Quickly creating a framework for communicating to an embedded gas sensor

… or …

How I develop

Stephan Effelsberg, EuroTcl 2023, Vienna

About the inception of this talk

- Manfred Rosenberger saw that I used the ukaž graph widget and suggested a lightning talk.
- It’s not my widget and I only use a small part of its capabilities.
- I’d better show the widget in the context of my application.
- Note: The code in here is not self-contained. These slides are supposed to be accompanied by live demonstrations.

Thanks to Manfred Rosenberger for pushing me to give a talk.

About me

Name	Stephan Effelsberg
Employer	GfG mbH, Dortmund, gas detection specialist
Job description	Programming mobile gas detectors in Embedded C
Also	Tcl/Tk has surpassed C++/FLTK more and more for programming supporting tools (My choice. The user facing code is done in C#.)

About me

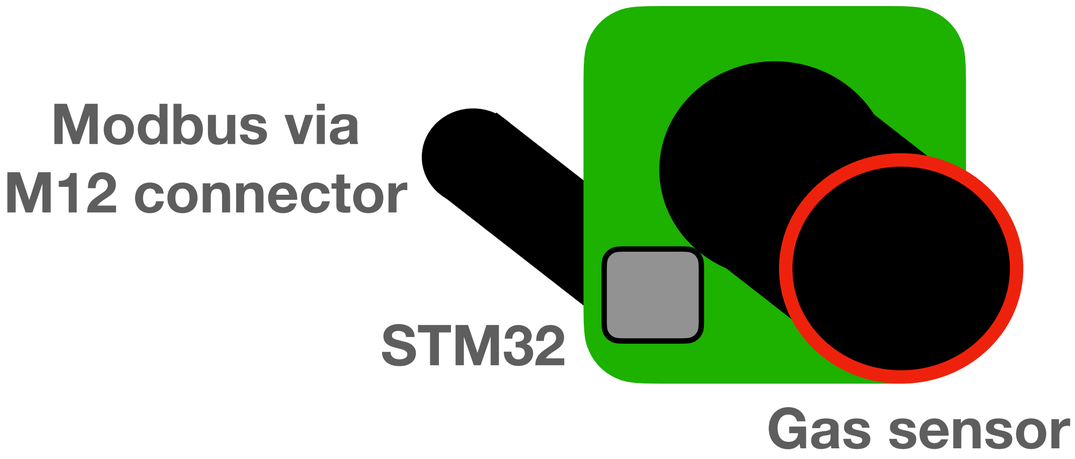
We’re not an open source company. However, what I’m going to show is a general (albeit my specific) idea to solve the problem at hand.

It just happens that the device at the other end of the communication line is a real industrial sensor device.

The main act

I’m tasked with programming an Arm-based processor to control a gas sensor an offer a Modbus communication interface.

This sensor cartrdige is supposed to be plugged into a larger device.



The supporting act

The sensor cartridge offers a Modbus server (taken from an existing project).

I never interfaced with a Modbus device before, so I want to check that I can talk to it as quickly as possible, where quickly means about one day of work. There are quite generic tools like Modbus Poll but I want something that is specialized for the job at hand. The application to be developed is a developer's tool, not a user facing tool.

Doing things quickly

- Use a binary distribution. Yes, I’m a programmer, but I don’t fell like building everything from source.
- With batteries included, even if only a few are used: tcllib, ukaž, BWidget, …
- Some of my own favorite code
- And of course a lot of experience

Thanks to Ashok P. Nadkarni for MagicSplat (and too many people for Tcl itself)

Mac interlude

This talk is going to be presented on a MacBook, so let's quickly check if I can talk to a Modbus server. I'm using an off-the-shelf USB-to-RS485 adapter from Digitus. I also learned how to read register 0 from Modbus slave 1 and used this fixed byte sequence.

```
set f [open /dev/cu.usbserial-AI02DYGR RDWR]
fconfigure $f -blocking 0 -buffering none \
              -mode 19200,e,8,1 -translation binary
puts -nonewline $f \x01\x04\x00\x00\x00\x01\x31\xca
after 500
set bin [read $f]
binary scan $bin cu* data ; puts $data
# The exact answer depends on the contents of the register
# but is somehow recognizable.
```

It's working. With stock Tcl 8.5.9!

Thanks, Apple, but no …

Mac interlude

There's no easily discoverable binary distribution for Apple Silicon Macs. (Or is there? Sorry, Alex, I know by chance that you have a precompiled version.)

BAWT to the rescue. A single small download, some simple commands, and about 20 minutes later …

```
% set tcl_patchLevel
8.6.13
```

Thanks to Paul Obermeier (and Alexander Schöpe)

A GUI skeleton

As a developer I like to type commands but I also like to click a button. I know already that I'll partition the functionality of the sensor, so there's a notebook for the subsystems, a text widget for the main output and one for telegram logs. Plus a toolbar and a statusbar.

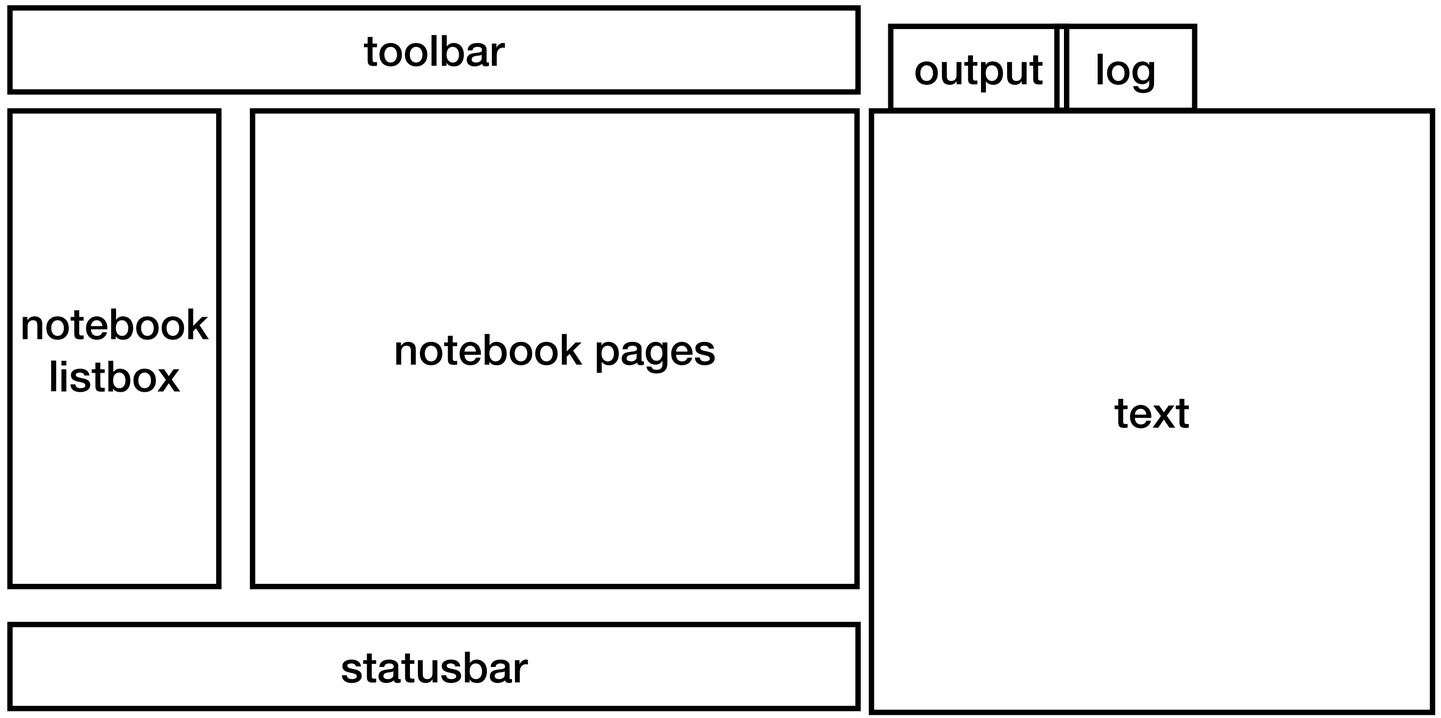
If a notbook has many pages, I like it to be controlled by a listbox. The contenders are:

- My own listbook (~ 150 lines)
- plainnotebook from scrollutils
- the Notebook from BWidget

The advantage of plainnotebook is that it's API is compatible to the API of ttk::notebook.

Thanks to Csaba Nemethi and Harald Oehlmann

A GUI skeleton



A GUI skeleton

One API of the GUI is the global gui array. It's used to configure parts of the GUI that is to be created as well as to hold catchy names of the widget I want to use later on.

```
% parray gui
gui(console)           = tkcon
gui(fcomm)             = .f1
gui(llcomframe)        = .f1.comframeModbus.llcomframe
gui(msgline)           = .stbar
gui(nb)                = .f2.nb
gui(notebookcommand)   = listbook
gui(output)            = .nbout.f_output.output
gui(raw_log)           = .nbout.f_raw_log.raw_log
gui(statusbar)         = .stbar
gui(use_bwidget)       = true
```

A simple plug-in system

The notebook at the left is going to be filled with numerous subsystems, so it's a good idea to create a simple API for that. It's used like this:

```
sub add "Hello" CreateHelloFrame

proc CreateHelloFrame {w} {
    set f [frame $w]
    set b [button $f.b -text "Hello" -command {
        Puts "Hello"
    }]
    pack $b
    return $w
}
```

Demo

The GUI so far. Further work will go into doing some real work.

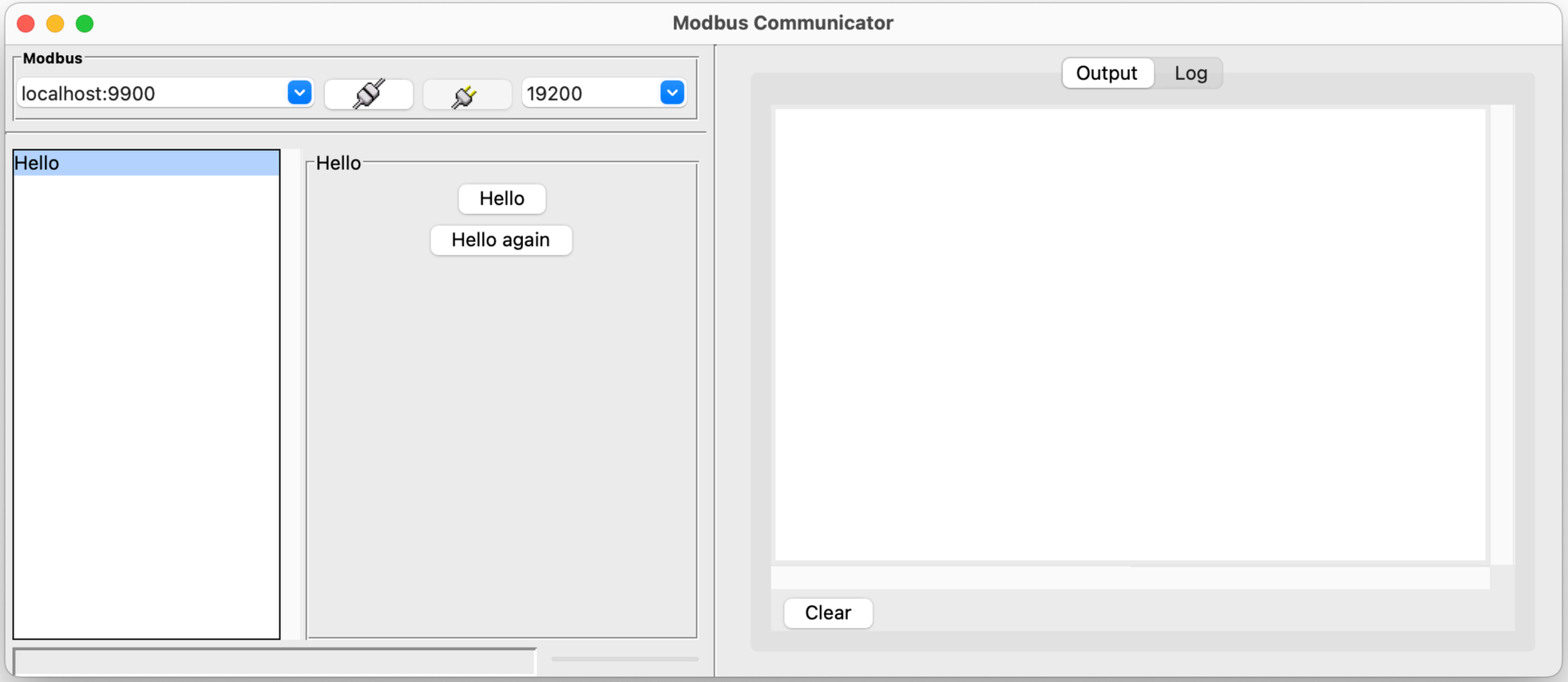
Enhancing the developer’s experience

The application offers a (menu) command that reloads and rebuilds all submodules. This means that I can work on the submodules while the GUI is running and continually improve the functionality with a somewhat tight feedback loop.

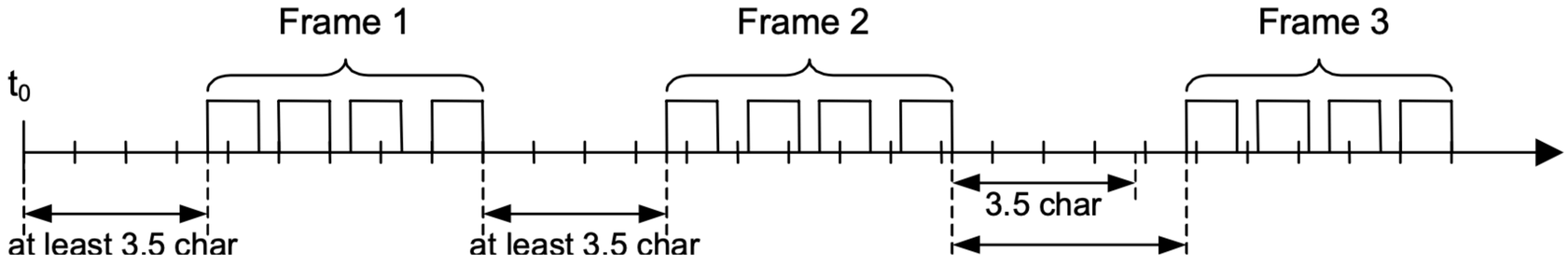
Let’s add a button to the Hello submodule.

