README

September 24, 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 DashT 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

The scope is for engine data and which is directly coming from NMEA-2000 data.

Excluded data as for now in the engine catetory are NMEA-2000 status information, NMEA-0183 all, IoT, GPIO and Bluetooth LE but this statement can evolve when new needs and instruments are invented.

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).

The scope of NMEA-0183 data is in all navigational data which can be received from OpenCPN or directly from a Signal K node server.

From the direct connection scope follow that out of the scope is Signal K data received from OpenCPN v5.2 or superior.

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 20+ LTS and Raspian on Raspberry Pi 4 or greater (armv7l).

Secondary operating system targets are MacOS and Android.

4.2 OpenCPN

OpenCPN 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

Platform	wxWidgets	back-end	file://	http://	viewport prop. font sizing
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.

5.1 An example candidate for a hybrid C++/JS instrument

Let's take the example of the excellent `avg_wind.cpp` - in fact, ass DashboardInstrument, this is - like other instruments - actually a custom-control widget, derived from wxControl.

Having one actual wx Widget non-static control in it (wxSlider) seems to be OK and, below we draw extra functions on canvas. However, adding more wxControls does not work - in wxControl Sizers, for example, do not work, controls would just keep piling up. If this is needed, one could move this as a specific pop-up window (since one can have only one of these, anyway).

But another way could be make a hybrid AverageWind instrument. It would first have a simple instrument with canvas drawing in C++ (the actual one, it excels in that compared to what one could do in JavaScript even though in JavaScript the graphs are a *bit* easier…) would create a wxWebView based JavaScript instrument.

This new class would derive from InstruJS-class and the resulting JavaScript instrument panel can show all sorts of fancy controls and derived information. However, for GTK2 based older systems

it would be necessary to keep the existing avg_wind.cpp instrument. In this case, the calculating AvgWind class should be separated from avg_wind.cpp.

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 Dealing with strangeness of wxWebView and environment

They are best dealt with in the base, `InstruJS` class. But in some cases, it is better to deal with the the vicious issues with wxWebView at the derived class level.

For example, wxWebView does not like at all (when this is written, wxWidgets v3.1.2) if one tries to load the page content and if the server is not actually there. When you move forward in the JavaScript initialization it is a core dump.

But knowing what file and from where need to be loaded, is known by the derived class. That's why the base class provides methods like `testURLretHost` and `testHTTPServer`. See examples of the derived classes how to use them.

6.1.2 C++/JS communication concentrated in the abstraction class

This design choice is somewhat limiting and can be later revised: this way, we need to have / will have in face of us an equivalent `iface.js` class, always the same.

Advantages of InstruJS.cpp/iface.js The advantage of this approach is that there will be less code copy paste caused duplication since many interface functions are common to each JavaScript instrument.

The second advantage is that the very complicated state machine management in the C++ side but also in the JavaScript side has at least one common nominator and, when developing a new JavaScript instrument one does not need to reinvent the complicated messaging process every time.

The third advantage is that a common nominator *iface.js* interface would allow to write an interface to a NodeJS-process which - in turn - could be made to **enable an instrument service outside of the OpenCPN ecosystem**, on a browser of the telephone in the cockpit, for example.

Disvantages of InstruJS.cpp/iface.js The disadvantage of this is that we need to know in the C++ abstract class about the questions the JavaScript may ask or the data they may want to receive. Also, in JavaScript side, the `index.js` or `index.ts` modules need to deal with the commands via this interface they actually do not need.

> Experience has shown that each JavaScript instrument is different and with freedom of choice also the requests for data or functions towards the C++ InstruJS class are almost impossible to design in advance, apart some basic functions. The `iface.js` class is increasing and increasing, and theoritically you need to revisit and rebuild all existing classes.
>
> A chore work to split the interface by recognizing the useful, common functions and instrument specific functions will be ahead if this work continues. It would also mean that the classed derived from `InstruJS.cpp` would need to be able to talk to this interface to create events on JavaScript instrument.

Complexity of the state machine wxWidgets vs JavaScript There are some limiting design factors like the slow execution of JavaScript code in wxWidgets via wxWebView. One wants to do less actions possible using it. That's why it is used to trigger events in the JavaScript side.

This fits nicely with the JavaScript GUI design which, itself should always be state machine driven (more about it below).

In each JavaScript instrument you can find a .dot file which is `Graphviz` input file. This .dot file can be retrieved via a function provided by the `iface` class. Take a look at the output .svg and .png files to visualize the JavaScript instrument's states and especially the transitions.

You got it: part of these transitions are internal event but some of them are coming from C++ abstract class. For example, when the JavaScript instrument asks something from the OpenCPN Plug-in, the answer is first put in `iface` object and then an appropriate event is triggered.

This is the most efficient way from the JavaScript side.

Unfortunately the efficiency is not so good in the C++ side, driven by wxWidgets threads, which cannot be running at full speed all the time only for this - that's why we used timer based polling. This makes the *InstruJS* class communication state machine to look pretty complex:

There is the communication state machine, which is driving the sub-state machine, which is describing the state of the JavaScript instrument - or rather the `iface` object of it, the abstract class is not supposed to know other details of the instrument.

Due to monotonic polling thread, we may arrive into it at any state of the communication itself. But in most of the time, it is a new demand. So, we want to move as quickly as possible and without waiting for the next poll cycle to the next state in this case. This makes it quite complicated since, of course there are exceptions to the question-answer paradigm. For example, when path or data updates are required they are send to the instrument without further handshake, of course.

This is to say that it might be necessary to revise the communication scheme so that the messaging would be split in two, allowing the derived classes to deal with their own specific communication issues with, say a derived `ifaceMyClass` interface/communication class while the actual `iface` class and the interchanges with it would be limited only to basic functions.

Anyway, once the current implementation has reached the `serving` state, the functions of it are quite easy to implement, in a classical case-structure.

6.1.3 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*.

6.1.4 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 TypeScript

Usage of TrueScript is **highly recommended** to increase transpiling-time type and variable name checking, but only whenever it is applicable with a reasonable effort. In some real-life use cases and issues, mainly caused by IE as a WebView back-end, it is not always possible to have .ts module but .js is more usable, especially if none of the older library dependencies do not have any definition files for TypeScript. In this case, maybe the entire instrument can be written in plain JavaScript, there is no point to make this a self torture because of the past mistakes of somebody else. In most cases, you will find the mixture of the two in the current development tree and also examples how this can be done with `webpack`.

8.2 Polyfill and wxWebView back-end compatibility

Above we have addressed the issue of wxWidgets' wxWebView back-ends being so archaic and thus forcing us to use polyfill techniques so that we can use some reasonably modern libraries. In this section two approach to maintain back-end compatibility of the created bundles is explained.

> One should no underestimate the effort and the time this work take in the early phases of the project, also it requires constant attention and testing when new features are added.

