Design and development of a robotic fish tracking vehicle

Nikolai Lauvås

January 2020

TTK4550 Project report Fall 2019

NTNU – Trondheim
Norwegian University of
Science and Technology

Abstract

The Fish Otter is a robotic fish tracking vehicle under development at NTNU, with the goal of making better positioning of acoustic fish tags available. This will be achieved by utilizing multiple vehicles to make tag detections. Using the time difference in signal arrival can then be used to estimate position of the tagged fish. This report describes the system integration of a single vehicle that has the ability to perform maneuvers while carrying a hydrophone that detects acoustic fish tags. The design is also documented, because multiple Otters will be made at a later stage.

System requirements for the vehicle are identified, and hardware and software to fulfill these requirements are developed. The hardware is described in detail, including the modifications done as part of this project. In addition, the software design is performed from the level of customizing and installing an operating system, to the application controlling peripheral sensors and actuators. The resulting system is then field-tested in one dry trial and two sea trials, with mostly positive outcomes.

Preface

The Otter is an autonomous surface vehicle that is under development at NTNU. The Otter has been the topic of my project work at NTNU in the fall of 2019, and this report documents my work. I am a 5th year student of engineering cybernetics, specializing in embedded systems design. As a part of my studies, I have also attended the courses "TTK15 - Oceanographic instrumentation and biotelemetry" and "TTK22 - Software tool chain for networked vehicle systems", both of which have helped me in my work on the Otter.

Acknowledgements

I want to thank professor Jo Arve Alfredsen for supervising this project, and Alberto Dallolio for co-supervising it. Some prior work on the Otter had been done by João Fortuna, which was used as an inspiration during work on the DUNE integration. The helpfulness of Frederik Stendahl Leira in answering questions about the LSTS toolchain was a great resource. The sea trials could not have been performed without the help of Terje Haugen. Gunnar Aske helped getting a server and domain location for the wiki.

NTNU
Norwegian University of
Science and Technology

Faculty of Information Technology and
Electrical Engineering
Department of Engineering Cybernetics

PROJECT ASSIGNMENT (15 Stp.)

Name: Nikolai Lauvås

Program: Cybernetics and robotics

Title: Design and development of robotic fish tracking vehicle

Title (Norw.): Design og utvikling av robotisert fiskesporingsfartøy

Project description:

The project aims to equip autonomous surface vehicles with an advanced acoustic fish telemetry payload to create a novel platform for robotic search, localization and tracking of migrating fish as well as other small and evasive underwater objects. The integration will shift the current operational limits of fish/underwater object tracking by making new enabling technology available for researchers and facilitate new discoveries within fish movement ecology and the marine sciences. The project includes the following tasks:

- Get an overview of, and make a wiki-description the current design of the ASV Otter platform (vehicle) with respect to controls, communications, instruments and sensors
- Make an installation of the LSTS DUNE unified navigation environment software adapted to the specific configuration and capabilities of the vehicle. The basic installation should integrate thruster control and power monitoring/management, GPS compass, long range WiFi communication, navigation/status lights and safety features (watchdog)
- Implement basic navigation and control features like manual control, autopilot, waypoint tracking and station keeping/loitering for the vehicle
- Enable front-end vehicle supervision, mission planning and execution through the NEPTUS interface
- Tune, validate and document vehicle performance through sea trials
- Make an integration of the acoustic telemetry receiver payload, including relaying of hydrophone messages from SLIM, message parsing and IMC message generation in DUNE
- Test and validate the complete integration in a sea trial (including 4G cellular communications, if time permits)
- Discuss and document vehicle design and performance

Project start: 19th August 2019

Project due: 17th December 2019

Host institution: NTNU/Department of Engineering Cybernetics

Supervisor: Jo Arve Alfredsen, NTNU/DEC

Co-supervisor: Alberto Dallolio, NTNU/DEC

Trondheim, 19th August 2019
Jo Arve Alfredsen

Contents

Abbreviations

ASV - Autonomous Surface Vehicle.

AUV - Autonomous Underwater Vehicle.

GPIO - General Purpose Input/Output.

GPS - Global Positioning System.

IMC - Inter module communication.

ITK - "Institutt for teknisk kybernetikk", or Department of engineering cybernetics[at NTNU].

LSTS - "Laboratório de Sistemas e Tecnologia Subaquática", or Underwater Systems and Technology Laboratory at the University in Porto.

NTNU - "Norges teknisk-naturvitenskapelige universitet", or Norwegian University of Science and Technology.

OS - Operating System.

PoE - Power over Ethernet.

PPS - Pulse-Per-Second.

RPI - Raspberry Pi.

RPM - Revolutions Per Minute.

SoC - System on Chip.

USV - Unmanned Surface Vehicle, not necessarily autonomous.

1 Introduction

In a country with as long a coastline as Norway, the ocean naturally becomes an important resource. The fishing industry, and in more recent years, aquaculture, have for centuries provided the Norwegian people with both food and valuable goods for export. To keep the industry thriving, good management is dependent on knowledge about life below the surface. One aspect of this is observing the movement of individual fish, to see how it behaves and acts. The Fish Otter project at NTNU aims to provide a platform for getting detailed information about these movements.

1.1 Background

At ITK NTNU[1], there is a long tradition for doing research on acoustic fish telemetry. This dates back to its founder, Jens Glad Balchen, and his experiments on tracking the life and behavior of fish in Hopanvågen during the 1970s. In one of these experiments, he developed what he called a fish spy. This was what is now called an unmanned surface vehicle (USV), and its goal was to follow right above a fish that had been tagged with an acoustic transmitter. The position of the vehicle would thus be the position of the tracked fish. Lack of funding ultimately led to the project being abandoned after several development iterations[8], but research on acoustic fish telemetry have continued at ITK.

Developments in digital technology has been made since the fish spy project was abandoned, and mass production has also reduced the cost of components. This means that computing power has become more accessible, paving the way for improvements both in the acoustic fish tags, and in hydrophones that have the ability to automatically detect tag registrations, not just sound. In addition, satellite based positioning systems like GPS has made sensor positioning available, opening the possibility to place telemetry on a map for context.

Aquatic animals are acoustically tagged in order to collect information about individuals, like position, temperature, depth and conductivity. Acoustic signals are used because radio waves are heavily attenuated by the salt water, reducing the range to a level where its impractical to use. This also means that the radio waves used by satellite positioning does not reach tags below the surface. For animals that surface at regular intervals, positioning at these moments are possible. For animals that never or seldomly surface, or if greater resolution is desired, other positioning strategies must be utilized.

There are great variations in the design of acoustic tags. The size of the target animal adds constraints on size. This also heavily influences what frequency the tags use, how many sensors can be added, and battery life. Sensors are often included to give information about conductivity, pressure (depth[2]), temperature, and information about the animal, like heartbeat, tail movement or jaw movement [7]. Some tags send continuous signals, but sending discrete signals has become the norm, because of the lower power usage resulting in longer battery life.

1.1.1 Positioning of acoustic tags

There are multiple strategies for positioning acoustic tags below the surface, both manual and automated. Manually positioning tagged aquatic animals can be done by utilizing the difference in received signal-strength when moving a directional hydrophone around. To accurately position a tag with this method, it's necessary to hover with the hydrophone right above the tag. This could affect the behavior of the animal, especially if the hydrophone is deployed from a vehicle with propellers. Another drawback of this method, is that it's labor intensive because it's often performed by humans, making long duration series of data hard to gather.[3]

To automatically position acoustic tags, a commonly used method is placing multiple acoustic receivers at strategical locations, or at regular intervals in a grid. The position of the receivers are known, so its inferred that when a detection is done, the animal is in the vicinity of the receiver. Some drawbacks with this approach are; the area of interest has to be decided ahead of time, and it's expensive/impractical to have

[1] Department of Engineering Cybernetics, Norwegian University of Science and Technology.
[2] An almost linear relationship between pressure and depth can be used in water
[3] This method could be possibly be made into an autonomous system, and was one of the concepts tried for the fish spy.

too large of a tracking field. This means that if the tagged fish wanders outside the tracking field, the tag position is lost.

Mounting the acoustic receivers on unmanned vessels with known locations can solve some of these drawbacks. This would require the vessels to follow the moving animal, but would allow for them to keep a distance to avoid disturbing it. As a proxy for tag position, vessel position could be used. To improve on this, multiple tag detections made at different positions could be used to estimate tag position relative to the vessels location by using the time difference in signal arrival. This can be done either by a single vehicle circling the tag, or multiple vehicles in a formation around the tag.

1.2 State of the Art

Localization of fish at liberty has been on researchers minds for a long time, with multiple solutions having been proposed and developed around the globe. In [4], a hydrophone is mounted to an AUV, and multiple strategies for localization of acoustic tags are discussed.

In a more recent integration with an acoustic receiver in an AUV, [6] proposes to use an extended Kalman filter or a particle filter as iterative estimators for position. In an experiment with a stationary fish tag, the AUV travels about the estimated location, in order to gather multiple tag detections. Localization errors below 20 m compared to a GPS position is achieved after 20 transmissions from the fish tag. Using tags that transmitted every seven seconds, that means it would take 140 seconds to locate the tag, so this method is only suited for slower moving targets.

Using multiple vehicles results in multiple detections being made for a single tag signal, reducing the time and increasing the accuracy of tag positioning. This was proposed in [1], and has been further developed in multiple experiments since. In [14], a proof-of-concept is demonstrated through an experiment with a formation of unmanned vehicles carrying hydrophones. The formation follows another vehicle with a acoustic tag mounted below its waterline, estimating its position using the time difference in signal a Comparing the estimated position with the GPS position of the tag carrying vehicle achieves a median localization error of 4.7 m, and an average accuracy of 6.34 m.

Another experiment was performed in [5], with three USVs carrying acoustic receivers and a forth USV carrying a submerged acoustic fish tag. Using an eXogenous Kalman filter, the location of the fourth USV is estimated and compared to other estimators and the GPS position. After the Kalman filter has stabilized, it's possible to locate the fish tag. It also demonstrates the performance benefit of using an eXogenous Kalman filter over an extended Kalman filter.

A more exotic experiment done tracking leopard sharks with an AUV is described by [3]. The system is further extended in [11] to use multiple AUVs that collaborates to position the tagged shark, using a Particle Filter. An important difference between these AUVs, and the one described in [6], is that these use stereo hydrophones, while [6] used a mono hydrophone. Using a stereo hydrophone had the advantage of making relative bearing from the AUV to the acoustic tag available.

1.3 Goal of Project

The long term goal for the Fish Otter vehicles at NTNU is to provide an autonomous multi-agent system for search, localization and persistent tracking of acoustic fish tags beneath the water surface. In collaboration with research groups involved in scientific fish tracking studies at NTNU, the Norwegian University of Life Sciences and others, this has the potential of providing a better understanding of the spatiotemporal distribution and behaviour of fish and other marine organisms.

The scope of the work described in this report, is to do a system integration of a single Otter. Most of the hardware design is already done, but needs to be verified and completed where components and connections are missing. No software design has been performed before this project, and so, must be developed from scratch. This entails the whole software stack, from choosing, installing and customizing an operating system (OS), to designing software that controls and monitors the state of the vessel. Software also has to be developed for communicating with the payload, an acoustic receiver.

1.4 Overview of Report Structure

This project report begins by stating the requirements of the system in Section 2. Based on this, the hardware design is documented and developed further in Section 3. The process of designing the custom software used in the Otter, is documented in Section4. The hardware and software are validated by the one dry test, and two sea trials described in Section 5.

An overview of how well the integration met the system requirements, is given in Section 6. Finally, the conclusion in Section 7 gives a short summary of what has been done, what was achieved and further work.

2 System Requirements

To achieve the goals mentioned in Section 1.3, this project aims to fulfill these requirements for the development.

1. The vehicle computer must have hardware support for controlling/communicating with the GPS, the thrusters, the batteries, the power management, a signal light, land station and payload.
2. The vehicle computer needs software so that it can control power management, thrusters, payload, navigation and communication.
3. The vehicle computer shall have a way to recover if it freezes/stops performing its function.
4. The signal light shall have a way to be turned on and of, to signal system states.
5. The vehicle shall support remote controlled operation through a console. This console should be able to support controlling multiple heterogeneous vehicles, in order to enable support vehicle collaboration.
6. The vehicle needs to be able to follow way-points.
7. The payload of the vehicle shall be able to register detections of acoustic fish tags.
8. The detections shall be timestamped with high accuracy, and be available through some interface for the vehicle computer. The timestamps should be synchronized against GPS time (GPST).
9. The vehicle configuration shall be documented on a wiki.
10. The software developed should be open source.
11. The console chosen should be able to function with other NTNU vehicles.
12. Information about a mission should be logged for review and analysis after finishing.

3 Hardware Design

The Fish Otter is a small unmanned catamaran propelled by two electrical fixed thrusters. The hull is based on the Maritime Robotics Otter, along with custom control hardware and software developed at NTNU to enable use of a custom sensor and navigation's system. The payload is a hydrophone that registers fish tag detections.

Figure 1 gives an overview of the NTNU Otter with highlighted hardware components. The control box is the only component where this project has modified hardware, but a description of the remaining hardware is included to provide an understanding of the software design in Section 4.

3.1 The Hull

Delivered by Maritime Robotics, the hull is designed to be light and portable. At 200cm length and 108cm width, transport and handling can become an issue, but the Otter solves this by being built in a highly modular way. According to the manufacturer, it's possible to disassemble it into parts of under 20Kg each, making it possible to move it by a single person[10].

3.2 Thrusters

The thrusters used by the Otter are two Ultralight 403 motors by Torqeedo. They can provide a maximum propulsive power of 180 W each, and would be the equivalent of 1HP[4] each according to the manufacturer [13]. They have a maximum angular velocity of 1200rpm at the propeller, and has a gear ratio of 7:1.

3.3 Batteries

The batteries are of the shelf Torqeedo 915Wh batteries developed for use with their electrical outboard motors [12]. The Otter can carry four of these batteries, for a total capacity of 3660Wh, or 124Ah.

3.4 Signaling Light

On top of the Otter, there is mounted a Hella Marine NaviLED 360. In addition to making the Otter more visible in the water, it is also used to signal vehicle state by toggling state in different patterns. This operation is enabled by a relay on the Strato Pi CAN board, which will be described in Section 3.8.2.

3.5 Communication

AirMax wireless equipment from Ubiquiti provides communication for the Otter and land station. On the Otter, a Ubiquiti Bullet AC IP67 has been used, because of its IP rating. On the land station, two access points are available. One is a 120° sector antenna connected to a Ubiquiti Rocket AC. This provides wide coverage at short distances. From experience, around 600-700m can be expected for a solid connection. The other antenna is a Ubiquiti PowerBeam Gen2 dish antenna/access point. This provides a more concentrated signal, and will thus be able to reach longer distances. This has not been used during this project.

Powering to the communications equipment is supplied by using passive 24V Power over Ethernet (PoE). To add power to an Ethernet cable, a PoE injector is placed between the computer and the wireless device. During this project, it was discovered that the Otter only supplied 12V PoE, which led to unstable operation of the Bullet AC. 24V was not available in the power distribution, so a 12V-24V converter was added, as shown in Figure 4.

Configuring the devices is done through a separate WiFi connection, since AirMax is not compatible with ordinary WiFi devices. As of this project, the land based station has been configured as an access point that the Otter connects to. The network topology may have to be changed when support for both the dish- and the sector-based access points are to be used as one logical network.

[4]Which is a strange claim, since $1HP = 735.75W$.

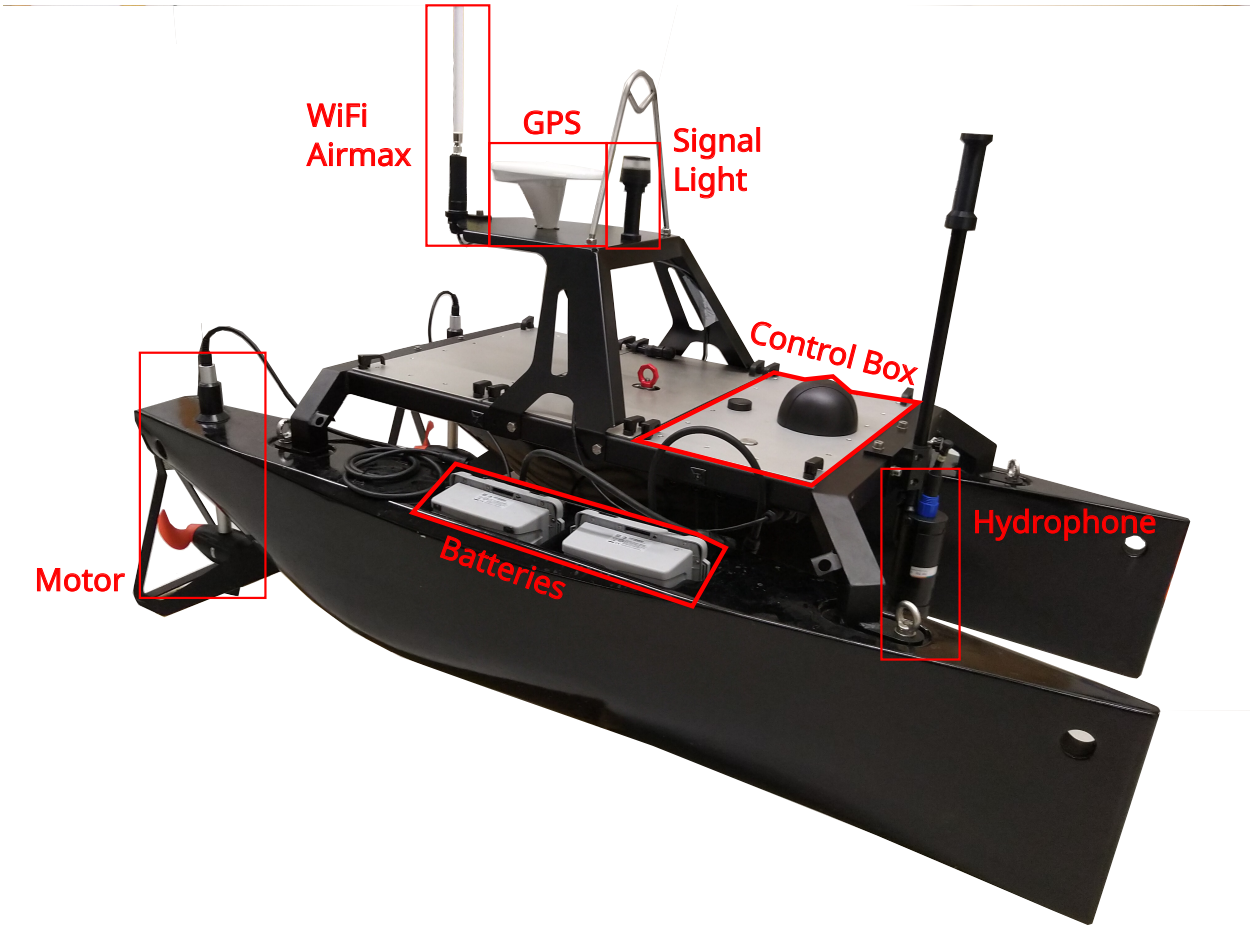

Figure 1: A picture of the Fish Otter with component labels.

3.6 Positioning

For positioning, the Otter relies upon a Hemisphere v104s. Using dual integrated GPS antennas, both GPS positioning and compass functions are achieved. It also comes with support for Space Based Augmentation Systems (SBAS) in addition to a single axis gyro and tilt sensors for x- and y-axis. This results in making a position accuracy of better than 1.0 m available 95% of the time.

Communication with the v104s is done through two full-duplex RS-232. For the Otter, only one of these connections is used, RS-232-A, while RS-232-B is unused. This connection goes through a RS-232-USB adapter, to the controlling computer. There is also a PPS output for time synchronization present, which use will be described in Section 3.8.4.

The format of the messages sent over RS232 is NMEA 0183, with the addition of some of Hemispheres custom messages.

3.7 Payload: Hydrophone

The hydrophone used on the Otter is a TBR700RT from Thelma Biotel. This provides detections of acoustic tag, as well as measurements taken in the hydrophone, fulfilling requirement 7. Examples of information gathered are shown in the second table[5] of both Figure 7 and Figure 8. As can be seen, the tag registrations contains a millisecond timestamp (the field named "millis"), meaning it fulfills requirement 8.

Depending on the software version ordered from the manufacturer, the hydrophone can detect tags transmitting on 63-77 kHz. Typically, lower frequencies are only suitable for larger tags, while higher frequencies are mostly used in smaller tags[6]. All tags send information using Differential Pulse Position Modulation (DPPM), but can use different protocols to give meaning to the time delays between pulses.

The communication with the controlling computer is through a RS-485 interface, delivering messages in a NMEA 0183 inspired format, described in Section 4.3.5. Bluetooth communication is also present, but not used for real-time operations like those the Otter target. Instead, it can be used to download lists of detections after a mission has been performed.

Power is supplied through the same wire as the RS-485 uses. It accepts a DC voltage of 5-12V, so 5V is used because it was available and unused by the other components in the Otter. Battery power is also possible, but not used in this project.

3.8 Control box

The control box is the only piece of hardware that has been modified in this project. This section describes what hardware has been used to control the Otter, as well as how it has been connected with regards to power and communication. For a quick overview, Figure 2 is given. A detailed description will be given in Section 3.8.5.

Note that in Figure 2, there is present both a 4G MIMO antenna and a GPS antenna on the control box. Neither of these are used in this project. The GPS antenna due to design changes, and the 4G antenna because the available hardware, a Huawei USB modem, was deemed to lack the physical robustness needed to withstand the rough conditions the Otter may be subject to. Alternative solutions were not properly explored, due to time constraints.

3.8.1 The controlling computer

To control the Otter, an ARM based single board computer (SBC) will be used. This computer should have the ability to support communication with the other components of the Fish Otter, as described in requirement 1. The Raspberry Pi series of OBCs fits the bill, with a 40-pin GPIO header to allow expansion, 4 USB ports and built in Ethernet support. Over the years, multiple models have been released, bringing new hardware and performance increases. Originally, the Fish Otter was set to use the RPI3 model, but

[5] The first table shows information specific to IMC messages, described in Section 4.2.1.
[6] Because larger transceivers are needed for lower frequencies.