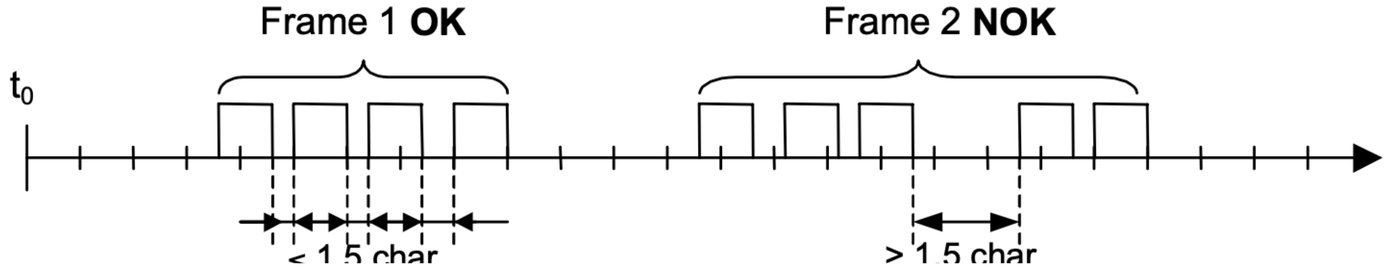
Demo

The GUI was not restarted, the submodules were destroyed and rebuilt.



Reading the Modbus spec





MODBUS message

Start	Address	Function	Data	CRC Check	End
≥ 3.5 char	8 bits	8 bits	N x 8 bits	16 bits	≥ 3.5 char

Figure 13: RTU Message Frame

Reading the Modbus spec

About the timing: The Modbus server in the cartridge meticulously implements the correct timing. However, you don’t need dedicated hardware for the PC to use the Modbus except for the purpose to check this very speciifc part.

Sending a telegram: The PC is capable of sending serial data without gaps between the bytes.

Receiving a telegram: What you need to know is that the PC gives you a data packet every 16 ms, so this dominates the character timeout. A value of 20 ms has so far proved to be good on Windows and macOS.

Constructing telegrams

On the most basic level, the telegram contains the slave is, the function code, the funtion-specific data and a CRC that can be generated by the tcllib module.

```
proc assembleTelegram {slave func bindata} {
    set tg    [binary format cucua* $slave $func $bindata]
    set crc   [crc::crc16 -seed 0xFFFF $tg]
    append tg [binary format su $crc]
    return $tg
}
```

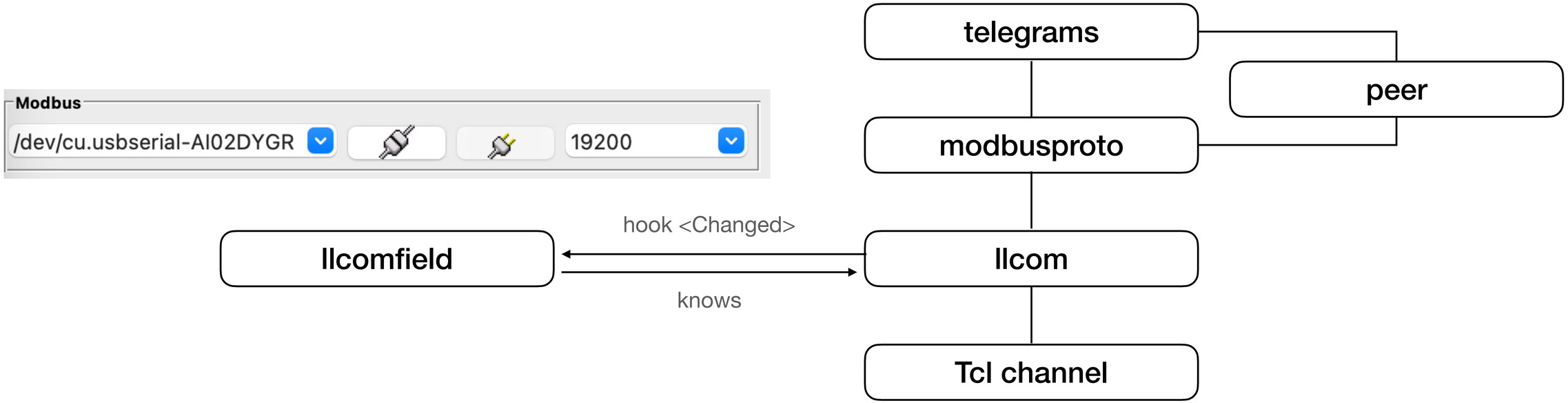
Constructing a special telegram

E.g. if you want to read multiple registers, the Modbus spec translates to this. Take care to use big endian encoding of the data words. Also, it's quite OK to use the binary command immediately for such a task. I leave receiveTelegram open as homework.

```
proc readRegisters {slave address count} {
    set bindata [binary format SuSu $address $count]
    set func 4
    set tg [assembleTelegram $slave $func $bindata]
    send $tg
    return [receiveTelegram]
}
```

Some functionality

Time to fill in some functionality. llcomfield is a snit::widget (ll = low level), llcom and modbusproto are snit::types. modbusproto is now the container for the previously introduced functions.



Thanks to Will Duquette for snit, hook and Vikings at Dino’s

Demo

Playing with the communication facilities in a console.

Simple C-Structure parsing

The C code of the device contains numerous structs to describe the data that can be exchanged.
Example:

```
typedef struct {
    uint8_t      sidx;
    uint8_t      status;
    uint8_t      result;
    uint8_t      testmode;
    uint32_t     timestamp;
    uint16_t     pulse_duration_ms;
    uint16_t     sample_period_ms;
    uint16_t     sample_count;
    int16_t      samples_mV[100];  // sample_count, max. 100
} SentestRsp;
```

Simple C-Structure parsing

I’d like to have these structs in the Tcl code as well, because the binary command can get a bit clumsy for such structure sizes. Havinfg the same syntax also helps checking that they are in sync.

I already had a Tcl module, I’d like to challenge my code sometimes, however. It’s a bit like playing chess. The openings are usually the same but the moves diverge quickly.

Let me replay my thoughts I had back then.

