University Of Southern Denmark
Faculty of Engineering

Bachelor thesis

Precision drone navigation using RTK GNSS

Author:
Mikkel Skaarup Jaedicke

Supervisors:
Kjeld Jensen
Morten Hansen

June 1, 2015

Precision drone navigation using RTK GNSS

A bachelor thesis submitted in partial fulfillment for the requirements of the degree of Bachelor in Science Robot Systems

at the

Faculty of Engineering
University of Southern Denmark
Maersk Mc-Kinney Moller Institute

Mikkel Skaarup Jaedicke mijae12@student.sdu.dk

Supervisors:

Morten Hansen	Associate Professor
Kjeld Jensen	Associate Professor

Date of start:	1^{st} of February 2015
Date of end:	1^{st} of June 2015

The report, source code, plotting script and test data can be found at:

`https://github.com/mikkeljae/bachelor`

Mikkel Skaarup Jaedicke	Date

Abstract

This paper describes the development of a high precision positioning system to use with a small UAV. Real Time Kinematic GNSS was used in order to obtain high precision positioning. A RTK rover and a RTK reference station was developed on Raspberry Pi using RTKLIB. Static tests of the system showed that 94.3% of all data points had an error of less than 1 cm, when using LEA-M8T GNSS receiver, GPS and a ZEPHYR antenna. When using the system with a small compact GNSS antenna 94% of all data points had an error of less than 10 centimeters.
The Pixhawk autopilot firmware is analyzed and it is described how to develop for the platform. A custom GNSS driver was written in order to receive data from the developed RTK rover. The driver updates the data from NMEA 0183 *GGA* messages correctly, but not enough GNSS information is given for the Kalman filter to work properly. This results in wrong position estimates.
High precision navigation by an UAV was not achieved as the interface between the custom driver and the Kalman filter was not correct. However a compact high precision positioning system was achieved by the use of RKTLIB on Raspberry Pi.

Abbreviations

This section lists abbreviations used in the report alphabetically.

CUI	Console User Interface
C/A	Coarse / Acquisition
DGNSS	Differential Global Navigation Satellite System
FDMA	Frequency Division Multiple Access
GNSS	Global Navigation Satellite System
GLONASS	Russian global navigation satellite system
GPS	Global Positioning System
GUI	Graphical User Interface
LEA-M8T	GNSS receiver developed by u-blox
LEA-6T	GPS receiver developed by u-blox
L1	1575.42 MHz in GPS and 1602.0 MHz in GLONASS
L2	1227.6 MHz in GPS and 1246.0 MHz in GLONASS
L5	1176.45 MHz in GPS
NMEA 0183	Protocol for electronic device communication made by the National Marine Electronics Association
NSH	NutSHell
P(Y)	Encrypted signal for military use
PX4	Pixhawk hardware module
RPI	Raspberry Pi
RTK	Real Time Kinematics
RTKLIB	Real Time Kinematics LIBrary
RTOS	Real Time Operating System
SDU	University of Southern Denmark
UAV	Unmanned Aerial Vehicle
UBX	u-blox
uORB	micro Object Request Broker
UTC	Coordinated Universal Time
UTM	Universal Transverse Mercator

Contents

1 Introduction

Small autonomous unmanned aerial vehicles (UAV) have just merely been introduced to the commercial world. Yet the European Union estimates that there will be 150,000 jobs in the UAV industry in Europe by 2050. UAVs are the future and can be used in many different industries. We have seen their entry to the military market, where they can be used for surveillance. But they are also used for more peaceful surveillance of for example huge fields of crops. Farmers are able to send out an UAV and get a full aerial view of their crops. This may help to discover pests or plants suffering from water shortage. UAVs can be used for a wide range of tasks and the world has just begun to see the usage areas.

Many of the tasks carried out by UAVs in the industry requires global positioning. This is typically done by ordinary GPS. This only yields a precision of about 10 meters. Combined with data from other sensors in a filter it yields sufficient precision for lots of tasks.

In recent years the price of Global Navigation Satellite System (GNSS) receivers, with the ability to make carrier wave measurements, have declined drastically. This method allows for much greater precision than ordinary GPS. The price reduction of these receivers have made it feasible to use them for UAV positioning.

A lot of task where very precise positioning of UAVs is needed can be thought of. **SDU Erhverv**, a unit at University of Southern Denmark aimed at strengthening the bonds to regional companies, made contact to this project with such an idea. The idea is not explained here as it is confidential, but very precise positioning on a small UAV is needed in order for it to realized.

SDU has initiated a great deal of new UAV related activities. This project is a small part of these activities. The Pixhawk autopilot is chosen as the autopilot used in these activities and therefore this project aims to provide general knowledge and experience with using the Pixhawk autopilot.

This project seeks to investigate whether high precision navigation using an UAV can be realized using the Pixhawk autopilot. The hypothesis of the project and the requirements for the project is developed in collaboration with **SDU Erhverv**.

Hypothesis

It is possible to get a drone to fly within $\pm$ 5 centimeters of a specified route 95% of the time.

System analysis

In order to achieve the aim of the project and to accept the hypothesis a very precise positioning system is needed. This positioning system needs to be small and compact in order to mount it on a UAV. The method of Real Time Kinematics (RTK) GNSS yields the highest possible precision global positioning. RTKLIB is a software package that is capable of doing RTK computations. In order to make the positioning system compact it needs to be implemented on a compact computer, such as a Raspberry Pi. The LEA 6-T GPS receiver is small and compact and can be used for a RTK system.
The Pixhawk autopilot is an open-source autopilot in both hardware and software. It is fully operational and used worldwide. An external GPS receiver needs to be connected to the Pixhawk module in order to do global positioning. Commonly an ordinary GPS receiver is connected, but a RTK system could be used instead.
The system analysis makes the following requirements necessary for the system to be realized.

Requirements

- Analysis of RTK GNSS using RTKLIB with the LEA 6-T and test of the performance.
- Analysis of Pixhawk software.
- Development of an interface between the Pixhawk and the RTK system on Raspberry Pi.
- Development of a program on the Pixhawk to exploit the precision of the RTK GNSS when flying.
- Development of a program on the Pixhawk to log flight data.

2 GNSS

This chapter seeks to evaluate the existing GNSS methods and their ability to fulfill the projects goal of high precision positioning. A basic explanation of different types of GNSS and different ways of using them will be provided. Different coordinate systems and protocols used in the field of GNSS will also be explained. The chapter is based upon knowledge from Kaplan [2].

2.1 GPS

GPS is an American GNSS. It has been fully operational since 1995 with 24 operational satellites. Since then new satellites have been taken to usage, while others have been shut down. 32 GPS satellites were in use as of 2012. The orbits of the satellites gives a global visibility of at least 8 satellites at any time.

Satellites

The original satellites transmitted data on L1 (1575.42 MHz) frequency and L2 (1227.6 MHz) frequency. On L1 coarse/acquisition (C/A) code was sent for civil use and precision (P(Y)), reserved for the American military, code was sent on L2. Newer satellites also transmits C/A code on L2 and on a new civilian frequency L5 (1176.45 MHz). The C/A or P(Y) code is transmitted along with a navigation message, which holds information about the satellite's position, health, orbit and etc.. The C/A or P(Y) code and the navigation message is modulated onto the L1 carrier wave. The concept is shown in figure 2.1.

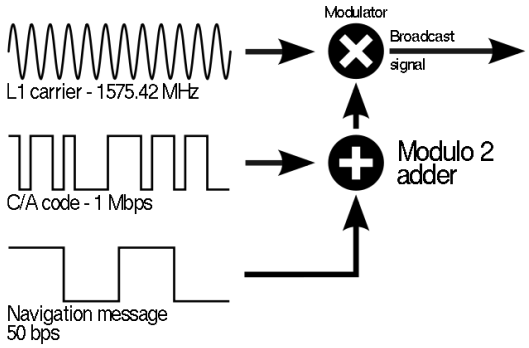


Figure 2.1: Modulation of a GPS signal on L1.

The C/A code is a pseudo random code of 1,023 bits, which is unique for each satellite.

Receivers

All GPS receivers know the satellites' unique C/A codes and can use this for calculating the signal's propagation time. The GPS signal is demodulated, when it is received by a receiver. Then the C/A code is compared and aligned to a replica C/A code and the propagation time is known. The method is illustrated in figure 2.2. The aligning of the two codes, done by the receiver, needs to be done very precise. An alignment error of half a cycle would be equivalent to a timing error of half a microsecond propagation time, which would translate into an error of 150 meters.
Receivers can fortunately align the codes very precisely and the error is within a couple of percent of a cycle. One percent error translates to an error of 3 meters.

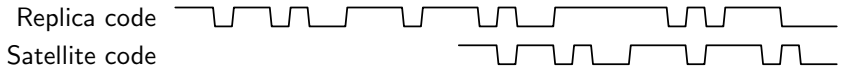


Figure 2.2: When comparing code from satellite with replica code the propagation time can be determined.

Carrier wave phase measurement

To achieve higher precision than with alignment of the C/A code a method called carrier wave phase measurement is used. Instead of aligning the C/A code this method seeks to align the carrier waves. The carrier wave has a frequency thats over a 1000 times the frequency of the C/A code. This means that if the carrier waves can be aligned within a couple of percents error, the resulting error would be in centimeters or less. One percent error on the L1 band would translate to an error of 1.9 mm. The problem with this method is that it is problematic to align the signals correct as the signal is quite uniform. The C/A code is designed to be complex, which makes the task of aligning signals easier. The receiver can match the phase of the signal, but it's difficult to determine if it is wave n or wave $n+1$. The problem arising is an integer ambiguity problem. The integer ambiguity problem needs to be solved correctly to yield precise positioning. The alignment of the C/A code is commonly used to guide in the search of the right integer.

2.2 GLONASS

GLONASS is a Russian GNSS. It is similar to GPS in many ways. GLONASS has equivalent codes to C/A and P(Y), namely a standard precision navigation signal and a high precision navigation signal. All satellites transmit the same standard precision navigation signal, but they each use a different frequency using a 15-channel frequency division multiple access (FDMA) technique. There is a L1 band with a center frequency of 1602.0 MHz and a L2 band with a center frequency of 1246.0 MHz. Carrier wave phase measurement is possible and is done as described in the GPS section.