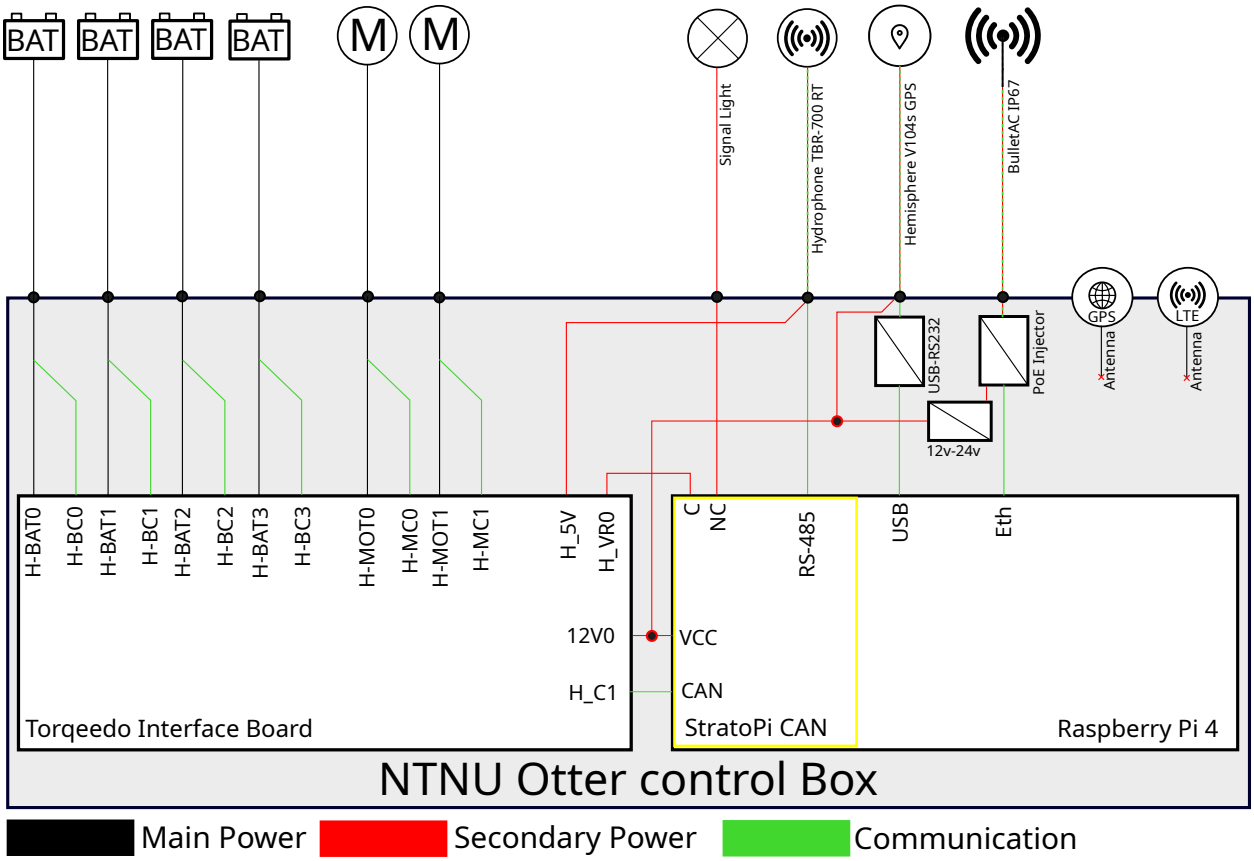

Figure 2: A diagram giving an overview of how the control box is connected, and connections to the outside.

during this project, it was decided to upgrade it to the newer RPI4 with 4GB RAM. The reasoning behind this decision, is that feature plans for the Fish Otter may require more computing power and memory, and the RPI4 also has an improved Ethernet adapter[7]. The RPI4 has the same physical dimensions as the RPI3, and the same GPIO output pins, making it a drop in replacement.

The 40-pin GPIO header breaks out power, I2C, SPI and UART in addition to digital inputs and outputs. These can be used to expand the possible interfaces supported.

3.8.2 Strato Pi CAN

The Strato Pi CAN is an expansion board made by Sfera Labs for the Raspberry Pi. It connects to the GPIO pins of the RPI, both powering it, and extending its hardware features. The Otter uses the power supply, relay, the CANbus support, the RS485 support, and the watchdog.

Some of the functions on the Strato Pi CAN are implemented on a MCU, and the watchdog is one of them. For the watchdog to work, it has to be set up properly before use. This is done by a serial interface from GPIO13 and GPIO19 on the RPI, but there is no hardware UART support available at those GPIOs. Using a software based serial implementation is possible, but for the Otter, a one-time configuration through an external USB-serial adapter provided the desired function.

During configuration, a problem was detected. One variable was set to 60 second by default, and no mechanisms for changing it was available from manufacturer. This led to the watchdog always needing at least 60 seconds before restarting the RPI, a delay deemed to be unacceptable for the Otter. After contacting Sfera Labs about the problem, they decided to add this feature, and sent us a new version of the firmware for the MCU. The specific model used is a PIC18F13K22, and can be reprogrammed using the MPLAB PICKit4. After soldering some header pins to the PCB and connecting the PICKit4, the custom hardware was loaded. After reconfiguring the watchdog with the new options, restart within 5 seconds after timeout was achieved.

3.8.3 Torqeedo Interface Board

The Torqeedo Interface Board (TIB) acts as a gateway between six Torqeedo RS-485 interfaces and a CAN bus interface. Four of the RS-485 connections come from batteries, which it also manages power from. The power is distributed to two motors, and 8 power outputs for use by the other components of the Otter.

There exist no comprehensive documentation on how to communicate with the TIB, only a C-header file and a sparse document giving an overview. To figure out the bit rate used on the CAN interface, an oscilloscope was used. Measuring the length of one symbol, it was found to be 125 Kbit/sec. Establishing a connection, 20-25 messages was received every second with telemetry about thruster, batteries and power states.

Other undocumented details of the workings of the TIC are given in the list below for future reference:

- To reach full thruster actuation, more than two batteries are needed. For two batteries, the thrusters are limited to 600 RPM, while three batteries tops out at 1100 RPM.
- The thruster speed needs to be set every second, or they will be stop.
- The revolution speed received, does not contain information about direction.

3.8.4 Time Synchronization

To fulfill requirement 8, the plan was originally to use the NTNU developed SLIM PCB. This has its own GPS receiver for receiving time, and also a RS-485 interface for the hydrophone. But during work with the navigational GPS system, it was noticed that the Hemisphere V104s also has a pulse-per-second (PPS) output that reaches the control box. This could be used by the RPI to synchronize the hydrophone clock, and so the reduction in complexity by removing a hardware component would be achieved.

[7]The previous RPI models used a USB-Ethernet chip connected to the SoC through a USB hub. The newer RPI4 has a dedicated Ethernet controller connected directly to the SoC.

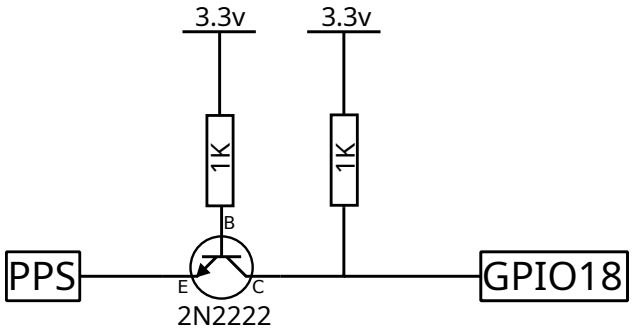

Figure 3: The logic level-shifting circuit from the GPS PPS signal to the GPIO pin of the RPI.

The hardware connections to the control box was already present, but it had to be internally wired. To get the PPS signal to the RPI, two solutions was proposed; using a GPIO pin, or using the Data Carrier Detect (DCD) of a serial to USB adapter. Both have their pros and cons. The GPIO pins are operating on another logic level than the V104s GPS, so a level shifter would have to be added to the system. In addition, the GPIO header was already physically connected to the Strato Pi CAN interface PCB. The RS-232 interface from the V104s is already connected through an RS-232 to USB adapter, so no level-shifting needed. After researching this option, it was noticed that the DCD signal was not available from the installed adapter, so we would have to add another serial-USB adapter. This made us choose to use the GPIO pins.

For a level-shifting circuit, many options are available. Since we only needed the shifter to work in one direction, a simple NPN BJT transistor was used in combination with two resistors. The circuit used, is shown in Figure 3. The values of the resistors are set to $1K\Omega$, and the BJT is a 2N2222 in a TO-92 package. This results in a signal level reduction from $0-3.5V$ to $0-2.6V$, which is tolerated by the RPI GPIO pins.

3.8.5 Connections

Figure 4 gives a detailed view of how the components of the Otter are connected to and in the control box. For the motors and batteries, the detail level is reduced, because these connections are already present from the system delivered by Maritime Robotics. The connection line colors for most of the system represent the actual colors used in the control box, with the exception of standard data interfaces like RS-232, RS-485 and Ethernet. For PPS, blue was chosen because the real wire is white, and would not be visible in the figure.

A picture of the actual control box, is shown in Figure 17. Note that the SLIM is still present, but is not connected or being used. Also note that there is a Molex connector between the control box and the water resistant connectors on the outside of the control box. The connections between the inside dry area, and the outside wet area is shown in Figure 16.

3.9 Power Usage

The Otter is battery powered, constraining the duration of missions it can perform. A power budget is given in table 1, showing the worst-case power usage for each component. From Section 3.3, we have that the total capacity of the four batteries is $3660Wh$. The worst-case duration of battery operation then becomes $3660Wh/826.762W = 4.427h$. As can be seen in the table, the motors dominate power consumption, and actuation should therefore be limited for long duration missions.

Note that the power usage of the hydrophone is based on the TBR700 battery usage of 6 mA at 3.6 V, and the TBR700RT may use a small amount more.

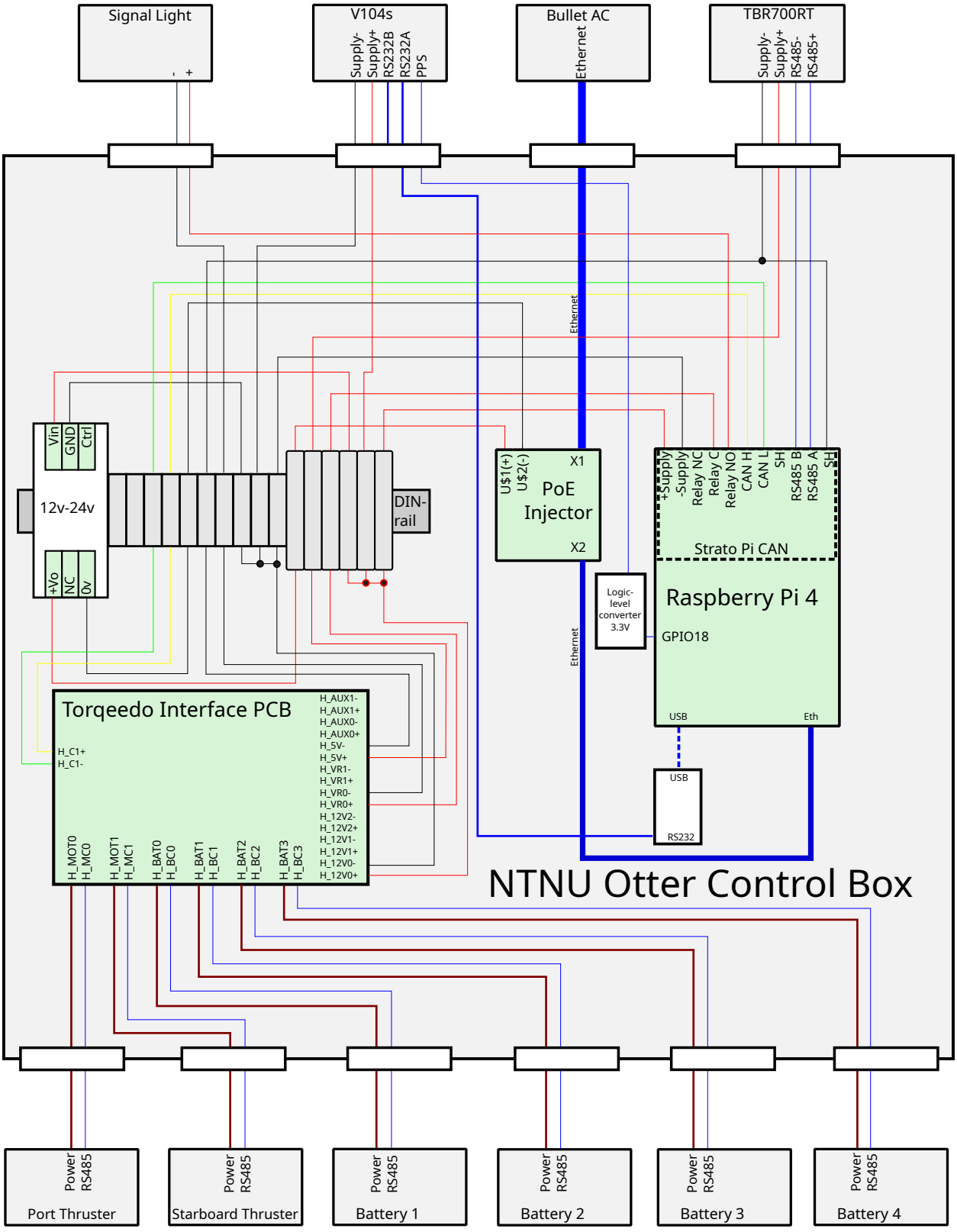

Figure 4: A diagram giving detailed insights to connections made in control box.

Unit	**Voltage[V]**	**Current[A]**	**Power[W]**
CPU	12	1.4	16.8
GPS	12	0.165	1.98
WiFi	24	0.3	7
SignalLED	12	0.08	0.96
Hydrophone	5	0.0044	0.022
Motors Full	24-32	Varies	2x400
Total worst-case power consumption:			826.762

Table 1: The maximum power requirements used by the hardware components in the Otter.

4 Software Design

This section describes the process from choosing and installing an OS in Section 4.1, to choosing a toolchain for vehicle control and operation in Section 4.2. Designing the unified navigational environment that is running on the Otter is done in Section 4.3, while the customization of the console application is described in Section 4.4. To finish, the configuration of the OS and application for PPS support is described in Section 4.5.

4.1 Raspian

Raspbian is a Linux based operating system optimized for use on the Raspberry Pi hardware. Due to the great amount of community support available, and previous experience, it was chosen as the operating system for the Otter.

Raspian comes in two official flavours, one with a full desktop environment, and one meant for headless[8] use. The Otter is controlled over a network connection, and having a desktop would only waste valuable resources. That is why the Otter runs the headless version called Raspian Lite.

Setting up Raspian for the Otter is a multi-stepped operation, beginning with flashing the microSD card with the OS, and enabling ssh. After this, the first boot can commence, with the standard changing of passwords and initial updates done. For the Otter, CAN bus support has be enabled, and this is done by enabling SPI and adding these options to */boot/config.txt*:

```
dtoverlay=mcp2515-can0,oscillator=16000000,interrupt=25
dtoverlay=spi-bcm2835-overlay
```

These lines tells the kernel to use drivers for the MCP2515 CAN controller running with a 16MHz oscillator and delivering interrupts on GPIO pin 25. This reflects the hardware on the Strato PI CAN daughterboard.

To automatically set up a CAN device when booting, these lines also have to be added to */etc/network/interfaces*:

```
auto can0
iface can0 inet manual
pre-up /sbin/ip link set can0 type can bitrate 125000
up /sbin/ifconfig can0 up
down /sbin/ifconfig can0 down
```

In order to start the unified navigational environment at boot, commands are added to */etc/rc.local*. This will work for any bash command, as long as it is ended with '&'. This allows the script to finish with commands still running in the background.

The final change made to Raspian is described in 4.5, where the kernel has to be reconfigured and compiled.

4.2 The LSTS Toolchain

The LSTS[9] toolchain is a software toolchain for developing networked vehicle systems. It aims at creating a modular system for heterogeneous systems of vehicles, in order to control them by much of the same software. The Otter is an autonomous surface vehicle (ASV), but it also supports autonomous underwater vehicles (AUV), unmanned aerial vehicles (UAV), remotly operated vehicles (ROV) and more [9].

An important aspect in why it was chosen, was that it is open source. This makes it easier to know how the software actually works, and also gives anybody a chance to fix faults, and implement new features.

[8] No screen, keyboard or mouse used, only remote access.

[9] "Laboratório de Sistemas e Tecnologia Subaquática", or Underwater Systems and Technology Laboratory at the University in Porto.

The three main components are Neptus, IMC and DUNE. Neptus is the operators console, IMC is the communication protocol, and DUNE is the software controlling the vehicles. The next three sections give more details of these components.

4.2.1 IMC

Inter-Module Communications is the message protocol used by the different LSTS tools for communications. It is a transport-agnostic protocol, meaning that it can be used over Ethernet, as well as acoustic transmission, satellite based transmission, cellular, and between threads running in an executable.

The protocol is message oriented, meaning that every concept of the language is split into one or more types of messages. An example of this is information about a battery, which is split into multiple messages, one for charge, one for voltage, one for current and so on. The same also applies to control signals between threads, so the high level path controllers communicate with the lower level course, speed and depth controllers with IMC messages, who also send messages to the actuator controllers.

The IMC protocol also contains mechanisms for broadcasting the existence of a device supporting IMC, as well discovering of other devices. During this process, an Announce message is sent, containing device state and its capabilities. For the Otter, this means that as long as both the console and the vehicle is on the same network, a connection will be established automatically.

4.2.2 DUNE

DUNE is the software that runs on the vehicle. It's written in C++, and is compiled with the help of CMake. This enables easier cross-compiling and adding of additional source files.

Modularity is a main design philosophy in DUNE, and it functions by having all modules, called tasks, in their own thread, running as a single process. What tasks are activated when running an instance of DUNE is decided by configuration files. These files also includes parameters configuring each task, making them reusable in a wide array of use-cases. This is a key to why so many tasks can be reused for different vehicle types and configurations.

The configuration files also have support for different profiles, deciding how the vehicle is used right now. By default, the profiles "Hardware" and "Simulation" exists, but creating custom ones for use in development or hardware-in-the-loop modes is also possible. For instance, on the Otter, one custom configuration is "StratoPi", which was used under development of the CANbus and Torqeedo Interface PCB implementations.

The tasks of DUNE use IMC to do all communication, both between themselves, and to the controlling console. For a DUNE task, there is no difference between communicating with another task, another vehicle, or the console giving commands. Multiple examples of how tasks communicate with each other are given in Section 4.3.6.

4.2.3 Neptus

Neptus is the user interface in the LSTS vehicle system, giving a console for operators to visually plan, simulate, monitor and review missions. It is developed in Java, and compiled with the Apache Ant system. This makes it possible to run on both Linux and Windows (and possibly others). A screenshot of the console is shown in Figure 5.

The Neptus console can be configured with different plug-ins being activated or deactivated, much like the tasks of DUNE. This can be map layers, separate windows or panels in the main window.

An important feature is that multiple, heterogeneous vehicles can be controlled by the same operator in Neptus. This will become important later on in the Otter development, when multiple Otters are to controlled on a mission together.

For reviewing and analyzing data captured in a mission, Neptus also has the ability to read logs from DUNE. Multiple visualizations are available, and custom ones can be developed where they are needed. Export to

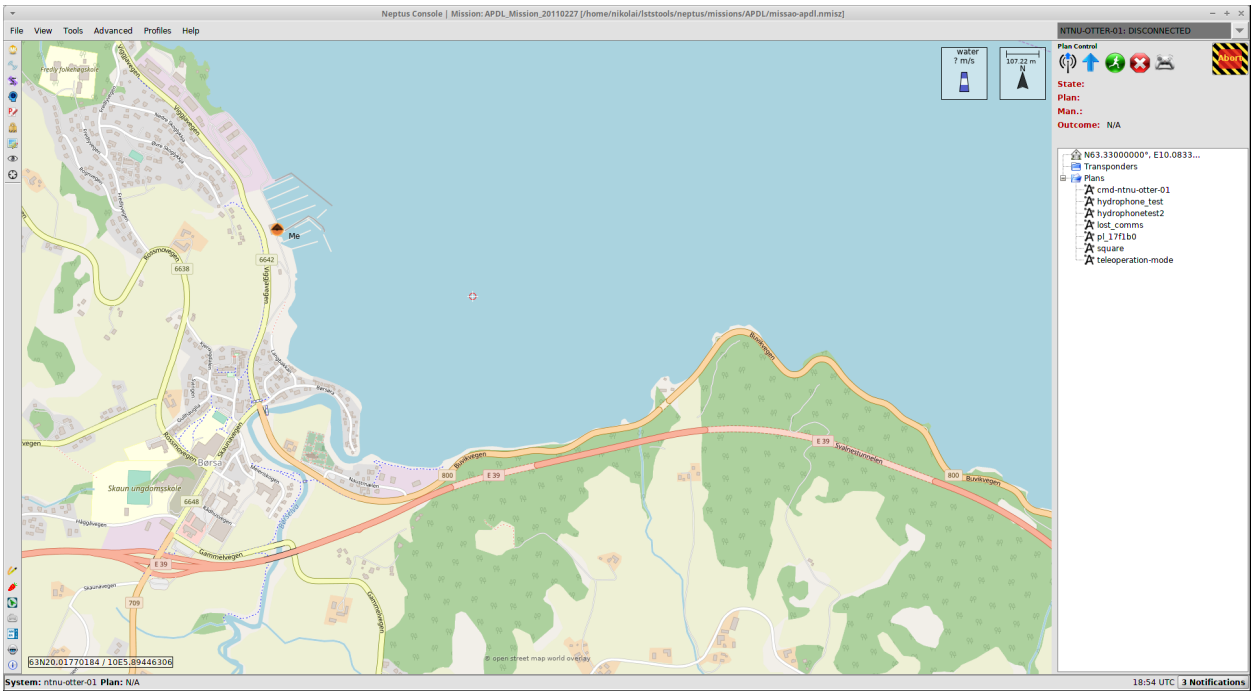

Figure 5: The console window of Neptus.

a wide array of formats are also available, like MATLAB or comma separated files supported by Microsoft Excel.

4.3 DUNE Integration

Adding support for new vehicles in DUNE, is done by creating a configuration file customized for the new vehicle. If some new feature is needed, actual code has to be written in the form of new tasks. For the Fish Otter, three custom tasks was created, in addition to implementing CAN support in the DUNE core.

The new tasks will handle the Torqeedo interface card, the TBR700RT and the Strato Pi CAN watchdog. There was already tasks available for both the signaling light and navigation.

4.3.1 Compiling DUNE for the RPI

Compiling DUNE on the Raspberry Pi takes a long time. For the RPI3 first used, it took about 40 minutes, while for the RPI4, it took 15 minutes. Therefore a cross-compiler seemed an essential tool to help speed up the development of the custom software used by the Otter. At first, the official Raspian cross-compiler was tested, which worked for simple "Hello World!" executable. But while trying to compile DUNE, a problem occurred. DUNE needs to be compiled using a newer version of GCC, to support some of the more recent C++ developments. Therefore, the cross-compiler made for GLUED[10] was used instead.

The compiled executable and needed configuration files are packed in a tarball, and transferred to the RPI with the secure copy (SCP) Linux command. On the RPI, after unpacking, the executable can be run as if it was locally compiled.

[10]GLUED is a minimal Linux distribution targeted at embedded systems, developed as a part of the LSTS toolchain. The Otter does not use it, because of it possibly making 4G communication harder at a future point.

4.3.2 The Strato Pi Wachdog Task

The task to interact with the Strato Pi watchdog functionality is a very simple DUNE task. There is no IMC messages[11], only a periodical DUNE task that control the GPIO pins to enable the watchdog and toggle the heartbeat pin. For completeness, it also monitors the GPIO signaling watchdog timeout and logs if it occurs.

The code for this task is available in appendix B.4.

4.3.3 CAN support in DUNE

Communication with the Torqeedo Interface PCB is done using the CAN bus. For convenience, DUNE comes with built in support for many hardware interfaces, like Serial, GPIO and I2C. The CAN bus was not available, so this had to be implemented before communication with the interface PCB could begin.

The CAN controller used on the Strato Pi CAN, is a MCP2515, and is connected to the RPI through SPI on the GPIO header. Luckily, the MCP2515 is one of the supported controllers by the SocketCAN drivers, which is a part of the Linux kernel. This set of drivers extends the POSIX socket API, which makes using CAN very similar to using IP sockets in C.

