

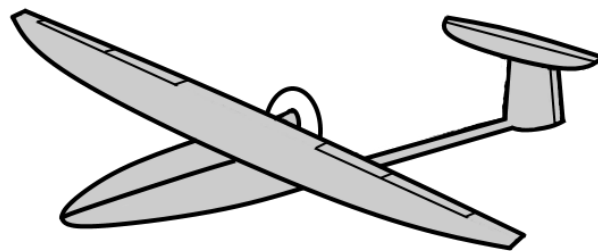
Petter Diderik Breedveld

Concept for Landing Unmanned Aerial Vehicles using a Ground Based Augmentation System

Master's thesis in Cybernetics and Robotics

Supervisor: Vendela Paxal and Nadezda Sokolova

June 2019



Petter Diderik Breedveld

Concept for Landing Unmanned Aerial Vehicles using a Ground Based Augmentation System

Master's thesis in Cybernetics and Robotics
Supervisor: Vendela Paxal and Nadezda Sokolova
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Abstract

This thesis, *Concept for Landing Unmanned Aerial Vehicles using a Ground Based Augmentation System*, focuses on the possibilities of autonomous landing of commercial Unmanned Aerial Vehicles (UAVs) by integrating the existing Ground Based Augmentation System (GBAS) with UAV avionics.

Potential ways of integrating GBAS and avionics signals were developed and resulted in a concept which was used to design the necessary hardware and software for a prototype application. The system was test fitted in an existing UAV system at Andøya Space Center (ASC).

The study resulted in a system that could be integrated neatly with existing avionics, without significant modification of the existing UAV hardware and requiring only minimal modification of the airframe.

Opensource software was successfully modified for the reception of GBAS signals. The contained data in the broadcast are made available for correcting the GPS pseudoranges.

GPS pseudoranges are smoothed using the carrier phase, so that corrections can be applied. This results in augmented position data. The consequences of pseudorange jumps were assessed, and mitigation methods were proposed.

Two different hardware prototypes were developed based on single-board computers and tested. Only the Raspberry Pi based system was able to handle the radio samples and communicate with the UAV avionics at ASC.

GBAS implementation will improve autonomous landing capabilities of UAVs, independent of airfield and weather conditions. This will extend the capabilities of unmanned operations in the future.

Sammendrag

Denne oppgaven, *Concept for Landing Unmanned Aerial Vehicles using a Ground Based Augmentation System*, fokuserer på mulighetene for autonom landing av kommersielle ubemannede luftfartøy (UAV) ved å integrere det eksisterende Ground Based Augmentation Systemet (GBAS) med UAV avionikk.

Ulike måter å integrere GBAS og avionikk signaler ble utviklet, og resulterte i et konsept som ble brukt til å designe den nødvendig maskin- og programvaren for en prototype. Systemet ble test-montert i en eksisterende UAV ved Andøya Space Center (ASC).

Studien resulterte i et system som enkelt kunne integreres med eksisterende avionikk, uten signifikant modifikasjon av eksisterende UAV-maskinvare, og som bare krever minimal modifikasjon av flyet.

Et program med åpen kildekode ble modifisert for mottak av GBAS-signaler. Datainnholdet i kringkastingen blir gjort tilgjengelig for å korrigere GPS-pseudorange målingene.

GPS-pseudorangene blir utjevnet ved hjelp av bæreølgen, slik at GBAS korreksjonene kan brukes. Dette resulterer i forbedret posisjonsberegning. Konsekvensene av hopp i pseudorange målingen ble vurdert, og metoder ble foreslått for å minimere effekten.

To forskjellige maskinvare prototyper ble utviklet basert på ettkortsdatamaskiner og testet. Bare det Raspberry Pi baserte systemet klarte å håndtere radiodekodingen og kommunisere med UAV-avionikken ved ASC.

GBAS-implementering vil forbedre autonome landingsegenskaper for UAV, uavhengig av flyplass og værforhold. Dette vil utvide mulighetene for ubemannede operasjoner i fremtiden.

Preface

This thesis work forms the final part of my Master of Technology studies in Cybernetics and Robotics at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU).

This study has been performed as a collaboration between Indra Navia, Andøya Space Center (ASC) and NTNU.

I would like to specially thank my supervisors, Vendela Paxal (Indra Navia) and Nadia Sokolova (NTNU) for taking interest in my work and providing good advice and continuous support along the way.

Jostein Sveen at ASC took time out of his busy schedule to host me during my visit at ASC, and was always willing to answer any of my questions regarding their UAV systems.

This work is based on my previous project assignment [1] as well as a number of GBAS related standards [2] [3] [4] supplied by Indra Navia. The UAV documentation [5] [6] [7] [8] [9] [10] and UAV hardware for testing have been supplied by ASC. The GBAS demodulation software developed as part of this project is strongly based on open source code by Lemiech [11].

Indra Navia was so kind to offer me office space at their facilities at Asker, which allowed me to be part of Vendela's team. This made for a great learning environment and I would like to thank Morten Topland, Thomas Jøndal and Asgaut Eng for their technical insight and interesting discussions.

Being introduced to the world of aviation standards was quite a discovery, and made me aware of the many requirements that have to be fulfilled before a system can be approved for full-scale operation.

The last 6 months have been an intensive learning period, within a field that has been one of my great interests for a long time. I have thoroughly enjoyed the opportunity given to me by Indra and NTNU to experience this adventure.

