REPLACING PROPRIETARY HARDWARE WITH OPEN SOURCE ALTERNATIVES ON SAILING VESSELS

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2024

Jocelyn Oliver Grimwood

Supervised by James Garside

Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

This report describes an investigation into replacing expensive proprietary hardware used on sailing vessels, specifically plotters, with open-source alternatives. This paper details the design and implementation of a Raspberry Pi HAT and a Windows application that allows a laptop to replace a plotter. The project aims to provide instruction for any person to install and use the Raspberry Pi on any NMEA2000 network for a fraction of the price of proprietary hardware.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor James Garside as well as Stephen Rhodes who gave me guidance on PCB design.

Chapter 1

Introduction

1.1 Brief introduction on sailing

When sailing it is useful, and sometimes critical, to have the information on what the vessel and its environment is doing or has done. As an example take a scenario where there is reduced visibility and no GPS (Global Positioning System). In this instance navigation is extremely tricky; however, there is hope. With a good chart [appendix A.1] and some vessel data, navigation is possible. If you know the depth below the vessel you know a line on which the vessel must be. If you know your heading and if depth is increasing or decreasing this also gives you a clue as to your location. If you know your speed and rate of change of depth this also gives a clue. Combine these clues and landmarks such as buoys and you can accurately navigate in sub-optimal conditions. Hopefully, this brief example conveys the importance of being able to see and track vessel data.

1.2 My Project

Most personal and industry sailing boats are equipped with instruments running on an NMEA2000(N2K) network [13]. Data critical to navigation is shared on this network. Typical instruments installed on boats include a log, anemometer, GPS, compass and engine management, giving data on boat speed, distance travelled, wind speed and direction, temperature, location, heading, engine revs and fuel levels. Needless to say, this information is invaluable to the normal operation of a vessel as well as navigation.

These instruments are expensive and the plotter (a tablet style display), which is one way to display this data, is extremely expensive. A cheap model can cost at least £500 while a more typical model can cost upward of £1000 (Source: `https://www.force4.co.uk/department/electronics/gps/chartplotters`). There are of course alternatives such as individual displays capable of displaying only one type of data like wind speed or boat depth and speed.

My aim is to provide an inexpensive way to take this data and display it on a laptop using open source plotting software. Essentially my motivation is price.

1.3 History of sailing instruments

Figure 1.1: Historic Log and Lead line
[Source: https://www.rmg.co.uk/collections/objects/rmgc-object-42932]

Marine navigation is based on a globe and distance is a division of a great circle of the earth [11]. One nautical mile is one minute of latitude where one minute is 1/60th of a degree. This is important as longitude lines all make great circles while only the Equatorial latitude line does. Historically speed and consequently

distance traveled was measured by a log. 'Log' is an apt name as it was essentially a log thrown overboard attached to some string with knots in it. By counting the number of knots that are pulled overboard in a given time you get an estimate of your speed in knots or nautical miles per hour. Depth can similarly be found by throwing a lead wait on a rope with length markings over the side and waiting till it hits the bottom.

Figure 1.2: Modern Log and depth sounder

The modern version of these two instruments is a single unit that gauges depth with an echo sounder that reflects sound off the sea floor and measures the time it takes to bounce back. It also has a paddle wheel that spins through the water and is used to tell speed. Modern units also have additional features such as temperature sensors. Historic units such as knots and nautical miles are still used and useful however depth is measured in metres nowadays as opposed to fathoms.

1.4 Requirements and standards

My project has two critical aims: it must be cheap and it must be simple from an end-user perspective.

Price is important and my aim is to provide a cost-effective way to view vessel data with just a laptop and some inexpensive hardware. My aim is to be under £100 excluding the price of the laptop. While I would like my final product to be visually professional, it is important that the end user should be able to install it easily and cheaply even if that means using a Tupperware box and some sealant as a water-resistant case.

Ease of use is also critical. My project should be plug and play; simply connect to an N2K network and use the power from this network for the hardware. The hardware should interface with the laptop by some conventional standards, Bluetooth, USB or over the network would be ideal. It shouldn't take an IT specialist to install nor should powering any devices be overly complicated. Ideally, the hardware should take power from the N2K network.

I'm developing for Microsoft's Windows for a number of reasons. Linux is ruled out due to being a less common operating system, MacOS may be an option in the future, however as I have no Apple machines or experience in MacOS I chose Windows. Microsoft's Windows is also the most widely used OS (Operating System) for personal computers. This application should be accessible so given the choice of one platform I chose to develop for Windows.

1.5 Intended user

My intended user is not someone who buys a brand-new boat, if you can afford to buy a new boat then the cost of this proprietary hardware is negligible. Instead, I aim to provide a solution for people in the second-hand market.

While a preowned boat can still cost huge amounts of money I'm targeting owners in the DIY or refurbish market who are buying boats for personal use and fixing them up. Second-hand sailing and power boats can sell for less than £10,000. While this is expensive many people can use these as homes and sometimes do.

Figure 1.3: Beneteau First 32 £9,500
[Source: https://uk.boats.com/sailing-boats/1982-beneteau-first-32-available-9136987/]

I also don't want to limit my range of intended users to only those with technical skills. Ideally, users should only need basic DIY (do it yourself) skills that already exist from boat maintenance and possibly some simple online tutorials. Essentially my users are boat owners on a budget willing to do a bit of DIY

1.6 Goals and evaluation strategy

My aims for this project are as follows:

- Replace the need for a plotter meeting the requirements:
  - Display data received from a vessel's instruments in real time
  - Integrate any data into chart software
  - Cost nothing in software

- Total cost less than £100 in hardware excluding a laptop
- Installation will be simple and require no technical skills
- Application will have scope for expansion

The first few goals are simple to evaluate. Displaying the data inside chart software for free is absolute, it is achieved or it is not. Cost is also simple to evaluate, simply combine the total cost of all hardware needed by the end user.

The last two goals are harder to quantify. I can perform installations myself in early development in order to test ease of use however, to completely justify achieving this goal real world tests and surveys are required. Given the opportunity I would involve a small, say ten people, test group and have them use my design over a month. Preferably the users would have varied needs in order to get a broad range of feedback. Surveys would allow me to quantify how effective my project has been.

Chapter 2

Background

2.1 NMEA2000 & NMEA0183

2.1.1 History of electronic navigation

One of the first electronic standards for marine electrons was introduced in the 1960s. RS-232 also referred to as EIA-232 [4] defined serial communication standards between devices. This standard supported devices such as GPS, radar, depth sounders and displays. Data was sent serially via two wires TX, the transmission line and RX the reception line. This standard had its limitations and issues. Cable length was limited and it suffered from interference as well as low data rates.

Manufacturers solved many of these issues by introducing their own proprietary standards such as SeaTalk from Raymarine [17] and SimNET [19] by Simrad. While these proprietary technologies did offer improvements over earlier standards they also limited the compatibility of devices and eventually led to the introduction of a standardised system. In the 1980s the National Marine Electronics Association (NMEA) developed NMEA0183. This new system established a common protocol for serial data communication and seamlessly seamless integration of devices from different manufacturers [13]. Additionally, NMEA0183 didn't suffer from many of the issues plaguing EIA-232.

NMEA0183 eventually started to suffer from issues such as bandwidth, network scalability, and compatibility with modern marine electronics. This resulted in the

development of NMEA2000 in the 1990s. Originally based on technology from the automotive world, NMEA2000 took Controller Area Network (CAN) bus technology and integrated power into a backbone for communication. This fixed many of the scalability and offered support for many newer technologies as well as room to add new ones.

2.1.2 Introduction to modern standards

NMEA2000 [13] and its predecessor NMEA0183 [12] [appendix: A.1] are a standard for data transfer and communication between vessel instruments over a bus network. NMEA0183 is the old standard for data transfer and NMEA2000 was a replacement. NMEA0183 devices can communicate on the newer N2K network but require a converter to do so.

NMEA0183 has a hardware standard but for this project that is unimportant. The important aspect is the sentence structure of NMEA0183 messages. Sentences always start with an '$' followed by two characters for the sender and three for the message type. This is then followed by all information in the message, for example:
$\$GPAAM,A,A,0.10,N,WPTNME*32$ **[2]**
this is then terminated by a checksum [appendix: A.1], a carriage return then a line feed. This is important as some chart software takes NMEA0183 messages as input.

NMEA2000 has a much broader range of possible messages including proprietary messages all in the form of PGN packets (Section: 2.1.4). These messages are passed over a bus network consisting of five wires: two data wires, two power wires and a shield wire all encased in a wire mesh and rubber. Power is 12V and can be used by all devices on the network providing current doesn't exceed 8A or 3A for Mini or Micro cable sizes respectively. The shield helps reduce interference in combination with the outer sheath. Finally, the data cables are connected on the backbone (bus) by two terminating 120 Ohm resistors on each end of the backbone for a 60 Ohm resistance between the high wire (CAN H) and the low wire (CAN L) (Source: 2.2).

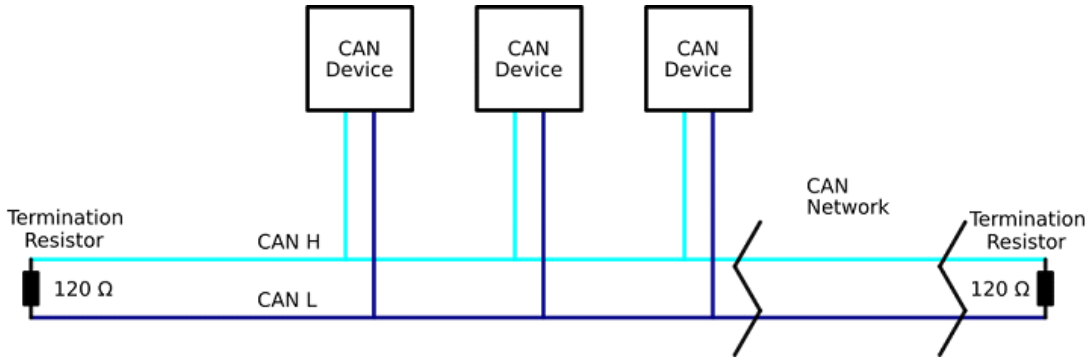

Figure 2.1: CAN Network
[Source: https://learn.sparkfun.com/tutorials/ast-can485-hookup-guide/introduction-to-can-bus#: :text=The%20message%20ID%20is%20also,0B).]

2.1.3 Instruments

Typical instruments onboard include a log, a depth sounder, an anemometer, a compass and engine management sensors. A log gives vessel speed and can often be combined with other sensors such as temperature and a depth sounder to give depth below the vessel. An anemometer gives wind data such as relative angle speed and possibly temperature and pressure. For this project, I have access to a combined Log, temperature sensor and depth sounder. This instrument sends 4 pieces of information over the N2K network, speed through the water in knots, depth below the sensor with a given offset, distance through the water in nautical miles and water temperature. This sensor should provide a good piece of test equipment and most testing will be done with this log.

While I log is one of the most common instruments on a boat, other instruments would be useful in testing and validation. Ideally, I would have an anemometer and a GPS receiver unfortunately budget constraints prevent me from acquiring these instruments.

