README

March 7, 2020

1 Developer's guide to DashT wxWebView based instruments

All the suffering started from this unfortunate question: what in the earth can we do with this? :

```
wxWebView           *m_pWebPanel;
```

2 Introduction

The requirement to be able to show engine and energy system data on Dash came first into mind where a browser techniques could be useful and fast (or slow) enough for that type of data. The usage of web techniques would allow to jail break from wxWidget's bit aging GUI and move to the other end of the GUI development paradigm, *i.e.* no limits! After short R&D prototyping and proof-of-concept the limitations of the run-time environment became known as well as the potential communication issues. Robustness and simplicity was put in front and a bit old fashioned web techniques were used because of the simple fact that the wxWebView itself and the back-ends it is using are not necessarily even from this decade!

More modern Signal K data format and especially one of the implementations for it, Signal K server node were chosen to provide the data for the instruments running on a wxWebView platform. In order to avoid multitude of network connections, DashT's Signal K streamer was chosen to feed the instruments by subscription and by a C++ call-back function. That is definitely a modern touch while it allows both to limit number of network connections towards the Signal K server node (which has certainly no problem to deal with those even if that would have been the case, it looks to me) and to continue to feed Tactics derived active calculations instruments from the same source of data.

3 Scope

Scope is in data which is originating from NMEA-2000 data bus.

Excluded data sources are NMEA-2000 navigational information, NMEA-2000 status information, NMEA-0183 all, IoT, GPIO and Bluetooth LE.

Data type scope is in data which can be presented as floating number, in scientific exponential format when converted to string format.

Excluded data type are that of string type of status and other messages and binary status bits (0/1).

4 Target environment

Multi-computer networked boat instrumentation system where computer interconnections takes place over wired or wireless using TCP/IP protocol.

4.1 Operating System

Priority operating system support is Windows 10, Linux Debian 18+ LTS and Raspian on Raspberry Pi 4 or greater (armv7l).

Secondary operating system targets are MacOS and Android.

4.2 OpenCPN

*O*penCPN v5 series is the compatibility required.

4.3 Signal K server node

Signal K server node v1.19.0 or greater is required (subscription to data imposed starting from this version).

4.3.1 Dashboard-Tactics plug-in

Dashboard-Tactics v2 alpha 1 (1.98.001) or superior.

4.3.2 Plug-in framework services

There are many issues with various configurations present in selected targets. It is clear that preferred protocol would be *http://* or even *https://*, but if *file://* is to be used because people do not want to use a HTTP server or they do not know how to set it up, the limitations and somewhat erratic way of interpreting the RFCs of the various backends is causing serious issues, now and certainly in the future!

> Some catastrophe scenarios could be, for example: Windows dropping IE being available for back-end, cf. note(1); Linux moving a the same time to wxWidgets 3.1 (which is good) but also to WebKit 2.x; there must be others threads...

Result observed without no tweaking in the policies of the out-of-the-box installation:

Platform	wxWidgets	back-end	file://	http://	viewport prop. font sizing
Windows	3.1	IE(*max: 8*)	cookie: Y(1) localStorage: N	cookie: Y localStorage: Y	Y *CSS.supports()*: N (but **works**!)
Linux	3.0	WebKit1x	cookie: N localStorage: Y	cookie: Y localStorage: Y	Y
Mac	3.1(?)	(Mac?)WebKit(?)x	cookie: ? localStorage: ?	cookie: ? localStorage: ?	?

(1) After security update of 2020-01-14 = N

Where there is no issues to obtain persistency when the protocol is *http://* it looks like that one

needs to use cookies in Windows and *localStorage()* elsewhere with *file://* protocol. For Mac, we do not know enough yet. This, of course is a bit shaky for the future, even if it works now. It is better to make a parameter storage which automatically adapts ot use *localStorage()* if it is available. This way, plan B can be that people just install that HTTP server and they get the persistent parameters back in case there is an issue with their back-end, which can change.

> Of course, one can imagine to integrate some hefty javascript code to make a standalone server *and* the page, but it seems, for now, overkilling. Let's start with *file://*, plan B being using *http://* with an external HTTP server and, finally, plan C being to use a local configuration file for static configuration.

Managing the above diversity is the main burden in this project and the risk taking should be minimized by selecting only solutions and techniques which are several years old (seeing the back-ends on each supported system). Even then, every implementation needs to be quickly tested on both supported priority platforms, Windows and Linux.

5 Development Paradigm

The resulting instrument shall be tightly integrated in the Dashboard-Tactics plug-in, albeit it has dependencies on one new external data system, Signal K server node for this particular implementation.

Therefore the instrument's functions shall be fulfilled by a hybrid methods: C++ for functions and call-backs for the Dashboard-Tactics plug-in integration and for performance, HTML/JavaScript to provide the GUI functions of the particular, developed instrument.

6 Implementation

Implementation shall be compatible with the development tool chains used in two of the important dependencies: C++ 14 / CMake for plug-in classed and node.js / npm as development platform for modules needed as in Signal K server node.

6.0.1 Base class

In order to allow compatibility with the existing Dashboard instruments and their window handling the base class shall be existing *DashboardInstrument*.

6.1 Abstraction layer class

The class creating and using wxWebView window and communicating with the JavaScript application part of the instrument shall be an abstract class called *InstruJS*.

6.1.1 Implementation class

The class implementing the instrument is implementing an *InstruJS* type of object is managing the interface toward the Dashboard-Tactics plug-in is called *EngineDJG*.

6.1.2 Implementation application

Albeit there is no base class for the implementation JavaScript application the implementation shall be called *enginedjg* with all possible modules with no direct dependencies to the implementation itself collected in a super structure above it, called *instujs* - this in view of in the future develop other instrument types similar to the *EngineDJG* implementation, such as status displays.

7 HTML/CSS/JS application development

The development cycle is a typical one for such an application. However, from the Section **??** follows that one needs always think and continously test on the final target, *i.e.* the worst case real-life environment and - if possible - make the application to automatically adapt to the conditions found on the target system and the intended usage on it.

7.1 Snippets and ideas

You may have your own cloud based favorite to develop your HTML/CSS/JS snippets. If not, why not try https://codepen.io, some snippets from the author are public, of course, and can be forked. Do not hesitate to browse the featured projects of this or any other similar site!

Of course, if you are more discrete and/or hate the cloud, you can use workspaces provided by the browser based tools (see below). Not sure, though about the obtained level of secrecy and the need to have it in this case… Use what you like the best!

7.2 Browser based development and testing

Using modern browser's `Ctrl+Shift+I` which opens the integrated tools for the web developers remains the ideal tool to debug your HTML/CSS/JS application. Nothing beats the possibility to set breakpoints, study the document structure, and to see what CSS sentences are, actually ignored by the browser and which one are doing what you expected them to do.

Because the "*browser*" compatibility - or rather the incompatibility is a real issue in this development, you can see that the `onload` event handler takes the usage of the `CSS.support()` function - when it is available! (Which we first need to detect.)

7.3 Verifying (often) on the target system

Careful out there! Fancy design may work on you browser but does it work on `wxWidgets` `WebView` on WebKit/IE back-end?

It is a good idea to keep open and **regularly** reload your project on the `WebView` based simple browser, called - surprisingly - ***webview***. If that browser is located on the most modest of the targeted platforms, like Raspberry Pi, that's even better.

8 Modular development and packaging

`node.js`, `npm` and `webpack` - well, consider `webpack` being the `CMake` equivalent of HTML/CSS/JS world! (Or worse…)

In case you wonder why there are so many folders and files in the development area, tt is better to read a nice tutorial like this about this dodgy subject with many opinions; the discussion is quite demanding in number of paragraphs...

8.1 Multi-language support

Complicated in JavaScript in general and `webpack` does not help in that. I found this example project: https://github.com/donaldpipowitch/webpack-i18n-example

For now, simple `lang.js` file which is to be replaced with, say `lang-fr.js` file.

