WQ-Sensor Array Software & Hardware notes

*Additional Documentation*

Jurrian Doornbos MSc. Thesis; November 2020

This document covers the hardware parts, assembly, and software inner workings of the sensor array. With this document, future users will be able to service the sensor array, to maintain its functionality or update it.

# 1. Theory of operation

The sensor array is created to measure in-situ water quality across four parameters. The sensors are small in size and inexpensive, and can therefore be fitted on a USV or UAV. Furthermore, a sonar is also housed in this array, for depth information.  
The sensors are physically separated from the electronics by being in a different housing, this ensures that they can always be in direct contact with the water. The Arduino code currently only sends the sensor information over serial, to be used in a broader system, such as a system running on the Robotics Operating System. Which is explained further down when the Raspberry Pi comes into focus. Upgradability of sensor hardware is currently not possible as the housing is made to be as small as possible and fit the sensors exactly.

# 2. Hardware

The hardware is mounted in such a way to protect the components from potential water damage. Also, it needs to be able to have reliable connection to the hull, as well as be able to change monitoring platforms. The sensors all require direct contact with the water, which means two things. First, the sensors should be mounted on a system that can be submerged, or withstand water, the sensors themselves also should be waterproof. Second, the electronics and processing should be done in a separate box, to prevent water entering and interfering with the electronics. This means that two sub-systems make up the sensor-array: a Sensor Box as well as an Electronics Box.

The Electronics Box is an off-the-shelve electronics box with an IP67 rating. To facilitate the various electronics-boards, an extra mounting surface was designed and 3D-printed. On this surface, the 4 different sensor-processing boards are placed, as well as the central processing board, an Arduino Uno, a small, programmable microcontroller. Power is delivered via the USB port, distributed via a simple distribution board with a 1000uF capacitor to eliminate noise. All connections are made with simple pin-header cables, directly soldered and secured with additional hot-glue. The information from the Arduino gets sent to the Teensy, a second microcontroller, which also receives GPS information from a U-Blox M8N GPS receiver. This information gets sent to the Raspberry Pi, running Ubuntu Linux 20.04, with the Robotics Operating System 2 (ROS2) on it. To handle the incoming data stream from the Teensy, as well as handle the sonar communication.

