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

```lua
function callback_all_buttons( name )
end
```

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 `calc.lua` in the editor and take look. The code is well documented.

3.6 Create a new dialog

Starting a new dialog is easy.

1. Open the Lua dialog by clicking the wizard wand button.
2. Click the new document icon (with the green plus)
3. Save the new empty document under a new name by clicking the save button in the editor toolbar or press Ctrl+S.
4. The new (and still empty) dialog appears in the Lua dialog.
5. Start your coding. Your modifications are automatically applied every time you save your work.

Opens the Lua dialog

CleverTerm comes with an integrated and full equipped script editor. The usage of the editor is covered in detail in 8.

How CleverTerm manages your dialogs

CleverTerm looks for new dialogs or script changes exclusively in the `dialogs` folder for local (not shared) user-dependent application data files. The directory location depends on the operating systems.

For Windows it is:
`C:\Users\USERNAME\AppData\Roaming\IFTOOLS\CleverTerm\VERSION\dialogs`

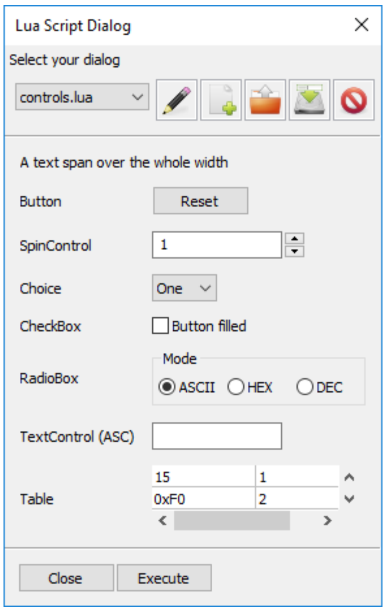
Under Linux:
`~/.IFTOOLS/CleverTerm/VERSION/dialogs`

If you store your dialog scripts under another place, CleverTerm cannot detect changes in your script and therefore won't update the according dialog.

Store your scripts always in the given dialogs folder

The Lua dialog monitors the scripts in the `dialogs` folder only. Every time the script of the currently shown dialog is changed, CleverTerm reloads and executes the script to apply your modification to the actual dialog. This will not happen if the script is stored in a different place!

3.7 Supported Dialog elements or widgets


Lua Script Dialog

Select your dialog: controls.lua

A text span over the whole width

Button: Reset

SpinControl: 1

Choice: One

CheckBox: ☐ Button filled

RadioBox: Mode: ● ASCII ○ HEX ○ DEC

TextControl (ASC):

Table		
	15	1
	0xF0	2

Close Execute

Supported dialog elements

All the elements listed here are part of the CleverTerm Lua widgets extension. Please note that the `widgets` functions work only in a CleverTerm dialog context and cannot be used outside of the CleverTerm program.

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

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:

```lua
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:

```
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.

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
```

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.
- ⊙ datatype : Especially the text input control can be used to handle different data types like binary, decimal and hexadecimal numbers but also normal (ASCII) text or HEX strings. With the |datatype| parameter you can determine the range of allowed input characters and also how to handle the given input value after a request. Valid parameters are:
  `BIN_NUMBER, DEC_NUMBER, HEX_NUMBER, ASCII_STRING, HEX_STRING`
- ⊙ datalen : Specifies the valid length of the input data. For instance: How many characters of an input should be used during the value request.
- ⊙ 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.

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
- ⊙ background : the background colour of the button
- ⊙ foreground : the foreground (text) colour of the button

Please note! The background and foreground colour may not work it the desktop theme uses colour gradient for buttons. This is especially the case with many Linux desktop themes.

Example

```lua
function dialog()
    -- a button on top left2
    widgets.Button{ name="MyButton", label="Press", col=1, row=1 }
end

-- this callback is executed every time the button is clicked
function callback_MyButton()
    -- do something...
end
```

CheckBox

A checkbox is a labeled box which can be either true (checkmark is visible) or false (checkmark is absence).

```lua
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

Example

```lua
function dialog()
    -- a button we want to stretch or shrink
    widgets.Button{ name="MyButton", label="Press", col=1, row=1 }
    -- a checkbox to toggle a fill parameter
    widgets.CheckBox{ name="MyCheckBox", label="Stretch the button",
      col=1, row=2 }
end

-- this callback is executed every time the checkbox is clicked
function callback_MyCheckBox( selection )
    -- query the position of the button widget
    row,col = widgets.GetPosition( "MyButton" )
    -- recreate the button with the new fill parameter
    widgets.Button{ name="MyButton", label="Press",
                    col=col, row=row, fill=(selection=="true") }
end
```

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.

```lua
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

Example

```lua
function dialog()
    -- a button we want to stretch or shrink
    widgets.Button{ name="MyButton", label="Press", col=1, row=1 }
    -- a checkbox to toggle a fill parameter
    widgets.Choice{ name="MyChoice", col=1, row=2,
                    choices={ "Shrink button", "Stretch button"} }
end

-- this callback is executed every time a choice is made
function callback_MyChoice( selection )
    -- query the position of the button widget
    row,col = widgets.GetPosition( "MyButton" )
    -- recreate the button with the new fill parameter
    widgets.Button{ name="MyButton", label="Press",
                    col=col, row=row,
                    fill=(selection=="Stretch button") }
end
```

Gauge

The Gauge widget is the GUI counterpart of a measuring bar and used to indicate a value within a specified min/max range. You can imagine it as a level indicator like the fuel gauge in your car.