2.3 Single point positioning

The simplest use of GNSS is called single point positioning and works by calculating distances to satellites. The distance is calculated by looking at the propagation time of the satellite message. In theory, when the distances from a receiver to three satellites are known the position of the object can be found. The problem with this is that the receiver would need to know the time precisely and with no offset to the clocks in the satellites. This is practically impossible for most purposes, so a fourth satellite is needed. Using single point positioning gives a precision of around 10 meters [2, page 379]. A part of the uncertainty derives from atmospherically changes leading to changes in propagation time.

2.4 Differential positioning

To account for change in the atmosphere a method called differential GNSS (DGNSS) may be used. The method seeks to enhance the GNSS positioning by correcting some of the errors. This is done by having a reference station, which position must be accurately known. The reference station is equipped with a GNSS module and because the position is known, corrections to the GNSS data can be calculated. These corrections are sent to a nearby moving unit usually called a rover. The rover can then more precisely calculate its position by using the correction data received from the reference station. An illustration of the concept is shown in figure 2.3.

2.5 Real Time Kinematics

RTK satellite navigation is a version of DGNSS that uses carrier wave phase measurements to achieve higher precision. The positioning is done in real time, which requires a connection between the reference station and the receiver.

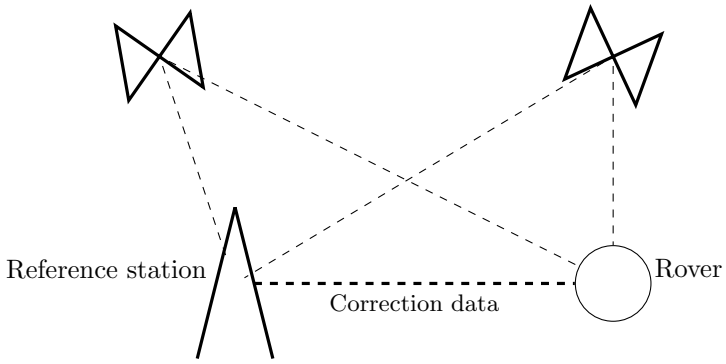


Figure 2.3: Both rover and reference station receive data from satellites. The reference station sends correction data to the rover.

2.6 Coordinates and Protocols

In the field of GNSS multiple coordinate systems are used. They are developed for different usage areas. Some of these are used in this project and will be explained here. There are also a number of protocols used to format GNSS information into messages. Those used in this project will be explained here.

2.6.1 Latitude and Longitude coordinate system

Latitude and longitude is a coordinate system for earth. Latitude lines are parallel to equator and longitude lines are perpendicular to equator. North of equator latitude degrees are called N and south is called S. The north pole is 90°N and the south pole 90 °S. For longitude the prime meridian goes through Greenwich and longitude west is called W and longitudes east are called E. Latitude and longitude are traditionally measured in degrees, minutes and seconds. When using latitude and longitude in calculation it is often easier to use a decimal degrees format.

2.6.2 UTM coordinate system

The Universal Transverse Mercator (UTM) coordinate system is a two dimensional Cartesian coordinate system for earth. The UTM system is a map projection of earth, but it is not a single map projection. Earth is divided into sixty zones, each is a six degree band of longitude. Latitude bands are not a part of the UTM system, but is often used along with it. Each of the sixty zones is divided into 20 latitude bands with a width of 8 degrees. Some countries have made exceptions from the UTM system to make navigation easier within their country.

2.6.3 NMEA

NMEA is a protocol made by the American National Marine Electronics Association. It was originally developed as an interface between various marine electronic equipment. GNSS receiver communication is also specified within this protocol and it is used by some GNSS receivers. The protocol exist in different versions e.g. NMEA 0180, NMEA 0183 and NMEA 2000. The protocol will be described, as it used in the project. The protocol consists of different message types. Each message starts with a `$` followed by a sequence of characters

defining the message type. In this project especially the $GPGGA message, which is a message containing essential fix data, is important. A NMEA 0183 $GPGGA message is shown below.

```
$GPGGA,141452.00,5522.40940,N,01023.86310,E,1,08,1.0,40.809,M,39.170,M,0.0,*73
```

A translation of the message can be seen in table 2.1. It is translated according to the specification in [3].

Field	**Information**
$*GPGGA*	GPS GGA message
141452.00	Sample was taken at 14:14:52:00 UTC
5522.40940, N	55°22.40940' latitude, Northern hemisphere
01023.86310, E	10°23.86310' longitude, Eastern hemisphere
1	Fix value. Values translates to different solutions depending on the user
08	Number of satellites being tracked
1.0	Horizontal dilution of position
40.809,M	Antenna Altitude above mean-sea-level (geoid), meters
39.170,M	Units of antenna altitude, meters
0.0	Time in seconds since last DGPS update
(empty field)	DGPS station ID number
*73	Checksum data, always begins with *

Table 2.1: Translation of a NMEA 0183 $*GPGGA* message.

2.6.4 UBX Protocol

The UBX protocol is used by u-blox GNSS receivers. It is a proprietary protocol to transmit data to a computer using asynchronous RS232 ports. The UBX protocol is compact and can deliver raw GNSS data to a computer.

2.7 GNSS Receivers

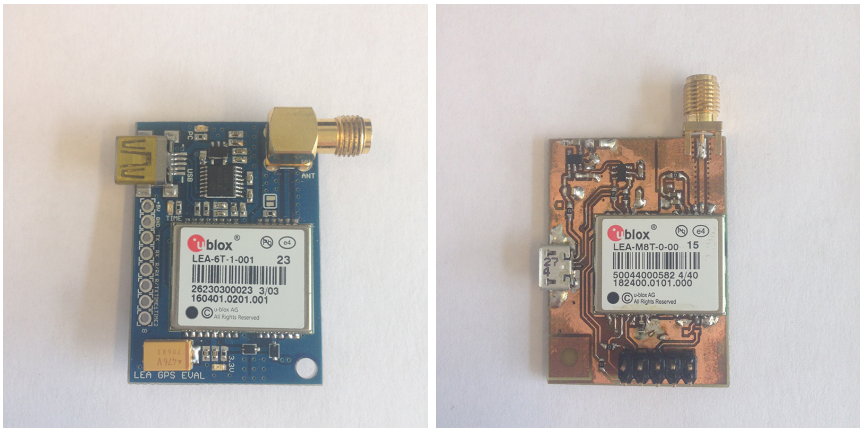
There are a wide range of GNSS receivers on the market today. This report will focus on two small low-cost GNSS receivers from the company u-blox. These two fit the project well, because they are small, which enables them to be used on flying drones and because they were available to the project.

2.7.1 u-blox receivers

u-blox is a Swiss technology company making wireless and positioning solutions. They produce several low-cost, compact and low-power GNSS modules. The modules range from single GNSS single point positioning to multi GNSS modules capable of doing carrier phase measurements.
u-blox has developed u-center which is a GNSS evaluation software for the u-blox modules. It can be used for evaluation and configuring of u-blox modules.

LEA-6T

LEA-6T is a single-GNSS (GPS) module capable of doing carrier phase measurements on L1, which enables it to be used for RTK GPS applications. It can output GNSS data in NMEA or UBX protocol. LEA-6T can be seen in figure 2.4a.



(a) LEA-6T GNSS module. (b) LEA M8T GNSS module.

Figure 2.4

The LEA-6T used in this project was set up to use UBX protocol on the USB port. This was configured in u-center by turning on UBX raw measurement data (RXM-RAW) and subframe data (RXM-SFRB) on the USB port. Then the module needed to be set to USB with UBX, NMEA and RTCM as the input protocol. Protocol out was set to be UBX. Lastly this configuration was flashed to the module.

LEA-M8T

LEA-M8T is a multi-GNSS (GPS + GLONASS) module capable of doing carrier phase measurements on L1. It was also configured with u-center to use the UBX protocol on the USB port. The configurations for this module is similar to the ones used for LEA-6T, but multi-GNSS raw measurement data (RXM-RAWX) and multi-GNSS subframe data (RXM-SFRBX) needs to be on instead of RXM-RAW and RXM-SFRB. The LEA-M8T can be seen in figure 2.4b.

2.8 Conclusion

The projects goal of precision within centimeters excludes the use of single point positioning, because the precision is simply too low. Using DGNSS and carrier wave phase measurements drastically improves the precision. If this is done at the same time it is called RTK, which yields the highest precision. The GNSS receivers LEA-6T and LEA-M8T are available to the project and are both capable of doing carrier wave phase measurements. LEA 6-T can only receive from GPS, while LEA-M8T can use both GPS and GLONASS. Both receivers were set up to output raw data in the UBX protocol. Using one of these two receivers along with RTK calculations should yield a precision high enough to meet the requirements of the project.

3 RTKLIB

As mentioned in the previous chapter there is a need for making a RTK system to achieve high precision positioning within 5cm. RTKLIB is a software library that can be utilized to make a RTK system. This chapter will go through the basics of RTKLIB and how to use it. Thereafter it will explain the making of a RTK system. The developed RTK system is partly built upon the work in Jensen [1]. The system is hereafter tested with different settings, different GNSS receivers and different antennas. The results are presented and discussed.

3.1 General RTKLIB

RTKLIB is an open-source software library for GNSS positioning developed by Takasu Tomoji. This chapter is partly based upon knowledge from the RTKLIB manual [8]. RTKLIB supports single point positioning, DGNSS and RTK calculations. The software is multi-GNSS and supports both GPS and GLONASS. It offers both real-time and post-processing calculations. It works on both Windows and Linux. There are more features in the Windows version and it has GUI applications while the applications are CUI on Linux. RTKLIB has several tools for working with GNSS. Used in this project is only the RTK tool `rtknavi` on Windows and `rtkrcv` on Linux and the communication server `strsvr` on Windows and `str2str` on Linux.
Version 2.4.2 is the newest stable version, but it doesn't support RXM-RAWX and RXM-SFRBX that needs to be used in order to exploit multi GNSS data from the LEA-M8T. Version 2.4.3 beta does support these formats and this version is therefore used in this project.

