RISE Self Driving Vehicle Platform
RC-Car Operator's Manual
Version: 0.1

Benjamin Vedder
benjamin.vedder@ri.se

Bo Joel Svensson
joel.svensson@ri.se

RControlStation is free software released under GPL version 3. For more information about this license form visit `https://www.gnu.org/licenses/gpl-3.0.en.html`.

Contents

1 Introduction

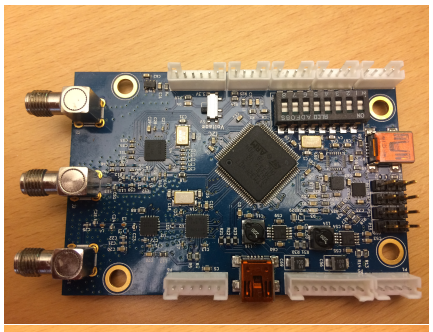
This manual aims to outline and exemplify configuration and use of the RISE Self Driving model Vehicle Platform (SDVP). It does not go in depth in describing the hardware or software that runs on the micro-controllers in for example the RC car, but rather tries to give enough background for starting operate the system using the RControlStation software which is a graphical user interface for controlling and monitoring the SDVP.

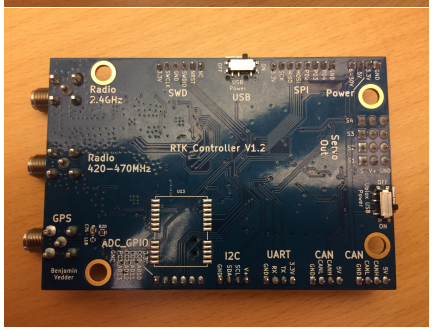
RControlStation can, among much more, be used for the following:

- Track and trace movement of the SDVP overlaid on a map implemented based on OpenStreetMap.
- Draw, load and save trajectories for the car to follow. Trajectories are visualized as points connected by line segments on the map.
- Control the car directly via the keyboard.
- Configure car dynamics.
- Configure numerous parameters related to the positioning system.
- Monitor and log data obtained from the SDVP.

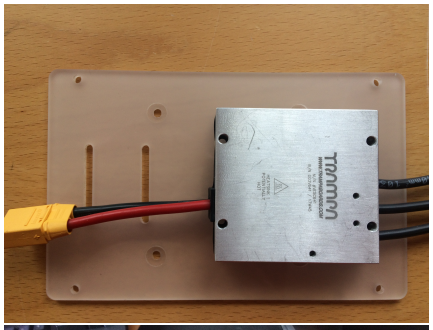
2 RISE SDVP Hardware Overview

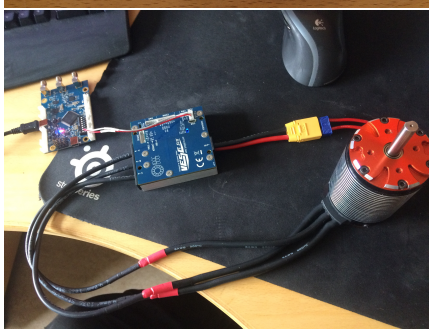
The hardware found on the RC car consists of a motor controller (the VESC), another controller board that we call the RTK Controller and a Raspberry Pi. The Raspberry Pi is there to provide Wifi and/or 4G cellular connectivity and enable for example remote debugging. The RTK Controller implements the self driving aspects of the SDVP and includes positioning capabilities using GPS and has two radios for communication. The RTK Controller connects to the car steering servo, providing its power and PWM signal for control. The RTK controller is also connected to the VESC motor controller over CAN-bus. If a Raspberry Pi is present in the system it is connected to the RTK Controller over USB, it is possible to power the Pi through this connection with a modification shown below. Some of the GPIO pins of the Pi can also be connected to the SWD port on the RTK Controller to enable remote flashing, using OpenOCD, of the embedded software.



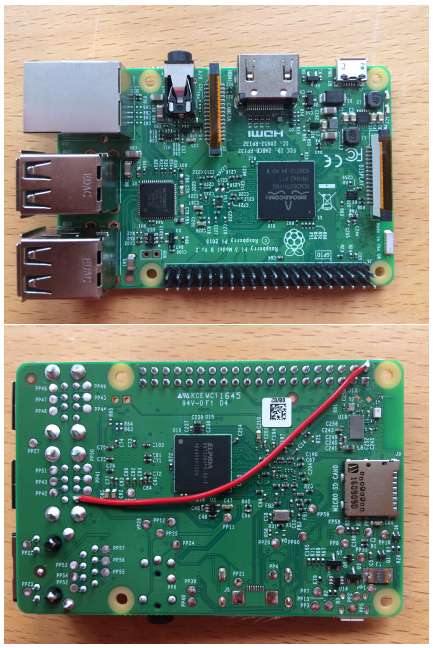


The pictures show the top (top) and bottom (bottom) view of the RTK Controller. On the top view you can see three antenna connectors for GPS and radio. There is a set of dip switches that configure the identity of the board. The set of connector pins along the right edge are for controlling servos and the white sockets are CAN, I2C, UART, SWD and SPI connectors. On the bottom view you can see solder pads for a UBLOX chip (this card happens to not be equipped with one) and a small switch for toggling "Power over USB" to power a connected Raspberry Pi backwards through its USB port. The RTK Controller connects to the computer running RControlStation either, over USB directly, over radio or over WIFI or 4G via the Raspberry Pi.





These pictures show the VESC motor controller, alone (top) and together with an RTK Controller and a motor (bottom). On the left side of the VESC (block of aluminum) you can see a battery connector. You can connect 3 - 12 cell lithium ion battery on this connector, resulting resulting in 8V to 50V. On the other side of the aluminum are three cables for connecting a brush-less motor. On the bottom of the VESC there is also a set of connectors for CAN, SWD and so on. On the bottom picture, the RTK Controller is connected to the VESC over CAN.



These picture show a top and bottom view of a modified Raspberry Pi. The modification is a wire soldered from the 5V pin on the USB port to the Pi 5V net. The Pi's own voltage regulator transforms this to the 3.3V needed.

3 Configuring a Linux Machine

This section describes how to set up a Linux system (an Ubuntu system is assumed) with the tools needed to run RControlStation and to develop towards the RISE SDVP. Start out by installing some basic dependencies:

```
\sudo apt-get install git build-essential libudev-dev
```

The IDEs (development environments) preferred by some of the authors of this documentation is Qt-Creator and Eclipse. To download Eclipse click here [1]. Qt can be downloaded from `https://www.qt.io/download` or click here to directly download the open source version. Now, install Qt and Eclipse, both have install wizards to help with this procedure.

Arm development tools are needed if you want to compile new versions of the various firmwares for the micro-controllers on the SDVP. Download arm development tools from `developer.arm.com` or click this link. To install the ARM tools simply extract the files from the archive and place them in a suitable location, for example `$HOME/opt/gcc-arm-none-eabi/`.

```
tar xvf gcc-arm-none-eabi-7-2018-q2-update-linux.tar.bz2
mv gcc-arm-none-eabi-7-2018-q2-update-linux $HOME/opt/gcc-arm-none-eabi
```

Then configure your `$PATH` path variable to include `$HOME/opt/gcc-arm-none-eabi/bin`

OpenOCD is used To flash the ARM Cortex M4 microcontrollers used throughout the SDVP. To install OpenOCD execute the following command:

```
sudo apt-get install openocd
```

Add yourself to the *dialout* group to be able to use, for example `ttyACM0` without having to be root user (sudo). This is useful when connecting to the various sdvp boards over a serial link (USB).

```
sudo usermod -a -G dialout <USER_NAME>
```

[1] `wget http://ftp-stud.fht-esslingen.de/pub/Mirrors/eclipse/oomph/epp/photon/R/eclipse-inst-linux64.tar.gz`

After adding yourself to the dialout group, you need to log out and in again for the change to take effect.