8.2.1 Simplistic JavaScript for simple instruments

The `enginedjg` instrument is like this, simple and based on hand-picked, a bit old fashioned libraries. For example, the author has participated to the JustGage project with a PR, just in order to maintain this good project's compatibility with IE!

Also, the author has chosen the stalled `javascript-state-machine` instead of some modern state machine more adapted to run large and complex page UIs. But since the year 2018 `javascript-state-machine` works with wxWebView both on Linux and Windows while the more modern ones will not (at least without some work), it was chosen.

Even when coding manually simple JavaScript, some nerve-wracking issues appear in coding for IE and one should not be surprised of some solutions chosen to code, for example event handling - it is a fruit of trial-and-error to make it work both on Linux and Windows platforms.

Some manually entered polyfills were copied from the unlimited source of good ideas in *stackoverflow.com*, typically dealing with an *Object* - see as an example the common source module `confvalid.js` for an example.

8.2.2 Babel / core-js v3 based polyfill for complex instruments

Example for this is `timestui` graphs instrument: while, after a really long search, it was possible to find a line graph library which works also on IE (`tui`) and has a polyfill-enabled distribution, it is not always possible to find such a library.

Example is *influxdb-client* (for v2.0 DB) - it is written in TypeScript and uses the latest JavaScript techniques for asynchronous operations. Of course it is not at all compatible out of the box with IE. Decision was made to adapt the application to make mixed JavaScript / TypeScript approach, where `influxdb-client` is now downloaded as source code and integrated in TypeScript part in Webpack. Of course, only a limited part of the source code is needed from the library and only those files need to be modified, since in this application we only read values and we do not plan to write to the DB, it is done in C++ thread already.

Now Babel and core-js v3 are used to make the polyfill.

You have take note that more or less same options are repeated in `webpack.config.js` and `tsconfing.json` for core-js: this is because the TypeScript transpiling is a process of its own, and is launched by the loader for this part of the build chain - one for TypeScript files and one for JavaScript files. We can take advantage of this having different strategy for JavaScript files (*entry* - explicit request for massive core-js usage per module) and for TypeScript file (*usage* - automatic Babel analysis and usage only of those polyfill deemed necessary for target).

Still, not everything works with Babel / core-js: `influxdb-client` uses *AbortController* object with *fetch*. It is still *experimental* according Mozilla! Well, see `FetchTransports.ts` how the author managed to make it work, after many good advises from the web, some of which were not so good...

Caveat with setTimeout() based delays w/ core-js(?) While in `enginedjg` (non Babel/core-js based) the following works, it does not work in `timestui`. This may or it may no be Babel/core-js related but I am suspecting the heavy *eval()* based execution:

```
waits: function( limit, cnt ) {
    alert ('waits (common): ' + limit  + ' ' + cnt)
    console.log('waits(): limit: ', limit, ' cnt: ', cnt)
    if ( cnt <= limit) {
        alert ('cnt (common): ' + cnt)
        window.setTimeout(function(){window.instrustat.waits(limit, ++cnt)},1000)
    }
    alert ('waits (common): return: ' + limit  + ' ' + cnt)
}
```

The below way for delay loops in `.ts` file is tested to work, but has no programmable delay:

```
var poll: () => void
(pol = function() {
....do something to get out from this poll ...
```

```
window.setTimeout(poll, 1000)
})();
```

It is noteworthy that the below works also in `timestui` as .js - there is is no recursive calls. Of course async function returns immediately so not so obvious how to use this since await cannot be used in top-level with old browsers like IE, TypeScript does not allow it! Nerve-wracking...

```
export async function test(limit) {
    alert ('promise')
    try {
        let promise = new Promise((resolve, reject) => {
            setTimeout(() => resolve("done!"), limit * 1000)
        })
        let result = await promise
        alert (result)
    } catch (e) {
        alert(e)
    }
}
```

> Debugging this type of issues with `alert()` only (console back-end being missing on run-time is painful and misleading, since we have asynchrounous events driven from the C++, on the other hand, and database queries events dropping back at any moment, on the other hand. How to deal with this is discussed below, in debugging chapter.

8.3 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.

8.4 IDE

Atom and Emacs have been used but nothing prevents to use Visual Studio Code, no religion!

8.5 Coding style

Well, it may bug you but I am **not using semicolons** but allow that pleasure either to run-time (.js) or to TypeScript transpiler (.ts). Again, no religion here but it is for my mental switch when I change between C++ and JS/TS modules, that's all. Yes, I am careful with some exceptional cases where semicolon is mandatory (by avoiding them!).

> Otherwise, **soft tabs (spaces)** and **tab-width = 4**, please!

That's it. Thank you for your comprehension.

8.6 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.

8.7 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.

8.7.1 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 Windows

`__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.

9.2.1 Sticky cache

Even using different web servers, *node* or *docker nginx* I noticed during development that after transpiling, *bundle.js* file never get updated even after restarting of *OpenCPN*.

Can be annoying if new updates or fixes are installed. Anyway, really annoying for the developer.

This is why you will find a separate *Reload()* polling wait after the page load in *InstruJS* class. Annoying flickering follows but since the files are small, the delays are not excessive. Anyway, I could not find a way to clean the cache in *WebView*.

9.3 WebView specific on Linux WXGTK

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

9.3.1 Linux distros and WXGTKx-blues

Notably wiht the *CMake* this thing can drive your crazy, from distro to distro - what works with the above method of compiling wxWidgets from scratch, that is not available for the installer.

As soon as one adds the dreaded `webview` in the list of libraries to search, problems start occurring:

```
In PluginConfigure.cmake:
            SET(wxWidgets_USE_LIBS base core net xml html adv aui webview)
```

First, CMake does not find the wxWidgets at all, usually. This is because the distro does not have WebView or have it in other GTK-version (3) than other wxWidgets libraires (gtk2).

To make things complicated, it just dies not telling the reason. Once you think you have fixed the issue, it is actually using GTK3 and if you do not notice (you are so happy that it links now), the resulting package installs and but does not work (it does not pass ELF-test of OpenCPN because it cannot allow either wrong wxWidgets version or different GTK-version it is itsel using (GTK2)!

> **Note:** This is particularly true with Continuous Integration, like with Travis - it happily links everything and build is all fine. But it is rejected by OpenCPN!