8.2 Character-set specific issues

`git` under different systems may change the encoding which can create an issue in the following case:

> Special characters as symbols must match the target systems - Javascript does not particularly know about the UTF-8, it just puts out the characters to the browser. For example, degree character ° in `common.js` is nerve-racking! This is how I managed to get it work in `justgage` `symbol` character: I used `emacs` on a Linux machine and entered the character with `C-x 8 RET B0 RET` which enters an UTF-8 encoded character. Tested OK. Now, syncing the Windows machine with GitHub Desktop. Surprise, the file is now encoded ANSI - one can verify this with `Notepad++` and the degree sign looks funny. **Do not touch it!**. As such, it actually works in IE based WebKit backend. You can test it with IE and it works both in `justgage` but also in the plain HTML (*i.e.* "simple") display.

8.3 Static code check ESLint etc.

The static code check for pull request - or any commit whatsover - is done witch `Codacy.com` and not continous build checks with integrated `ESlint` in the `webpack` - I do not see the need to have both. Also, there is a problem to get the `ESlinit` configuration imported every time it changes into `Codacy.com` so I have given up and I use its on-line tool to set up the rule. You are invited to learn those chosen for this project from https://app.codacy.com/manual/petri38-github/dashboard_tactics_pi/patterns/list - they cannot be exported, unfortunately.

One can discuss about the meaningfulness of any rule, of course but in general, I expect grade `A` code, that means *zero* static check error - with the given rules, of course. I will seriously hesitate if I get a grade `B` pull request reports from `Codacy.com`. So please iterate a few times to get the `A` grade so that time consuming discussions and work can be avoided.

8.3.1 The innerHTML rule

We follow the (somewhat old, in purpose since we are working an older back-ends) https://developer.mozilla.org/en-US/docs/Archive/B2G_OS/Security/Security_Automation

I have ported Firefox's class mentioned in the above instructions in `../src/escapeHTML.js` - use the class as `Sanitizer` - there are plenty of examples of its suggested usage in the code, `grep` them with `innerHTML`.

9 WebView specific issues

Please find and collect also your findings about the wonderfully complex world of the WebView - on differen platforms having different back-ends!

9.1 WebView specific on Linux `__WXGTK__ && wxUSE_WEBVIEW_WEBKIT`

You will find that the on Raspberry Pi, or on any other Linux based system the `wxWidgets` environment is not the same than on your fancy laptop. You can find the *webview* sample from the source distribution, here:

`/usr/share/doc/wx3.0-examples/examples/`

Use the unpack script to unpack the *webview* example, run `make` and then run the application. Load your page on it and see if it works the same. If it does not, see below for debug instructions.

I tried to put TRUE these settings: `enable-file-access-from-file-uris` and `enable-universal-access-from-file-uris`, unfortunately they do not have any effect on the handling of `file://` URIs **if there are cookie-saving involved**: it does not work.

Let's try with this browser: `/usr/lib/arm-linux-gnueabihf/webkit2gtk-4.0 $ ./MiniBrowser --cookies-policy=always` : Proof of Concept (POC) which confirms that while the OpenCPN and wxWidgets still use WebKit1 on Linux, it would not be any better with WebKit2Gtk! (I was planning to try this parameter.)

> Both WebKit1x and WebKit2Gtk silently refuse to save cookies, if the URI is not `http://` or `https://`, respecting the RFC6265. To maintain porting compatibility, use LocalStorage instead, perhaps with JSON parsing.

WebView specific on Mac `__WXGTK__ contribution welcome`: likel, is it `wxUSE_WEBVIEW_WEBKIT` or `wxUSE_WEBVIEW_WEBKIT2`?

9.2 WebView specific on Linux `__WXGTK__ && wxUSE_WEBVIEW__IE`

On Windows: `C:\wxWidgets-3.1.2\samples\webview\vc_mswud` (after the build). Compiled with default settings it works for the development purposes in this project without problem.

9.3 WebView specific on Linux WXGTK

`/usr/share/doc/wx3.0-examples/examples/` Used makefile OK.

10 Debugging on the target system

First of all, make symbolic links from the OpenCPN plug-in installation folders into your development folder: you are going to need it since modifications by trial and error are needed when all you have is a blank screen on ***webview*** when you open your application which works damn fine on your fancy browser! You do not want to spend your life to reinstall the plug-in after every modification.

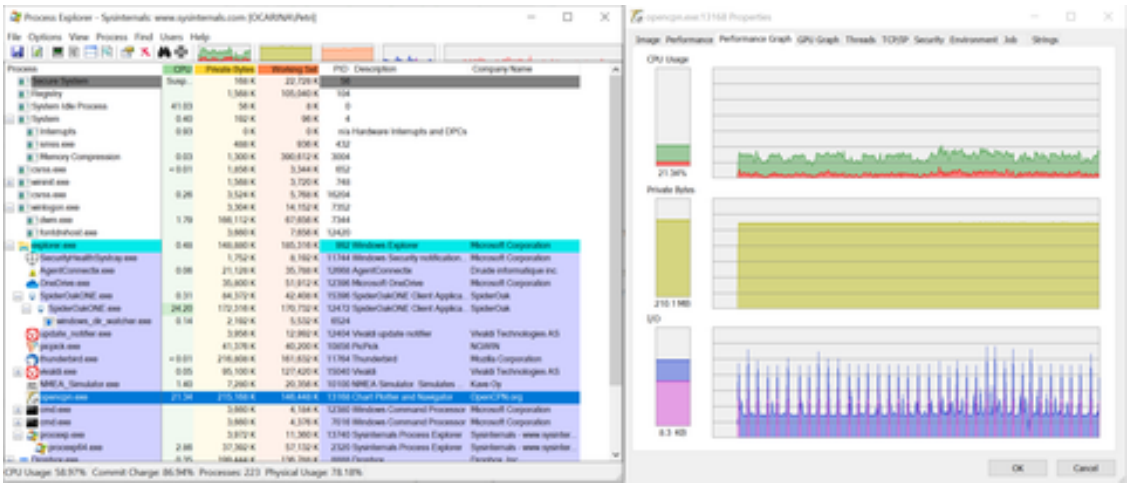
10.1 Performance issues

Executing in a tight loop scripts on multiple windows soon brings up the CPU load!

Tools to observe this are the usual process tools, in Windows `procexp.exe` and on Linux `htop`

Small things to details can make a big difference! In the design paradigm we expect the data coming from engine, not from wind vane, for example. The CPU load should be not big since the dial is not moving so often, like the wind direction dial which is jumping back and forth. But we should not send the useless data to the dial: if it has the value, do not use heavy script execution to send the same value again! Remember, we do not send float values but strings. We make sure that we do not have more than one decimal. And if the value is the same, do not send it to the instrument!

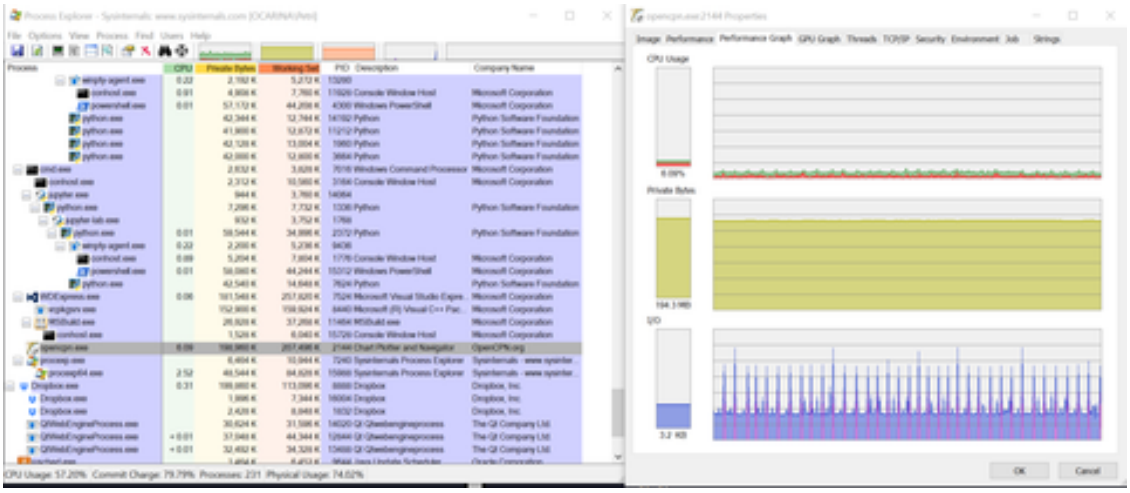
The below process image details is from Windows when there is no filtering of repetitive data with twelve (12!) JS instruments, fed by the NMEA Simulator via Signal K delta channels via Signal K input streamer about 130 floats per second but with static data set (the peaks are InfluxDB Out flusing to a file):