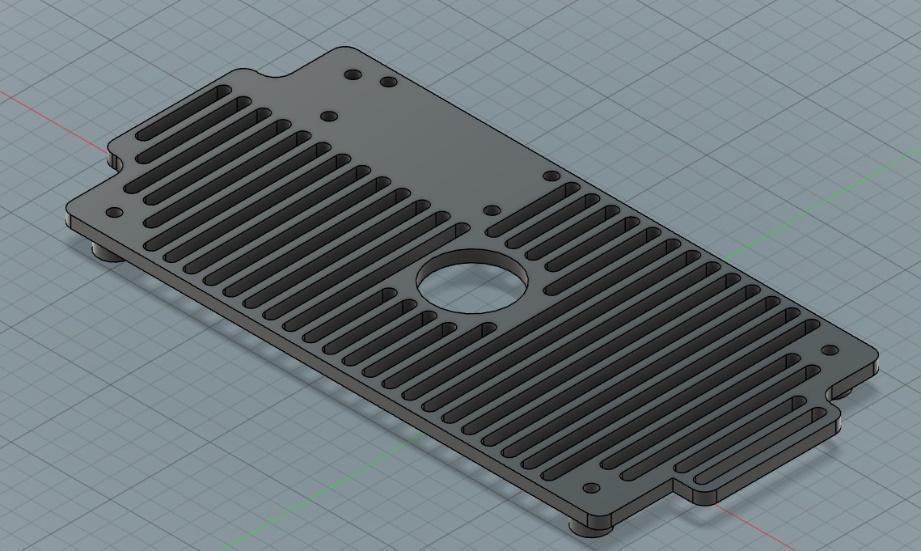


Image 1. CAD of the Electronics platform

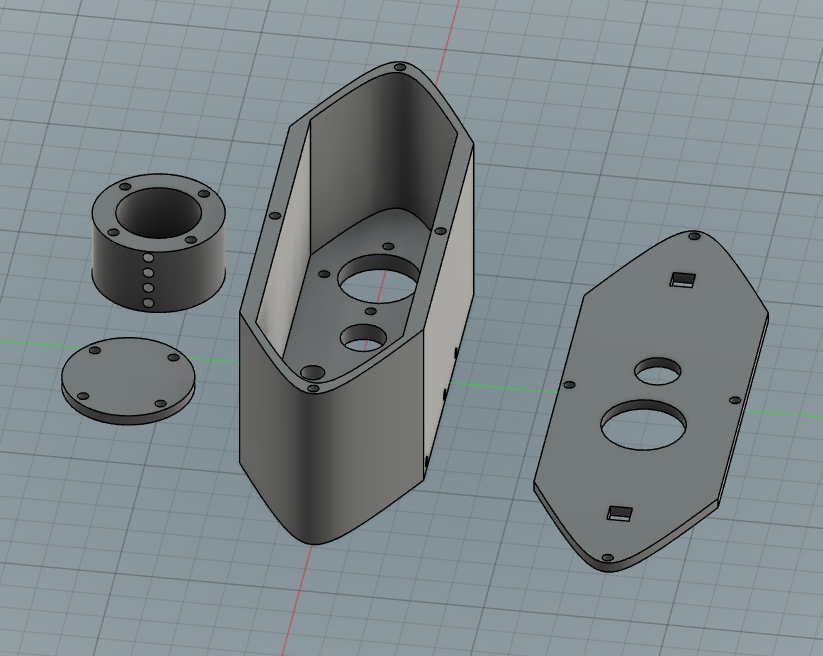
The Sensor Box is a 3D-printed enclosure with holes to fit all the sensors. Important here is the following: the ability to secure the sensors; make sure as little as water as possible enters the enclosure; create a mounting system that can be used on a variety of platforms; connect the sensors to the electronics. With this in mind, a boat-shaped box was created in a CAD environment (Fusion 360), with the proper holes in the bottom to fit all the sensors, as well as side-mounted screw-holes to make sure the sensors stay in place. The lid has holes to ensure the passthrough of the longer pH sensor and the wires. It has square holes to accommodate M6 threaded nuts and bolts for mounting.

Image 2. CAD of the Sensor Box

## 2.1. Parts

Table 1. Parts-list of the Sensor Array

|  |  |  |  |
| --- | --- | --- | --- |
| **Sensor name** | **Measures** | **Separate Board info** | **Quantity** |
| [Analog pH meter](https://www.dfrobot.com/product-1025.html) | Acidity | BNC connection to breakout | 1 |
| [Turbidity Sensor](https://www.dfrobot.com/product-1394.html) | Clarity | 3-pin ILS mini to breakout | 1 |
| [Waterproof DS180 kit](https://www.dfrobot.com/product-1354.html) | Temperature | 3-wire open connection to breakout | 1 |
| [TDS Sensor](https://www.dfrobot.com/product-1662.html) | Hardness | 2-pin ILS mini to breakout | 1 |
| [Kogger Sonar 2D enhanced](https://kogger.tech/product/sonar-2d-enhanced/) | Depth | Built-in board; serial cable | 1 |
| DFRobotics Nano | Arduino Uno | Main control board of the sensors | 1 |
| **Enclosure** | **Material** | **STL name** | **Quantity** |
| Main body hull | PLA | pr\_3D\_main\_hull.stl | 1 |
| Main body lid | PLA | pr\_3D\_main\_lid.stl | 1 |
| Sonar mount | PLA | pr\_3D\_sonar\_mount.stl | 1 |
| Sonar lid | PLA | pr\_3D\_sonar\_lid.stl | 1 |
| [Waterproof Electronics Box](https://www.okaphone.com/artikel.asp?id=481327) | ABS | NaN | 1 |
| Electronics Mount | PLA | pr\_3D\_electronics\_mount.stl | 1 |
| **Additional** | **Dimensions** | **Notes** | **Quantity** |
| Mounting Nut & Bolt | 100mm M6 |  | 2 |
| Sensor hold screw | 40mm M3 |  | 5 |
| Lid screws | 20mm M3 |  | 4 |
| Cable sleeve | 40cm 22mm diameter |  | 1 |
| Cable passthrough | 22mm diameter |  | 2 |
| USB passthrough | 12mm diameter |  | 1 |
| USB cable | 60cm |  | 1 |