Altenatives-horror and can one automate it (Ubuntu 18.04 LTS) Let's take the example of Ubuntu 18.04LTS. It is particularly painful.

```
~$ sudo  update-alternatives --display wx-config
wx-config - auto mode
  link best version is /usr/lib/x86_64-linux-gnu/wx/config/gtk2-unicode-3.0
  link currently points to /usr/lib/x86_64-linux-gnu/wx/config/gtk2-unicode-3.0
  link wx-config is /usr/bin/wx-config
/usr/lib/x86_64-linux-gnu/wx/config/base-unicode-3.0 - priority 306
/usr/lib/x86_64-linux-gnu/wx/config/gtk2-unicode-3.0 - priority 308
/usr/lib/x86_64-linux-gnu/wx/config/gtk3-unicode-3.0 - priority 307
```

Gtk2 is selected:

```
~$ sudo wx-config --selected-config
gtk2-unicode-3.0
```

It should be all fine, but it does not work, CMAke does not find wxWidgets! Let's make a try to GTK3:

```
~$ sudo  update-alternatives --config wx-config
There are 3 choices for the alternative wx-config (providing /usr/bin/wx-config).

  Selection    Path                                                Priority   Status
------------------------------------------------------------
* 0            /usr/lib/x86_64-linux-gnu/wx/config/gtk2-unicode-3.0   308       auto mode
  1            /usr/lib/x86_64-linux-gnu/wx/config/base-unicode-3.0   306       manual mode
  2            /usr/lib/x86_64-linux-gnu/wx/config/gtk2-unicode-3.0   308       manual mode
  3            /usr/lib/x86_64-linux-gnu/wx/config/gtk3-unicode-3.0   307       manual mode

Press <enter> to keep the current choice[*], or type selection number: 3
update-alternatives: using /usr/lib/x86_64-linux-gnu/wx/config/gtk3-unicode-3.0 to provide
/usr/bin/wx-config (wx-config) in manual mode
```

Gtk3 is selected:

```
~$ sudo wx-config --selected-config
gtk3-unicode-3.0
```

Now it builds OK

```
build$ cmake -DCMAKE_BUILD_TYPE=Release ..
```

```
...
-- Found wxWidgets: -L/usr/lib/x86_64-linux-gnu;-pthread;;;-lwx_baseu-3.0;-lwx_gtk3u_core-3.0;
-lwx_baseu_net-3.0;-lwx_baseu_xml-3.0;-lwx_gtk3u_html-3.0;-lwx_gtk3u_adv-3.0;
-lwx_gtk3u_aui-3.0;-lwx_gtk3u_webview-3.0;-lwx_gtk3u_gl-3.0 (found version "3.0.4")
```

But that's not good, since OpenCPN is GTK2.

There is a possibility to get the GTK2 version linking working with these CMakefile changes (select option 2 in the update-alternatives above first). In *PluginConfigure.cmake* :

```
-  SET(wxWidgets_USE_LIBS base core net xml html adv)
+  SET(wxWidgets_FIND_COMPONENTS base core net xml html adv)
...
-    FIND_PACKAGE(wxWidgets REQUIRED)
+    FIND_PACKAGE(wxWidgets COMPONENTS ${wxWidgets_FIND_COMPONENTS}
```

Now it works what comes to detection (but does not now build since I took the webview out for this GTK2 test).

In fact, what happens is that there is **no** *libwx_gtk**2**u_webview-3.0* only *libwx_gtk3u_webview-3.0* in Ubuntu 18.04LTS and probably not in Debian 10! I got mine in Raspberry Raspian by compiling myself.

The reason why thiere is no *libwx_gtk**2**u_webview-3.0* is, according to this page because "*webkitgtk2 is no longer being updated and is considered a security risk. wx3.1.3 can utilise its successor, webkit2gtk, but only for GTK+3 builds*".

It is necessary to make a notice about this for those who builds their own wxWidgets for GTK2 like I did for Raspberry Pi: it works and perhaps is not a risk as long as we stay local. However, people may find strange ways to put their files into, and *voilà*, you are exposed to Internet. Better is not to use GTK2, period.

> **Note:** From the above follows that one should automate the above configuration somehow in Continuous Integration (CI) processes - otherwise a binary is produced but it is not compatible with OpenCPN on that platform! (like it happened to me). See this script: https://git.io/JfcNi - maybe it is possible, but would require some effort. Maybe it is better to let CI compile and then build for distribution anyway manually - one cannot really trust the remote binaries without making their QA, anyway.

What about Ubuntu 20.04 LTS - supposedly all GTK3? This being all GTK3 is scoop I learned from OpenCPN v5.1 beta testers: https://github.com/OpenCPN/OpenCPN/issues/1898

OpenCPN v5.0 (stable) on Ubuntu 20.04 LTS - GTK From `ppa:opencpn/opencpn`, let's inspect after:

```
~# apt list --installed | grep wx
...
libwxbase3.0-0v5/focal,now 3.0.4+dfsg-15build1 amd64 [installed,automatic]
libwxgtk3.0-gtk3-0v5/focal,now 3.0.4+dfsg-15build1 amd64 [installed,automatic]
libwxsvg3/focal,now 2:1.5.21+dfsg.1-1build1 amd64 [installed,automatic]
wx3.0-i18n/focal,focal,now 3.0.4+dfsg-15build1 all [installed,automatic]
```

Looks, indeed that it is all GTK3, even with v5.0

```
root@pbox:~# which opencpn
/usr/bin/opencpn
root@pbox:~# readelf -d /usr/bin/opencpn | grep wx
 0x0000000000000001 (NEEDED)             Shared library: [libwxsvg.so.3]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_gl-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_baseu_net-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_baseu_xml-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_html-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_adv-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_aui-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_core-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_baseu-3.0.so.0]
```

Built-in plug-ins:

```
root@pbox:/usr/lib/opencpn# readelf -d libdashboard_pi.so  | grep wx
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_aui-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_core-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_baseu-3.0.so.0]
```

Let's try with OpenCPN v5.1 beta

```
apt remove opencpn --purge
...
root@pbox:~# apt list --installed | grep wx
```

Clean.

```
~# add-apt-repository --remove ppa:opencpn/opencpn
~# apt update
~# add-apt-repository ppa:bdbcat/opencpn
~# apt update
~# apt install opencpn
```

New installation of OpenCPN v5.1beta

```
:~# apt list --installed | grep wx
...
libwxbase3.0-0v5/focal,now 3.0.4+dfsg-15build1 amd64 [installed,automatic]
libwxgtk-webview3.0-gtk3-0v5/focal,now 3.0.4+dfsg-15build1 amd64 [installed,automatic]
libwxgtk3.0-gtk3-0v5/focal,now 3.0.4+dfsg-15build1 amd64 [installed,automatic]
libwxsvg3/focal,now 2:1.5.21+dfsg.1-1build1 amd64 [installed,automatic]
wx3.0-i18n/focal,focal,now 3.0.4+dfsg-15build1 all [installed,automatic]
```

Let's check the installed binary:

```
:~# !99
readelf -d /usr/bin/opencpn | grep wx
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_gl-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_baseu_net-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_baseu_xml-3.0.so.0]
```

```
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_html-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_adv-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_aui-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_core-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_baseu-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_webview-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwxsvg.so.3]
```

This looks promising, there **is even GTK3 WebView**! Let's check the dashboard_pi plug-in:

```
readelf -d libdashboard_pi.so  | grep wx
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_aui-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_gtk3u_core-3.0.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libwx_baseu-3.0.so.0]
```

What about `alternatives` in Ubuntu 12.04LTS?

```
:/etc/alternatives# update-alternatives --display wx-config
update-alternatives: error: no alternatives for wx-config
```

Good, let's try to build, without any modification to CMake-files:

> Fails, both for OpenGL and wxWidgets libraries, therefore the modifications are needed as they were done for OpenCPN in CMake-files, the old template will not work in Ubuntu 20.04LTS.