(zoom)

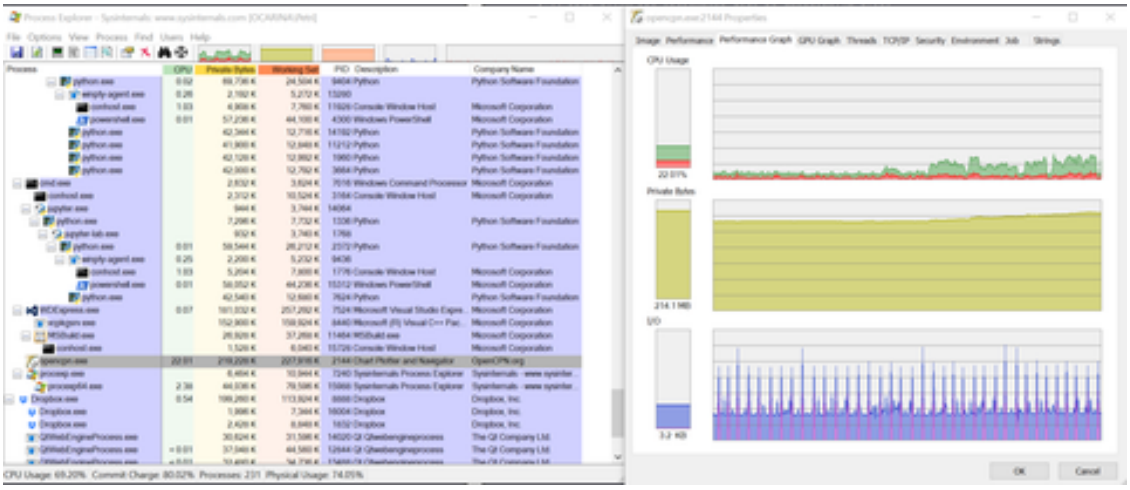
That is huge waste of the CPU power since the dials are not moving at all. Let's make the following change in the script sending code and then observe again:

```
if ( m_istate == JSI_SHOWDATA ) {
    if ( m_data != m_lastdataout ) {
        wxString javascript =
            wxString::Format(
                L"%s%s%s",
                "window.iface.newdata(",
                m_data,
                ");");
        RunScript( javascript );
        m_lastdataout = m_data;
    } // then do not load the system with the same script execution multiple times
} // the instrument is ready for data
```

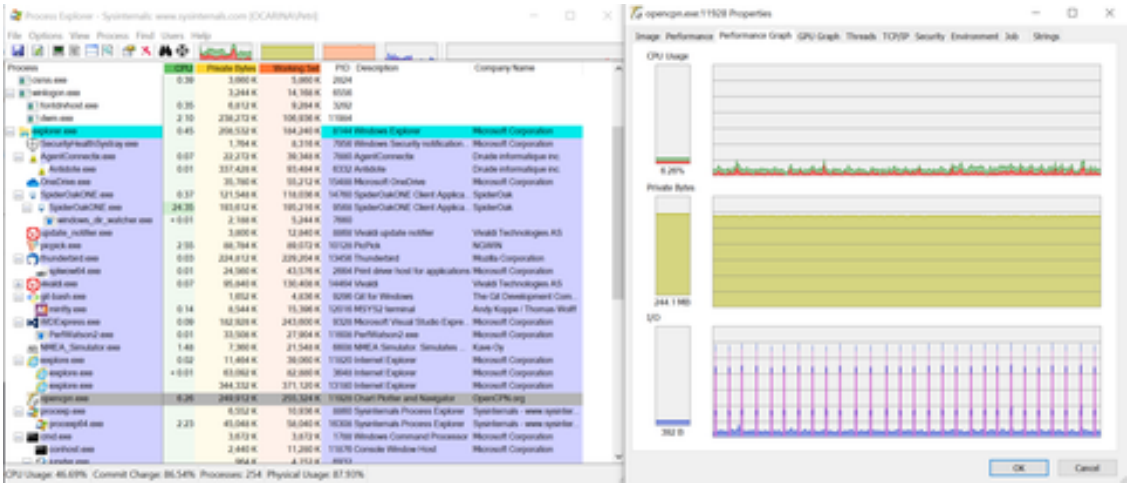

(zoom)

The CPU-load looks quite reasonable around 7 percent w/ i7 CPU and the number of I/O operations is less than half of what it used to be!

Of course, when one creates heavy oscillation in the r.p.m. values, for example, the CPU load goes up, since the SVG-rendering of several dials in this case is entering into the game - there is always a price to pay for jumping dials! But it is noteworthy that the number of I/O operations remain low, nevertheless.


(zoom)

The background load can be further reduced by **randomizing the threads** which are driving the JS instruments:

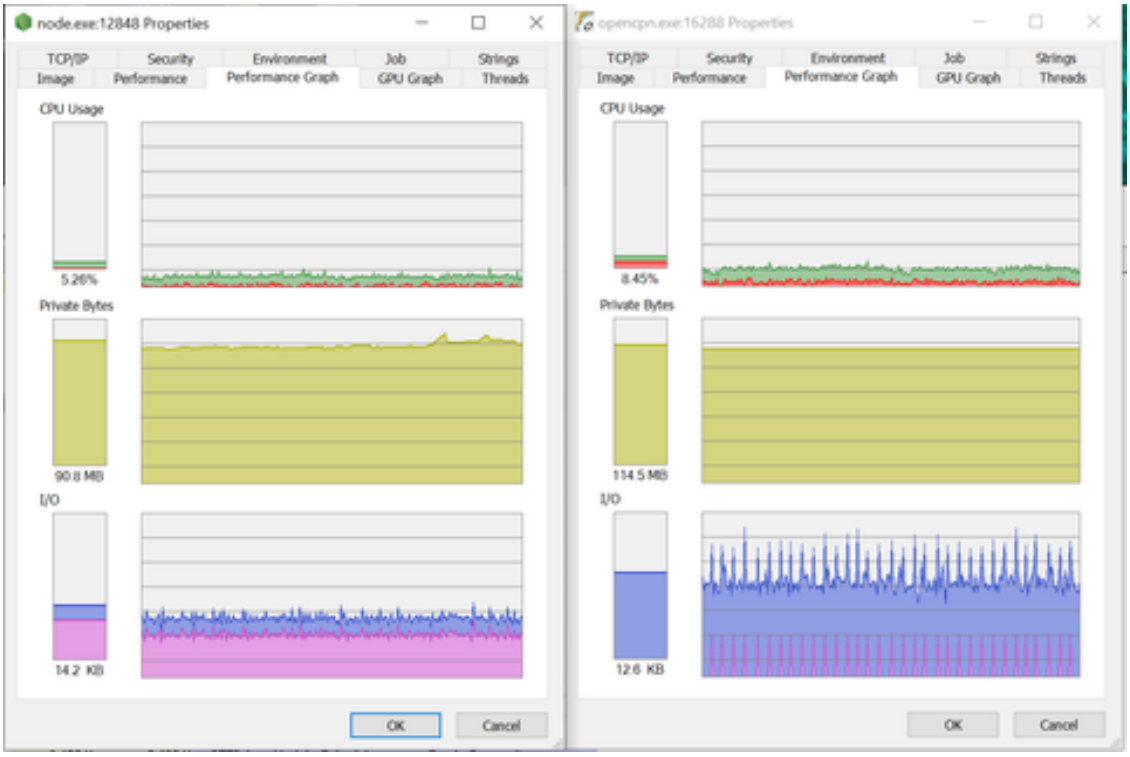

(zoom)

10.2 Overall performance in hosting system(s)

The paradigm of subscription of the JS instruments into the single datasource which is provided by the `streamin-sk.cpp` is efficient by the speed of tha call-backs and their little overhead. It also limits the number of connections to the Signal K server node.