Hardware interaction in DUNE is implemented as C++ classes. To start using the newly developed CAN support in a DUNE task, a call to the constructor has to be made. This is very similar to using peripherals on an Arduino. Creating a new CANbus hardware instance in DUNE looks like this:

```
m_can = new Hardware::SocketCAN(can0, SocketCAN::CAN_BASIC_EFF);
```

The two arguments are device and CAN message format.

IO operations in DUNE has a standardized interface, defined by the class "IO::Handle" located in "src/DUNE/IO/handle.hpp" of the DUNE repository. The CAN implementation inherits from this class, and implements the virtual functions so that they can be run. An example of reading and writing from the CAN bus is given below:

```
// CAN buffer used for storing and sending messages
char m_can_bfr[9];
// Read to buffer
m_can->read(m_can_bfr, sizeof(m_can_bfr));
// Write from buffer
m_can_bfr[0] = 't';
m_can_bfr[1] = 'e';
m_can_bfr[2] = 's';
m_can_bfr[3] = 't';
m_can->write(m_can_bfr, 4);
```

This will work, but the read function is missing a very important feature, a timeout. This would mean that execution would halt after a read was performed, unless there actually was a message on the bus. To solve this, there is another DUNE class called "Poll" located in the same folder as "IO::Handle". To get this to work, all the CANbus class has to do, is implement the virtual function "doGetNative(void)" from "IO::Handle". Now, reading with a timeout looks like this:

```
if (Poll::poll(*m_can, 0.01)) { // m_can is instance of CAN class in DUNE
  m_can->read(m_can_bfr, sizeof(m_can_bfr));
}
```

The CANbus messages each have an id, and this implementation also needs a way to access these. In development, returning it along with the message like "msgid#msg" seemed smart, but in use, separate functions was more practical. The "IO::Handle" interface did not define this operation, so two new functions

[11] Technically, there is, but only those used for task state monitoring by DUNE.

was made; “getRXID” to see the id of the previously read message, and “setTXID” to set the id of the next message to be sent.

The code is available in appendix B.7.

4.3.4 The Torqeedo Interface PCB Task

To enable communication between the IMC messages in DUNE, and the CAN bus protocol used by the Torqeedo Interface PCB (TIP), a new task has been created for use in the Otter. This task controls the thrusters and the power channels, and it also receives telemetry related to these and the batteries.

The motors of the TIP must be set at least once per second, so the task needs to run periodically. DUNE has a task type that can be used for this. But, since the task also has to read data about 25 times a second[12], a divider was used. So the task can run 40 times a second (specified by a task parameter), but with a divider of 10 (specified by a task parameter), it only writes to the motors $40/10 = 4$ times a second. When not sending motor messages the task is either sending power channel instructions or reading messages sent from TIP. The main flow of the task is illustrated in Figure 6. In addition to the main flow, it also responds to IMC::SetThrusterActuation and IMC::PowerChannelControl messages[13]. These set internal variables, which are sent to the CAN bus in the main flow.

The code for the torqeedo DUNE task is available in appendix B.5.

4.3.5 The TBR700RT Task

The messages protocol used by the TBR700RT is inspired by the NMEA 0183 standard used for receiving information by the GPS. Two types of messages are sent from the TBR; internal sensor readings and tag detections. Examples of these are:

```
$001234,1527073500,TBRSensor,323,10,16,69,2469\r
$001234,1527073474,945,S256,241,150,32,69,2467\r
```

Because of this similarity with NMEA 0183, the task for receiving GPS messages was used as a template for the TBR task. It uses a "Reader" thread that reads the UART until ’\r’ appears. Then, it sends a "IMC::DevDataText" to the main task. The main task then checks for the message start sign ’$’, separates the values by the commas, and then makes an IMC message containing the received data.

To send the information on the IMC bus, two custom message types have been created; IMC::TBRSensor and IMC::TBRFishTag. Creating new message types in both DUNE and Neptus, is done by editing a XML-file. Examples of the messages sent from DUNE to Neptus is shown in Figure 7 and Figure 8.

As mentioned in Section 3.8.4, the RPI is responsible for clock synchronization on the TBR700RT. This is done by sending two types of messages on the RS485 interface;

Message example	**Valid response**	**Description**
(+)	ack01	Millisecond sync
(+)XXXXXXXXXL	ack01ack02	Millisecond and UTC UNIX timestamp sync

When the millisecond sync message arrives, the Unix timestamp is rounded to the closest 10. In the Unix timestamp sync message, the 9 ’X’ positions is the timestamp, with an implied ’0’ at the end. The ’L’ is the Luhns verification number for the 9 previous digits.

The main flow of the task, is shown in Figure 9. Not all functions relating to time sync has been implemented yet in the code, due to time constraints. Also, some crucial discoveries about communicating with the TBR700RT were made; at given intervals, it enters a safe section of code where it does processing and turns off interrupts. Thus, when sending a string of characters, like the UNIX timestamp, a 1ms delay between each character should be added, to ensure that the TBR700RT is able to read the timestamp. This has nothing with the baud rate it uses, which is 19200 baud.

[12]The TIP sends 20-25 messages/second.

[13]Consuming IMC messages are done before each time the timer runs the task.

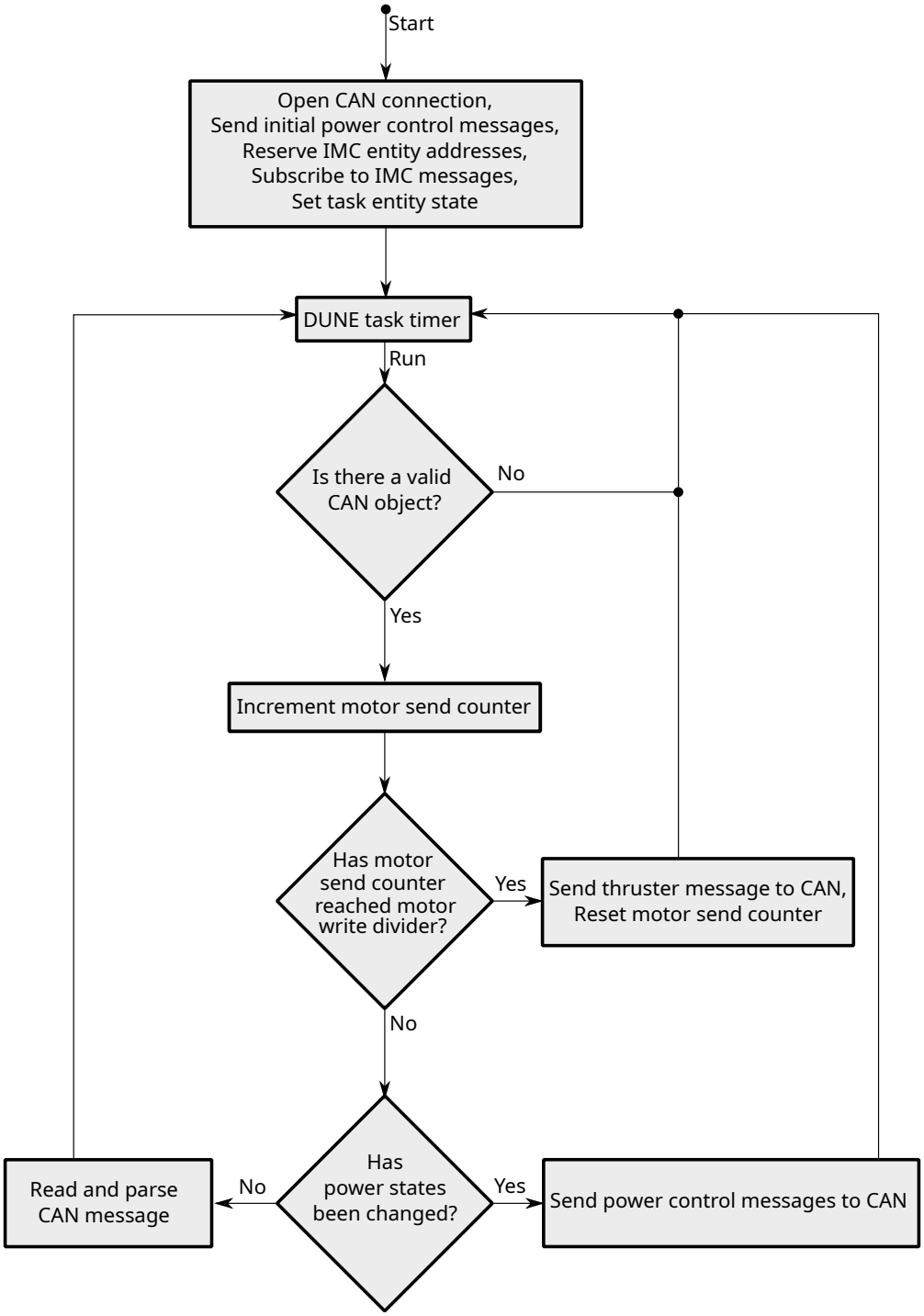

Figure 6: Torqeedo main task flow chart.

TBRSensor

Header	8 fields
sync	65108 [0xFE54]
mgid	2008
size	17 byte
timestamp	2019-Nov-07 12:00:05.544 UTC
src	11360 [0x2C60]
src_ent	49
dst	65535 [0xFFFF]
dst_ent	255

TBRSensor	7 fields
serial_no	734
unix_timestamp	1573128000
temperature	4.6 °c
avg_noise_level	15
peak_noise_level	31
recv_listen_freq	67
recv_mem_addr	49708

Figure 7: Screen capture of TBR sensor IMC message in Neptus.

TBRFishTag

Header	8 fields
sync	65108 [0xFE54]
mgid	2007
size	21 byte
timestamp	2019-Oct-21 13:05:24.694 UTC
src	11360 [0x2C60]
src_ent	35
dst	65535 [0xFFFF]
dst_ent	255

TBRFishTag	9 fields
serial_no	6
unix_timestamp	1551087594
millis	285 ms
trans_protocol	S256
trans_id	15
trans_data	36
SNR	31 db
trans_freq	69
recv_mem_addr	438

Figure 8: Screen capture of TBR tag detection IMC message in Neptus.

The code for this task is available in appendix B.6. As noted, some of the time synchronization functionality is not made periodical yet. Figure 9 gives a better picture of how it will work when feature complete.

4.3.6 The DUNE Configuration File

The configuration file made for the Otter, tells DUNE which tasks to run. The file[14] can be found in appendix B.3.

In order to get the GPS to work, two tasks is needed; *Sensors.GPS* and *Navigation.General.GPSNavigation*. Sensors.GPS takes the NMEA 0183 messages received from a UART port, and makes the information available through the IMC messages IMC::GpsFix, IMC::EulerAngles and IMC::AngularVelocity. These messages are subscribed to by *Navigation.General.GPSNavigation*, which checks the validity of the GPSFix, and then uses it to send a IMC::EstimatedState messages for further use.

The course and speed control is performed by the task *Control.ASV.HeadingAndSpeed*[15]. This reads the vehicles IMC::EstimatedState, and through the use of PID-controllers set the level of actuation for each thruster. Through sending IMC::SetThrusterActuation, the actuation levels reaches the *Actuators.Torqeedo* task described in Section 4.3.4, which commands the actual hardware.

The course and speed is set by the a path controller. For the Otter, *Control.Path.ILOS* task implements the integral line of sight (ILOS) guidance law described in [2]. The goal of using integral action, is to avoid disturbances from the environment like current or waves.

The path is conveyed through IMC::DesiredPath messages sent from one of the many maneuver control tasks. These are included from the "etc/common/maneuvers.ini" file from the official DUNE repository. As described in Section 4.2.2, maneuvers are part of plans. In DUNE, this is controlled by *Plan.Engine*, which selects what maneuver should be activated through IMC::ManeuverControlState messages. This is also the receiver off IMC::PlanControl messages.

The plans are typically sent to the vehicle from a Neptus console. This communication can be over multiple protocols, but for the Otter, Ethernet is used. The task *Transports.UDP* implements this functionality, and can also make IMC messages from the vehicle available to the console. An example of this is the IMC::EstimatedState, used by Neptus for showing the Otters position on a map.

For communication with Neptus, a unique identifier for the vehicle is needed. This is an IMC address, and should normally be stored in "etc/common/imc-addresses.ini". For the Otter, it is stored at the top of the configuration file for now, while the Otters have no official IMC addresses. The parameters are shown below:

```
[IMC Addresses]
ntnu-otter-01                           = 0x2c60

[General]
Vehicle                                 = ntnu-otter-01
```

When multiple Otters are to be used together, each has to be given its own unique IMC address.

Controlling the signal light is done by switching a GPIO on and off. The task *UserInterfaces.LEDs* can do this, and also has support for changing pattern based on both IMC::VehicleState, and also the states of important entities from IMC::EntityState. This completes system requirements 4.

Logging is an important feature, as it allows for reviewing previous missions. The task *Transports.Logging* reads IMC messages, and stores them in a file format supported for use in Neptus. The task needs to be told what IMC messages are to be stored, and this is done through a parameter. The task also automatically creates separate log-files for each plan.

[14] The file is best viewed in a editor with code highlights, like VSCode or Sublime Text. The appendix is meant more as a quick reference.

[15] Note that even though "Heading" is in the task name, and it is an available metric from the GPS, the value controlled is course over ground (COG). The difference is that the heading is pointing in the direction the vehicle is pointing, whereas the course is pointing in the direction of movement.

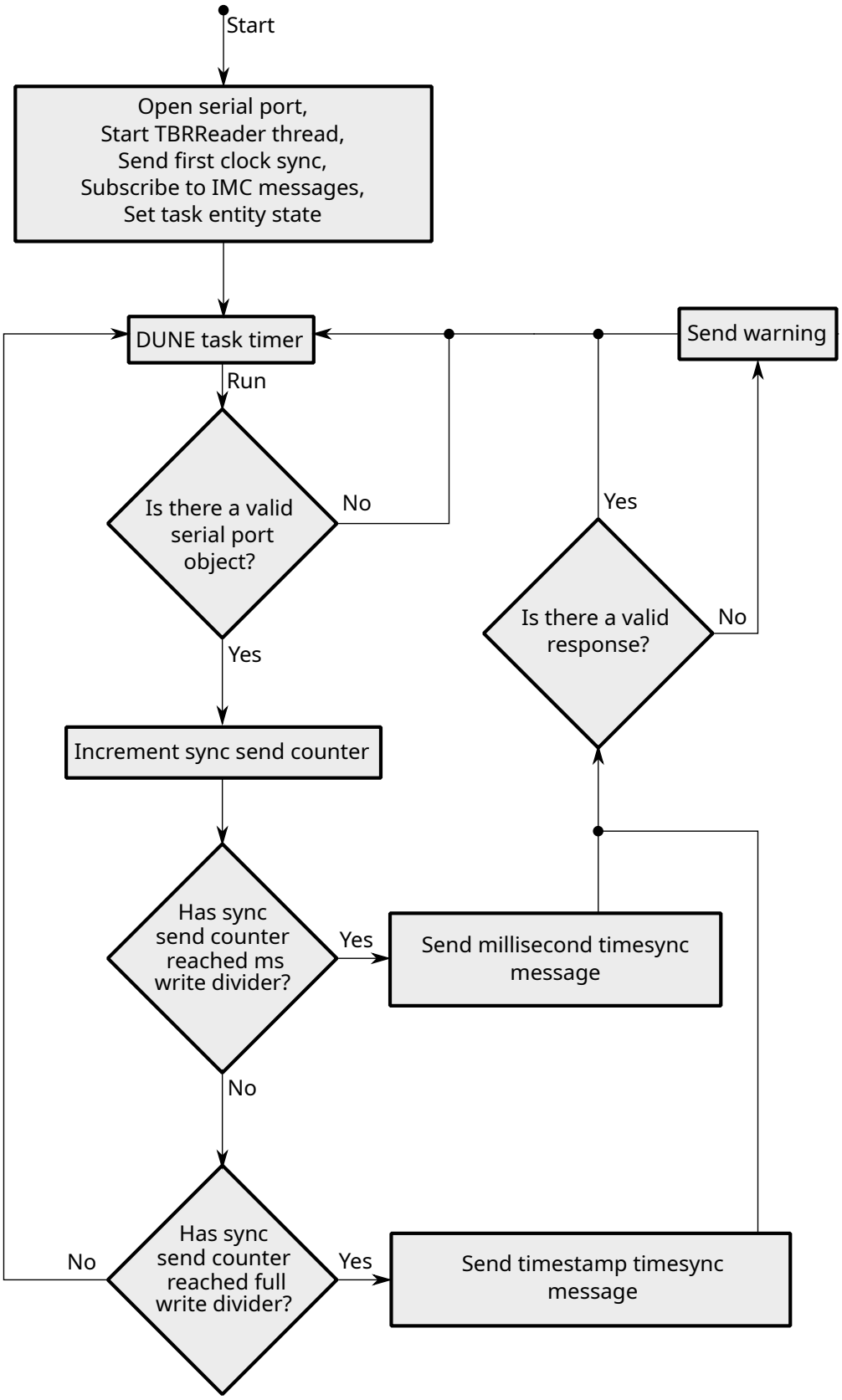

Figure 9: TBR700RT main task flow chart.

During development, replacing hardware components with simulations is possible. For the Otter, both thrusters and the GPS have been configured for use in the Simulation profile. A lot of the development was done with just a RPI and a StratoPi, so a special profile, named "StratoPi" has been created for this purpose. This makes it possible to use the control box without having to connect the real hardware.

Some of the task are crucial for the function of the vehicle, like the path controller. If the path controller has a failure, the vehicle will continue with the previously set course and speed. To avoid situations like this, the task *Monitor.Entities* is configured to monitor the state of important tasks. It will then restart the tasks, to reset them.

For completeness, there are also many supporting tasks present to enable features of the other tasks. Examples of this is a plan database task for the plan engine task, or the daemon making sure IMC messages reach the intended tasks. Not all are mentioned in this section, but a selection of the most important ones have been made. An overview can be gotten by looking at Figure 10, but for details, B.3 should be studied.

4.4 Neptus Integration

The Otter is not registered in the official IMC repository, which also has a list of known vehicles for use by Neptus. To fix this, a new .nvcl file has been added to the "vehicle-defs/" in the Neptus source. This file is in the XML format, and gives information about vehicles, where to find consoles for the vehicles, and also their IMC address. For the Otter, the IMC address 0x2c60 has been chosen, because of prior work done with the Otter. When asking to be included in the official IMC repository, the fish Otters should ask for addresses starting with 0x28, as the two first digits are set after vehicle type and owner. 0x28 means ASV and non-LSTS vehicle, while 0x2c means UAV and non-LSTS vehicle.

A new console configuration file also had to be created, which tells Neptus what modules are relevant when using the Otter. The console view of the Otter is shown in Figure 5.

To edit DUNE task parameters from Neptus, a special XML file has to be created in DUNE and moved to the "conf/params" folder in the Neptus source. This was a welcome feature when trying to tune the PID controllers for course and speed used in the Otter in Section 5.

Because of the custom IMC messages created for the TBR700RT task described in Section 4.3.5, the IMC definition used by Neptus has to be updated. This is done by compiling a java library based on the new IMC version. The library is then just copied and pasted to the Neptus source.

4.5 PPS software

To utilize the PPS signal received, first the Linux kernel support has to be enabled. Using the "make menuconfig", the options that have to be enabled are:

- CONFIG_PPS (“PPS support” in “make menuconfig”)
- CONFIG_NTP_PPS (“PPS kernel consumer support” in “make menuconfig”)

After compiling and running the new kernel, a PPS device has to be configured. In Raspian, this is done by adding the following to "/boot/config.txt":

```
dtoverlay=pps-gpio,gpiopin=[GPIOPIN]
```

For the configuration used by the Otter, [GPIOPIN] is replaced with 18. This makes the device "/dev/pps0" available for use further use. Synchronizing the system clock is done by the Linux system call adjtimex()[16]. In most cases, this is done by the NTP deamon, because the common use-case when using PPS in Linux seems to be to create a Stratum-1 time-server[17]. For the Otter, this is not the goal, and running an NTP server seems excessive.

[16]This function synchronizes the clock, but in a way that gives no sudden jump in time.
[17]The Stratum levels increases for each level away from the source clock, so here, Stratum-0 is the GPS time.

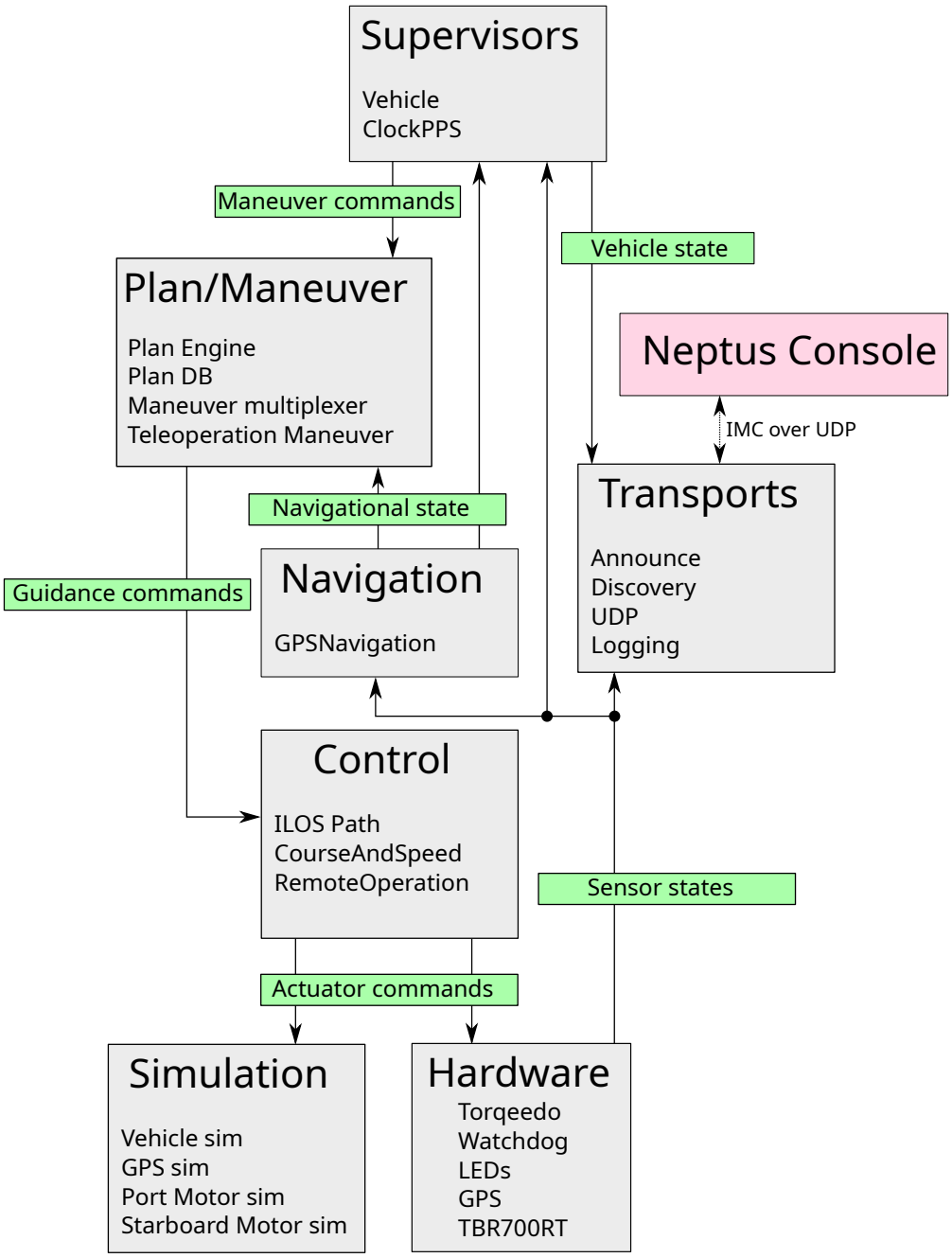

Figure 10: Overview of the most important tasks used in the DUNE configuration of the Otter, as well as an (incomplete) view of communications between them.