It can also be good to uninstall modemmanager, if installed, as it will try to talk to serial devices that connect as ttyACMx (as our controller cards does).

```
sudo apt-get remove modemmanager
```

Your Linux system is now set up for SDVP development (and use). The next step is to clone the rise_sdvp repository from github. Executing the following command will create a new directory called rise_sdvp, so make sure are already in a suitable directory:

```
cd <SUITABLE_DIR>
git clone git@github.com:vedderb/rise_sdvp
```

The directory `<SUITABLE_DIR>/rise_sdvp` will in the rest of the document be referred to as the `<SDVP_DIR>`.

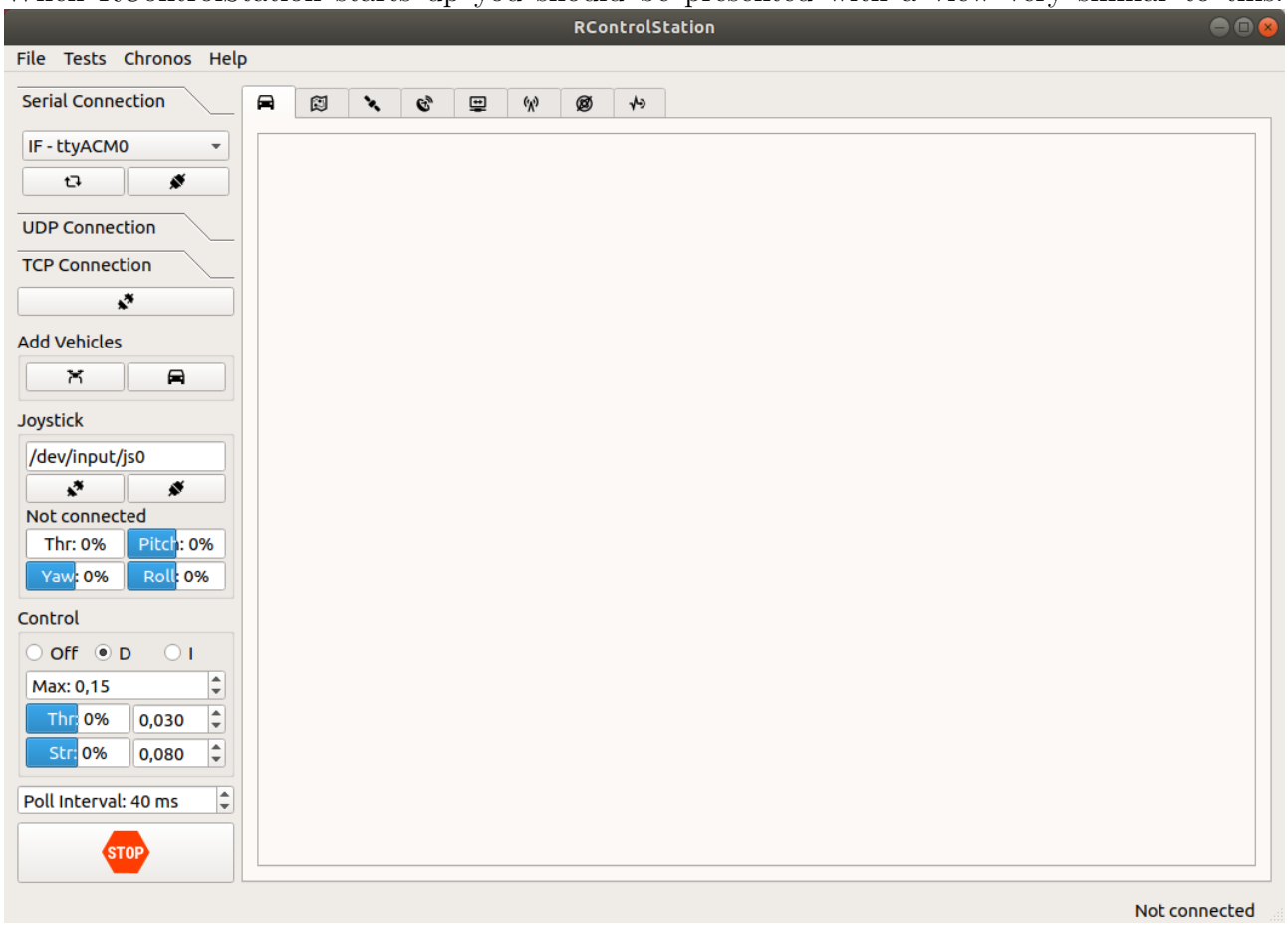
4 Buidling RControlStation

To build and run RControlStation, start Qt-Creator and on the welcome screen click *Open Project*. Navigate in the directory hierarchy to where you cloned the rise_sdvp repository and go to `<SDVP_DIR>/Linux/RControlStation`. Select the file `RControlStation.pro` and click open. Qt-Creator will now open the project and switch to a new view that allows you to edit files. On the left side of the window you can browse the RControlStation project. To build, locate the *Build* menu and select *Build Project "RControlStation"*. At the bottom of the Qt-Creator window there is a collection of tabs, one of them reads *4 Compile Output*. You can keep an eye on this tab for any error outputs during compilation.

When compilation concludes successfully, click the little green "play" button located along the left edge of Qt-Creator. This should bring up RControlStation.

5 RControlStation GUI

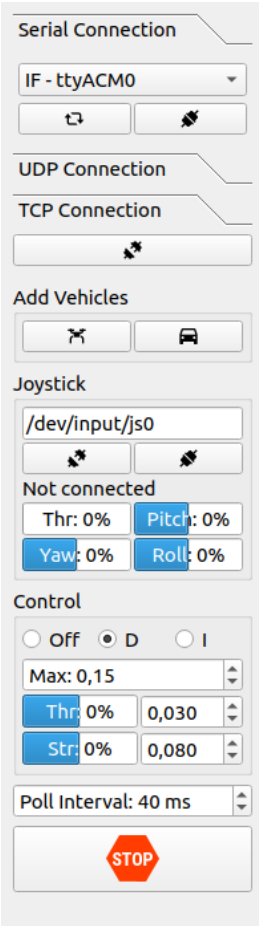
When RControlStation starts up you should be presented with a view very similar to this:



In this section we will go through and briefly explain the meaning and functionality behind most of the elements of this GUI.