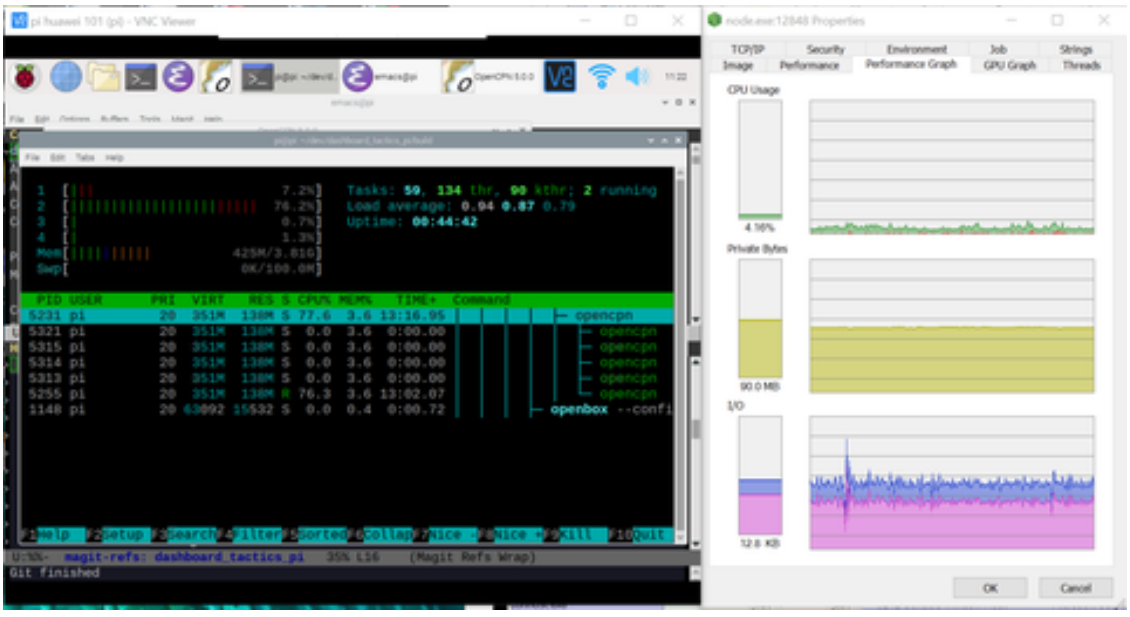
The subscription paradigm probably (but not necessarily) needs to be expanded to that steamer/parser itself. Starting from *SignalK server node v1.19.0* the subscription policy is made mandatory also for the TCP socket data read. This can be used to our advantage regarding the CPU load. The streamer is centralizing the data connection which is good, but once we have set up of what we need to subscribe for, we could reduce the CPU load of that streamer by asking it to subscribe only for the values we are interested in.

With NMEA Simulator we can reach quite considerable amount of messages, which leads the `streamin-sk.cpp` thread to process about 290 - 300 floats per second resulting about 10 - 12 % of constant CPU load (Win20/i7), at the same time `npm` process, hosting the Signal K server node has 7 - 8 % CPU load.



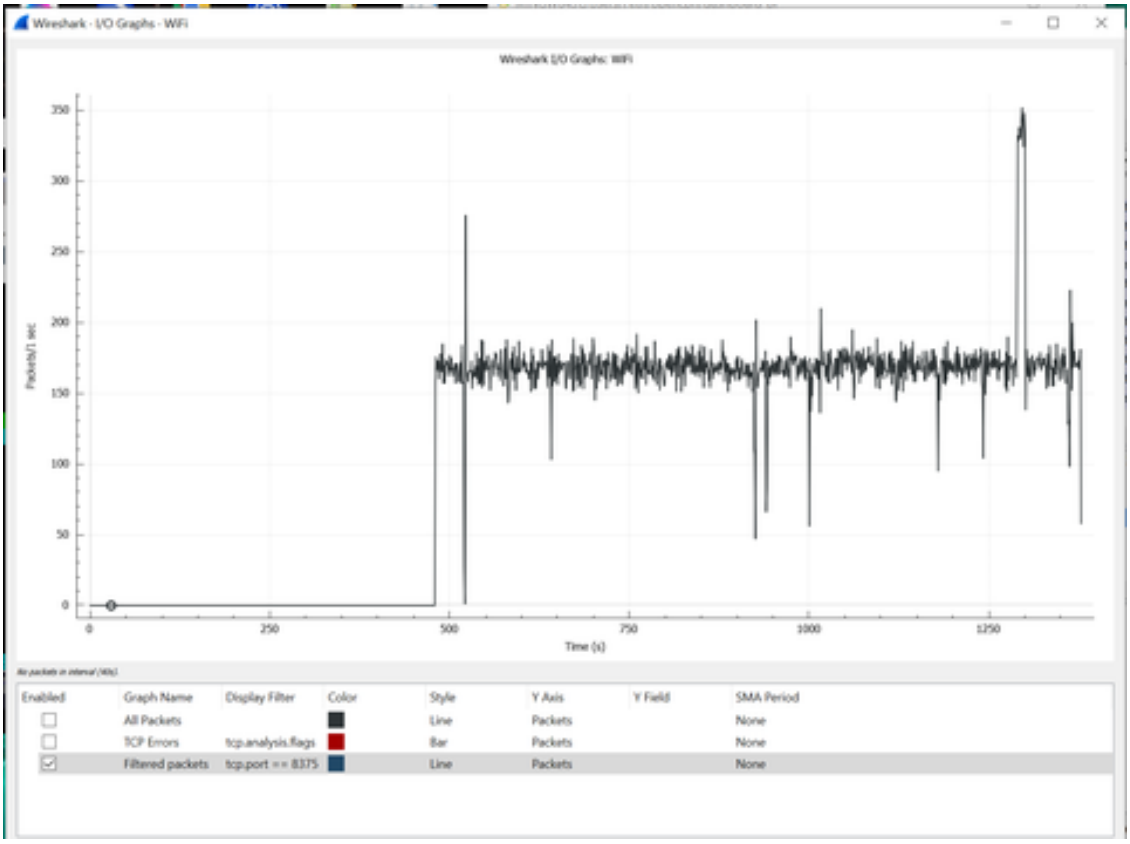
(zoom)

There is no doubt that the network distributed data producing and consuming is a better configuration scheme: the Signal K server on a RPI and the data consumer on another computer, like on Windows laptop or a tablet. Here we make the other way around (because of the NMEA Simulator being a Windows program) but it shows the idea of consuming one CPU core on a RPI 4 for communication tasks:



(zoom)

Resulting network traffic on the TCP port 8375, over WiFi:



(zoom)

11 Help! My fancy application looks rotten / does not work on RPI

Let's consider our design constraints first: we are developing for cross-platform, limited or old-fashioned web application run-time support environment with very small screen or canvas area.

Sounds familiar? Yes, it is like developing a HTML/CSS/JS application for a telephone back in the day!

You need to have the patience - and more importantly - the possibility to carefully check and adjust at pixel level CSS files and debug stubborningly failing JavaScript which worked fine on your browser.

Unfortunately the *iOS* or *Android* USB-based development tool connections to your *Safari* or *Chrome* will not be any good in this case.

There is, however a solution. It is not developed any more (since 2017) but it still exists and **it still works**: WEINRE. Let's hope it is not going away, or that other tools for `wxWidgets WebView` would emerge!

> **Attention vulnerabilities** - *weinre* module has its development stopped and it is as such a very vulnerable product. Use `npm audit` to see the details of those. Make sure to use –save-dev switch when installing to use it only for occasional debugging during the development phase.

While waiting the new tools, I archived this paper (2011) which nicely explains the remote portable device debugging concept and what you can expect and what you cannot expect from WEINRE.