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 Do not be shy on console.log/error() and alert() behind a debug switch

Secondly, do not think that putting some configuration file activated debug printing and especially alerts is unprofessional or that it somehow makes the code too big - compared to the size of the included *node_modules* libraries they present nothing! Below it is explained how to get `console.log()` shown from a remote system (often headless). However, most of the time one is developing on single machine and there is no time to set multiple systems. Or, one needs to ask the end user to debug something. In these case he code which contains the debug code incorporated, especially the user activated `alert()` pop-ups will bring you a valuable information. Like this:

```
if ( dbglevel > 3 )
    console.log ( 'showData(): sampleFrequency: ', sampleFrequency,
                  'nofSamples: ', nofSamples,
                  'getRetrieveSeconds(): ', getRetrieveSeconds(),
                  'startStamp: ', startStamp,
                  'endStamp: ', endStamp )
```

```
        if ( (dbglevel > 4) && alerts )
            alert ( 'sampleFrequency: ' + sampleFrequency + '\n' +
                    'nofSamples: ' + nofSamples + '\n' +
                    'getRetrieveSeconds(): ' + getRetrieveSeconds() + '\n' +
                    'startStamp: ' + startStamp + '\n' +
                    'endStamp: ' + endStamp )
    }
```

And when a need rise, put `alert()` breakpoints in your code as much as you feel necessary, especially if you have JavaScript and not TypeScript modules, the typos in variable names can be nerve-wracking to detect!

> In this project, we use console as first debugging tool since quasi-realtime (in JavaScript terms) and not breaking the timing behaviour of the event driven finite state machine. `alert()`is convinient though but execution stops. Therefore, please put it as secondary debugging tool, useful for the end user it they encounter problems, with one notch bigger debug level than you consider necessary for the console prints.

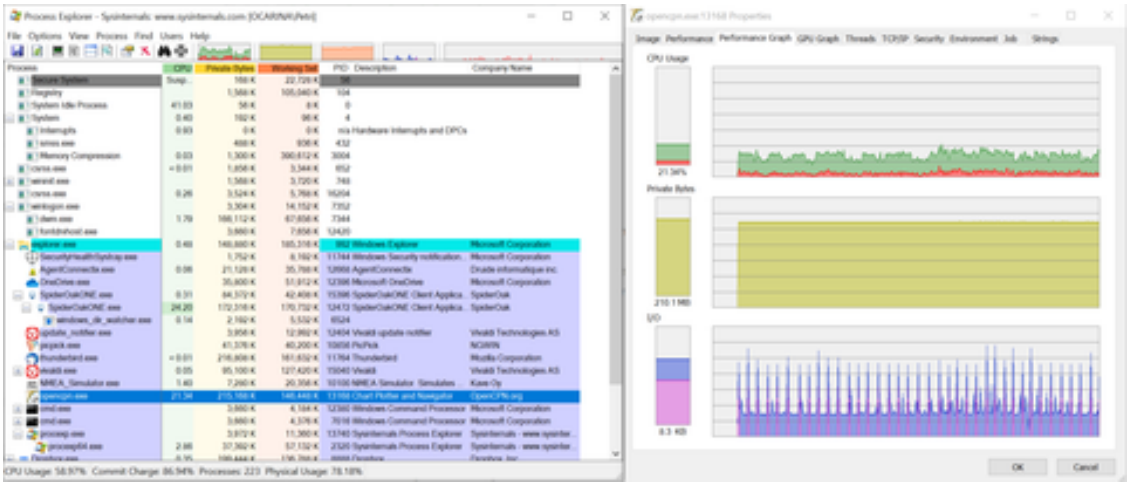
10.2 Performance issues

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!

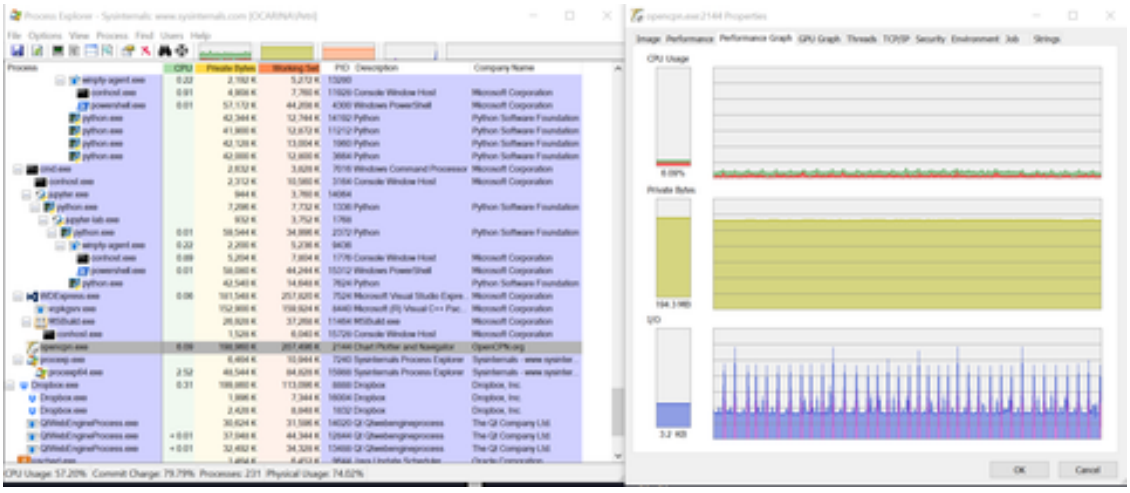
The below process image details is from Windows when there is no filtering of repetitive data with twelve (12!) JS instruments, fed by the NMEA Simulator via Signal K delta channels via Signal K input streamer about 130 floats per second but with static data set (the peaks are InfluxDB Out flusing to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