## 2.2. Assembly

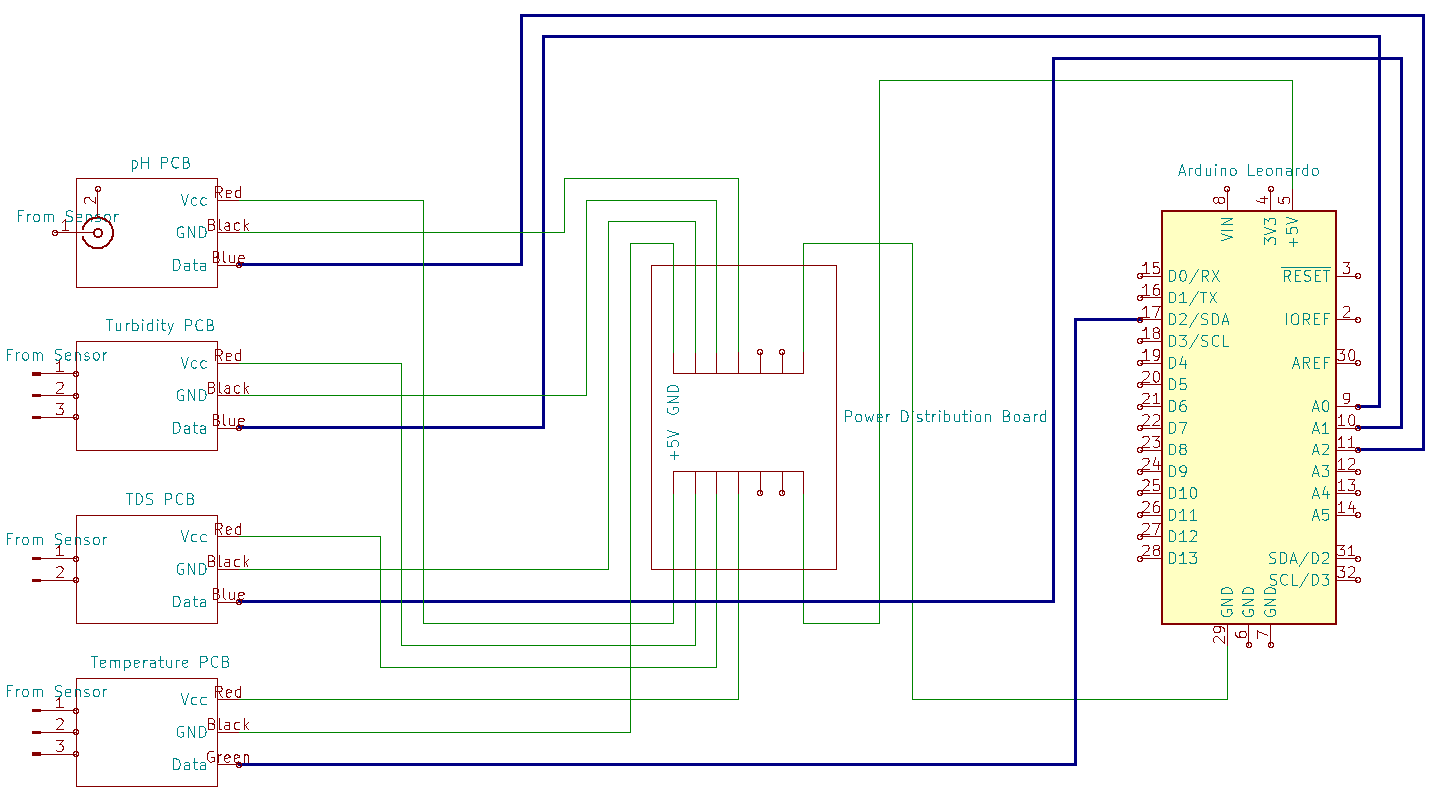
The overall assembly is straightforward. The housings are assembled with M3 screws. The PCBs are connected to the Arduino Uno according to figure 1. The Arduino communicates via a serial connection over the USB port.

Figure 1. Wiring Diagram in the Electronics Box.

### 2.2.3. Electronics Box

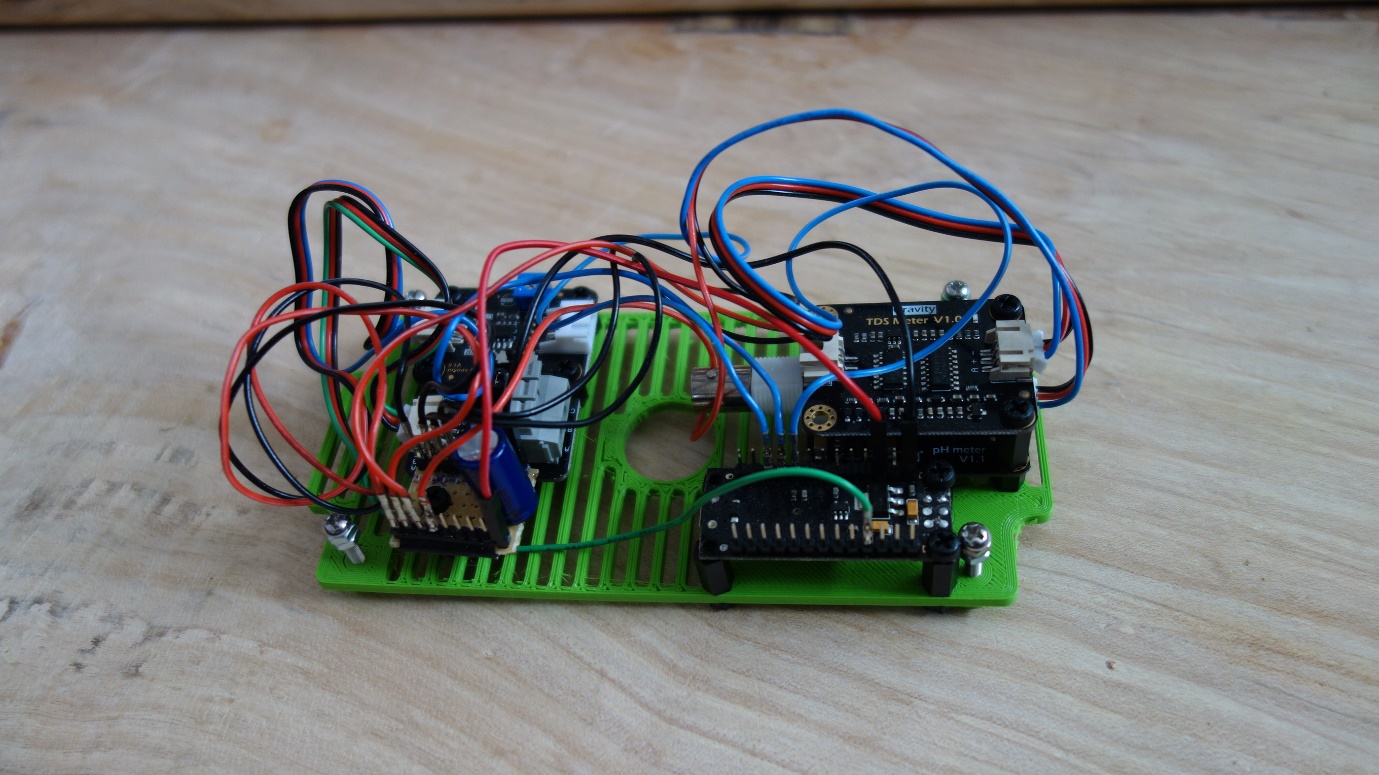
The Electronics Box consists of the waterproof box and the mounting platform. The PCBs are laid out on the platform, in a somewhat efficient manner (image 3), mounted via M3 standoffs. With the easy access, the wires are connected according to figure 1.