11.1 Installating WEINRE on RPI

To use WEINRE, one needs to have web server. Luckily, on RPI this is easy:

Installing http-server on RPI

If you are reading this, it is highly likely that you are using the excellent SignalK node server on your RPI. That means that you have `node.js` and, with it `npm` package manager.

Install http-server with command `sudo npm install http-server -g`.

Using the command line, move to the directory where your application file root is located and give command `http-server`. Leave it running *et voilà*, you have a web server!

11.1.1 Get WEINRE

```
sudo npm intall --save-dev weinre
```

11.1.2 Configuring WEINRE

In your home folder, create a file `~/.weinre/server.properties` with the following contents:

```
boundHost:    -all-
httpPort:     8081
reuseAddr:    true
readTimeout:  1
deathTimeout: 5
```

11.1.3 Launching WEINRE server

Using (another shell) command line, type `weinre` and leave it running.

Now you should have two servers, `http-server` and `weinre` running.

11.1.4 Opening the WEINRE debugger

Start the RPI's browser, probably *Chromium* (but I am using *Vivaldi*) and navigate to `localhost:8081`.

The server's welcome page, *weinre - web inspector remote* will open. It will give you interesting information and even demos you may want to try first opening them on a **separate** screen or tab.

Likewise, you would open the debugger service, `localhost:8081/client/#anonymous` on a separate screen or tab. This window is now waiting for a connection from your remote (or local) `wxWidgets WebView` based application.

11.1.5 Preparing your application for remote debugging

With "*remote*" we understand your debug server running on the Raspberry Pi, and either a *webview* browser or OpenCPN with its `WebView` class based instruments running on your Windows, Mac or Linux development system

Of course, it is not so "*remote*" if the main debugging target being the *webview* browser or OpenCPN running on Raspberry Pi. But since WEINRE access event those applications through the http-server, it does not make any difference, in fact:

One can have multiple hosts used at the same time, for testing and debugging the same application on different run-time platforms.

Each instance of the application must know where WEINRE server is located. So you need to know your Raspberry Pi's IP-address in your network. Let's say it is 192.168.8.103 : In really "*remote*" test environments you would add the following line in your HTML:

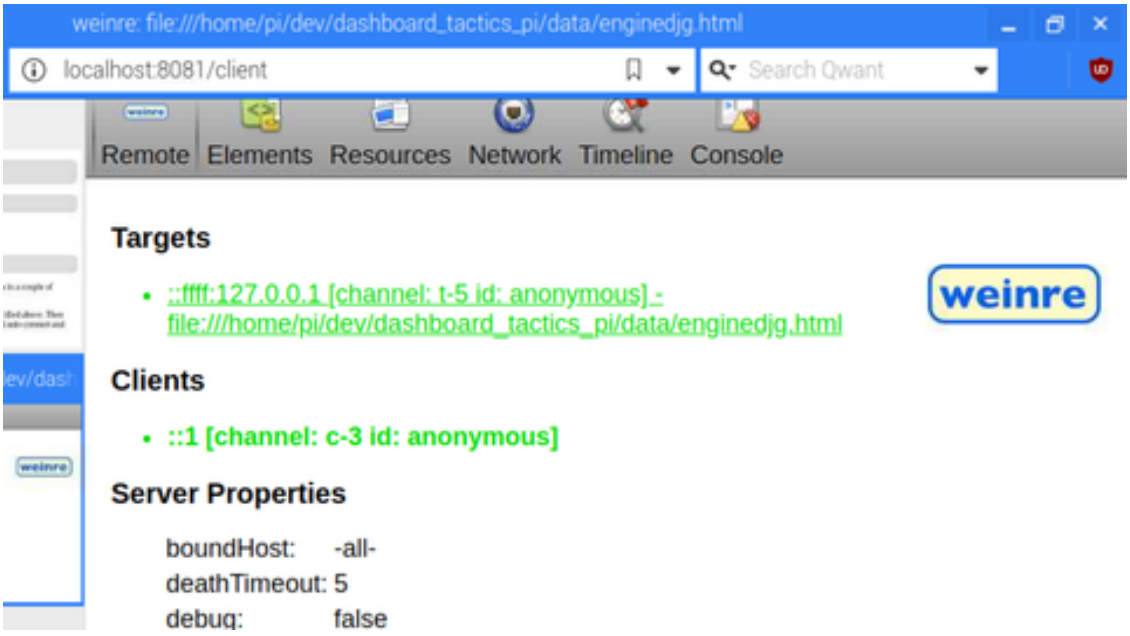
```
<!--script src="http://192.168.8.103:8081/target/target-script-min.js#anonymous"></script-->
```

It is not mandatory to use it in exclusively "*local*" development, you can set:

```
<!--script src="http://127.0.0.1:8081/target/target-script-min.js#anonymous"></script-->
```

Open the your application's HTML-file in *webview* browser - drag and drop works.

As you can see above, this makes the application to download a javascript module from the WEINRE server. It connects you the WEINRE server's debug environment. If you not see the connection under the *Remote* tab, reload the page:



(zoom)

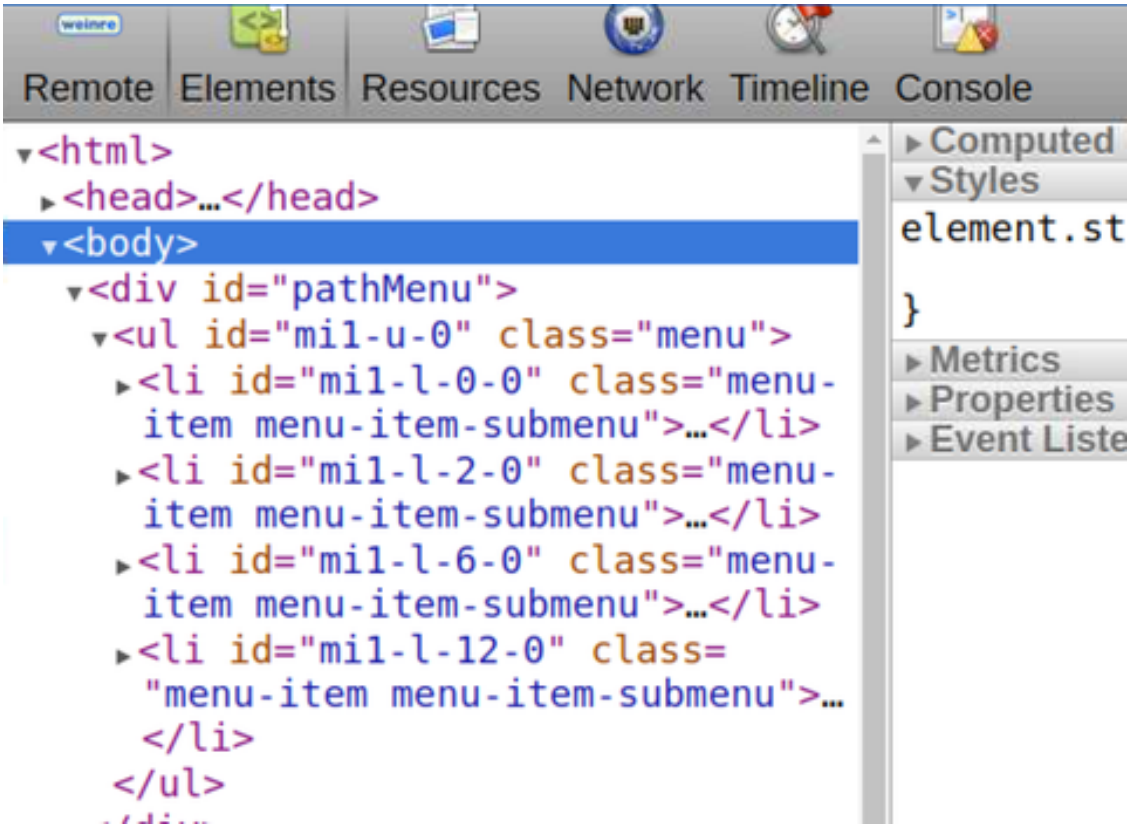
11.1.6 Debugging with WEINRE

Please remember that this is **not** full blown JavaScript debugger like the one which you can find from your browser. Before coming here you have debugged your algorithms and you will use WEINRE only to find and sort out the snagging incompatibility issues between the various run-time environments.