```
widgets.Gauge{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
               range=NUM}
```

- ⊗ name : the gauge name as a Lua string.
- ⊗ col : the column index as an integer
- ⊗ row : the row index as an integer
- ⊗ range : the valid range (maximum value) of the gauge
- ⊙ fill : fill the grid cell completely, true or false
- ⊙ span : the number of spanned columns as an integer

In the following example a gauge follows a slider. See 3.7 for the Slider details.

Example

```
function dialog()
    -- a gauge showing the user slider input
    widgets.Gauge{ name="mygauge", range=1000,
                   row=1, col=1, fill=true }
    -- the slider
    widgets.Slider{ name="myslider", min=0, max=1000, value=1,
                    row=2, col=1, fill=true }
end

-- this callback is executed every time the slider was moved
function callback_myslider( val )
    widgets.SetValue( "mygauge", val )
end
```

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 label 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
- ⊙ bold : if set to true, the label is displayed in a bold font, default is false

Example

```
function dialog()
    -- a explaining text for the button
    widgets.Label{ name="MyLabel", text="You can click me",
                   col=1, row=1, bold=true }
    -- the button
    widgets.Button{ name="MyButton", label="Disable me",
                    col=1, row=2 }
end

-- this callback is executed every time the button is clicked
function callback_MyButton()
    -- disable the button
    widgets.Enable( ‘‘MyButton", false )
    -- and adapt the label text
    local col, row = widgets.GetPosition( ‘‘MyLabel" )
    widgets.Label{ name="MyLabel", text="You cannot click me anymore",
                   col=col, row=row }
end
```

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

```
function dialog()
    widgets.Label{ name="MyLabel", text="A label", col=1, row=1 }
    widgets.Button{ name="Button1", label="Press me", col=2, row=1 }
    widgets.Button{ name="Button2", label="Or me", col=3, row=1 }
    -- draw a line over all columns (span=3)
    widgets.Line{ span=3, col=1, row=2 }
end
```

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' (the default) or 'horizontal'.

Example

```lua
function dialog()
    widgets.RadioBox{ name="MyRadioBox", col=1, row=1,
                      choices={ "vertical", "horizontal" } }
end

-- each click changes the orientation of MyRadioBox
function callback_MyRadioBox( selection )
    widgets.RadioBox{ name="MyRadioBox", col=1, row=1,
                      choices={ "vertical", "horizontal" },
                      orientation=selection,
                      value=selection }
end
```

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

```lua
function dialog()
    widgets.Button{ name="MyButton", col=1, row=1,
                    label="Click me", fill=true }
    widgets.RadioBox{ name="MyRadioBox", col=1, row=2,
                      choices={ "Show button", "Hide button" } }
end

-- Show or hide the button according to the RadioBox selection
function callback_MyRadioBox( selection )
    if selection == "Show button" then
        widgets.Button{ name="MyButton", col=1, row=1,
                        label="Click me", fill=true }
    else
        widgets.Spacer{ col=1, row=1 }
    end
end
```

Slider

A Slider is similar to a scrollbar and provides a control with a handle which can be pulled back and forth to vary a value in a given range from min to max.

```lua
widgets.Slider{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
                min=NUM, max=NUM, value=NUM}}
```

- ⊗ name : the Slider 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 1
- ⊙ span : the number of spanned columns as an integer

Example

```lua
function dialog()
    -- a gauge showing the user slider input
    widgets.Gauge{ name="mygauge", range=1000,
                   row=1, col=1, fill=true }
    -- the slider
    widgets.Slider{ name="myslider", min=0, max=1000, value=1,
                    row=2, col=1, fill=true }
end

-- this callback is executed every time the slider was moved
function callback_myslider( val )
    widgets.SetValue( "mygauge", val )
end
```

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}}
```

- ⊗ 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 1
- ⊙ span : the number of spanned columns as an integer

Example

```lua
function dialog()
    widgets.Label{ name="MyLabel", text="Valid numbers are 1...100",
                   col=1, row=1 }
    widgets.SpinCtrl{ name="MySpinCtrl", min=1, max=100,
                      col=1, row=2, value=10 }
end

-- always called when the value in the SpinCtrl was changed
function callback_MySpinCtrl( value )
    local num = tonumber( value )
    if num >= 100 then
        widgets.Label{ name="MyLabel", text="Maximum number reached",
                       col=1, row=1 }
    elseif num <= 1 then
        widgets.Label{ name="MyLabel", text="Minimum number reached",
                       col=1, row=1 }
    else
        widgets.Label{ name="MyLabel", text="Valid numbers are
            1...100",
                       col=1, row=1 }
    end
end
```

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
- ⊙ 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

```lua
function dialog()
    -- the number of columns
    local tcols = 3
     -- the number of rows
    local trows = 20
    -- an empty table holding the default values
    local tvalues = {}

    -- fill the table with incrementing numbers starting with 1
    for i=1,tcols*trows do
        tvalues[i ] = i
    end

    -- special mark of the first and last table entry
    tvalues[ 1 ] = "FIRST"
    tvalues[ #tvalues ] = "LAST"
    -- create a table widget and pass the tvalues table as initial
       values
    -- to initiate all table cells
    widgets.Table{ name="table", col=1, row=1, cols=tcols, rows=trows,
                   preset="FFFF", content=tvalues }
end
```

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, 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 TextCtrl 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) 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.

⊙ 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

```lua
function dialog()
    widgets.RadioBox{ name="MyRadioBox", col=1, row=1, label="Mode",
                      orientation="horizontal",
                  choices={ "ASCII", "HEX", "DEC", "BIN" } }
    widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
                      datatype=ASCII_STRING, fill=true }
end

function callback_MyRadioBox( mode)
    local filter = ASCII_STRING
    if mode== "DEC" then filter = DEC_NUMBER
    elseif mode== "HEX" then filter = HEX_NUMBER
    elseif mode== "BIN" then filter = BIN_NUMBER
    end
    widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
                      datatype=filter, fill=true }
end
```

TextOutput

The `TextCtrl` described above is mainly intended as an one-line input control (to handle different inputs like ASCII, decimal and hexadecimal numbers and so on). The `TextOutput` on the other hand serves as a general way to output any multi-lined text. Lines can be wrapped either by a linefeed (in Lua strings a `\n`) or automatically by the `TextOutput` itself. Furthermore: You can create it with a read-only flag to forbid any text manipulation by the user. All this makes the `TextOutput` ideal to display any text data received by the CleverTerm or as a debug output alternative if you like your debug information right in your dialog.

```
widgets.TextOutput{ name=STRING, col=NUM, row=NUM, fill=BOOL, span=NUM,
                    lines=LINES, readonly=BOOL, wraplines=BOOL,
                    value=STRING }
```

⊗ name : the TextOutput 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

⊙ lines : specifies the height of the control as a number of lines, default is 1.

⊙ readonly : if true the text is not editable by the user, default is false.