5.1 Connectivity Controls

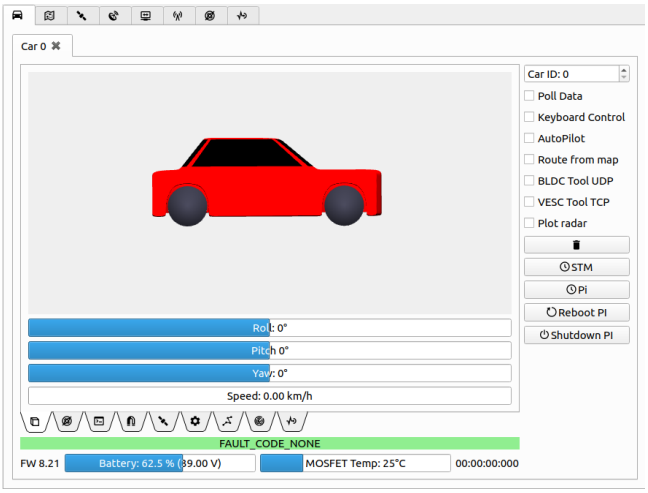
To the left side of RControlStation are fields for selection method of connecting to the SDVP, adding vehicles and control related configuration and indicators.



- **Connection:** Serial, UDP or TCP connection can be selected. Serial is used when connecting to the RTK controller using the radio mote or directly via USB cable. UDP and TCP connects to the SDVP via Raspberry Pi using either WIFI or 4G. Below the connection tabs is a button for disconnecting whatever connection has been set up.
- **Add Vehicles:** Add a car or quadcopter. Adding for example a car, changes the view on the right side to display a car and its set of car specific controls. The car specific controls are shown in Section 5.2.
- **Joystick:** Control car using a PS3 or PS4 joystick, supported on linux only. The thrust and yaw meters are in of interest when controlling a car and pitch and roll only when controlling a quadcopter.
- **Control:** Is related to the keyboard control functionality. It can be turned off, put in "D" mode or "I" mode. "D" refers to duty mode and "I" representing current mode. A "gain" value can be set for thrust and steering that influences how strongly these respond to keyboard interaction.
- **Poll Interval:** Interval with which to poll car for updates to status. This sets the frequency with which a "getState" message is sent to cars. The reply to this message from the car contains, among much more, position and speed.
- **Stop:** Stops a moving car that is running under auto pilot. This is one of the many "Emergency stop" buttons available in RControlStation.

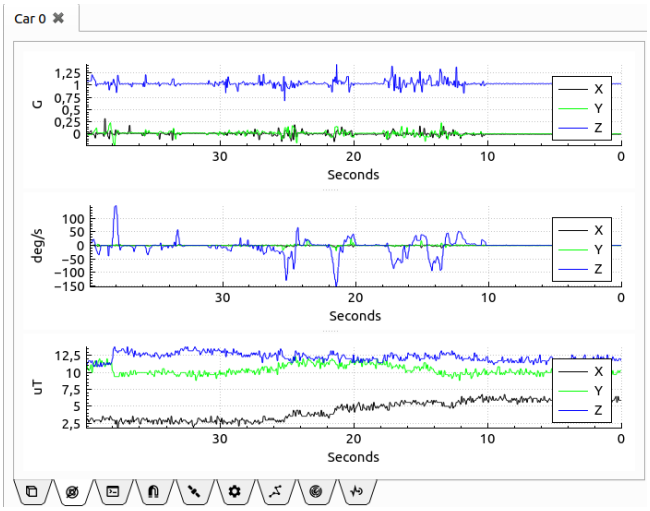
5.2 Cars

After adding a car, "Car 0" will appear in the *Cars* tab with a set controls similar to this image. To the right of the displayed 3D image of a car is a set of controls, toggles and buttons, the "Car ID" field should be set so that it corresponds to the setting of the dip switches on the RTK controller. A way to test if the connection to the car is working, is to check, click, the **Poll data** box and watch the rendered image of the car to see if it seems to respond to physically moving the SDVP.

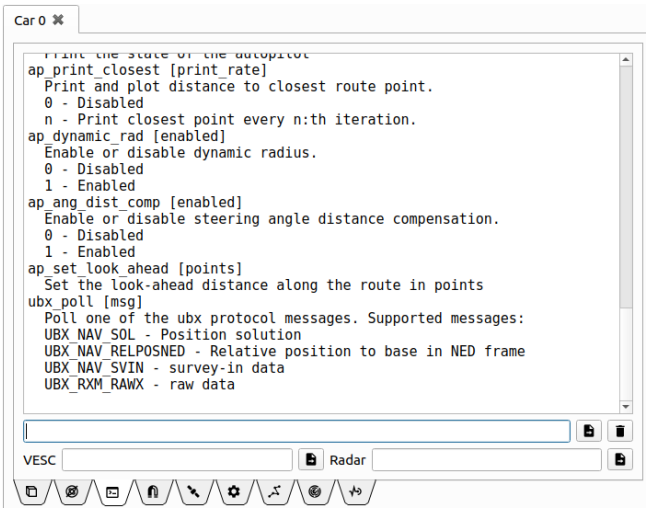


Car orientation: Toggling **Keyboard Control** allows control of the car via the keyboard arrow keys and **AutoPilot** starts trajectory following mode.
The **Route from map** toggle, turns on or off direct upload of trajectory points to the car as they are added to the map.
The next two toggles, **BLDC Tool UDP** and **VESC Tool TCP** start servers for interfacing with external tools.
Plot radar turns plotting of radar samples on. Note that a radar is not part of the standard setup of an RISE SDVP car.

There is a set of indicator bars under the 3D model of the car showing pitch, roll and yaw degrees and a speed display. Then there is a set of tabs for further configurations or interfacing, these tabs are all explained below.

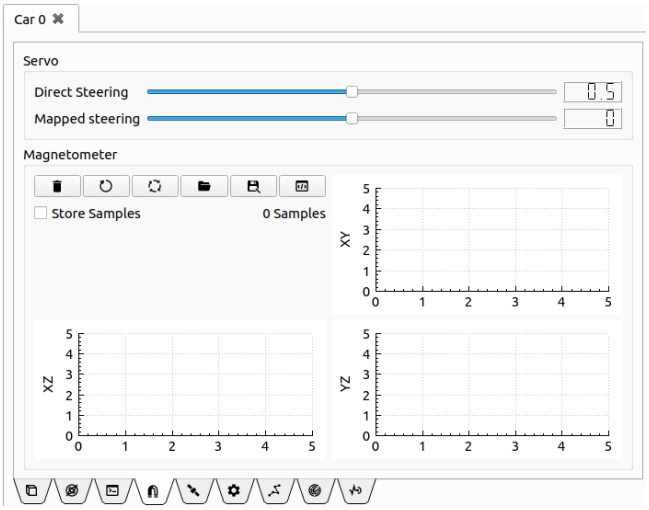


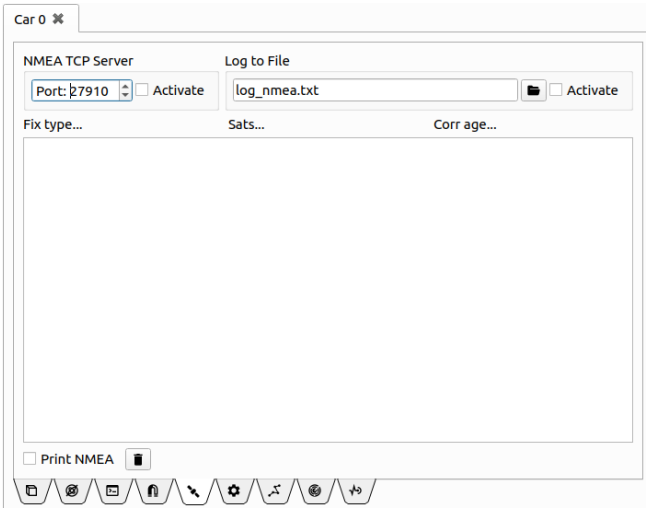
IMU Realtime: The IMU tab displays realtime sensor data from the Inertial Measurements Unit. The top graph shows accelerometer readings, measured in G:s, in the middle gyroscope readings in degrees per second are displayed and at the bottom the magnetometer data in micro Teslas.