Simple C-Structure parsing

A struct looks very much like a type-identifier list. Let's construct a binary scan command from it. For the intended purpose I don't like the identifiers to be available immediately but rather nicely packed into an array of identifiers.

```
typedef struct {
    int16_t    x;
    int16_t    y;
} Point;

-> binary scan $bindata ss x y

# or rather

-> binary scan $bindata ss idarray(x) idarray(y)
```

Simple C-Structure parsing

Type look-up uses an array, int16_t is supported.

```
namespace eval scstruct {
    variable binformat_from_ctype
    array set binformat_from_ctype {
       int16_t     s
    }
}

proc scstruct::fromBinary {decllist bindata} {
    variable binformat_from_ctype
    set formatstring ""
    set idarraylist [list]
    foreach {ctype id} $decllist {
        append formatstring $binformat_from_ctype($ctype)
        lappend idarraylist idarray($id)
    }
    binary scan $bindata $formatstring {*}$idarraylist
    return [array get idarray]
}
```

Simple C-Structure parsing

Example usage

```
set decllist {
    int16_t x
    int16_t y
}

scstruct::fromBinary $decllist \x01\x00\x02\x00
-> x 1 y 2
```

Simple C-Structure parsing

Don’t construct the whole format string for a single binary command, use binary immediately for each identifier. This also helps checking for enough data.

```
proc scstruct::fromBinary {decllist bindata} {
    variable binformat_from_ctype
    foreach {ctype id} $decllist {
        set fmtstr $binformat_from_ctype($ctype)$len
        if {[binary scan $bindata ${fmtstr}a* idarray($id) rest] == 0} {
            error "Fewer bytes than expected"
        }
        set bindata $rest
    }
    return [array get idarray]
}
```

Simple C-Structure parsing

Freeing the mind from the fact that the struct's contents look like a list, parse as lines instead. The regexp also immediately supports the semicolon after the identifier. In fact, it supports many things. However, this is not supposed to be a syntax checker.

```
proc scstruct::fromBinary {decllist bindata} {
    variable binformat_from_ctype
    foreach decl [split $decllist \n] {
        if {[string is space $decl]} {
            # empty line
        } elseif {[regexp -expanded {
            (\w+)         (?# type)
            \s+
            (\w+)         (?# identifier)
        } $decl -> ctype id]} {
            set fmtstr $binformat_from_ctype($ctype)
            if {[binary scan $bindata ${fmtstr}a* idarray($id) rest] == 0} {
                error "Fewer bytes than expected"
            }
            set bindata $rest
        } else {
            error "Unknown declaration: $decl"
        }
    }
    return [array get idarray]
}
```

Simple C-Structure parsing

Accept either a decllist immediately or the name of typedef'd struct. Add a struct repository for this case.

```
namespace eval scstruct {
    variable internal_from_typename
    array set internal_from_typename {}
}
proc scstruct::fromBinary {decllist bindata} {
    variable internal_from_typename
    if {[llength $decllist_or_typename] == 1} {
        return $internal_from_typename($decllist_or_typename)
    } else {
        return [ToInternal $decllist_or_typename]
    }
    variable binformat_from_ctype
    foreach decl [split $decllist \n] {
        . . .
    }
    return [array get idarray]
}
```

Simple C-Structure parsing

Add the typedef. And comments.

```
proc scstruct::declare {typename decllist} {
    variable decllist_from_typename
    set decllist_from_typename($typename) $decllist
}

proc typedef {struct decllist typename} {
    scstruct::declare $typename $decllist
}

proc // {args} {}
```

Simple C-Structure parsing

Need more data types. Int16_t and Uint16_t are self-made types for big endian words like they are used for Modbus.

```
namespace eval scstruct {
    variable binformat_from_ctype
    array set binformat_from_ctype {
        int8_t      c
        int16_t     s
        int32_t     i
        uint8_t     cu
        uint16_t    su
        uint32_t    iu
        Int16_t     S
        Uint16_t    Su
    }
}
```

Simple C-Structure parsing

Add support for C arrays and comment lines in structs.

```
foreach decl [split $decllist \n] {
    if {[regexp {^(//.*)?$} $decl]} {
        # empty line or comment line
    if {[string is space $decl]} {
        # empty line
    } elseif {[regexp -expanded {
        (\w+)         (?# type)
        \s+
        (\w+)         (?# identifier)
        (\[(.*)\])?   (?# optional array length)
    } $decl -> ctype id (len) len]} {
        set fmtstr $binformat_from_ctype($ctype)$len
        if {[binary scan $bindata ${fmtstr}a* idarray($id) rest] == 0} {
            error "Fewer bytes than expected"
        }
        set bindata $rest
    } else {
        error "Unknown declaration: $decl"
    }
}
```

Simple C-Structure parsing

Status report: I could by now write a file that is both valid Tcl code and a C header.

```
#ifndef TELEGRAM_STRUCTS
#define TELEGRAM_STRUCTS
// Declarations of structs
// for telegram exchange

#pragma pack(push, 1)

typedef struct {
    uint8_t      sidx;
    uint8_t      status;
} SentestShortRsp;
```

```
typedef struct {
    uint8_t      sidx;
    uint8_t      status;
    uint8_t      result;
    uint8_t      testmode;
    uint32_t     timestamp;
    uint16_t     pulse_duration_ms;
    uint16_t     sample_period_ms;
    uint16_t     sample_count;
    int16_t      samples_mV[*];
} SentestRsp;
#pragma pack(pop)
#endif  // TELEGRAM_STRUCTS
```

Simple C-Structure parsing

Not shown:

- The complementary toBinary function, this involves extracting the common parts; also parsing the struct at the moment of declaration, not at the moment of usage, remembering the parsed representation and using this when needed.
- Supporting a flexible array member as last element of a struct (returns * as len which denotes „the rest“ for the binary command).
- Looking up a known typedef’d struct name in addition to the primitive types. Structs inside of structs!

Demoing a struct usage for a purpose

The device has a fixed scheme of doing its work in steps of 50 ms. It's also counting from 1 to 10 which is the base for a heartbeat. Let's tap into the counter by listening to a trace event.

```
DTraceAdd 5 "Main sequence timing" {
    uint16_t    start_ticks;
    uint8_t     duration_ticks;
    uint8_t     state_and_half;
} -formatter {
    set state [expr {$state_and_half & 0x7f}]
    set half  [expr {$state_and_half >> 7}]
    append str "Main seq: $start_ticks ms
$duration_ticks ms  half: $half  state: $state"
}
```

Gently introducing ukaž

Idea: Can I visualize the data instead of having a plain text output?
Ad-hoc development plotting.

```
proc gplot {x y} {
    if {![winfo exists .plotwin]} {
        toplevel .plotwin
        ukaz::graph .plotwin.g
        .plotwin.g set grid on
        button .plotwin.b_clear -text "Clear" -command {
            .plotwin.g clear
        }
        pack .plotwin.g        -expand true -fill both
        pack .plotwin.b_clear
    }
    raise .plotwin
    .plotwin.g plot [list $x $y]
}
```