⊙ warplines : if true the lines are wrapped at word boundaries or at any other character. This is the default. If you don't like the automatic wrapping, just set it to false.

⊙ value : the initial value, default is an empty string

Example

```lua
function dialog()
    widgets.TextOutput{ name="log", col=1, row=2,
                        fill=true, lines=20,
                        wraplines=false, readonly=true }
end
```

3.8 Functions dealing with widget elements

You have already seen some of this functions in the examples when we were querying an input from a widget or modifying it's value.

The following functions are provided by the CleverTerm program. All these functions expect an 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 `()`.

Enable

This function enables or disables a specific widget or the whole Lua dialog for user interaction (which is the default state of a widget element). A disabled widget appears greyed out and is not accessible by the user.

```
widgets.Enable( NAME, STATUS )
widgets.Enable( STATUS )
```

⊙ NAME : The name of the widget as a Lua string. Without a given name the whole dialog is enabled or disabled.

⊗ STATUS : The new enable state of the specific widget or dialog, true or false

Example

```lua
function dialog()
    widgets.RadioBox{ name="MyRadioBox", col=1, row=1, label="Mode",
                      orientation="horizontal",
                  choices={ "ENABLED", "DISABLED" } }
    widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
                      datatype=ASCII_STRING, fill=true }
end

function callback_MyRadioBox( mode)
    local enable = true
    if mode== "DISABLED" then enable = false
    else enable = true
    end
    widgets.Enable( "MyTextInput", enable )
end
```

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.

```
POSITION = widgets.GetPosition( NAME )
```

⊗ NAME : The name of the widget as a Lua string.

= POSITION : The position as a value pair column, row.

Example

```
function dialog()
    for row=1,4 do
        for col=1,4 do
            local name = "B"..col.."x"..row
          if col == 3 and row == 2 then name="PRESS" end
          widgets.Button{ name=name, label=name, col=col, row=row }
        end
    end
end

function callback_PRESS( control)
    local col,row = widgets.GetPosition( "PRESS" )
    widgets.Label{ name="PRESS", text="Ready", col=col, row=row }
end
```

Note the returning of two variables. This is a special feature of Lua.

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

```
function dialog()
    widgets.Label{ name="MyLabel", text="Input a hex number", col=1,
        row=1 }
    widgets.TextCtrl{ name="MyInput", col=2, row=1, datatype=HEX_NUMBER
        }
end

function apply()
    -- pass the input to the send mechanism
    return widgets.GetValue( "MyInput" )
end
```

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

```lua
function dialog()
    widgets.Button{ name="MyButton", label="Toogle inout field", col=1,
        row=1 }
    widgets.TextCtrl{ name="MyInput", col=1, row=2, fill=true}
end

function callback_MyButton()
    widgets.Enable( "MyInput", widgets.IsEnabled( "MyInput" ) == false
        )
end
```

SetValue

Sets the internal value of the given widget.

```lua
widgets.SetValue( NAME, VALUE )
```

- NAME : The name of the widget.
- VALUE : The new value displayed by the given widget.

Example

```lua
function dialog()
    widgets.Button{ name="MyButton", label="Default value", col=1, row
        =1 }
    widgets.SpinCtrl{ name="MySpinCtrl", col=1, row=2, fill=true}
end

function callback_MyButton()
    widgets.SetValue( "MySpinCtrl", 50 )
end
```

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.

```lua
widgets.SetDialogSize{ width=400, height=500 }
```

- width : The new width of the dialog in pixel.
- height : The new height of the dialog in pixel.

Example

```
function dialog()
    widgets.SetDialogSize{ width="600", height="400" }
    widgets.RadioBox{ name="MyRadioBox", col=1, row=1, label="Mode",
                      orientation="horizontal",
                  choices={ "ENABLED", "DISABLED" } }
    widgets.TextCtrl{ name="MyTextInput", col=1, row=2,
                      datatype=ASCII_STRING, fill=true }
end
```

3.9 Timers

Lua dialogs are pure event driven code. You can define action handlers (callbacks) which are called when a certain user interaction happens. But if you want to execute a function periodically or after a given time the Lua dialog has nothing to offer. Of course you can run any code within the dialog function. However this code will be executed only once and it would be a terrible idea to run a loop in the dialog function because it makes your whole dialog inoperable!

In other words: If the dialog function never returns so does your whole dialog.

Nevertheless it would be a real drawback, if you are limited to control callbacks only. Just think of handling timeouts.

The CleverTerm therefore combines the design of user interaction callbacks with the usage of timers (and which is the reason why timers belong to the widgets modul). With a timer you can executed a callback once or periodically.

A timer is specified by a unique name (for the same reasons as every other widgets control), the time interval and an option one-shot parameter:

```
widgets.Timer{ name=STRING, interval=NUM, oneshot=BOOL }
```

⊗ name : the timer name as a Lua string.

⊗ interval : the timer interval in milliseconds an integer

⊙ oneshot : if true, a timer callback is executed only once, otherwise every given interval

When thinking about timers, most questions concerning how to stop a running timer and how to restart it again. As like with all widgets controls, the timer name is the pivotal point. A timer has no special functions for start/stop, the only function you have to keep in mind, is the function above. If you want to stop a running timer, just call the function again with an interval of zero. To restart the timer, call it again with an interval greater than zero. With the optional oneshot parameter you can either call for a one-time shot or a periodical run (which is the default behaviour). CleverTerm will take care of the rest.