2.1.4 PGN packets

PGN or Parameter Group Numbers are 18-bit identifiers package inside the full 29-bit can header 2.2. These identifiers describe what data is being sent and how it is structured so it can be read and used accordingly. What this allows is a huge variety of messages with varied structures suited to the message type. While

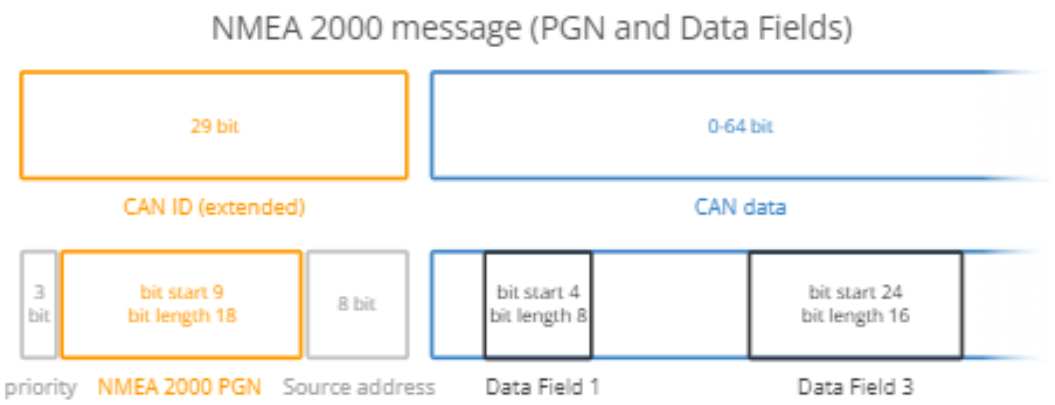

Figure 2.2: NMEA2000 CAN packet
[Source:
https://www.csselectronics.com/pages/nmea-2000-n2k-intro-tutorial#nmea-pgn]

this is useful for manufacturers to add new instruments with new data to the existing PGN database, it makes it extremely hard to decode all possible messages especially when some are proprietary and their datasheet costs money. Overall there are more than 170+ PGNs unique to NMEA2000 with 1500+ parameters.

The problem presented by PGN is the number of possible identifiers and cost of accessing proprietary PGNs however, there are open source communities that work on decoding proprietary PGNs and distributing decoding software. These communities can bypass the cost legally because they don't pirate the PGN data sheets, instead, they reverse engineer the relevant information by reading packets and extracting the location of relevant data.

2.2 CAN Bus

2.2.1 Introduction

CAN or Controller Area Network is a common standard to share data over a serial bus network [6]. Typically used in cars and commercial machinery CAN uses a two wire bus network. CAN is designed for reduced cost over running wires for each device on the network and interference resistance. The bus style network reduces cost by allowing multiple devices to share one set of wires rather

than each device having to have data wires to a switch or controller. The two wire design also reduces interference.

2.2.2 Hardware Standard

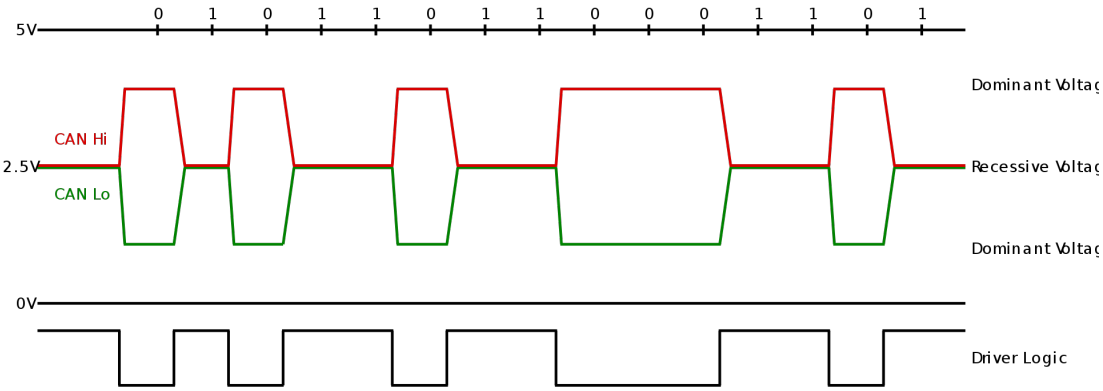

Figure 2.3: CAN signalling
[Source: https://en.wikipedia.org/wiki/CAN_bus#/media/File:ISO11898-2.svg]

The two CAN data wires, named CAN-H and CAN-L, are held at a nominal voltage and then driven high and low respectively using a method called differential signalling. When CAN-H is driven high, CAN-L is driven towards 0 resulting in the potential difference across these lines converging and diverging. When CAN-H and CAN-L diverge this is a 0 and when at the same potential difference or zero difference this is a 1. This design is resilient to interference as both lines will be affected by interference equally and, as such, the difference in pd between lines will be unaffected this is called common mode rejection. [6]

2.2.3 Packet Standard

The packet standard is fairly basic consisting of an ID, data length, the data, a CRC (Cyclic Redundancy Check) and control bits [6]. Data in our case is instrument readings or instructions for instruments in a format specified by the PGN 2.1.4 held in the ID. The ID has to be unique to a device but a device can send multiple messages with different IDs. As the PGN is only part of an ID multiple devices can use the PGN. The priority and PGN also determine who 'wins' in arbitration when two devices try to send simultaneously. Arbitration results in lower

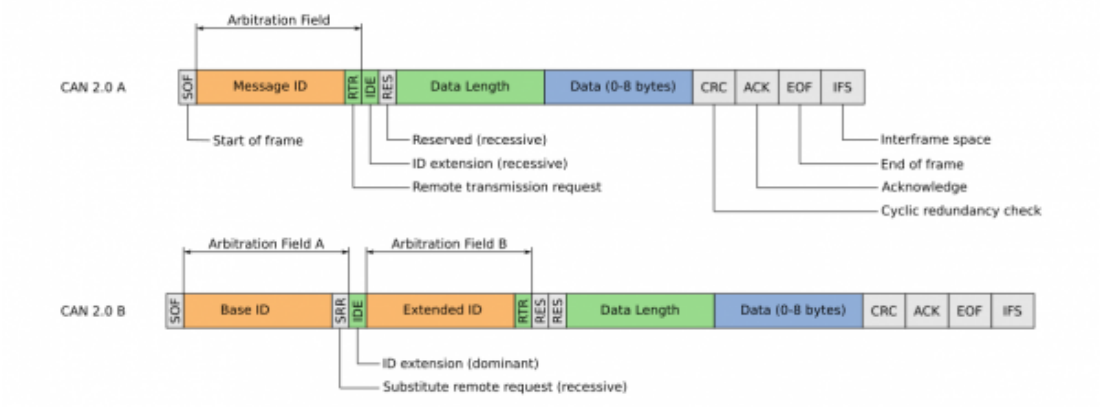

Figure 2.4: CAN Packet
[Source: https://learn.sparkfun.com/tutorials/ast-can485-hookup-guide/introduction-to-can-bus#: :text=The%20message%20ID%20is%20also,0B).]

IDs being sent first or 'winning'. This works by sending devices listening and if a recessive bit (1) is driven dominant (0) by another device it stops transmitting.

2.2.4 Hardware controllers

For interfaces there are two common and relevant microchips: MCP2515 (reference:[8]) and MCP2551 (reference:[9]). The first MCP2515 is described as a "Stand-Alone CAN Controller with SPI Interface". The important features are SPI which is a common serial interface for micro-controllers and peripherals [appendix: A.1] and CAN controller meaning it sends and receives messages using a CAN Transceiver. MCP2551 is a CAN transceiver and acts as the interface between the controller and the physical bus.

2.3 Open source marine projects

The marine world has a number of open source projects aiming to deliver functionality normally locked behind high price tags. To name a few large projects OpenCPN [14], OpenPlotter [15] and canboat [1].

Canboat, developed by the Signal K community, is an open-source project that aims to provide software tools for working with boat data. One of the key components of Canboat is the library, which provides a set of command line utilities

for parsing and generating messages according to NMEA2000 standards. This library allows developers to interface with a wide range of marine electronics, including GPS receivers, depth sounders, autopilots, and engine monitors, and extract relevant data for analysis or visualization in a JSON format supported by other open source libraries.

OpenCPN is an open-source chartplotter and navigation software designed for use on boats and ships. Developed by a global community of volunteers, OpenCPN provides features such as GPS navigation, chart plotting and route planning. The project aims to provide mariners with free access to high-quality navigation tools.

OpenPlotter is a Raspberry Pi-based platform for marine electronics, navigation, and automation. Built on open-source software, OpenPlotter offers a range of functionalities, including NMEA 0183/2000 data handling, AIS reception, GPS positioning, and sensor integration. The platform enables boaters to create custom navigation systems using affordable and readily available hardware components. It essentially allows users to make their own plotters. OpenPlotter has a few disadvantages, it is limited by the capabilities of the Pi. It also has a steep learning curve for both installation and usage and is limited by the hardware installed on the Pi.

Chapter 3

Design

3.1 Hardware

3.1.1 Introduction

My initial approach to design was to use a Raspberry Pi or Arduino due to their price, versatility and strong community support in development and debugging. Neither supports CAN directly but can with additional hardware using an SPI extension bus [appendix: A.1]. To support CAN we need two chips (figure: 3.1), MCP2551 and MCP2515 (section: 2.2.4), and some additional hardware for which there are a number of schematics and tutorials online.

3.1.2 Raspberry Pi vs Arduino

I chose a Raspberry Pi for a few reasons. Firstly a Raspberry Pi is essentially a small computer or microprocessor complete with a Linux OS, unlike an Arduino which is a microcontroller. Because of this, a Raspberry Pi is a much more versatile choice for this project. This allows more space for experimentation and design choices. As this was my first time on a project of this type the Raspberry Pi was the safer choice.

3.1.3 Online Tutorial and Issues

I found a number of schematics online for a CAN interface using SPI. Eventually, I found one specific to a Raspberry Pi as seen in figure 3.1. Unfortunately, there were some errors discovered during implementation that needed fixing for this

design to function. Initially, this design could be 'breadboarded' but the final design would be a PCB.

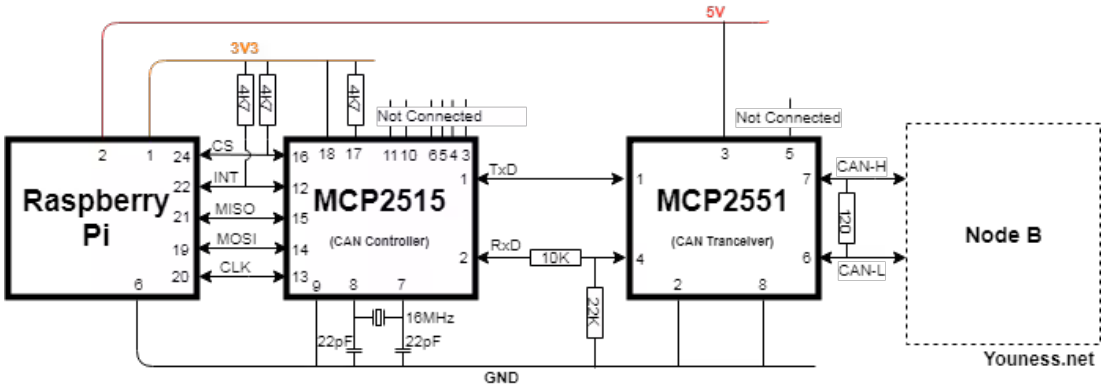

Figure 3.1: CAN Schematic
[Source: https://www.hackster.io/youness/how-to-connect-Raspberry-pi-to-can-bus-b60235#toc-2–software-1]

'breadboarding was crucial to the design process. Breadboards allow quick and cheap design, although not robust or compact design. I was able to test designs for the PCB without having to spend money or time on PCB printing. Being able to test like this allowed me to find a lot of the issues and test if I had fixed them.