Image 3. Electronics Platform.

The platform can now be placed in the box. With the platform in place, screw the platform down in the four corners, ensuring it is firmly secured in the box (image 4). The USB cable can now be connected to the Arduino and routed through the side-hole to an external processing system.

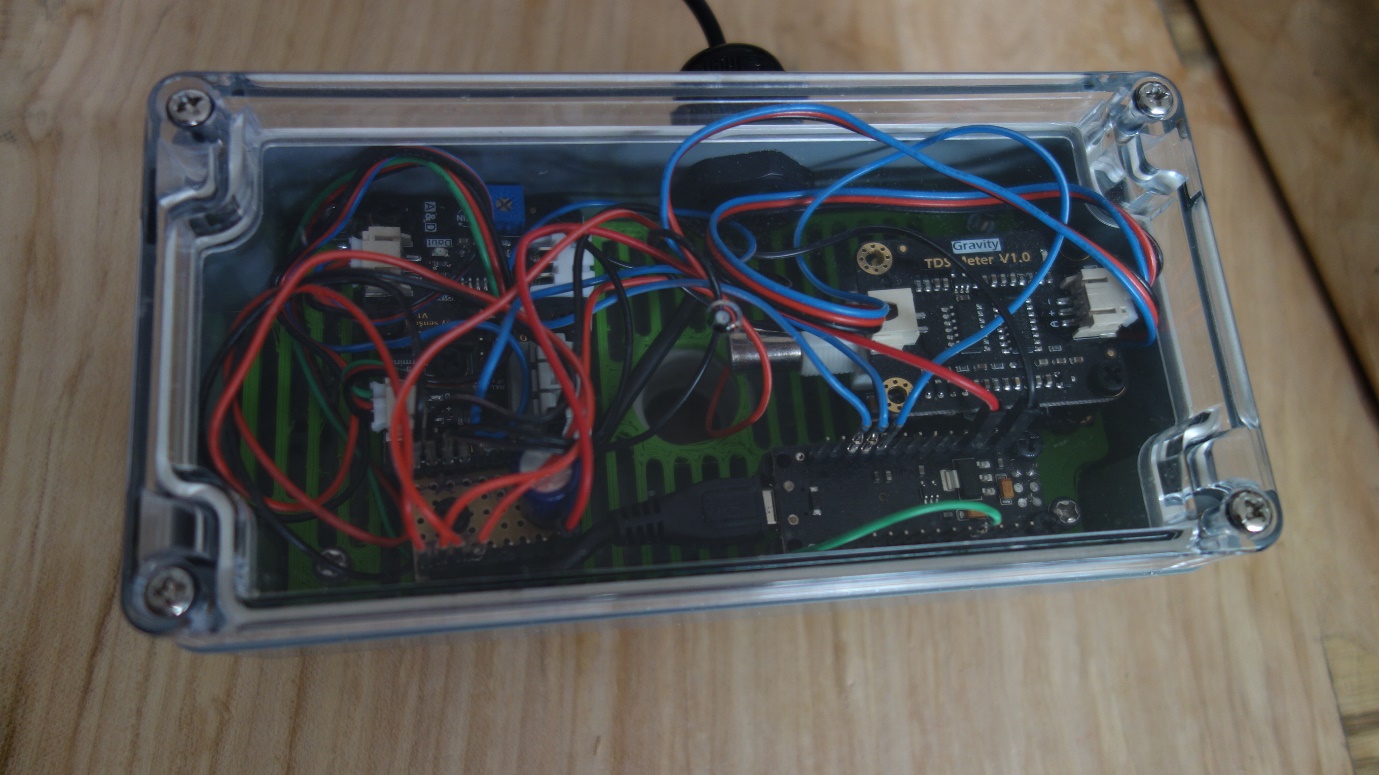
The cabling from the Sensor Box can also be connected to the corresponding PCBs, according to Figure 1. The cables are routed through the hole in the bottom of the box and platform. Both the USB and sensor holes have an additional waterproofing with cable gromets.

