SignalKInputStreamer

February 15, 2020

1 Signal K input streamer - System Design Description

You have probably already used or got interested in Signal K, the free and open source marine data exchange format. If not, let's hope that that the Signal K input streamer in Dashboard-Tactics plug-in of OpenCPN gets you motivated to start moving your boat's instrumentation from 1990's to 2020's!

You may want to skip the theory and move directly to the *Signal K input streamer usage* document (ipynb | html | pdf).

1.1 Introduction

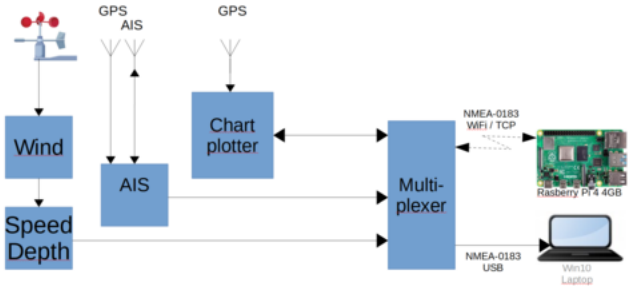
The Signal K input streamer of Dashboard Tactics plug-in has been designed to meet the following three requirements:

1. Read data in NMEA-0183, NMEA-2000 and Signal K data sources
2. Provide data with millisecond resolution timestamps closest to the data source possible
3. Reduce the jitter and latency from the data source to the instruments

Increasing of performance is out of scope - neither the Signal K input streamer nor Dashboard Tactics pretend to miraculously increase the throughput or multiply the update frequency of your boat's instrumentation.

1.2 Example system configuration

The below diagram illustrates an example configuration which does not require to invest on any new hardware if one already is using TCP/IP (network) or USB to get the NMEA-0183 data into the OpenCPN chartplotter.



(zoom)

The Raspberry Pi 4 4G running Raspian is configured as a WiFi hotspot while underway. A cockpit tablet (not illustrated) is used to connect to it using a browser. Raspberry has a local screen and

keyboard/trackball on the chart table.

The Windows 10 laptop is used as backup navigation computer and to collect and visualize meteorological data.

Both systems are having OpenCPN v5 with Dashboard-Tactics plug-in.

Other such configurations exists - Signal K provides more examples; OpenPlotter project provides entire packages containing all the necessary software components for Debian based computers.

1.3 Jitter and latency in OpenCPN / Dashboard data distribution

The third requirement, reduction in jitter and latency in the data distribution is the hardest to meet. In this section we will analyze the data signal path in OpenCPN with plug-ins and, of course in Dashboard plug-in.

Typically, one would need a two channel oscilloscope or accurate timestamps from different phases of the data acquisition chain to determine the inaccuracies in timing caused by signal chain. However, the data in boat systems is sometime coming from systems which are so old or based on old technologies that one can actually see the jitter with the naked eye; values in instruments are jumping back and forth and, very well visible in the dial instrument's needle, the frequency of the jumping is not constant.

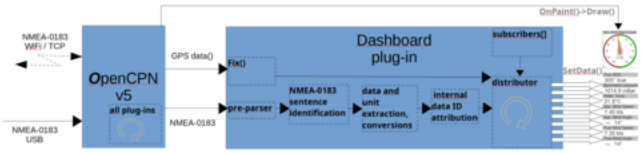
It is to be reminded that it is not normal for any system which attempts to give information about the analog world to give feedback to the user with an indicator which is not following the real-life signal as accurately as possible.

Taking the example of the dial instrument's needle, it shall move so smoothly that the eventual micro-movements are not visible to the human eye. Furthermore, this movement shall be as accurate as possible reflecting in real-time the actual, measured signal. Filters shall be applied only at user's demand so that the observer is always in control. Even if the filters are applied, it should be possible to keep both the original data and the resulting filtered data for off-line analysis - blind faith is not an instrumentation paradigm.

OpenCPN / Dashboard cannot do anything for the data update frequency, it is for the owner to update boat's instrumentation keeping in mind that for any algorithm it is essential to have as much as possible information about the boat's movements, its heading and wind condition it is facing. Logically, one should seek to increase the frequency and reduce the latency.

OpenCPN chartplotter, Dashboard plug-in and the other plug-ins create an extremely complicated system what comes to its dynamic behaviour. It is not a surprise that it introduces jitter and latency in the signal chain by its location between the user and his or her data sources, i.e. the boat's sensors. Of course, the boat's own electronics play an equally important role in this, but since we cannot present an universal model for those, let's study the signal chain from the point onwards where it enters OpenCPN, *i.e.* from USB serial line or WiFi/Ethernet TCP/IP communication in our case.