What can one do with WEINRE, then?

11.1.7 Inspect the document structure

In the case of it is your JavaScript which dynamically constructs your document structure contents, it would be a good omen for the rest of the session if you can find actually the intended identifiers in the structure:



(zoom)

Like with the modern browsers, you can select a structure in the "*Elements*" window and it will be highlighted in the *webview* browser, which is quite handy sometimes.

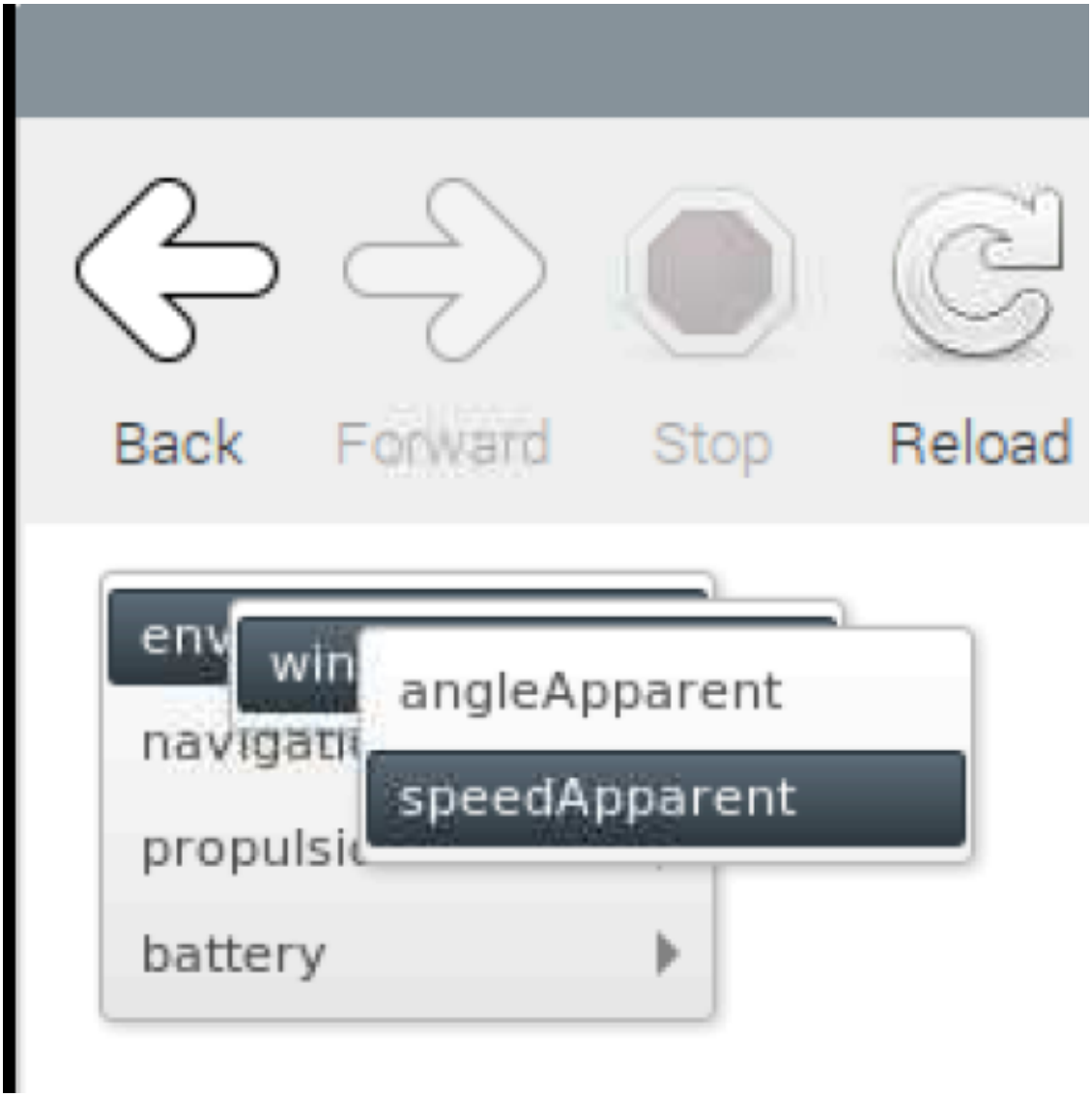
"*Resources*", "*Network*" and "*Timeline*" tabs are quite useless and you should not expect to see anything interesting there.

"*Console*", as the name implies allows you to inspect the variables and show the `console.log()` output.

```
function onMouseDown(e){
    console.log('onMouseDown()');
    document.removeEventListener('mousedown', onMouseDown);
    e= e.srcElement;
```

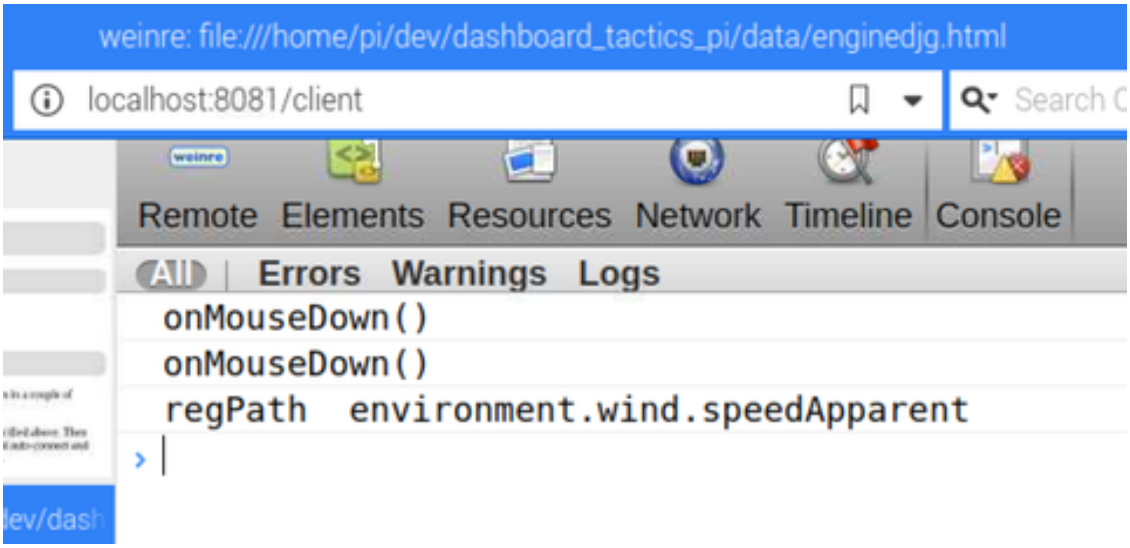
(zoom)

Finally, the answer to the question "*what to do?*" if your screen remains blank or if the mouse event does not work as you expect is the following: Use the good old "*comment out suspicious code blocks until your application loads*"-method!. Then put `console.log()` calls in critical points.



(zoom)

NOTE An older browser with developers tools is often enough to resolve startup problems related to typos in variable names and such (you do not need Weinre for that). Blow the dust out from your obsolete Internet Explorer and hit key **F12**! See this how-to video.



(zoom)

Rudimentary? Yes, but supposing that your algorithms are already tested elsewhere, WEINRE allows you to have the necessary tools to avoid guesswork based development!

12 Debugging Data

First of all, our interface is Signal K node server which puts out Signal K. But it does not hurt to know how does it get the data from the NMEA-2000 bus

> Since nobody can count the number of NMEA-2000 adaptations out there, we use the excellent NMEA-0183/2000 Simulator

12.1 NMEA-2000 data analysis

By far the easiest way is to use the log files provided by a proven Signal K server node device support. It is unlikely that one will find a bug here, but it is good practice to start the signal path debugging as close to the source as possible.