3.1.1 Solutions in RTKLIB

As mentioned above, RTKLIB supports several positioning modes. The different solutions can be seen in table 3.1. When RTKLIB is set to do RTK GNSS it will obtain one of the solutions listed, depending on the signal from the satellites. Fixed is the most precise solution and is RTK GNSS with the integer ambiguity properly resolved.

Q	Solution	Description
1	Fixed	Solution by carrier based relative positioning and the integer ambiguity is properly resolved.
2	Float	Solution by carrier based relative positioning but the integer ambiguity is not resolved.
3	Reserved	Reserved
4	DGPS	Solution by code-based DGPS solutions or single point positioning with SBAS corrections
5	Single	Solution by single point positioning

Table 3.1: Solutions in RTKLIB.

3.1.2 Windows

Installing on Windows is straightforward and gives, when finished, a folder of applications. The Windows applications are all GUI and all settings can be changed in the GUI. `rtknavi` is shown in 3.1.

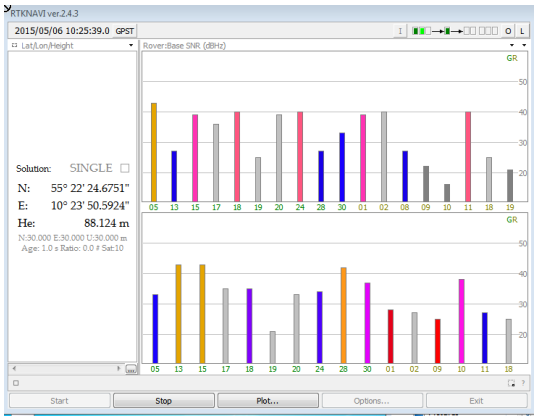


Figure 3.1: RTKLIB rtknavi running on windows.

3.1.3 Ubuntu

Installing on Ubuntu requires building the software. The complete shell commands for the installation is shown in listing 3.1. The installation requires dependencies, installed with apt-get. The software is then downloaded, unzipped and builded.

```
$ sudo apt-get install build-essential automake checkinstall liblapack3gf libblas3gf

$ wget https://github.com/tomojitakasu/RTKLIB/archive/rtklib_2.4.3.zip
$ unzip rtklib_2.4.3.zip
$ cd RTKLIB-rtklib_2.4.3/app

$ sudo bash makeall.sh
```

Listing 3.1: Shell commands for installation of RTKLIB 2.4.3.

Launching `rtkrcv` is done by navigation into the `rtkrcv` folder and running `rtkrcv` as root shown in listing 3.2.

```
$ cd RTKLIB-rtklib_2.4.3/app/rtkrcv/gcc/

$ sudo ./rtkrcv
```

Listing 3.2: Shell commands for running rtkrcv.

3.2 Setting up a RTKLIB base station

In this project a RTK base station was set up on a RPi with RTKLIB's str2str software. Hardware used for the base station was a RPi, a LEA-M8T, a 433 MHz 3DR radio link, an ethernet cable and a Gutec GPSL1L2A GNSS antenna.
The RPi needs to have a Linux operating system like Ubuntu, Debian or Raspbian. Then RTKLIB needs to be installed as described earlier.
The settings for the RTK base station are adopted from a previous RTK base station made in Jensen [1]. The settings are kept as the base station are used by multiple users at SDU. This way the users do not need to change their settings, if they just need GPS signal, but they will be able to also get data from GLONASS.
On RPi the file `rc.locale`, which runs every time it boots, was changed to run a custom startup script `refstat.sh`, which can be seen in listing 3.3. Line 18 runs `str2str` with the

specified settings for input and output. Input is coming from the serial port `ttyACM0` with baudrate 57600. Output from `str2str` is given to a TCP client on IP address 176.28.20.31 socket 42001. Output is also given to a serial device on `ttyUSB0` with baudrate 57600. The device is a 433 MHz 3DR radio link, that broadcasts the data. Station ID and station position is also given in line 18.

```
1  #!/bin/sh
2
3  # Parameters
4  GPS_DEV=ttyACM0
5  GPS_BAUD=57600
6
7  REF_IP_ADDR=176.28.20.31
8  REF_SOCKET_PORT=42001
9  REF_ID=SDU0
10 REF_LAT=55.374183827
11 REF_LON=10.398601242
12 REF_ELE=26.924
13
14 RDO_DEV=ttyUSB0
15 RDO_BAUD=57600
16
17 # Start RTKLIB str2str
18 ./RTKLIB-rtklib_2.4.3/app/str2str/gcc/str2str -in serial://$GPS_DEV:GPS_BAUD -out tcpcli://
       $REF_IP_ADDR:$REF_SOCKET_PORT -out serial://$RDO_DEV:RDO_BAYD -sta $REF_ID -p $REF_LAT
       $REF_LON $REF_ELE
```

Listing 3.3: Custom startup file; refstat.sh.

3.3 Setting up a RTKLIB rover

A RTK rover was set up on a RPi with RTKLIB's `rtkrcv` software. Hardware used was a RPi, u-blox GNSS module, GNSS antenna and a 433 MHz 3DR radio link.
The RPi needs to have a Linux operating system and RTKLIB needs to be installed.
The file `rtkrcv.conf` contains all settings for `rtkrcv` and this file should be customized for each setup. The GNSS receivers used are u-blox receivers and these were setup to output in UBX protocol format on a serial connection. Therefore `rtkrcv` was setup to receive data in UBX protocol format from the GNSS receiver. The radio link is connected by a serial connection to the RPi and it receives data broadcasted from the radio link on the RTK base station. The data is also formatted in the UBX protocol. Therefore `rtkrcv.conf` is configured to get reference station data from a serial port in the UBX protocol. `rtkrcv.conf` is configured to output in two ways. It ouputs in NMEA 0183 protocol format to a log file on the RPi and on a serial port. A `rtkrcv` configuration file can be seen in appendix B.
When `rtkrcv` is running, different commands can be used to check the status of the RTK rover. A selection of the most important commands can be seen in table 3.2. The status and stream commands can be followed by a cycle length e.g. 1 for updates for every second.

Command	**Meaning**
start	Start RTK server
stop	Stop RTK server
restart	Restart RTK server
status	Show RTK status
stream	Show stream status

Table 3.2: A selection of commands for rtkrcv.

3.4 Static tests

Static tests were made to test the precision of RTKLIB with the two different GNSS receivers. GNSS receiver used and antenna used was varied in the static tests.

3.4.1 Test setup

When collecting data for more than a couple of minutes it is not feasible to use a laptop as power supply as the setup needs to be outside with a good distance to buildings. Therefore a test setup was made with a power supply. The RPi needs a power supply of 5V $\pm$ 5% and it will use approximately 500 mA in this setting. Therefore it needs a fairly large battery to power it for several hours. A 12V 7AH battery, Y7-12, was available for this project. It can run the RPi setup for approximately 14 hours. The voltage needs to be converted to 5V in order to be used directly as a power source for the RPi. A simple voltage regulator circuit was built of a LM7805 and two capacitors. It can be seen in figure 3.2. The two capacitors act as lowpass filters and prevents high frequent noise in the output.

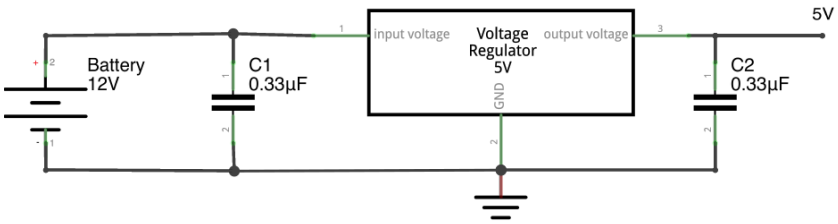


Figure 3.2: Power supply circuit.

The test setup was placed away from buildings and trees and with a clear view of the sky. The test setup itself can be seen in figure 3.3. The electronics are concealed inside a box and was furthermore covered in a plastic bag to shield for rain.

(a) Test setup with small GNSS antenna. (b) Test setup with ZEPHYR GNSS antenna.

Figure 3.3: Test setup.

Two different antennas were used for the tests. A small GNSS antenna and a large ZEPHYR

GEODETIC model 2 GNSS antenna. The small antenna was made available by a fellow student. Unfortunately it has not been possible to find the specifications of it. It is 3.5 cm x 3.5 cm and it is shown in in figure 3.4.

Figure 3.4: Small GNSS antenna used in this project.

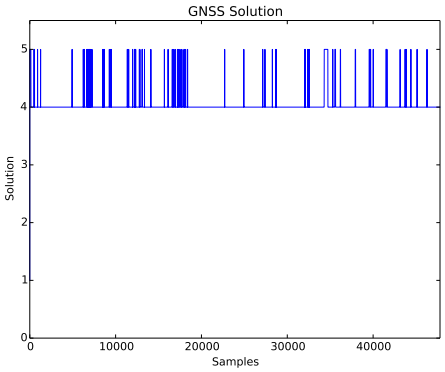
3.4.2 Plotting data

A python script was made to plot the data from the tests. A measure of the precision of the system is the distance of each data point from the position of the GNSS receiver. However it was not possible to determine the exact position of the GNSS receiver. The best estimate of the position of the GNSS receiver is the geographic midpoint of the data collected by the GNSS system. There can be data points from all solution types. *Fix solution* is the most precise and the midpoint is therefore calculated from the data points with *fix solution*. The geographical midpoint is calculated by converting data from latitude and longitude degrees to Cartesian coordinates, finding the averages of the Cartesian coordinates and converting them back to latitude and longitude degrees. Hereafter each data point's distance to the midpoint is calculated. Some data point's distance to the midpoint may be above an user specified outlier threshold and are removed. The number of data points removed is printed to the user. If data is removed a new midpoint is calculated based on the remaining data.