A PPS task exists for DUNE[18], and it supports time synchronization with adjtimex from a PPS device. The steps described above has to be performed in order for this task to run without error..

This has been confirmed to work with the PPS signal from GPS through level-shifter.

[18]The DUNE PPS Task has been removed from the LSTS repo. When asked, the LSTS developers seem to think it was removed by an error. For this project, it was acquired by looking through older git commits to the repository.

5 System Validation

The performance of the Otter has been validated in one land trial, and two sea trials. The goal of the trials, is to validate the complete system with multiple components working together. Each component was tested by itself before these trials were performed.

5.1 Dry Test at NTNU Gløshaugen 26/09/2019

The goal of this test was to see navigation, communication and motor control working together outdoors. The Otter was placed on a trolley and rolled outside on the tarmac. Connection between Neptus and DUNE was established through the Airmax connection, so after getting a GPS fix and verifying the position, some GoTo maneuvers were performed. The thrusters started and ran until the Otters position aligned with the point set in the GoTo maneuver. Rotating away from the desired heading resulted in differential thrust as expected. This was considered an success, and we began planning the first sea trial.

5.2 Sea Trial at Børsa 10/10/2019

The weather conditions was beautiful with blue skies and no waves on the water. This was the first time the Otter would be running in the actual water, so the goal was just to see general performance in water.

This was also the first time wireless equipment for land use would be tested outside the short distances of the lab. There was no problems in maintaining an connection with distances of 250-300 meters, but we did not push the limits.

The main systems of the Otter worked without any big problems, but the controllers need further tuning. Tele-operation[19] from land was used to drive the Otter to a safe spot, where maneuvers could be tested without danger to equipment or people. Both were confirmed to work, but the maneuvers suffered from badly tuned controllers.

On the hardware side, we noticed that the Otter is very direction bound when at speed. This gave it a very bad turning radius at around 10 meters. This tells us that the parameter "Maximum Heading Error to Thrust"[20] in the heading and speed controller should be set at some low value.

We also noted that the weight balance is off on the Otter, making the thrusters be to close to the surface. The result of this was that at higher RPMs, they sucked in air, significantly reducing performance. To solve this, a weight system should be made to mount towards the aft of the vehicle.

5.3 Sea Trial at Børsa 07/11/2019

The weather conditions was beautiful with blue skies and no waves on the water, but with an air temperature of $-8°C$, it was quite cold. This meant that some parts of the water was covered in ice, but lucky for the Otter, not the areas closest to the marina at Børsa.

The goal of this sea trial was to test:

- The range for wireless communication using the sector antenna and the Bullet AC.
- General Otter behavior
- The TBR700RT integration in DUNE and on the Otter
- If adding the new weights to the aft helped balance the vehicle.
- Thrusters that suddenly increased speed from 600RPM to 1100RPM at max thrust. This performance increase came as a result of adding a third battery. The controllers needs to be tuned again to account for this change in thruster dynamics.

[19] Using a PlayStation4 DualShock controller connected through Neptus to control the vehicle.
[20] This parameter makes the speed controller switch off if the heading error is greater than the set value.

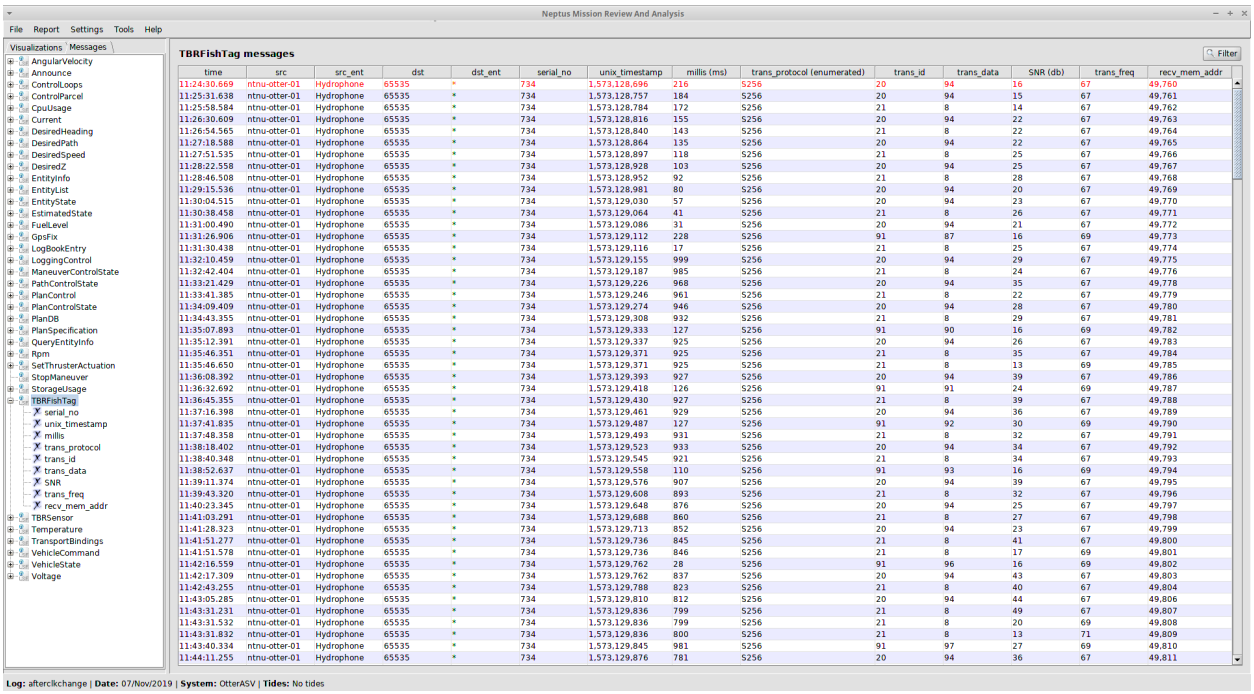

Figure 11: Neptus showing logged fish tag registrations from the second sea trial.

- In combination with the course "TTK15 - Oceanographic instrumentation and biotelemetry":
  - Check the effect of active thrusters when making fish tag detections.
  - Check reach of the tag registrations

The TBR700RT functionality worked, and a log of tag detections was obtained for further analysis, as is shown in Figure 11.

A map showing the the path the Otter traveled during this sea trial is shown in Figure 14. This is not a complete map, as an error in clock synchronization led to different timelines making data unavailable. This occurred due to testing a task named "Monitors.Clock", which sets the clock by the NMEA0183 clock messages from the GPS. This task should not be used, before further investigations are made into why this happened.

The new weights were placed in the frame connecting the starboard and port pontoons, and by visual inspection, prevented the thrusters from sucking air. Combined with the now increased speed of the thrusters, this resulted in better performance both in speed and turning rate.

For the experiment in TTK15, the OceanSonics icListen HF hydrophone was attached to the Otter. Comparing the sound power spectrum of a period with full thruster actuation and no thruster actuation, as shown in Figure 12, it becomes evident that the thrusters will affect the tag registrations when running. This can also be seen from the Signal-to-Noise ratios of tag detections registered by the TBR700RT shown in Figure 13. Registrations within 200m show that when the thrusters are actuated, the SNR is reduced by 10-15 dB. This is an argument for stopping the thrusters at regular intervals when the Otter using its hydrophone.

It was decided to change path controller to "PurePursuit" during this trial, because the parameters of ILOS had not yet been decided. Wrongly configuring the "PurePursuit" task led to it reaching an error state where no new course and speed messages where sent. This resulted in the Otter running away, and had to be rescued by another boat. Since the HeadingAndSpeed controller still was functioning, the Otter put up a fight. To turn it off, the battery connections had to be physically disconnected from the vehicle.

Before this incident, the range of the AirMax sector antenna was tested to work stable in a range of 600m,

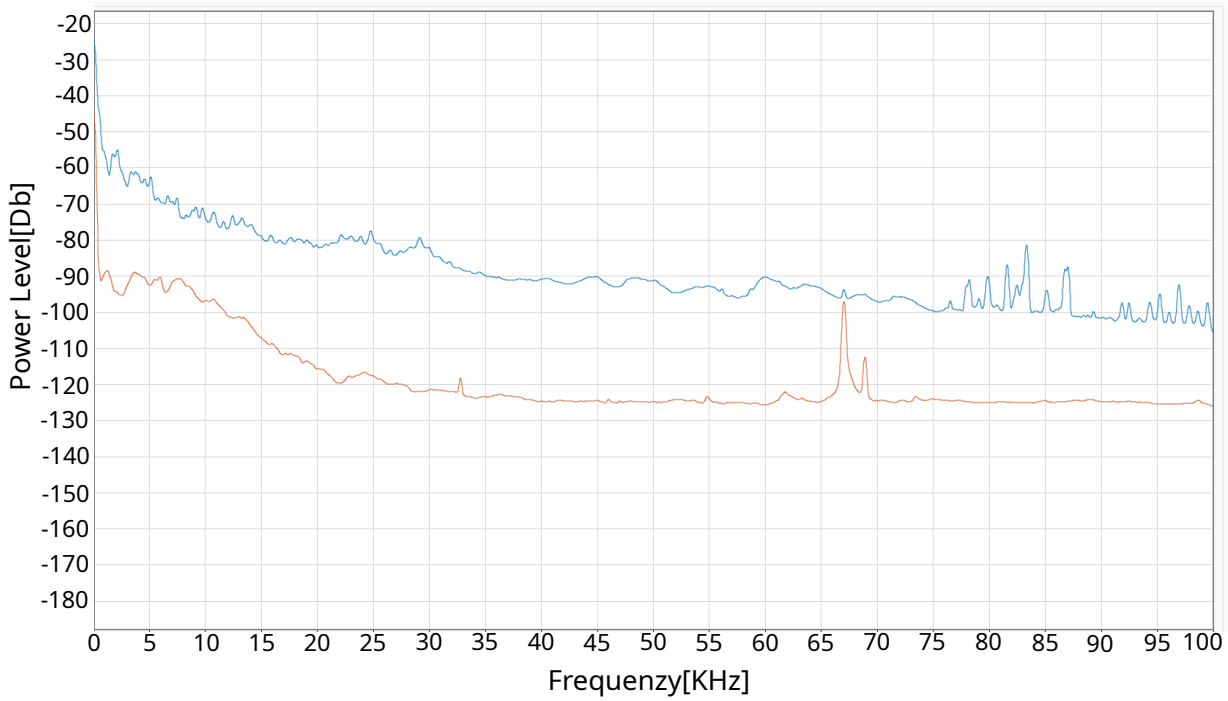

Figure 12: Screenshot of sound power spectrum of a period with full thruster (Blue) actuation and another period with no thruster actuation (Orange). The measurements were taken within 5 meters of each other, and in 1 minute intervals with 1 minute between the two measurements. The distance to the tag was below around 30 meters. The peaks shown at around 67KHz when the thrusters where off (orange), are from the acoustic tag.

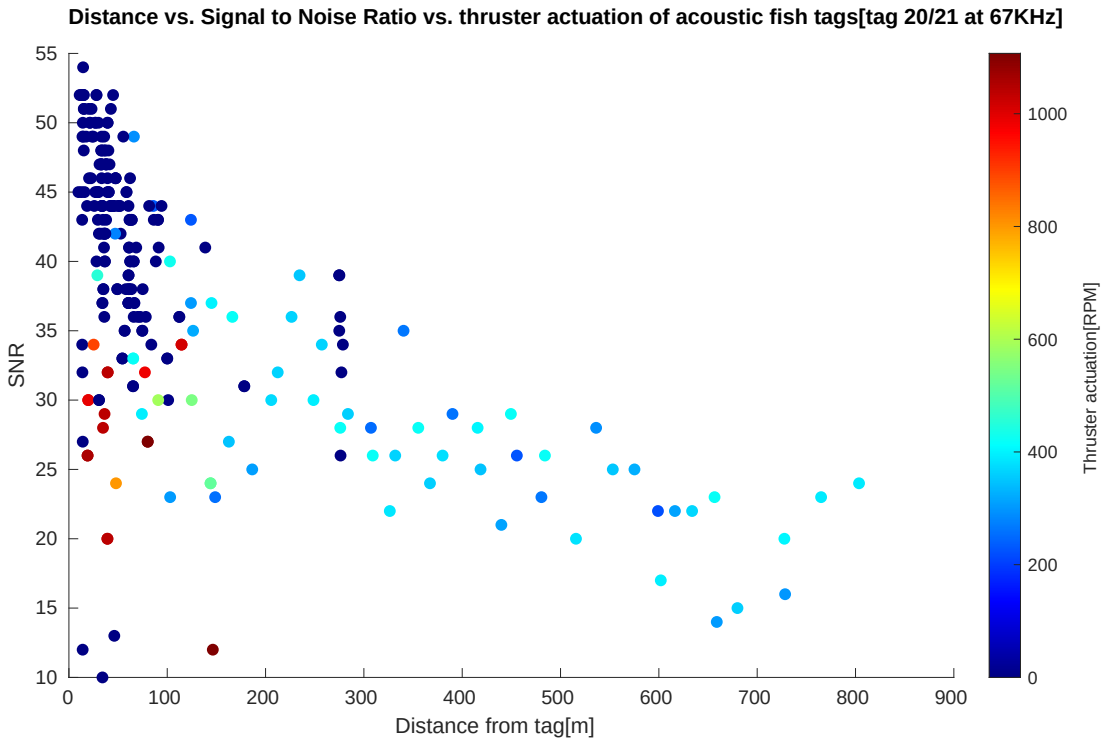

Figure 13: A scatterplot showing Signal-to-Noise ratio of tag registrations at different distances and thruster actuation levels. The data is was gathered by DUNE on the second sea trial, then exported by Neptus to MATLAB for further analysis.

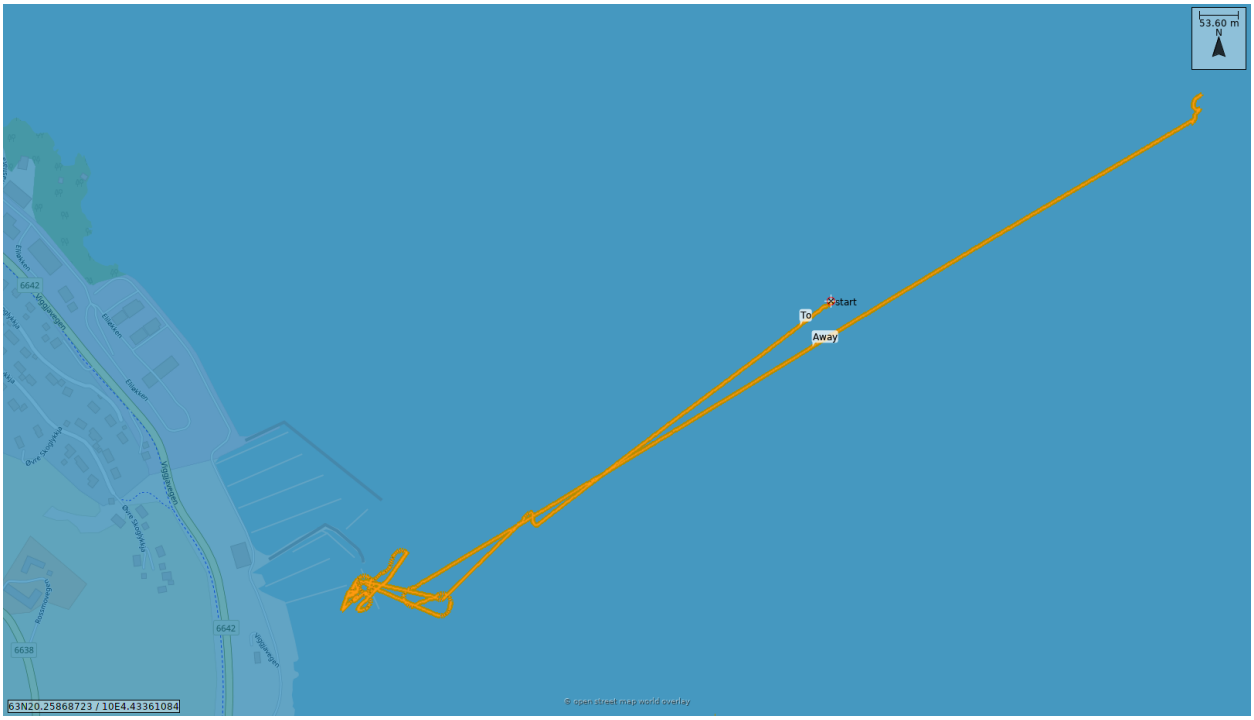

Figure 14: A screenshot of the path traveled by the Otter during the second sea trial.

and with occasional transfers at 1000m, but with no connection showing.

During tele-operation[21], a turning radius of about 7m was achieved while going forward full speed. This was a improvement compared to the previous sea trial, probably because of the increased rotational speed of the thrusters.

[21] Remote operation mode in Neptus.

6 Results and Discussion

The work described in this report has been guided by the system requirements from Section 2. To fulfill requirement 9, a Wiki documenting the Otter design has been made, and the process is described in Section 6.2. For the vehicle integration, the field trials described in Section 5 has been used as a basis for the system validation, and also the grounds for deciding if a requirement has been fulfilled.

6.1 System Integration

The process of doing a system integration for the fish Otter began with assessing what hardware was available, and where at ITK it was located. As work progressed, more hardware components was located, or in some instances bought, to complete the hardware design. The tactics behind this, was to ask for hardware long before it was needed, so it was available when needed for software development.

Using the RPI in combination with the Strato Pi CAN gave the system the abilities to fulfill requirement 1.

In parallel, the LSTS tools was explored, as it was already decided by the supervisor that it would be used as the basis for the software. This meant that to fulfill the requirements 5, 6, 10, 11 and 12 morphed into understanding, installing, configuring and learning to operate the LSTS toolchain. Installing DUNE was first performed on a x86_64 processor architecture, and a configuration file for the Otter was created. Included with DUNE came the configuration file for a vehicle named LSTS Caravela. This vehicle is also a catamaran differential thrust to move, so it was used as a template for the Otter.

Continuing with DUNE, a cross-compiler was found and used when developing for the ARM64 architecture used on the RPIs SoC, which is used in the fish Otter control box. Once this was done, Neptus was installed, and the communication between Neptus and DUNE was configured. This mostly meant making both applications aware of each other by configuring IMC addresses. Having a working Neptus configuration that communicates with the DUNE instance on the Otter fulfills the requirements 5, 6 and 11.

The hardware components was included in the design one by one. This started with finding and configuring tasks for the GPS and the signaling light. For the signaling light, the configuration was choosing patterns and what hardware protocol should be used, in our case GPIO. The GPS task configuration begins with telling it which serial interface the device is connected to, and then setting initial setup commands that should be sent. To find what commands were needed, the technical documentation for the Hemisphere v104s was studied. These commands made it possible to choose the NMEA0183 messages that were interesting for the Otter.

For the hydrophone, Torqeedo interface board(TIB) and the Strato Pi CAN watchdog, custom DUNE tasks has been developed. Running DUNE with these tasks in addition to the previously mentioned, aims to fulfill requirement 2, 3, 4 and 7. Based on the sea trials, this requirement has been demonstrated, with only small modifications to the hydrophone task and the tuning of the course, speed, and path controllers being needed in the future.

An important change was added to the configuration, as a result of the Otter running away in 5.3. After analysing the logs in Neptus, it was detected that the path controller had not been added to the task monitoring entities. Running simulations with this change did not solve the problem with "PurePursuit" ending in an error state, but it did automatically stop the vehicle from runnning away. Based on this incident, an emergency stop button should probably be added to the vehicle, as similar situations may occur at a later point in the development as well.

The final work that was done, was to get a PPS signal to the RPI, and to enable support in the Linux kernel. This should theoretically synchronize the system clock, which will be used by the DUNE task to synchronize the TBR700RT with millisecond accuracy. This should satisfy requirement 8, but has not been completely implemented, and not been verified by comparison to previously known-working systems.

Performing this system integration has brought the fish Otter project from the stage where hardware was available, but not completely finished, to the stage where a working Otter can be deployed for further work on tuning controllers and finalizing payload integration.

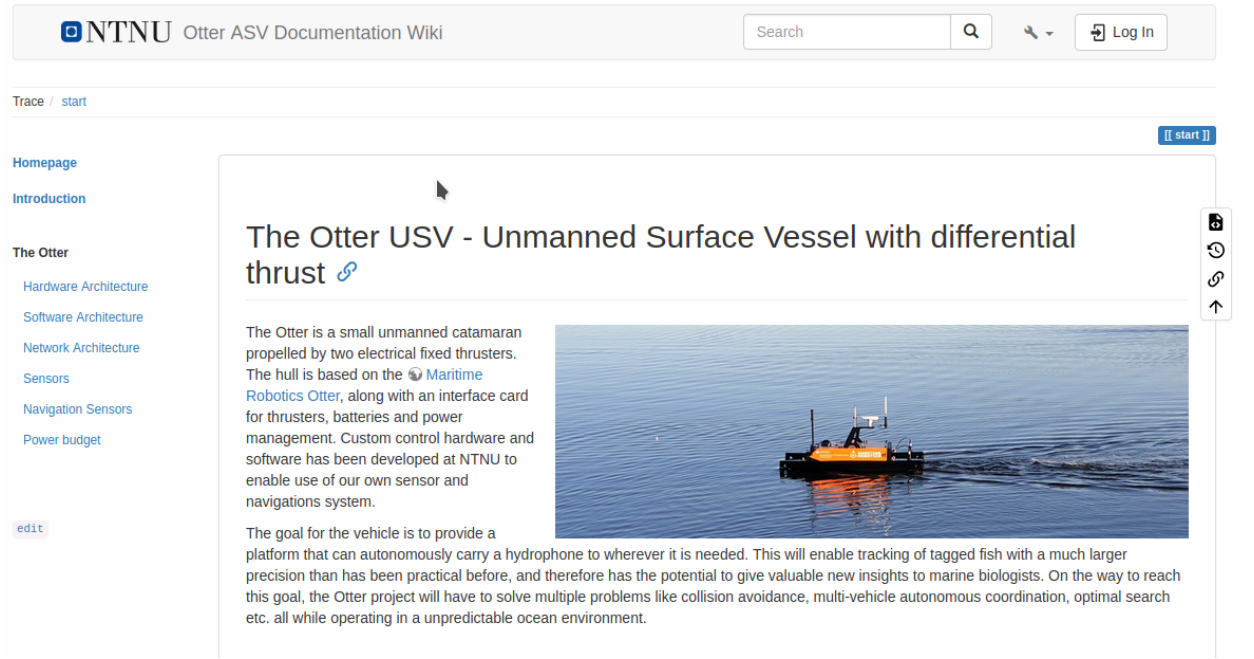

Figure 15: A screenshot of the Wiki made for the Otter, accessed 02/01/2020.