Terminal: This tab provides a terminal interface to the car. The “help” commands provides a list of valid commands. Section 9 shows a list of commands that are valid at 2018-09-03.

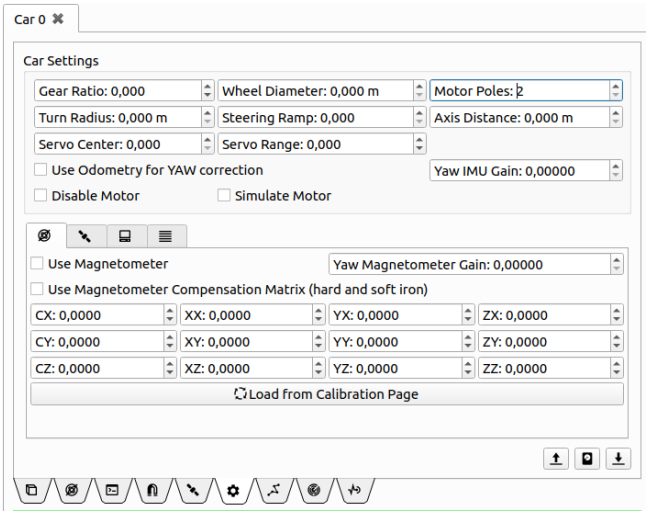
Calibration: On this tab there are controls for steering servo and magnetometer calibration. To calibrate the magnetometer, turn on **poll data** and check the store samples box. There are buttons for calculating compensation, loading and storing of samples and to view C code representing the calibrations settings.

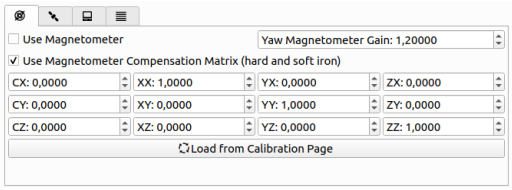


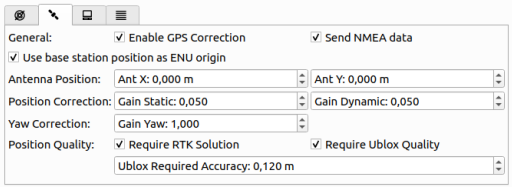


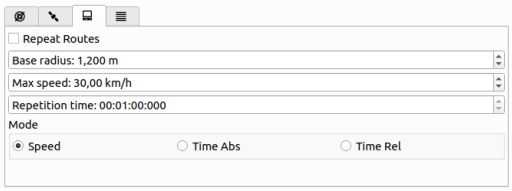
GPS: The GPS tab displays information about the GPS status. **Fix type:** will indicate if the system is running on regular GPS/GNSS or if the RTK computations have converged and reached Fix. This status is important as there are big differences in the precision of obtained positions depending on fix type. **Sats:** will indicate the number of satellites that are visible to the car. It is also possible to start an "NMEA server" here for sending the data obtained from car onwards to some arbitrary external software for further processing or logging.

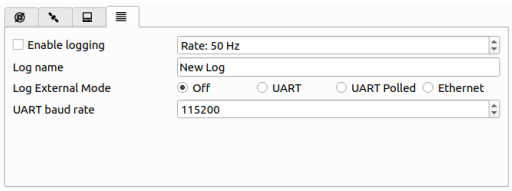
Configuration: The *Car Settings* tab contains configurations that can be loaded and stored to the car. Loading and storing is done using the up and down arrow buttons located at the bottom right. The preferred way of changing these is to first load the current settings from the car with the up button, then perform changes followed by writing the configuration back to the car. Changing the information in the top half of this tab requires knowledge of the dynamics of the car.
The bottom half of this tab contains a number of sub-tabs that are explained below.



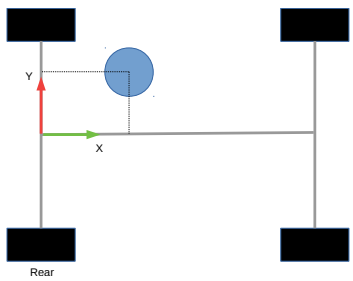




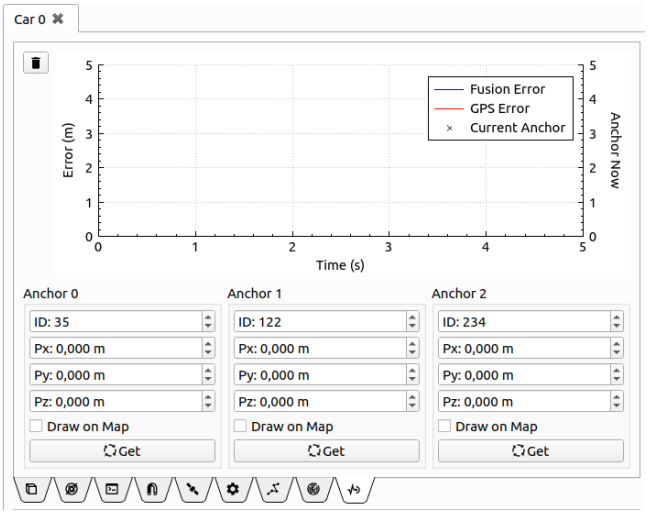




Magnetometer: Turn usage of magnetometer data on or off and configure compensation matrix.
GPS: Here you can enable the use of GPS in the positioning calculations. This can be left always on, the system will make use of GPS information only if available. The only reason to turn this off, would be if you do not want to use any GPS information even if available. The *Send NMEA data* checkbox should be checked if you want to send GPS messages back to RControlStation (to the above mentioned GPS tab). For SDVPs equipped with the M8P variant of the Ublox chip you can also check *Require RTK Solution* and *Require Ublox Quality* and set these to desired values. The SDVP will then only use these values in position computations if those conditions are met. The *Position Correction* parameters (gain values), control how strongly the GPS measurements will interface the positioning computations that are also based on for example odometry. The *Antenna Position* settings are explained below using an additional image.
AutoPilot: Configure the behavior of the auto pilot. If the car should repeat the route after completion check the *Repeat Route* box. The *Base Radius* setting is related to the variant of the PurePersuit algorithm used for trajectory following where the car heads towards the closest point intersecting the trajectory on a circle surrounding the car. There are also three different modes for trajectory following: in *Speed* mode the car uses speed information stored in the trajectory, then there is absolute time and relative time where timestamps on the trajectory either are treated as an absolute time when car should be at point or time relative to previous point.
Logging: Enable, disable and set frequency of logging. There is also an *External* mode where the log data is sent over for example UART. .