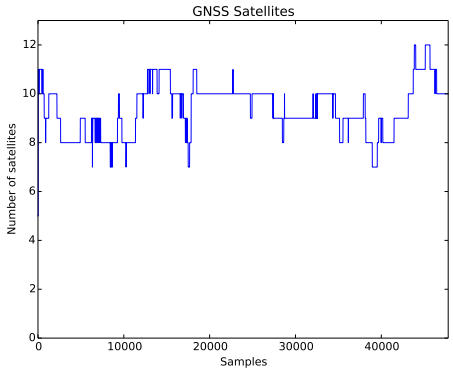
3.4.3 LEA-6T

RTK GPS with ZEPHYR antenna

RTKLIB was configured to kinematic mode and to use GPS. The setup collected data for 12 hours and 17 minutes with a sample rate of 1Hz. A plot of the solution over time can be seen in figure 3.5a. Of the total data 1.8% were *float solution* and 98.2% were *fix solution*. A plot of the number of verified satellites over time can be seen in figure 3.5b. There seems to be no correlation between the number of satellites and the solution.
A plot of all data can be seen in figure 3.6a and it is seen that some of the data differ quite a lot from the midpoint of the *fix* data. In figure 3.6b all *fix* data is shown. It is seen that there are a couple of outlier points, but other than that the data seems to be close together. In figure 3.7a a plot of the *fix* data without outliers are shown. A histogram of the data is shown in 3.7b. 96.4% of all data is within 1 cm from the geographical midpoint. Similar data was found.

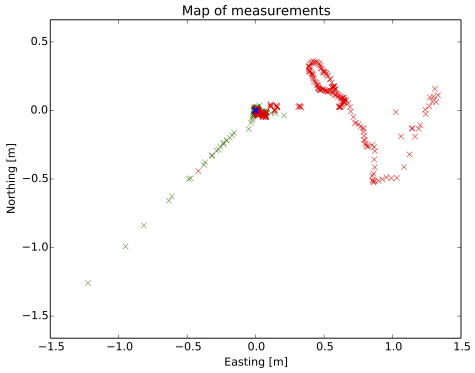


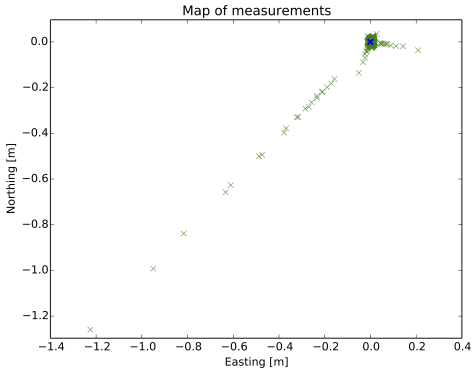
(a) GNSS solution over time.
4 = RTK *Fix* and 5 = RKT *Float*.



(b) Number of verified satellites in RTKLIB over time.

Figure 3.5

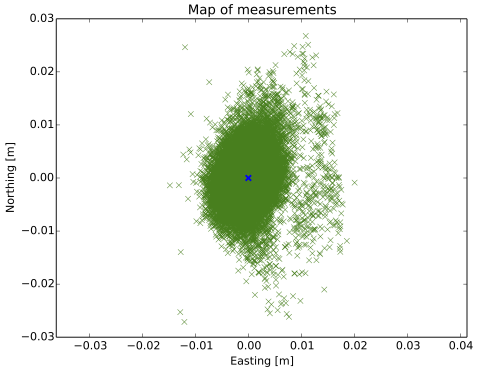




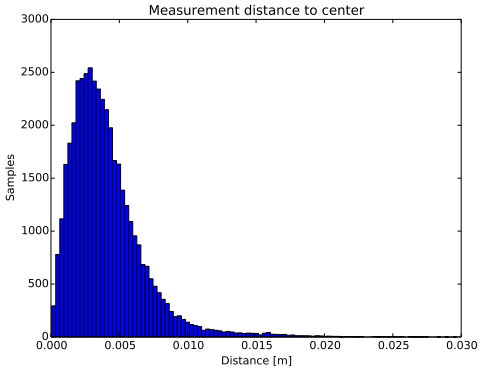
(a) Map of measurements with *float* and *fix* data.

(b) Map of measurements with only *fix* data.

Figure 3.6: Map of measurements in UTM coordinate. The red 'x' are *float* points, the green are *fix* points and the blue denotes the midpoint. The midpoint is calculated from *fix* data.



(a) Map of measurements with *fix* data.
Outliers are removed.



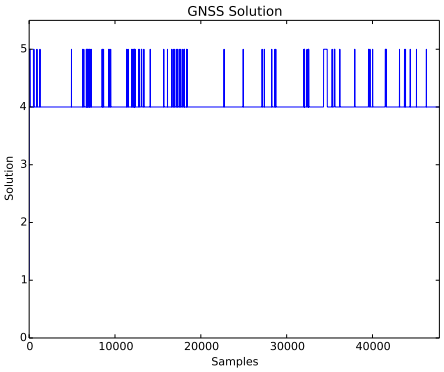
(b) Histogram of measurement distance to center.
Outliers are removed.

Figure 3.7: Plots of *fix* data with outliers removed. Outliers are measurements with a total distance larger than 0.03 [m] to center. 0.09% of *fix* data are seen as outliers.

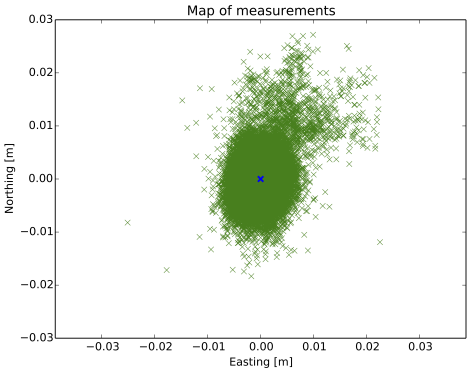
3.4.4 LEA-M8T

RTK GPS with ZEPHYR antenna

RTKLIB was configured to kinematic mode and to use GPS. The setup collected data for 13 hours and 17 minutes with a sample rate of 1Hz. A plot of the solution over time can be seen in figure 3.8a. Of the total data 3.1 % were *float solution* and 96.9% were *fix solution*. It is seen that the *fix solution* is fairly stable. A plot of *fix* data with outliers removed can be seen in figure 3.8b.



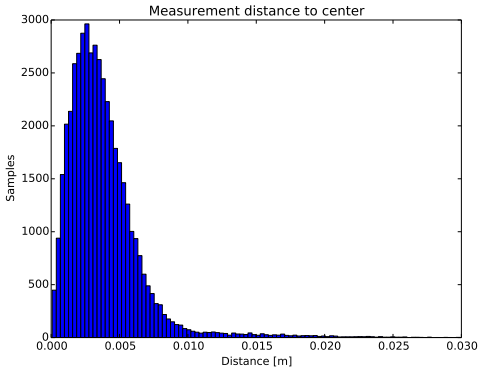
(a) GNSS solution over time.
4 = *RTK Fix* and 5 = *RKT Float*.



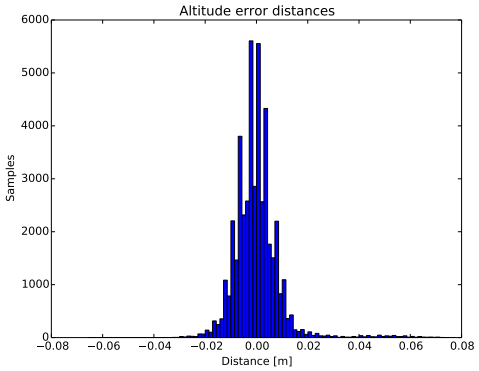
(b) Map of measurements with *fix* data. Outliers are removed.

Figure 3.8

A histogram of measurement distance to center can be seen in figure 3.9a. 95.3% of all data has a distance smaller than 1.2 cm from the midpoint and 94.3 % of all data is within 1 cm. A histogram of the altitude measurement distance to altitude mean is shown in figure 3.9b. It is seen that all *fix* data is within 8 cm when outliers are removed. Similar data was found.



(a) Histogram of measurement distance to center. Outliers are removed.

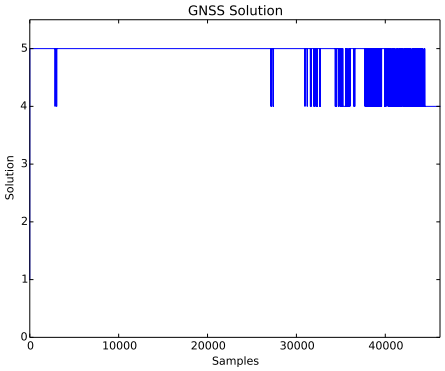


(b) Histogram of altitude measurement distance to altitude mean. Outliers are removed.

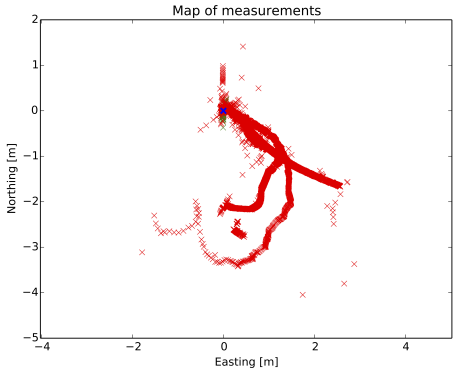
Figure 3.9: Histograms of the data with outliers removed. Outliers are measurements with a total distance larger than 0.03 [m] to center. 0.41% of *fix* data are seen as outliers.

RTK GPS and GLONASS with ZEPHYR antenna

RTKLIB was configured to kinematic mode and to use GPS and GLONASS. The setup collected data for 12 hours and 50 minutes with a sample rate of 1Hz. In average there were 12 verified satellites. A plot of the solution over time can be seen in figure 3.10a. Of the total data 87.2 % were *float solution* and only 12.8% were *fix solution*. It is seen that the *float solution* is fairly stable except for in the last small time period. Similar data was found, where the *float solution* was somewhat stable with only about 10% of the data being *fix solution*. A plot of all data can be seen in figure 3.10b. It is seen that the *float* points seem to been influenced by random-walk noise.



(a) GNSS solution over time.
4 = RTK *Fix* and 5 = RKT *Float*.



(b) Map of measurements with *float* and *fix* data in UTM coordinate. The red 'x' are *float* points, the green are *fix* points and the blue denotes the midpoint. The midpoint is calculated from *fix* data.