Demo: Explorative plotting

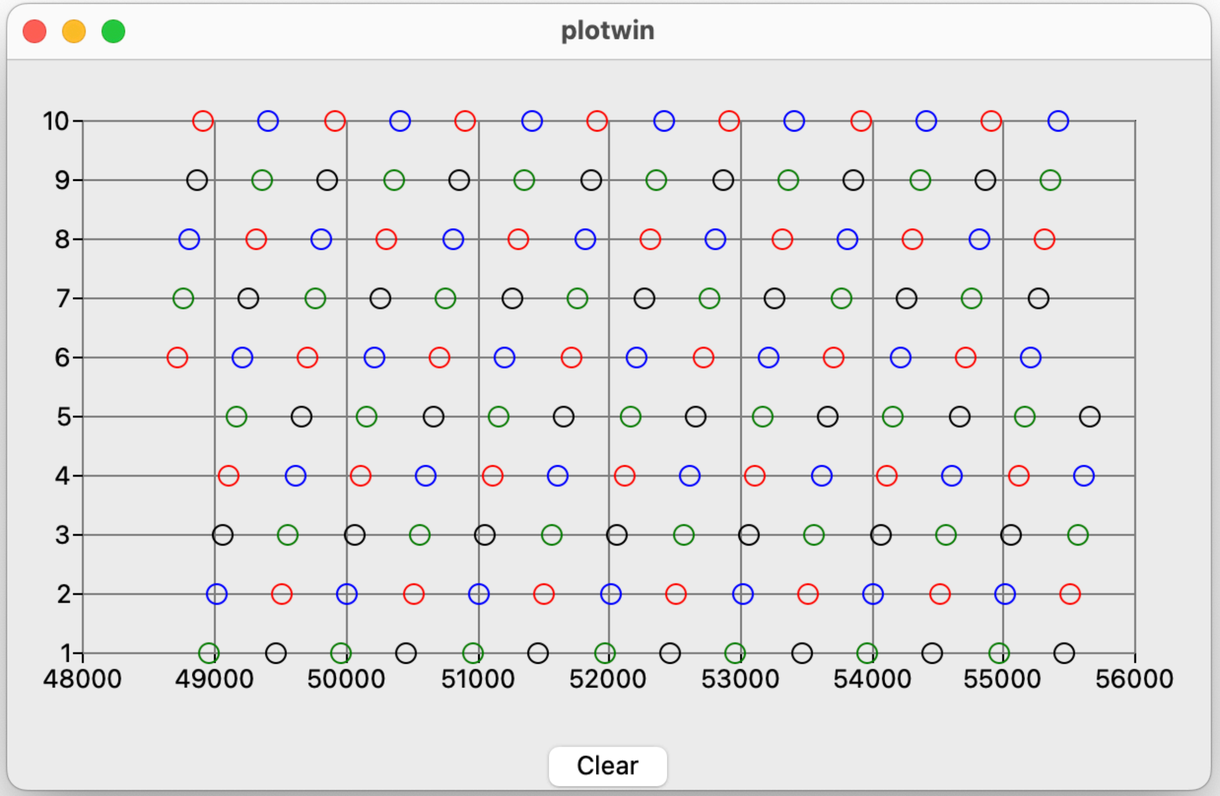
The extra line produces some computational art.

```
DTraceAdd 5 "Main sequence timing" {
    uint16_t    start_ticks;
    uint8_t     duration_ticks;
    uint8_t     state_and_half;
} -formatter {
    set state [expr {$state_and_half & 0x7f}]
    set half  [expr {$state_and_half >> 7}]
    gplot $start_ticks $state
    append str "Main seq: $start_ticks ms  $duration_ticks ms
half: $half  state: $state"
}
```

Thanks to Christian Gollwitzer

Demo: Explorative plotting

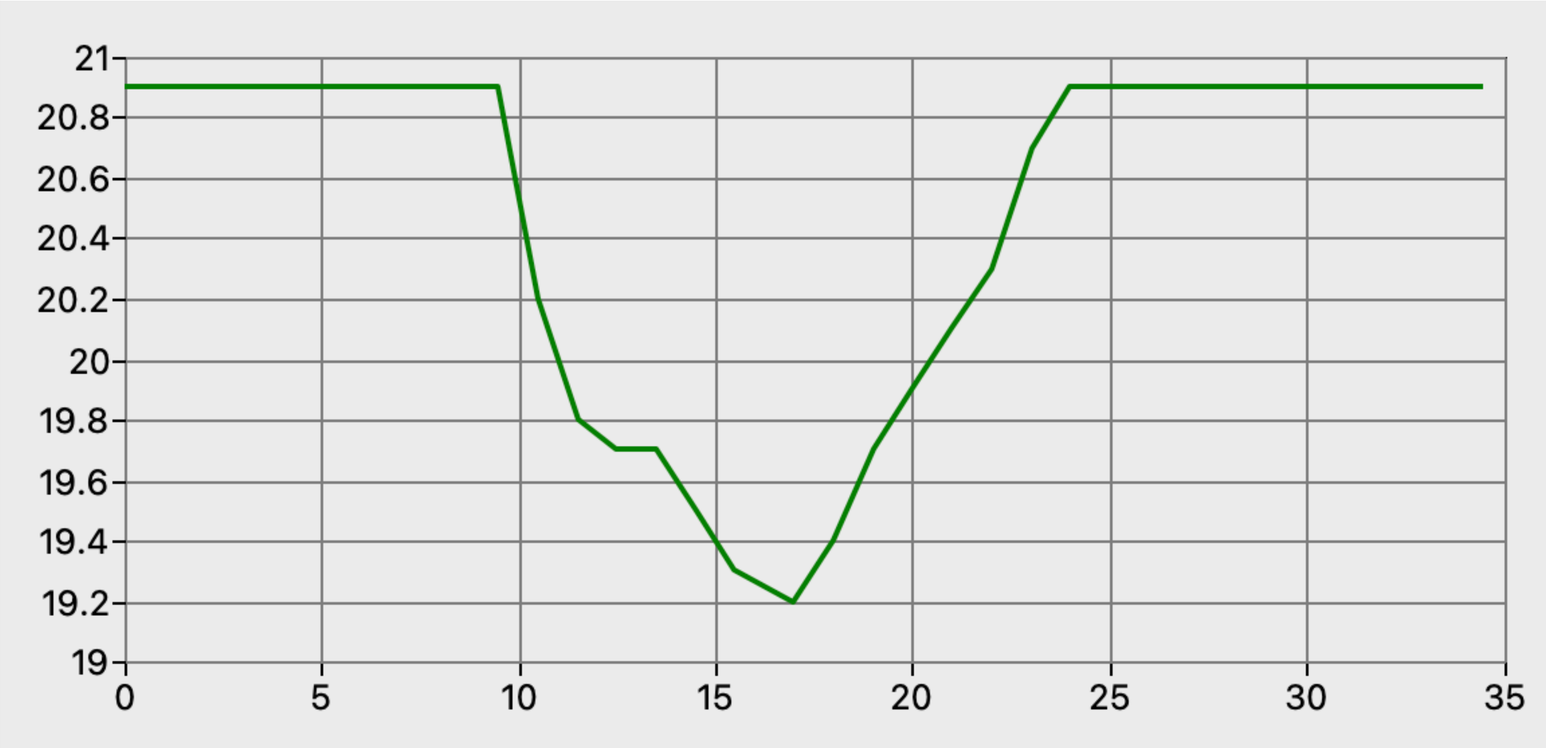
Example heartbeat counter, counting from 1 to 10, one step per 50 ms.



Demo: Plotting a signal

Now you'll see what Manfred Rosenberger has seen. Blowing into an O_2 sensor.

Loop closed.



Afterword

This was written after the presentation, therefore I can use finer print because it was never meant to be beamed onto a wall.

