1 Explore RS232 with Lua

Imagine you just want to send and read data through a serial port for testing purposes. A scripting language will offer you a lot of benefits in case of periodical or repetitive tests. Luactb combines the facilities of the iftools library with the amazing script language of Lua. This chapter shows you how to use the Luactb shell to communicate interactively with a serial device and run scripts automatically.

luactb is a stand alone Lua interpreter based on the Lua sources of version 5.1.4 (see Lua license and copyright). It comnes with built-in functions to make the communication with RS232/RS485 devices as easy as possible. For this, **luactb** integrates the iftools library and adds a lot of useful extensions for Modbus, time measuring and keyboard query (kbhit). And all this comes as a single executable for Linux or Windows with a size less than 1 MByte.
By the use of the iftools library **luactb** allows you to search and list all available serial ports, accessing virtual serial ports simply by passing one or more of the USB specific informations (don't worry about changing COM port numbers), sending Modbus RTU or ASCII sequences with automatically generated checksums and a lot more.

1.1 Where can I get the newest luactb

luactb is free software and available from the IFTOOLS website. For updates and news take a look at: https://iftools.com/tools/luactb/index.en.php. There are no further requirements. Just download the archive file, unpack it in any directory and you are ready to start.

1.2 The first steps

In the following I will introduce **luactb** with some small examples so you will get a feeling for the usage and - of course - the chances **luactb** will offer you for your own applications. Later we will go deeper in more special cases.

If you don't have any experience with Lua, don't worry!
Lua is one of the fastest scripting languages in the world. Because of it's small and simple design it's also easy to learn. You will find some links about Lua at the end of this chapter.

Ok, lets go. **luactb** is a command line tool, so you have to start first a terminal or DOS command shell, change to the directory where the program is stored and input **luactb**. The interpreter greets you with:

```
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio, extended by IFTOOLS
>
```

You can always exit the interpreter by pressing Ctrl+C.
But I suggest that we stay a little bit longer in the **luactb** and take a look about the available serial ports on your PC[1].

Query the available serial ports

For this, just input the following lines (without the leading line numbers, they serve only as reference for my later explanation):

```
1  ports=ctb.ListPorts()
2  for i=1,#ports do print(ports[i]) end
```

Variables in Lua are typeless. The first line calls the ListPort function of the built-in ctb module and returns all port informations as an array (or in the Lua way spoken as a table). Module functions are called by the module name, a dot and the function name.
The second line iterates over all entries in the table and print out each record.
Depending on the existing serial ports you will get something like this:

```
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio, extended by IFTOOLS
> ports=ctb.ListPorts()
> for i=1,#ports do print(ports[i]) end
COM1    :          (Standard port types)       FREE
COM24   0403:6001 FTDI     FT3W5E11            FREE
```

The example above is taken from a Windows computer. On Linux the device names differ but the rest of the information is the same. The ListPorts function returns for each port the according device (COM port), the vendor and product id, the name of the vendor, the serial number and the status of the port. The information are separated by a tab character.
A few words to the second line: a Lua array (or table) always starts with index 1 (in contrary to C for instance). The '#' means the size operator and returns the count of entries in the table. The following line outputs the number of items listed in the ports table.

[1] If you have a ready installed USB to RS232 converter it would be a good idea, to connect it with your computer now.

```
1  print(#ports)
```

Line 2 iterates over all items in the ports table beginning with the first one (1) to the last (the available items requested by `#ports` and printout every item (serial port) in a line.
An alternative way to iterate over all ports is to use the Lua table iterator `ipairs` which returns the pair of the next table index and table content every turn.

```
1  for i,v in ipairs( ctb.ListPorts() ) do print(v) end
```

Open a serial port

Next we will open `COM1` and send the usual 'Hello world'. Linux user replace the port name with `/dev/ttyS0`. You can input the following example in your active interpreter but please note that not every connected device is pleased with our friendly message.

```
1  com = SerialPort.new()
2  if com:Open( "COM1", 115200, "8N1" ) > 0 then
3      com:Write( "Hello world" )
4  end
```

The code looks reasonably clear at first sight (I hope so). Nevertheless there are some details which require an explanation.
Before you can access a serial port you must create a handler or object which represent a serial port of your system. Line 1 creates a new serial port object and assign it to the variable `com`.

In the next line we connect the object with a certain port by opening the device, here `COM1`. You may notice the colon between the variable and the method 'Open'. The function 'Open' as well as 'Write' belong to the SerialPort object. This belonging is expressed by the colon between the object and function name and must not mixed up with a module membership.

The open call requires at least two parameters, the name of the serial port device (COM1) and the baud rate. The third parameter is the optional data format or protocol. The default is '8N1'. The name and the protocol must be passed as a text string. A string in Lua is defined as a character sentence between two quotation marks.

The Open call returns 1 when successful, and 0 otherwise. Line 3 will only be executed when the port is open for access. The Write function itself simply expect a string to send through the serial port. Because Lua strings are allowed to contain any data you can just as simple send binary data.