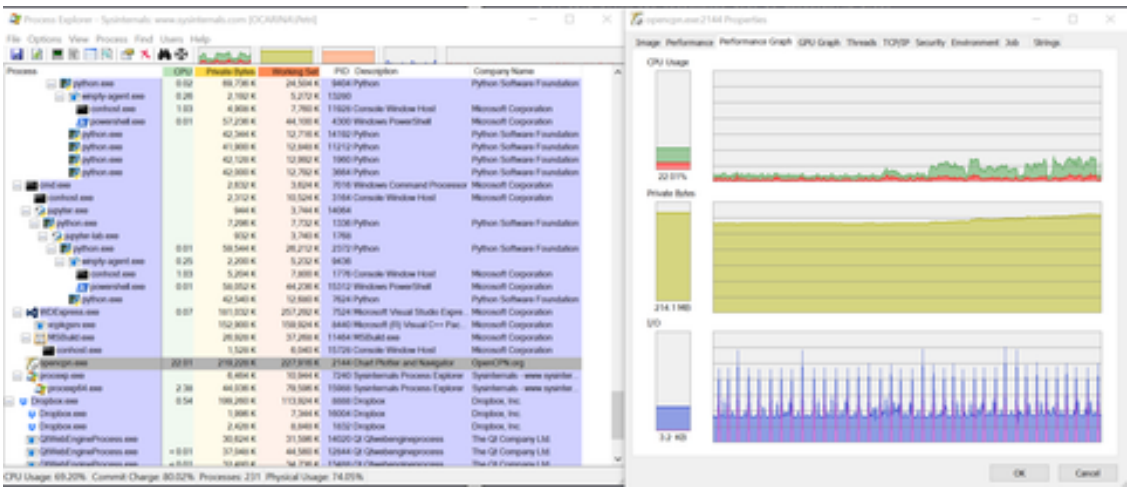
```
if ( m_istate == JSI_SHOWDATA ) {
    if ( m_data != m_lastdataout ) {
        wxString javascript =
            wxString::Format(
                L"%s%s%s",
                "window.iface.newdata(",
                m_data,
                ");");
        RunScript( javascript );
        m_lastdataout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



(zoom)

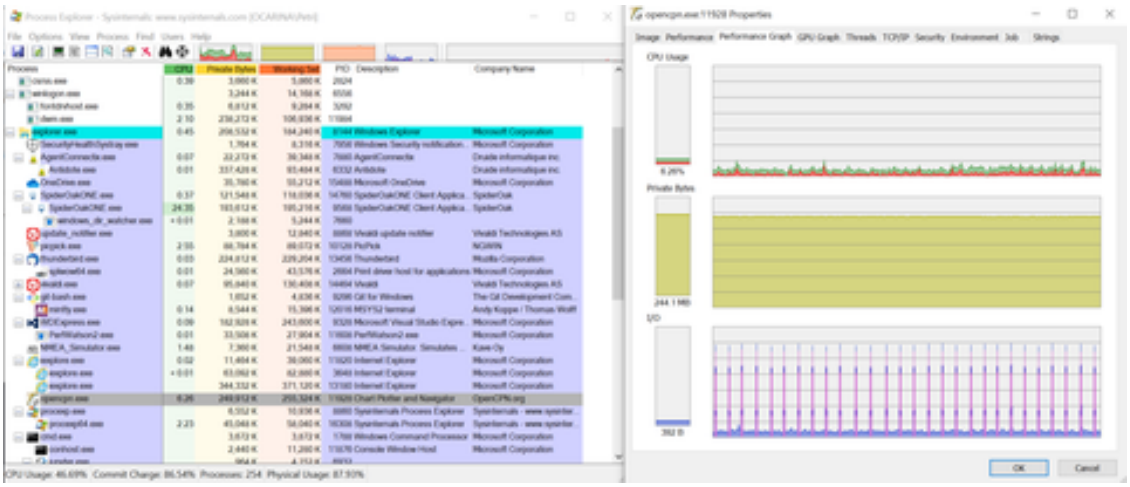
The CPU-load looks quite reasonable around 7 percent w/ i7 CPU and the number of I/O operations is less than half of what it used to be!

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:



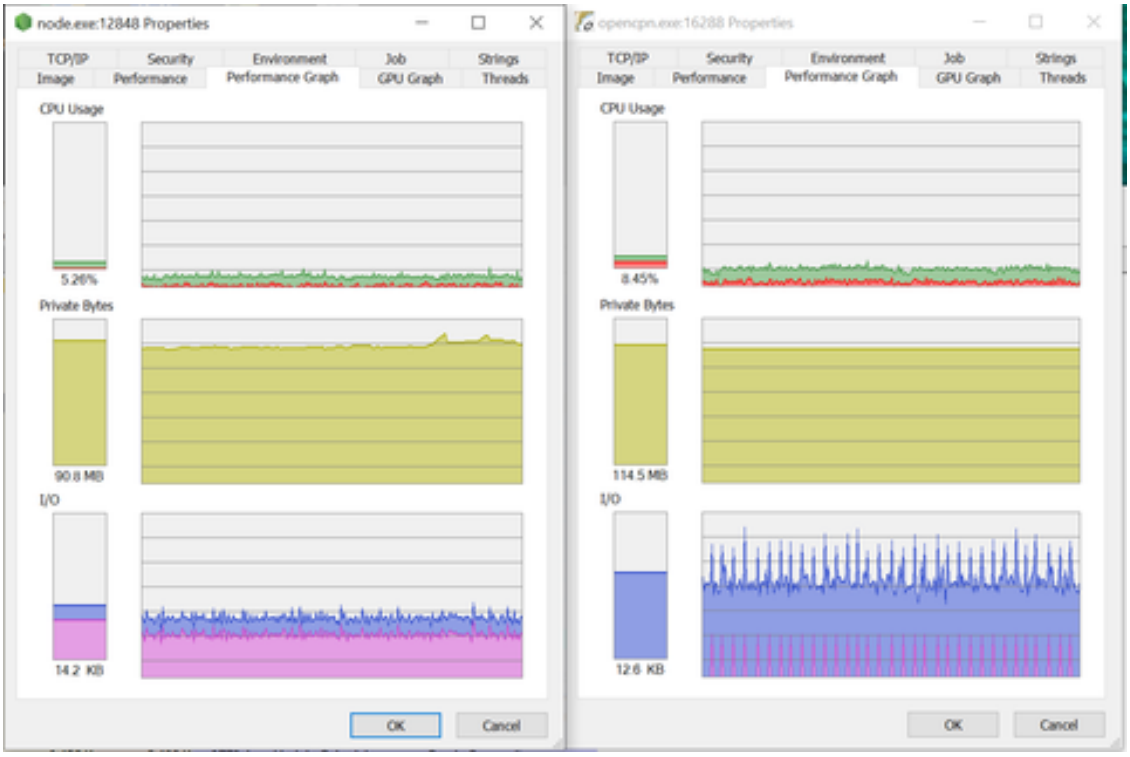
(zoom)

10.3 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.

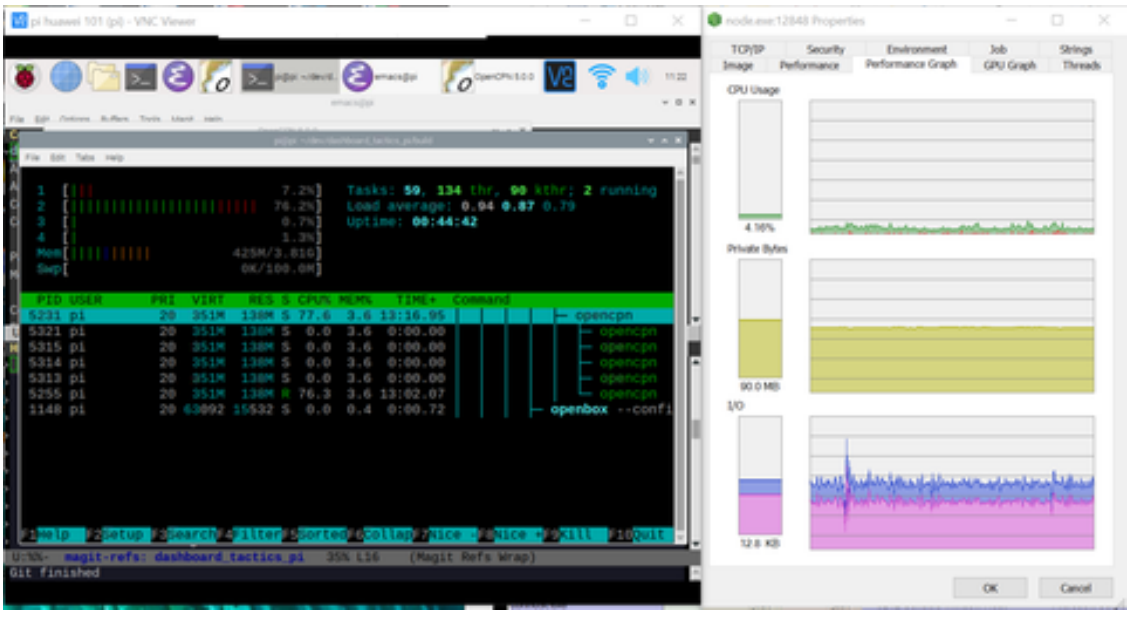
The subscription paradigm probably (but not necessarily) needs to be expanded to that steamer/parser itself. Starting from *SignalK server node v1.19.0* the subscription policy is made mandatory also for the TCP socket data read. This can be used to our advantage regarding the CPU load. The streamer 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.



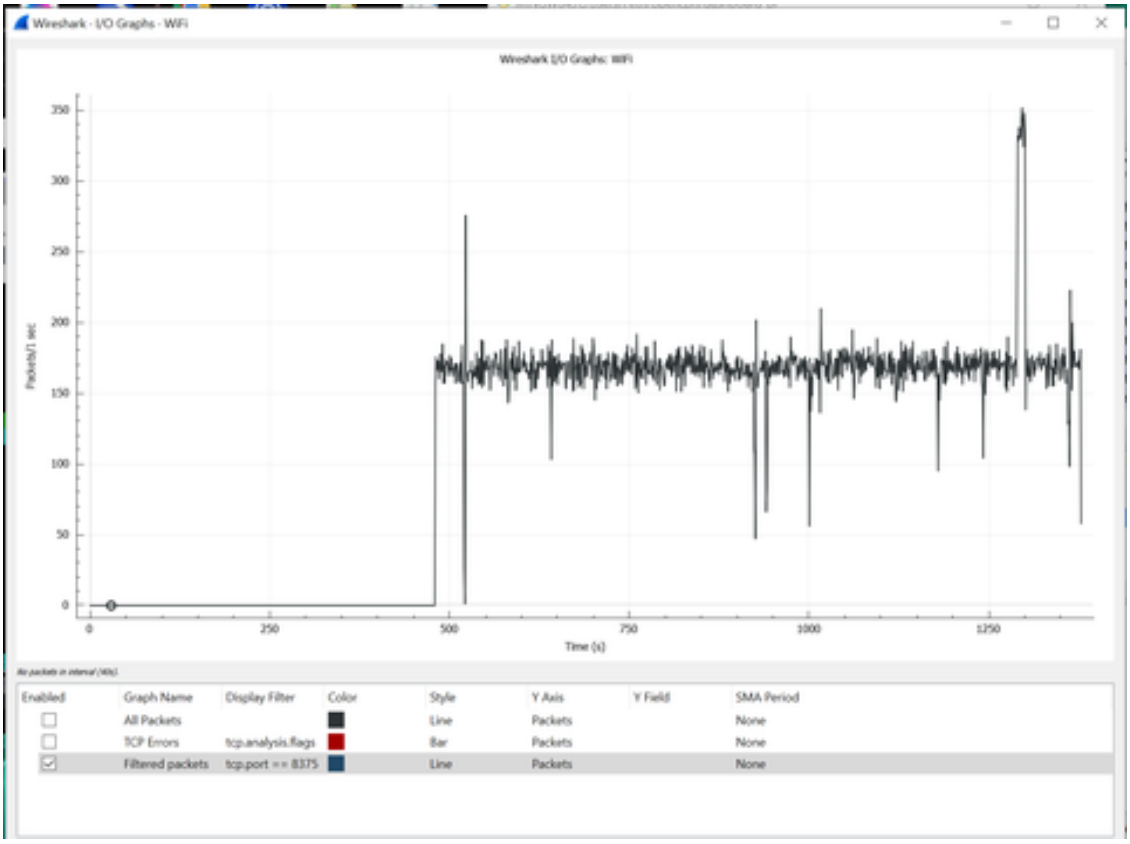
(zoom)

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

The title could continue "“**…it gets frozen on Windows**” - the main issue not being able to debug with a modern browser tools is that you do not have access to the console or its data browser and other developers tools.

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 stubbornly 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.

> **NOTE**: One can do most of the static debugging, still using modern browser's developer's tools: they memorize nicely the repetitive *iface.* structure commands which allows attributing an uid, etc. to the module under the test. It is tedious but very useful since the console messages are your best friend. Only that I have no other way than Weinre to get anything out from a running wxWebView application running the same code but an *alert()*. WEINRE *may* help you to resolve a tricky dynamic problem. Or not.

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-->
```