3.1.4 Fixes

As this design was non-functional some research was needed. Firstly the CLK line is connected to pin 20 on the Pi which is a ground, it should be connected to pin 23 (SPI0 SCLK). Next, the 10k and 22k resistors acting as a voltage divider have been drawn wrongly. MCP2551 outputs at 5V (reference: [9]) and (reference: MCP2515) [8] has a 3.3V supply (max 'high') so the voltage needs to be dropped; to do this the 22k resistor needs to be placed on the MCP2515 side.

While not strictly an issue, as the design will work, it is generally advisable to have capacitors on power lines to help with interference and filter out unwanted fluctuations and spikes. This is an important inclusion in my design as it should be robust.

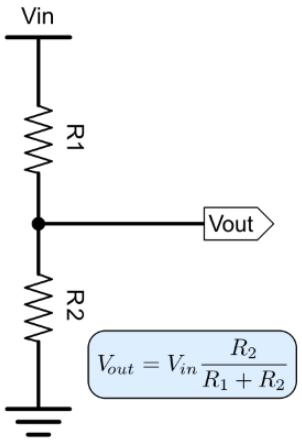

Figure 3.2: Voltage Divider
[Source:
https://study.com/cimages/multimages/16/voltagedivider_resize7989134006010354891.png]

3.2 Software

In the diagram (figure: 3.3) the whole system is shown. CAN HAT is the hardware designed to fit on top of the Raspberry Pi GPIO pins and this interfaces with the pi using SPI. Canboat uses the data from the HAT to send useful JSON data via a network switch to my application that converts it to a useful format and sends it to OpenCPN via a virtual COM port. I have already covered the design of the HAT and will cover the software section, canboat and NMEA data parsing, in this section.

3.2.1 Specification

This project can be broken down into two key aspects, the laptop and the Raspberry Pi, that need to communicate with each other. The Raspberry Pi needs to be able to read and translate NMEA2000 data from the CAN network and output useful data in a common format. The Pi should handle this for the purpose of future-proofing. When developing for multiple platforms much of the burden should be taken by the Pi to reduce the repetition of effort for different platforms. The computer side should be able to receive data from the Pi and parse relevant information to the plotting/displaying software. Additionally, installation should be simple and shouldn't require any special programming skills.

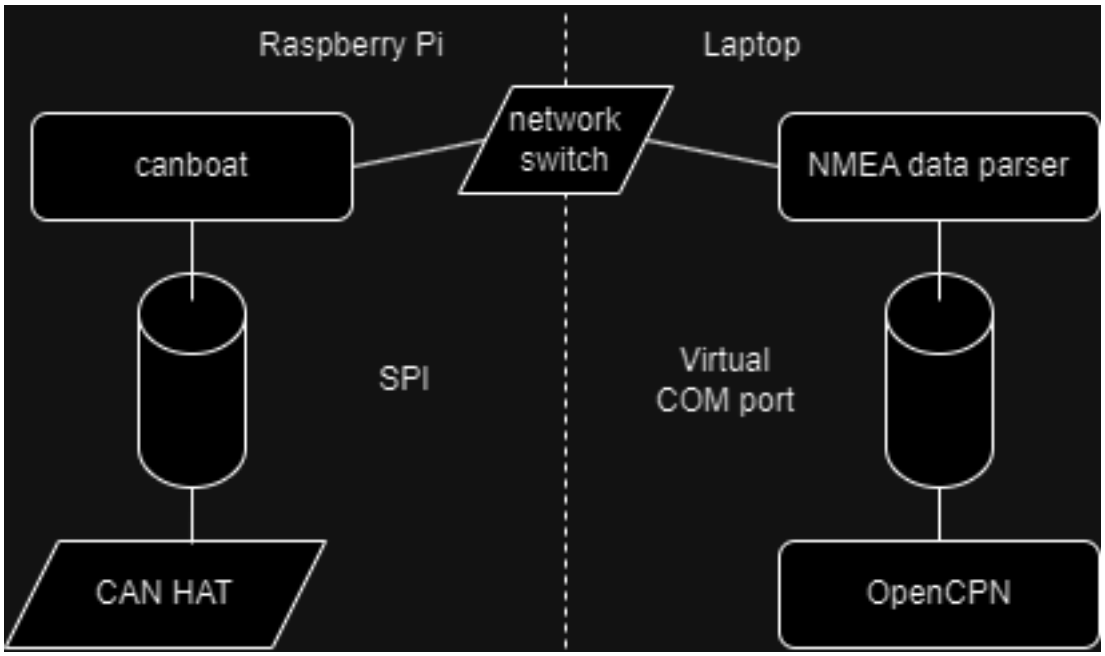

Figure 3.3: System schematic

3.2.2 Raspberry Pi

The main challenge on the Raspberry Pi side is the number of NMEA2000 PGN codes, all of which the program needs to be able to translate. The most viable solution as previously mentioned (Section: 2.1.4) is to use open-source software from communities that have reverse-engineered the packet structure. Obviously, this is possible to do myself, however is infeasible with the time and budget I have. I can't afford to buy all the equipment I would need to test and don't have the time to test it.

The next problem is what software should I use. There are few options and the majority of them use canboat [1]. Canboat is a series of Linux command line tools when combined take the standard out from candump, a command from a Linux CAN library [5], and translate the data into a JSON format then share data via TCP. The tools parse data using standard in and out, so the TCP tool can be replaced to share data via another standard if needed or as a future route of expansion.

TCP or Transmission Control is Protocol is part of the IP suite of protocols. TCP is important as this limits the project to needing a router or network switch to operate unless the TCP tool is replaced with a custom solution. However simple network switches are inexpensive and require little power; while this isn't ideal it doesn't present any major issues.

Finally, when designing for simple installation, creating an image of a Pi with all installation and setup already completed allows any user to take the image and install it on their Pi. Additionally, the Pi should work from boot and need no additional peripherals such as a screen or keyboard. This is all handled by software provided by the Raspberry Pi company.

3.2.3 Computer

As mentioned, I'm initially designing for Windows and I want installation to be as simple as possible. As such my final product should have an installer or be an MSIX package [10]. MSIX is just a Windows format for packaging applications with Windows support for installation. MSIX has a few advantages over a conventional installer mainly because it has better integration with Windows. Installation and uninstalling are automatically handled by Windows; additionally, only two files are needed for the end user, a certificate and the actual MSIX package. The package can also be deployed to the Windows store. For these reasons, it is my aim to create an MSIX application.

While MSIX can support Python its intended use is for .net C#, C++, web apps in JavaScript and some Windows-specific languages. Of those languages I only have experience in C++, additionally, C++ has Windows support. C++ supports classes so I will have an Object-Orientated Paradigm (OOP). I can split my application into clear objects consisting of a TCP client, an object to translate JSON to a format usable by the chart software, an object to parse data to the chart software and an object to handle Windows-related tasks.

The TCP client should handle connecting to and receiving data from the Pi and then parsing it to the translator. In Python, it's as simple as 4 lines of code however, in C++ this requires using the Windows API. Windows helpfully provides

an example TCP client implementation in C++ that doesn't work due to a depreciated function. The easiest fix for this is compiling for an older version of Windows.

TCP client

Once the TCP client receives the data in a JSON format the next step is to convert this into an appropriate format. As my choice of charting software is OpenCPN the data needs to be converted into an NMEA0183 (Section: 2.1.2) format. I check the canboat code to find all possible JSON messages that can be received but I'm not able to test them. For this reason, I want to make it as easy as possible to add rules to translate a message and ideally for a non-technical user to add their own rules.

Translation

The JSON data has a description attached which can be used to identify what parameters are held in the 'fields' section. So this description can be used to identify what rule to use. The issue is how to describe a rule and how to allow a non-technical user to create and edit rules.

I started with choosing an easy to use format for the rules. A text file is easy to edit and doesn't require an IDE. My design for the rule format is to have one per line and identify each rule by its description. Each rule describes a method to translate JSON to an NMEA0183 string, as such a simple rule would be to include the corresponding string with blank data sections and an identifier of the corresponding JSON key (listing: 2).

All a user should need to do is copy the description from a JSON message, copy the NMEA0183 sentence that has the correct data and simply replace the data section with the corresponding JSON key. For example when receiving temperature data the description is "Environmental Parameters" the NMEA0183 message is "$–MTW,0.0,C" where 0.0 is the temperature value and the JSON key is "Temperature" so the rule would be "Environmental Parameters$–MTW,Temperature,C". See listing (listing: 2) for examples.

```
{
    "timestamp":"2024-04-09-13:24:32.195",
    "prio":5,
    "src":35
    "dst":255,
    "pgn":130311,
    "description":"Environmental Parameters",
    "fields":
    {
        "SID":47,
        "Temperature Source":
        {
            "value":0,
            "name":"Sea Temperature"
        },
        "Temperature":7.10
    }
}
```

Listing 1: JSON example

```
Distance Log$--VLW,{Log},N,{Trip Log},N
Environmental Parameters$--MTW,{Temperature},C
Speed$VWVHW,,T,,M,{Speed Water Referenced},N,,K
Water Depth$SDDBT,,f,{Depth},M,,F
```

Listing 2: Translation rules example (repeated)

As for creating a program to use these rules, it should be a simple case of text manipulation. The program would need to extract the description from the JSON, then search for it in the text file then extract the sentence and replace all parameters with the values corresponding to the keys in the sentence. This is fairly simple to implement in C++.

Interface

The program needs to interface with OpenCPN by using COM ports. COM ports are serial communications ports meaning I will need to create virtual ports to

properly interface. OpenCPN accepts NMEA0183 from COM Ports so the program needs to create two paired COM ports one as an input to OpenCPN and one as the output for the program. Creating virtual COM ports is challenging because of the need to emulate hardware while maintaining compatibility. So I use com0com [3], a free open-source software for Windows, to create the ports for me and simplify the task. Using the ports is as simple as connecting and writing to the ports like a file.

Windows functions

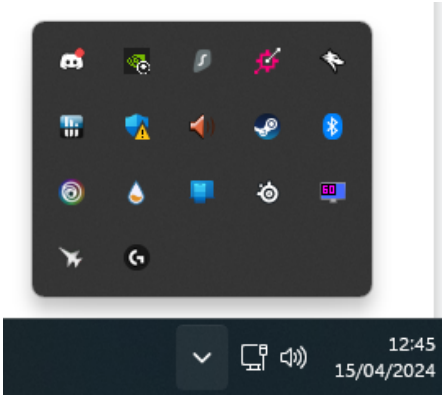

Figure 3.4: Windows system tray

There are two functions the application needs to perform in Windows. The first is proper 'install' and 'uninstall' including adding the application app directory. The second involves proper running including an icon in the system tray and the ability to exit the application. As the application is designed to constantly spin it will need to run as a separate process or thread. So maintaining contact with Windows can run simultaneously with translation. As an example should you wish to exit and the application is waiting on the TCP server then the application would hang until a packet is received. A far better design is for the thread or process to be halted and then cleaned up.