Below diagram illustrates the signal chain from the point of view of Dashboard plug-in, which is the module we plan to modify.



(zoom)

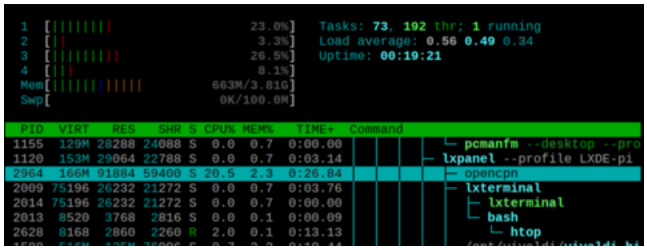
OpenCPN is based on wxWidgets Graphical User Interface (GUI) framework, and like many of them it is event driven: clicking a button is an event; moving a window over another creates events which need to be handled in both windows - an excellent and well established principle for such a software framework. But it is less useful for a data acquisition system unless there is a priority scheme which allows the data event dealt with a high priority interrupt mechanism. It should allow a continuous transport of the arrived data all the way through the system. But if we would follow such as principle in a graphical application, the actual GUI operation could become sluggish and, in general the priority is given to the graphical elements visible to user.

OpenCPN is not a real multi-threaded in application - in the POSIX sense of view. It is threaded using the wxWidgets thread model which is not POSIX compliant. It is by default serving graphical events with what they call detached behavior. Detached threads delete themselves once they have completed which is, of course really useful feature in a GUI since you do not have to make call-backs or otherwise manage the thread.

But it is not necessarily the best adapted way to deal with socket based communication since it can be considered to be overkilling in a multisocket environment to launch a thread - even lightweight - for every arriving frame in a socket, and this for every socket (*i.e.* one thread to deal with any socket "event"). In the current user state multi-socket server paradigm there is rather a POSIX thread per socket. It can be specialized - a protocol handler, for example -, it is seen in the process tables, it can have an adjustable priority, its affinity and can be set to manage the load between the CPUs and their cores.

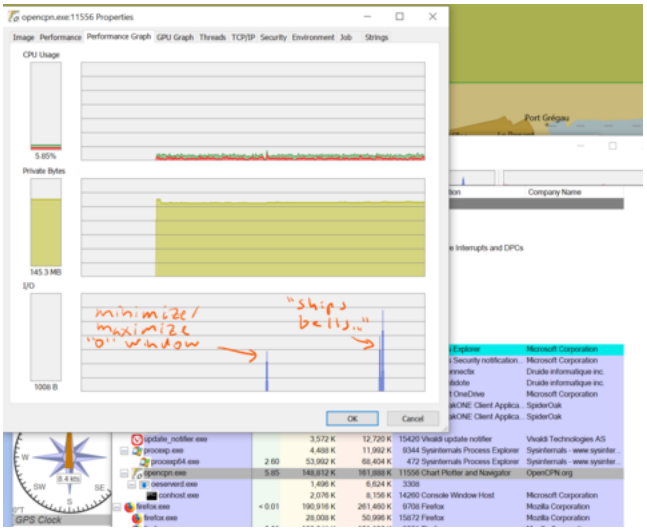
Next we check that this is not the case in OpenCPN already. For the test we use the above configuration with the exception that the Raspberry Pi 4GB running Rasbian 2019-07-10 (Buster) is serving as WiFi hotspot for the boat's installation and the Windows 10 laptop and its OpenCPN is obtaining both its NMEA data and AIS information from it over a wireless TCP/IP connection with the Signal K server, running on the Raspberry device.

The below screenshot illustrates the threads in the Raspberry. There is none for OpenCPN so it must be running entirely in the wxWidgets "detached" thread mode.



On the Windows 10 system, the notion of POSIX threads does not exist but the thread information can be observed with SDK tools. Like above, also in this case we have a standard OpenCPN - Dashboard plug-in datapath for NMEA.

On the Windows 10 system we can see that there is plenty of capacity available, the WiFi based NMEA / AIS information does not create any big load to the system. The I/O load is almost insignifiant compared to the load which occur when one mimimize/maximize the OpenCPN window, or when it chimes the ship's bells!