Figure 3.10

A histogram of measurement distance to center can be seen in figure 3.11. It is seen that most data points has an error > 0.1 meters. This means that using RTKLIB with GPS and GLONASS does not yield a precision that satisfies the goals of this project.

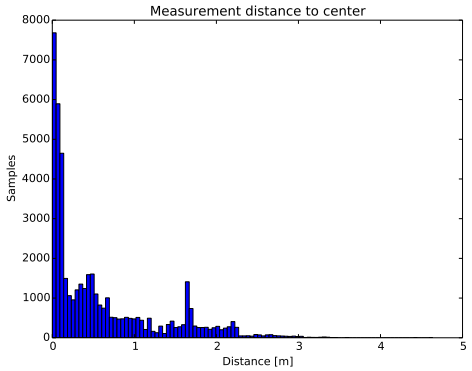


Figure 3.11: Histogram of measurements distance to center.

RTK GPS with small antenna

The setup with RTKLIB and only GPS yielded the highest precision. As the ZEPHYR is too big to mount on a small UAV the setup was tested with a smaller antenna. RTKLIB was configured to kinematic mode and to use GPS only. The setup collected data for 12 hours and 30 minutes with a sample rate of 1Hz. In average there were 9 verified satellites. A plot of the solution over time can be seen 3.12. It is seen that the solution jumps back and fourth from *float* to *fix*. Of the total data 49.3 % were *float solution* and 50.7% were *fix solution*. Half the data are *float solution* and therefore cannot just be discarded. A map plot of *float* and *fix solution* is shown in figure 3.13a. It is seen that the *float* points are spread more than the *fix* points. A histogram of the errors can be seen in figure 3.13b. It is seen that 95.3% of all data is within 15 centimeters of the calculated midpoint. 94% of all data is within 10 centimeters of the calculated midpoint. Similar data was found.

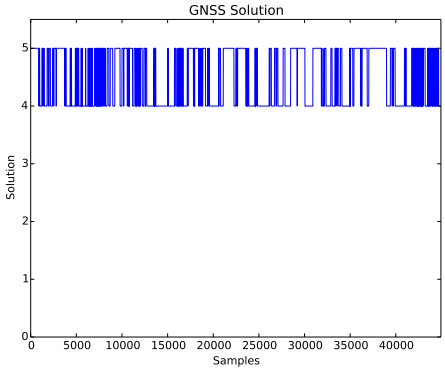
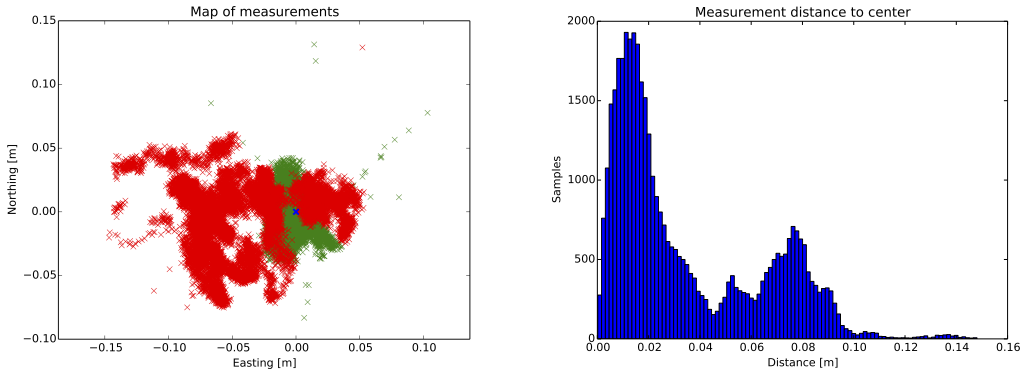


Figure 3.12: GNSS solution over time.
4 = RTK *Fix* and 5 = RKT *Float*.



(a) Map of measurements with *float* and *fix* data in UTM coordinate. The red 'x' are *float* points, the green are *fix* points and the blue denotes the midpoint. Outliers are removed.

(b) Histogram of measurements distance to center. Outliers are removed.

Figure 3.13: The midpoint is calculated from *fix* data. Outliers are measurements with a total distance larger than 0.15 [m] to center. 4.67% of all data are seen as outliers. Outliers are removed.

3.5 Conclusion

RTKLIB was used on a RPi to set up a working RKT system. A RTK reference station was setup with RTKLIB's `str2str` and data from it where broadcasted on a radio link and online. A RTK rover was setup with RTKLIB's `rtkrcv`, a radio link, LEA-6T and LEA-M8T GNSS receivers and two different antennas. Different setups were tested in a static test set-up. LEA-6T and LEA-M8T showed similar performance, when using GPS in kinematic mode and the ZEPHYR antenna. Using both GPS and GLONASS with LEA-M8T yielded low precision as most data was in *float solution*. LEA-M8T yielded satisfactorily high precision for the project. 94.3% of all data was within 1 cm to the midpoint. The ZEPHYR antenna is too big to be mounted on a UAV, therefore the RTK system was tested with a smaller antenna. The test showed that half of the data had *fix solution* and the other half *float solution*. 94% of all data had a smaller distance to the calculated midpoint than 10 centimers. This might be enough precision for flying within 5 centimeters of a specified route if GPS data is combined with data from other sensors in a filter.

4 Pixhawk

To fly an UAV requires a good pilot or a good autopilot. As this projects seeks to make autonomous precision flying, an autopilot is needed. There are several autopilots available on the market both proprietary and open-source projects. The latter is appealing for this project as it is possible to change the firmware if it is required in order to meet the project specifications. Pixhawk is an open-source and open-hardware autopilot that is used by many users worldwide. SDU has chosen to use this platform for all its UAV activities. Therefore Pixhawk is also chosen as the autopilot in this project.
This chapter explains the basics of the Pixhawk autopilot based on information from [4]. It also explains how GNSS data is used in conjunction with other sensor data to give a position estimate. Furthermore it explains how flight data can be logged and analyzed.

4.1 Hardware

The Pixhawk project has developed different hardware autopilots, the latest being the PX4 autopilot. The PX4 is a all-in-one module consisting of microprocessor, various sensors and I/O modules. The PX4 can be seen in figure 4.1.

Figure 4.1: PX4 autopilot module.

4.1.1 Microprocessor

The microprocessor is a Cortex-M4F with the following specifications [5]:

- 32 bit
- 168 MHz
- 256 KB RAM
- 2 MB Flash

The logic level on PX4 is 3.3V.

4.1.2 Sensors

The internal sensors on the board is:

- ST Micro L3GD20H 16 bit gyroscope
- ST Micro LSM303D 14 bit accelerometer / magnetometer
- MEAS MS5611 barometer

The gyro, accelerometer and magnetometer are all 3-axis.
External sensors can be added. Commonly a GNSS module, which would typically be the LEA-6H GPS module, would be added alongside an external magnetometer. A safety switch, two telemetry links and a buzzer are also normally added.

4.2 Operating system

Pixhawk uses NuttX Real-Time Operating System (RTOS) as the lowest level on the microcontroller. NuttX has a simple shell application called NuttShell (NSH), which can perform many standard shell applications. NuttX schedules tasks to the processor by running a round-robin scheduling with priorities. The priorities are as follows: fast sensor drivers, watchdogs/system state monitors, actuator outputs, attitude controllers, slow/blocking sensors, position controllers, default priority and logger.
At system startup a startup script located at `/etc/init.d/rcS` will start all the necessary applications, including the ones that together form the middleware.

4.3 Middleware

The Pixhawk middleware runs on top of NuttX. It provides drivers for various devices and a micro Object Request Broker (uORB). uORB is used to share data structures between threads. The middleware is shown in the blue and green boxes in figure 4.2. A description of all applications can be found in appendix A.

4.4 Flight Control Stack

On top of Pixhawk middleware there are two possibilities: Pixhawk flight stack and APM flight stack. Both flight stacks are fully operational, open-source and has many users. This project will only focus on the Pixhawk flight stack. The Pixhawk firmware does not support NMEA 0183 protocol and therefore a custom driver needs to be developed in order to make it work with output from RTKLIB. The Pixhawk flight control stack applications are shown in the red boxes of figure 4.2. A short description of all applications can be found in appendix A.

4.5 Inter Process Communication

Inter process communication is done by uORB in Pixhawk. It works as a simple publish-subscribe pattern. Applications can subscribe or publish to different buses of data, called topics. The concept is shown in figure 4.3. Applications do not know which applications subscribe to which topics and a topic may have multiple publishers and subscribers. A topic contains only one message, often a user defined struct, and a topic is not a queue. A topic declaration and registration is shown in listing 4.1.

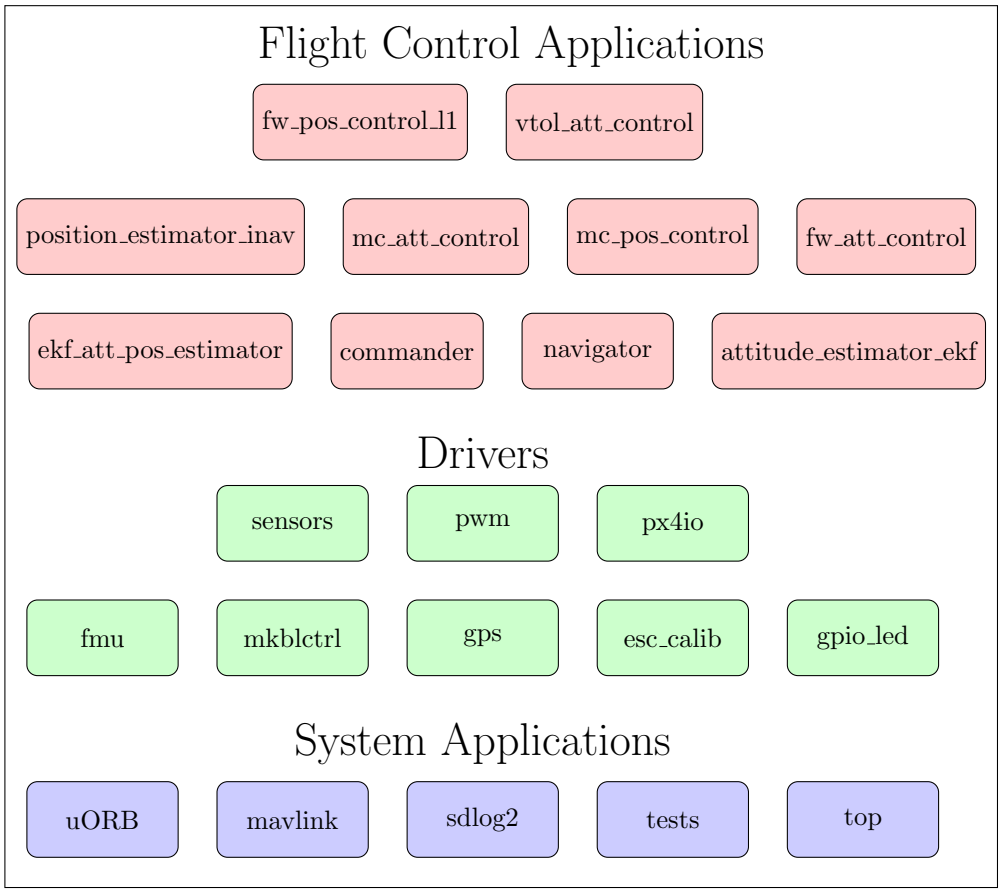


Figure 4.2: All onboard applications on Pixhawk. The green and blue applications form the middleware layer. The red applications form the Pixhawk flight stack.