The antenna position to be entered in the GPS tab shown above is defined as this picture shows. The origo of the car is located between the rear wheels, so the distances that should be measured and entered are how much in front/behind or above/below of this point the antenna is located. The unit used is metres, so if in this case the car is 0.5m between the axles then the antenna position would be roughly $x = 0.15m$ and $y = 0.07m$



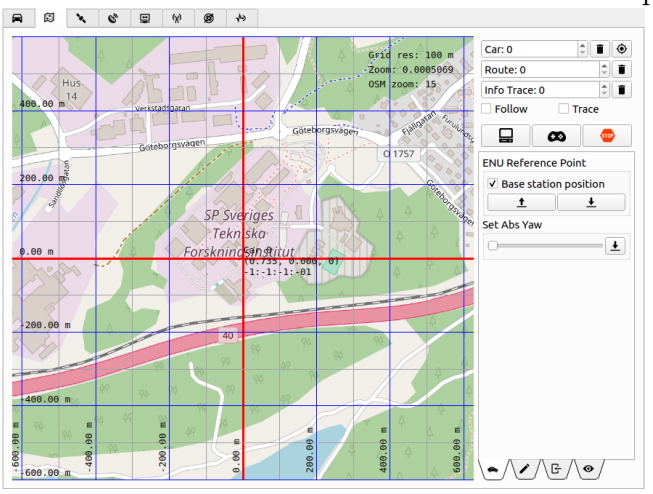
DecaWave UWB Measurement: This tab allows the configuration of three UWB anchors and also displays a plot of relative error at time.

- **ID:** Corresponds to an ID coded into the firmware or the UWB anchor.
- **Px,Py, Pz:** Position of the anchor. The *Get* button reads the current position of the car into these fields.

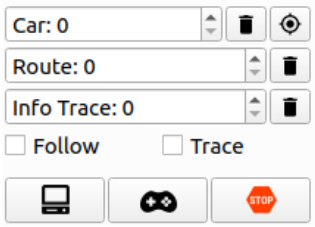
5.3 Map

The map view is based on OpenStreetMap that is overlaid with a grid showing distances from an origo. The origo is set to the ENU (East North Up) reference point used in positioning calculations. This ENU point can be configured to be postion of the GNSS-RTK base station.

To zoom in or out on the map use the scrollwheel on the mouse or two fingers on the touchpad. Panning is done by clicking and dragging with the mouse. The following keyboard shortcuts are also useful for interaction the map view:

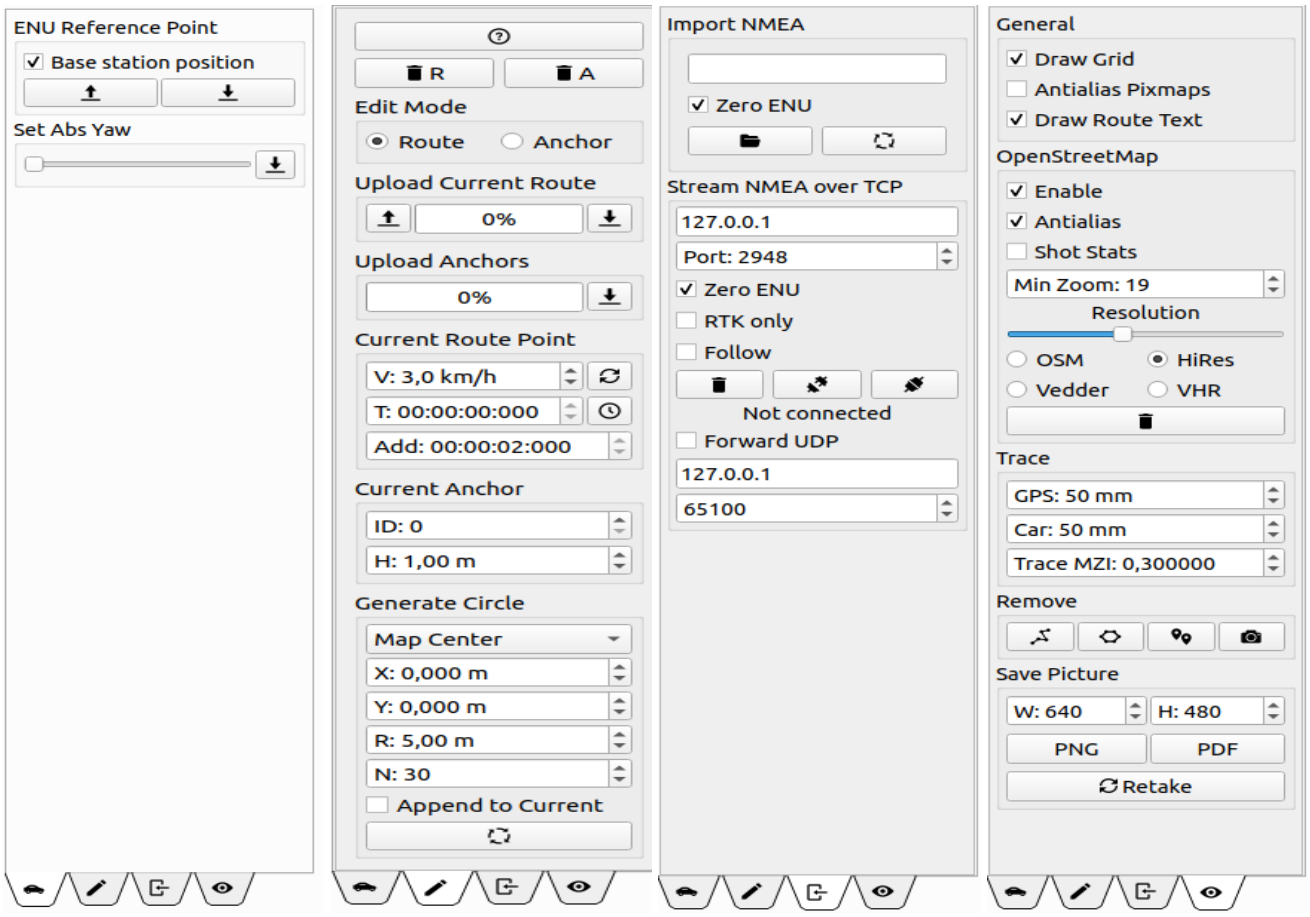


- **CTRL + Right click:** Update route point or anchor settings
- **Shift + Left click:** Add route point or anchor
- **Shift + Left drag:** Move route point or anchor
- **Shift + right click:** Delete route point or anchor
- **CTRL + SHIFT + Left click:** Zero map ENU coordinates



These controls to the right of the map allows selections of car, route and trace based on ID. Checking the *Follow* box results in the map following the selected cars movements automatically. The buttons are for turning auto pilot, manual control and stopping a car running on auto pilot.