Let us write a simple stop watch dialog (you will find the dialog in the examples). Nevertheless here is the Lua code:

```
function dialog()
    local row = 1
    time = 0
    intval = 10
    widgets.TextCtrl{ name="clock", value=time,
                      row=row, col=1, span=2, fill=true }
    row = row + 1

    widgets.Button{ name="start", label="Start",
                    row=row, col=1, fill=true,
                    background="#00FF00"}

    widgets.Button{ name="stop", label="Stop",
                    row=row, col=2, fill=true,
                    background="#ff0000"}
end

function callback_start()
    widgets.Timer{ name="ticks", interval=intval, oneshot=false }
end

function callback_stop()
    widgets.Timer{ name="ticks", interval=0 }
    time = 0
end

function callback_ticks()
    time = time + intval
    widgets.SetValue( "clock", string.format("%3.2fs", time/1000 ) )
end
```

The dialog is pretty simple - and it shows how easy you can integrate timers in your code. The dialog uses a TextCtrl to show the elapsed time.
Two buttons, a start and a stop button complete the dialog.
The rest of the code contains the necessary callbacks. The first callback is assigned to the start button and initiate a periodical timer named `ticks` with an interval of 10ms as specified in the dialog function. Remember: Lua variables are global by default. Therefore the variable `intval` is visible to the rest of the code.
The second callback is called when the stop button was clicked. To stop the timer, we just initiate a timer with the same name and an interval of 0. Every time a timer is created, CleverTerm checks if a timer with the same name already exists. If so, it deletes the timer and creates a new one if the interval is not 0.
The last callback is assigned to the timer itself (`callback_` + timer name). Since we have a periodical timer, the callback is executed every 10ms. Here we just add the interval time to the global time variable (also defined in the `dialog()` function) and update the TextCtrl with the new time as a floating point value with 2 decimal digits.
As soon as you click the start button, the internal time is reset to zero and then increases in 10ms steps by the timer. A click on the stop button stops the timer (internally CleverTerm removes it to free resources) and the TextCtrl shows the last time - as we expected from a stop watch.

Finally a simple example for one-shot timers. A one-shot timer suits best for an alarm and by the way we can keep our clock metaphor.

```lua
function dialog()
    widgets.SpinCtrl{ name="seconds", value=10, min=1, max=60,
                      row=1, col=1, fill=true }
    widgets.Button{ name="start", label="Start",
                    row=1, col=2, fill=true }
end

function callback_start()
    local secs = widgets.GetValue( "seconds" )
    widgets.Timer{ name="mytimer", interval=secs*1000, oneshot=true }
    widgets.Enable( "seconds", false )
    widgets.Enable( "start", false )
end

function callback_mytimer()
    widgets.Enable( "seconds", true )
    widgets.Enable( "start", true )
end
```

In the example we use a SpinCtrl widget to input a alarm time range of 1 to 10 seconds. A start button activates the alarm. To do so, we create a one-shot timer in the start button callback and disable both, the SpinCtrl and the button to visualize an active alarm.
In the timer callback (we named the timer `alarm`, so it's `callback_alarm()`) we simply enabled the both controls again.
You will find all examples in the selection list of the CleverTerm Lua dialog.

When you timer callback wasn't executed

I wasted some minutes by myself trying to figure out what's wrong with my code when I wrote this examples. So it may be worth to mention: Don't forget to pass the timer `name`, `interval` and `overshot` parameters properly (as key=value assignments). The following timer initiation won't work (although it does not throw an error):

```lua
widgets.Timer{ "alarm", 1000 }
```

4 Intercept transmitted bytes

With the introduction of dialogs it became clear, that there is a lot more we can achieve with Lua. One thing is to process the received but also sent data in your dialog. Which means: The display of data is not longer limited to the CleverTerm hex dump or telegram window. Instead you can intercept every transmitted data byte and make it part of your Lua dialog.

What does this mean for your application?
You can show a response in different formats (extract bits, or combine data bytes to numbers or sub strings), even display the data according your telegram/protocol specifications - or just count all or a certain transmitted byte.

To make this possible, CleverTerm internally passes every received or transmitted data byte including the direction and timestamp to the Lua function `datahook`.

```
function datahook( data, direction, timestamp )
    -- do something...
end
```

If the function does not exist - don't worry! CleverTerm will proceed as usual, showing the data in it's hex dump or telegram window. But if you have already defined this function in your dialog code, your code will be additionally executed every time a data byte is handled by CleverTerm.

One note to the parameter `timestamp`.
It should be clear, that CleverTerm cannot provide precise time stamps for every sent or received data (see also the chapter 2.3). As a pure software solution, the real time when a data byte is transmitted, is falsified/delayed by the OS driver, the schedular, in case of a USB to serial converters, the whole chain of the different linked USB subsystems. Depending on the system load, interrupt priorities and others a software solution cannot tell when a data byte is really on the line. The best you can get is an estimated time. CleverTerm provides timestamps with a 100ms resolution. But even this must be viewed with caution because it's the time, when CleverTerm reads the byte from the driver

queue, NOT the time when the byte arrives on the serial port lines!
Nevertheless it may be a good solution, if the time between the telegrams is bigger, which is often the case when sending requests to a device (bus participant) manually.
BTW: If you have a need for a serial field-bus analyzer for RS232 or RS422/485 bus systems with a time precision of 10ns! see our Serial Analyzers.

Ok, for a start here a code snippet to count the transmitted bytes for each direction. The `datahook` function increments the number of bytes and updates the widget which displays the values. At the moment the function dialog code doesn't matter. Just imagine, both widgets are simple labels.

```lua
local rd = 0
local wr = 0

function datahook( data, direction, timestamp )
    if data.dir == 1 then
        wr = wr + 1
        -- update the widget displaying the number of sent bytes
        widgets.SetValue( "wr", wr )
    else
        rd = rd + 1
        -- update the widget displaying the number of received bytes
        widgets.SetValue( "rd", rd )
    end
end
```

This will work fine as long as you sent small data sequences by hand. And the response also includes only a few bytes. But as soon as you transfer a bigger file the whole program including the dialog becomes inoperable. Why that?

Avoid time consuming code in the datahook

How often the `datahook` function will be executed depends on the data throughput. But even with a moderate baud/bit rate of 9600 baud it means nearly 1000 calls of `datahook`. Even updating a counter widget (showing the receipt bytes) would mean a refresh rate far beyond an operable limit. With other words, CleverTerm becomes partly inoperable. We will learn more about this in the following section.