MSIX

The end goal of this application is to be an MSIX application and this warrants some design considerations. Firstly a logo will be needed; a simple image with text will suffice. As for code design, MSIX is only concerned with binaries post-compilation. The only major consideration is that file paths need to be robust to

location changes as Windows handles installation and file location. As for how to implement, Windows provides templates and detailed documentation on the build process [10].

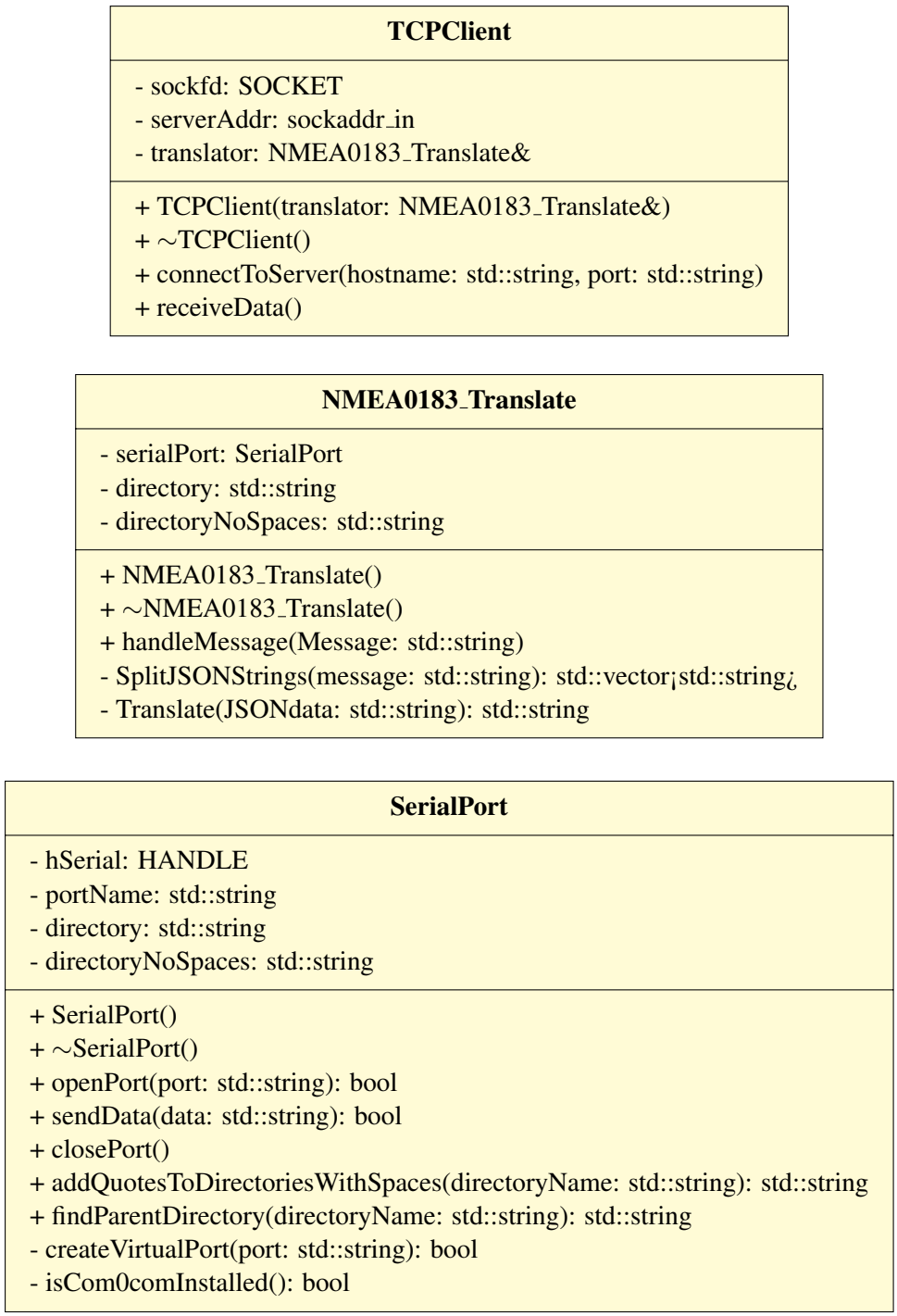

Figure 3.5: UML diagram

Chapter 4

Implementation

4.1 Hardware

4.1.1 Breadboarding

Initially I laid my design out on a breadboard with a pin out from the Pi. This allowed me to verify the design and use it in testing before needing to design and print a PCB, allowing me to start software design and testing earlier. As mentioned previously the design used had some issues that were found during this stage (reference: 3.1.4). Once corrected I could use my Raspberry Pi to interface with CAN networks.

To test I also needed a CAN network, operating as part of an NMEA2000 network. I didn't have access to a commercial N2K backbone so I created my own. Power is typically shared on the backbone however, for my purposes, I can power devices directly using a separate power supply. Aside from power the backbone is just shielding and two data lines connected by two 120 ohm resistors, something I can recreate on a breadboard. I don't need shielding for a small design such as this due to the size and lack of interference.

4.1.2 PCB design

Before this project I didn't have any experience in PCB design and CAD software. My starting point was to find free software which provided easy tutorials. I started by using DesignSpark PCB 11.0 a CAD software with a free version [18].

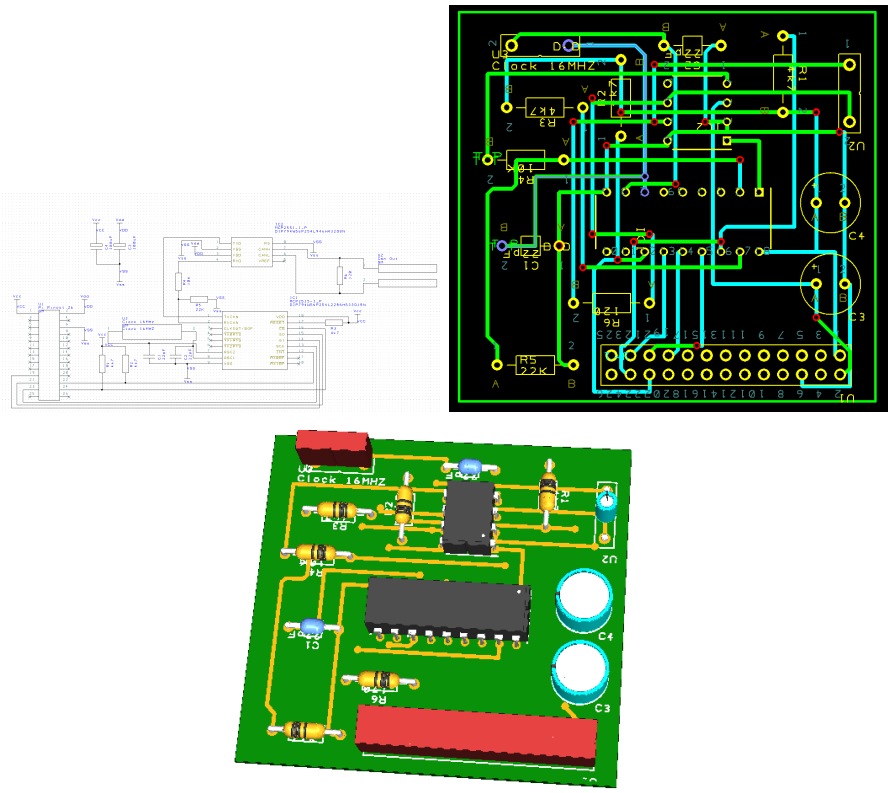

Figure 4.1: PCB Design: iteration 1

My first design had a few iterations. Initially, I included no capacitors on Vcc or Vss and I then added them to cope with interference and filter out unwanted fluctuations and spikes. However the design still had some issues; the capacitors from the clock , and the board was large and inefficiently routed with excessive vias. Additionally, the Design Sparks software isn't used by the university PCB printing team and I needed to use another CAD tool.

My second design used KiCad 8.0 software [7] and had some improvements. The board is smaller and more efficiently routed. A ground and power plane were also included to help with interference reduction. The board is also designed for a newer Raspberry Pi with 40 pins instead of the old 26-pin header. This was the PCB I eventually printed. When testing the PCB I had some issues, mainly that the older Raspberry Pi I was using had 26 pins. This wouldn't be an issue except the physical connector is blocked by the video out so that needed to be desoldered and removed.