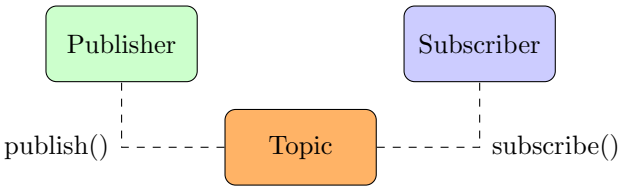


Figure 4.3: Inter process communication on Pixhawk.

```
struct vehicle_gps_position_s {
  uint64_t timestamp_position;
  int32_t lat;
  int32_t lon;
  int32_t alt;

                                        ⋮

  uint8_t satellites_used;
};

ORB_DECLARE(vehicle_gps_position);
```

Listing 4.1: Part of the declaration of vehicle_gps_position_s struct and registration of topic.

4.6 GPS driver

The GPS driver is located in `/Firmware/src/drivers/gps`. There are drivers for ashtech, mtk and ubx receivers. Ashtech receivers use NMEA protocol, but in a custom way not corresponding to the NMEA standards. There is no driver that can obtain correct information from data in NMEA 0183 format. This needs to be developed in order to use the RTK rover developed in the previous chapter.

4.7 Position estimation

Position estimation is done in the `ekf_att_pos_estimator` application. It uses data from IMU, barometer, magnetometer and GPS. Data from IMU and barometer only contain information of the relative position, while GPS and magnetometer data contain information of position in global position. Merging the data together using a filter allows for a removal of noisy data and to compensate for low sample rate on GPS data. The Kalman filter uses the variance of each sensor to estimate the states of the system. The filter in `ekf_att_pos_estimator` is a 22 state extended Kalman filter. The vehicles position in 3D space is represented by three of these states.

4.8 Logging data

The `sdlog2` application automatically starts logging data to the SD card when the Pixhawk is flying. `sdlog2` creates a directory for each flight and logs data to a `.bin` or `.px4log` file.

4.8.1 Analyzing logs

Logs can be analyzed directly using the tool FlightPlot. It can use `.bin` or `.px4log` files directly and can plot different parameters directly. An example plot of GPS latitude data in FlightPlot can be seen in figure 4.4.

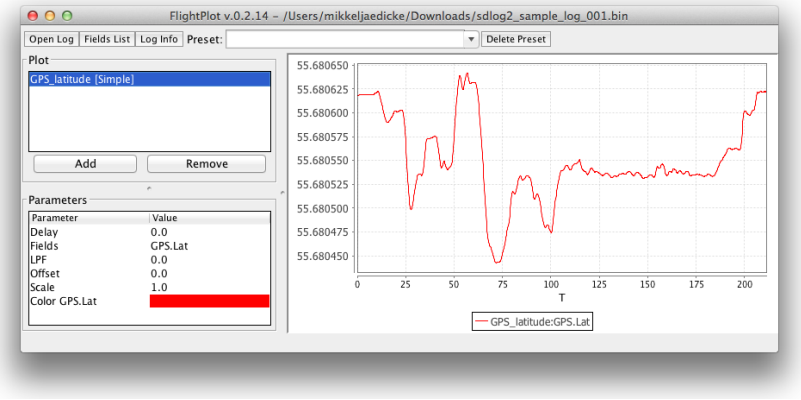


Figure 4.4: Plotting of GPS latitude data with FlightPlot tool.

The `.bin` log files can also be converted to `.csv` files with the python tool `sdlog2_dump.py`. A command for converting all data from a log file into a `.csv` file using `sdlog2_dump.py` is shown in listing 4.2.

```
1 python sdlog2_dump.py log001.bin -f "export.csv" -t "TIME" -d "," -n ""
```

Listing 4.2: Converting all data in log001.bin to export.csv.

If only some data from a log file is needed e.g. GPS data the command in listing 4.3 can be used. If IMU data is also needed `-m IMU` is simply added. A description of message types can be found in [6].

```
1 python sdlog2_dump.py log001.bin -t TIME -m TIME -m GPS -e > gps_log.csv
```

Listing 4.3: Converting all GPS data in log001.bin to gps_log.csv.

4.9 QGroundControl

QGroundControl is an open-source software ground control station developed for Pixhawk flight stack. It is a GUI tool that runs on both Windows, Linux and OS X. It can be used for calibration of the Pixhawk as well as calibration of a corresponding rc controller. It can upload firmware and it can be used to send commands to NSH.

4.10 Conclusion

Pixhawk is an open-source project autopilot in both software and hardware. The Pixhawk hardware module is PX4, which is a Cortex-M4F 168 MHz processor with internal IMU, magnetometer and barometer. NuttX RTOS runs on the Pixhawk. Pixhawk middleware runs on top of NuttX and offers various drivers and uORB to inter process communication. GPS, magnetometer, barometer and IMU data is merged into a 3D position estimate using an extended Kalman filter. Data is logged for each flight by the `sdlog2` application. The log files can be analyzed directly by using `FlightPlot` or can be converted to `.csv` files. QGroundControl can be used for sending commands to NSH, calibrate sensors and upload firmware. The GPS driver does not support the NMEA 0183 protocol, meaning that in order to use the developed RTK rover a new GPS protocol needs to be created.

5 Developing Pixhawk Software

As revealed in the previous chapter, the Pixhawk flight stack does not support NMEA 0183 messages, which is the output from the RTK rover. Therefore there is a need for developing a new driver. This chapter will describe in general how developing for Pixhawk is done and it will be described how a GNSS driver is implemented. Ubuntu 14.04 was used for developing.

5.1 Preliminaries

Before being able to program the Pixhawk there are some preliminaries, which will be stated here.

5.1.1 Toolchain Installation

Toolchain installation is straightforward and a thorough guide can be found in [7].

5.1.2 Downloading, Building and flashing Firmware

The firmware is available on Github and was cloned to the hard drive. Then the submodules were fetched as well. The commands used are shown in listing 5.1.

```
cd ~/src
git clone https://github.com/PX4/Firmware

cd Firmware
git submodule init
git submodule update
```

Listing 5.1: Shell commands for cloning Pixhawk firmware to directory src and fetching submodules.

Then the operating system was compiled. This took some time and only needs to be done once or if changes are made to the operating system. The command is shown in 5.2.

```
make archives
```

Listing 5.2: Shell command for compiling operating system.

Then the the rest of the firmware was compiled. Firstly the application build was cleaned, this needs to be done if new modules or drivers are added. Then the firmware was built and uploaded to the PX4. The commands used are shown in listing 5.3.

```
make clean
make px4fmu-v2_default
make upload px4fmu-v2_default
```

Listing 5.3: Shell commands for cleaning application build, compiling and uploading the firmware.

The PX4 board obviously needs to be connected to the computer, when uploading.

5.1.3 Using NutShell

NuttShell (NSH) is, as mentioned before, a simple shell application for NuttX. NSH can be reached through serial port 5 on PX4 and on the USB. It is possible to connect to NSH through a serial client or through QGroundControl. In this project USB and QGroundControl were used to connect to NSH.

5.2 Existing GNSS drivers

All drivers in the firmware are placed in the directory `/Firmware/src/drivers`. The GPS driver is located in the directory gps, while the gps driver's .h file `drv_gps.h` is placed directly in the driver directory. The GPS driver itself consist of `gps.cpp` and `gps_helper.cpp`. `gps.cpp` will start a main loop thread. The main loop thread will use one of the protocols to receive data from the serial port and translate it to data, which can be used to update the uORB topic `vehicle_gps_position`. The existing protocols are ashtech, mtk and ubx.

5.3 Custom NMEA 0183 protocol

The custom NMEA 0183 protocol is based upon `ashtech.cpp`. It is named `rtklib.cpp` as users will easily recognize that it works with RTKLIB. Changes were also made to the files `gps.cpp` and `drv_gps.h`. Only the most important changes will be explained here. All source code can be found on Github as described on the titlepage.
Data is extracted from the serial port and put into a buffer. When a full message is received, checksum is calculated and if it is correct the message is parsed to the `handle_message` function. If the message is a $GPGGA message data will be extracted from it. Latitude and longitude is in degree minutes format and needs to converted into decimal degree format. Lastly the uORB topic `vehicle_gps_position_s` is updated with the new data. `vehicle_gps_position_s` holds a lot of data. Only latitude, longitude, altitude, timestamp, fix_type and satellites are updated.

5.4 Conclusion

Toolchain installation was done and the Pixhawk firmware was downloaded. QGroundControl was used to access NSH and send commands to PX4. A new GNSS driver compatible with NMEA 0183 was developed that updates the uORB topic `vehicle_gps_position_s` with latitude, longitude, altitude, timestamp, fix_type and satellites information.