To the right of the map, there is also a set of tabs with controls and configuration options for use together with the map. The first of these tabs is called *Car Control*

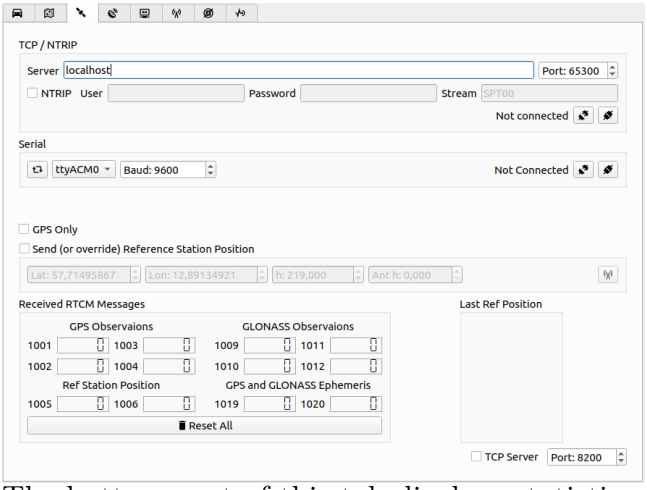


A **B** **C** **D**

A: This tab lets you load base station position from the car or store an updated one (from the map view) to the car.
B: Contains controls for route and anchor editing. Here you can read and write routes to and from the car and write anchor positions to the car. You can change speed settings that affect the next added route point and generate circles, arcs.
C: On this tab you can import a logfile containing NMEA data and plot based on this data on the map view. You can also connect to TCP servers that stream out NMEA data and plot on the map.
D: This tab contains settings for how and what is drawn in the map view. It is also possible to save the map view here as PNG or as vector format in a PDF.

5.4 RTCM Client

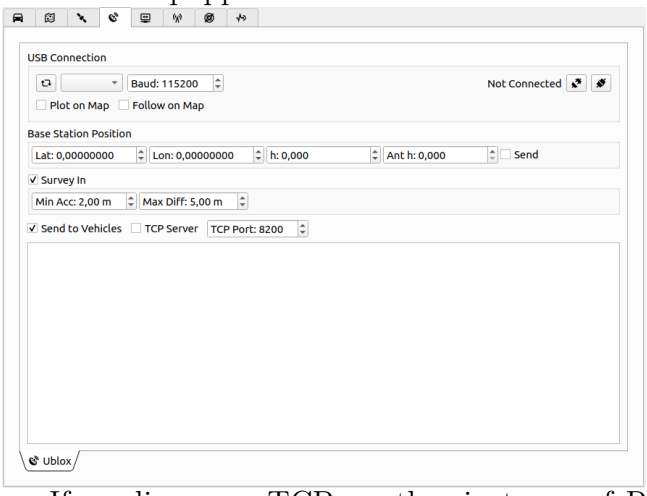
On this tab you can configure a connection to an NTRIP service for RTK correction data. NTRIP stands for *(Networked Transport of RTCM via Internet Protocol* and RTCM here is a standard for for transmission of DGNSS data.



If you have a subscription with a streaming service you can enter the details here (for example `www.euref-ip.net` may be a starting point).
you can also configure the RTK Controller to receive these messages over the serial-port using the *Serial* field.
It is possible that the data stream does not contain information about the base station position of which relative measurements are performed. In that case, the position of the base station can be hard-coded here using the *Send (or override) Reference Station Position*.
The bottom part of this tab displays statistics of messages received.

5.5 Base Station

You can set up a personal RTK GPS base station using a laptop and a Ublox. For example the Ublox equipped RTK Controller could be connected to the laptop USB port.

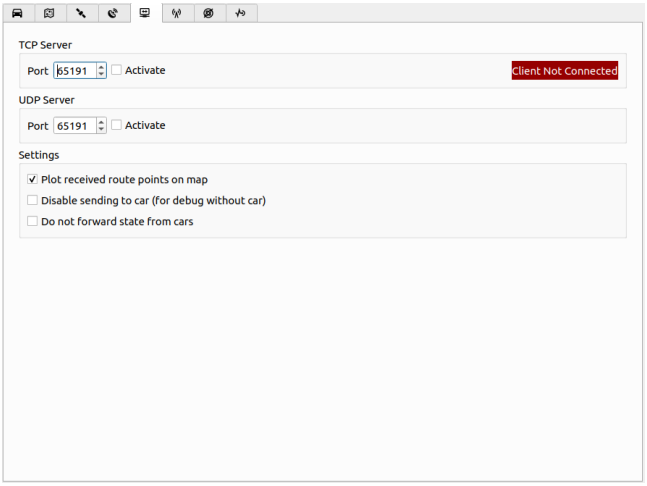


The position of the base station can be hard-coded in the *Base Station Position* or derived using the *survey in* functionality. When using the survey in functionality you can also set constraints on the desired accuracy.
You can choose to include the position or not in the data stream being sent out, check or uncheck the *Send* box.
The two bottom checkboxes activates sending of messages via the internal SDVP message transmission system or via a TCP server.

If sending over TCP another instance of RControlStation can subscribe to the data sent using the previous tab, Section 5.4, but could also be used by arbitrary software wanting these readings.

5.6 Network Interface

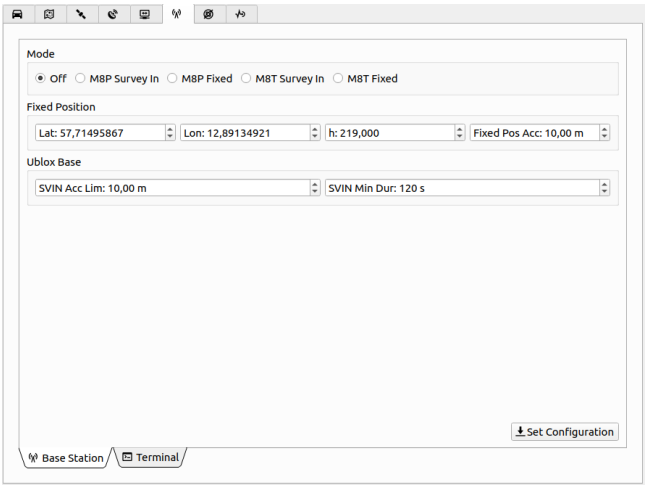
Here, TCP and UDP servers can be startet for allowing external programs to connect to RControlStation for example using the RControlStationComm library that is outlined in Section 8.



You can chose what ports to use and activate TCP and UDP independently. The indicator will show when external software has connected. Using this network interface gives external software control over the car, via RControlStation, this can be disabled if undesirable or if debugging. It is also possible to choose not to forward state updates from the car to external software. Currently it seems that using external software over TCP does not work when also checking the *Poll Data* box in RControlStation, see Section 5.2.

External software can modify and add routes via an *addRoutePoints* function (if using the RControlStationComm library), it is possible to toggle the displaying of these in map view on or off here.

5.7 Mote



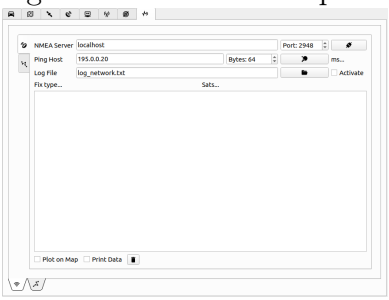
This tab is for when you use an RTK controller with the firmware compiled in "Mote mode". I this mode the RTK controller can act as a base station and send correction data to cars over radio.

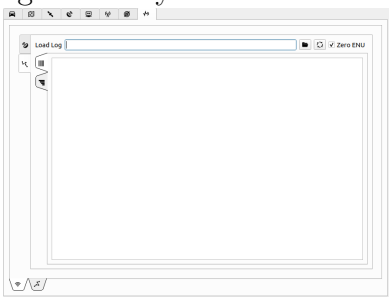
The configuration options here are similar in concept to the ones we showed earlier (the soon to be changed *base station* tab). But are here already more automated.