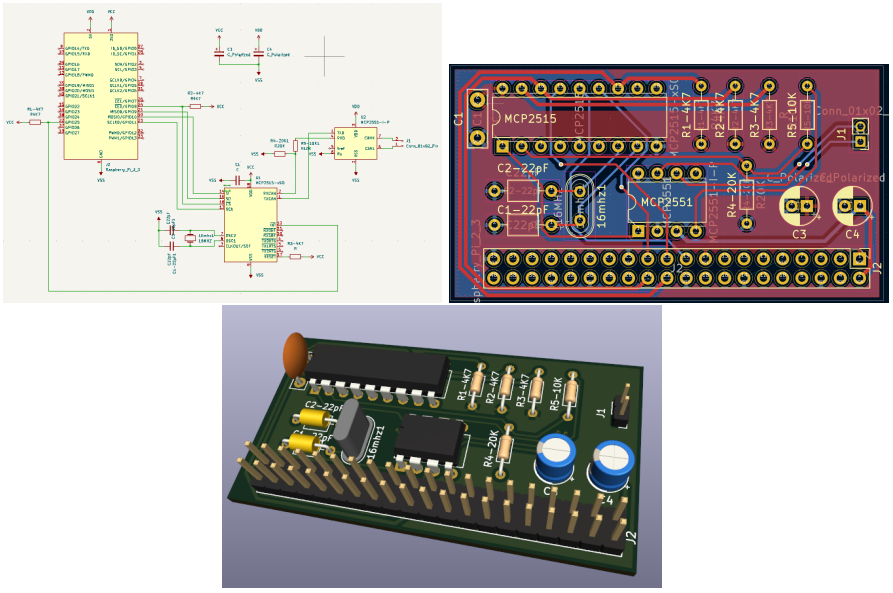

Figure 4.2: PCB Design: iteration 2

4.1.3 Usage

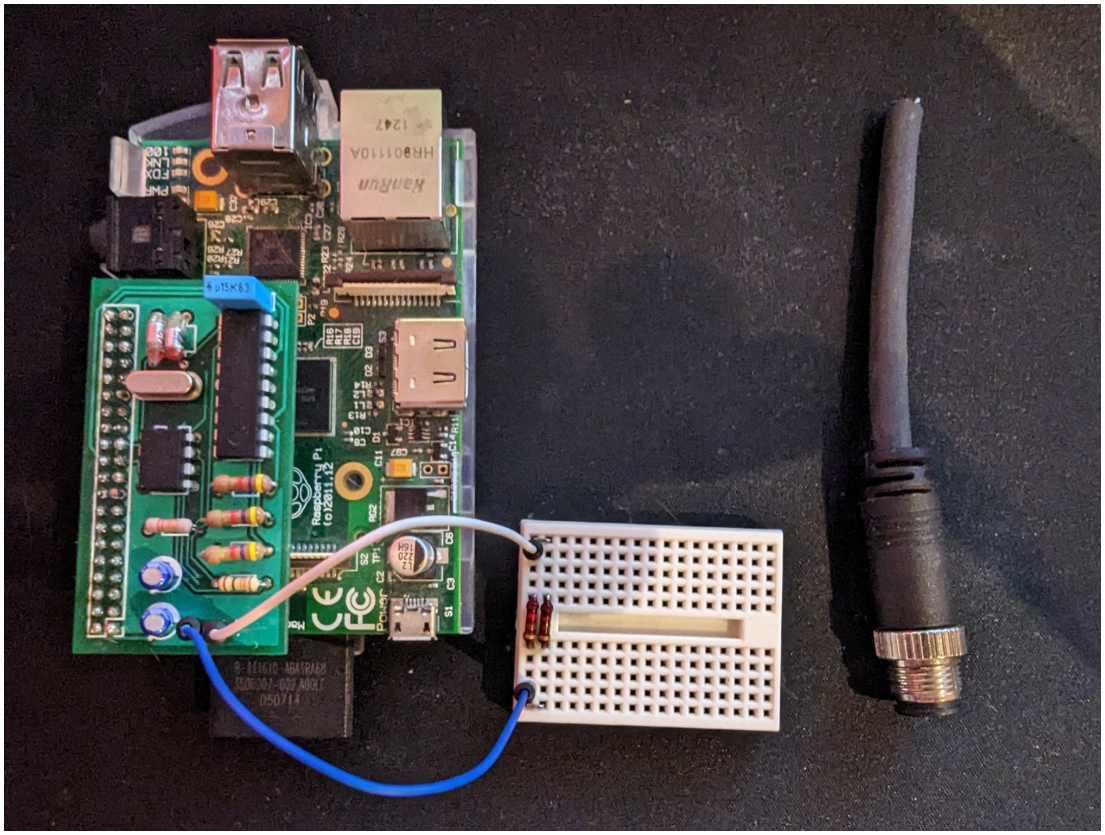

Figure 4.3: PCB with Pi model bus and NMEA2000 connector

The PCB has 2 pins out for CAN-H and CAN-L and power is provided by the Pi. The idea is to power the Pi off the 12V NMEA200 network. This can be achieved with an N2K connector for the backbone and soldering the two data lines to the pin out and the 2 power lines to a 12V to 5V converter to power the Pi and a small switch. The switch allows an Ethernet-capable Windows laptop to receive information from the Pi and display it.

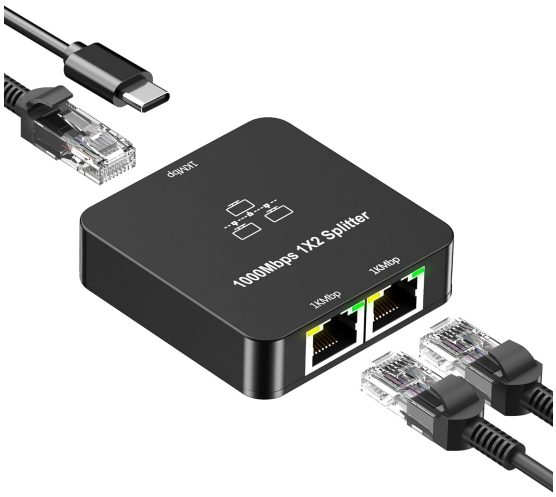

Figure 4.4: Example switch: priced £9.99 04/04/2024
[Source: https://www.amazon.co.uk/Ethernet-Splitter-1000Mbps-Network-Simultaneously-Black/dp/B0CQ1WSJ28]

With a small and inexpensive switch capable of supporting 2 devices the Pi can be connected to the switch and a laptop then also be attached. Data can then be received and used by the laptop and internet access isn't required. Notably with a larger switch or network multiple laptops can access data simultaneously although I am unsure of the use case for this.

4.2 Software

4.2.1 Raspberry Pi

On the Pi the first step is to interface with the CAN controllers using SPI. First I enabled SPI and added the data in, frequency and the interrupt pin in a config file. Once SPI is enabled I used the can-utils library installed using apt. When set up to the correct baud rate for an NMEA2000 bus, the candump will output can packets to the standard out. At this stage, I could verify that I was receiving data from my mock CAN bus and that it initially wasn't working and I had to check my design (section: 3.1.4).

Once the fixes had been implemented I tested again and received data. I installed and built the canboat library and started command line tools, correctly connected together. Starting with the candump it's piped into the candump2analyzer tool

which produces an acceptable format for the analyzer. The analyzer tool is the key part of this process and is what reads, translates and outputs CAN data as readable JSON. This can then be piped into a tool to share data, in our case a tool called n2kd which takes JSON stores it and runs as a TCP server. Now any device on the same network can access received data.

Once this had been implemented I ran a simple Python script on my PC to test and was able to quickly check for functionality. To ensure the Raspberry Pi is robust to reboots I set up a shell script to execute on boot that set up the CAN hardware as a peripheral and started the canboat tools. Additionally to aid in debugging I installed tunnel software, called PiTunnel [16], to allow remote and secure SSH from outside the local network.

The next issue is connecting to the Pi. This can be done using the IP address of the Pi but for this to be consistent a static IP would be needed. Setting up a static IP is a little complicated and a simpler solution was to allow connections using a hostname instead. Avahi is a tool that allows a Raspberry Pi to be connected using a hostname in place of an IP address. For my Pi that name is nmea2000Pi. This name doesn't change and allows the Pi to be robust against network and IP changes.

4.2.2 Computer

The approach to implementation was progressive starting with TCP then translation then piping into third-party software. This allowed for testing during development as translation can't be tested without data from the TCP client. I finished by 'wrapping' it in a Windows interface.

TCP client

The first step is to receive data from the Pi. Initially, a simple Python TCP client allowed confirmation that the server worked and data was being sent. Implementation in C++ proved difficult due to deprecated functions on Microsoft's own reference TCP client. A fix can be achieved by specifying a minimum version of Windows to compile for. In this case, that is version 0x0501 or Windows XP,

note that this doesn't prevent the application from working on the latest Windows version. Additionally on failure the application exits with an error code.

Translation

Essentially translation is a lot of string manipulation and can be broken down into sections. Firstly it splits the message into individual JSON objects as multiple messages from different instruments can be received at once in a single string. Then iterates over through the list of messages translating each one.

```
{
    "timestamp":"2024-04-09-13:24:32.195",
    "prio":5,
    "src":35
    "dst":255,
    "pgn":130311,
    "description":"Environmental Parameters",
    "fields":
    {
        "SID":47,
        "Temperature Source":
        {
            "value":0,
            "name":"Sea Temperature"
        },
        "Temperature":7.10
    }
}
```

Listing 3: JSON example (repeated)

While it would make sense to store the JSON as a special JSON type I haven't done this. Firstly c++ doesn't natively support JSON so a non-standard library would be needed. Secondly, there is no need to treat it as JSON due to the consistent format canboat produces. The only fields of consequence are description and the unique field that holds the values obtained. As such they can just be searched for using standard string regex search.

```
1 Distance Log$--VLW,{Log},N,{Trip Log},N
2 Environmental Parameters$--MTW,{Temperature},C
3 Speed$VWVHW,,T,,M,{Speed Water Referenced},N,,K
4 Water Depth$SDDBT,,f,{Depth},M,,F
```

Listing 4: Translation rules example (repeated)

Finding the description is as simple as searching for the pattern "description":"then extracting the all text up to the next close quotation. With this value the translation text file 6 can be searched for the rule line by line comparing the obtained value with the start of each line.

Once a rule has been found the description is cut off the start. Now the application has the correct NMEA0183 2.1.2 sentence with blank value sections marked by {<description>}. The translate function searches for these keys in the JSON file and matches the value. Once the value is found the section of the NMEA0183 sentence is replaced with the value. This repeats until a regex search reveals no more empty sections. Regex makes this process simple as it returns matches and locations that fit the pattern.

COM port

Creating your own virtual COM ports requires the design of a custom driver that needs to emulate hardware and do this for many different systems. Due to these challenges and as mentioned during design, I'm using com0com [3]. I packaged the com0com binaries with my own binaries and I use them to create the virtual COM ports. Constructing the ports is as simple as checking if they already exist and running a single command to create them if not. Now that a pair of COM ports exists using them requires some set-up. Firstly a file to write data is needed and stored as a handle. Once this exists parameters and timeout values need to be set. OpenCPN uses a baud rate of 4800 by default a byte size of 8, 1 stop bit and no parity. Timeouts can be set to the desired length.

Once the port is set up, it's as simple as writing to the file. For OpenCPN to receive from the other port it simply needs to be added in the connections section of the options menu with default baud rate and the checksum option not selected.

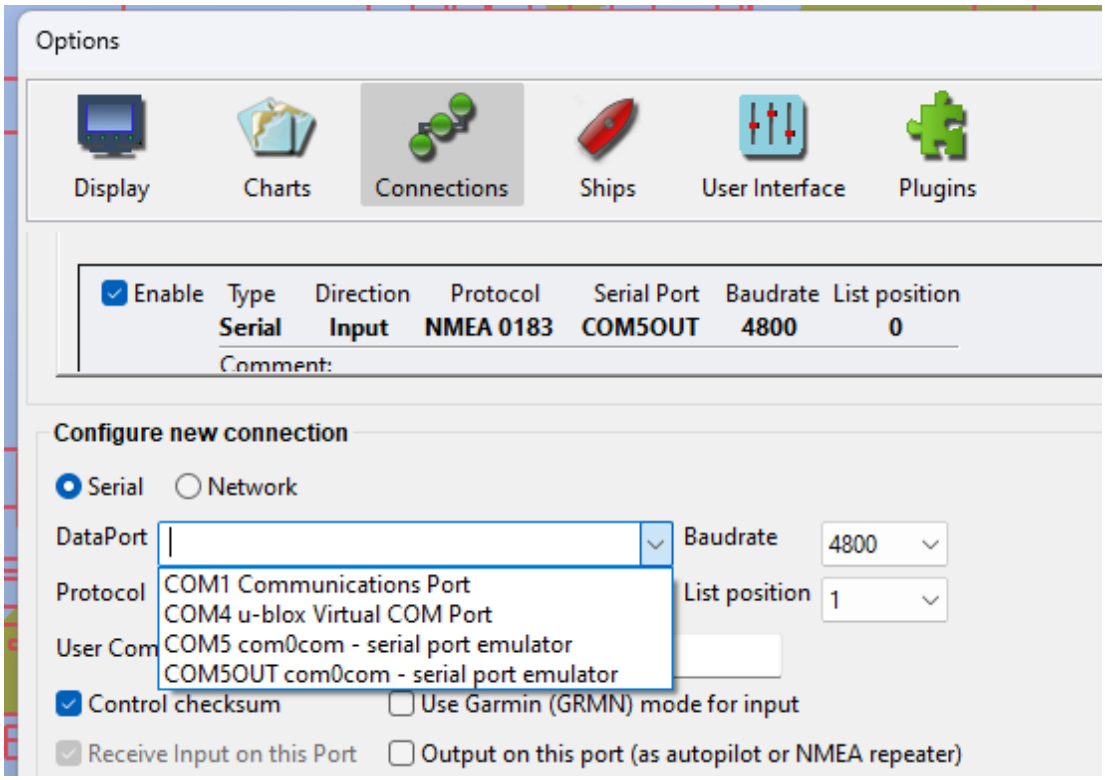

Figure 4.5: Virtual COM ports

While a checksum is important there is little risk of corruption as the data isn't actually being sent on a physical serial wire. A checksum is something that should be implemented later.