Petter D. Breedveld

Asker, May 2019

Table of Contents

List of Figures	xi
List of Tables.....	xii
List of Abbreviations.....	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
2 Background.....	3
2.1 GPS.....	3
2.2 Differential Processing	5
2.3 UAV.....	5
2.4 Landing procedures at ASC.....	6
2.5 Limitations of the existing system	8
3 GBAS	9
3.1 Signal Broadcast	10
3.1.1 D8PSK modulation	11
3.1.2 The data burst	12
3.1.3 The GBAS message	13
3.2 UAV application.....	14
3.3 System design	15
4 Avionics Software	16
4.1 The AP-GPS interface	16
4.1.1 Error detection.....	16
4.1.2 Autopilot initialisation	17
4.1.3 GBAS-required data.....	18
4.2 Demodulating the GBAS broadcast	19
4.2.1 VHF Data Link Mode 2.....	20
4.2.2 DumpVDL2.....	22
4.3 Decoding the GBAS broadcast.....	24
4.3.1 CRC32 redundancy check	24
4.3.2 Processing datafields	25
4.4 GPS calculations.....	25
4.4.1 Pseudorange correction.....	25
4.4.2 Smoothing filter	26
4.4.3 Position solution calculation	29
5 Avionics Hardware	31

5.1	Hardware interfaces.....	31
5.1.1	Serial port.....	31
5.1.2	UART	33
5.1.3	USB.....	33
5.1.4	CAN bus.....	33
5.2	BeagleBone based system	34
5.2.1	BeagleBone design	34
5.2.2	BeagleBone evaluation.....	36
5.3	Raspberry Pi based system.....	36
5.3.1	Raspberry Pi design	37
5.3.2	Raspberry Pi evaluation	38
6	UAV Implementation	40
6.1	Avionics bay	40
6.2	Payload bay.....	42
6.3	Antenna Placement.....	42
6.3.1	Current antennae.....	43
6.3.2	GBAS VHF antenna mounting.....	43
6.4	Interface verification.....	44
7	Results and Discussion	46
7.1	GBAS demodulation and decoding software	46
7.2	Correcting raw GPS data	46
7.3	Reception of GBAS using off-the-shelf components.....	47
7.4	Installation of GBAS module in existing UAV systems.....	47
7.5	Selection of hardware	48
7.6	“DumpGBAS”	48
7.7	Potential implications of GBAS navigation in UAVs	49
8	Conclusion	50
	References	51
	Appendices	54
	Appendix A: GBAS message content by type number	55
	Appendix B: DumpVDL2 source code modifications	58
	Appendix C: Python modules for GBAS decoding and use.....	67
	Appendix D: Schematics and design files for BeagleBone RS232 cape	73

List of Figures

Figure 2.1: Modular system overview of a GBAS equipped UAV near a GBAS ground station [1]	3
Figure 2.2: Illustration of the GPS satellite orbits [12].....	4
Figure 2.3: ASCs Cruiser 2 UAV in flight (photo by ASC).....	6
Figure 2.4: Steps of a UAV landing operation at ASC and the decision window where the autopilot will have to take the final landing decision.....	7
Figure 3.1: Illustration showing the vertical and lateral alarm limits, and an aircraft that is exceeding the vertical protection level as indicated by the white rectangle (from presentation by Indra Navia)	9
Figure 3.2: 3D view of the GBAS service volume for an airfield supporting auto-land and guided take-off [2]. The grey rectangle represents the runway	10
Figure 3.3: Oslo airport's (OSL) GBAS broadcast at 113.050MHz, showing the 2Hz frame frequency. The signal is observed from the author's kitchen, approximately 23km from the OSL runway. The broadcast is captured using an RTL-SDR with a dipole antenna and visualized in SDRsharp	10
Figure 3.4: Visualization of how GBAS frames, slots and data bursts relate to each other. The three main parts of the data burst are also shown [2].....	11
Figure 3.5: D8PSK phase shift mapping on a unit circle relative to (1,0).....	12
Figure 3.6: The pseudo-noise scrambler/descrambler shift register, showing its initial state. Descrambling can be done the same way due to the symmetry of the XOR function [2].....	13
Figure 3.7: UAV avionics data flow for GBAS corrections during normal operation (modified from [1])	14
Figure 3.8: Detailed block module of the internal and external interfaces of the GBAS module (modified from [1])	15
Figure 4.1: GPS response on the command "log loglist once". Every line is a log, on the form [port] [name] [trigger] [period] [offset]. "Nohold" means the log will be removed by an "unlogall" command.....	18
Figure 4.2: Possible failsafe mechanism for the GPS to autopilot interface	19
Figure 4.3: Block diagram of a D8PSK receiver based on an SDR (modified from [4]) ...	20
Figure 4.4: Down conversion and analog-digital conversion of an RF signal into discrete in-phase and quadrature values (modified from [1])	20
Figure 4.5: VHF Data Link Mode 2 data burst [17]	21
Figure 4.6: The corrections that are applied to the pseudoranges, visualized as flowchart (modified from [4])	26
Figure 4.7: Difference between the received pseudorange and the 100 second smoothed pseudorange for a single satellite. Y-axis in meters, X-axis in samples at 4Hz	28
Figure 4.8: Log of pseudorange and filtered pseudorange, showing a 1ms clock jump ...	28
Figure 4.9: Error between the received pseudorange and the 30 and 100 second filtered pseudorange after a 1ms clock jump. Y-axis in meters, X-axis in samples at 4Hz	29
Figure 5.1: The RTL-SDR V.3, an inexpensive, easily available and much used SDR that is well suited for GBAS reception (photo rtl-sdr.com)	31
Figure 5.2: Oscilloscope trace of a binary message sent from the GPS receiver over the serial port. The signal switches between $\pm 5V$, and the live decoding of the data is shown.	32
Figure 5.3: The BeagleBone Black rev. C single board computer, with the custom circuit board for serial ports and CAN bus attached.	34