6.2 Documentation

Fulfilling requirement 9 has been done by creating a Wiki, and populating it with pages documentating the Otter. First a server had to be obtained, and was provided by the university technical support in what they called a semi-administrated server with Linux. These servers are kept updated and backed up by the IT-department, but can be customized by the user for different purposes by setting up services. For the purpose of a wiki, this is what was done:

- Configuring the firewall to allow traffic on TCP port 80 for networks outside NTNU.
- Configuring the tool *pkgsync*, which decides what .deb or .rpm software packages are used by the server. For serving web pages, apache2 is used, and it also needs the package libapache2-mod-php for running php scripts. To support XML reading in php, php-xml is also installed.
- Installing and configuring the wiki software, Dokuwiki.

Dokuwiki allows for easy creation and editing of wiki-pages through an online management system. The layout and look of the wiki is the only setting changed from a default installation by using a layout template named "Bootstrap", adding a sidebar and putting the NTNU logo in the header.

The pages and their content are very similar to the sections of this report, but with the addition of some more detailed step-by-step descriptions of how to replicate parts of the software design described in Section 4. The purpose of these guides is to ensure nearly identical setup for all deployed Otters, with only slight modifications of the DUNE configuration being needed.

The wiki can be found by accessing `http://otter.itk.ntnu.no/doku.php` on the web, and a screenshot of the front page is shown in Figure 15.

7 Conclusion

This project has performed a system integration for an ASV called the Fish Otter. The purpose of this system, is to be one vehicle in a formation of vehicles, registering acoustic fish tags detections. This data is useful, because the the time of the signal arrival can be used to locate the tag, giving more information about fish behavior.

To further this goal, a custom version of the LSTS toolchaing has been made. For DUNE, a configuration file was made to activate the tasks relevant for the Otter to function. Support for CAN in DUNE was missing, so this was developed. This made it possible to create a task to communicate with the Torqeedo Interface Board, in order to control the vehicle power and thrusters. Tasks has also been created for the Strato Pi CAN watchdog, and the TBR700RT hydrophone.

For the hydrophone, time synchronization has been established using the PPS signal from the GPS. The hardware connections was not present, so these has also been made. In the software, the Linux kernel has been customized, with support for use of PPS. In DUNE, custom IMC messages has been created to make the fish tag detections and hydrophone data available to other tasks.

Logging in DUNE has also been configured for relevant vehicle state metrics, as well as payload messages. Using Neptus, these logs are available for mission review and analysis.

7.1 Further Work

These tasks should be performed before the design is duplicated to multiple Otters:

- Verify PPS synchronization for system.
- Make the TBR700RT DUNE task send periodic sync messages.
- Make a custom PCB for the logic-level-shifter used on the PPS signal.
- Hardware and software support for 4G communication.
- Setting up and testing the Ubiquiti PowerBeam AC Gen2.
- Getting IMC addresses for the fish Otters in the official IMC repository.
- Properly tune the PID controllers of the course and speed controller, as well as the parameters for the ILOS path controller.

References

[1] D. Bhadauria, V. Isler, A. Studenski, and P. Tokekar. A robotic sensor network for monitoring carp in minnesota lakes. In *2010 IEEE International Conference on Robotics and Automation*, pages 3837–3842, May 2010.

[2] Walter Caharija, Mauro Candeloro, Kristin Y. Pettersen, and Asgeir J. Sørensen. Relative velocity control and integral los for path following of underactuated surface vessels. *IFAC Proceedings Volumes*, 45(27):380 – 385, 2012. 9th IFAC Conference on Manoeuvring and Control of Marine Craft.

[3] C. Forney, E. Manii, M. Farris, M. A. Moline, C. G. Lowe, and C. M. Clark. Tracking of a tagged leopard shark with an auv: Sensor calibration and state estimation. *2012 IEEE International Conference on Robotics and Automation*, 2012.

[4] T. M. Grothues, J. Dobarro, and J. Eiler. Collecting, interpreting, and merging fish telemetry data from an auv: Remote sensing from an already remote platform. *IEEE/OES Autonomous Underwater Vehicles*, 1695-1702, 2010.

[5] Aguiar A.P. De Sousa J.B. Zolich A. Johansen T.A. Alfredsen J.A. Erstorp E. Kuttenkeuler J. Jain, R.P. Localization of an acoustic fish-tag using the time-of-arrival measurements: preliminary results using the exogeneous kalman filter. *IEEE Int. Conf. on Intelligent Robots and Systems(IROS)*, 1695-1702, 2018.

[6] S. S. Løvskar. *Positioning of periodic acoustic emitters using an omnidirectional hydrophone on an AUV platform.* Master's thesis – NTNU, Department of Engineering Cybernetics, 2017.

[7] Kim Aarestrup Steven J. Cooke Paul D. Cowley Aaron T. Fisk Robert G. Harcourt Kim N. Holland Sara J. Iverson John F. Kocik Joanna E. Mills Flemming Fred G. Whoriskey Nigel E. Hussey, Steven T. Kessel. Aquatic animal telemetry: A panoramic window into the underwater world. *Science 12 Jun 2015: Vol. 348, Issue 6240, 1255642*, 2015.

[8] Gard Paulsen. *Alltid rabiat(Jens Glad Balchen og den kybernetiske tenkemåten).* Fagbokforlaget, 2019.

[9] José Pinto, Pedro Calado, José Braga, Paulo Sousa Dias, Ricardo Martins, Eduardo R. B. Marques, and João Borges de Sousa. Implementation of a control architecture for networked vehicle systems. *IFAC Workshop on Navigation, Guidance and Control of Underwater Vehicles (NGCUV'2012)*, 270180, 2012.

[10] Maritime Robotics. Otter. `https://www.maritimerobotics.com/otter`. Accessed: 2019-12-26.

[11] D. Shinzaki, C. Gage, S. Tang, M. Moline, B. Wolfe, C. G. Lowe, and C. Clark. A multi-auv system for cooperative tracking and following of leopard sharks. In *2013 IEEE International Conference on Robotics and Automation*, pages 4153–4158, May 2013.

[12] Torqeedo. Battery 915 wh ultralight 403. `https://www.torqeedo.com/en/products/accessories/spare-batteries/battery-915-wh-ultralight-403/1417-00.html`. Accessed: 2019-12-26.

[13] Torqeedo. Ultralight 403 a. `https://www.torqeedo.com/en/products/outboards/ultralight/ultralight-403-a/1405-00.html`. Accessed: 2019-12-26.

[14] Johansen T.A. Alfredsen J.A. Kuttenkeuler J. Erstorp E. Zolich, A. A formation of unmanned vehicles for tracking of an acoustic fish-tag. *IEEE OCEANS*, 1-6., 2017.

Appendices

A Pictures

A.1 The Control Box

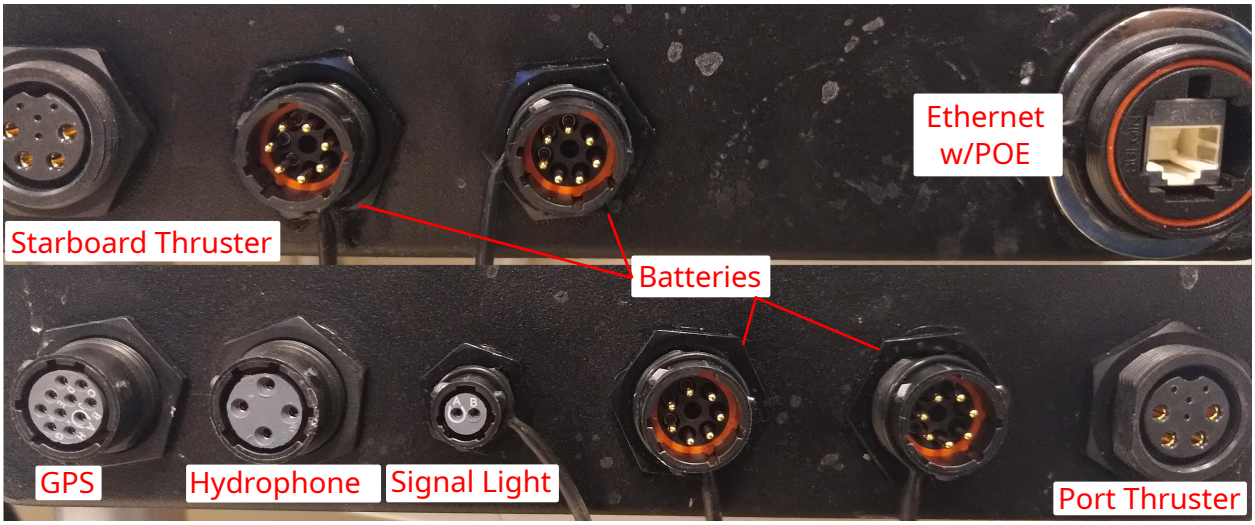

Figure 16: The connectors on the outside of the control box.

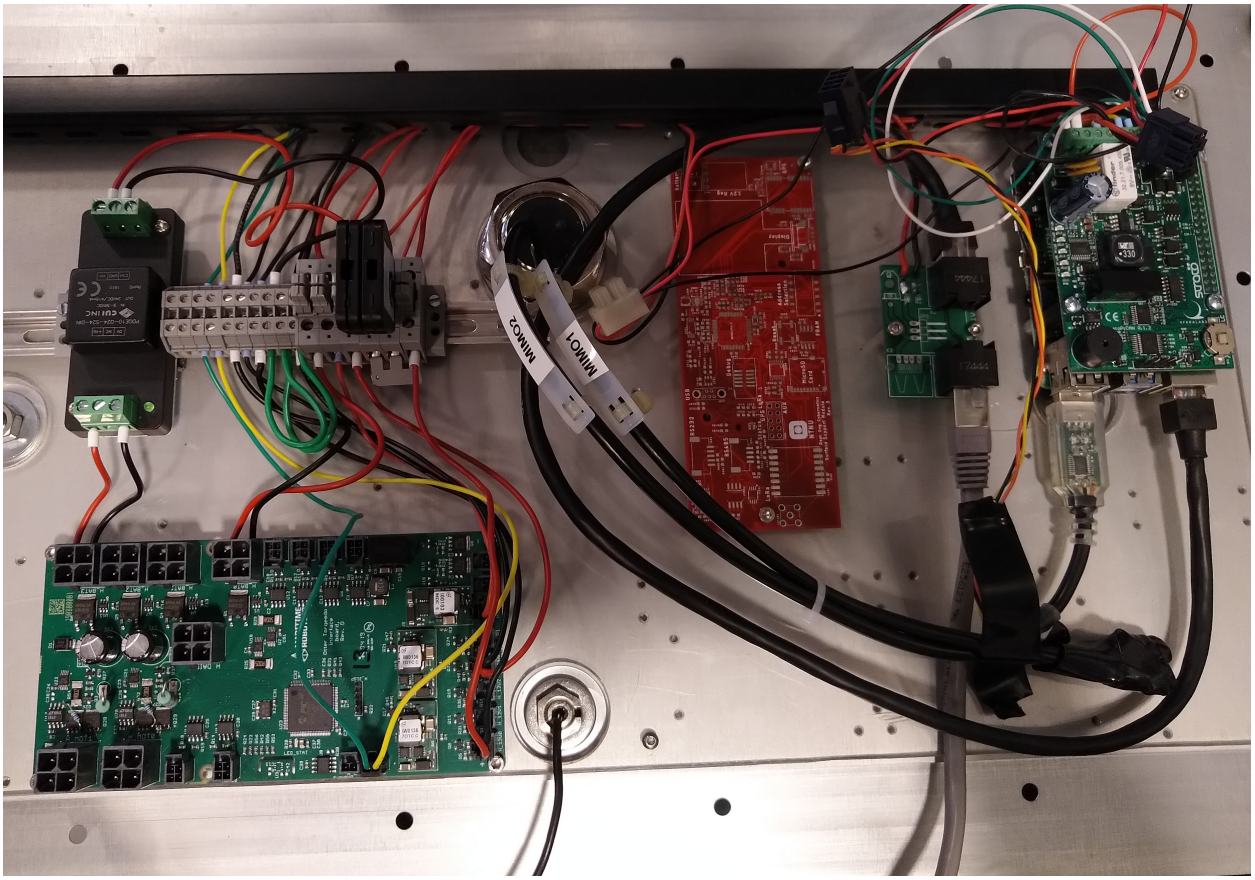

Figure 17: Picture of the control box components.

B Source code

B.1 Online Source Code and Documentation

The source of the software developed for this project available in these GitHub repositories:

- `https://github.com/nikkone/dune-nikolai`
- `https://github.com/nikkone/imc/tree/ntnuOtterASV`

The software of this project is based on the LSTS software toolchain. To see a description of this, and what changes/additions have been made in this project, see Section 4.

In addition to this text, a wiki has been created to document the work on the Otters. This can be found at:

- `http://otter.itk.ntnu.no/doku.php`

More information about the wiki is given in Section 6.2.

B.2 Source Code archives

This report should be distributed with three archives, one for Neptus, one for IMC and one for DUNE. Due to limitations with the digital systems used by NTNU, these were sent directly to my supervisor, and not distributed with this pdf.

B.3 etc/otter/basic.ini

```
############################################################################
# Copyright 2013-2019 Norwegian University of Science and Technology (NTNU)#
# Department of Engineering Cybernetics (ITK)                              #
############################################################################
# This file is part of DUNE: Unified Navigation Environment.               #
#                                                                          #
# Commercial Licence Usage                                                 #
# Licencees holding valid commercial DUNE licences may use this file in    #
# accordance with the commercial licence agreement provided with the       #
# Software or, alternatively, in accordance with the terms contained in a  #
# written agreement between you and Faculdade de Engenharia da             #
# Universidade do Porto. For licensing terms, conditions, and further      #
# information contact lsts@fe.up.pt.                                       #
#                                                                          #
# Modified European Union Public Licence - EUPL v.1.1 Usage                #
# Alternatively, this file may be used under the terms of the Modified     #
# EUPL, Version 1.1 only (the "Licence"), appearing in the file LICENCE.md #
# included in the packaging of this file. You may not use this work        #
# except in compliance with the Licence. Unless required by applicable     #
# law or agreed to in writing, software distributed under the Licence is   #
# distributed on an "AS IS" basis, WITHOUT WARRANTIES OR CONDITIONS OF      #
# ANY KIND, either express or implied. See the Licence for the specific    #
# language governing permissions and limitations at                        #
# https://github.com/LSTS/dune/blob/master/LICENCE.md and                  #
# http://ec.europa.eu/idabc/eupl.html.                                     #
############################################################################
# Author: Nikolai LauvÃěs                                                  #
############################################################################
# Otter USV configuration file.                                            #
############################################################################

[Require ../common/imc-addresses.ini]
[Require ../common/transports.ini]
[Require ../common/maneuvers.ini]

[Profiles]
StratoPi = Special simulation mode where only the hardware features of the StratoPi are active

############################################################################
# General Parameters.                                                      #
############################################################################
[IMC Addresses]
ntnu-otter-01                           = 0x2c60

[General]
```

```
Vehicle                                 = ntnu-otter-01
Vehicle Type                            = asv
Speed Conversion -- Actuation           = 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0
Speed Conversion -- RPM                 = 0.0, 120, 245, 360, 490, 615, 725, 845, 980, 980, 980
Speed Conversion -- MPS                 = 0.0, 1.0, 1.3, 1.6, 1.9, 2.2, 2.5, 2.8, 3.09, 3.09, 3.09
Absolute Maximum Depth                  = 0
Time Of Arrival Factor                  = 5.0

[Transports.Announce]
Enabled                                 = Always
Entity Label                            = Announce
Announcement Periodicity                = 10
Enable Loopback                         = 1
Enable Multicast                        = 1
Enable Broadcast                        = 1
Multicast Address                       = 224.0.75.69
Ports                                   = 30100, 30101, 30102, 30103, 30104
System Type                             = USV

[Transports.Discovery]
Enabled                                 = Always
Entity Label                            = Discovery
Multicast Address                       = 224.0.75.69
Ports                                   = 30100, 30101, 30102, 30103, 30104

#[Transports.Logging]
#Flush Interval                         = 0.5

################################################################################
# Navigation.                                                                  #
################################################################################

[Navigation.General.GPSNavigation]
Enabled                                 = Always
Entity Label                            = Navigation
Entity Label - GPS                      = GPS
Entity Label - Yaw                      = GPS

################################################################################
# Control.                                                                     #
################################################################################

[Control.ASV.HeadingAndSpeed]
Enabled                                 = Always
Entity Label                            = Course & Speed Controller
Debug Level                             = None
Maximum Thrust Actuation                = 1.0
Maximum Thrust Differential Actuation   = 0.4
Ramp Actuation Limit                    = 0.0
Hardware RPMs Control                   = true
RPMs at Maximum Thrust                  = 1100
RPMs PID Gains                          = 0.2e-3, 0.21e-3, 29.0e-6
RPMs Feedforward Gain                   = 0.46e-3
MPS PID Gains                           = 0.0, 25.0, 0.0
MPS Integral Limit                      = 200.0
MPS Feedforward Gain                    = 100.0
Minimum RPM Limit                       = 62
Maximum RPM Limit                       = 1100
Maximum RPM Acceleration                = 62
Yaw PID Gains                           = 1.5, 0.0, 0.0
Maximum Heading Error to Thrust         = 30.0
Entity Label - Port Motor               = Torqeedo - Motor 0
Entity Label - Starboard Motor          = Torqeedo - Motor 1
Share Saturation                        = true
Log PID Parcels                         = true

[Control.Path.PurePursuit]
Enabled                                 = Always
Entity Label                            = Path Control


#Integral line of sight
[Control.Path.ILOS]
Enabled                                 = Never
Entity Label                            = Path Control
Debug Level                             = None
Control Frequency                       = 10
Along-track -- Monitor                                  = true
Along-track -- Check Period             = 20
Along-track -- Minimum Speed            = 0.05
Along-track -- Minimum Yaw              = 2
```

```
Cross-track -- Monitor                  = true
Cross-track -- Nav. Unc. Factor         = -1
Cross-track -- Distance Limit           = 25
Cross-track -- Time Limit               = 20
Position Jump Threshold                 = 10.0
Position Jump Time Factor               = 0.5
ETA Minimum Speed                       = 0.1
New Reference Timeout                   = 5.0
Course Control                          = false
Corridor -- Width                       = 1.5
Corridor -- Entry Angle                 = 15.0
Corridor -- Out Vector Field            = true
Corridor -- Out LOS                     = false
ILOS Lookahead Distance                 = 4.0
ILOS Integrator Gain                    = 0.5
ILOS Integrator Initial Value           = 0.0
Bottom Track -- Enabled                 = false

[Control.Path.VectorField]
Enabled                                 = Never
Entity Label                            = Path Control
Debug Level                             = None
ETA Minimum Speed                       = 0.1
Control Frequency                       = 10
Along-track -- Monitor                  = false
Along-track -- Check Period             = 20
Along-track -- Minimum Speed            = 0.05
Along-track -- Minimum Yaw              = 2
Cross-track -- Monitor                  = false
Cross-track -- Nav. Unc. Factor         = 1
Cross-track -- Distance Limit           = 25
Cross-track -- Time Limit               = 20
Position Jump Threshold                 = 10.0
Position Jump Time Factor               = 0.5
ETA Minimum Speed                       = 0.1
New Reference Timeout                   = 5.0
Course Control                          = false
Corridor -- Width                       = 2.5
Corridor -- Entry Angle                 = 15.0
Extended Control -- Enabled             = false
Extended Control -- Controller Gain     = 1.0
Extended Control -- Turn Rate Gain      = 1.0
Bottom Track -- Enabled                 = false
Bottom Track -- Forward Samples         = 7
Bottom Track -- Safe Pitch              = 35.0
Bottom Track -- Minimum Range           = 4.0
Bottom Track -- Slope Hysteresis        = 1.5
Bottom Track -- Check Trend             = false
Bottom Track -- Execution Frequency     = 5
Bottom Track -- Depth Avoidance         = true
Bottom Track -- Admissible Altitude     = 2.5


[Control.ASV.RemoteOperation]
Enabled                                 = Always
Entity Label                            = Remote Control
Active                                  = true
Active - Scope                          = maneuver
Active - Visibility                     = developer
Execution Frequency                     = 10
Connection Timeout                      = 2.0

##########################################################################
# Maneuvers.                                                             #
##########################################################################

# Can this be removed?
[Maneuver.FollowReference.AUV]
Enabled                                 = Always
Entity Label                            = Follow Reference Maneuver
Horizontal Tolerance                    = 15.0
Vertical Tolerance                      = 1.0
Loitering Radius                        = 7.5
Default Speed                           = 50
Default Speed Units                     = percent
Default Z                               = 0
Default Z Units                         = DEPTH

[Maneuver.RowsCoverage]
Enabled                                 = Always
Entity Label                            = Rows Coverage Maneuver
```

```
##############################################################################
# Monitors / Supervisors                                                     #
##############################################################################

[Monitors.Clock]
Enabled                                 = Never
Entity Label                            = Clock
Minimum GPS Fixes                       = 30
Maximum Clock Offset                    = 2
Boot Synchronization Timeout            = 60
Hardware Clock Synchronization Command  = hwclock -w

[Monitors.Entities]
Enabled                                 = Always
Entity Label                            = Entity Monitor
Activation Time                         = 0
Deactivation Time                       = 0
Debug Level                             = None
Execution Priority                      = 10
Report Timeout                          = 5
Transition Time Gap                     = 4.0
Maximum Consecutive Transitions         = 3
Default Monitoring                      = Daemon,
                                          GPS,
                                          Navigation,
                                          Path Control
Default Monitoring -- Hardware                  = Torqeedo

[Supervisors.Vehicle]
Enabled                                 = Always
Entity Label                            = Vehicle Supervisor
Activation Time                         = 0
Deactivation Time                       = 0
Debug Level                             = None
Execution Priority                      = 10
Execution Frequency                     = 2
Allows External Control                 = false
Maneuver Handling Timeout               = 1.0

[Supervisors.AUV.LostComms]
Enabled                                                 = Never
Entity Label                                    = LostComms Supervisor AUV
Plan Name                                               = lost_comms
Lost Comms Timeout                              = 10.0
Debug Level                             = Spew

##############################################################################
# Hardware.                                                                  #
##############################################################################

[Actuators.Torqeedo]
Enabled                                 = Hardware, StratoPi
Execution Frequency                     = 40
Debug Level                             = Spew
Entity Label                            = Torqeedo
CAN Port - Device                       = can0
Power Channel H_MOT0 - Name             = Starboardmotor_pwr
Power Channel H_MOT0 - State            = 1
Power Channel H_MOT1 - Name             = Portmotor_pwr
Power Channel H_MOT1 - State            = 1
Power Channel H_VR0 - Name              = Signal_Light
Power Channel H_VR0 - State             = 1
Power Channel H_5V - Name               = Hydrophone
Power Channel H_5V - State              = 1
Motor write divider                     = 10

[Safety.StratoPIWatchdog]
Enabled                                                 = Hardware, StratoPi
Entity Label                                    = Watchdog
Execution Frequency                             = 0.5
TimeToggled                                     = 0.25
Debug Level                                     = Spew

# To use with the signal light
# The Identifiers are separated by commas, so more can be implemented easily
# The patterns are given first by on/off(0/1) for each led, followed by how long in millis. The pattern loops/repe

[UserInterfaces.LEDs]
Enabled                                                 = Hardware, StratoPi
```

```
Entity Label                                                = Signal Light
Interface                                                   = GPIO
Identifiers                                                 = 26
Critical Entities                                           = Logger
Pattern - Normal                                            = 1, 2000, 0, 2000
Pattern - Fuel Low                                          = 1, 200, 0, 200, 1, 200, 0, 2000
Pattern - Plan Starting                                     = 1, 200, 0, 2000
Pattern - Plan Executing                                    = 1, 500, 0, 500
Pattern - Error                                             = 1, 200, 0, 2000
Pattern - Fatal Error                                       = 1, 200, 0, 2000
Pattern - Shutdown                                          = 1, 200, 0, 2000

[Sensors.GPS]
Enabled                                 = Hardware
Entity Label                            = GPS
Serial Port - Device                    = /dev/ttyUSB0
Serial Port - Baud Rate                 = 19200
Sentence Order                          = GPHDT, GPROT, GPHDM, GPGGA, GPVTG, GPZDA
Debug Level                             = Spew
Initialization String 0 - Command       = $JASC,GPGGA,1\r\n
Initialization String 1 - Command       = $JASC,GPVTG,1\r\n
Initialization String 2 - Command       = $JASC,GPZDA,1\r\n
Initialization String 3 - Command       = $JATT,NMEAHE,0\r\n
Initialization String 4 - Command       = $JASC,GPROT,1\r\n
Initialization String 5 - Command       = $JASC,GPHDT,1\r\n
Initialization String 6 - Command       = $JASC,GPHDM,1\r\n
Initialization String 7 - Command       = $JSAVE\r\n

[Sensors.TBR700RT]
Enabled                                 = Hardware, StratoPi
Debug Level                             = Spew
Entity Label                            = Hydrophone
Serial Port - Device                    = /dev/ttyAMA0
Serial Port - Baud Rate                 = 115200

################################################################################
# Simulators.                                                                  #
################################################################################

[Require ../common/vsim-models.ini]

# Vehicle simulator.
[Simulators.VSIM]
Enabled                                 = Simulation, StratoPi
Entity Label                            = Simulation Engine
Execution Frequency                     = 25
Stream Speed East                       = 0
Stream Speed North                      = 0

# GPS simulator.
[Simulators.GPS]
Enabled                                 = Simulation, StratoPi
Execution Frequency                     = 1
Entity Label                            = GPS
Number of Satellites                    = 9
HACC                                    = 2
HDOP                                    = 0.9
Activation Depth                        = 0.2
Report Ground Velocity                  = false
Report Yaw                              = false
Initial Position                        = 63.33, 10.083333

# Port motor.
[Simulators.Motor/Port]
Enabled                                 = Simulation, StratoPi
Entity Label                            = Motor 0
Execution Frequency                     = 20
Thruster Act to RPM Factor              = 62, 620.0
Thruster Id                             = 0

# Starboard motor.
[Simulators.Motor/Starboard]
Enabled                                 = Simulation, StratoPi
Entity Label                            = Motor 1
Execution Frequency                     = 20
Thruster Act to RPM Factor              = 62, 620.0
Thruster Id                             = 1

################################################################################
# Transports.                                                                  #
################################################################################
```

```
[Transports.UDP]
Enabled                                 = Always
Entity Label                            = UDP
Debug Level                             = None
Activation Time                         = 0
Deactivation Time                       = 0
Execution Priority                      = 10
Announce Service                        = true
Contact Refresh Periodicity             = 5.0
Contact Timeout                         = 30
Dynamic Nodes                           = true
Local Messages Only                     = false
Transports                              = Acceleration,
                                          AngularVelocity,
                                          AutopilotMode,
                                          ControlParcel,
                                          CpuUsage,
                                          Current,
                                          DesiredPath,
                                          DesiredRoll,
                                          DesiredSpeed,
                                          DesiredVerticalRate,
                                          DesiredZ,
                                          EntityList,
                                          EntityParameters,
                                          EntityState,
                                          EstimatedState,
                                          EstimatedStreamVelocity,
                                          EulerAnglesDelta,
                                          FollowRefState,
                                          FuelLevel,
                                          GpsFix,
                                          GpsNavData,
                                          Heartbeat,
                                          IndicatedSpeed,
                                          LeaderState,
                                          LinkLevel,
                                          LogBookControl,
                                          LoggingControl,
                                          MagneticField,
                                          OperationalLimits,
                                          PathControlState,
                                          PlanControl,
                                          PlanControlState,
                                          PlanDB,
                                          PlanGeneration,
                                          PlanSpecification,
                                          PowerChannelControl,
                                          Pressure,
                                          QueryEntityParameters,
                                          RemoteActions,
                                          RemoteActionsRequest,
                                          Rpm,
                                          RSSI,
                                          SaveEntityParameters,
                                          SetEntityParameters,
                                          SetServoPosition,
                                          SetThrusterActuation,
                                          SimulatedState,
                                          StorageUsage,
                                          Target,
                                          TBRFishTag,
                                          TBRSensor,
                                          Temperature,
                                          TrexOperation,
                                          TrueSpeed,
                                          VehicleMedium,
                                          VehicleState,
                                          VelocityDelta,
                                          Voltage
Filtered Entities                       = CpuUsage:Daemon
Local Port                              = 6002
Print Incoming Messages                 = 0
Print Outgoing Messages                 = 0
Rate Limiters                           = CpuUsage:1,
                                          EntityState:1,
                                          EstimatedState:10,
                                          FuelLevel:0.1,
                                          SimulatedState:0.5,
                                          StorageUsage:0.05,
```

```
                                    Acceleration:10,
                                    AngularVelocity:10,
                                    MagneticField:10,
                                    Temperature:10,
                                    Pressure:10,
                                    EulerAnglesDelta:10,
                                    VelocityDelta:10

[Transports.Logging]
Enabled                           = Always
Entity Label                      = Logger
Flush Interval                    = 5
LSF Compression Method            = gzip
Transports                        = Acceleration,
                                    AngularVelocity,
                                    Announce,
                                    AutopilotMode,
                                    ControlLoops,
                                    ControlParcel,
                                    CpuUsage,
                                    Current,
                                    DesiredHeading,
                                    DesiredPath,
                                    DesiredRoll,
                                    DesiredSpeed,
                                    DesiredVerticalRate,
                                    DesiredZ,
                                    DevCalibrationControl,
                                    EntityList,
                                    EntityState,
                                    EstimatedState,
                                    EstimatedStreamVelocity,
                                    EulerAnglesDelta,
                                    FollowReference,
                                    FollowRefState,
                                    FuelLevel,
                                    GpsFix,
                                    GpsNavData,
                                    IndicatedSpeed,
                                    LeaderState,
                                    LinkLevel,
                                    LogBookEntry,
                                    MagneticField,
                                    ManeuverControlState,
                                    PathControlState,
                                    PlanControl,
                                    PlanSpecification,
                                    PlanControlState,
                                    PlanDB,
                                    PowerChannelControl,
                                    Pressure,
                                    Reference,
                                    Rpm,
                                    RSSI,
                                    SetControlSurfaceDeflection,
                                    SetThrusterActuation,
                                    SimulatedState,
                                    StopManeuver,
                                    StorageUsage,
                                    Temperature,
                                    TBRFishTag,
                                    TBRSensor,
                                    TrueSpeed,
                                    TrexObservation,
                                    TrexPlan,
                                    TrexToken,
                                    TrueSpeed,
                                    VehicleCommand,
                                    VehicleMedium,
                                    VehicleState,
                                    VelocityDelta,
                                    Voltage
```