Translate process

All the previous sections when combined receive data from the TCP server and send it to OpenCPN in a usable format. I compiled these together, with a main loop to run them. The resulting binary can then be run by the Windows wrapper.

Windows functions

For the application to run as a Windows process it needs a few things. First, a callback function that handles startup, exit and any external interaction via Windows. On start the program will set up the system tray icon and start the main loop. Other callback sections include a right-click of the icon in the system tray to bring up the option to exit.

The main loop will start the binary produced earlier as a process then loop checking that it is still running. If the process has finished but exited with a failure it starts the process again. This loop will 'sleep' to avoid putting unnecessary load on the processor. Should an exit call be received the main loop stops and the process is stopped.

Error handling

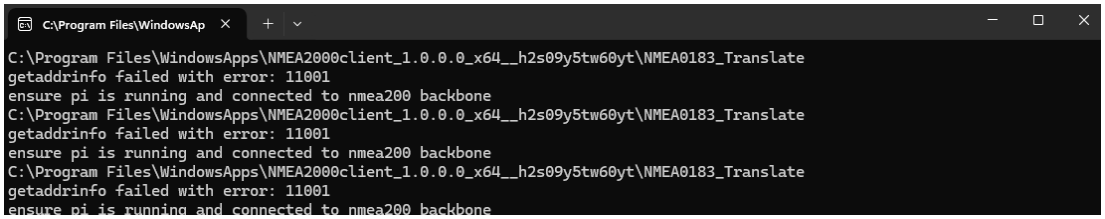

Figure 4.6: Example error: caused by Pi being off

Error handling for this project is extremely important. Many Windows and third-party software commands can fail out of my control and these need to be handled gracefully. To name a few errors that can happen: the TCP might not exist or the server may refuse the connection, the user may not give admin access needed to create ports or opening the rule file may fail. Essentially every time a call to a function may fail or return an error I write where it failed and the error code to the standard error stream and exit with a failure code. This exit results in my Windows wrapper restarting the process and trying again.

MSIX

MSIX is relatively simple to build by following Microsoft's documentation [10]. Three files are needed: appxmanifest.xml and myMapping.txt any other files a generated based on these. The appxmanifest file describes the application including but not limited to: the logo, privileges such as file access and the name. The myMappings file is simply a list of what files to include in the package and where to place them inside the package. With these two files and a series of command line tools an MSIX package can be produced but not used yet. To install the application it needs to be signed with a certificate and the certificate needs to be installed. Then simply double-clicking the package will install and run it.

The main challenge I had with MSIX was using relative file paths. Constant absolute file paths can't be used as the installation and location are handled by Windows, so the path varies from machine to machine. Relative file paths in Windows are by default from an OS folder when running as an MSIX application. The fix for this is to find where the current executable is running using the Windows API and then append to this to create an absolute path. The issue was many Windows functions are exclusive to .NET or the kernel so finding the location of the executable was difficult; eventually, I found a function that could handle this called 'GetModuleFileName'.

Chapter 5

Results

5.1 Raspberry Pi

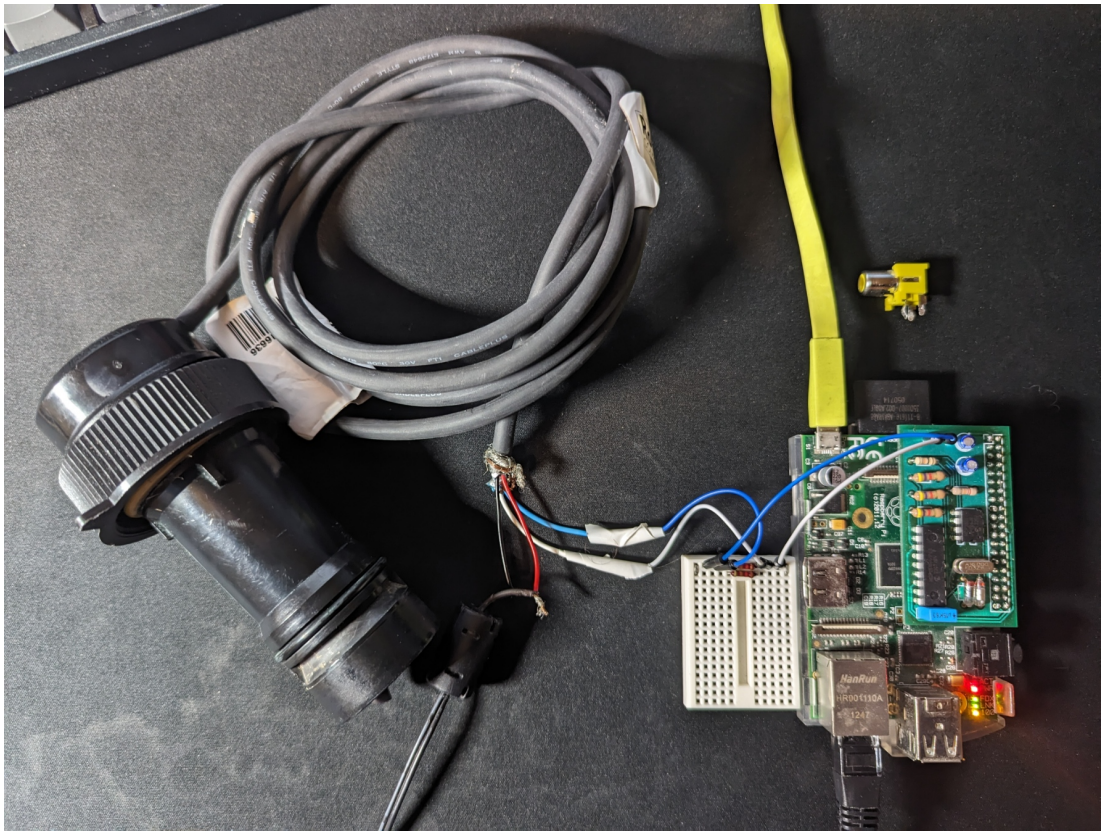

Figure 5.1: Pi running with log

Figure 5.2: NMEA2000 connector

The Raspberry Pi setup is simple. All that's needed is to install my disk image onto an SD card and start the Pi with the HAT attached and a network connection. My only issue was is the HAT is designed for the newer Raspberry Pis and not the original with the shorter pinout. The pins have the same function but the video output had to be desoldered and removed as seen in yellow at the top left (figure: 5.1).

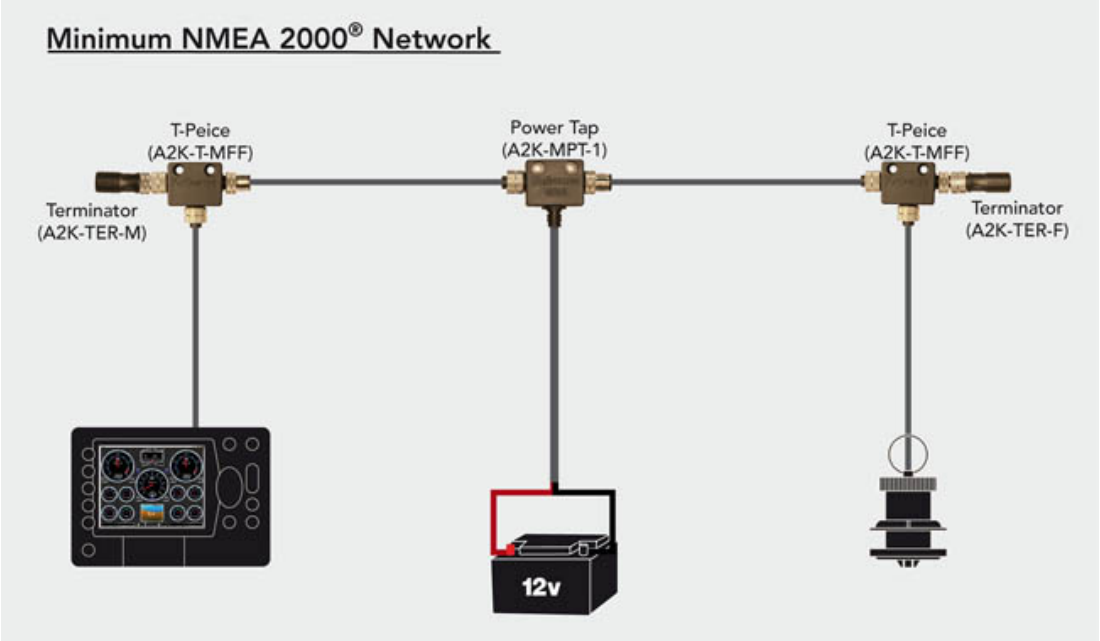

Figure 5.3: NMEA2000 network example
[Source: https://www.yachtbits.com/blog/nmea2000_basics]

In my example setup the Pi isn't connected to an actual backbone and power comes from a mains supply. The real-world usage wouldn't look like this instead, the Pi HAT would be soldered to a connector (figure 5.2) with the red and black lines wires connected to a 12V to 5V converter for the Pi. That connector would then be used to plug into a standard NMEA2000 backbone. Additionally, the log, as seen in figure 5.1, would be connected to the backbone by a standard connector. This example functions identically to a real-world system and can be used to showcase the application.

Figure 5.3 shows an example NMEA2000 network powered by a 12V battery. Across the top is the backbone with 2 terminators or 120 ohm resistors. Connected to and powered by this backbone are a log and a plotter. The plotter (left) is what I have replaced in my setup.

5.2 Windows application

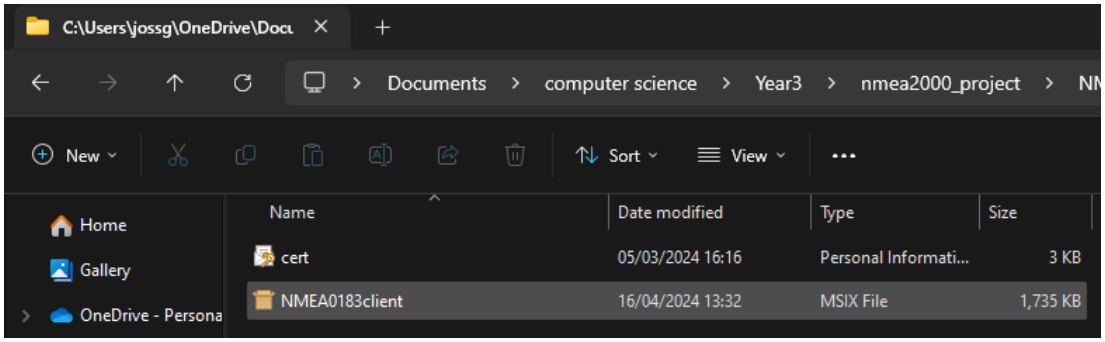

Figure 5.4: MSIX package and certificate

One of my goals was the ease of use for the end user and I believe I've achieved that. Installation is simple, there are two steps.

The first step is to install the certificate (used to authenticate the ID of the owner) as seen in figure 5.4 by double-clicking and continuing to follow the prompts. A password is needed and that is 'ADMIN'. Once the page in figure 5.5 is seen simply select trusted people as the location and continue.