Figure 5.4: Top down view of the serial and CAN circuit board	35
Figure 5.5: The Raspberry Pi 3 Model B+ single board computer, with the serial port modules as well as the SDR.	36
Figure 5.6: Top-down view of the assembled RPi system	38
Figure 5.7: The RPi based system powered up and communicating with the GPS and autopilot in the UAV.....	39
Figure 6.1: Illustration of the different compartments in the Cruiser 2 UAV.....	40
Figure 6.2: View of the Cruiser 2 avionics bay, as seen from the front with the fuel tank removed, clearly showing the GPS receiver (A), autopilot (B), battery compartment (C) and vibration dampeners (D)	41
Figure 6.3: The Cruiser 2 payload bay. Raspberry Pi GBAS module on the workbench demonstrates the amount of space available.....	42
Figure 6.4: The GPS antenna (A) and Iridium antenna (B) that are mounted on the center part of the main wing.....	43
Figure 6.5: VHF antenna mounted on top of the main wing.....	44
Figure 6.6: VHF antenna mounted on top of the tail wing	44
Figure 6.7: Avionics compartment with center-part of wing placed outside in the parking lot at ASC, collecting GPS data.	45

List of Tables

Table 2.1: RPAS Operator categories summarized	5
Table 3.1: The GBAS data burst fields. The double line marks the start of scrambled data	12
Table 3.2: The main GBAS message types. Additional special messages have not been considered here.....	14
Table 3.3: Data fields in a GBAS message	14
Table 4.1: Comparison of the GBAS and VDLM2 data burst header fields.....	21
Table 4.2: VDLM2 data length and the resulting number of FEC bytes appended	22
Table 5.1: Pins of interest for serial communication on the GPS male D-sub 9 connector	33
Table 5.2: Voltage levels for RS232 and UART communication	33

List of Abbreviations

ADC	Analog-Digital Converter
AP	Autopilot
ASC	Andøya Space Center
BBB	BeagleBone Black
BLOS	Beyond Line of Sight
BRLOS	Beyond Radio Line of Sight
BVLOS	Beyond Visual Line of Sight
CAN	Control Area Network
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
D8PSK	Differential 8-Phase Shift Keying
DGPS	Differential GPS
DPSK	Differential Phase Shift Keying
eMMC	Embedded MultiMediaCard
ESD	Electro-static Discharge
EVLOS	Extended Visual Line of Sight
FAS	Final Approach Segment
FEC	Forward Error Correction
FIFO	First In, First Out
GBAS	Ground Based Augmentation System
GNSS	Global Navigation Satellite System
GPIO	General Purpose Input / Output
GPS	Global Positioning System
IA5	International Alphabet nr.5
ILS	Instrument Landing System
IPC	Inter Process Communication
LAAS	Local Area Augmentation System
LAL	Lateral Alarm Limit
LPL	Lateral Protection Level
LSB	Least Significant Bit
MSB	Most Significant Bit
MT#	Message Type
MTOM	Max Take Off Mass
NTNU	The Norwegian University of Science and Technology
PRC	Pseudo Range Correction
PRN	Pseudo Random Number
PSK	Phase Shift Keying
PVT	Position, Velocity, Time
RF	Radio Frequency
RO	RPAS Operator
RPAS	Remotely Piloted Aircraft Systems
RPi	Raspberry Pi
RRC	Range Rate Correction
RS	Reed-Solomon
RS-FEC	Reed-Solomon Forward Error Correction
RTK	Real-time Kinematic
RX	Receiver

SBC	Single Board Computer
SCAT	Special Category
SD	Secure Digital
SDR	Software Defined Radio
SSH	Secure Shell
SSID	Service Set Identifier
TDMA	Time Divided Multiple Access
TX	Transmitter
UART	Universal Asynchronous Receiver-Transmitter
UAS	Unmanned Aerial Systems
UAV	Unmanned Aerial Vehicle
UHF	Ultra High Frequency
USB	Universal Serial Bus
VAL	Vertical Alarm Limit
VDL	VHF Data Link
VDLM2	VHF Data Link Mode 2
VHF	Very High Frequency
VLOS	Visual Line of Sight
VPL	Vertical Protection Level

1 Introduction

This thesis focuses on the possibilities of autonomous landing of commercial Unmanned Aerial Vehicles (UAVs) by integrating the existing Ground Based Augmentation System (GBAS) with UAV avionics.

This was the topic of a preliminary project assignment [1] that studied potential ways of integrating GBAS and avionics signals and the necessary system design.

This thesis has further developed the concept and attempted to design the necessary hardware and software for a prototype application.