Good practice for the datahook function

If you are experienced with low level programming, consider the `datahook` as a kind of interrupt handler. And like a interrupt handler which spends more time to execute its code as it takes until the next interrupt occurs, the `datahook` just consumes all computing power without nothing or only a little left to the rest of the program.
So a good practise is to do only as little as possible in the `datahook` function. But coming back to our example of simply calling the received bytes. How can

we do this? In fact, how can we get a smooth refresh rate of the received bytes without wasting to much time?
Here our timers come into play. If you hear about timers the first time, please read section 3.9. We delegate the refresh of the counter widgets (which display the number of received and sent bytes) to a timer callback with a very moderate refresh rate of 10 Hz or every 100ms.
In the `datahook` function we only increment the number of received/sent bytes but nothing more! Under no circumstances will we update a widget! Here is a small code snippet.

```
function dialog()
    local row = 1
    widgets.Label{ name="headline",
        text="Counts the transmitted bytes via datahook",
        row=row, col=1, span=2 }
    row = row + 1
    -- sent data counter
    widgets.Label{ name="label1", text="Sent data",
                   row=row, col=1 }
    widgets.Label{ name="wr", text="00000",
                   row=row, col=2 }
    row = row + 1
    -- received data counter
    widgets.Label{ name="label2", text="Recv data",
                   row=row, col=1 }
    widgets.Label{ name="rd", text="00000",
                   row=row, col=2 }
    row = row + 1
    -- clear all fields
    widgets.Line{ row=row, col=1, span=3, fill=true }
    row = row + 1
    widgets.Button{ name="reset", label="Reset",
                    row=row, col=1, span=3, fill=true }
    -- our display refresh timer with 10 Hz or 100ms
    widgets.Timer{ name="guiupdate", interval=100 }
 end

-- stores the counter data for both directions
local rd = 0
local wr = 0

function callback_reset()
    rd = 0
    wr = 0
    widgets.SetValue( "rd", rd )
    widgets.SetValue( "wr", wr )
end

-- here we update the dialog (every 100ms)
function callback_guiupdate()
    widgets.SetValue( "rd", rd )
    widgets.SetValue( "wr", wr )
end

-- the datahook ONLY counts - not more!
```

```lua
function datahook( data , dir , time )
    if dir == 1 then
        wr = wr + 1
    else
        rd = rd + 1
    end
end
```

As you can see we only increment the counter variables `rd` and `wr` in the `datahook` function. The display refresh is not executed in the function, but will be done in the timer callback `callback_guiupdate`. And only every 100ms, which is fast enough but far from consuming to much time.
You can test it by open the Lua dialog, select the script count-transmitted-data.lua and transfer a file with - lets say - 115200 baud. The CleverTerm and also the dialog stay always operable. For example: You can always reset the counter by clicking the dialog reset button or stop the transmission in the CleverTerm statusbar.

5 Serial port functions

In the last chapter we showed you how to intercept sent and received data within your Lua dialogs. But CleverTerm offers more. You can even send data directly by a button click or react on a certain received data sequence. Further more: CleverTerm allows you to manipulate the modem control lines DTR and RTS or to reset a communication with a break signal.

A few words before we are going into the details!
You may not mistake the `com` module with an option to access the serial port hardware (the UART) directly. Rather these functions provide a layer to address the internal CleverTerm methods to communicate with the connected serial port. Eventually all `com` module functions acts as if you input the data in the input field or click the DTR, RTS or Break button. This ensures, that no race conditions are occuring between a normal input and a runinng Lua dialog.
You may also consider the response time when accessing the serial port as elaborated in chapter **??**.

With the ability to intercept all incoming data (received by CleverTerm) it is also possible to create a kind of automatic response. This may not work with time critical protocols (we discuss the reasons in chapter **??**) but it nevertheless may work well for testing purposes where the reaction/response time does not matter or is above several 100 milliseconds.

Intercepting the sent and received data was already a topic in chapter 4. CleverTerm uses a similar method to handle changes with the connected serial port in a dialog. We explain this later in section 5.2.

5.1 Functions overview

All relevant serial port functions in alphabetic order.

GetBaudrate

Returns the baud rate of the used serial port device.

GetDeviceName

Returns the device name of the serial port currently used (opened) by CleverTerm. Under Windows `COM*`, under Linux typically a `/dev/tty*` device name.

```
NAME = com.GetDeviceName()
```

⇒ NAME : The name of the used and connected serial port, or `nil` if no serial port is opened.

GetFormat

Every asynchronous serial communication transfers the data in a frame of data bits initiated with the start bit, a number of data bits, an optional parity bit and one or more stop bits. Except of the start bit, which is always 1 bit, the number of data bits, the parity and the number of stop bits (1 or 2) are specified as the data format. Like the baud rate the correct data format is mandatory for all communication participants and may be important for the data evaluation in your dialog.
This function returns the data format in the usual abbreviate spelling like ’8N1’, ’7E2’ or similar.

```
FORMAT = com.GetFormat()
```

⇒ FORMAT : The data format as a Lua string like ’8N1’.

IsConnected

With this function you can check if the CleverTerm is connected with a serial port. This is especially useful if some code or widgets in your dialog depends on an opened port. For instance: You may consider to disable GUI elememts with serial port access when the connection is closed. See also the `updatehook` function in section 5.2.

```
BOOL = com.IsConnected()
```

⇒ BOOL : true if CleverTerm is connected with a serial port, false otherwise.

SendBreak

Triggers a break signal. This means: The serial port send line (TxD) goes to a low level for about 1/4 second. A break is normally used to reset a communication or device.

```
com.SendBreak()
```

Example

```lua
function dialog()
    widgets.Button{ name="reset", label="Reset", col=1, row=1 }
end

function callback_reset()
    com.SendBreak()
end
```

SetDtr

Set or clears the DTR line state.

```
com.SetDtr( STATE )
```

⊗ STATE : The new state of the DTR line, true means high, false low.

Example

```lua
function dialog()
    widgets.Button{ name="dtr_on", label="DTR On", col=1, row=1 }
    widgets.Button{ name="dtr_off", label="DTR Off", col=2, row=1 }
end

function callback_dtr_on()
    com.SetDtr( true )
end

function callback_dtr_off()
    com.SetDtr( false )
end
```

SetRts

Set or clears the RTS line state.

```
com.SetRts( STATE )
```

⊗ STATE : The new state of the RTS line, true means high, false low.