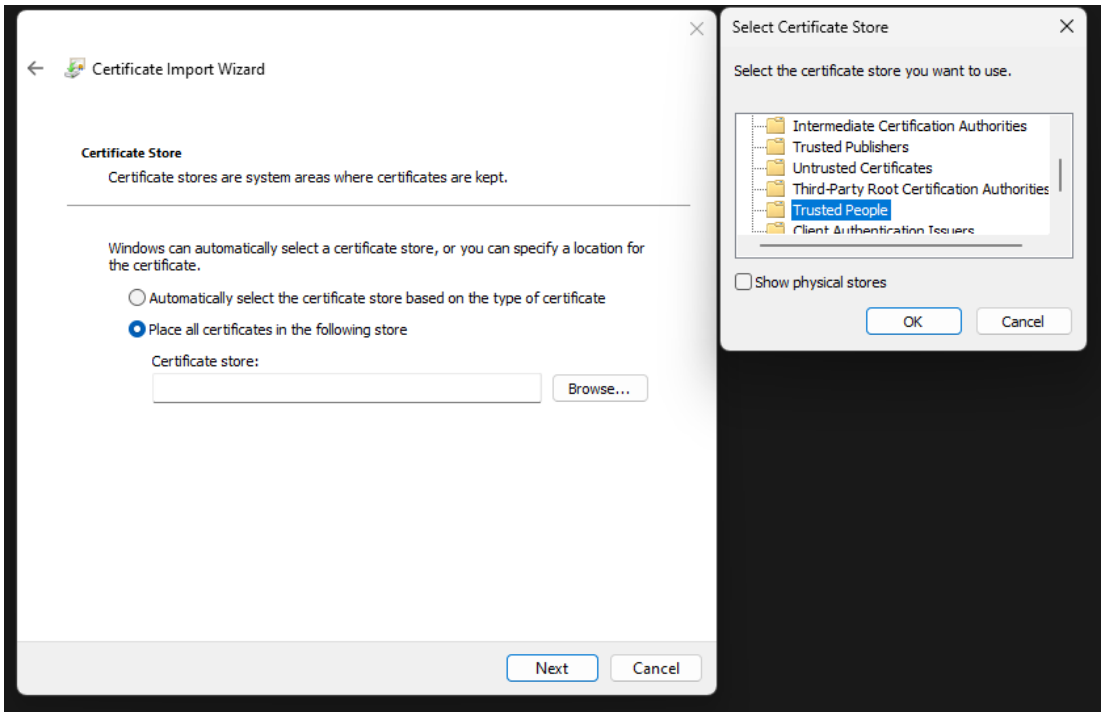

Figure 5.5: Certificate installation

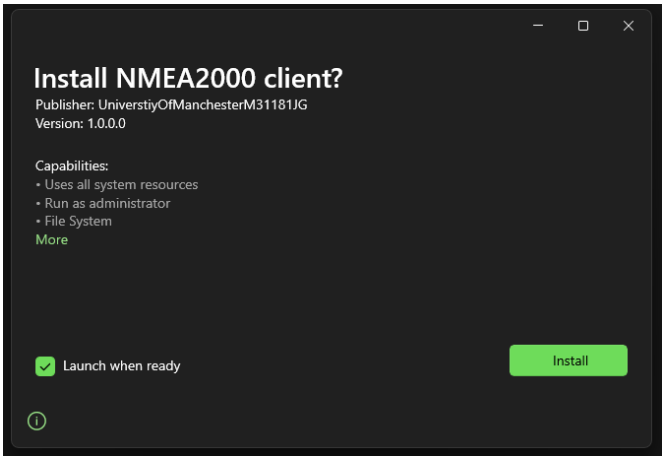

Figure 5.6: Package installation

Once the certificate is installed doubling clicking the application will bring up an install menu (figure: 5.6). Installation is easy, just press install and it will now be available to run either by searching the Windows menu or by adding it to the desktop.

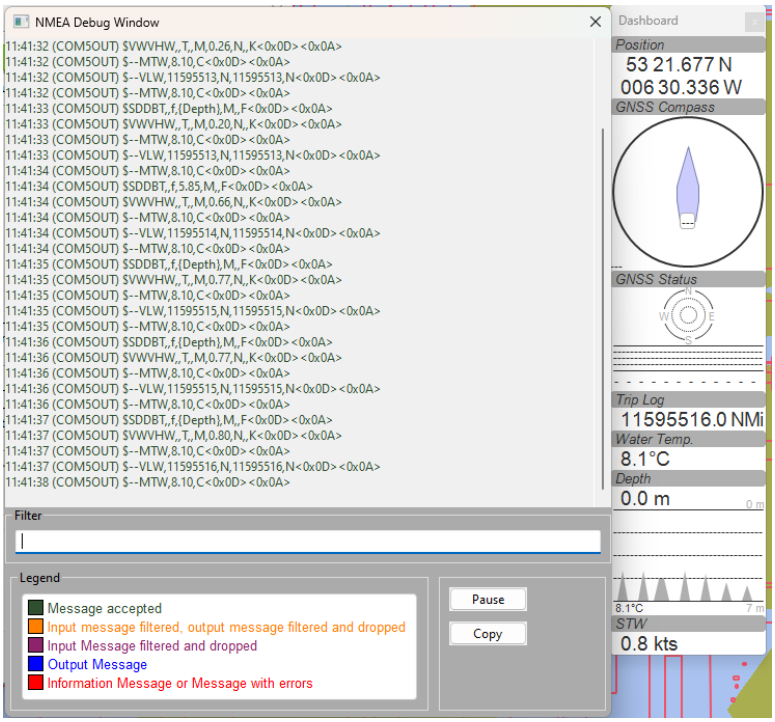

Figure 5.7: Application running with debug window

Once the application is running it will automatically attempt a connection with the Pi. The first run requires admin access to make the ports. If connected the only thing left to do is set up OpenCPN to receive data. Simply add the COM5OUT port as a connection and you now have access to data from the vessel.

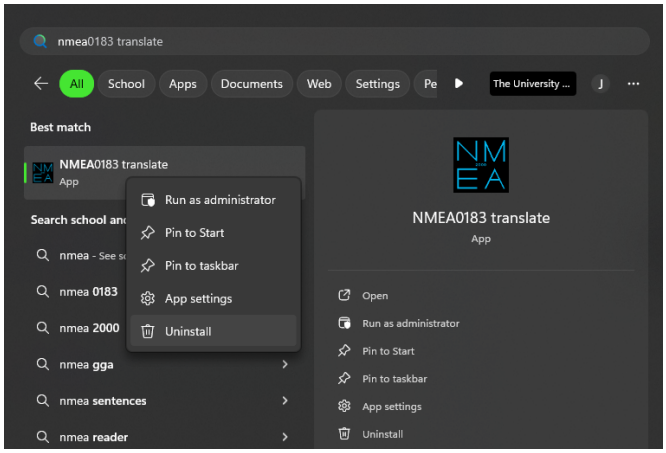

Figure 5.8: Application uninstall

Uninstalling the application is the same as any other Windows store app. Right-click the application and select uninstall.

Adding rules

Adding translation rules is slightly more involved and while a user with no programming skills should be able to create rules, it does require some effort. Included with my code are two files, the first is a PDF on the NMEA0183 protocol (reference: [2]) and the second is a Python script.

```
{
    "timestamp":"2024-04-09-13:24:32.195",
    "prio":5,
    "src":35
    "dst":255,
    "pgn":130311,
    "description":"Environmental Parameters",
    "fields":
    {
        "SID":47,
        "Temperature Source":
        {
            "value":0,
            "name":"Sea Temperature"
        },
        "Temperature":7.10
    }
}
```

Listing 5: JSON example

The first step is to know what data is on the network by running the Python script. All JSON data received from the Pi will be printed out. To start with add the description from each JSON object as the start of each line in the file 'TranslateLookup.txt'.

```
1 Distance Log$--VLW,{Log},N,{Trip Log},N
2 Environmental Parameters$--MTW,{Temperature},C
3 Speed$VWVHW,,T,,M,{Speed Water Referenced},N,,K
4 Water Depth$SDDBT,,f,{Depth},M,,F
```

Listing 6: Translation rules example

Then reference the provided PDF for the correct NMEA0183 sentence and add it after the description. The final step is to simply add the labels of relevant data to the correct location inside the sentence with { } around the data. As shown in listing 6.

5.3 Testing

When testing my code I used a manual test plan. Some errors that occur are ones that I can't force to happen so I simply replaced the called function with an integer representing an error.

5.3.1 TCP client

Test Case	Details	Result
WSAStartup	Initiates use of the Winsock, Window's network API. Expect exit and reattempt on fail. Failure of this function indicates an issue with Windows.	Pass
WSAStartup	Expect startup and no error under normal run	Pass
getaddrinfo	Gets the address of the host (Raspberry Pi). Expect exit and reattempt when the Pi isn't running or not on the local network.	Pass
getaddrinfo	Expect address to be found when Pi is running on the local network.	Pass
socket	Attempt to connect to server at host address. Expect exit and reattempt when the server doesn't exist.	Pass
socket	Expect a successful connection when Pi is running the server.	Pass
recv	Receive data from TCP server. Expect exit and reconnect when the receive fails.	Pass
recv	Expect data from Pi to be received when Pi is running and connected to the log.	Pass
recv	Expect the program to wait when Pi is running and not connected to any instrument.	Pass

Table 5.1: Test Results

TCP client handles starting Windows functions for TCP and connecting to and receiving data from the Pi. This section won't work if the Raspberry Pi is on the local network or if it hasn't received any data from an NMEA2000 since it booted. The application should continue to attempt to connect to the Pi until it is closed or connected.

5.3.2 Translation

Test Case	Details	Result
Split JSON	**When given a string containing a JSON message should split into a list of JSON messages. Expect a list with one empty string element when given an empty string.**	**Pass**
Split JSON	**When given a single JSON object returns a list with string as the first element.**	**Pass**
Split JSON	**When given multiple JSON objects in a single string expect a list of individual JSON strings.**	**Pass**
Split JSON	**When given multiple JSON objects with internal braces in a single string expect a list of individual JSON strings that are still intact with internal braces.**	**Pass**
Translate	**When the lookup file doesn't exist return error string.**	**Pass**
Translate	**When given a JSON string with a corresponding rule return a correctly formatted NMEA0183 string with correct data.**	**Pass**
Translate	**When given a JSON string with no corresponding rule return an empty string.**	**Pass**
Translate	**When given a JSON string with a corresponding rule but an incorrect identifier return a correctly formatted NMEA0183 string with no data.**	**Pass**
Translate	**When given a non JSON string return an empty string.**	**Pass**
Handle messages	**Any translated message, correct or otherwise, is sent to the COM port.**	**Pass**

Table 5.2: Test Results

Translation should take JSON data in string format and translate using the rule file then send it onto the COM port module. Messages that don't translate properly should still be sent for debugging purposes. Incorrect messages can be viewed inside OpenCPN using the debug menu and should help users when creating their own translation rules.