Port names in a plattform independent way

Remember that I told the Linux users to use `/dev/ttyS0` instead of `COM1`. As soon as you try to use the same script on different OS you may encounter a problem.
Luckily **`luactb`** starts from the beginning as a plattform independent tool (this isn't really surprising, is it?). That means: All scripts executed by the interpreter are the same for Linux and Windows.
Linux names the serial ports as `/dev/ttySx` or `/dev/ttyUSBx` whereas Windows uses the term `COMx` (x is the port number, starting with 0 under Linux and 1 within Windows).
You can of course - as we have seen - use the system specific naming. But there is a more elegant way. If you like to open the first serial port (`COM1` or `/dev/ttyS0` the built-in ctb module offers you a method to query the right port name for both OS.

```
1  com = SerialPort.new()
2  if com:Open( ctb.GetPortName( 1 ), 115200, "8N1" ) > 0 then
3  then
4      -- do something
5  end
```

The function `GetPortName(portno)` returns the 'native' device name on both operating systems.
This will work very well with Windows systems but under Linux there is another pitfall. Linux distinguish between 'normal' serial ports (named as `/dev/ttySx`) and 'virtual' ports. The latter where created on the fly as soon as you connect an USB to RS232 converter with your PC. Linux named them as `/dev/ttyUSBx`.
For now just memorize that GetPortName returns only 'normal' serial ports under Linux, whereas it always return the relating port within Windows (if it exist).
Later I will show you a more convenient way to access virtual com ports on both systems without to worry about the right port number.

Send (write) data

In normal case you just want to send a given string or binary data sequence throughout a serial port and it doesn't matter how long it takes until all data was sent (which mostly always is a matter of split seconds).
Assumed you have an already opened com port (as described in the listings above), the sending of any data will be simply done with:

```
1  com:Writev( "\002Data between STX and ETX\003")
```

Here we create a text sequence enclosed by a STX (Start of Text) and ETX (End of Text) control bytes. Both are non-printable characters and therefore you have to input their decimal value instead. Lua offers a simple way to mix

any binary value with normal ASCII characters. Just input the decimal value of the desired byte headed by a backslash. The decimal value must always entered as a three-digit value. For instance: `"\016"` means the new line (LF), `"\009"` the tab character.
With Lua's concatenation operator .. it's especially easy to compose a string from several parts like:

```
1  eos = "\r\n"
2  com:Write( "some command"..eos)
```

`eos` is a string variable containing a CRLF sequence (`"\r\n"` is a short form for `"\013\010"`) which is appended to the sending sequence with the concatenation operator.

By default the function `Write` uses an infinite timeout and always tries to send the complete given data before it returns. So it might wait or blocks for a longer time in case the recipient cannot process the sent data as fast as needed.
A blocking behaviour seems fittingly when working interactively, but in other cases it sometimes is of more importance that a program or script doesn't block at all. For instance: When you have to send a lot of data and want to do other things during the transmission.

```
1  -- a data sequence of 1000000 hex FF bytes
2  local data = string.rep( "\255", 1000000 )
3  local left = #data
4  while left > 0 do
5      -- non-blocking, timeout is 0s
6      local wr = com:Write( data:sub(-left), 0 )
7      -- device doesn't accept more data at the moment
8      if wr == 0 then time.pause( 0.1 ) end
9      -- error check
10     if wr < 0 then break end
11     left = left - wr
12     -- do something, e.g. show a progress counter
13 end
```

By adding a timeout of 0 (seconds) `Write` now stops as soon as the receiver doesn't accept more data and returns with the numbers of written data.
In the example above the number of written data is subtracted from the remaining (`left`) data on line 11. Line 6 uses the `left` variable to address the remaining data by indexing it from behind using an negative index.
Negative indexing is one of the great features (beside many others) Lua provides you with. The reserve indexing starts with -1 (the last byte). Here a short example:

```
1  local data = "hello world again!"
2  print( data:sub( -6 ) ) --> "again!"
```

In many other cases it is often sufficient, just to take a certain send time limit into account (reflecting the baud rate and data length) and pass that time as a timeout.

```
1 com:Write( "some command", 1.0)
```

If the sentence couldn't send within that given time it maybe won't even work with a greater timeout and an appropriate error handling can be considered when the call returns.

Receive (read) data

Read data seems a trivial issue (at a first glimpse), but before you continue stop for a moment and answer the following questions.

1. Do you know the expected bytes you have to read?
2. What do you like to do when there isn't any data at all (no response)?

If you cannot answer to the first question you may read too few data of the response or - just as bad - read ahead bytes of a possible second response. And afterwards it's difficult to pick the received data apart.
The second case may also block your script when no data is available.