Example

```lua
function dialog()
    widgets.Button{ name="dtr_on", label="DTR On", col=1, row=1 }
    widgets.Button{ name="dtr_off", label="DTR Off", col=2, row=1 }
end

function callback_dtr_on()
    com.SetDtr( true )
end
```

```

function callback_dtr_off()
    com.SetDtr( false )
end
```

ToggleDtr

Toggles the DTR line state which means: If the current state is high, it will set the line to low and vice versa. This invokes the same action as if you click the DTR button in the CleverTerm GUI directly.

```
com.ToggleDtr()
```

Example

```
function dialog()
    widgets.Button{name="toggle_dtr", label="Toggle DTR", col=1, row=1}
end

function callback_toggle_dtr()
    com.ToggleDtr()
end
```

ToggleRts

Toggles the RTS line state. Same as with DTR. If the current state is high, it will set the line to low and vice versa. Clicking the RTS button in the CleverTerm GUI is the same.

```
com.ToggleRTS()
```

Example

```
function dialog()
    widgets.Button{name="toggle_rts", label="Toggle RTS", col=1, row=1}
end

function callback_toggle_rts()
    com.ToggleRts()
end
```

WriteData

Passes the given data sequence (any Lua string) to the CleverTerm send mechanism where it is handled like a normal data input. The function does not wait for the complitness and returns immediately. But CleverTerm ensures that even a great chunk of data is transmitted properly.

```
com.WriteData(DATA)
```

⊗ DATA : The data as a Lua string.

Example

```
function dialog()
    widgets.Button{ name="greeting", label="Greeting", col=1, row=1 }
end

function callback_greeting()
    com.WriteData('Hello World')
end
```

5.2 React on serial port status changes

The status of the connected serial port may change independent of the dialog. The user can close the selected port to apply port settings or to connect the CleverTerm with another port/device.
In all this cases it is mandatory for all scripts with serial port access to react accordingly. But how do you or better your dialog script know whether the port status has changed?
CleverTerm uses the same approach like with the `datahook` function. Every time the status of the connected/selected device (serial port) changes the function `updatehook` is called and the cause passed as a event string. The function definition is:

```
function updatehook( event )
    if event == "DEVICE" then
        -- the device state has changed (closed/opened)
    end
end
```

As soon as you add `updatehook` to your script you can undertake all steps which are necessary every time the user opens or closes a serial port.
The function is not limited to update events by the serial port only (therefore the event string). In the future other events may be added but for now `DEVICE` is the only one. The following code lines illustrate the functioning.

```
function dialog()
    local row = 1
    widgets.Label{ name="info", text="",
                   row=row, col=1, fill=true }
    row = row + 1
    widgets.Button{ name="test", label="Send Test",
                    row=row, col=1, fill=true }
    widgets.SetValue( "info", getDeviceInfo() )
end

function getDeviceInfo()
    if com.IsConnected() then
        widgets.Enable( true )
```

```
            return "Device "..com.GetDeviceName().." "..
                   com.GetBaudrate().." "..
                   com.GetFormat()
        else
            widgets.Enable( false )
            return "Device not connected!"
        end
end

function updatehook( event )
    if event == "DEVICE" then
        widgets.SetValue( "info", getDeviceInfo() )
    end
    widgets.Enable( com.IsConnected() )
end

function callback_test()
    com.WriteData( "Hello World!" )
end
```

For more details the dialog `com.lua` in the dialog selection.

6 Time functions

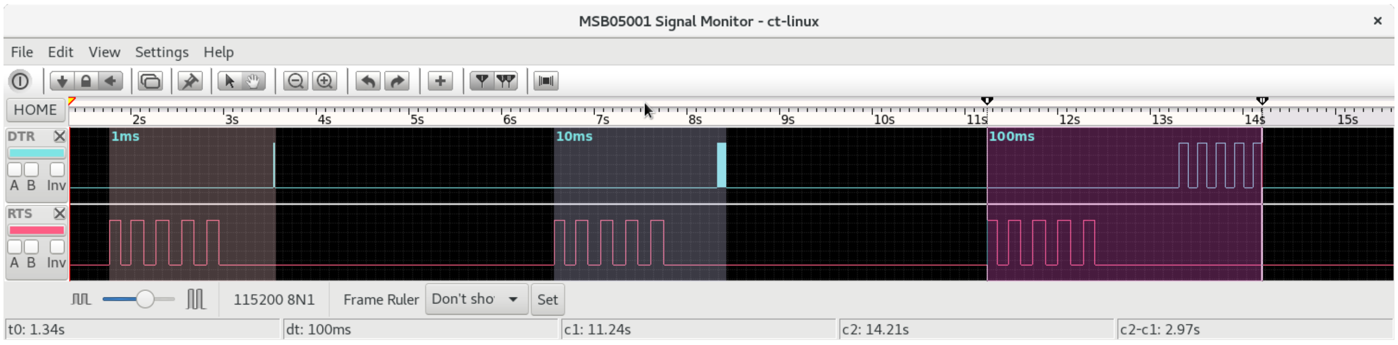
The vanilla Lua language does not provide time functions with a better resolution than one second. The CleverTerm time modul fills the gap and offers a not blocking sleep function, a fine-grained wait and a way to meter the elapsed time in milliseconds.

For periodical actions timers a surely better suited but if you just need a small delay between two actions, for instance sending a sequence before reading the response, these time module functions come into play.
In contrary to timers the provided functions `Sleep` and `Wait` don't need a callback. This makes them ideal when your code demands just a little delay in execution - particular in milliseconds range.

Both functions are called with the wanted time in seconds like:

```
time.Sleep(1.5)
time.Wait(0.01)
```

CleverTerm does not offer two function to confuse you since both functions have important pro and cons. Depending on the application you will use one or another. So what are the differences?
Simply put, `Sleep` does not block the user interaction with CleverTerm but by the costs of additional time delays. Whereas `Wait` has the finer time granularity but blocks! The following picture illustrate this[1]:



115200 8N1

t0: 1.34s | dt: 100ms | c1: 11.24s | c2: 14.21s | c2-c1: 2.97s

[1]Taken on Linux with an IFTOOLS RS232 analyzer MSB-RS232-PLUS

In a CleverTerm dialog we toggle the DTR signal line (the light blue signal) every 1ms, 10ms and 100ms using the following code piece:

```lua
function toggle( dt )
    for i=1,10 do
        com.ToggleDtr()
        time.Wait(dt)
    end
end
```

The RTS signal is switched with:

```lua
function toggle( dt )
    for i=1,10 do
        com.ToggleRts()
        time.Sleep(dt)
    end
end
```

As you can see the period of the RTS signal is almost the same. It only starts to become different when the sleep time is greater than 100ms. The reason: To prevent an inoperable state of the program, the `Sleep` function triggers the internal CleverTerm event loop (which is responsible to handle all user inputs) at least one time and then every further 50ms. The event loop itself adds several 10ms which leads to the estimate 100ms. So even if you call `Sleep` with a time lower than 100ms the function will return first after triggering the CleverTerm event loop to make sure, that you can interact with the program independent of the sleep time.