6 RTK GNSS on Pixhawk

The preceding chapters have described the Pixhawk autopilot, the development of a GNSS driver for Pixhawk and the making of a RTK rover on a RPi with RTKLIB. This chapter will describe how the RTK rover and the PX4 was combined into one system. It will describe how it was done and what the results were.

6.1 UAV platform

The UAV platform used for this project is an IRIS drone. The autopilot on the IRIS drone is, of course, a Pixhawk autopilot. The only feature used from the IRIS platform is an u-blox LEA-6H magnetometer and GPS receiver. Only the magnetometer was connected to the PX4 when doing tests. Since no flying was done it should not change anything that an IRIS drone was used. Any other UAV with the PX4 could be used instead.

6.2 RTKLIB and Pixhawk

The custom firmware with the NMEA 0183 driver was uploaded to the PX4. The PX4 serial port's rx pin was connected to the tx pin on the RPi RTK rover and the RPi's ground pin was connected to the ground pin on the PX4. The RPi's rx pin and the Pixhawk's tx pin was not connected, as the RPi does not read the pin anyway. The RTK rover on RPi was booted and so was the PX4 and QGroundControl. In QGroundControl a connection was made to NSH. `gpsstatus` command was sent to NSH. The PX4 answered with GPS driver status and it was observed that the custom RTK protocol was in use and that status was `OK`. The shown latitude and longitude values was compared to latitude and longitude values on the RTK rover and they were consistent. This test has been done a couple of times and everything seems to work.
A screenshot of the connection to NSH in QGroundControl is shown in figure 6.1.

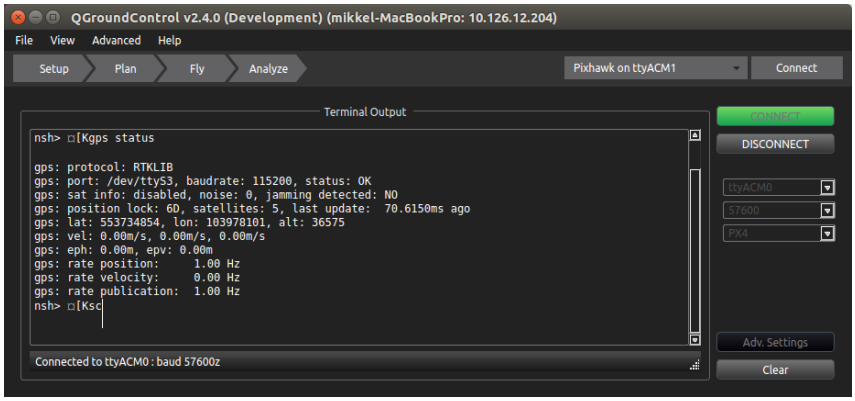


Figure 6.1: Connection to NSH in QGroundControl. Shown is gps driver status.

When connection were made to PX4 in ordinary way in QGroundControl the communication console showed messages that stated *GPS signal found, GPS timeout, GPS signal found* continuously. This indicates an error in the firmware. After a while, approximately 30 seconds, the stream of messages terminated with the last message being *GPS timeout*. This indicates that the custom protocol stopped working. NSH command *gps status* was sent and it showed that the RTK protocol was still in use and the status was `OK`. Latitude and longitude information was also still being updated and the protocol was still running. The scenario explained in this section has been repeated and showed the same error.

6.3 Determining the problem

To investigate what went wrong data was logged with `sdlog2`. The data showed that all values in `vehicle_gps_position_s` updated by the custom RTKLIB protocol was correct. This shows that the custom protocol updates the values correctly. There were no more data after 30 seconds, which matches that after 30 seconds the last *GPS timeout* message were received.
The estimated vehicle position was also checked in order to see if the Kalman filter was working correctly. The values were completely off and the position estimate had enormous errors. This indicates that the errors are introduced in the Kalman filter. Since the code for the Kalman filter was not changed the problem cannot lie there. The error must be that not all variables in `vehicle_gps_position_s` are updated. Some of the variables that were not updated, must be used by the Kalman Filter and when they are invalid the Kalman filter does not work. This problem should be solved by updating all variables in `vehicle_gps_position_s`. In the current version of the custom driver, only information from $GPGGA messages are used to update `vehicle_gps_position_s`. The driver should be further developed to use information from all NMEA messages to update `vehicle_gps_position_s`. Then the Kalman filter should get all the information needed to operate properly. A problem might be that not all information for `vehicle_gps_position_s` is given by the RTK rover. If the missing information is needed by the Kalman filter the RTK rover needs to be modified.

6.4 Conclusion

The custom GNSS protocol seems to work when connected to the PX4 through NSH. Using the *gps status* command shows that the GPS status was `OK` and correct updates of latitude, longitude and satellites in use. The logged data showed that the values updated by the custom protocol were correct, but it also showed that the estimated vehicle position was far off. This means that the error must be that the Kalman filter uses values that are not updated by the custom protocol. The custom protocol should be further developed to update more values to `vehicle_gps_position_s`. The problem was discovered late in the projectwork and was therefore not corrected due to time constraints.

7 Conclusion

An analysis of the possibilities of using GNSS for high precision navigation was made. The outcome was that RTK GNSS should be used in order to achieve the highest possible positioning.

RTKLIB was used to make a working RTK system with rover and reference station. Both were set up on RPi. Static test were made with LEA 6-T and LEA M8T GNSS receivers and with a ZEPHYR and a small GNSS antenna. The test were designed to show the precision of the system. LEA 6-T and LEA M8T yielded approximately the same precision, when using RTK and GPS. The precision was satisfactory high and 94.3% of all data, when using LEA M8T, was within 1 cm. Using GLONASS and GPS together yielded low precision as most data was *float* solution. The ZEPHYR antenna is too big to mount on a small UAV, therefore the RTK system was tested with a small GNSS antenna. It showed good precision, but not as high as with the ZEPHYR antenna. 94% of all data was within 10 centimeters. This precision is not good enough, but data could be combined with sensors from an IMU in a filter to yield a better positioning.

The Pixhawk autopilot was chosen as SDU has chosen to use it for most of its UAV activities. Pixhawk is a complete open-source autopilot with both hardware and software. The software were analyzed and it was found that there was a need for a new driver in order to use data from the RTK rover. Logging is already a part of the Pixhawk firmware and is done by the `sdlog2` application automatically. State estimation is done in a 22 state extended Kalman filter that used date from the IMU, magnetometer, barometer and GNSS receiver.

A GNSS driver was developed to receive data in NMEA 0183 format. It updates some of the values of uORB topic `vehicle_gps_position_s`. The RPi RTK rover was connected to the PX4 and connection through NSH showed that the custom protocol were in use and updated essential position data such as latitude and longitude. Examination of the logs showed that the values updated by the custom protocol were updated correctly, but it also showed that the results of the state estimation done in the Kalman filter was far off. It was concluded that the cause of it must be that the custom driver does not update all variables in `vehicle_gps_position_s`. Some of variables that were not updated must be used by the Kalman filter in order for it to work properly. The problem should be corrected by further developing the custom driver to use all information from the RKT rover to update `vehicle_gps_position_s`.

The hypothesis of the project was to test if it was possible to get a drone to fly within $\pm$ 5 centimeters of a specified route 95% of the time. The project has not been able to accept or decline the hypothesis. However a working RTK system with rover and reference station that can be used with the Pixhawk autopilot was developed. The results of the RTK system alone were promising and the precision should be high enough to achieve positioning errors less than $\pm$ 5 centimeters when using a big GNSS antenna.

8 Perspectives

Clearly more work should be put in the development of the GNSS driver. When the driver is up and working it would be good to test the performance of the Kalman filter. This could be done by making static test or by moving in a known path.

An investigation should be made to determine what antenna would be best suited for use with small UAVs. The ZEPHYR antenna was clearly too big and the small GNSS antenna was maybe too small. An antenna size in between the two might be the best option.

Better precision for the RTK system should be possible by receiving data on both L1, L2 and L5 frequencies. The RTK system should be further developed to use correction data from commercial RKT base station, as it would give more opportunities for commercial use.

Bibliography

[1] Kjeld Jensen. *An Application-oriented Open Software Platform for Multi-purpose Field Robotics*. PhD thesis, Faculty of Engineering, University of Southern Denmark, 2014.

[2] Elliott D. Kaplan. *Understanding GPS: Principles and Applications, Second Edition*. Artech House, 2005. ISBN 1580538940.

[3] National Marine Electronics Association. *NMEA 0183 Standard For Interfacing Marine Electronic Devices*. 2012. URL `http://www.plaisance-pratique.com/IMG/pdf/NMEA0183-2.pdf`. Accessed 31. May 2015.

[4] Pixhawk. *PX4 Developer Documentation*. 2015. URL `https://pixhawk.org/dev/start`. Accessed 31. May 2015.

[5] Pixhawk. *PX4 Developer Documentation*. 2015. URL `https://pixhawk.org/modules/pixhawk`. Accessed 31. May 2015.

[6] Pixhawk. *PX4 Developer Documentation*. 2015. URL `https://pixhawk.org/firmware/apps/sdlog2`. Accessed 31. May 2015.

[7] Pixhawk. *PX4 Developer Documentation*. 2015. URL `https://pixhawk.org/dev/toolchain_installation`. Accessed 31. May 2015.

[8] Takasu Tomoji. *RTKLIB ver. 2.4.2 Manual*, 2013. URL `http://www.rtklib.com/prog/manual\_2.4.2.pdf`. Accessed 31. May 2015.

A List of onboard applications on Pixhawk

System Applications

mavlink

The mavlink application is used for communication from the Pixhawk to the outer world on a serial port. It sends and receives MAVLink packets and updates uORB topics with the data.

sdlog2

Logs flight data to a SD card. The application is described further in section 4.8.

tests

This application is capable of testing interfaces and sensors on PX4. It is launched from NSH.

top

This application shows the system usage. It is launched from NSH.

uORB

This application is responsible for inter process communication. It is described further in section 4.5.