Read lines

luactb provides you (beside a raw read functionality) with an easy method which should sufficient for the most applications without getting a headache.

```
1 answer = com:GetLine( "\r\n" )
```

`GetLine(delimiter)` reads all available data until the given delimiter is found and returns the whole data including the delimiter. This covers all protocols with a specified EOS (End Of String). But it's obvious that the call will block when the EOS is missed or no data was received at all. A second optional parameter ensures that the reading call returns always after a given - so called - time out. Here, for instance, after 5 seconds.

```
1 answer = com:GetLine( "\r\n", 5.0 )
```

In little or simple scripts you may not pay attention about a timeout (and the relating incomplete or unavailable data). But serious applications requires a need for a correct error handling. Every time you call the `GetLine` with a timeout you have to check the correctness of the delivered data by yourself because you cannot be sure, that the function returns with a complete response or just after a timeout expiration.
A simple method would be to check the response for the specified delimiter as shown in the next example:

```
answer = com:GetLine( "\r\n", 5.0 )
if answer:find( "\r\n" ) then
    -- correct answer
else
    -- a timeout occurred
end
```

Read pure data

The `GetLine` function is not suitable for handling raw data without any defined delimiter or end-of-string character (EOS). **luactb** therefor provides you with the more applicable `read` function, which offers you blocking and non-blocking reading of data of any length. The simplest call of `Read` is:

```
answer = com:Read( 10 )
```

which just will read the next ten bytes arriving on the serial port. As long as there isn't enough data, the function won't return.
You can limit the waiting time to any value between 0 seconds (non-blocking) and (by leaving it out as in the example above) infinity. In most cases you will either use the non-blocking behaviour or a well considered time which fits your program behaviour best. For instance:

```
local timeout = 0.5
com:Write( "Some command sequence" )
answer = com:Read( 1000, timeout )
```

You want to read the answer of a certain command sent to a device. Since you know, that the device must respond within a reasonable time (here 0.5 seconds), you can restrict the maximum time to avoid unnecessary time consumption. The count of requested bytes doesn't matter as long as it is greater as the expected response (otherwise you might get less data as you want). `Read` will always returns either when the number of demanded bytes is reached or the passed timeout is expired.

Now imagine you want to write all bytes received by a serial port into a file. You neither know the amount of data nor how long it will be take until all bytes are transferred. Since there is not 'end' condition (EOS) in the data stream (except for - perhaps - the change in the line states when the sender close its part of the connection), the only thing you can be sure of it is:
When no data arrives for a certain time, the sender has stopped sending data. In the meantime you have to read all bytes and write them into a file. For a good performance the reading should be done without any delays. And it would be nice to show the user some kind of progress information - wouldn't it?

Before we start into coding let us see how you can display a counting number

in a single line. It is an often need task and Lua has a simple solution for it.
In the example below we iterate from 1 to 100 and output the counter. A little delay of 0.1 second helps to show each step.

```
1  for i=1,100 do
2      io.write("\r"..i)
3      io.flush()
4      time.pause( 0.1 )
5  end
```

Since the Lua `print` function always add a new line we cannot use it. But Lua provides a mighty io module which offers all kind of input/output facilities you know from other programming languages.
In line 2 we place the output cursor with a carriage return (CR) at the beginning of the line and append the current counter value with the Lua concatenation operator. `io.write` sends the data to the standard output channel.
In normal cases outputs will be cached by the io module (or the OS part of it). So we must flush the data explicitly (line 3).
Ok, back to our actual purpose. To keep the code simple, we forgo any error handling (keep in mind, that this is a bad idea...). Here we go:

```
1  com = SerialPort.new()
2  file = io.open( "log.bin", "wb" )
3  if com:Open( "/dev/ttyS0", 115200 ) then
4     -- wait for the first byte but not longer as 20s
5     local data = com:Read( 1, 20.0 )
6     if #data == 1 then
7       local n = 1
8       file:write( data )
9       -- now start reading all bytes until no further arrives for 5s
10      while true do
11         data = com:Read( 1000, 0.0 )
12         if #data == 0 then
13           data = com:Read( 1000, 5.0 )
14           if #data == 0 then break end
15         end
16         if #data > 0 then
17           file:write( data )
18           n = n + #data
19           io.write( "\rRead bytes: "..n )
20           io.flush()
21         end
22      end
23    end
24 end
```

You will notice that we use the `Read` function three times. At the first call at line 5 the script waits for the arriving of the first byte and gives up after 20 seconds. As soon as a byte was received (and after writing the byte into the given file) the program starts a loop reading all available bytes (but not more than 1000).

This time the timeout is set to 0 seconds and every `Read` call returns immediately and independent of the amount of received bytes.
A returning value of 0 must be treated in particular. A zero result indicates, that the port (or better the according driver) 'actually' has no available bytes. But this doesn't mean, that the port has already received all data! By reading out the port faster than the sender is able to provide data, the `Read` function will obviously return zero bytes at times.
In such a case let the sender have some time to catch up as we do in line 13. When there is still no more data to read, the sender has quit and so can our script.
There are - of course - better methods to test for a starting and ending transmission. For instance: You can check the DSR and DTR state, but here the focus was laid on the reading itself.

Close a port

In Lua you don't have to worry about unused objects. Lua comes with a garbage collection and frees all variables and objects which are no longer reachable. This is valid also for a serial port object.
When a script ends all active serial port handlers (objects) are closed automatically. This will also apply for local objects in a function.
Nevertheless sometimes you have to close a port explicitly, i.e. if you like to reopen it with another protocol.

```
1  if com:Open( ctb.GetPortName( 1 ), 115200, "8N1" ) > 0 then
2      -- do something
3      com:Close()
4  end
5  if com:Open( ctb.GetPortName( 1 ), 115200, "7E1" ) > 0 then
6      -- do more
7  end
```

Without `com:Close()` in line 3 the second opening fails because the port remains still open.