Image 4. Closed Electronics Box.

### 2.2.4. Sensor Box

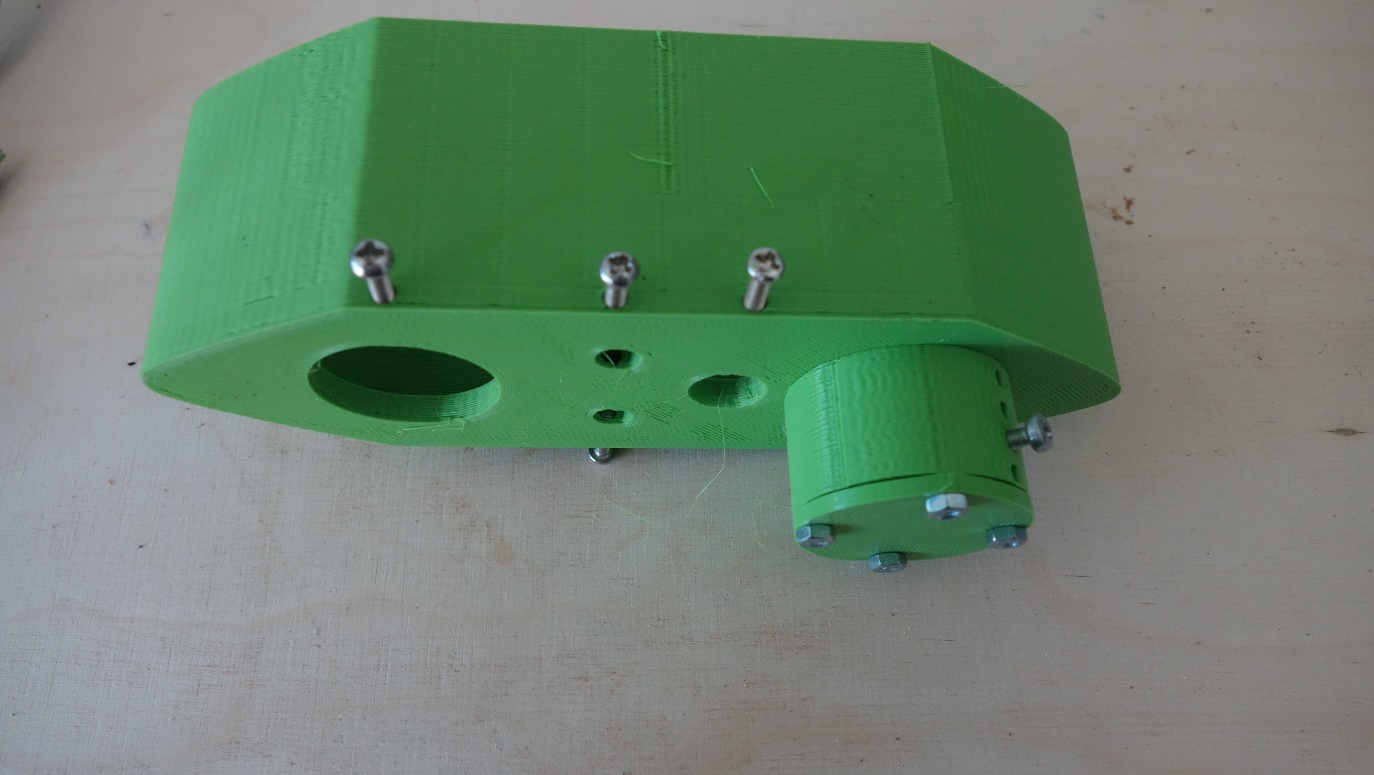
The Sensor Box consists of the hull, lid, sonar mount and sonar lid. All can be joined with M3 screws. Four 40mm M3 bolts are pushed through the bottom of the hull, where the sonar mount fits. The sonar mount is connected to these bolts, and the lid is joined to the mount with four M3 nuts. Furthermore, the sensor set screws can be screwed into the corresponding holes, making sure not to close the holes through which the sensors fit.

Image 5. Empty Sensor Box.

The box is now ready to fit the Turbidity, TDS, and Temperature sensor. The Turbidity fits in the largest hole. With the Turbidity sensor facing north, the left hole is for the TDS sensor, and the temperature on the right. Set the sensors in place by lightly tightening the set screws, until the sensors do not twist anymore. The cables of these sensors can be routed through the gromet at the top of the lid (image 5). The lid in this picture already has the gromet and two bolts mounted.