B.4 src/Safety/StratoPIWatchdog/Task.cpp

```
//***************************************************************************
// Copyright 2007-2019 Universidade do Porto - Faculdade de Engenharia      *
// LaboratÃ³rio de Sistemas e Tecnologia SubaquÃ¡tica (LSTS)                  *
//***************************************************************************
```

```
// This file is part of DUNE: Unified Navigation Environment.                 *
//                                                                            *
// Commercial Licence Usage                                                   *
// Licencees holding valid commercial DUNE licences may use this file in      *
// accordance with the commercial licence agreement provided with the         *
// Software or, alternatively, in accordance with the terms contained in a    *
// written agreement between you and Faculdade de Engenharia da               *
// Universidade do Porto. For licensing terms, conditions, and further        *
// information contact lsts@fe.up.pt.                                         *
//                                                                            *
// Modified European Union Public Licence - EUPL v.1.1 Usage                  *
// Alternatively, this file may be used under the terms of the Modified       *
// EUPL, Version 1.1 only (the "Licence"), appearing in the file LICENCE.md   *
// included in the packaging of this file. You may not use this work          *
// except in compliance with the Licence. Unless required by applicable       *
// law or agreed to in writing, software distributed under the Licence is     *
// distributed on an "AS IS" basis, WITHOUT WARRANTIES OR CONDITIONS OF        *
// ANY KIND, either express or implied. See the Licence for the specific      *
// language governing permissions and limitations at                          *
// https://github.com/LSTS/dune/blob/master/LICENCE.md and                    *
// http://ec.europa.eu/idabc/eupl.html.                                       *
//****************************************************************************
// Author: Nikolai LauvÃěs                                                    *
//****************************************************************************

// DUNE headers.
#include <DUNE/DUNE.hpp>
#include <DUNE/Hardware/GPIO.hpp>

namespace Safety
{
  //! This task is used with the watchdog feature of the StratoPi CAN expansion board for the Raspberry Pi.
  //!
  //! The task activates the watchdog when it's activated.
  //! The task is periodic, and toggles the heartbeat pin once for every time it executes.
  //! @author Nikolai LauvÃěs
  namespace StratoPIWatchdog
  {
    using DUNE_NAMESPACES;

    struct Arguments
    {
      //! Toggle
      float toggled_time;
    };

    struct Task: public DUNE::Tasks::Periodic
    {
      Hardware::GPIO* m_gpio_activation_pin;
      Hardware::GPIO* m_gpio_heartbeat_pin;
      Hardware::GPIO* m_gpio_watchdog_timeout_pin;
      Hardware::GPIO* m_gpio_watchdog_timeout_answer_pin;
      Arguments m_args;
      //! Constructor.
      //! @param[in] name task name.
      //! @param[in] ctx context.
      Task(const std::string& name, Tasks::Context& ctx):
        DUNE::Tasks::Periodic(name, ctx),
        m_gpio_activation_pin(NULL),
        m_gpio_heartbeat_pin(NULL),
        m_gpio_watchdog_timeout_pin(NULL),
        m_gpio_watchdog_timeout_answer_pin(NULL)
      {

        param("TimeToggled", m_args.toggled_time)
        .units(Units::Second)
        .description("How long GPIO5 stays toggled")
        .defaultValue("1.00");

      }

      //! Update internal state with new parameter values.
      void
      onUpdateParameters(void)
      {
      }

      //! Reserve entity identifiers.
      void
      onEntityReservation(void)
```

```
        {
        }

        //! Resolve entity names.
        void
        onEntityResolution(void)
        {
        }

        //! Acquire resources.
        void
        onResourceAcquisition(void)
        {
          m_gpio_heartbeat_pin = new Hardware::GPIO(5);
          m_gpio_activation_pin = new Hardware::GPIO(6);
          m_gpio_watchdog_timeout_pin = new Hardware::GPIO(12);
          m_gpio_watchdog_timeout_answer_pin = new Hardware::GPIO(16);
          setEntityState(IMC::EntityState::ESTA_NORMAL, Status::CODE_ACTIVE);
        }

        //! Initialize resources.
        void
        onResourceInitialization(void)
        {
          m_gpio_heartbeat_pin->setDirection(Hardware::GPIO::GPIO_DIR_OUTPUT);
          m_gpio_activation_pin->setDirection(Hardware::GPIO::GPIO_DIR_OUTPUT);
          m_gpio_watchdog_timeout_pin->setDirection(Hardware::GPIO::GPIO_DIR_INPUT);
          m_gpio_watchdog_timeout_answer_pin->setDirection(Hardware::GPIO::GPIO_DIR_OUTPUT);

          m_gpio_heartbeat_pin->setValue(0);
          m_gpio_activation_pin->setValue(1);
          m_gpio_watchdog_timeout_answer_pin->setValue(0);
        }

        //! Release resources.
        void
        onResourceRelease(void)
        {
          Memory::clear(m_gpio_heartbeat_pin);
          Memory::clear(m_gpio_activation_pin);
          Memory::clear(m_gpio_watchdog_timeout_pin);
          Memory::clear(m_gpio_watchdog_timeout_answer_pin);
        }

        //! Main loop.
        void
        task(void)
        {
          if(m_gpio_watchdog_timeout_pin->getValue()) {
            err(DTR("StratoPIWatchdog timeout"));
            setEntityState(IMC::EntityState::ESTA_FAILURE, "StratoPIWatchdog timeout");
          }

          debug(DTR("Toggled heartbeat gpio"));
          m_gpio_heartbeat_pin->setValue(1);
          Delay::wait(m_args.toggled_time);
          m_gpio_heartbeat_pin->setValue(0);
        }
      };
    }
}

DUNE_TASK
```