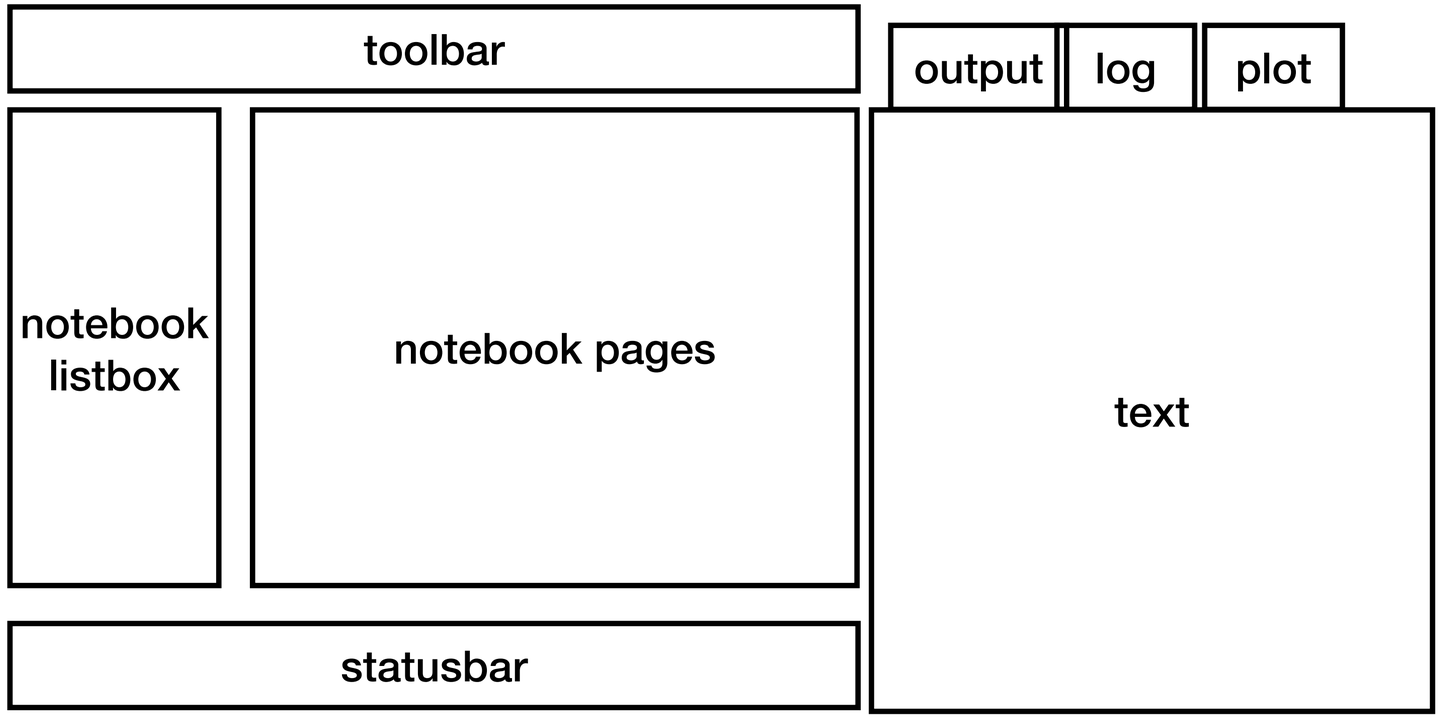
When I felt the urge wo give a talk, after Manfred's inspiration, I thought I had about 5 minutes worth to say. I'm not advancing the state-of-the-art of Tcl and the initial idea focused on a widget that was not mine. When I finally started writing something about the topic, many ideas came up and I thought they were important or at least interesting.

Where does the data come from? The source is a device that is programmed in C and the data is described using C structs. I've taken the opportunity to write a struct parser from scratch at home. To challenge the one I've written at work. To show that my basic needs are served by 30 or 40 lines of code that are quickly assembled by layering up small incremental ideas. To show how I my development process sometimes works. I start at work and run out of ideas. On the way home on my bicycle (just an accidental reference to Manfred) an inspiration strikes me. I'll try it out at home and the next day again at work. That's already a code kata. I've got an idea how to solve a problem, solve it, and solve it again.

Since „quick" also was a topic of the talk, I thought it might be a good Idea to show that I often keep the application running while modifying the code and the key for it is the submodule system that states how to create a UI frame for the submodule, thus delivering the necessary process of create and destroy. I can simply overwrite procs but not widgets. Modifying running code for ad hoc visualization of data is a nice trick. However, it is not essential, it just sucked up the available time.

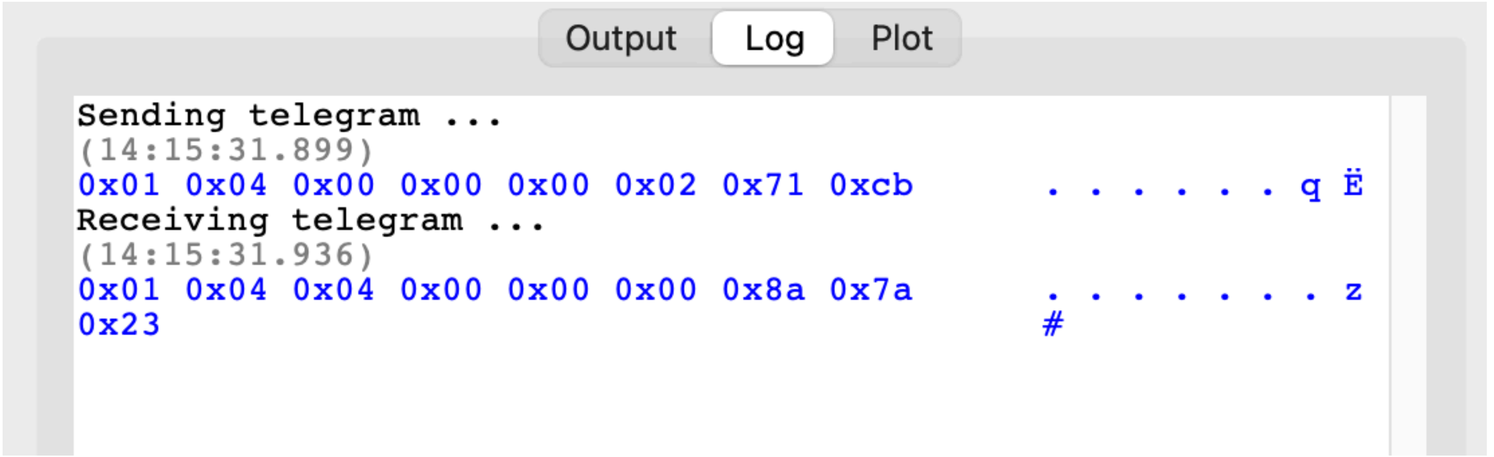
Afterword

Did I mention that I get some of my best ideas not when working on a thing but rather when doing something else? Well, after the talk, still in Vienna, I thought that the ad hoc window for visualizations was too intrusive. I already have to output panes, named output and log, why not add a plotting pane?



Afterword

Since there's no screen capture of the panes, let me quickly show what the initial output and log are supposed to show. Here I'm reading two Modbus registers, containing the values 0 and 138.