5.3.3 COM ports

Test Case	Details	Result
Create Port	**When the ports don't exist the program creates them.**	**Pass**
Create port	**When the ports don't exist and the user doesn't give admin access the program exits and reattempts.**	**Pass**
Create port	**When the ports do exist the program doesn't try to create them again.**	**Pass**
Open port	**If the port is deleted after checking it exists the program exits, reattempts and creates the port.**	**Pass**
Open port	**If the port exists the program opens the port with the correct parameters for sending data**	**Pass**
Open port	**If the port exists the program opens the port and if setting the parameters fails the program exits and reattempts.**	**Pass**
Send data	**If the port isn't open the program exits it reopens the port and starts sending data.**	**Pass**
Send data	**If the port is open data is written to.**	**Pass**
Send data	**If the port is open but writing fails the program exits and reattempts.**	**Pass**
Close port	**If the program exits the port is closed.**	**Pass**
com0com	**If com0com is not installed the program installs it**	**Pass**
com0com	**If com0com is installed the program doesn't attempt to install it again.**	**Pass**

Table 5.3: Test Results

This module handles creating opening, closing and sending data on COM ports. This section can fail if not given admin privileges when prompted and can also fail when a port is in use.

5.3.4 Windows functions

Test Case	Details	Result
Installing	**The package can be installed by the user when the correct certificate is installed.**	**Pass**
Installing	**The package can't be installed when no certificate or the incorrect certificate is installed.**	**Pass**
Uninstalling	**The package can be uninstalled from the Windows menu**	**Pass**
Running	**If installed the program can be run from the Windows start menu.**	**Pass**
Running	**When running an icon for the program appears in the Windows system tray.**	**Pass**
Running	**When not running an icon for the program doesn't appear in the Windows system tray.**	**Pass**
Running	**When running right-clicking the system tray icon brings up an option to exit the application.**	**Pass**
Exiting	**When the exit option is selected the program closes.**	**Pass**
Running	**Multiple instances of the same program don't crash and one and only one instance functions properly.**	**Pass**

Table 5.4: Test Results

The Windows functions include installing, uninstalling and exiting properly.

Chapter 6

Conclusion

6.1 My project

This project has produced two items. The first is a Raspberry Pi HAT capable of interfacing with CAN networks and more specifically NMEA2000 networks. The second is an application for Windows that allows data from a Pi with said HAT to be sent to OpenCPN for navigation uses.

My Pi HAT design allows any Raspberry Pi model to interface with a CAN network. While I made this HAT intending to use it for NMEA2000 networks it could be used in a number of other projects. For example to interface with the electronics of a car although this isn't advisable unless you're highly experienced in the automotive world. The Pi HAT is a simple two layer design that is easy to print and solder even for inexperienced solderers.

The application I created runs in the background on a Windows machine and provided the Raspberry Pi is set up with the HAT and provided image on the local network then any data received of an NMEA2000 network can be displayed in OpenCPN. The application is also designed such that it can be put on the Windows store for better Windows integration.

6.2 Lessons

I learnt a lot of lessons over this project, mainly to do with PCB design. One of the main takeaways was not to just trust online tutorials as that caused me some

issues but it also helped me to understand schematics and PCB design. I found that I had to do a lot more research on the topic than expected and this helped me fix the design but also understand it. I also received a lot of advice on good practices when designing PCBs such as using ground and power plains as well as capacitors to reduce unwanted fluctuations and spikes.

I also learned that even official documentation can be wrong or outdated when working with Windows. I have learnt a lot about programming for Windows after starting with almost no experience and feel a lot more confident to make Windows applications in the future.

Finally while I have lots of experience sailing, I have very little experience of boat hardware and standards. I have learnt a lot about communication standards such as CAN, NMEA2000, NMEA0183, and SPI

6.3 Goals and meeting criteria

My main motivation for this project was price with the secondary goal of ease of use and setup. I believe I've achieved both of these and laid the groundwork for further extension of the project.

Repeat of initial goals:

- Replace the need for a plotter meeting the requirements:
  - Display data received from a vessel's instruments in real time
  - Integrate any data into chart software
  - Cost nothing in software
- Total cost less than £100 in hardware excluding a laptop
- Installation will be simple and require no technical skills
- Application will have scope for expansion

As shown earlier in figure 5.7 my application is sending real time data to OpenCPN where it is being displayed over a chart. Both my software and OpenCPN are free to use and achieve the aims set out in my introduction.

As for the goal of price the total cost of the system is roughly £70 meeting my aim of sub £100. As mentioned earlier a small network switch costs as little as £9.99 (figure: 4.4). Other components such as a Raspberry Pi and the PCB cost £30 and sub £10 respectively. Plus the cost of components and connectors which I estimate to be below £20 based on experience during the build process. The sub £20 prices include the MCP2551 and MCP2515 chips and NMEA2000 connector as well as general components such as resistors, capacitors and the clock. That price doesn't include the cost of a laptop. My reasoning for this is most people should already own a laptop.

`https://www.amazon.co.uk/Raspberry-Pi-Model-Quad-Motherboard/dp/B01CD5VC92`

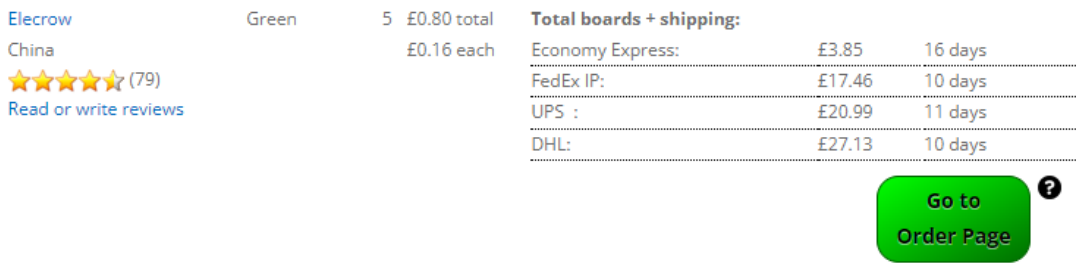

Figure 6.1: Quote for PCB printing cost
[Source: https://pcbshopper.com/]

As for ease of use and setup I believe I've achieved that goal however, this is a subjective aim. Given more time and budget I would engage in user testing and use feedback to improve my design. If I was able to produce many of the Pi HATs I would conduct a real-world test involving actual sailing vessels and produce quantifiable data on the usability of my design from actual users. Additionally, I have identified several areas I would like to add to or improve.

Firstly I would in the future increase the range of products my application is compatible with. That includes a variety of plotting software, not just OpenCPN as well as more operating systems including MacOS and possibly Linux. I would also like to add the option to interface with the Pi via USB to remove the need for a switch. having a wireless version for Android or IOS is also an option. This would

allow users to use a handheld device which much more portable and waterproof.

I believe my choice of design and use of OOP makes these additions possible without major code restructuring. For example, to add USB support I would need to replace the TCP tool on the Pi with a tool to send via USB. This would require no changes to the other tools. Next would be replacing the TCP object in my code with a USB object that parses data to the existing translation object. This whole process would require little modification of existing code and I believe this is also true of other additions I can make to my code.

I would also like to increase the number of translation rules my program comes with at installation. This would decrease the need for the user to have the knowledge needed to create their own rules and make my design more user-friendly. I believe the best way to achieve this goal is to release my code and allow contributions from people with hardware I don't have access to test. Alternatively, if I had access to more hardware such as an anemometer or NMEA2000 GPS then I could create my own rules.

From the just software side I believe the creating of rules is the hardest task for a user when setting up my system. While a simple tutorial should allow anyone to create rules, I would prefer is users didn't have to.

Finally the largest barrier to entry, in my opinion, is soldering the PCB. I believe this is the hardest part of implementing this project that the user has to do themself. Printing of the board is handled by a company but Soldering requires some skill. I don't consider this a major barrier to entry as this skill can be picked up easily by using online tutorials. While I don't consider A soldering iron part of the cost, this is a tool usable in the future. Furthermore with the existence of makerspaces (reference: [20]) buying a soldering iron isn't even required.

In summary all my goals have been achieved with the exception of ease of use. While I believe I have made this project simple to install I can not justify this conclusion without user testing.

Bibliography

[1] canboat/canboat, 2024. URL: https://github.com/canboat/canboat/wiki.

[2] Klaus Betke. The nmea 0183 protocol. 2001. URL: https://www.tronico.fi/OH6NT/docs/NMEA0183.pdf.

[3] com0com Development Team. com0com: Null-modem emulator. Software, 2003–Present. URL: https://com0com.sourceforge.net/.

[4] Electronic Industries Alliance (EIA). EIA-232 interface between data terminal equipment and data circuit-terminating equipment employing serial binary data interchange. Technical Standard, 1969.

[5] eLinux.org Contributors. Can-utils, Accessed 2024. URL: https://elinux.org/Can-utils.

[6] Steve Corrigan HPL. Introduction to the controller area network (can). *Application Report SLOA101*, pages 1–17, 2002.

[7] KiCad Development Team. *KiCad: Electronic Design Automation Suite*. KiCad Development Team, 1992–Present. URL: https://www.kicad.org.

[8] Microchip Technology Inc. *MCP2515 Stand-Alone CAN Controller with SPI*. Microchip Technology Inc. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf.

[9] Microchip Technology Inc. *MCP2551 Datasheet*. Microchip Technology Inc. URL: https://www.sparkfun.com/datasheets/DevTools/Arduino/MCP2551.pdf.

[10] Microsoft Corporation. *MSIX: Package Format*. Microsoft Corporation, 2018. URL: https://docs.microsoft.com/en-us/windows/msix/overview.

[11] Alton B Moody. The nautical mile. *The International Hydrographic Review*, 1950.

[12] National Marine Electronics Association (NMEA). NMEA 2000 standard for serial-data networking of marine electronic devices. Technical Standard, 2018. URL: https://www.nmea.org/nmea-0183.html.

[13] National Marine Electronics Association (NMEA). NMEA 2000 standard for serial-data networking of marine electronic devices. Technical Standard, 2022. URL: https://www.nmea.org/nmea-2000.html.

[14] OpenCPN Development Team. OpenCPN: Open source chart plotter / navigator. Software, 2008–Present. URL: https://opencpn.org.

[15] OpenPlotter Development Team. OpenPlotter: Free marine navigation platform. Software, 2016–Present. URL: https://openmarine.net/openplotter.

[16] PiTunnel Development Team. *PiTunnel: Secure Tunneling for Raspberry Pi*. PiTunnel Development Team, 2018–Present. URL: https://www.pitunnel.com.

[17] Raymarine. SeaTalk protocol specification, 1989. URL: https://www.raymarine.com/en-us/download/seatalk-seatalkng-softwaref.

[18] RS Components. *DesignSpark: Free Electronics Design Software*. RS Components, 2009–Present. URL: https://www.rs-online.com/designspark.

[19] Simrad. SimNet installation manual, 2005. URL: https://softwaredownloads.navico.com/Simrad/SimradYachting_Software%20-%20Copy/Downloads/documents/OCEANIS3_BEN_IM_EN_988-10300-001_w.pdf.

[20] Nick Taylor, Ursula Hurley, and Philip Connolly. Making community: the wider role of makerspaces in public life. In *Proceedings of the 2016 CHI Conference on human factors in Computing systems*, pages 1415–1425, 2016.

Appendix A

Appendix Title

A.1 Key terminology

Chart - a map for the ocean with navigational information not limited to depth, Buoys and landmarks.

SPI - Serial Peripheral Interface

NMEA - The National Marine Electronics Association, an organisation that sets standards for communication between marine instruments

Checksum - a mathematical calculation that produces a number for error checking purposes