B.5 src/Actuators/Torqeedo/Task.cpp

```
//***************************************************************************
// Copyright 2013-2019 Norwegian University of Science and Technology (NTNU)*
// Department of Engineering Cybernetics (ITK)                              *
//***************************************************************************
// This file is part of DUNE: Unified Navigation Environment.               *
//                                                                          *
// Commercial Licence Usage                                                 *
// Licencees holding valid commercial DUNE licences may use this file in    *
// accordance with the commercial licence agreement provided with the       *
// Software or, alternatively, in accordance with the terms contained in a  *
// written agreement between you and Faculdade de Engenharia da             *
// Universidade do Porto. For licensing terms, conditions, and further      *
// information contact lsts@fe.up.pt.                                       *
//                                                                          *
```

```
// Modified European Union Public Licence - EUPL v.1.1 Usage                  *
// Alternatively, this file may be used under the terms of the Modified       *
// EUPL, Version 1.1 only (the "Licence"), appearing in the file LICENCE.md   *
// included in the packaging of this file. You may not use this work          *
// except in compliance with the Licence. Unless required by applicable       *
// law or agreed to in writing, software distributed under the Licence is     *
// distributed on an "AS IS" basis, WITHOUT WARRANTIES OR CONDITIONS OF        *
// ANY KIND, either express or implied. See the Licence for the specific      *
// language governing permissions and limitations at                          *
// https://github.com/LSTS/dune/blob/master/LICENCE.md and                    *
// http://ec.europa.eu/idabc/eupl.html.                                       *
//*****************************************************************************
// Author: Nikolai LauvÃěs                                                    *
//*****************************************************************************

// ISO C++ 98 headers.
#include <map>
// DUNE headers.
#include <DUNE/DUNE.hpp>
#include <DUNE/Hardware/SocketCAN.hpp>

#define ADDR_SOURCE 0xfe
#define ADDR_TQIF 0xab
namespace Actuators
{
  //! This task acts as a bridge between the Maritime Robotics(MR) Interface card to Torqeedo motors and batteries
  //! Messages are sent to the motors periodically, at least once per/second, or else motors stops(see msg_tq_mot
  //!
  //! Reads and writes CAN frames to a buffer that is sent to Hardware::SocketCAN.
  //! @author Nikolai LauvÃěs
  namespace Torqeedo
  {
    using DUNE_NAMESPACES;
    //! Maximum number of batteries connected to the Torqeedo board
    static const unsigned c_num_batteries = 4;
    //! Number of power channels
    static const unsigned c_pwrs_count = 10;
    //! Number of power rails
    static const unsigned c_pwr_rails_count = 4;
    //! Number of motors
    static const unsigned c_motors = 2;
    enum torqeedo_msg_identifiers_t
    {
      MSG_TEXT = 0,
      MSG_CAP_AMP = 1,
      MSG_CAP_WATT = 2,
      MSG_RAIL = 3,
      MSG_HOUSEKEEPING = 4,
      MSG_TEMPERATURE = 5,
      MSG_ID = 6,
      MSG_BATCELLS = 7,
      MSG_OUTPUTS = 8,
      MSG_OUTPUT_SET = 9,
      MSG_UPTIME = 10,
      MSG_BOOTLOADER = 11,
      MSG_TQ_MOTOR_DRIVE = 12,
      MSG_TQ_MOTOR_SET = 13,
      MSG_TQ_BAT_STATUS = 14,
      MSG_TQ_BATCTL = 15,
      MSG_TQ_MOTOR_STATUS_BITS = 16,
      MSG_RESET = 17,
      MSG_WINCH_TELEMETRY = 18,
      MSG_WINCH_COMMAND = 19,
      MSG_WINCH_MOVING = 20,
      MSG_ID_V2 = 21
    };

    enum torqeedo_power_rails_t
    {
      R_H_MOT0 = 0,
      R_H_MOT1 = 1,
      R_H_AUX0 = 2,
      R_H_AUX1 = 2,
      R_H_12V0 = 3,
      R_H_12V1 = 3,
      R_H_12V2 = 3,
      R_H_VR0 = 3,
      R_H_VR1 = 3,
      R_H_5V = 3
    };
```

```
enum torqeedo_power_channels_t
{
  CH_H_MOT0 = 0,
  CH_H_MOT1 = 0,
  CH_H_AUX0 = 0,
  CH_H_AUX1 = 1,
  CH_H_12V0 = 0,
  CH_H_12V1 = 1,
  CH_H_12V2 = 2,
  CH_H_VR0 = 3,
  CH_H_VR1 = 4,
  CH_H_5V = 5
};
struct Arguments
{
  //! CAN bus device name
  std::string can_dev;
  //! Power channels names.
  std::string pwr_names[c_pwrs_count];
  //! Initial power channels states.
  unsigned pwr_states[c_pwrs_count];
  //! Write to motor every motor_write_divider times task is run
  unsigned int motor_write_divider;
};
//! Power Channel data structure.
struct PowerChannel
{
  torqeedo_power_rails_t rail;
  torqeedo_power_channels_t channel;
  IMC::PowerChannelState::StateEnum state;
};
struct Task: public DUNE::Tasks::Periodic
{
  //! Is there unsent power control messages
  bool m_unsent_power_parameters;
  //! Most recent throttle values.
  unsigned int motor_send_counter;
  // Datatype for storing power lines and states
  typedef std::map<std::string, PowerChannel> PowerChannelMap;
  //! Power channels by name.
  PowerChannelMap m_pwr_chs;
  //! Batteries Entities
  unsigned m_battery_eid[c_num_batteries];
  //! Motors Entities
  unsigned m_motor_eid[c_motors];
  //! Power Rail Entities
  unsigned m_power_rail_eid[c_pwr_rails_count];
  //! Most recent throttle values.
  int16_t motor0_throttle, motor1_throttle;
  //! CAN connection variable
  Hardware::SocketCAN* m_can;
  //! CAN buffer used for storing and sending messages
  char m_can_bfr[9];
  //! Task arguments.
  Arguments m_args;
  //! Constructor.
  //! @param[in] name task name.
  //! @param[in] ctx context.
  Task(const std::string& name, Tasks::Context& ctx):
    DUNE::Tasks::Periodic(name, ctx),
    m_unsent_power_parameters(false),
    motor_send_counter(0),
    motor0_throttle(0),
    motor1_throttle(0),
    m_can(NULL)
  {
    param("CAN Port - Device", m_args.can_dev)
    .defaultValue("")
    .description("CAN port used to communicate with the Torqeedo board.");

    param("Motor write divider", m_args.motor_write_divider)
    .defaultValue("20")
    .description("Write to motor every motor_write_divider times task is run");

    char power_channel_pcb_labels[c_pwrs_count][8] = {"H_MOT0\0","H_MOT1\0","H_AUX0\0","H_AUX1\0","H_12V0\0","
    for (unsigned i= 0; i < c_pwrs_count; i++)
    {
      std::string option = String::str("Power Channel %s - Name", power_channel_pcb_labels[i]);
      param(option, m_args.pwr_names[i]);

      option = String::str("Power Channel %s - State", power_channel_pcb_labels[i]);
```

```
      param(option, m_args.pwr_states[i])
      .defaultValue("0");
    }
    // Register handler routines.
    bind<IMC::SetThrusterActuation>(this);
    bind<IMC::PowerChannelControl>(this);
}

//! Update internal state with new parameter values.
void
onUpdateParameters(void)
{
    inf(DTR("Update parameters"));
    if(m_pwr_chs.size() != 0) {
      m_pwr_chs.clear();
      m_unsent_power_parameters = true;
    }
    // Set up powerchannels
    PowerChannel pcs[c_pwrs_count];
    pcs[0] = {R_H_MOT0, CH_H_MOT0, IMC::PowerChannelState::PCS_OFF};
    pcs[1] = {R_H_MOT1, CH_H_MOT1, IMC::PowerChannelState::PCS_OFF};
    pcs[2] = {R_H_AUX0, CH_H_AUX0, IMC::PowerChannelState::PCS_OFF};
    pcs[3] = {R_H_AUX1, CH_H_AUX1, IMC::PowerChannelState::PCS_OFF};
    pcs[4] = {R_H_12V0, CH_H_12V0, IMC::PowerChannelState::PCS_OFF};
    pcs[5] = {R_H_12V1, CH_H_12V1, IMC::PowerChannelState::PCS_OFF};
    pcs[6] = {R_H_12V2, CH_H_12V2, IMC::PowerChannelState::PCS_OFF};
    pcs[7] = {R_H_VR0, CH_H_VR0, IMC::PowerChannelState::PCS_OFF};
    pcs[8] = {R_H_VR1, CH_H_VR1, IMC::PowerChannelState::PCS_OFF};
    pcs[9] = {R_H_5V, CH_H_5V, IMC::PowerChannelState::PCS_OFF};
    for (unsigned i = 0; i < c_pwrs_count; i++)
    {
      if(m_args.pwr_states[i]) {
        pcs[i].state = IMC::PowerChannelState::PCS_ON;
      }
      if(!(m_args.pwr_names[i].empty())) {
        m_pwr_chs[m_args.pwr_names[i]] = pcs[i];
      }
    }
}

//! Reserve entity identifiers.
void
onEntityReservation(void)
{
    std::string label = getEntityLabel();

    for (unsigned i = 0; i < c_motors; i++)
    {
      m_motor_eid[i] = reserveEntity(label + " - Motor " + std::to_string(i));
    }

    for (unsigned i = 0; i < c_num_batteries; i++)
    {
      m_battery_eid[i] = reserveEntity(label + " - Battery " + std::to_string(i));
    }

    for (unsigned i = 0; i < c_pwr_rails_count; i++)
    {
      m_power_rail_eid[i] = reserveEntity(label + " - Rail " + std::to_string(i));
    }
}

//! Resolve entity names.
void
onEntityResolution(void)
{
}

//! Acquire resources.
void
onResourceAcquisition(void)
{
    try {
      m_can = new Hardware::SocketCAN(m_args.can_dev, SocketCAN::CAN_BASIC_EFF);
      setEntityState(IMC::EntityState::ESTA_NORMAL, Status::CODE_ACTIVE);
    }
    catch(std::runtime_error& e) {
      cri(DTR("Could not open CAN: %s"), e.what());
      setEntityState(IMC::EntityState::ESTA_ERROR, Status::CODE_IO_ERROR);
    }
}
```

```
void sendPowerChannelMessages() {
  for (PowerChannelMap::iterator itr = m_pwr_chs.begin(); itr != m_pwr_chs.end(); ++itr)
  {
    sendSetPower(itr->second);
  }
}

//! Initialize resources.
void
onResourceInitialization(void)
{
  spew(DTR("Init resources"));
  if(m_can != NULL) {
    sendPowerChannelMessages();
  }
}

//! Release resources.
void
onResourceRelease(void)
{
  try {
   Memory::clear(m_can);
  }
  catch(std::runtime_error& e) {
    err(DTR("Could not clear CAN: %s"), e.what());
  }
}

//! Consume SetThrusterActuation messages
void
consume(const IMC::SetThrusterActuation* msg)
{
  switch (msg->id)
  {
  case 0:
    motor0_throttle = int16_t(1000 * msg->value);
    break;
  case 1:
    motor1_throttle = int16_t(1000 * msg->value);
    break;
  }
}

//! Consume PowerChannelControl messages, forward them to CAN bus
void
consume(const IMC::PowerChannelControl* msg)
{
  PowerChannelMap::const_iterator itr = m_pwr_chs.find(msg->name);
  if (itr == m_pwr_chs.end())
    return;

  IMC::PowerChannelControl::OperationEnum op = static_cast<IMC::PowerChannelControl::OperationEnum>(msg->op
  if (op == IMC::PowerChannelControl::PCC_OP_TURN_ON)
    m_pwr_chs[msg->name].state = IMC::PowerChannelState::PCS_ON;
  else if (op == IMC::PowerChannelControl::PCC_OP_TURN_OFF)
    m_pwr_chs[msg->name].state = IMC::PowerChannelState::PCS_OFF;
  else
    war("Chosen power state not implemented.");
  sendSetPower(m_pwr_chs[msg->name]);
}

//! Convenience/readability function for combining two char inputs to one uint16_t
uint16_t
combine2charToUint16(char most_significant, char least_significant)
{
  return (uint16_t)(most_significant << 8) | least_significant;
}

//! Convenience/readability function for combining two char inputs to one int16_t
int16_t
combine2charToInt16(char most_significant, char least_significant)
{
  return (int16_t)(most_significant << 8) | least_significant;
}

//! Parses a received MSG_TQ_BAT_STATUS from CAN bus buffer and sends relevant data to IMC
void
parseMSG_TQ_BAT_STATUS()
```

```
{
  uint8_t bat_idx = m_can_bfr[0];
  uint8_t temp_C = m_can_bfr[1];
  uint16_t voltage_raw = combine2charToUint16(m_can_bfr[3], m_can_bfr[2]);
  uint16_t current_raw = combine2charToUint16(m_can_bfr[5], m_can_bfr[4]);
  uint8_t soc = m_can_bfr[6];  // State of charge
  uint8_t err_code = m_can_bfr[7];

  fp32_t voltage = fp32_t(voltage_raw) * 0.01;
  fp32_t current = fp32_t(current_raw) * 0.1;
  trace("MSG_TQ_BAT_STATUS: Batt#%d - Charge: %d; Voltage %0.2fV; Current: %0.1fA; Temp: %d, Error: %d",
        bat_idx, soc, voltage, current, temp_C, err_code);

  IMC::Temperature temp_msg;
  temp_msg.setSourceEntity(m_battery_eid[bat_idx]);
  temp_msg.value = fp32_t(temp_C);
  dispatch(temp_msg);

  IMC::Voltage voltage_msg;
  voltage_msg.setSourceEntity(m_battery_eid[bat_idx]);
  voltage_msg.value = voltage;
  dispatch(voltage_msg);

  IMC::Current current_msg;
  current_msg.setSourceEntity(m_battery_eid[bat_idx]);
  current_msg.value = current;
  dispatch(current_msg);

  IMC::FuelLevel level_msg;
  level_msg.setSourceEntity(m_battery_eid[bat_idx]);
  level_msg.value = fp32_t(soc);
  dispatch(level_msg);
}

//! Parses a received MSG_RAIL from CAN bus buffer and sends relevant data to IMC
void
parseMSG_RAIL()
{
  uint8_t rail_idx = m_can_bfr[0];
  ///Voltage (mV)
  uint16_t voltage_mV = combine2charToUint16(m_can_bfr[2], m_can_bfr[1]);
  ///Current (mA) TODO: Should this be int32_t? signed in mrcan
  int32_t current_mA = (int32_t)0 | (m_can_bfr[5] << 16) | (m_can_bfr[4] << 8) | m_can_bfr[3];
  ///Electronic fuse trip current (A*2)
  uint8_t fuse_halfamps = m_can_bfr[6];
  char flags = m_can_bfr[7];


  fp32_t voltage_V = fp32_t(voltage_mV) * 0.001;
  fp32_t current_A = fp32_t(current_mA) * 0.001;
  trace("MSG_RAIL: Rail#%d - Voltage: %0.3fV, Current: %f A, fuse_halfamps: %u, flags: %02X", rail_idx, volt

  IMC::Voltage voltage_msg;
  voltage_msg.setSourceEntity(m_power_rail_eid[rail_idx]);
  voltage_msg.value = voltage_V;
  dispatch(voltage_msg);

  IMC::Current current_msg;
  current_msg.setSourceEntity(m_power_rail_eid[rail_idx]);
  current_msg.value = current_A;
  dispatch(current_msg);
}

//! Parses a received MSG_TQ_MOTOR_DRIVE from CAN bus buffer and sends relevant data to IMC
void
parseMSG_TQ_MOTOR_DRIVE()
{
  uint8_t mot_idx = m_can_bfr[0];
  /// Power in whole watts
  uint16_t power = combine2charToUint16(m_can_bfr[2], m_can_bfr[1]);
  /// PCB temperature in tenths of degrees celsius
  int16_t temp_raw = combine2charToInt16(m_can_bfr[4], m_can_bfr[3]);
  /// Divide by 7 (gear ratio) to get propeller RPM
  uint16_t rpm_raw = combine2charToUint16(m_can_bfr[6], m_can_bfr[5]);

  fp32_t temp = fp32_t(temp_raw) * 0.1;
  int16_t rpm = (int16_t)rpm_raw / 7; // Rounds down to nearest whole number

  trace("MSG_TQ_MOTOR_DRIVE: Motor#%d - Power: %dW; Temp %0.1fC; RPM: %d",
        mot_idx, power, temp, rpm);
```

```
  IMC::Temperature temp_msg;
  temp_msg.setSourceEntity(m_motor_eid[mot_idx]);
  temp_msg.value = temp;
  dispatch(temp_msg);

  IMC::Rpm rpm_msg;
  rpm_msg.setSourceEntity(m_motor_eid[mot_idx]);
  rpm_msg.value = rpm;
  dispatch(rpm_msg);
}

//! Parses a received MSG_TEXT and displays it
void
parseMSG_TEXT()
{
  inf("MSG_TEXT: %s", m_can_bfr);
}

//! Parses a received MSG_TQ_BATCTL from CAN bus buffer and makes it available for trace debug
void
parseMSG_TQ_BATCTL()
{
  uint8_t motor_index = m_can_bfr[0]; // MAY need to be masked, only 4 last bits id, rest reserved
  uint8_t master_error = m_can_bfr[1];
  uint8_t error_count = m_can_bfr[2];
  uint8_t firmware_ver = m_can_bfr[3];
  trace(DTR("MSG_TQ_BATCTL: Motor#%u - Master error: %u; Error count %u; Firmware version: %u"),
        motor_index, master_error, error_count, firmware_ver);

}

//! Parses a received MSG_OUTPUTS from CAN bus buffer and makes it available for trace debug
void
parseMSG_OUTPUTS()
{
  uint8_t rail_index = m_can_bfr[0];
  uint32_t states = (m_can_bfr[4] << 24) | (m_can_bfr[3] << 16) | (m_can_bfr[2] << 8) | m_can_bfr[1];
  trace(DTR("MSG_OUTPUTS: Rail#%u - Master error: %08X;"),
        rail_index, states);

}

//! Parses a received MSG_UPTIME from CAN bus buffer and makes it available for trace debug
void
parseMSG_UPTIME()
{
  uint32_t uptime_s = (uint32_t)0 | (m_can_bfr[2] << 16) | (m_can_bfr[1] << 8) | m_can_bfr[0];
  uint8_t last_reset_case = m_can_bfr[3];
  trace(DTR("MSG_UPTIME: Uptime#%ds; Last reset case: %01X;"),
        uptime_s, last_reset_case);
}

//! Parses a received MSG_ID_V2 from CAN bus buffer and makes it available for trace debug
void
parseMSG_ID_V2()
{
  uint16_t company = combine2charToUint16(m_can_bfr[1], m_can_bfr[0]);
  uint16_t product = combine2charToUint16(m_can_bfr[3], m_can_bfr[2]);
  uint16_t serial_number = combine2charToUint16(m_can_bfr[5], m_can_bfr[4]);
  uint16_t firmware_version = combine2charToUint16(m_can_bfr[7], m_can_bfr[6]);
  trace(DTR("MSG_ID_V2: Company#%d; Product: %d; Serial number: %d; Firmware: %d;"),
        company, product, serial_number, firmware_version);
}

//! Parses a received MSG_TQ_MOTOR_STATUS_BITS from CAN bus buffer and makes it available for trace debug
void
parseMSG_TQ_MOTOR_STATUS_BITS()
{
  uint8_t motor_index = m_can_bfr[0];
  uint16_t errors = combine2charToUint16(m_can_bfr[2], m_can_bfr[1]);
  uint8_t status = m_can_bfr[3];
  trace(DTR("MSG_TQ_MOTOR_STATUS_BITS: Motor#%u - Errors: %u; Status %u"),
        motor_index, errors, status);
}
//! Tries to read a message from CAN bus, if successful, call relevant parser
void
readCanMessage()
{
  // Read message
  uint32_t id;
  if (Poll::poll(*m_can, 0.01)) {
```

```
    m_can->readString(m_can_bfr, sizeof(m_can_bfr));
    id = m_can->getRXID();
  } else {
    return;
  }

  // Extract message identifier
  uint8_t msg_id = uint8_t(id >> 20);
  // Parse message
  switch(msg_id) {
    case MSG_TEXT:
      parseMSG_TEXT();
      break;
    case MSG_RAIL:
      parseMSG_RAIL();
      break;
    case MSG_TQ_MOTOR_DRIVE:
      parseMSG_TQ_MOTOR_DRIVE();
      break;
    case MSG_TQ_BAT_STATUS:
      parseMSG_TQ_BAT_STATUS();
      break;
    case MSG_TQ_BATCTL:
      parseMSG_TQ_BATCTL();
      break;
    case MSG_OUTPUTS:
      parseMSG_OUTPUTS();
      break;
    case MSG_UPTIME:
      parseMSG_UPTIME();
      break;
    case MSG_ID_V2:
      parseMSG_ID_V2();
      break;
    case MSG_TQ_MOTOR_STATUS_BITS:
      parseMSG_TQ_MOTOR_STATUS_BITS();
      break;
    case MSG_CAP_AMP:
    case MSG_CAP_WATT:
    case MSG_HOUSEKEEPING:
    case MSG_TEMPERATURE:
    case MSG_ID:
    case MSG_BATCELLS:
    case MSG_OUTPUT_SET:
    case MSG_BOOTLOADER:
    case MSG_TQ_MOTOR_SET:
    case MSG_RESET:
    case MSG_WINCH_TELEMETRY:
    case MSG_WINCH_COMMAND:
    case MSG_WINCH_MOVING:

      trace(DTR("Known unimplemented MSG type received: %08X"), id);
      break;
    default:
      inf(DTR("Unknown CAN MSG received: %08X"), id);
  }
}

//! Compiles a CAN id in a format supported by the Torqeedo interface board
uint32_t
prepareTorqeedoCANID(torqeedo_msg_identifiers_t msg_id)
{
  return uint32_t(ADDR_TQIF | (ADDR_SOURCE << 8) | (msg_id << 20));
}

//! Sends MSG_OUTPUT_SET to CAN bus
void
sendSetPower(PowerChannel pc)
{ // Dont send struct, send pointer or something
  m_can_bfr[0] = pc.rail;
  m_can_bfr[1] = pc.channel;
  if(pc.state == IMC::PowerChannelState::PCS_OFF) {
    m_can_bfr[2] = 0;
  } else {
    m_can_bfr[2] = 1;
  }
  m_can->setTXID(prepareTorqeedoCANID(MSG_OUTPUT_SET));
  m_can->write(m_can_bfr, 3);
}

//! Sends MSG_TQ_MOTOR_SET to CAN bus
```

```
        void
        sendSetMotorThrottle( int16_t motor0, int16_t motor1)
        {

          m_can_bfr[0] = (char)(motor0 & 0x00FF);
          m_can_bfr[1] = (char)((motor0 & 0xFF00) >> 8);
          m_can_bfr[2] = (char)(motor1 & 0x00FF);
          m_can_bfr[3] = (char)((motor1 & 0xFF00) >> 8);

          m_can->setTXID(prepareTorqeedoCANID(MSG_TQ_MOTOR_SET));
          m_can->write(m_can_bfr, 4);
        }

        //! Main loop.
        void
        task(void)
        {
          if(m_can != NULL) {
            waitForMessages(0.01); // Parametriser?
            motor_send_counter++;
            if(motor_send_counter >= m_args.motor_write_divider) {
              spew(DTR("Motor send: %d, %d"), motor0_throttle, motor1_throttle);
              sendSetMotorThrottle(motor0_throttle, motor1_throttle);
              motor_send_counter = 0;
            } else if(m_unsent_power_parameters) {
              sendPowerChannelMessages();
              m_unsent_power_parameters = false;
            } else {
              readCanMessage();
            }
          }
        }
      };
    }
}
```

DUNE_TASK

B.6 src/Sensors/TBR700RT/Task.cpp

```
//***************************************************************************
// Copyright 2007-2019 Universidade do Porto - Faculdade de Engenharia      *
// LaboratÃ³rio de Sistemas e Tecnologia SubaquÃ¡tica (LSTS)                  *
//***************************************************************************
// This file is part of DUNE: Unified Navigation Environment.               *
//                                                                          *
// Commercial Licence Usage                                                 *
// Licencees holding valid commercial DUNE licences may use this file in    *
// accordance with the commercial licence agreement provided with the       *
// Software or, alternatively, in accordance with the terms contained in a  *
// written agreement between you and Faculdade de Engenharia da             *
// Universidade do Porto. For licensing terms, conditions, and further      *
// information contact lsts@fe.up.pt.                                       *
//                                                                          *
// Modified European Union Public Licence - EUPL v.1.1 Usage                *
// Alternatively, this file may be used under the terms of the Modified     *
// EUPL, Version 1.1 only (the "Licence"), appearing in the file LICENCE.md *
// included in the packaging of this file. You may not use this work        *
// except in compliance with the Licence. Unless required by applicable     *
// law or agreed to in writing, software distributed under the Licence is   *
// distributed on an "AS IS" basis, WITHOUT WARRANTIES OR CONDITIONS OF      *
// ANY KIND, either express or implied. See the Licence for the specific    *
// language governing permissions and limitations at                        *
// https://github.com/LSTS/dune/blob/master/LICENCE.md and                  *
// http://ec.europa.eu/idabc/eupl.html.                                     *
//***************************************************************************
// Author: Nikolai LauvÃ«s (based on GPS by Ricardo Martins)                 *
//***************************************************************************

// ISO C++ 98 headers.
#include <cstring>
#include <algorithm>
#include <cstddef>
#include <ctime>
#include <string>
#include <sstream>

// DUNE headers.
#include <DUNE/DUNE.hpp>
```

```
// Local headers.
#include "Reader.hpp"

namespace Sensors
{
  //! Device driver for TBR700RT
  namespace TBR700RT
  {
    using DUNE_NAMESPACES;

    //! Maximum number of initialization commands.
    static const unsigned c_max_init_cmds = 14;
    //! Timeout for waitReply() function.
    static const float c_wait_reply_tout = 4.0;
    //! Power on delay.
    static const double c_pwr_on_delay = 5.0;
    //! Number of fields in fish tag message
    static const unsigned c_tag_fields = 9;
    //! Number of fields in TBR-700RT sensor reading
    static const unsigned c_tbr_sensor_fields = 8;
    struct Arguments
    {
      //! Serial port device.
      std::string uart_dev;
      //! Serial port baud rate.
      unsigned uart_baud;
      //! Order of sentences.
      std::vector<std::string> stn_order;
      //! Input timeout in seconds.
      float inp_tout;
      //! Initialization commands.
      std::string init_cmds[c_max_init_cmds];
      //! Initialization replies.
      std::string init_rpls[c_max_init_cmds];
      //! Power channels.
      std::vector<std::string> pwr_channels;
    };

    struct Task: public Tasks::Task
    {
      //! Serial port handle.
      IO::Handle* m_handle;
      //! Task arguments.
      Arguments m_args;
      //! Last initialization line read.
      std::string m_init_line;
      //! TBRReader thread.
      TBRReader* m_TBRReader;

      Task(const std::string& name, Tasks::Context& ctx):
        Tasks::Task(name, ctx),
        m_handle(NULL),
        m_TBRReader(NULL)
      {
        // Define configuration parameters.
        param("Serial Port - Device", m_args.uart_dev)
        .defaultValue("")
        .description("Serial port device used to communicate with the sensor");

        param("Serial Port - Baud Rate", m_args.uart_baud)
        .defaultValue("4800")
        .description("Serial port baud rate");

        param("Input Timeout", m_args.inp_tout)
        .units(Units::Second)
        .defaultValue("4.0")
        .minimumValue("0.0")
        .description("Input timeout");

        param("Power Channel - Names", m_args.pwr_channels)
        .defaultValue("")
        .description("Device's power channels");

        param("Sentence Order", m_args.stn_order)
        .defaultValue("")
        .description("Sentence order");

        for (unsigned i = 0; i < c_max_init_cmds; ++i)
        {
          std::string cmd_label = String::str("Initialization String %u - Command", i);
```

```
      param(cmd_label, m_args.init_cmds[i])
      .defaultValue("");

      std::string rpl_label = String::str("Initialization String %u - Reply", i);
      param(rpl_label, m_args.init_rpls[i])
      .defaultValue("");
    }

    bind<IMC::DevDataText>(this);
    bind<IMC::IoEvent>(this);
  }

  void
  onResourceAcquisition(void)
  {
    if (m_args.pwr_channels.size() > 0)
    {
      IMC::PowerChannelControl pcc;
      pcc.op = IMC::PowerChannelControl::PCC_OP_TURN_ON;
      for (size_t i = 0; i < m_args.pwr_channels.size(); ++i)
      {
        pcc.name = m_args.pwr_channels[i];
        dispatch(pcc);
      }
    }

    Counter<double> timer(c_pwr_on_delay);
    while (!stopping() && !timer.overflow())
      waitForMessages(timer.getRemaining());

    try
    {
      if (!openSocket())
        m_handle = new SerialPort(m_args.uart_dev, m_args.uart_baud);

      m_TBRReader = new TBRReader(this, m_handle);
      m_TBRReader->start();
    }
    catch (...)
    {
      throw RestartNeeded(DTR("1"), 5);
    }
  }

  bool
  openSocket(void)
  {
    char addr[128] = {0};
    unsigned port = 0;

    if (std::sscanf(m_args.uart_dev.c_str(), "tcp://%[^:]:%u", addr, &port) != 2)
      return false;

    TCPSocket* sock = new TCPSocket;
    sock->connect(addr, port);
    m_handle = sock;
    return true;
  }

  void
  onResourceRelease(void)
  {
    if (m_TBRReader != NULL)
    {
      m_TBRReader->stopAndJoin();
      delete m_TBRReader;
      m_TBRReader = NULL;
    }

    Memory::clear(m_handle);
  }

  void
  onResourceInitialization(void)
  {
    for (unsigned i = 0; i < c_max_init_cmds; ++i)
    {
      if (m_args.init_cmds[i].empty())
        continue;

      std::string cmd = String::unescape(m_args.init_cmds[i]);
```

```
    m_handle->writeString(cmd.c_str());

    if (!m_args.init_rpls[i].empty())
    {
      std::string rpl = String::unescape(m_args.init_rpls[i]);
      if (!waitInitReply(rpl))
      {
        err("%s: %s", DTR("no reply to command"), m_args.init_cmds[i].c_str());
        throw std::runtime_error(DTR("failed to setup device"));
      }
    }
  }

  sendTbrClockSync();

  setEntityState(IMC::EntityState::ESTA_NORMAL, Status::CODE_ACTIVE);
}

void
consume(const IMC::DevDataText* msg)
{
  if (msg->getDestination() != getSystemId())
    return;

  if (msg->getDestinationEntity() != getEntityId())
    return;

  spew("%s", sanitize(msg->value).c_str());

  if (getEntityState() == IMC::EntityState::ESTA_BOOT)
    m_init_line = msg->value;
  else
    processSentence(msg->value);
}

void
consume(const IMC::IoEvent* msg)
{
  if (msg->getDestination() != getSystemId())
    return;

  if (msg->getDestinationEntity() != getEntityId())
    return;

  if (msg->type == IMC::IoEvent::IOV_TYPE_INPUT_ERROR)
    throw RestartNeeded(msg->error, 5);
}

//! Wait reply to initialization command.
//! @param[in] stn string to compare.
//! @return true on successful match, false otherwise.
bool
waitInitReply(const std::string& stn)
{
  Counter<float> counter(c_wait_reply_tout);
  while (!stopping() && !counter.overflow())
  {
    waitForMessages(counter.getRemaining());
    if (m_init_line == stn)
    {
      m_init_line.clear();
      return true;
    }
  }

  return false;
}

int calcLuhnVerifDigit(const char *number)
{
    int i, sum, ch, num, twoup, len;

    len = std::strlen(number);
    sum = 0;
    twoup = 1;
    for (i = len - 1; i >= 0; --i) {
        ch = number[i];
        num = (ch >= '0' && ch <= '9') ? ch - '0' : 0;
        if (twoup) {
            num += num;
            if (num > 9) num = (num % 10) + 1;
```

```
        }
        sum += num;
        twoup = ++twoup & 1;
    }
    sum = 10 - (sum % 10);
    if (sum == 10) sum = 0;
    return sum;
}

void sendTbrClockSync() {
  // Get timestamp from system clock
  std::stringstream ss;
  ss << std::time(0);
  std::string UTCUnixTimestamp = ss.str();

  // Remove last digit
  UTCUnixTimestamp = UTCUnixTimestamp.substr(0,UTCUnixTimestamp.length()-1);

  // Add Luhn verification number
  UTCUnixTimestamp += std::to_string(calcLuhnVerifDigit(UTCUnixTimestamp.c_str()));

  // Add preamble
  std::string cmd = "(+)" + UTCUnixTimestamp;

  // Send sync signal slowly, because TBR700RT can't handle the speed(max 1 char per microsecond)
  char a[1] = {'0'};
  for(char& c : cmd) {
      a[0] = c;
      m_handle->write(a, 1);
      Delay::waitMsec(1);
  }

  spew(DTR("Send: %s"), cmd.c_str());
}

//! Read int from input string.
//! @param[in] str input string.
//! @param[out] dst number.
//! @return true if successful, false otherwise.
bool readIntFromString(const std::string& str, int& dst) {
  try {
    dst = std::stoi(str);
    return true;
  }
  catch (const std::invalid_argument& ia) {
    err(DTR("Invalid argument: %s"), ia.what());
    return false;
  }
  return true;
}

//! Process sentence.
//! @param[in] line line.
void
processSentence(const std::string& line)
{
  spew(DTR("Process"));
  if (line.find("ack01") != std::string::npos) {
    spew(DTR("Sensor clock diciplined"));
  } if(line.find("ack02") != std::string::npos) {
    spew(DTR("Sensor timestamp set"));
  } if (line.find("$") != std::string::npos) {

    // Discard leading noise.
    size_t sidx = 0;
    for (sidx = 0; sidx < line.size(); ++sidx)
    {
      if (line[sidx] == '$')
        break;
    }

    // Split sentence
    std::vector<std::string> parts;
    try {
      spew(DTR("try"));
      String::split(line.substr(sidx + 1, line.size()), ",", parts);
    } catch(const std::exception& ex) {
      err(DTR("Invalid argument: %s"), ex.what());
      return;
    }
```

```
    interpretSentence(parts);
  }
}

//! Interpret given sentence.
//! @param[in] parts vector of strings from sentence.
void
interpretSentence(std::vector<std::string>& parts)
{
  spew(DTR("Interpret"));
  /*if (parts[0] == m_args.stn_order.front())
  {
    // Test if all sentences received, TODO, can probably be removed
  }*/

  if(parts.size() >= 3) {
    if(parts[2]  == "TBR Sensor") {
      interpretSensorReading(parts);
    } else {
      interpretTagDetection(parts);
    }
  }
}
//! Interpret SensorReading sentence.
//! @param[in] parts vector of strings from sentence.
void
interpretSensorReading(const std::vector<std::string>& parts) {
  spew(DTR("Interpret SensorReading"));
  if (parts.size() < c_tbr_sensor_fields)
  {
    war(DTR("invalid SensorReading sentence"));
    return;
  }

  int serial_no = 0;
  int unix_timestamp = 0;
  int temperature = 0;
  int avg_noise_level = 0;
  int peak_noise_level = 0;
  int recv_listen_freq = 0;
  int recv_mem_addr = 0;
  float temp_C = 0.0;

  if (readIntFromString(parts[0], serial_no))
  {
    // Receiver serial number
    spew(DTR("Serial number: %u"), serial_no);
  }
  if (readIntFromString(parts[1], unix_timestamp))
  {
    //UTC UNIX timestamp
    spew(DTR("UTC UNIX timestamp: %u"), unix_timestamp);
  }
  if (readIntFromString(parts[3], temperature))
  {
    // Temperature
    temp_C = float(temperature-50)/10.0;
    spew(DTR("Temperature(C): %f"), temp_C);
    IMC::Temperature temp_msg;
    temp_msg.setSourceEntity(255);
    temp_msg.value = fp32_t(temp_C);
    dispatch(temp_msg);
  }
  if (readIntFromString(parts[4], avg_noise_level))
  {
    // Average Noise Level
    spew(DTR("Average Noise Level: %u"), avg_noise_level);
  }
  if (readIntFromString(parts[5], peak_noise_level))
  {
    // Peak noise level
    spew(DTR("Peak noise level: %u"), peak_noise_level);
  }
  if (readIntFromString(parts[6], recv_listen_freq))
  {
    // Noise logging frequency
    spew(DTR("Noise logging frequency: %u"), recv_listen_freq);
  }
  if (readIntFromString(parts[7], recv_mem_addr))
  {
    // Receiver memory address
```

```
    spew(DTR("Receiver memory address: %u"), recv_mem_addr);
  }
  IMC::TBRSensor sensor_msg;
  sensor_msg.serial_no = serial_no;
  sensor_msg.unix_timestamp = unix_timestamp;
  sensor_msg.temperature = fp32_t(temp_C);
  sensor_msg.avg_noise_level = avg_noise_level;
  sensor_msg.peak_noise_level = peak_noise_level;
  sensor_msg.recv_listen_freq = recv_listen_freq;
  sensor_msg.recv_mem_addr = recv_mem_addr;
  dispatch(sensor_msg);
}
//! Interpret fishtag sentence.
//! @param[in] parts vector of strings from sentence.
void
interpretTagDetection(const std::vector<std::string>& parts)
{
  spew(DTR("Interpret tag"));
  if (parts.size() < c_tag_fields)
  {
    war(DTR("invalid tag sentence"));
    return;
  }

  int serial_no = 0;
  int unix_timestamp = 0;
  int millis = 0;
  int trans_protocol = 0;
  int trans_id = 0;
  int trans_data = 0;
  int SNR = 0;
  int trans_freq = 0;
  int recv_mem_addr = 0;

  if (readIntFromString(parts[0], serial_no))
  {
    // Receiver serial number
    spew(DTR("Serial number: %u"), serial_no);
  }
  if (readIntFromString(parts[1], unix_timestamp))
  {
    //UTC UNIX timestamp
    spew(DTR("UTC UNIX timestamp: %u"), unix_timestamp);
  }
  if (readIntFromString(parts[2], millis))
  {
    //Millisecond timestamp
    spew(DTR("Millisecond timestamp: %u"), millis);
  }

  //Transmit protocol
  if(parts[3] == "R256")
    trans_protocol = IMC::TBRFishTag::TBR_R256;
  else if(parts[3] == "R04K")
    trans_protocol = IMC::TBRFishTag::TBR_R04K;
  else if(parts[3] == "R06K")
    trans_protocol = IMC::TBRFishTag::TBR_R06K;
  else if(parts[3] == "R64K")
    trans_protocol = IMC::TBRFishTag::TBR_R64K;
  else if(parts[3] == "R01M")
    trans_protocol = IMC::TBRFishTag::TBR_R01M;
  else if(parts[3] == "S256")
    trans_protocol = IMC::TBRFishTag::TBR_S256;
  else if(parts[3] == "HS256")
    trans_protocol = IMC::TBRFishTag::TBR_HS256;
  else if(parts[3] == "DS256")
    trans_protocol = IMC::TBRFishTag::TBR_DS256;
  spew(DTR("Transmit protocol: %s, enum: %i"), parts[3].c_str(), trans_protocol);


  if (readIntFromString(parts[4], trans_id))
  {
    // Tag ID number
    spew(DTR("Tag ID: %u"), trans_id);
  }
  if (readIntFromString(parts[5], trans_data))
  {
    // Tag raw data
    spew(DTR("Tag raw data: %u"), trans_data);
  }
  if (readIntFromString(parts[6], SNR))
```

```
        {
          // Signal to noise ratio
          spew(DTR("SNR: %u"), SNR);
        }
        if (readIntFromString(parts[7], trans_freq))
        {
          // Signal frequency
          spew(DTR("Signal frequency: %u"), trans_freq);
        }
        if (readIntFromString(parts[8], recv_mem_addr))
        {
          // Receiver memory address
          spew(DTR("Receiver memory address: %u"), recv_mem_addr);
        }
        IMC::TBRFishTag tag_msg;
        tag_msg.serial_no = serial_no;
        tag_msg.unix_timestamp = unix_timestamp;
        tag_msg.millis = millis;
        tag_msg.trans_protocol = trans_protocol;
        tag_msg.trans_id = trans_id;
        tag_msg.trans_data = trans_data;
        tag_msg.snr = SNR;
        tag_msg.trans_freq = trans_freq;
        tag_msg.recv_mem_addr = recv_mem_addr;
        dispatch(tag_msg);
      }
      void
      onMain(void)
      {
        while (!stopping())
        {
          waitForMessages(1.0);
          //sendTbrClockSync();
        }
      }
    };
  }
}
```