1.1 Motivation

Multiple companies have and are currently developing GBAS aircraft hardware (avionics). At present, these systems are only intended for use in manned aircraft and require a pilot to operate. The goal here is therefore to develop a prototype of a GBAS capable navigation platform that would be suitable for use on board a UAV. This introduces multiple technical challenges, as a UAV imposes significant constraints on both cost, weight and size of such a platform. Furthermore, the safety requirements for UAV operations are entirely different from those for manned aircraft. While the safety is a natural aspect to be considered, the focus here will be on the technical challenges in such a prototype.

Andøya Space Centre

Andøya Space Centre (ASC) is focusing heavily on the development and operation of drones and unmanned aircraft for a variety of applications. They have already run operations for customers like the Andøya Test Centre and power delivery companies. One of the bases for their UAVs is Andøya Air Station (Andøya Flystasjon), and ASC is looking to expand its activities there to allow an even wider selection of UAV test and mission services.

As Andøya Air Station is in close proximity to inhabited areas and does suffer from weather related challenges, one of which is the aurora that can interfere with Global Navigation Satellite Systems (GNSS). It would be an advantage to improve the positioning accuracy and provide integrity support during landing operations. ASC would benefit from both safety and greater availability of their services independent of weather conditions. This is especially critical for operations involving larger UAVs.

GBAS would also provide other features of interest, such as allowing remote landing of UAVs on other GBAS-enabled airfields. This could for instance allow long range flights such as from Andøya to Svalbard.

Indra Navia

Indra Navia is a global supplier of communication, navigation and tower solutions for the aviation industry. Landing systems form one of their key areas of expertise, both through the well-established Instrument Landing System (ILS) and the more recently developed GPS based Special Category-I (SCAT-I).

The last few years Indra Navia has been developing the ground infrastructure for the Ground Based Augmentation System (GBAS), which will secure the safe landing of aircraft even in no-sight conditions. The development and testing of this system is now completed and after validation, it will become Indra Navias latest product for civil airport operations.

Indra Navia is looking for ways to expand the use of the GBAS product to UAVs as well, which represents a new market for the system.

1.2 Objectives

The main objective of this thesis is to develop a system to integrate GBAS and avionics signals for autonomous landing of UAVs. This has resulted in the following subobjectives:

- Study the system requirements to develop the integration concept
- Inspect the technical details of an existing UAV system at ASC
- Develop the necessary software for GBAS signal reception and decoding based on an existing demodulator of a similar signal
- Design a prototype of the required avionics hardware using easy to acquire off-the-shelf components

2 Background

This section provides the main background that is relevant for the discussion and design of GBAS avionics hardware for UAVs. Figure 2.1 gives an overview of the GBAS system, the interfaces between the UAV and the surrounding world, as well as the GBAS module as part of the UAV avionics.

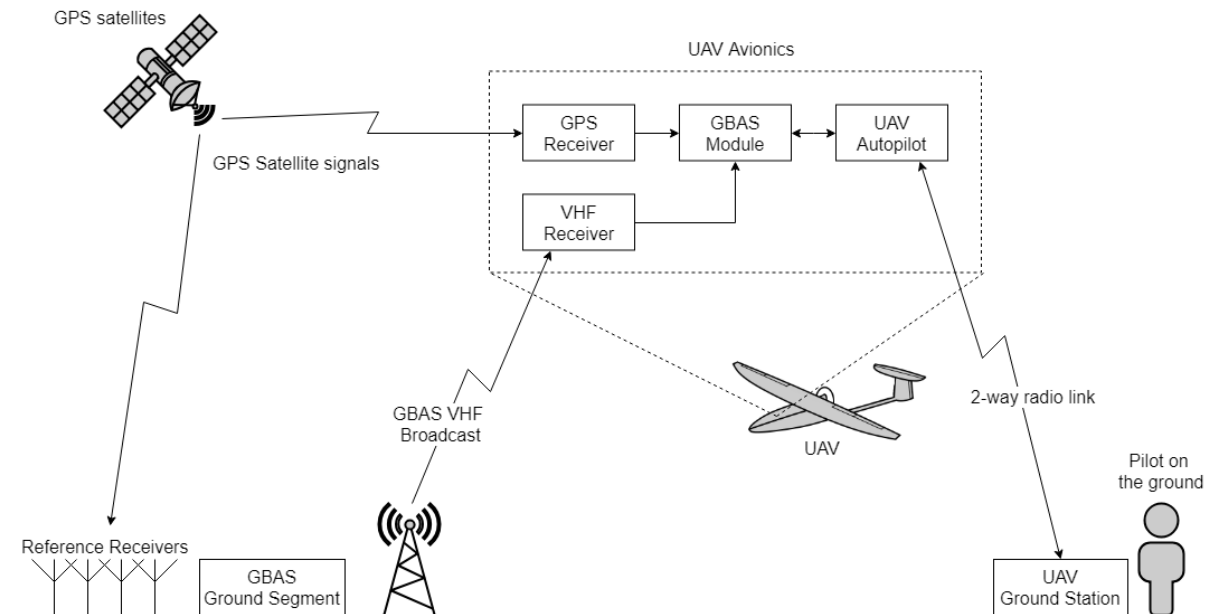


Figure 2.1: Modular system overview of a GBAS equipped UAV near a GBAS ground station [1]

The GBAS ground station broadcasts information over VHF to the receiver in the UAV, which samples and demodulates the GBAS VHF broadcast. The contents of broadcast is used together with pseudorange information from the GPS receiver in order to improve the calculated position solution used by the UAV autopilot for navigation. In addition, integrity information is obtained. These data are transferred over the two-way radio link to the pilot on the ground for his consideration.