System Applications

mkblctrl

This application sets up the motors according to predefined model configurations.

esc_calib

This application is used to calibrate the ESCs on the drone. It is launched from NSH. All propellers should be removed before performing the calibration.

fmu

This application configures all gpio pins on the PX4.

pwm

This application configures PWM output ports for servo and ESC control.

sensor

This application collects all data from on-board sensors that is IMU, magnetometer and barometer. The corresponding uORB topic is updated with the data.

px4io

This application interfaces to PX4IO. It also receives battery voltage, current and rc input.

uavcan

This application interfaces to all CAN devices using the UAVCAN protocol.

Flight Control Applications

commander

This application runs the global state machine for the system. The LED and buzzer are controlled to indicate the current state. The global state machine includes states like armed, standby and emergency landing.

navigator

This application is responsible for all modes, where the autopilot controls the UAV autonomously. This includes missions, failsafe and return-to-land.

attitude_estimator_ekf

This application is a extended Kalman filter that estimates the attitude of the UAV. It is only used when neither ekf_att_pos_estimator or position_estimator_inav are used.

ekf_att_pos_estimator

This application is a extended Kalman filter that estimates the attitude and position of the UAV. It is further described in section 4.7. It is only used when neither attitude_estimator_ekf or position_estimator_inav are used.

position_estimator_inav

This application estimates the position of the UAV by inertial sensor data. It is only used when neither attitude_estimator_ekf or ekf_att_pos_estimator are used.

mc_att_control

This application holds the attitude control loop for multirotor UAVs.

mc_pos_control

This application holds the position control loop for multirotor UAVs.

fw_att_control

This application holds the attitude control loop for fixed wing UAVs.

fw_pos_control_l1

This application holds the position control loop for fixed wing UAVs.

vtol_att_control

This application holds the attiude control loop for fixed wing vertical take-off and landing UAVs.

B rtkrcv settings

```
# RTKRCV options (2015/05/27)

pos1-posmode       =kinematic  # (0:single,1:dgps,2:kinematic,3:static,4:movingbase,5:fixed,6:ppp-kine,7:ppp-static)
pos1-frequency     =l1         # (1:l1,2:l1+l2,3:l1+l2+l5,4:l1+l2+l5+l6,5:l1+l2+l5+l6+l7)
pos1-soltype       =forward    # (0:forward,1:backward,2:combined)
pos1-elmask        =10         # (deg)
pos1-snrmask_r     =off        # (0:off,1:on)
pos1-snrmask_b     =off        # (0:off,1:on)
pos1-snrmask_L1    =0,0,0,0,0,0,0,0,0
pos1-snrmask_L2    =0,0,0,0,0,0,0,0,0
pos1-snrmask_L5    =0,0,0,0,0,0,0,0,0
pos1-dynamics      =off        # (0:off,1:on)
pos1-tidecorr      =off        # (0:off,1:on)
pos1-ionoopt       =brdc       # (0:off,1:brdc,2:sbas,3:dual-freq,4:est-stec,5:ionex-tec,6:qzs-brdc,7:qzs-lex,8:vtec_sf,9:vtec_ef,10:gtec)
pos1-tropopt       =saas       # (0:off,1:saas,2:sbas,3:est-ztd,4:est-ztdgrad)
pos1-sateph        =brdc       # (0:brdc,1:precise,2:brdc+sbas,3:brdc+ssrapc,4:brdc+ssrcom)
pos1-posopt1       =off        # (0:off,1:on)
pos1-posopt2       =off        # (0:off,1:on)
pos1-posopt3       =off        # (0:off,1:on)
pos1-posopt4       =off        # (0:off,1:on)
pos1-posopt5       =off        # (0:off,1:on)
pos1-exclsats      =           # (prn ...)
pos1-navsys        =1          # (1:gps+2:sbas+4:glo+8:gal+16:qzs+32:comp)
pos2-armode        =continuous # (0:off,1:continuous,2:instantaneous,3:fix-and-hold)
pos2-gloarmode     =on         # (0:off,1:on,2:autocal)
pos2-arthres       =3
pos2-arlockcnt     =0
pos2-arelmask      =0          # (deg)
pos2-arminfix      =10
pos2-elmaskhold    =0          # (deg)
pos2-aroutcnt      =5
pos2-maxage        =30         # (s)
pos2-slipthres     =0.05       # (m)
pos2-rejionno      =30         # (m)
pos2-rejgdop       =30
pos2-niter         =1
pos2-baselen       =0          # (m)
pos2-basesig       =0          # (m)
out-solformat      =llh        # (0:llh,1:xyz,2:enu,3:nmea)
out-outhead        =off        # (0:off,1:on)
out-outopt         =off        # (0:off,1:on)
out-timesys        =gpst       # (0:gpst,1:utc,2:jst)
out-timeform       =hms        # (0:tow,1:hms)
out-timendec       =3
out-degform        =deg        # (0:deg,1:dms)
out-fieldsep       =
out-height         =ellipsoidal # (0:ellipsoidal,1:geodetic)
out-geoid          =internal   # (0:internal,1:egm96,2:egm08_2.5,3:egm08_1,4:gsi2000)
out-solstatic      =all        # (0:all,1:single)
out-nmeaintv1      =0.2        # (s)
out-nmeaintv2      =0          # (s)
out-outstat        =off        # (0:off,1:state,2:residual)
stats-eratio1      =100
stats-eratio2      =100
stats-errphase     =0.003      # (m)
stats-errphaseel   =0.003      # (m)
stats-errphasebl   =0          # (m/10km)
stats-errdoppler   =1          # (Hz)
stats-stdbias      =30         # (m)
stats-stdiono      =0.03       # (m)
stats-stdtrop      =0.3        # (m)
stats-prnaccelh    =10         # (m/s^2)
stats-prnaccelv    =10         # (m/s^2)
```

```
stats-prnbias       =0.0001      # (m)
stats-prniono       =0.001       # (m)
stats-prntrop       =0.0001      # (m)
stats-clkstab       =5e-12       # (s/s)
ant1-postype        =llh         # (0:llh,1:xyz,2:single,3:posfile,4:rinexhead,5:rtcm)
ant1-pos1           =0           # (deg|m)
ant1-pos2           =0           # (deg|m)
ant1-pos3           =0           # (m|m)
ant1-anttype        =
ant1-antdele        =0           # (m)
ant1-antdeln        =0           # (m)
ant1-antdelu        =0           # (m)
ant2-postype        =llh         # (0:llh,1:xyz,2:single,3:posfile,4:rinexhead,5:rtcm)
ant2-pos1           =55.374183827 # (deg|m)
ant2-pos2           =10.398601242 # (deg|m)
ant2-pos3           =70.968 # (m|m)
ant2-anttype        =
ant2-antdele        =0           # (m)
ant2-antdeln        =0           # (m)
ant2-antdelu        =0           # (m)
misc-timeinterp     =off         # (0:off,1:on)
misc-sbasatsel      =0           # (0:all)
misc-rnxopt1        =
misc-rnxopt2        =
file-satantfile     =
file-rcvantfile     =
file-staposfile     =
file-geoidfile      =
file-ionofile       =
file-dcbfile        =
file-eopfile        =
file-blqfile        =
file-tempdir        =C:\Temp
file-geexefile      =
file-solstatfile    =
file-tracefile      =
inpstr1-type        =serial      # (0:off,1:serial,2:file,3:tcpsvr,4:tcpcli,7:ntripcli,8:ftp,9:
    http)
inpstr2-type        =serial      # (0:off,1:serial,2:file,3:tcpsvr,4:tcpcli,7:ntripcli,8:ftp,9:
    http)
inpstr3-type        =off         # (0:off,1:serial,2:file,3:tcpsvr,4:tcpcli,7:ntripcli,8:ftp,9:
    http)
inpstr1-path        =ttyACM0:115200:8:n:1:off
inpstr2-path        =ttyUSB0:57600:8:n:1:off
inpstr3-path        =
inpstr1-format      =ubx         # (0:rtcm2,1:rtcm3,2:oem4,3:oem3,4:ubx,5:ss2,6:hemis,7:skytraq
    ,8:gw10,9:javad,15:sp3)
inpstr2-format      =ubx         # (0:rtcm2,1:rtcm3,2:oem4,3:oem3,4:ubx,5:ss2,6:hemis,7:skytraq
    ,8:gw10,9:javad,15:sp3)
inpstr3-format      =rtcm2       # (0:rtcm2,1:rtcm3,2:oem4,3:oem3,4:ubx,5:ss2,6:hemis,7:skytraq
    ,8:gw10,9:javad,15:sp3)
inpstr2-nmeareq     =off         # (0:off,1:latlon,2:single)
inpstr2-nmealat     =0           # (deg)
inpstr2-nmealon     =0           # (deg)
outstr1-type        =serial              #
#outstr1-type        file       # (0:off,1:serial,2:file,3:tcpsvr,4:tcpcli,6:ntripsvr)
outstr2-type        =file
#outstr2-type        =serial     # (0:off,1:serial,2:file,3:tcpsvr,4:tcpcli,6:ntripsvr)
outstr1-path        =ttyAMA0:115200:8:n:1:off
outstr2-path        =output_data/test_nmea
#outstr2-path        =ttyAMA0:115200:8:n:1:off
#outstr2-path        =../../../../solution_%Y%m%d_%h%M.nmea
outstr1-format      =nmea         # (0:llh,1:xyz,2:enu,3:nmea)
outstr2-format      =nmea        # (0:llh,1:xyz,2:enu,3:nmea)
logstr1-type        =off         # (0:off,1:serial,2:file,3:tcpsvr,4:tcpcli,6:ntripsvr)
logstr2-type        =off         # (0:off,1:serial,2:file,3:tcpsvr,4:tcpcli,6:ntripsvr)
logstr3-type        =off         # (0:off,1:serial,2:file,3:tcpsvr,4:tcpcli,6:ntripsvr)
logstr1-path        =
logstr2-path        =
logstr3-path        =
misc-svrcycle       =10          # (ms)
```

```
misc-timeout        =10000      # (ms)
misc-reconnect      =10000      # (ms)
misc-nmeacycle      =5000       # (ms)
misc-buffsize       =32768      # (bytes)
misc-navmsgsel      =all        # (0:all,1:rover,2:base,3:corr)
misc-proxyaddr      =
misc-fswapmargin    =30         # (s)
```

Listing B.1: rtkrcv.conf