1.3 Access a virtual com port

USB to RS232 converters are handled by the OS (Linux and Windows) as so called 'virtual COM ports'. Every time such a converter is connected with the PC, the OS assigns a prearranged port number to it. Unfortunately the number varies from PC to PC and also depends on previously connected converters.
You can - of course - specify a certain number for the converter in the property dialog of the virtual COM port with Windows. Linux users has to add a 'udev' rule to accomplish the same. And you have to do this on every PC you like to use with the converter. But both is annoying - isn't it?

When I introduced the `GetPortName(portno)` function I promised you a more convenient way to access a serial port without having to know the relating COM port number. All it requires is an unique description of the connected device. Something which differs from device to device. Often only the vendor name is sufficient (if you only use one device from this supplier), at other times you need the serial number of the device as an unique characteristic.
The responsible function `ctb.FindPort{...}` therefore accepts five different attributes for the detection:

```
1 ctb.FindPort{ product="",vendor="", serial="", pid="", vid="" }
```

'Empty' parameters are obsolete. You can simply remove them.
`FindPort` scans all available ports and returns the first one which fits all given attributes.
Please note! The function doesn't distinguish between a free or used port. It always returns the relating native port name.

Now let us put the function into practise. First at all you need some basic informations about your connected devices. Remember the `ctb.ListPorts()` at the beginning of this chapter. It will give you a list of all available serial ports with their attributes.
Now pick that information that describes the device you like to access and pass it to the `FindPort{}` function. In the example below we open a USB to RS232 converter from FTDI with the serial number FT3W5E11:

```
1 com = SerialPort.new()
2 if com:Open( ctb.FindPort{ vendor="FTDI", serial="FT3W5E11" }, 115200 ) > 0
3 then
4     -- do something
5 end
```

This code will work on all supported operating systems, independent of the port number assigned by the OS.

1.4 Handshake

The data transmission through an opened port runs without handshake unless you pass the wanted kind of flow control to the `Open` function. **`luactb`** supports RTS/CTS and XON/XOFF. The following command opens a connection with an active RTS/CTS handshake.

```
1 com:Open( ctb.GetPortName( 1 ), 115200, '8N1', 'rtscts' )
```

And here the same with XON/XOFF:

```
1 com:Open( ctb.GetPortName( 1 ), 115200, '8N1', 'xonxoff' )
```

The handshake protocol is simply passed as a string to the `Open` function.

1.5 Toggle RTS and DTR

RTS and DTR are the only control lines you can set. Both lines are intended to be part of a hardware flow control. Sometimes however they are misused to drive a special hardware or signaling a certain state independent of the data transmission lines.
The RTS and DTR line states are switched to a logical 1 state with:

```
1 com:SetLineState( ctb.RTS, ctb.DTR )
```

and set to logical 0 with:

```
1 com:ClearLineState( ctb.RTS, ctb.DTR)
```

Both functions accept one or two parameters, the order of the arguments doesn't matter. Here a little example which toggles the RTS line of COM1 every 10 milliseconds for twenty times.

```
1 com = SerialPort.new()
2 if com:Open( com:GetPortName( 1 ), 115200 ) > 0 then
3    for i = 1, 20 do
4      com:SetLineState( ctb.RTS )
5      time.pause( 0.01 )
6      com:ClearLineState( ctb.RTS, ctb.DTR)
7      time.pause( 0.01 )
8    end
9 end
```

1.6 Send a break

Breaks are often used to reset a communication partner. Basically a Break is a low state of the sending line with a duration longer than the time it takes to send a complete byte. Sending a break is simply done with:

```
1 com:SendBreak()
```

which set the TX line in a low state for about 0.25 seconds[2]. For a different break time simply pass the wanted duration time in milliseconds to the function. For example a duration time of 1.5s:

```
1 com:SendBreak( 1500 )
```

1.7 Using script files

All examples above were directly inputed in the **`luactb`** shell. This is nice for trying little code snippets. But what when you like to run larger Lua scripts or doesn't want to input the same code again and again?

[2]The duration is system depending. On Linux the result is in the range of 0.2s...0.3s.

luactb supports as almost all interpreters the execution of a Lua script file given at command line. For instance:[3]

```
luactb.exe yourscript.lua
```

This command starts the **luactb** shell and executes the content of the passed file. (In reality the lua interpreter first complied the script in its internal byte code and runs it afterwards).
Alternatively you might be interested to execute instructions in a script and then enter the interactive mode for further tests. Imagine you want to open a serial port with certain parameters before you start with your own instructions. The following command does exactly that, assumed the file `init.lua` encloses the relevant instructions:

```
luactb.exe -i init.lua
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio, extended by IFTOOLS
>
```

A third variant allows the passing of a short Lua instruction directly via command line and comes into handy for short commands.

```
luactb.exe -e "print(math.sin(45))"
0.85090352453412
```

For further informations about all supported options call the **luactb** with the parameter '-h' or '–help'.