An introduction to the concepts of GPS positioning used in this system is provided in chapter 2.1 and 2.2. The requirements for UAV operations in general and ASCs current UAV landing procedures are discussed in chapter 2.3 and 2.4. Limitations of this system are discussed in 2.5.

2.1 GPS

The Global Positioning System (GPS) is one of an increasing number of available Global Navigation Satellite Systems (GNSS). The constellation consists of 24 satellites, with an additional three spares for full global coverage (Figure 2.2).

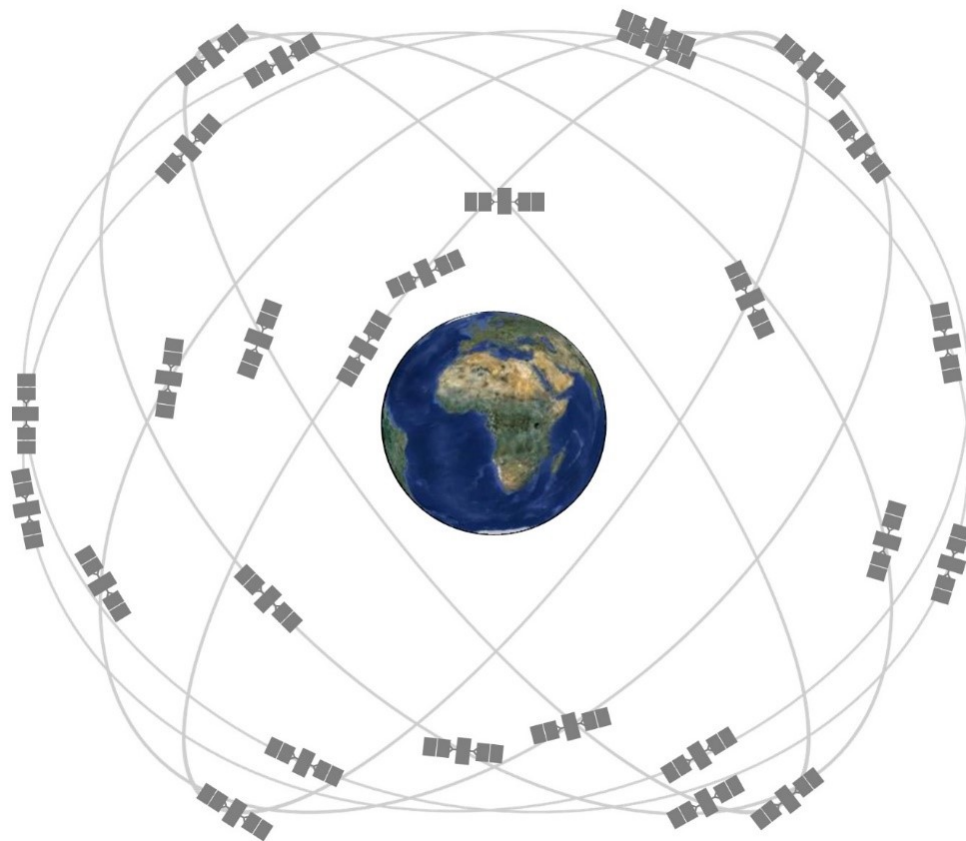


Figure 2.2: Illustration of the GPS satellite orbits [12]

A satellite's position can be described by its ephemeris, a number of orbital elements that allow the satellite's position to be accurately computed at any time. Each satellite broadcasts their ephemeris and current time. By measuring the transmission time of the signal, a receiver can calculate the distance (pseudorange) to each satellite in view.

The satellites have high accuracy atomic clocks on board for timekeeping, but the GPS receiver does not. The receiver's clock will drift, influencing all the measured pseudorange. This is solved by introducing the clock error as a fourth unknown to the positioning problem. Using the ephemeris and pseudorange measurements of four or more satellites, an accurate position and time can be determined by trilateration.

The functionality of the GPS satellites, their signals and the receiver algorithms for tracking and ranging are well documented in both standards [13] and literature [14].

Several error sources interfere with the pseudorange measurement, such that the observed pseudorange p can be more accurately modelled as:

$$p = \rho + d_\rho + c(dt - dT) + d_{ion} + d_{trop} + \epsilon_{mp} + \epsilon_p \quad (2.1)$$

That is, the measurement p is a function of the true range ρ , with added effects from:

- Satellite orbital errors (d_ρ)
- The difference in clock offset for satellite (dt) and receiver (dT) multiplied by the speed of light constant (c)
- Ionospheric delay (d_{ion}) and tropospheric delay (d_{trop})
- Multipath delay (ϵ_{mp})
- Receiver noise (ϵ_p)

The most significant error source is atmospheric disturbance, but orbital inaccuracy and clock deviations in the satellite and the receiver can be significant contributing factors as well [14]. Since the calculation of the pseudorange (p) is based on the speed of light (c), even small inaccuracies in the measured time (Δt) will result in large errors according to:

$$\Delta p = c * \Delta t \tag{2.2}$$

2.2 Differential Processing

Many of the GPS error sources are slowly varying and spatially correlated over short baselines [15]. The assumption is made that two receivers in the same area will experience approximately the same errors. Using differential processing, these errors can be estimated and corrected.