Afterword

I like to see the actual telegram data which is formatted using these 10+ lines of code. This is by the way the result of a code kata I did in Vienna. It's not the original function version 1 I used at work. (In fact, most code you've by chance seen in the talk is a code kata style recreation.)

```
proc HexFormat {bindata {n 8}} {
    set lines {}
    set ascii_offset [expr {$n * 5 + 5}]
    set n_1 [expr {$n - 1}]

    for {set i 0} {$i < [string length $bindata]} {incr i $n} {
        set linedata [string range $bindata $i $i+$n_1]
        # hexes: e.g. {0x48 0x65 0x6c 0x6c 0x6f ...}
        binary scan $linedata cu* values
        set hexes [lmap v $values {format "0x%02x" $v}]
        # chars: e.g. {H e l l o ...}
        set chars [split $linedata ""]
        set chars [lmap c $chars {expr {[string is print $c] ? $c : "."}}]
        lappend lines [format "%-${ascii_offset}s%s" [join $hexes] [join $chars]]
    }

    return $lines
}
```

Afterword

Back to the plot window that is supposed to turn into a plot panel in the output ttk::notebook. So instead of gplot the proc is now called Plot so I can have both implementations at the same time.

```
proc PlotFrame {w} {
    global gui
    set f [frame $w]
    ukaz::graph $f.g
    $f.g set grid on
    button $f.b_clear -text "Clear" -command [list $f.g clear]
    pack $f.g       -expand true -fill both
    pack $f.b_clear
    set gui(plot) $f.g
    set gui(plotcombo) $f.cb
    return $w
}

proc Plot {x y} {
    global gui
    $gui(plot) plot [list $x $y]
}
```

Afterword

With this in place I had the next idea: Why enabling the plot ad hoc and disabling it afterwards? Why not have it there all the time? Give it a name and let me choose an active plot. This feels a bit like using an oscilloscope to probe a signal.

```
proc PlotFrame {w} {
    global gui
    set f [frame $w]
    ttk::combobox $f.cb -values $gui(plotlist)
    ukaz::graph $f.g
    $f.g set grid on
    button $f.b_clear -text "Clear" -command [list $f.g clear]
    pack $f.cb
    pack $f.g       -expand true -fill both
    pack $f.b_clear
    set gui(plot) $f.g
    set gui(plotcombo) $f.cb
    return $w
}

proc Plot {plotname x y args} {
    global gui
    if {$plotname ni $gui(plotlist)} {
        $gui(plotcombo) configure -values [lappend gui(plotlist) $plotname]
    }
    if {$plotname eq [$gui(plotcombo) get]} {
       $gui(plot) plot [list $x $y] with points pointtype filled-circles {*}$args
    }
}
```

Afterword

Auto clear when changing the plot.

```
proc Plot {plotname x y args} {
    global gui
    if {$plotname ni $gui(plotlist)} {
        $gui(plotcombo) configure -values [lappend gui(plotlist) $plotname]
    }
    if {$plotname eq [$gui(plotcombo) get]} {
        # Auto clear when changing the plot
        if {$plotname ne $gui(plotname)} {
            $gui(plot) clear
            set gui(plotname) $plotname
        }
        $gui(plot) plot [list $x $y] with points pointtype filled-circles {*}$args
    }
}
```

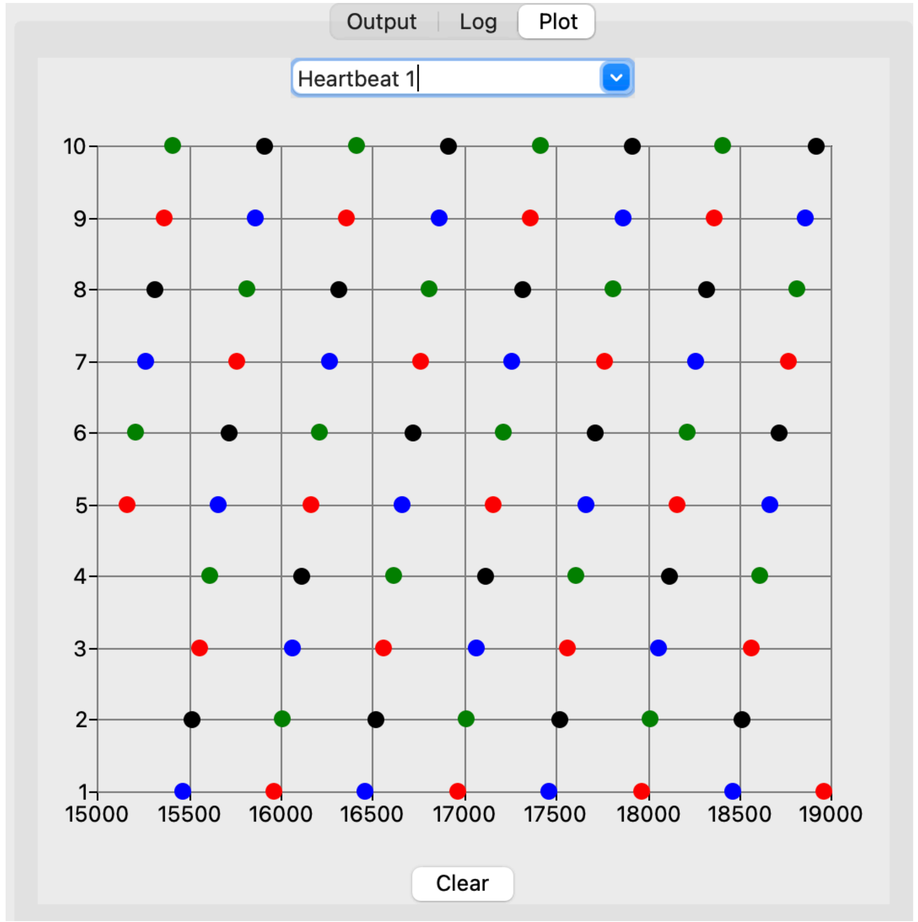
Afterword

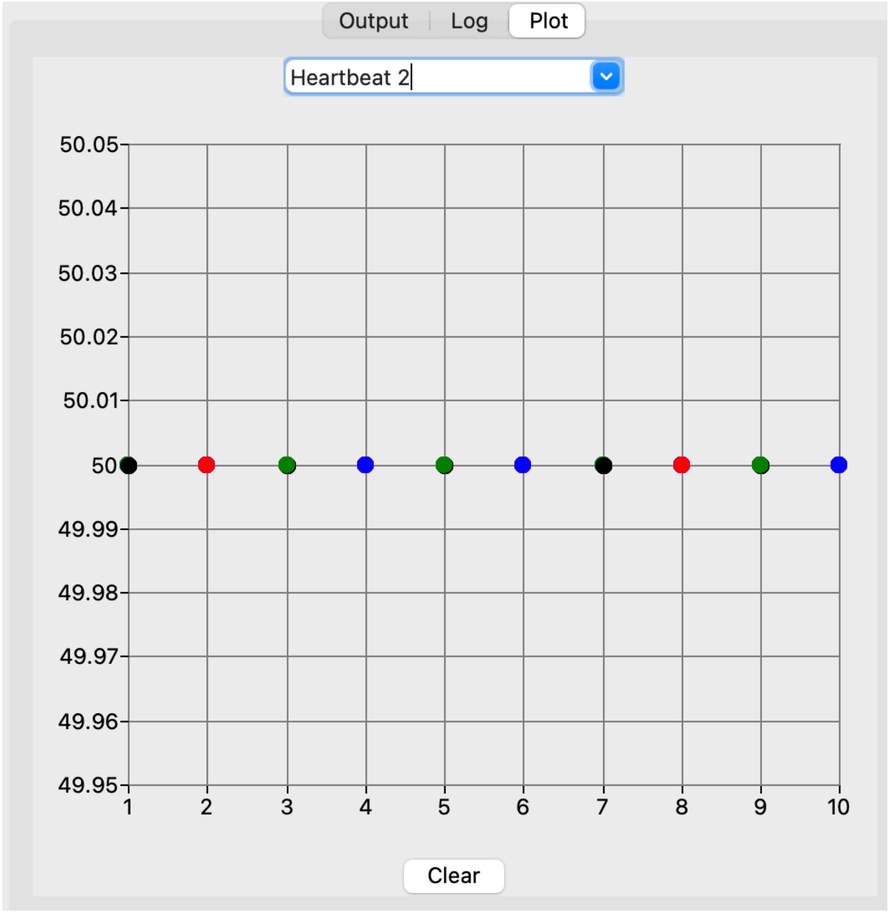
See how the ad hoc usage changes to a permanent feature. Also: a different interpretation of the heartbeat that focuses on the time between the counts. If I never activate the heartbeat trace, it will never appear in the list of plots, therefore not clobbering the list if I've got too many data sources I want to potentially observe.

```
# 2
set last_ticks 0
DTraceAdd 5 "Main sequence timing" {
    uint16_t    start_ticks;
    uint8_t     duration_ticks;
    uint8_t     state_and_half;
} -formatter {
    set state [expr {$state_and_half & 0x7f}]
    set half  [expr {$state_and_half >> 7}]
    # 1
    Plot "Heartbeat 1" $start_ticks $state
    # 2
    set delta [expr {$start_ticks - $::last_ticks}]
    set ::last_ticks $start_ticks
    Plot "Heartbeat 2" $state $delta
    append str "Main seq: $start_ticks ms  $duration_ticks ms  half: $half  state: $state"
}
```