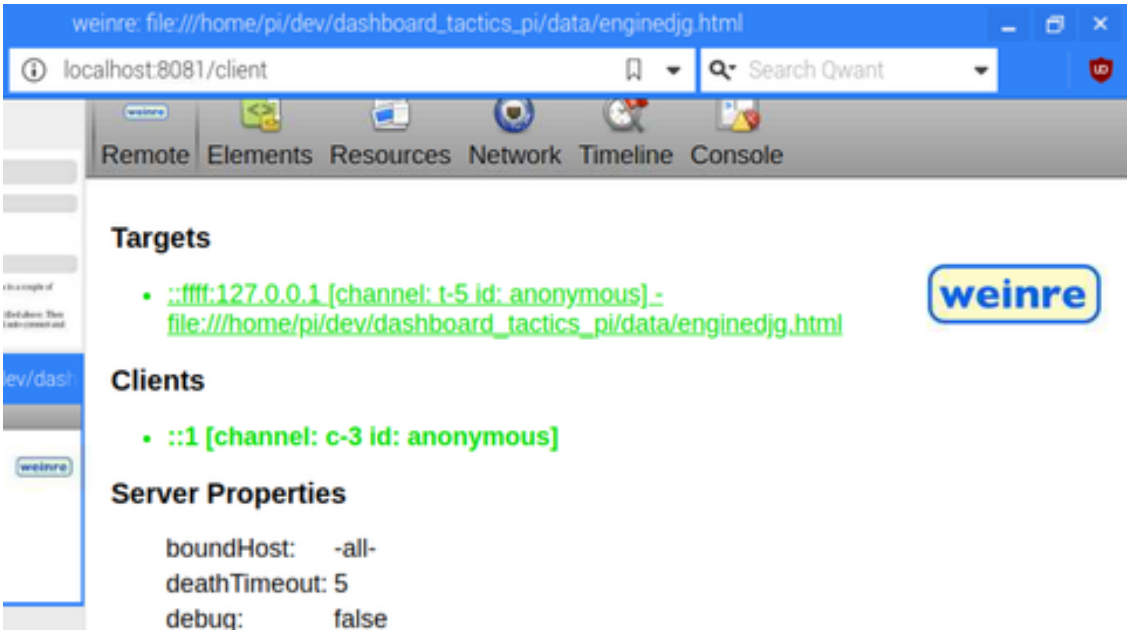
It is not mandatory to use it in exclusively remote, it works also on "*local*" development, you can set:

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

> NOTE: With experience, WEINRE catches up better the console output of a remote system. It is therefore suggested that you run it on a separate system than the system on which you are debugging.

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)

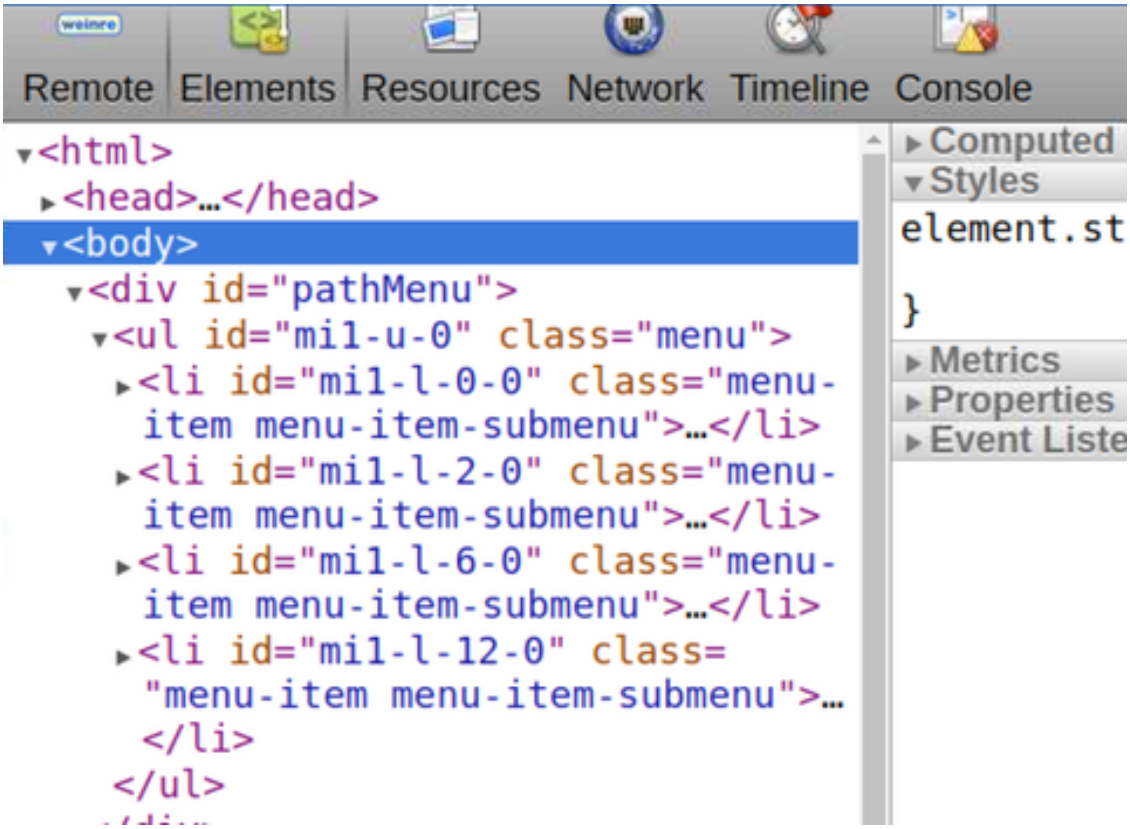
11.1.6 Debugging with WEINRE

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.