1.8 Helpful functions

luactb hasn't any debugger skills - and you will seldom need them. As an interpreter **luactb** is able to execute single instructions interactively. And to examine variables or function results it's common practice to display them simply with the `print` function.
Nevertheless there are situations where you want a more comfortable way to show the content of data. For instance when you try to inspect the data bytes in a binary string sequence - a frequent task with serial communication.
Here comes the internal `xd(data)` function into play. `xd` displays every passed data as a hex dump. For instance:

```
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio, extended by IFTOOLS
> s = "Hello world\r\n"
> xd(s)
00000000  48 65 6c 6c 6f 20 77 6f  72 6c 64 0d 0a           Hello world..
```

[3] Linux users - of course - call the **luactb** shell without an extension.

1.9 The SerialPort type

The `SerialPort` type provides you with a object like interface to access serial ports. After creating a new instance with:

```
1 com = SerialPort.new()
```

you can address a given port simply by using the new variable as a port reference. The `SerialPort` type covers all necessary functions, which are listed below.

Function	Description
ClearLineState	Set the line state of RTS and/or DTR to logical 0.
Close	Closes the assigned serial port.
Flush	Forces a write of all pending or still buffered output data.
GetLine	Reads a sequence of bytes until a given delimiter (default is newline) occurs or a given timeout is reached.
GetLineState	Returns the logical state of every line coded as a 8 bit value.
Open	Opens the given port with the passed settings.
Read	Reads a given number of bytes with an optional timeout and returns it as a Lua string. Without a passed timeout the function returns immediately. Otherwise the function returns when either the given number of data was received or the timeout was expired.
SendBreak	Forces the serial port to send a break signal (logical 0 state of TxD). The default duration is 250ms, other times can passed as an optional parameter in milliseconds.
SetBaudrate	Changes the current baud rate of an even open port to another one. Unusual baud rates are allowed but are not always supported by every UART chip.
SetLineState	Set the line state of RTS and/or DTR to logical 1.
SetParity	Set the parity bit statically to 0 or 1, e.g to simulate a 9-bit data value. Please note, that not every UART chip and/or driver supports this functionality.
Write	Writes a given data sequence (Lua string) to the port and returns the number of really written bytes. Without a passed timeout the function returns immediately. Otherwise it returns when either all data was sent or the timeout was expired.

ClearLineState

Clears the RTS and/or DTR lines by setting them to a logical 0 state.

```
SerialPort.ClearLineState( list )
```

⊗ list : a comma separated list of the two possible line names: ctb.RTS, ctb.DTR

Example

Toggle the RTS line every 1/10s ten times.

```
1 com = SerialPort.new()
2 if com:Open( "COM1", 115200 ) then
3    for i = 0, 10 do
4      com:SetLineState( ctb.RTS, ctb.DTR )
5      time.pause( 0.1 )
6      com:ClearLineState( ctb.RTS, ctb.DTR)
7      time.pause( 0.1 )
8    end
9 end
```

Close

Normally ports are closed automatically when the script ends. But there are situations when you need to close a port immediately.

```
SerialPort.Close()
```

Example

```
1 com = SerialPort.new()
2 com:Open( "COM1", 115200 )
3 -- do something
4 com:Write( "Hello World!" )
5 com:Close()
```

Flush

All data written to a serial port is normally buffered by the OS. If a serial port is closed before the whole data was sent by the serial port device, the data may got lost. To make sure, that all data is really output, call `Flush`. But please note! This function may block for a time depending on the size of the still buffered data.

```
result SerialPort.Flush()
```

= result : True, if the flush was successful, false otherwise.

Example

```
com = SerialPort.new()
if com:Open( "COM1", 115200 ) then
    hugedata = string.rep( "\255", 100000 )
    com:Write( hugedata )
    com:Flush()
end
```

GetLine

Reads data bytes until a passed delimiter (or EOS sequence) was received.

```
line = SerialPort.GetLine( delim, timeout )
```

⊙ delim : The delimiter passed as a Lua string. Default is the newline character. Allowed is any byte sequence, including the nul (hex $00) character.

⊙ timeout : The default timeout is infinite. The function returns either when a data sequence ending with the given delimiter was received or the timeout in seconds was expired.

= line : The returning string includes the delimiter only when the operation was complete. In case of an expired timeout the delimiter will be missed.

Example

```
com = SerialPort.new()
com:Open( "COM1", 115200 )
-- waiting max. 0.5s for a sequence ending with CRLF
delim = "\r\n"
line = com:GetLine( delim, 0.5 )
if line:find( delim ) >= 1 then
    -- ok
end
```

GetLineState

Returns the current state of all modem control lines.

```
linestate = SerialPort.GetLineState()
```

= linestate : The line states are returned as a 9 bit number. Every line state is represented by a single bit:
RTS=0x004, DTR=0x002, CTS=0x020, DSR=0x100, DCD=0x040, RI=0x080

Example

```
1  com = SerialPort.new()
2  if com:Open( ctb.GetPortName( 1 ), 115200 ) then
3     while kb.getkey() == nil do
4       local lines = string.format( "\r%02X", com:GetLineState() )
5       io.write( lines )
6       io.flush()
7        -- refresh the line state each 1/10 second
8       time.pause( 0.1 )
9     end
10 end
```

Open

Opens a serial port with the given name, baudrate, data format and flow control.

```
result = SerialPort.Open( portname, baudrate, dataformat, flowcontrol )
```

⊗ portname : The portname passed as a Lua string like "COM1ör "/dev/ttyS0".