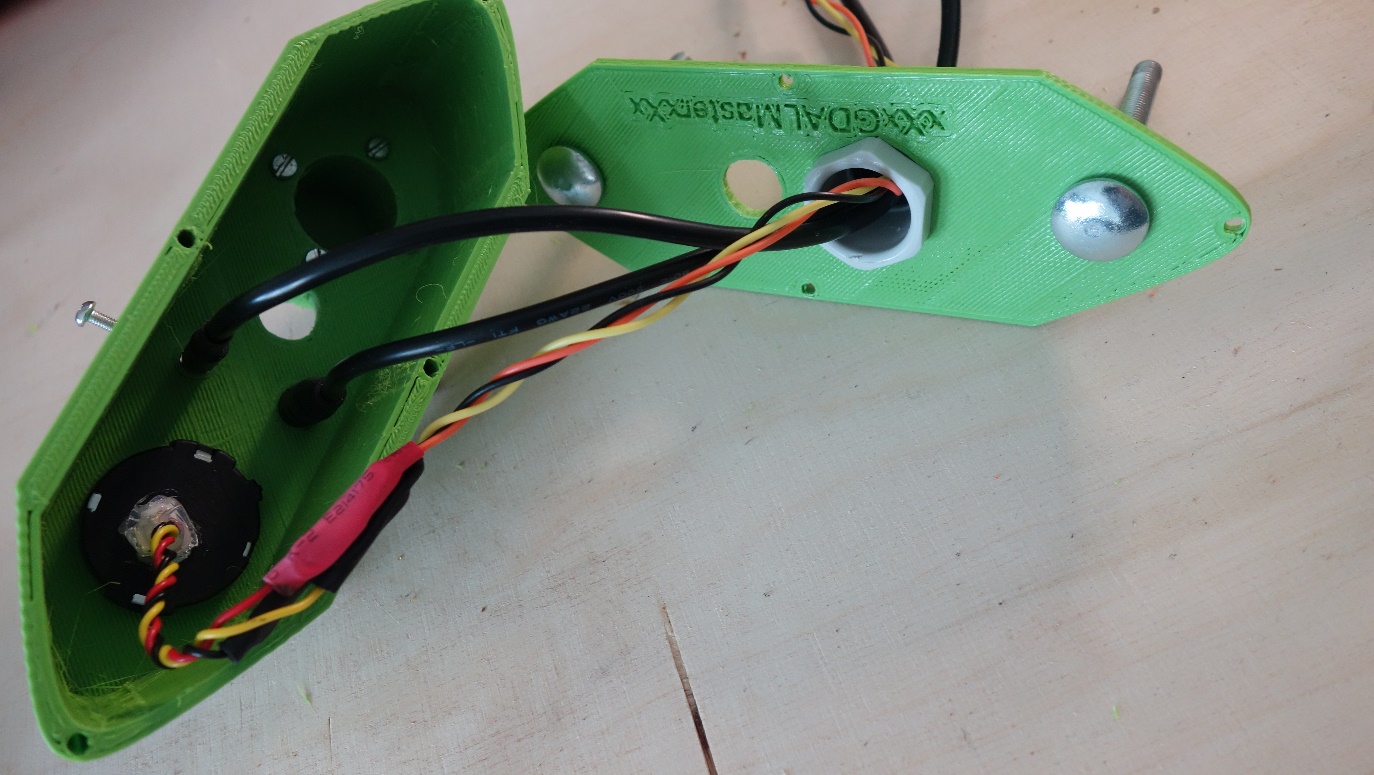
The lid can now be closed by screwing in the four 20mm M3 bolts, note the correspondence of the two pH holes. The pH sensor is added by removing the cap of the probe and pushing it through the two holes, until the point of the probe is of similar depth to the other sensors. The cap can be placed back, to ensure longevity of the pH probe.

Image 6. Sensor Box and Lid.

The sonar can be added by removing the sonar lid, pushing it in from the bottom, ensuring that the sonar can be completely submerged. Place the sonar in the correct place and tighten the set screw. Finally, the nuts should be screwed on the protruding bolts. The sonar cable can now also be routed through the gromet. The sonar works best without the lid, so keep the lid safe until the mission is complete.

All the cables can now be connected to the electronics in the other box.

# 3. Software

## 3.1. Functionality

The purpose of the sensor array is to measure the water quality of a pond whilst mounted on a USV. The hardware ensures that the sensors are in good, reliable, and repeatable contact with the water. They do not do anything yet. They are connected to an Arduino Uno which needs to be programmed to read the sensor values and publish them over a serial USB connection. The source code is found on the wqdrone GitHub. The same counts for the Teensy and ROS nodes.

## 3.2. Arduino code

There are at least a dozen ways to approach the concept of an Arduino reading values and pushing them over the serial port to another machine, all in varying complexity. The simplest way is to directly sense the analog voltage every second and sending that to the USB port. However, noise is easily created by wiggling the cables, or other effects which only last a very short moment.