DUNE_TASK

B.6.1 src/Sensors/TBR700RT/Reader.cpp

```
//***************************************************************************
// Copyright 2007-2019 Universidade do Porto - Faculdade de Engenharia      *
// LaboratÃ³rio de Sistemas e Tecnologia SubaquÃ¡tica (LSTS)                  *
//***************************************************************************
// This file is part of DUNE: Unified Navigation Environment.               *
//                                                                          *
// Commercial Licence Usage                                                 *
// Licencees holding valid commercial DUNE licences may use this file in    *
// accordance with the commercial licence agreement provided with the       *
// Software or, alternatively, in accordance with the terms contained in a  *
// written agreement between you and Faculdade de Engenharia da             *
// Universidade do Porto. For licensing terms, conditions, and further      *
// information contact lsts@fe.up.pt.                                       *
//                                                                          *
// Modified European Union Public Licence - EUPL v.1.1 Usage                *
// Alternatively, this file may be used under the terms of the Modified     *
// EUPL, Version 1.1 only (the "Licence"), appearing in the file LICENCE.md *
// included in the packaging of this file. You may not use this work        *
// except in compliance with the Licence. Unless required by applicable     *
// law or agreed to in writing, software distributed under the Licence is   *
// distributed on an "AS IS" basis, WITHOUT WARRANTIES OR CONDITIONS OF      *
// ANY KIND, either express or implied. See the Licence for the specific    *
// language governing permissions and limitations at                        *
// https://github.com/LSTS/dune/blob/master/LICENCE.md and                  *
// http://ec.europa.eu/idabc/eupl.html.                                     *
//***************************************************************************
// Author: Ricardo Martins (sligthly modified by Nikolai LauvÃ«s)            *
//***************************************************************************

#ifndef SENSORS_TBR700RT_TBRReader_HPP_INCLUDED_
#define SENSORS_TBR700RT_TBRReader_HPP_INCLUDED_

// DUNE headers.
#include <DUNE/DUNE.hpp>

namespace Sensors
{
```

```
namespace TBR700RT
{
  using DUNE_NAMESPACES;

  //! Read buffer size.
  static const size_t c_read_buffer_size = 4096;
  //! Line termination character.
  static const char c_line_term = '\r';

  class TBRReader: public Concurrency::Thread
  {
  public:
    //! Constructor.
    //! @param[in] task parent task.
    //! @param[in] handle I/O handle.
    TBRReader(Tasks::Task* task, IO::Handle* handle):
      m_task(task),
      m_handle(handle)
    {
      m_buffer.resize(c_read_buffer_size);
    }

  private:
    //! Parent task.
    Tasks::Task* m_task;
    //! I/O handle.
    IO::Handle* m_handle;
    //! Internal read buffer.
    std::vector<char> m_buffer;
    //! Current line.
    std::string m_line;

    void
    dispatch(IMC::Message& msg)
    {
      msg.setDestination(m_task->getSystemId());
      msg.setDestinationEntity(m_task->getEntityId());
      m_task->dispatch(msg, DF_LOOP_BACK);
    }

    void
    read(void)
    {
      if (!Poll::poll(*m_handle, 1.0))
        return;

      size_t rv = m_handle->read(&m_buffer[0], m_buffer.size());
      if (rv == 0)
        throw std::runtime_error(DTR("invalid read size"));

      for (size_t i = 0; i < rv; ++i)
      {
        m_line.push_back(m_buffer[i]);
        if (m_buffer[i] == c_line_term)
        {
          IMC::DevDataText line;
          line.value = m_line;
          dispatch(line);
          m_line.clear();
        }
      }
    }

    void
    run(void)
    {
      while (!isStopping())
      {
        try
        {
          read();
        }
        catch (std::runtime_error& e)
        {
          IMC::IoEvent evt;
          evt.type = IMC::IoEvent::IOV_TYPE_INPUT_ERROR;
          evt.error = e.what();
          dispatch(evt);
          break;
        }
      }
```

```
      }
    };
  }
}

#endif
```

B.7 src/DUNE/Hardware/

B.7.1 SocketCAN.cpp

```
//***************************************************************************
// Copyright 2013-2019 Norwegian University of Science and Technology (NTNU)*
// Department of Engineering Cybernetics (ITK)                              *
//***************************************************************************
// This file is part of DUNE: Unified Navigation Environment.               *
//                                                                          *
// Commercial Licence Usage                                                 *
// Licencees holding valid commercial DUNE licences may use this file in    *
// accordance with the commercial licence agreement provided with the       *
// Software or, alternatively, in accordance with the terms contained in a  *
// written agreement between you and Faculdade de Engenharia da             *
// Universidade do Porto. For licensing terms, conditions, and further      *
// information contact lsts@fe.up.pt.                                       *
//                                                                          *
// Modified European Union Public Licence - EUPL v.1.1 Usage                *
// Alternatively, this file may be used under the terms of the Modified     *
// EUPL, Version 1.1 only (the "Licence"), appearing in the file LICENCE.md *
// included in the packaging of this file. You may not use this work        *
// except in compliance with the Licence. Unless required by applicable     *
// law or agreed to in writing, software distributed under the Licence is   *
// distributed on an "AS IS" basis, WITHOUT WARRANTIES OR CONDITIONS OF      *
// ANY KIND, either express or implied. See the Licence for the specific    *
// language governing permissions and limitations at                        *
// https://github.com/LSTS/dune/blob/master/LICENCE.md and                  *
// http://ec.europa.eu/idabc/eupl.html.                                     *
//***************************************************************************
// Author: Nikolai LauvÃěs                                                  *
//***************************************************************************

// ISO C++ 98 headers.
#include <string>
#include <sstream>
#include <cstring>
#include <stdexcept>
#include <iomanip>

// DUNE headers.
#include <DUNE/System/Error.hpp>
#include <DUNE/Hardware/SocketCAN.hpp>

#if defined(DUNE_OS_LINUX)
// CAN interface haders.
#include <linux/can.h>
#include <linux/can/raw.h>
#include <net/if.h>
#include <sys/ioctl.h>

#include <termios.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#else
      throw Error("unimplemented feature", "DUNE::Hardware::SocketCAN");
#endif

#define CAN_SFF_MASK 0x000007FFU // standard frame format (SFF)
#define CAN_EFF_MASK 0x1FFFFFFFU // extended frame format (EFF)

namespace DUNE
{
  namespace Hardware
  {
#if defined(DUNE_OS_LINUX)
    SocketCAN::SocketCAN(const std::string& can_dev, can_frame_t frame_type)
    {
      can_frame_type = frame_type;
      m_can_socket = ::socket(PF_CAN, SOCK_RAW, CAN_RAW);
```

```
if (m_can_socket < 0) {
      throw Error("Error while opening socket for CANbus", System::Error::getLastMessage()); //TODO: Check
  }

  int enable, rc;
  switch(can_frame_type) {
    case CAN_BASIC_SFF:
        break;
        case CAN_BASIC_EFF:
    break;
    case CAN_FD:
      enable = 1;
      rc = ::setsockopt(m_can_socket, SOL_CAN_RAW, CAN_RAW_FD_FRAMES, &enable, sizeof(enable));
      if (rc == -1)
        throw Error("Failed to enable FD frames", System::Error::getLastMessage()); //TODO: Check
    break;
    default:
      throw Error("Frame type not recognized", System::Error::getLastMessage());
  }

  std::strncpy(m_ifr.ifr_name, can_dev.c_str(), IFNAMSIZ);


  if(::ioctl(m_can_socket, SIOCGIFFLAGS, &m_ifr) < 0) {
      throw Error("Could not read SIOCGIFFLAGS with ioctl", System::Error::getLastMessage());
  }
  if ( !(m_ifr.ifr_flags & IFF_UP) ) {
      throw Error("CAN network is down", System::Error::getLastMessage());
  }

  // Get the index of the network interface
  if (::ioctl(m_can_socket, SIOCGIFINDEX, &m_ifr) == -1)
    throw Error("Coult not get interface index with ioctl", System::Error::getLastMessage());

  // Bind the socket to the network interface
  m_addr.can_family = AF_CAN;
  m_addr.can_ifindex = m_ifr.ifr_ifindex;
  rc = ::bind(m_can_socket, reinterpret_cast<struct sockaddr*>(&m_addr), sizeof(m_addr));

  if (rc == -1)
    throw Error("Could not bind CAN socket", System::Error::getLastMessage()); //TODO: Check
}

//! Serial port destructor.
SocketCAN::~SocketCAN(void)
{
  if (::close(m_can_socket) == -1)
    throw Error("Could not close CAN port", System::Error::getLastMessage()); //TODO: Check
}

void SocketCAN::setTXID(uint32_t id) {
    cantxid = id | CAN_EFF_FLAG; // TODO: Check if similar for SFF(CAN_SFF_FLAG does not exist)
}
uint32_t SocketCAN::getRXID() {
  switch(can_frame_type) {
    case CAN_BASIC_SFF:
      return canrxid & CAN_SFF_MASK; // TODO: Check for SFF
    break;
    case CAN_BASIC_EFF:
      return canrxid & CAN_EFF_MASK;
    break;
    case CAN_FD:
      return canrxid & CAN_EFF_MASK;
    break;
    default:
      throw Error("Frame type not recognized", System::Error::getLastMessage());
  }
  return 0;
}

size_t SocketCAN::readHexString(char* bfr, size_t length) {
    size_t readSize = readString(bfr, length);
    std::stringstream ss;
    ss << std::internal // fill between the prefix and the number
    << std::setfill('0') << std::uppercase; // fill with 0s
    ss << std::hex << std::setw(8) << int(getRXID()) << "#";

    for(size_t i=0; i < readSize; i++) {
      ss << std::hex << std::setw(2) << int(bfr[i]);
    }
```

```
    std::string out = ss.str();
    strncpy(bfr, out.c_str(), out.length()+1); // +1 because of '\0 added in c_str'

  return out.length()+1;
}

size_t
SocketCAN::doWrite(const uint8_t* bfr, size_t size) { // TODO: Add exceptions
  int writtenBytes;
  switch(can_frame_type) {
    case CAN_BASIC_SFF:
    case CAN_BASIC_EFF:
      struct can_frame frame;
      frame.can_dlc = size;
      frame.can_id = cantxid;
      memcpy(frame.data, bfr, size);
      writtenBytes = ::write(m_can_socket, &frame, CAN_MTU);
    break;
    case CAN_FD:
      struct canfd_frame fdframe;
      fdframe.len = size;
      fdframe.can_id = cantxid;
      memcpy(fdframe.data, bfr, size);
      writtenBytes = ::write(m_can_socket, &fdframe, CANFD_MTU);
    break;
    default:
      throw Error("Frame type not recognized", System::Error::getLastMessage());
  }
  return size;
}

size_t
SocketCAN::doRead(uint8_t* bfr, size_t size) { //TODO: Add timeout
  int numBytes;
  switch(can_frame_type) {
    case CAN_BASIC_EFF:
    case CAN_BASIC_SFF:
      struct can_frame frame;
      numBytes = ::read(m_can_socket, &frame, CAN_MTU);
      if(numBytes) {
        for(uint8_t i=0; i<frame.can_dlc && i<size; i++) {
          bfr[i] = frame.data[i];
        }
        canrxid = frame.can_id;
        return frame.can_dlc;
      }
    break;
    case CAN_FD:
      struct canfd_frame fdframe;
      numBytes = ::read(m_can_socket, &fdframe, CANFD_MTU);
      if(numBytes) {
        for(uint8_t i=0; i<fdframe.len && i<size; i++) {
          bfr[i] = fdframe.data[i];
        }
        canrxid = fdframe.can_id;
        return fdframe.len;
      }
    break;
    default:
      throw Error("Frame type not recognized", System::Error::getLastMessage());
  }
  return 0; //Should never be reached
}

//! Flush input buffer, discarding all of it's contents.
void
SocketCAN::doFlushInput(void) {
    tcflush(m_can_socket, TCIFLUSH); //Probably does not work, untested
}

//! Flush output buffer, aborting output.
void
SocketCAN::doFlushOutput(void) {
    tcflush(m_can_socket, TCOFLUSH); //Probably does not work, untested
}

//! Flush both input and output buffers.
void
SocketCAN::doFlush(void) {
    tcflush(m_can_socket, TCIOFLUSH); //Probably does not work, untested
}
```

```
#else
    throw Error("unimplemented feature", "DUNE::Hardware::SocketCAN");
#endif
  }
}
```

B.7.2 SocketCAN.hpp

```
//***************************************************************************
// Copyright 2013-2019 Norwegian University of Science and Technology (NTNU)*
// Department of Engineering Cybernetics (ITK)                              *
//***************************************************************************
// This file is part of DUNE: Unified Navigation Environment.               *
//                                                                          *
// Commercial Licence Usage                                                 *
// Licencees holding valid commercial DUNE licences may use this file in    *
// accordance with the commercial licence agreement provided with the       *
// Software or, alternatively, in accordance with the terms contained in a  *
// written agreement between you and Faculdade de Engenharia da             *
// Universidade do Porto. For licensing terms, conditions, and further      *
// information contact lsts@fe.up.pt.                                       *
//                                                                          *
// Modified European Union Public Licence - EUPL v.1.1 Usage                *
// Alternatively, this file may be used under the terms of the Modified     *
// EUPL, Version 1.1 only (the "Licence"), appearing in the file LICENCE.md *
// included in the packaging of this file. You may not use this work        *
// except in compliance with the Licence. Unless required by applicable     *
// law or agreed to in writing, software distributed under the Licence is   *
// distributed on an "AS IS" basis, WITHOUT WARRANTIES OR CONDITIONS OF      *
// ANY KIND, either express or implied. See the Licence for the specific    *
// language governing permissions and limitations at                        *
// https://github.com/LSTS/dune/blob/master/LICENCE.md and                  *
// http://ec.europa.eu/idabc/eupl.html.                                     *
//***************************************************************************
// Author: Nikolai LauvÃěs                                                  *
//***************************************************************************

#ifndef DUNE_HARDWARE_SOCKETCAN_HPP_INCLUDED_
#define DUNE_HARDWARE_SOCKETCAN_HPP_INCLUDED_
/* TODO/missing functionality:
 * Research flush functions

*/
// DUNE headers.
#include <DUNE/Config.hpp>
#include <DUNE/IO/Handle.hpp>

#if defined(DUNE_OS_LINUX)
#include <linux/can.h>
#include <net/if.h>
#else
      throw Error("Unimplemented feature", "DUNE::Hardware::SocketCAN");
#endif


namespace DUNE
{
  namespace Hardware
  {
    // Export DLL Symbol.
    class DUNE_DLL_SYM SocketCAN;

    //! The SocketCAN class encapsulates CAN access.
    class SocketCAN: public IO::Handle
    {
    public:
      class Error: public std::runtime_error
      {
      public:
        Error(std::string op, std::string msg):
          std::runtime_error("Socket CAN error (" + op + "): " + msg)
        { }
      };
#if defined(DUNE_OS_LINUX)
      enum can_frame_t {
        CAN_BASIC_SFF = 0,
        CAN_BASIC_EFF = 1,
        CAN_FD = 2
      };
      //! SocketCAN constructor.
```

```
      SocketCAN(const std::string& can_dev, can_frame_t frame_type);

      //! Socket CAN destructor.
      ~SocketCAN(void);

      void setTXID(uint32_t id);
      uint32_t getRXID();
      size_t readHexString(char* bfr, size_t length);
      //size_t writeHexString(const char* cstr);
    private:
      //! CAN connection variables.
      struct sockaddr_can m_addr;
      struct ifreq m_ifr;
      int m_can_socket;
      can_frame_t can_frame_type;
      uint32_t cantxid = 0;
      uint32_t canrxid = 0;

      IO::NativeHandle
      doGetNative(void) const
      {
        return m_can_socket; // Makes Poll::poll work
      }

      size_t
      doWrite(const uint8_t* bfr, size_t size);

      size_t
      doRead(uint8_t* bfr, size_t size);

      //! Flush input buffer, discarding all of it's contents.
      void
      doFlushInput(void);

      //! Flush output buffer, aborting output.
      void
      doFlushOutput(void);

      //! Flush both input and output buffers.
      void
      doFlush(void);
#else
      throw Error("Unimplemented feature", "DUNE::Hardware::SocketCAN");
#endif
    };
  }
}


#endif
```