⊗ baudrate : The wanted baudrate as a number. The supported baud rates depend on the device/driver.

⊙ dataformat : A string representing the dataformat. The default value is ’8N1’. Aaccepted data formats are:
A length from 5...8, a parity passed as ’N’, ’E’ ’O’, ’M’, ’S’ (None, Even, Odd, Mark, Space) and 1 or 2 stop bits.

⊙ flowcontrol : A Lua string describing the flow-control. Default is no flow-control, but you can pass a ’rtscts’ or ’xonxoff’ string to activate on of them.

= result : Returns true, when the open call was successful, false otherwise.

Example

```
1  com = SerialPort.new()
2  if( com:Open( "COM1", 115200, "8N1" )
3      -- do something
4      com:Write( "Hello World!" )
5  end
```

Read

Reads data from the serial port. The function returns only when all given bytes were read or the passed timeout in seconds was expired.

```
data = SerialPort.Read( count, timeout )
```

⊗ count : Count of bytes you want to read.

⊙ timeout : An optional timeout in seconds. The default is infinite.

= data : The received data as a Lua string. Remember that a Lua string can contain every byte value.

Example

```
com = SerialPort.new()
if com:Open( "COM1", 115200 ) then
    -- do something
    com:Write( "Some requesting command" )
    -- read the answer
    local answer = com:Read( 100, 0.5 )
    if #answer == 0 then
        print( "No response!" )
    end
    com:Close()
end
```

SendBreak

Sends a break signal for an optional time.

```
SerialPort.SendBreak( duration )
```

⊗ duration : The default duration for a break is 250 milliseconds. You can pass another duration in milliseconds. The resolution depends of your OS.

Example

```
com = SerialPort.new()
if com:Open( "COM1", 115200 ) then
    -- send a break for 1s
    com:SendBreak( 1000 )
com:Close()
```

SetBaudrate

Changes the baudrate of an current opened port. Normally you pass the wanted baudrate when you open the port. But sometimes it is of interest to manipulate the actual rate, e.g. for testing purposes.
Please note that the change only affects the byte still pending.

```
SerialPort.SetBaudrate( baudrate )
```

⊗ baudrate : The new baudrate as a number.

Example

```
com = SerialPort.new()
if com:Open( "COM1", baudrate ) then
   baudrates = { 300, 600, 1200, 2400, 4800, 9600, 19200,
                 38400, 57600, 115200, 230400, 460800 }
   for i = 1, #baudrates do
     com:SetBaudrate( baudrates[ i ] )
     time.pause( 0.1 )
     com:Write( "This is a baudrate test!" )
   end
end
```

SetLineSate

Changes the line state of the RTS and/or DTR line.

```
SerialPort.SetLineState( list )
```

⊗ list : a comma separated list of the two possible line names: ctb.RTS, ctb.DTR

Example

Toggle the RTS line every 1/10s ten times.

```
\begin{luacode}
\begin{lstlisting}
com = SerialPort.new()
if com:Open( "COM1", 115200 ) then
   for i = 0, 10 do
     com:SetLineState( ctb.RTS, ctb.DTR )
     time.pause( 0.1 )
     com:ClearLineState( ctb.RTS, ctb.DTR)
     time.pause( 0.1 )
   end
end
```

SetParity

Set the parity for the next byte to mark or space.
Please note! This function is not always supported by every device and/or driver.

```
SerialPort.SetParity( state )
```

⊗ state : The new parity state. True means a logic high state, false a logic low state.

Example

Simulating a 9-bit address in a multi-drop protocol sequence.

```
1  com = SerialPort.new()
2  if com:Open( "COM1", 115200 ) then
3      -- the parity used as the ninth bit
4      com:SetParity( 1 )
5      time.pause( 0.1 )
6      -- the first byte is the 9-bit address (here 0x101)
7      com:Write( "\001" )
8      com:SetParity( 0 )
9      time.pause( 0.1 )
10     com:Write( "Some data" )
11     time.pause( 0.1 )
12 end
```

Write

Writes the given data to the serial port. The function will block until all data is written or the timeout is expired. The default timeout is infinite.

```
written = SerialPort.Write( data, timeout )
```

⊗ data : The data passed as a Lua string.

⊙ timeout : An optional timeout value in seconds. Default is infinite and the function call will not return until all data are sent.

= written : The number of written bytes. By using a timeout the number can be less than the passed data. A negative result is caused by an error.