Differential processing typically takes the form of a stationary reference station, “the base”, which has an accurately known position. Information from measurements made at the base are sent to “the rover”, a moving GPS receiver with uncertain position. The rover combines this information with its own measurements so that the errors common to both base and rover cancel out.

Both phase-based Real-Time Kinematic GPS, currently in use at ASC (chapter 2.4) and pseudorange-based GBAS (chapter 3) are examples of such differential GPS processing.

2.3 UAV

Unmanned Aerial Vehicles (UAV) are increasingly becoming a part of airspace activities, both at a hobby level as well as on a professional scale. Data collection by UAV is more common than ever before. Applications range from simple photography to research, surveillance and monitoring both for civil and defence purposes.

The increase in UAV use has required the introduction of regulation to prevent conflicts and accidents in the airspace. In Norway, UAV operations are sorted into three categories, depending on the potential for damage in case of failure. The categories and associated rules for remotely piloted aircraft systems (RPAS) as well as required permissions are described in [16]. Table 2.1 summarises the basic metrics that can be used to determine RPAS operator (RO) category for an airframe.

Table 2.1: RPAS Operator categories summarized

Category	Max take-off mass	Max velocity	Max altitude
RO1	2,5 kg	60 knots	120 m
RO2	25 kg	80 knots	120 m
RO3	> 25kg	>80 knots or turbine engine	>120 m

In addition, the contact between operator and UAV can be classified based on the degree of visual communication:

- VLOS (Visual Line of Sight): Pilot can observe the UAV with the naked eye.
- EVLOS (Extended Visual Line of Sight): Pilot is in touch with external observers that have VLOS to the UAV.
- BLOS (Beyond Line of Sight): Neither pilot nor observers have VLOS on the UAV.

- BRLOS (Beyond Radio Line of Sight): No direct radio link between pilot and UAV (Satellite communication, cellular network, radio relays, etc.). Commonly considered a subcategory of BLOS.

If there is no direct radio link between pilot and UAV, the operation is considered BRLOS by definition, even if the UAV is physically located in VLOS. As the later categories introduce additional sources of potential error, each category adds additional requirements and regulations that have to be adhered to for legal UAV operation [16].

ASCs Cruiser 2 UAV

ASC has a large selection of UAVs, both multi rotor and fixed wing. The work presented in this thesis uses ASCs Cruiser 2 UAV as a representation of the typical UAV for which GBAS support could be beneficial. The software and hardware are developed with this UAV in mind. Figure 2.3 shows a picture of the UAV in flight. With a wingspan of 5.2 meters and a minimum take-off weight of 53 kg [7], this sizeable fixed wing aircraft will always require RO3 permissions to fly.

This aircraft is controlled with Cloud Cap Technology's Piccolo 2 autopilot, which is connected to an external NovAtel OEMV2 GPS receiver. These two avionics modules form the basis for the GBAS avionics design in this thesis.



Figure 2.3: ASCs Cruiser 2 UAV in flight (photo by ASC)

2.4 Landing procedures at ASC

ASC can land their UAVs both manually and autonomously. To limit the potential of pilot error, autonomous landing is preferred, using a landing system based around Real Time Kinematic GPS. This is presently the most widely used UAV landing solution for commercial UAVs of this size. RTK GPS uses the difference in measured GPS carrier phase at the UAV and a fixed reference station at a known position to determine the

difference in distance to satellites, giving a very accurate relative position. The system in use at ASC typically achieves an accuracy of $\pm 5\text{cm}$ [5].

The RTK reference station is in ASCs case contained in the UAV ground station, and the RTK observations are transmitted over the same data link as general pilot commands. Because the ground station can be moved around between operations, it must first determine its own location before it can act as a precise reference for an RTK UAV.

This can primarily be done in two ways. The base stations GPS antenna can be placed on a precisely surveyed location, such that the coordinates can be supplied by the pilot to the ground station. Alternatively, having the antenna in a fixed position, the location can be found by averaging the GPS solution found over a period of time. In general, 15 to 30 minutes of GPS data is deemed to provide a satisfactory position fix by ASC, but as much data as time allows is collected, since it will only improve the fix. A benefit of the second method is that it works anywhere, and the averaging can be done while other preparative tasks for UAV launch are being carried out. The drawback is that the UAV navigation solution can only ever be as accurate as the base station coordinates.

A landing operation with a fixed wing UAV of this size is very similar to the landing of a conventional manned aircraft. The autopilot uses an entirely model based approach to control, taking into consideration lift coefficients, weight and remaining fuel when steering the aircraft. The landing steps described below are visualised in Figure 2.4.

- While approaching to the landing location, the autopilot will reduce speed and extend the flaps, following an approach path defined by its mission waypoints.
- 8 seconds before estimated touchdown, the AP has to decide if it will commit to the landing attempt and bring the aircraft down. In order to determine if the conditions are acceptable, the aircraft has to hit a virtual 2x2 meter window in space. The AP will not attempt to abort the landing beyond this point. Failing to hit the window causes the AP to abort and go around for a new try.
- 4 seconds before estimated touchdown the motor is stopped. This is the point of no return; the pilot can no longer trigger an abort manually.
- The aircraft will flare to lose even more speed, before finally touching down on its main landing gear.