Turn on the log file usage for the connection, let the NMEA-2000 data flow and check the result. You should find something like this:

```
1580932802754;A;2020-02-05T20:00:02.735Z,2,127488,18,255,8,00,ac,0c,d6,02,02,ff,ff
1580932802758;A;2020-02-05T20:00:02.755Z,6,126996,18,255,134,14,05,9a,02,4e,4d,45,41,32,30,30,30
73,69,6d,75,6c,61,74,6f,72,20,65,6e,67,69,6e,65,00,00,00,00,00,00,00,31,2e,
32,2e,30,2e,31,30,39,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,
00,00,00,00,00,31,2e,32,2e,30,2e,30,00,00,00,00,00,00,00,00,00,00,00,00,00,
00,00,00,00,00,00,00,00,00,00,00,00,32,30,39,37,31,34,37,00,00,00,00,00,00,
00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,01,01
1580932802812;A;2020-02-05T20:00:02.811Z,6,60928,18,255,8,fb,ff,bf,ff,00,a0,64,c0
1580932802813;A;2020-02-05T20:00:02.812Z,2,127488,18,255,8,00,ac,0c,d6,02,02,ff,ff
1580932802813;A;2020-02-05T20:00:02.813Z,2,127488,18,255,8,00,ac,0c,d6,02,02,ff,ff
1580932802813;A;2020-02-05T20:00:02.813Z,2,127488,18,255,8,00,ac,0c,d6,02,02,ff,ff
1580932802814;A;2020-02-05T20:00:02.813Z,2,127488,18,255,8,00,ac,0c,d6,02,02,ff,ff
1580932802889;A;2020-02-05T20:00:02.814Z,2,127489,18,255,26,00,4c,0a,66,0e,5d,75,32,05,83,01,e1,
```

```
a7,00,01,00,1f,00,ff,00,00,00,00,43,3d
1580932802889;A;2020-02-05T20:00:02.889Z,2,127488,18,255,8,00,ac,0c,d6,02,02,ff,ff
1580932802890;A;2020-02-05T20:00:02.889Z,2,127488,18,255,8,00,ac,0c,d6,02,02,ff,ff
...
```

The first line `1580932802754;A;2020-02-05T20:00:02.735Z,2,127488,18,255,8,00,ac,0c,d6,02,02,ff,ff` is 127488 Engine Parameters, Rapid Update

From nmea2000_v1-301_app_b_pgn_field_list.pdf

```
Field # Field Description
Provides data with a high update rate for a specific engine in a single frame message. The first
which engine.
1 Engine Instance
2 Engine Speed
3 Engine Boost Pressure
4 Engine tilt/trim
5 Reserved Bits
```

See also (there are some contradictory information, probably vendor dependencies!): * NMEA2000-explained-white-paper.pdf * http://continuouswave.com/whaler/reference/PGN.html * NMEA2K_Network_Design_v2.pdf) :

```
6.9 PGN 127488 - Engine Parameters, Rapid Update
Field 1: Engine Instance - This field indicates the particular engine for which this
data applies. A single engine will have an instance of 0. Engines in multi-engine
boats will be numbered starting at 0 at the bow of the boat incrementing to n going
in towards the stern of the boat. For engines at the same distance from the bow are
stern, the engines are numbered starting from the port side and proceeding towards
the starboard side.
2: Engine Speed - This field indicates the rotational speed of the engine in units of ¼
RPM.
3: Engine Boost Pressure - This field indicates the turbocharger boost pressure in
units of 100 Pa.
4: Engine tilt/trim - This field indicates the tilt or trim (positive or negative) of the
engine in units of 1 percent.
5: Reserved - This field is reserved by NMEA
```

1: Engine instance = 00

2: Engine speed: AC/0C = 0xCAC = 3244 . . . $3244 \div 4 = 811 r.p.m.$ (NMEA Simulator shows 811 OK)

3: Engine Boost Pressure: D6/02 = 0x2D6 = 726 . . . $726 * 100 Pa = 72.6 kPa$ (NMEA simulator was showing 72.6kPa at this moment OK!)

4: Engine Tilt/trim: 02 . . . 2% (NMEA simulator was showing +2% OK)

Let's try with a negative value, with the simulator we set trim to -24°

We read now: 1581197536148;A;2020-02-08T21:32:16.139Z,2,127488,18,255,8,00,60,12,d6,02,**e8**,ff,ff

e8 . . . 100 - e8 = 0x18 = 24

Suspected anomaly in Signal K server node v1.21.0 conversion (maybe that the driver is working correctly with intended hardware? In this case, the hardware is screwed...): $\pm$ values are converted in Signal K in an incoherent manner: like +2 becomes 2.0E-2 but -24 remains -12.0 and not -1.2E-1 as expected. I am fixing this in the subcscription callback of `instrujs.cpp` now.

127489-sentence We read: `1580932802889;A;2020-02-05T20:00:02.814Z,2,127489,18,255,26,00,4c,0a,66,83,01,e1,d9,a7,00,01,00,1f,00,ff,00,00,00,00,43,3d`

```
6.10 PGN 127489 - Engine Parameters, Dynamic
Field 1: Engine Instance - This field indicates the particular engine for which this
```

(see above, engine data is same)

```
2: Engine Oil Pressure - This field indicates the oil pressure of the engine in units of
100 Pa.
```

4C/0A = 0xA4C = 2636 . . . $2636 * 100Pa = 263.6kPa$ (NMEA simulator was showing $263.2kPa$? $\equiv$ A48 . . . 48/0A ? also Instrujs shows 264 kPa - looks like a rounding error!)

```
3: Engine Oil Temperature - This field indicates the oil temperature of the engine in
units of 0.1°K.
```

66/0E = 0xE66 = 3686 . . . $368.6°K \approx 95.6°C$ (NMEA simulator was showing $95.6°C$ = OK!)

We read: `1580932802889;A;2020-02-05T20:00:02.814Z,2,127489,18,255,26,00,4c,0a,66,0e,5d,75,32,05,83,01,e1,d9,a7,00,01,00,1f,00,ff,00,00,00,00,43,3d`

```
4: Engine Temperature - This field indicates the temperature of the engine coolant in
units of 0.1°K.
```

5D/75 = 0x755D = 30045 . . . $300.45°K \equiv 27.3°C$ (NMEA simulator was showing $27.3°C$ OK, so does the *instrujs*)

Therefore : the units are `0.01°K` and **not** `0.1°K`

We read: `1580932802889;A;2020-02-05T20:00:02.814Z,2,127489,18,255,26,00,4c,0a,66,0e,5d,75,32,05,83,01,e1,d9,a7,00,01,00,1f,00,ff,00,00,00,00,43,3d`

```
5: Alternator Potential - This field indicates the alternator voltage in units of 0.01V.
```

32/05 = 0x532 = 1330 . . . %13.3V% (NMEA simulator was showing the same OK).

```
6: Fuel Rate - This field indicates the fuel consumption rate of the engine in units of
0.0001 cubic meters / hour.
```

83/01 = 0x183 = 387 . . . $387m^3/h * 0.0001 \equiv 38.7l/h$ which is the value NMEA simulator is showing OK

```
7: Total Engine Hours - This field indicates the cumulative runtime of the engine in
units of 1 second.
```

We need to take another example than the rest what is discussed above and below, since the hour counter is running all the time!