In contary `Wait` does nothing of that sort. It just stops the execution of the CleverTerm GUI and dialog by calling the appropriate Windows or Linux OS function to pause a process for a specific time. Under Linux `Wait` provides a time resolution of about 1ms, under Windows its at least about 10ms.

You can see the difference in the picture. By the selected time base of 100ms in the analyzer SignalMonitor picture, DTR is almost a single peak when toggling with 1ms (left side), then a small area (middle) before it is displayed with exactly 100ms on the right side.
So remember: Every time you call `Wait` the GUI of CleverTerm and your dialog become inoperable since the OS schedular stops the execution for the given time. The only exception is the data reception. This runs under CleverTerm in a separated task and therefore is executed independent of the main program/-task.
You should limit `Wait` for short delays where the compliance of the delay is mandatory. For longer times (if the duration does not care of additional 100ms) use `Sleep` instead.

Avoid blocking calls and loops

Don't call blocking functions like `time.Wait()` for more than several 100ms, your application otherwise becomes inoperable! This includes also loops within you call `time.Wait()` because the blocking time will be summerized by every single call!
Use `Wait` only for short and precise delays!

6.1 Functions overview

Sleep

Sleeps for a given time in seconds. It precision is OS dependent, but in general not better than 100 milliseconds and not worse than 200 milliseconds.
In contrary to `Wait` the function `Sleep` does not block! Means, even if your dialog sleeps for several seconds, the CleverTerm is processing incoming data and handles user interactions as usual. That goes so far, that you can even intercept received data in the `datahook` function and update GUI elements in your dialog during a 10 second lasting `Sleep(10)` call!
`Sleep` is called like this:

```
time.Sleep(seconds)
```

⊗ seconds : The seconds as a Lua floating-point number, for instance 10ms is input as 0.01

Example

```
function toggle( dt )
    for i=1,10 do
        com.ToggleRts()
        time.Sleep(dt)
    end
end
```

Ticks

Returns the count of milliseconds (on Linux microseconds) from the time when the CleverTerm program was started. We call it ticks. Ticks are a adequate means when you must measure the time between two actions.

```
ticks = time.Ticks()
```

The following example uses the debug output window to display the ticks queried in a loop 100 times. The time lag between every output is a measure how long your computer needs to execute line 7 with the surrounding loop and depends on your OS and - of course - your hardware.

Example

```lua
function dialog()
    widgets.Button{ name="start", label="Ticks?", row=1, col=1 }
end

function callback_start()
    for i=1,100 do
        debug.print( time.Ticks() )
    end
end
```

Wait

Pauses the whole application (also CleverTerm) for a given time in seconds. The time resolution is in the range of some milliseconds (and a lot better than with `Sleep`) but the price is that `Wait` blocks the whole application and not only the dialog. So you better use this function only for small delays and not in long running loops.

```lua
time.Wait(seconds)
```

⊗ seconds : The seconds as a Lua floating-point number. One millisecond is input as 0.001

Example

```lua
function toggle( dt )
    for i=1,10 do
        com.ToggleDtr()
        time.Wait(dt)
    end
end
```

7 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 CleverTerm. Here you learn how to write your own modules which you afterwards can 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 name[1].
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 paths[2] for a Lua file with the given name and executes the code. The result is cached so if you require the same

[1] In this case it works like a name space in other programming languages, e.g. C++
[2] Where Lua looks for modules is part of a later section

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:

```lua
-- 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!

```lua
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 commands, special port name aliases and so on. But they represent not really a module with the meaning of a collection of functions as mentioned before.

7.1 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:

```lua
-- algorithm.lua
return {
    min = function(a,b) if a > b then return b else return a end end,
    max = function(a,b) if a < b then return b else return a end end
}
```

Remember, Lua does not differ between numbers, strings or functions. These are all first-class values[3] 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.

```lua
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:

```lua
local algorithm = require "algorithm"
debug.print( algorithm.min(1,2) ) --> outputs 1
debug.print( algorithm.max(1,2) ) --> outputs 2
```

[3]First-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.

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:

```
 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. It's the same like accessing a table variable or a function in the Lua `string` module.
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.

```lua
function m.set_stack_limit( n )
   if n > 2 and n <= 100 then
      MAX_STACK = n
   end
end

function m.fact( n )
   if n < 0 or n >= MAX_STACK then return nil
   elseif n == 1 then return 1
   else return n * m.fact( n - 1 )
   end
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.

7.2 Module path

This is the path where CleverTerm looks for modules loaded with `require`. Under Linux it is:

```
~/.IFTOOLS/CleverTerm/2.5.6/modules
```

Under Windows the module path is:

```
C:\Users\USERNAME\AppData\Roaming\IFTOOLS\CleverTerm\2.5.6\modules
```

You can check it by yourself. Just use an invalid module name in the require command like:

```lua
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 '/home/jb/.IFTOOLS/CleverTerm/2.5.5/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`[4]. The load command then looks like:

```
algorithm = require "1.0/algorithm"
```