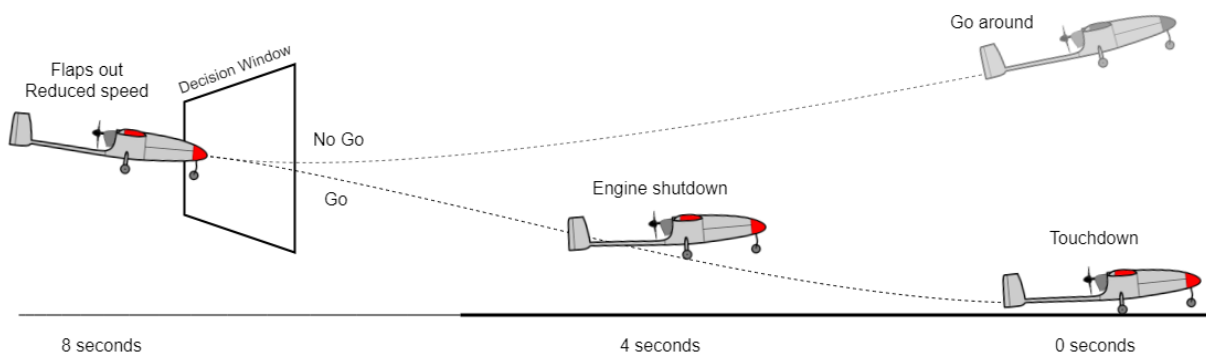


Figure 2.4: Steps of a UAV landing operation at ASC and the decision window where the autopilot will have to take the final landing decision.

2.5 Limitations of the existing system

The present RTK solution is primarily designed to facilitate operations near the reference base station. For most practical use, this means landing in the same general area where the UAV took off.

The accuracy generally only holds for a range of approximately 20km from the reference station, where after the accuracy drops. This limits the use of the system in its present configuration.

UAVs that depart from one base station will not easily be able to use the RTK information of a different station to land there. On the other hand, GBAS will allow for a standardised communication and correction method independent of airfield and UAV type.

GBAS will in addition to increased accuracy in position estimates, also supply detailed error and integrity information. This allows the calculation of protection levels required for safe landing in civil aviation. Operation of large UAVs will benefit in a similar way from the reliability and integrity of the GBAS system.

3 GBAS

The Ground Based Augmentation System is a GNSS augmentation service for aircraft typically employed on and surrounding airfields. Using the principles of differential GPS, it provides aircraft in the vicinity with information on the local GNSS conditions. This results in high precision positioning information during approach and landing operations. What sets the GBAS system apart from other systems that give similar or even better positional accuracy is the way it deals with uncertainty and integrity. Due to the reduced uncertainty in position and high system integrity using GBAS, the aircraft can continue its flight trajectory even under very low visibility conditions. This makes the system valuable for safe landing of aircraft under bad weather conditions.

The system consists of two parts, the ground station and the aircraft receiver, as shown in Figure 2.1. The ground station is a stationary structure at the airport. Using multiple GNSS receivers, the station tracks the satellite ranging sources in view. As the receiver antennas positions and each satellite's ephemeris are known, a comparison can be made between the measured pseudo-range and the calculated geometric range. The difference between the two form the basis for the so-called "pseudorange correction" for each satellite. The exact steps for the pseudorange correction calculation can be found in the GBAS EUROCAE standard [2].

This data, along with system integrity information as well as general information about satellites health and the airport configuration are broadcast over VHF for any GBAS equipped aircraft in the area to receive. This makes GBAS a cost effective solution, as a single ground station can service an entire airport and all aircraft in the vicinity.

The aircraft avionics uses the GBAS ground stations estimate of the current errors as well as integrity parameters based on its own satellite signals to calculate the Vertical and Lateral Protection Levels (VPL/LPL). If the calculated protection levels exceed the runway alarm limits, as shown in Figure 3.1, the approach and landing conditions are not considered safe.

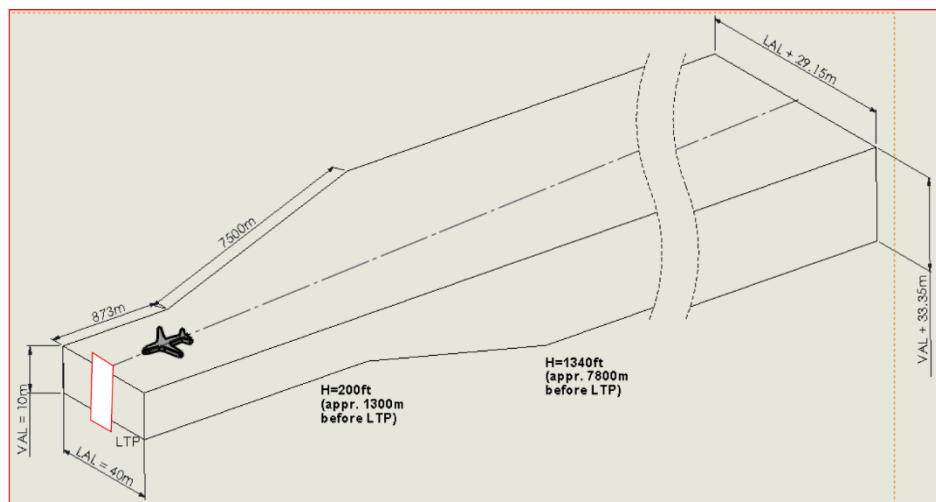


Figure 3.1: Illustration showing the vertical and lateral alarm limits, and an aircraft that is exceeding the vertical protection level as indicated by the white rectangle (from presentation by Indra Navia)

3.1 Signal Broadcast

The GBAS data is transmitted as a differential 8-phase shift key modulated broadcast in the aeronautical navigation band from 108.025MHz to 117.975MHz, using channels spaced by 0.025MHz. The broadcast is such that aircraft up to 43 kilometre (23 nautical miles) away are able to receive it. This is called the GBAS’s service volume, shown in Figure 3.2, and it covers the entire approach and landing such that aircraft can receive GBAS corrections all the way down to the runway.