Example

```
1  com = SerialPort.new()
2  if com:Open( ctb.GetPortName( 1 ), 115200 ) then
3     local data = string.rep( "\255", 10000 )
4     local left = #data
5     while left > 0 do
6       -- non-blocking, timeout is 0s
7       local wr = com:Write( data:sub(-left), 0 )
8       -- error check
9       if wr < 0 then
10         print( "Error!" )
11         break
12      end
13      left = left - wr
14      -- do something, e.g. show a progress counter
15      io.write( "\rRemaining bytes:"..left)
16      io.flush()
17    end
18 end
19 com:Flush()
20 print( "\nComplete!\n")
```

1.10 The base16 Module

A lot of protocols use a base16 encoding when transferring the data payload. You may have seen such a thing before. Data sequences, which always look like a ASCII string containing only the characters '0'...'9' and 'A'...'F'. Modbus ASCII for instance is one of them.
Encoding the data in a Base16 format let you choose any other character as a telegram delimiter and makes it easy to separate the single telegrams. The disadvantage: It increases the data volume for twice.
Despite of the advantages and drawbacks is base16 an often used coding standard and **`luactb`** offers you an already integrated module to do the job.

Function	Description
decode	Decodes a given Lua string containing a valid Base16 sequence and return the result as a Lua string.
encode	Encodes a given Lua string and returns a Lua string containing the Base16 sequence.

decode

Converts a Lua string covering a valid base16 sequence into its binary representation. The decoding stops with the first invalid character or when the string end was reached. In case of an odd sequence length, the last character will be ignored.

```
result = base16.decode( sequence )
```

⊗ sequence : A valid base16 sequence passed as a Lua string.

= data : The binary representation of the given base16 sentence. Without an error the resulting string length must exactly be half of the passed sequence.

Example

Converting a base16 sequence into its binary equivalent.

```
1 b16 = "68656C6C6F20776F726C64"
2 bin = base16.decode( b16 )
3 if #bin * 2 ~= #b16 then
4     print( "Error!"
5 else
6     print( bin ) --> "hello world"
7 end
```

encode

Converts a given Lua string into a base16 sequence. Since all bytes in a Lua string can be represented as two base16 characters, the function will never fails.

```
result = base16.encode( sequence )
```

⊗ sequence : Any Lua string.

= data : The base16 representation of the given sequence as a Lua string.

Example

Converting a Lua string to its base16 equivalent.

```
1 print( base16.encode( "hello world\r\n" ) )
```

1.11 The checksum module

luactb provides you with an already integrated checksum module supporting a checksum generator for Modbus ASCII (LRC) and Modbus RTU (CRC16). All checksum functions requires a Lua string to calculate the checksum from and return the result as a number.

Function	Description

crc16_modbus	Calculates a CRC16 checksum with the crc16 polynom used by the Modbus RTU protocol.
lrc	Calculates the LRC checksum used by the Modbus ASCII protocol.

crc16_modbus

Calculates the Modbus CRC16 checksum (Cyclical Redundany Checking) of the given Lua string.

```
checksum = checksum.crc16_modbus( sequence )
```

⊗ sequence : A Lua string representing the data sequence.

= checksum : The 16 Bit checksum result.

Example

Calculate the CRC16 checksum of the given string.

```
1 checksum = checksum.crc16_modbus( "Hello world" )
```

lrc

Calculates the LRC checksum (Longitudinal Redundany Checksum) of the given Lua string.

```
checksum = checksum.lrc( sequence )
```

⊗ sequence : A Lua string representing the data sequence.

= checksum : The 8 Bit checksum result.

Example

Calculate the LRC checksum of the given string.

```
1 checksum = checksum.lrc( "Hello world" )
```

1.12 The ctb module

The `ctb` (communication tool box) module covers several helpful functions which are not assigned to a single serial port (like the SerialPort module or class). It provides you with functions to list all available ports, and to find a port

by passing information like the serial number, vendor or product ID. It also defines name qualifiers for the modem control lines.

Function	Description
FindPort	Queries for a port which matches all specified search parameters and returns the name of the first found device or nil otherwise.
GetPortName	Returns the standard port name of the given number.
ListPorts	Lists all available ports including hardware information and state (used or free).
Defined names	Not a function but a list of predefined names for the modem control lines as used in some SerialPort functions.

FindPort

FindPort looks for a serial port which matches the given search criteria. The function accepts beside a serial number, vendor and product ID also the producer name and a product name.
The arguments are all optional and are passed as so called named parameter to make the call preferably understandable. In case of a hit the device name (COMx or /dev/tty...) is returned, `nil` otherwise.

```
devname = ctb.FindPort{ vendor="", serial="", vid="", pid=""}
```

- vendor : The vendor or producer name as a Lua string. E.g. "FTDI".
- serial : The serial number of the device as a Lua string. E.g. "FT3W5E11".
- vid : The vendor ID (especially used by USB to serial port converters. The ID is always a 4-digit hex number and must passed as a Lua string with only lowercase letters. For instance: The Profilic vendor ID is given as "067b".
- pid : The product ID. Like the vid the ID must passed as a 4-digit hex number sting in lowercase letters. E.g. "2303"for the Profilic PL2303.

= devname : The device name, nil otherwise.