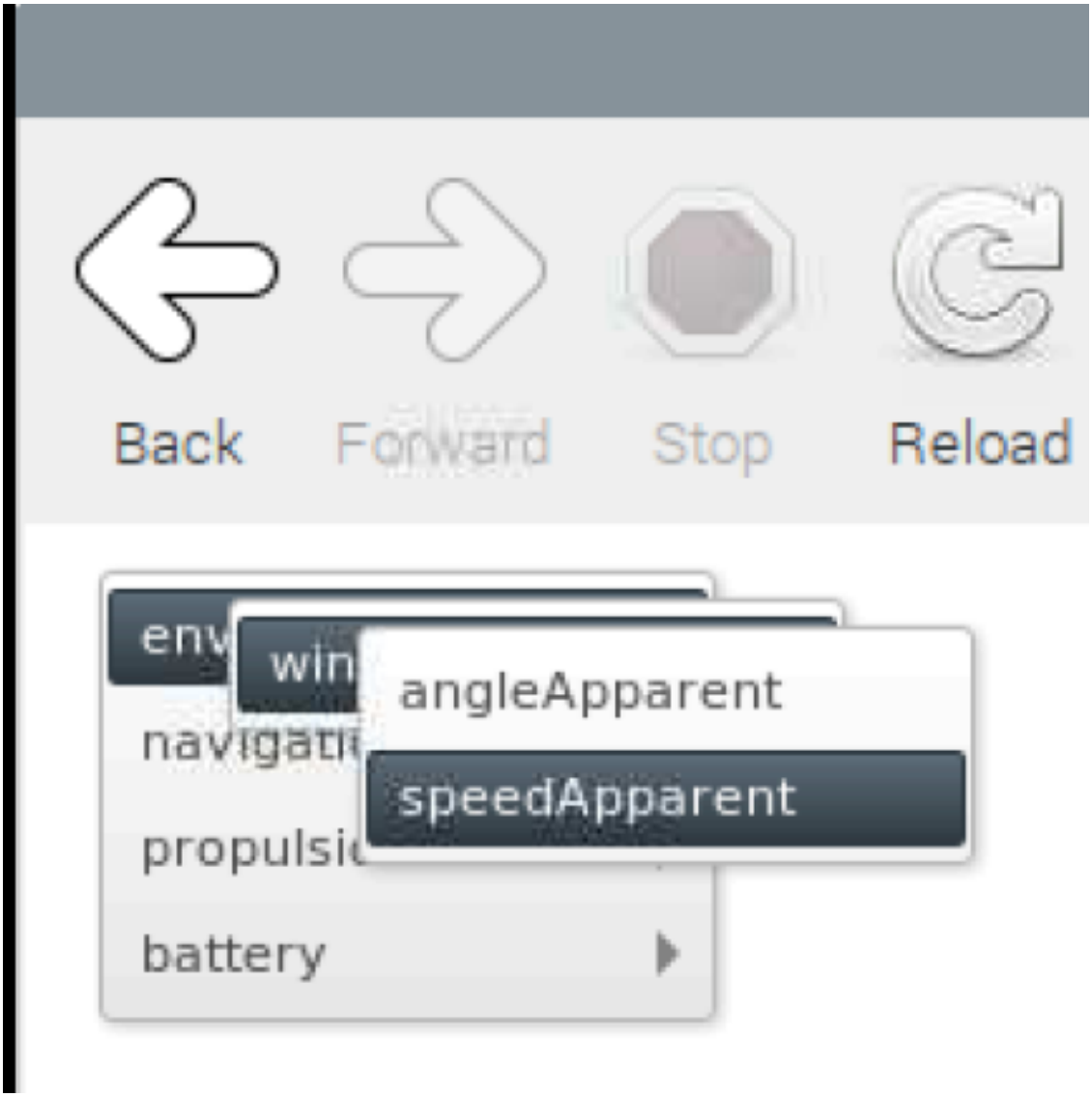
"*Resources*", "*Network*" and "*Timeline*" tabs are quite useless and you should not expect to see anything interesting there.

"*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;
```

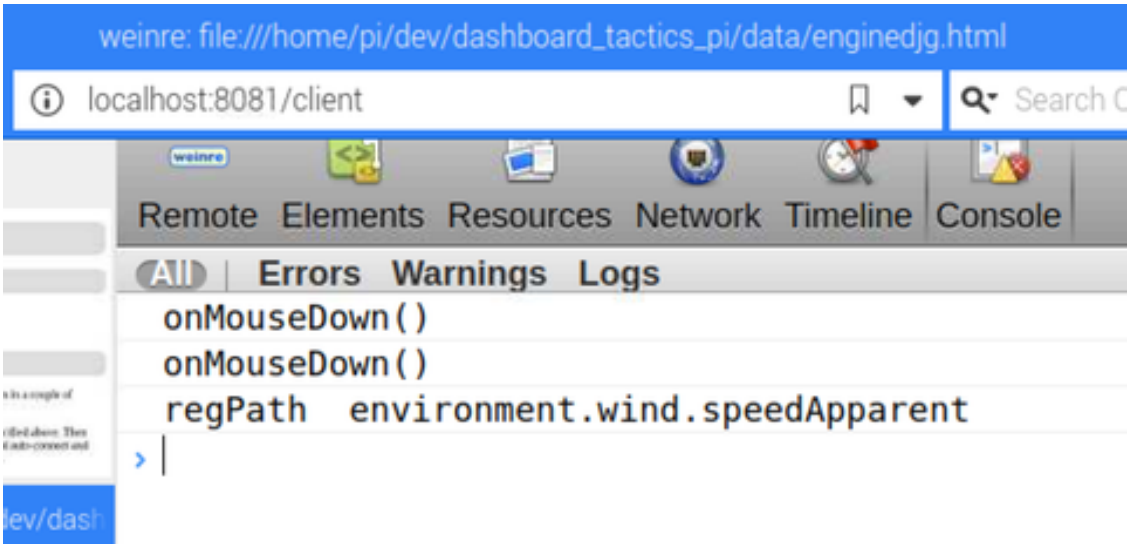
(zoom)

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.



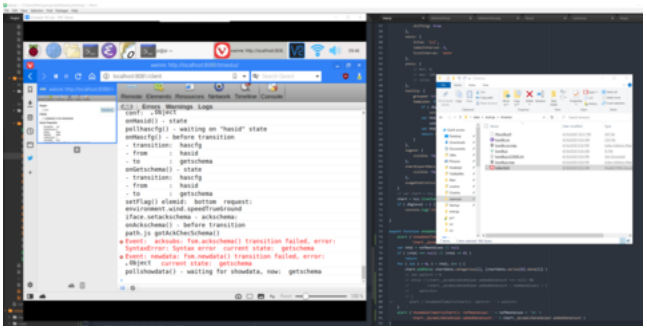
(zoom)

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)

The issues in an event driven finite state machine can only be found using debug printing on console, not with `alert()` because it stops the execution entirely. Here, WEINRE becomes valuable. For example, run `timestui` and its database access on Windows target and WEINRE on a RPI. Set the common debug level to an appropriate level and observe the `console.log()` vs. `consoler.error()` messages and if you have been systematic in you error handling, you are not a blind anymore with WebView anymore:



(zoom)

> Note: do not expect WEINRE handling arrival of many console clients at the same time. It is advisable to restart the service just before you launch your OpenCPN on the remote host for its WebView debugging and using it with only target system at a time:

```
pi@pi:~ $ weinre
2020-04-18T07:43:48.747Z weinre: starting server at http://localhost:8081
2020-04-18T07:44:42.797Z weinre: target t-3: weinre: target t-3 connected to client c-1
^C
pi@pi:~ $ weinre
2020-04-18T08:26:36.403Z weinre: starting server at http://localhost:8081
```

> Otherwise, on the same host, it is enough to reload the WEINRE page to start a new debug sessions on the same host, on which you will start OpenCPN right after.

> Note; remember that if you edit the target's `index.html` file, and then you build again with modifications, your webpack probably builds again the `index.html` file from the template. If the template does not enable the WEINRE connection, you need to man-

ually re-enable it again after the rebuild of your project - of course, you can set for WEINRE debug in the HTML-template but do not forget to take it out before publishing since it will create a high degree of end user confusion!

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,
00,00,00,00,00,00,00,00,00,00,00,32,30,39,37,31,34,37,00,00,00,00,00,00,
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. . .): ± 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.