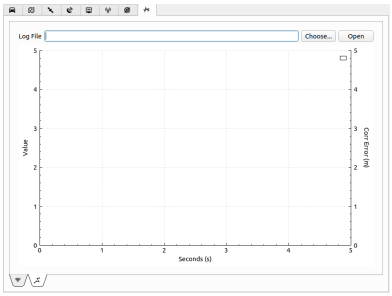
You can select whether to use a hand-coded position for the base station using modes "Fixed" or to allow the automatic position computation "Survey in". If using the survey in modes you can also set what accuracy that should be present and for how long it must be stable before accepting it as position.

5.8 Logging and Analysis

The logging and analysis tabs collect different kinds of data and allows loading and storing of log files. We are not explaining these any further in this version of this manual.

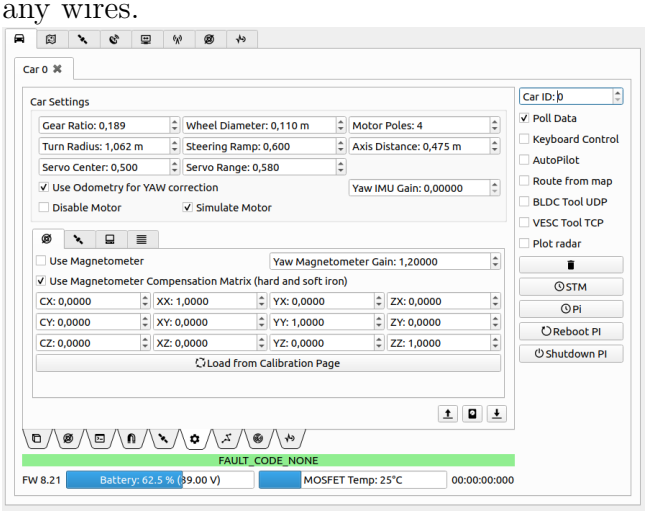






6 Example Setup: HIL Simulator

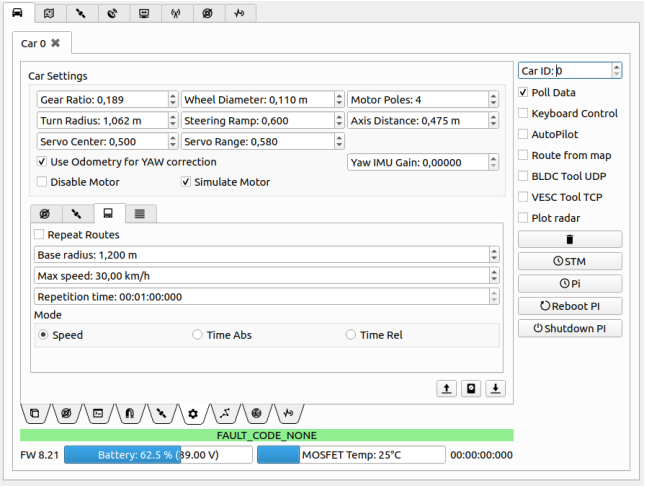
RControlStation together with a RTK controller can be used as a hardware in the loop simulator. The simplest form of this setup is an RTK controller connected to a computer over a USB cable, but adding the Raspberry pi and using wifi or ethernet is also possible. Yet another option is to use the RTK controller connected over radio with a tranceiver connected to the computer's USB port. Note that it is also possible to use a fully assembled car as a HiL simulator by doing the configurations shown below, without having to physically disconnect any wires.



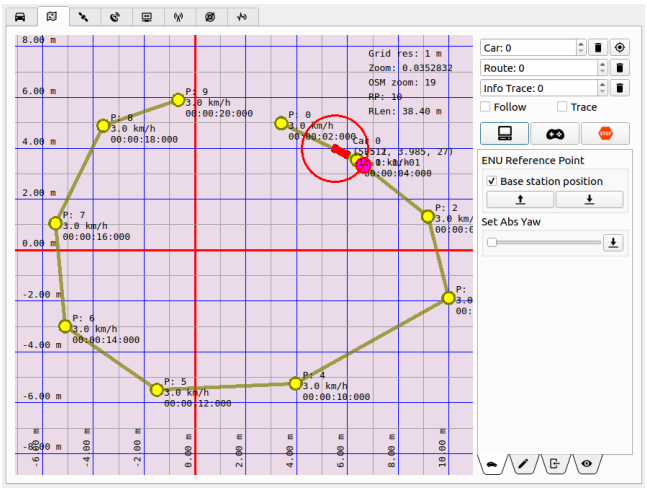
The important settings here for simulation is:

- **Use Odometry for YAW Correction:** Should be checked.
- **YAW IMU Gain:** should be set to zero.
- **Simulate Motor:** Should be checked, unless you are running the car raised from the ground on for example a box.

Before changing settings, read the default values from the car using the disk icon or the current settings on the car using the up-arrow icon.

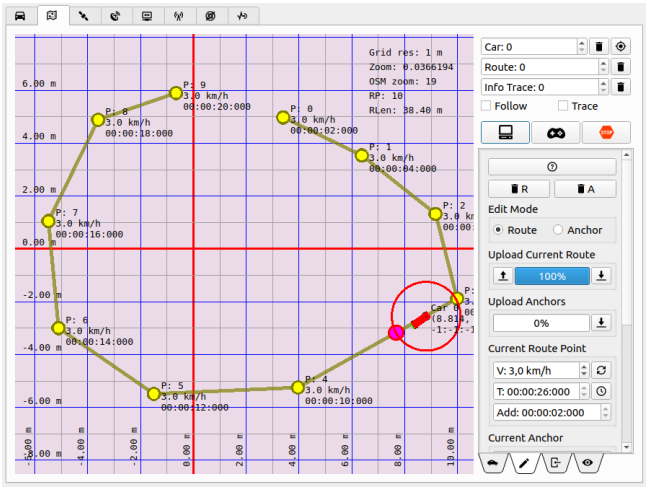


There are also some settings related to the autopilot that may want to look at before proceeding. These settings are decided by what you want to do such as repeal routes or what mode of trajectory following you are interested in. Since we are not moving any car for real these can be set rather freely without much risk. Maybe consider the maximum speed setting if you are running without motor simulations on a full car.



Now draw a route on the map by **Shift + click** in a number of positions on the map.
You can also experiment with moving the car around using the arrow keys. First press the joystick icon to enable this. Remember to turn on **Poll Data** on the car tab.

For the car to automatically follow the route, it needs to be uploaded to the car. This is done on the tab activated to the right of the map view as shown in this picture. Uploading to the car is done using the down-arrow icon under the *Upload Current Route* label. If this is successfully acknowledged by the car (Green text bottom right of window), you can start the auto pilot by clicking the icon that looks like a computer.



7 Example Setup: Fully Operational System

This is the case when you have a full car setup and want to drive outside. Expecting that it is configured correctly, which is out of scope of this document, this setup is quite easy. If the above HiL experiment worked well then the changes needed are as follows:

- **Simulate Motor:** Should be unchecked.
- **YAW IMU Gain:** Set to 0.5.
- **USE Odometry for YAW Correctio:** Uncheck.