1581024474143;A;2020-02-06T21:27:54.141Z,2,127489,18,255,26,00,4c,0a,66,0e,5d,75,32,05,83,01,**59,3a,b0,00**,01,00,1f

59 / 3a / b0 / 00 = 0x 00 b0 3a 59 = 11000289

$11549273s \equiv 192487.88m \equiv 3208.13h$

(NMEA simulator is showing 3213.8 hours right now, so the data is simply a bit old, from the log file)

We read: `1580932802889;A;2020-02-05T20:00:02.814Z,2,127489,18,255,26,00,4c,0a,66,0e,5d,75,32,05,83,01,e1,d9,a7,00,01,00,1f,00,ff,00,00,00,00,43,3d`

`8: Engine Coolant Pressure - This field indicates the pressure of the engine coolant in units of 100 Pa.`

01/00 = 0001 . . . 100Pa = 0.1kPa (NMEA simulator is showing exactly this value OK)

`9: Fuel Pressure - This field indicates the pressure of the engine fuel in units of 1000 Pa.`

1F/00 = 0x001F = 31 . . . $31 * 1000Pa \equiv 31kPa$ (NMEA Simulator was showing 30.7kPa OK)

We read: `1580932802889;A;2020-02-05T20:00:02.814Z,2,127489,18,255,26,00,4c,0a,66,0e,5d,75,32,05,83,01,e1,d9,a7,00,01,00,1f,00,ff,00,00,00,00,43,3d`

`13: Percent Engine Load - This field indicates the percent load of the engine in units of 1 percent.`

43 = 67 . . . 67% (NMEA Simulator was showing 67% OK)

`14: Percent Engine Torque - This field indicates the percent torque of the engine in units of 1 percent.`

3D = 61 . . . 67% (NMEA Simulator was showing 61% OK)

12.2 Debugging using Python CAN and Serial Python

There are as many alternatives for debugging NMEA-2000 bus - which is essentially an enhanced CAN-bus - as there are adapters to connect into it. You are welcome enhance this section with your knowledge about those. Meanwhile, in order to have an idea how the NMEA-2000 device support of Signal K server nodes sees the data, let's study a serial line connected NMEA-2000

```
import serial
ser = serial.Serial('COM30')
s = ser.read(100)
print(s)
```

One gets something like this:

```
b'\xf2\x01\xff\x12\x00\x00\x00\x00\x08\x00<\x08\xd6\x02\x02\xff\xff0\x10\x03
\x10\x02\x93%\x02\x01\xf2\x01\xff\x12\x00\x00\x00\x00\x1a\x00L\nf\x0e]u2\x05
\x83\x01\x10\x10\xb4\xb5\x00\x01\x00\x1f\x00\xff\x00\x00\x00\x00C=\xb8\x10
\x03\x10\x02\x93\x13\x02\x00\xf2\x01\xff\x12\x00\x00\x00\x00\x08\x00<\x08
\xd6\x02\x02\xff\xff0\x10\x03\x10\x02\x93\x91\x06\x14\xf0\x01\xff'
```

Which is not that good, since one has both hex value and - when the interpretation is possible - ASCII values mixed.

This should, theoretically at least, work but does not show anything, probaly because of the protocol differences between CAN and enhancing NMEA-2000. We keep it here in case somebody has an idea how to get it work.

```
import can

bus = can.interface.Bus(bustype='serial', channel='COM30', bitrate=115200)
for msg in bus:
    print(msg)
```

One can find attempts to print structured data without the aboev class, but they are not apated to NMEA-2000:

```
import serial
import struct

ser = serial.Serial('COM30')

fmt = "<IB3x8s"

while True:
   can_pkt = ser.read(16)
   can_id, length, data = struct.unpack(fmt, can_pkt)
   data = data[:length]
   print(data, can_id , can_pkt)
```

We get something like this:

```
b'\xff\x12' 328401424 b'\x10\x02\x93\x13\x02\x00\xf2\x01\xff\x12\x00\x00\x00\x00\x10\x02'
b'\x00\x00\x00\x00\x86\x14\x05\x9a' 335974803 b'\x93\x91\x06\x14\xf0\x01\xff\x12\x00\x00\x00\x00
b'0 simula' 1162694146 b'\x02NMEA2000 simula'
b'ne\x00\x00\x00\x00\x00\x00' 544370548 b'tor engine\x00\x00\x00\x00\x00\x00'
b'09\x00\x00\x00\x00\x00\x00' 841888000
...
```

Pure hex is the best, finally, one can search for `F201` which is hex signature of PGN `127489`.

```
import serial
import struct
import binascii

ser = serial.Serial('COM30')

while True:
   can_pkt = ser.read(4)
   print(binascii.hexlify(can_pkt))
```

Now, let's look from this data, for example, PGN 127489, Field, 3: Engine Oil Temperature.

b'10029325' b'0201**f201**' b'ff120000' b'00001a00' b'4c0a**660e**' b'5d753205' b'8301802e' b'b8000100' b'1f00ff00' b'00000043'

Yes, it is still there, value 0x0e66 (see above for the interpretation).

I reckon that now we can just trust the Signal K driver's author what comes to the debug printing!

12.3 Observing the data coming from Signal K server node delta channel

Starting from v1.19.0 the TCP delta channel requires also subscription without exception. Good for the standard and overall charge of the server but bad for debugging with an standard browser. One needs to be able to send the subscription in non-human readable JSON-format and - strangely - no headers are accepted before that, only the JSON structure! That makes the usage of the developer tools a bit awkward in the browser since how to compose such a message using the developer tools?

In case you manage to get the Signal K input streamer streaming something (by default, it asks for everything), you can increase its debug level in its JSON-configuration file **above reasonable** (*the thread will fail in its real-time parsing job because it needs to interrupt its running to call for the log file writing from the plugin process, also the log-file gets really quick really full, be warned*):

From `streamin-sk.cpp`:

```
if ( m_verbosity > 5 ) {
   m_threadMsg = wxString::Format(
      "dashboard_tactics_pi: Signal K type (%s) sentence (%s) talker (%s) "
       "src (%s) pgn (%d) timestamp (%s) path (%s) value (%f), valStr (%s)",
       type, sentence, talker, src, pgn, timestamp, path, value, valStr);
       wxQueueEvent( m_frame, event.Clone() );
} // then slowing down seriously with the indirect debug log
```

From `streamin-sk.json` (in data directory):

```
"streaminsk" : {
   "source"          : "localhost:8375", // not limited to localhost
   "api"             : "v1.19.0",        // version of Signal K server
   "connectionretry" : 5,                // [s] (min.=1s to reduce CPU load)
   "timestamps"      : "server",         // Signal K "server" or "local"
   "verbosity"       : 6                 //0=no,1=events,2=verbose,3+=debug
```

You will be well served:

```
7:53:48 PM: dashboard_tactics_pi: DEBUG: Signal K JSON update server received delta-message:
{
   "context" : "vessels.urn:mrn:signalk:uuid:e5a702ea-0cb8-42cd-8f06-08c43bb5a4b6",
   "updates" : [
      {
         "source" : {
            "src" : "18",
            "label" : "Emu2000",
            "pgn" : 127489,
            "type" : "NMEA2000"
         },
         "$source" : "Emu2000.18",
         "timestamp" : "2020-02-08T18:53:47.921Z",
         "values" : [
```

```
            {
               "path" : "propulsion.port.temperature",
               "value" : 300.45
            }
         ]
      }
   ]
}
```

```
7:53:48 PM: dashboard_tactics_pi: Signal K type (NMEA2000) sentence () talker () src (18) pgn (1
```

12.4 C++ debugging for the subscribed value

> The key point is to have only **one** *instrujs* instrument subcribed to data under inspection, in our example the `propulsion.port.temperature` path. Please remind that we have C++ object, subscription based data - it s getting pretty complex pretty quickly from the debugging point of view!

With your (single) subcribed instrument active and working and hopefully even showing some value, put your breakpoint here in `instrujs.cpp`:

```
void InstruJS::PushData(double data, wxString unit, long long timestamp)
{
    if ( !std::isnan(data) ) {
        setTimestamp( timestamp );  // Triggers also base class' watchdog
...
```

Using the above, example debugging data, you should see the same value, *i.e.* 300.45.

12.5 Javascript data debugging

For performance reasons, `data.js` is not constantly printing debug information to the `console.log()`. But nothing prevents you to add such a statement in there. But since you will be running within the OpenCPN, you would probably need to use `weinre`-tool as discussed above to see the log output.

Instead, I have used slow but more verbose method of using a browser and its developer's tools. On can give the same commands through the window.iface.* methods than the `instrujs.cpp` is doing (much faster). With few such commands (with fake ID string but keep it always the same, like `555-666` to profit the persistence), one can then issue `iface.newdata()` commands.