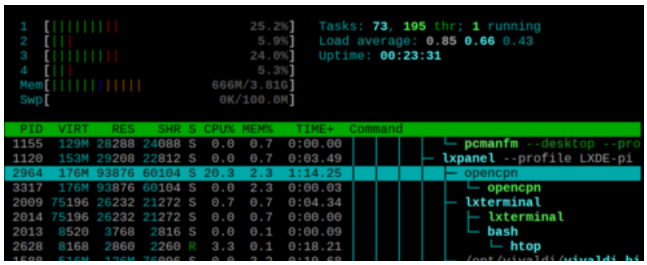
(zoom)

1.3.1 Threaded model of the Influx DB output stream service

We use the Influx DB output stream to store the results in a file with a timestamp, generated locally. Otherwise we would have no way to define the jitter. For the latency we cannot make the measurement since the OpenCPN does not timestamp the data at its arrival. Therefore we will observe the overall throughput only.

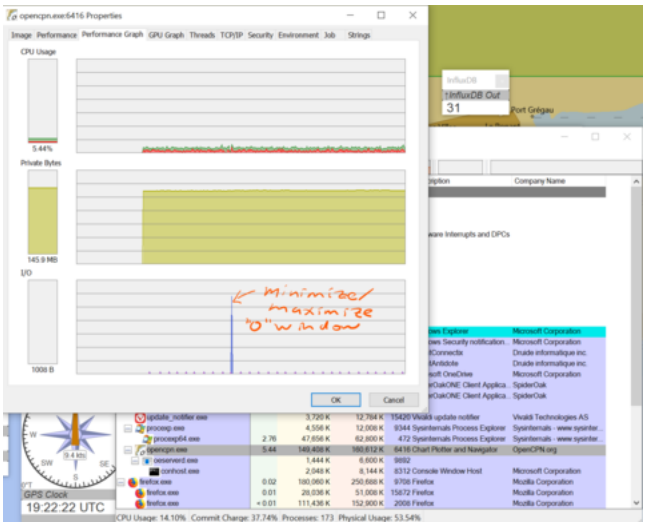
The collected data and the results are reported later in this document. Here we take the opportunity to study the impact of the Dashboard-Tactics' thread model for Influx DB output in view of its usage in the data input socket. Albeit we do not use its direct HTTP-socket streaming capability but only the file writing, the possibility to stream out to a HTTP-socket of the Influx DB server makes the implementation as an attractive candidate to the input socket as well.

The Dashboard-Tactics' Influx DB output stream service thread, which is implemented with wxWidgets thread model *joinable* is clearly visible in the Raspberry device, so wxWidgets has implemented it quite probably with Linux POSIX threads:



It is less visible in the Windows 10 thread model but one can count one more CPU time consuming thread, clearly a wxWidgets DLL:

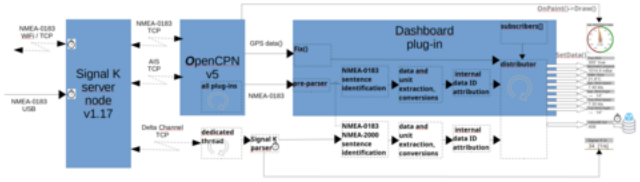
The I/O system load now shows also clearly the flush-mechanism which is built-in into the file writing part of the threaded service. The load of the I/O operations is in the disk writes not in the network sockets.



(zoom)

1.4 System design of the Signal K input streamer

The below diagram illustrates the system architecture with the Signal K input streamer extension.



(zoom)

We could have written our own socket server if the requirement would have been only to have timestamps at the arrival of data. However, when we add the requirement to support not only NMEA-0183 but also NMEA-2000, the choice of a Signal K server node is clear. The ligtweight and robust server based on *Node.js* is fast and stable. In addition it provides additional services which we skip in this study. We use it to feed the OpenCPN application which remains the distributor of NMEA-0183 sentences to plug-ins.

In addition it feeds a *joinable* wxWidgets thread which is available continuously to serve the socket. This is advantageous compared to the OpenCPN distribution model, since we are detached from it and therefore independent from other plug-ins which may have been subscribed to NMEA-0183 data.

The thread contains a built-in parser of the Signal K data: the socket is listening the so-called delta format channel of Signal K which transmits a lightweight (relatively speaking) data, or changes in it. The data is converted to simple expression of C++ data types or wxWidgets string objects so that its handling after this step will be extremely fast.

After the parsing, we pass to a step by step structure, similar to the one that already exists in Dashboard - the only difference is that now the notion of NMEA-0183 specific sentences has disappeared - the data can be either of NMEA-0183 or NMEA-2000 origin, the chain decides based on Signal K schema to which instruments the data should be distributed.