Afterword

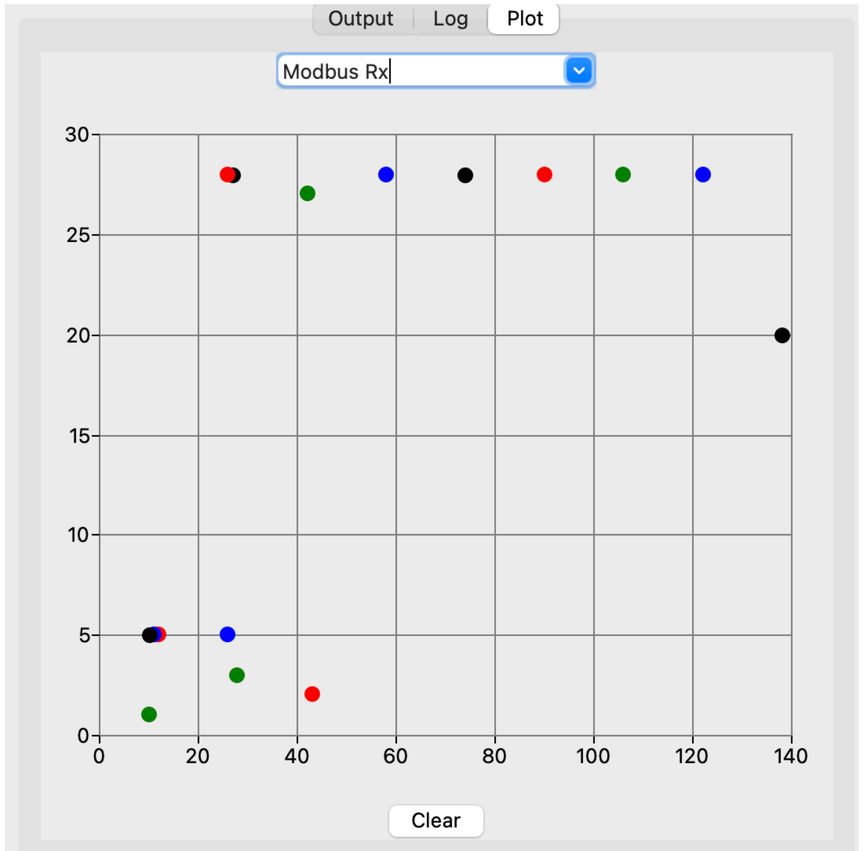
See it in action! Yes, indeed 50 ms between the heartbeat ticks.

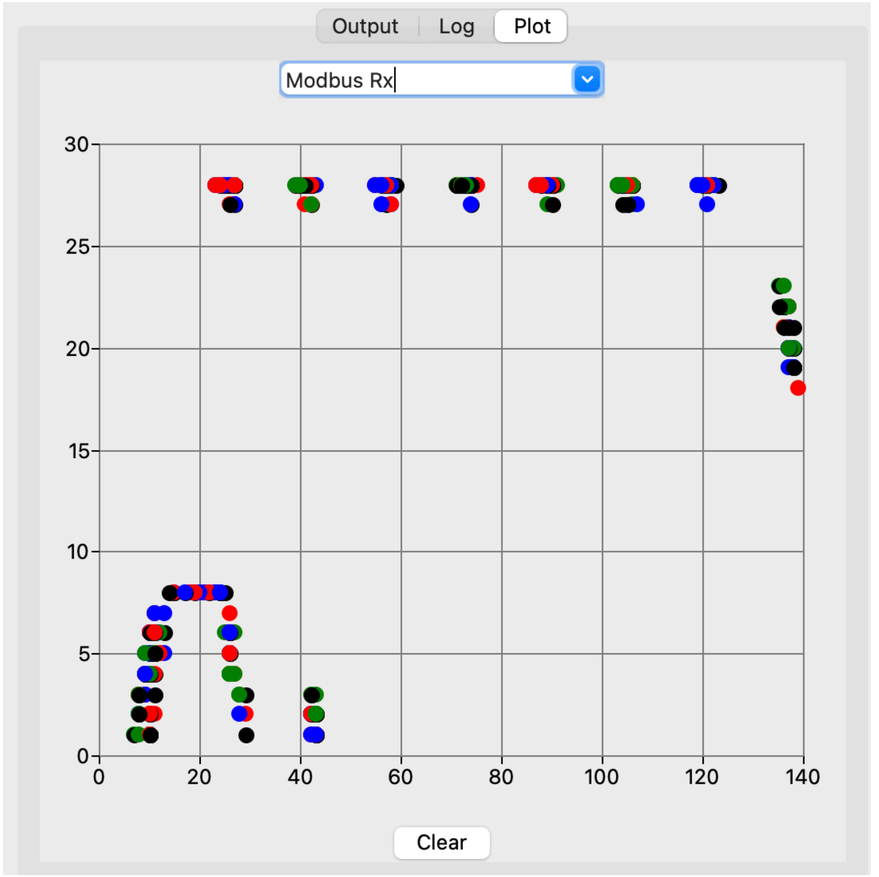




Afterword

Bonus: I mentioned that the character timeout while receiving a telegram is dictated by the I/O latency of the OS (here macOS, but it looks very similar on Windows) and that it is 16 ms. I instrumented the receiver's code with a call to Plot, the x-axis is the time in ms relative to the start of reception, the y-axis is the number of bytes received. The plots show the reception of a short and a long telegram with a single shot in the left plot and an accumulation of about 50 shots in the right plot. Yes, I'm really into this low level stuff.





Afterword

I’ve read somewhere that a presentation is giving the presenter more than it is giving the audience. That was not really my intention but it indeed happened that the talk gave me a lot and every day at work I will look at my application and occasionally think: „Ah, that’s the Vienna code!“

I hope the audience still got something out of my attempt to present a simple design that goes from the bytes on the wire to the dots on a graph.