In this setup it is more important to think about safety and start out slow. So set auto pilot max speed to a low number. Also, familiarize yourself with the positions of the **Stop** buttons to be able to quickly stop the car if something seems wrong. When looking at the map view you will have a stop button on the bottom left of the window and one on the right side of the window close to the top.

Also look at the GPS tabs and check the status of the positioning system, a **Fix** status is prefereable.

Just as in the HiL case, first try driving the car around with the arrow keys before allowing it to follow a route automatically.

8 RControlStationComm Library

The RControlStationComm library exists as a help for people who are interested in writing software with the ability to interface with the SDVP. This library exists in a C and C++ flavour and can be used to for example upload and download trajectories to and from a car, check error statuses, activate/deactivate autopilot and set/get ENU reference point.

An example for how to use this library is available in the main SDVP repository at github (`www.github.com/vedderb/rise_sdvp`) under Linux/RControlStationComm/test and another example exists in github repository `www.github.com/svenssonjoel/rise_sdvp_shell`.

The interfaces (H-files) for these libraries are shown in the subsections below.

8.1 C++ library

```
class RCONTROLSTATIONCOMMSHARED_EXPORT RControlStationComm
{
```

```
public:
    RControlStationComm();
    ~RControlStationComm();
    bool connectTcp(QString host, int port);
    void disconnectTcp();
    void setDebugLevel(int level);
    bool hasError();
    char *lastError();
    bool getState(int car, CAR_STATE *state, int timeoutMs = 1000);
    bool getEnuRef(int car, bool fromMap, double *llh, int timeoutMs = 1000);
    bool setEnuRef(int car, double *llh, int timeoutMs = 1000);
    bool addRoutePoints(int car, ROUTE_POINT *route, int len,
                        bool replace = false, bool mapOnly = false,
                        int mapRoute = -1, int timeoutMs = 1000);
    bool clearRoute(int car, int mapRoute = -1, int timeoutMs = 1000);
    bool setAutopilotActive(int car, bool active, int timeoutMs = 1000);
    bool rcControl(int car, int mode, double value, double steering);
    bool getRoutePoints(int car, ROUTE_POINT *route, int *len,
                        int maxLen = 500, int mapRoute = -1, int timeoutMs = 1000);
    bool sendTerminalCmd(int car, char *cmd, char *reply, int timeoutMs = 1000);

private:
    struct ERROR_MSG {
        QString command;
        QString description;
    };

    QTcpSocket *mTcpSocket;
    int mDebugLevel;
    QByteArray mRxBuffer;
    QVector<QByteArray> mXmlBuffer;
    QVector<ERROR_MSG> mErrorMsgs;
    char mTextBuffer[10000];

    // CoreApplication
    QCoreApplication *mApp;
    int mAppArgc;
    char const *mAppArgv[2];

    void processData(const QByteArray &data);
    void sendData(const QByteArray &data);
    QByteArray waitForXml(int timeoutMs = 1000);
    bool waitForAck(QString cmd, int timeoutMs = 1000);
    bool isTcpConnected();
    QByteArray requestAnswer(int car, QString cmd, int timeoutMs = 1000);
    bool checkError(QString xml);

};
```

8.2 C library

```
bool rcsc_connectTcp(const char* host, int port);
void rcsc_disconnectTcp(void);
void rcsc_setDebugLevel(int level);
bool rcsc_hasError();
char *rcsc_lastError();
bool rcsc_getState(int car, CAR_STATE *state, int timeoutMs);
bool rcsc_getEnuRef(int car, bool fromMap, double *llh, int timeoutMs);
bool rcsc_setEnuRef(int car, double *llh, int timeoutMs);
bool rcsc_addRoutePoints(int car, ROUTE_POINT *route, int len,
                         bool replace, bool mapOnly,
                         int mapRoute, int timeoutMs);
bool rcsc_clearRoute(int car, int mapRoute, int timeoutMs);
bool rcsc_setAutopilotActive(int car, bool active, int timeoutMs);
bool rcsc_rcControl(int car, int mode, double value, double steering);
bool rcsc_getRoutePoints(int car, ROUTE_POINT *route, int *len,
                         int maxLen, int mapRoute, int timeoutMs);
bool rcsc_sendTerminalCmd(int car, char *cmd, char *reply, int timeoutMs);
```

9 Car Terminal Commands

The valid commands to use in the terminal are:

- **help** Shows list of valid commands.
- **ping** Print pong here to see if the reply works
- **mem** Show memory usage
- **threads** List all threads
- **vesc** Forward command to VESC
- **reset_att** Re-initialize the attitude estimation
- **reset_enu** Re-initialize the ENU reference on the next GNSS sample
- **cc1120_state** Print the state of the CC1120
- **cc1120_update_rf [rf_setting]**
  Set one of the cc1120 RF settings
  0: CC1120_SET_434_0M_1_2K_2FSK_BW25K_4K
  1: CC1120_SET_434_0M_1_2K_2FSK_BW50K_20K
  2: CC1120_SET_434_0M_1_2K_2FSK_BW10K_4K
  3: CC1120_SET_434_0M_50K_2GFSK_BW100K_25K
  4: CC1120_SET_434_0M_100K_4FSK_BW100K_25K
  5: CC1120_SET_434_0M_4_8K_2FSK_BW40K_9K
  6: CC1120_SET_434_0M_4_8K_2FSK_BW50K_14K
  7: CC1120_SET_434_0M_4_8K_2FSK_BW100K_39K
  8: CC1120_SET_434_0M_9_6K_2FSK_BW50K_12K
  9: CC1120_SET_452_0M_9_6K_2GFSK_BW33K_2_4K
  10: CC1120_SET_452_0M_9_6K_2GFSK_BW50K_2_4K

- **dw_range [dest]** Measure the distance to DW module [dest] with ultra wideband.
- **zero_gyro** Zero the gyro bias. Note: The PCB must be completely still when running this command.
- **pos_delay_info [print_en]** Print and plot delay information when doing GNSS position correction.
  0 - Disabled
  1 - Enabled
- **pos_gnss_corr_info [print_en]** Print and plot correction information when doing GNSS position correction.
  0 - Disabled
  1 - Enabled
- **pos_delay_comp [enabled]** Enable or disable delay compensation.
  0 - Disabled
  1 - Enabled
- **pos_print_sat_info** Print all satellite information.
- **pos_sat_info [prn]** Print and plot information about a satellite.
  0 - Disabled
  prn - satellite with prn.
- **ap_state** Print the state of the autopilot
- **ap_print_closest [print_rate]** Print and plot distance to closest route point.
  0 - Disabled
  n - Print closest point every n:th iteration.
- **ap_dynamic_rad [enabled]** Enable or disable dynamic radius.
  0 - Disabled
  1 - Enabled
- **ap_ang_dist_comp [enabled]** Enable or disable steering angle distance compensation.
  0 - Disabled
  1 - Enabled
- **ap_set_look_ahead [points]** Set the look-ahead distance along the route in points
- **ubx_poll [msg]** Poll one of the ubx protocol messages. Supported messages:
  UBX_NAV_SOL - Position solution
  UBX_NAV_RELPOSNED - Relative position to base in NED frame
  UBX_NAV_SVIN - survey-in data
  UBX_RXM_RAWX - raw data

10 Changelog

2018-09-13 : Joel
Initial version focusing on the RControlStation GUI in relation to the RC-car.