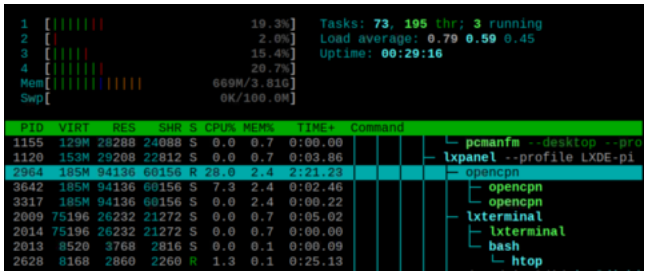
> **Note**: this allows to add relatively easily (coding still needed) new instruments which are dedicated to the engine control. As a proof of concept, three such a engine control instruments have been added into Dashboard-Tactics for engine speed, cooling water temperature and oil pressure.

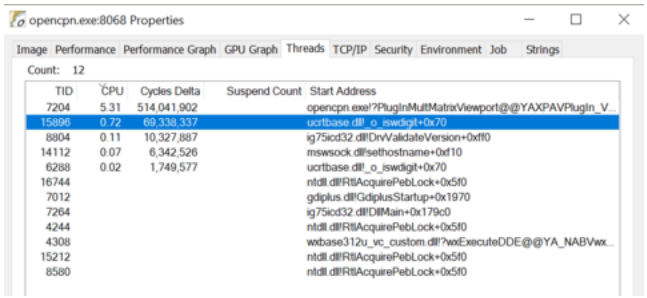
There is a software switch with a timeout: if the data is not available from the Signal K input socket thread, or if there is no equivalent data than in the Signal K delta data for a NMEA-0183 sentence, the software switch closes and let the NMEA-0183 sentence to be passed to the instruments.

One of these instruments receiving the data can be - if user so wish - Influx DB output stream. Now the data has the timestamp of when it arrives into the system, not the time of when it was received by the instrument.

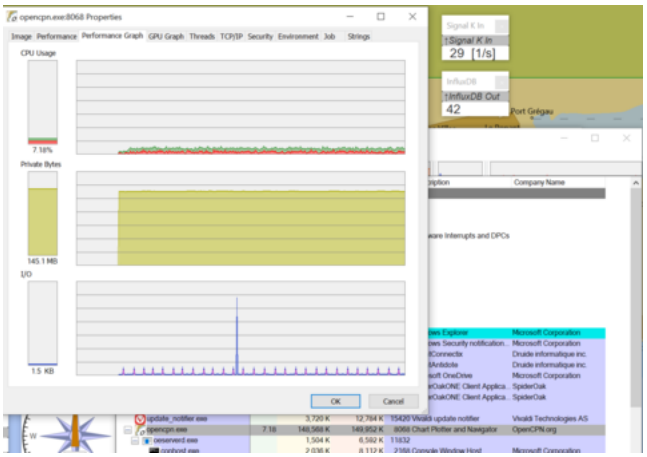
The timestamp is distributed to all instruments which are subscribed to a given data and they can use it for new purposes which did not exist before. For example, all instrument base classes contain now a default behavior what shall happen if no new data is received within a timeout period. This goes beyond Dasboard plug-in's own timeout with data invalidation which is only applicable for some data classes, like GPS data.

Let's observe the new thread both in the Raspberry (Linux) and Windows 10 systems:





We can observe above the increased CPU load. This socket thread is doing something! Let's observe does it translate as increased number of I/O operations:



(zoom)

Yes! Please zoom in and you can see that there is clear increase in the overall I/O operations, about 500 kB delta/s caused by the new channel of data, Signal K delta data.

> **Note**: the payload of the Signal K delta data is by no means containing strictly and only the data which the instruments need: it is the delta of all data Signal K has got and therefore contains data which is certainly not used by any instrument. The Signal K data format in JSON is compact but still human readable and as such it is much longer than the cryptic but compact NMEA-0183 data. However, this has not much impact of today's fast communication channels since we are talking about TCP/IP sockets here and not some RS-232-C 4800 baud line which, for all practical purposes NMEA-0183 is.

1.4.1 Detailed performance analysis of the implementation

A set of example data was collected in real-life condition (in a boat with the above configuration) for three use cases, both in a USB connected Raspberry system and in a USB connected Window 10 system. No WiFi was used but all TCP/IP connections were internal TCP/IP which, in Linux means that the message is not passing through the device driver stack.

The detailed reports can be read using the below links.

Raspberry Pi results (ipynb | html | pdf)

Windows 10 results (ipynb | html | pdf)

1.5 Conclusion

The implementation meets the requirements. The obtained improvements in the stability and in the reduction of the jitter are significant.

There is performance margin which allows the boat owner to take advantage of the modern socket based communication by increasing the sampling rate of his or her sensors and instruments (by replacing them).

The implementation enables the shortest possible data path to instruments and, also importantly to Tactics "regatta computer" or other algorithms, it enables greater data throughput with accurate timestamps.