An Arduino is powerful enough for light on-board processing. The implementation here takes an analog voltage reading and converts it to the correct sensed value (pH, NTU, Celsius, etc.), according to the principals to which these sensors function. It does this every 0.04 seconds and places the values in an array. The mean value of this array is taken every 0.8 seconds and sent over the serial connection.

A few additional libraries have been used to make this work. The OneWire library, which has been specifically created to communicate with the Temperature probe. The probe uses the OneWire technology to send digital information over a single wire.

The company which sells the sensors used in this project have also created a library specifically for the TDS sensor. Which was directly implemented according to their sample code.

The created Arduino package can be found on the Github Repository of the thesis. Below are a few important highlights for the code.

Code Snippet 1. Acquiring Sensor Values

//Temperature sampling, temperature is also used in TDS adjustment

temperature = getTemp();

//TDS sampling

gravityTds.setTemperature(temperature);

gravityTds.update(); //sample and calculate

tdsValue = gravityTds.getTdsValue(); // then get the value

//Turbidity sampling

int turbValue = analogRead(TurbPin);

float turbVoltage = turbValue \* (5.0/1024.0);

//pH Sampling

int phSensorValue = analogRead(PhPin);

float phVoltage = phSensorValue \* (5.0/1024.0);

float phValue = 3.5\*phVoltage + Offset;

The sensors all acquire their data slightly differently. The above snippet occurs every 0.04 seconds. For the temperature probe, the function getTemp() is used, which reads the actual temperature, as signalled over the digital connection to the Arduino and converts it from hexadecimal to a temperature reading. The TDS library from DFRobot has a built-in function which samples, and adjusts the acquired value for the temperature of the water, which has been measured above, as TDS is sensitive to temperature.

The turbidity voltage is read from the analog pin and converted from the 10-bit integer to a 5v reading. During the writing of this script, the linear values of the conversion from voltage to NTU was not yet known and therefore not implemented yet. DFRobot and an academic paper disagree over the exact values. The pH has a similar situation, in this case however, the sampled values are converted to pH values by taking the example code from DFRobot. This means that the exact linear relation of the turbidity and pH voltages to their corresponding values still needs to be calibrated.

Code Snippet 2. Median filtering and sending over serial

//Temperature filtering and printing

medianTemp = getMedianNum(analogBufferTemp, SCOUNT);

Serial.print("Temperature: ");

Serial.print(medianTemp);

Serial.print(" Celsius; ");

The code snippet above is the same for all the sensors and occurs every 0.8 seconds. First, it filters the temperature array of 20 sampled values, by taking the median (middle value in an ordered array). With this filtered value, it prints it over the serial connection, by formatting it as “Temperature: xx Celsius;”. This is repeated for all the sensors, with the correct formatting. This will make it easy to read with another program, on another system, if the Arduino is connected over the USB port. The voltages are also published in a similar format.

## 3.3 Kogger Sonar

The sonar is a single beam echosounder made by Kogger.tech in Russia. They implemented a UART interface for communication. Each of the UART cables were soldered to a UART-USB converter. Using the available GUI (KoggerApp). One can change the settings of the device in this GUI. The following settings where changed: 1. Period: 100ms, 2. Binary distance data, 3. 30m max distance.4 turn boost on. These changes are sent to the sonar after pressing WRITE. These settings make it so that the sonar continuously sends data over the UART connection by using their Kogger Communication Protocol (documentation available online).

Without going over the protocol too in-depth, the main idea is that each distance measurement is a frame. This frame is structured into different sections. One of these sections is the actual distance measurement in 32-bit binary unsigned integer formatting.

A simple python script implementing this is available in the kogger\_testing\_serial folder of the github repository. As well as the complete ROS node in /ros\_dev\_ws.

## 3.4 Teensy

In order to simplify the communication to the central Raspberry Pi and the different sensors, a second microcontroller was implemented. A Teensy 3.2, chosen for the wide array of serial communication pins on board, and the ability to program it using the Arduino IDE, as well as Arduino (C++) code. The Teensy has two functions: The first is to read the incoming sensor-array information, and the second one is to read the GPS information. All coming in via serial ports. Which is then sent to the Raspberry Pi over USB. The code is available on the wqdrone GitHub, under /arduino/teensy. Personally, I would have liked it if the Teensy also read the sonar information. But I could not get the sonar to communicate with the Teensy. In the end, the USB converter with some python scripts also worked.

The GPS information is parsed using the NeoGPS library. Which is a low-resource intensive Arduino library for reading GPS data, from all sorts of receivers. It also comes with a variety of default scripts to use. One of these default scripts was used for the Teensy, with some slight alterations. This script reads the incoming NMEA GPS data and publishes it over the USB port.

Reading the incoming serial data is done by finding the start and end markers of the serial line (< and >), as programmed in the Arduino Uno. After it has found these, it prints the whole information over the USB port.

Which means that by using the Teensy, we reduce the need for the Raspberry Pi to handle the GPS parsing and serial communication. As well as use only a single USB port for multiple sensors.