You can even use a variable to switch between the two versions with:

```
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:

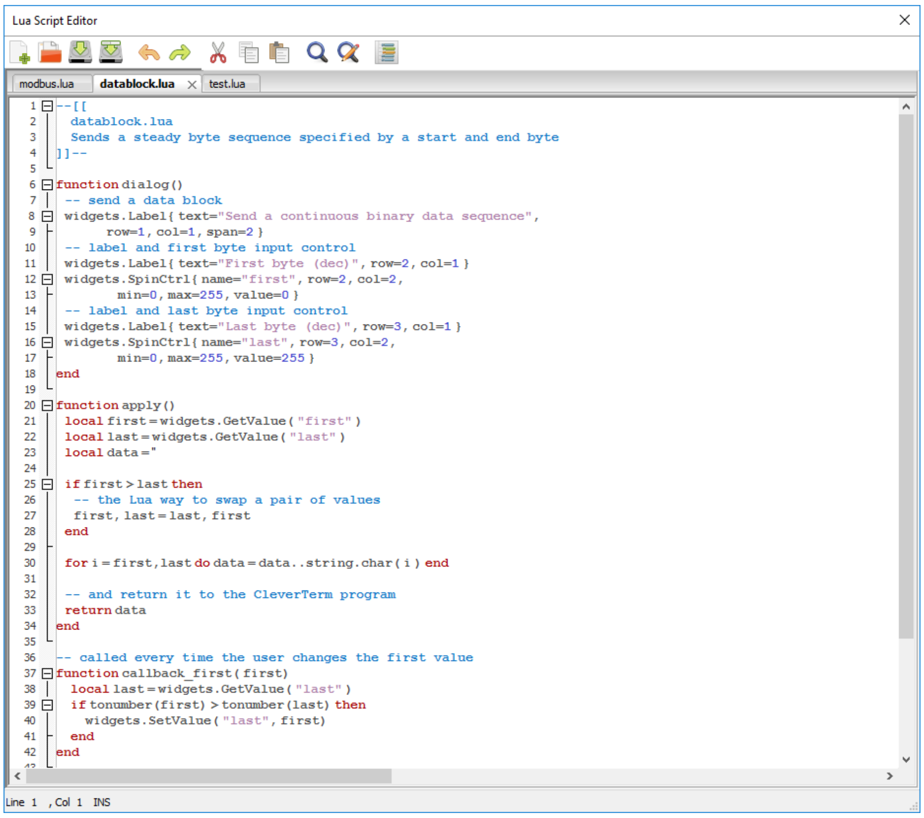
```
-- in a string a backslash must be input as '\\'
local modpath = "1.0\\algorithm"
algorithm = require( modpath )
```

[4]Windows user have backslashes as path separators

8 The editor

The editor integrated into CleverTerm 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.

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



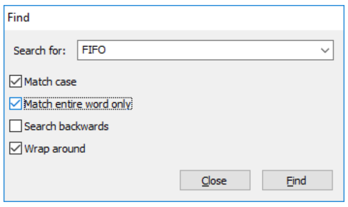
```lua
--[[
  datablock.lua
  Sends a steady byte sequence specified by a start and end byte
]]--

function dialog()
 -- send a data block
 widgets.Label{ text="Send a continuous binary data sequence",
      row=1, col=1, span=2 }
 -- label and first byte input control
 widgets.Label{ text="First byte (dec)", row=2, col=1 }
 widgets.SpinCtrl{ name="first", row=2, col=2,
       min=0, max=255, value=0 }
 -- label and last byte input control
 widgets.Label{ text="Last byte (dec)", row=3, col=1 }
 widgets.SpinCtrl{ name="last", row=3, col=2,
       min=0, max=255, value=255 }
end

function apply()
 local first = widgets.GetValue( "first" )
 local last = widgets.GetValue( "last" )
 local data = "

 if first > last then
  -- the Lua way to swap a pair of values
  first, last = last, first
 end

 for i = first,last do data = data..string.char( i ) end

 -- and return it to the CleverTerm program
 return data
end

-- called every time the user changes the first value
function callback_first( first)
  local last = widgets.GetValue( "last" )
  if tonumber(first) > tonumber(last) then
    widgets.SetValue( "last", first)
  end
end
```

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.
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 CleverTerm program) will never ends without informing you about open changes and asking how you will proceed.
If you close the editor by clicking the close symbol in the editor window frame, the editor is only hidden but all its content is still there. The editor definitely ends not before the CleverTerm was finished.

8.1 Interactive coding

As an firmly integrated part of the CleverTerm program the editor is intended to interact with the Lua dialog automatically. In particular to trigger the updating or redrawing of a dialog when saving the according Lua script. As simple as it is, it makes the design of new a dialog an amazing experience.
Add a new widget element in your code and press CTRL+S and - voila - the ready to use widget appears in the dialog as if by magic.
The same is true when you start with a new dialog script. As soon as you save the Lua script as a new file name, the new dialog appears under this name in the dialog frame.

8.2 Find


Find

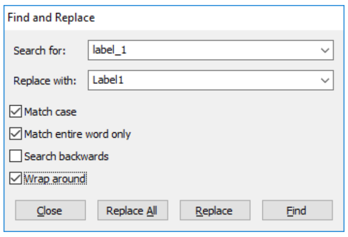
Search for: FIFO

☑ Match case
☑ Match entire word only
☐ Search backwards
☑ Wrap around

Find dialog

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 CleverTerm 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.

8.3 Find and replace


Find and Replace

Search for: label_1
Replace with: Label1

☑ Match case
☑ Match entire word only
☐ Search backwards
☑ Wrap around

Find & replace dialog

It's the usual business of a programmer to rename a certain variable after re-thinking the meaning of it. The CleverTerm 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.

8.4 Code folding

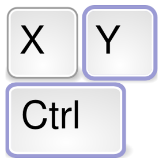
Code folding is a nice feature when your script consists of a lot of functions or other code blocks like tables. Activated in the toolbar it collapse every function into their very first code line. In case of a function, it's the function definition or name. Tables collapse into the first line of the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

8.5 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.



Short keys of the most important functions

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	Ctrl + L
Create a new script/document	Ctrl + N
Load a script file into the editor	Ctrl + O
Save the current script/document and trigger a dialog update	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

A

ASCII character table

ASCII (American Standard Code for Information Interchange) is a form for the character coding, which, coming from teletype machines, now is established as the standard code for character representation.

The first 32 characters of the ASCII code (hex 00 to 1F) are non printable signs, reserved for control purposes. The main control characters are line feed or carriage return. They are used with devices which need the ASCII code for control purposes as printer or terminals. Their definiton is caused for historic reasons.

Code hex 20 is the blank and hex 7F is a special character which is used for deleting.

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEK	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	SP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	∧	_
6...	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	∼	DEL

The upper table regards only 7 bits per byte, the first 128 characters. Extentions of the ASCII code use the next 128 characters for national language codings or graphical signs. They are very different in usage. So we will limit the description to the standard 7 bit version.