Example

Open the FTDI USB to RS232 converter with the serial number FT3W5E11

```
1 com = SerialPort.new()
2 if com:Open( ctb.FindPort{ serial="FT3W5E11" }, 115200, "8N1" ) then
3    com:Write( "This data is only for the given device!" )
4 end
```

GetPortName

This function requires a positive integer number and returns the according serial port name given by the OS and independent of an existing port. Under Windows the result is "COMx"where x is the given number. Under Linux the function returns "/dev/ttySx"where x is the passed number - 1.
Please note!
Since Linux handles USB to Serial Port devices differently, `GetPortName` works only for standard ports named as /dev/ttySx.

```
devname = ctb.GetPortName( number )
```

⊗ number : A valid port number starting from 1.

= devname : The device name, nil in case of an invalid parameter.

Example

Open the first COM port (COM1 on Windows or /dev/ttyS0 on Linux)

```
1 com = SerialPort.new()
2 if com:Open( ctb.GetPortName( 1 ), 115200, "8N1" ) then
3    com:Write( "This data is only for COM1 or dev/ttyS0!" )
4 end
```

ListPorts

With `ListPorts` you can retrieve information about all serial ports actually connected with or installed in your PC. This includes information about the vendor (manufacturer), the vendor and product ID of USB devices, the serial number, and optional driver name and version.
The result is a table (or array) or strings. Every string contains the information of a single port separated by a tab in the following order:

- OS device file name (like COM1 or /dev/ttyS0)
- vid:pid (vendor and product ID separated by a colon)
- vendor name
- serial number
- state (used or free)
- driver name (optional)
- driver version (optional)

A typical output may look like this:

```
/dev/ttyUSB     10403:6001        FTDI     FT3W5E11        FREE
```

```
ports = ctb.ListPorts( 'showDrivers' )
```

- showDrivers : By passing the optional string 'showDrivers' additional driver information are listed too.

devname : A Lua table of strings. The length (size) of the table indicates the count of found serial ports.

Example

List all serial ports include driver information

```
1 ports = ctb.ListPorts( 'showDriver' )
2 for i=1,#ports do
3     print( ports[ i ] )
4 end
```

Defined names

The following predefined names can be used to check the result of the SerialPort function GetLineState or to pass a certain line argument to ClearLineState or SetLineState. For more information see the description of these functions.

```
ctb.DTR
ctb.RTS
ctb.CTS
ctb.DCD
ctb.RI
ctb.DSR
```

1.13 The kb (keyboard) module

The kb module covers additional functions for handling keyboard inputs which are not part of Lua or which are are difficult to realize in a platform independent way. Actually it contains only one function to check if a key was pressed without blocking the program flow.

Function	Description
getkey	Checks, if any key was pressed and returns the key code or nil otherwise.

getkey

Unlike the standard input functions `getkey` don't wait for any key. It just checks if a key or combination of keys is actually pressed and returns the key code or nil. This function becomes especially convenient if you want to check for a abort key in a loop.

```
keycode = kb.getkey()
```

= keycode : The according keycode or character when a key was pressed, nil otherwise.

Example

Run a loop until the user hits the 'q' key.

```
1 print( "Press 'q' to abort loop")
2 while true do
3    key = kb.getkey()
4    if key == string.byte('q') then break end
5 end
```

1.14 The time module

Function	Description
pause	Checks, if any key was pressed and returns the key code or nil otherwise.
ticks	Checks, if any key was pressed and returns the key code or nil otherwise.

pause

Pauses the execution of the script for the given time of seconds. The current process (running the Lua interpreter) will 'sleep' and doesn't consume any CPU time, until the pause time was expired. The time resolution is in milliseconds.

```
time.pause( seconds )
```

⊗ seconds : Pauses the program for the given time.

Example

Send a sequence every 1/2 second

```
while not kb.getkey() do
   print( os.date() )
   time.pause( 0.5 )
end
```

ticks

Gives you the number of milliseconds since start of the OS (Windows) respectively the start of the Unix epoch (1.1.1970). Lua comes with its own `os.time` function but with a resolution of one second only. The `time.ticks` function however allows a time measuring in milliseconds in a very simple way.

```
milliseconds = time.ticks()
```

= milliseconds : The current number of milliseconds since start of the internal millisecond timer.

Example

Meters the transmission of 10000 bytes

```
com = SerialPort.new()
if com:Open( ctb.GetPortName( 1 ), 115200 ) then
   local t0 = time.ticks()
   com:Write( string.rep("\255", 10000 ) )
   com:Flush()
   print( time.ticks() - t0 )
end
```

1.15 Further information

This chapter can't replace any good introduction to Lua. It only covers the necessary information you need to undertake the first steps with **luactb**.
It also can give you a small outlook of all the language features Lua comes with.
For more information about Lua please visit the Lua website at http://www.lua.org.
A very good tutorial is available at http://lua-users.org/wiki/TutorialDirectory too.

1.16 FAQs

How can I check the program version?

You can query the program version with the function `version`. The result is a string containing the MAJOR, MINOR and RELEASE numbers. For instance:

```
print( version() ) --> returns something like 0.9.0
```

1.17 Lua license and copyright

Lua is licensed under the terms of the MIT license reproduced below. This means that Lua is free software and can be used for both academic and commercial purposes at absolutely no cost.

For details and rationale, see http://www.lua.org/license.html.