# 4. Calibration

The pH, TDS and turbidity sensors all require a 2-point calibration. Recommended is that they are calibrated before each measuring session. A two-point calibration means that a measurement is taken of a fluid with a known value, at a high and low value. The underlying assumption here is that the relation is linear and each of the voltages (or other values), can be transposed to the wanted values. With a linear combination of the input variables. Which consist of the 2 known calibration values. As well as the 2 voltages the probe measured with the calibration solutions.

For this system, it is recommended that this is implemented at the post-processing stage. As reprogramming the Arduino each time can be a hassle. Post processing is easier implemented.

Below is explained step-by-step how this is done for the Turbidity sensor. But the same principles apply for the others.

# 5. Sensor array ROS nodes

The main idea of this sensor array is that it could be implemented into ROS and used as different information data points for a broader USV system. The incoming data can be easily stored using built-in ROS functions such as ros2 bag, which creates a SQL database from all the different nodes. This database can be used later for analysis.

The basis of ROS, is the ability to create or use previously made nodes. These nodes can be started, and will do ‘something’. It could publish data from all sorts of sensors, it could be a service to parse the information to another format. But the most important thing that ROS does, is handle callback timers and synchronicity, next to having a wide array of default data formats. All of this is done inside C++ or Python written nodes. Data used in these nodes is published on topics.

Getting back to the sensor array. The Raspberry Pi, running the ROS2 software is connected to the Teensy, sending GPS and water quality information, as well as the sonar, sending depth information. Just as most programming endeavors, this can be tackled in a variety of ways. My personal choice was to create 3 different topics: for the GPS, for the sonar and the water quality sensors. However, because the Teensy handles two of these, the GPS and sensors need to be done in a single node.

In this project, the nodes have been written in python, and mainly uses the pyserial package, which handles the incoming data streams. The source codes of these nodes can be found in the ros\_dev\_ws folder of the GitHub. Which is also an actual ROS2 workspace, so it can be directly copied and built to work on another ROS2 system.

## 5.1 The Teensy Node.

The source can be found under /ros\_dev\_ws/src/teensy/teensy/teensy\_pub.py

The Teensy node reads information from the Teensy USB port. In case of the Raspberry Pi connection, this is /dev/ttyACM0. The node reads each of the incoming lines and checks whether it is a GPS line, or a sensors line.

If it is a GPS line, it takes the whole line, converts it from bytes to a string, and splits it on the “,” separator. Each of the items in this new list corresponds to one of the NMEA protocol datapoints, created in the NeoGPS library. Having this information, one can parse the data into correct formats. For latitudes and longitudes on the WGS-84 coordinate system, it has seven digits of floating-point precision. The other characters are placed in front of it, with a “.” Placed inbetween. This is then converted to a python float. Each of the useful numbers (lat,lon and altitude) goes through this conversion and is placed in a NavSatFix message. A default ROS2 message, which predefines that there is a latitude, longitude and altitude. This is then published on the /teensy\_fix topic in ROS2, when running the node.

If it is a sensor line. The incoming line is read to its’ completion and then converted to a python string. This string is then parsed using regular expressions, to acquire all of the numbers from the different sensors. Regular Expressions are a quick way to do something with a string. In this case, extract all the sensor numbers out of the string. These numbers are then converted to python floating-point numbers. These numbers are placed in a list, and this list is published on the /wq\_sensors topic. Using a Float64MultiArray default message from ROS.

Two other python scripts are found next to this teensy\_pub file, which are two simple topic publishers, they listen to the /teensy\_fix or /wq\_sensors topic and repost that information again.

## 5.2 The Sonar Node.

The source can ben found under /ros\_dev\_ws/src/kogger\_sonar/kogger\_sonar/kogger\_dist\_rec.py

The Sonar node reads the incoming binary data from the sonar at USB port location /dev/ttyUSB0. As explained previously, this data is sent over as a frame, the frame contains different sections, and one of these sections is the actual depth information. Which means that the sonar node needs to read the incoming stream, stop at the correct length, split it up into the different sections, extract the depth information and post this on the /sonar\_dist topic.

The first step is to open the serial port and await incoming information. Whenever a byte is coming in, it is placed into a string, the payload. The length of this string is always the same, as the distance-frames are always the same length. By checking whether the length of this payload is the same length as the distance-frame. We know if the data has fully arrived yet. If the data has completely arrived, the distance information (in bytes) is converted into the unsigned 32-bit integer, representing the millimeters of depth that the sonar has measured. All of this is done each time the ROS timer fires, and this whole sequence reruns again. This depth information is placed into a simple, default ROS2 UInt32 message. Which is posted on the /sonar\_dist topic.

There are three other python files in this directory. Sendtokogger was used to send a request for data to the sonar and also publishes this line to the /sonar\_send topic. But ultimately was not used anymore, since the sonar now sends a message every 0.1 seconds, without a request. Listentokogger listens to the messages that sendtokogger has sent. Similarly, kogger\_dist\_list listens to the messages published on the /sonar\_dist topic. Just like the other two files, it is superfluous.