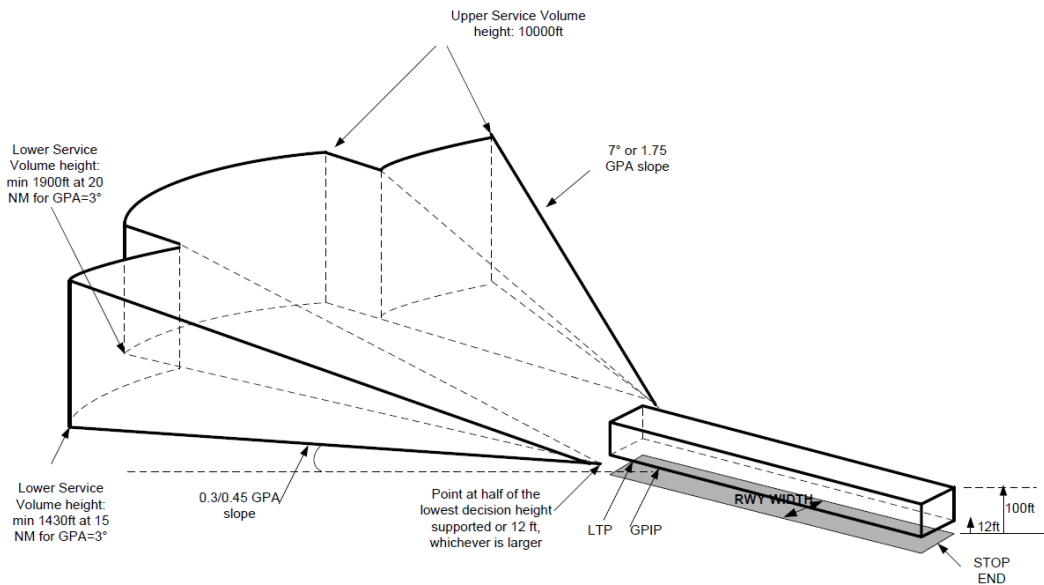


Figure 3.2: 3D view of the GBAS service volume for an airfield supporting auto-land and guided take-off [2]. The grey rectangle represents the runway

Any aircraft in the service volume, having tuned their GBAS receiver to the appropriate frequency for the airport they are approaching, can start to receive the broadcast. Figure 3.3 shows the signal from Oslo airport (OSL) on the air. The signal is weak as it is received from the ground without line of sight to the airport. The activity observed occurs on half-second intervals. Each mark a GBAS frame.

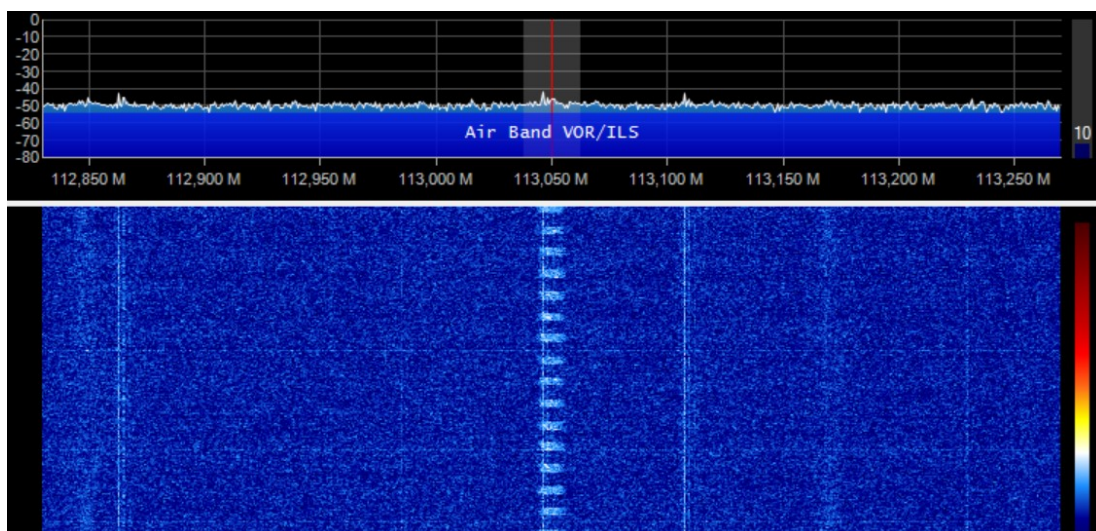


Figure 3.3: Oslo airport’s (OSL) GBAS broadcast at 113.050MHz, showing the 2Hz frame frequency. The signal is observed from the author’s kitchen, approximately 23km from the OSL runway. The broadcast is captured using an RTL-SDR with a dipole antenna and visualized in SDRsharp

GBAS frames occur with a frequency of 2 Hz, synchronised with the GPS second. Every frame is divided into 8 slots, and a GBAS data burst (covered in section 3.1.2) can occur in each of them. This structure is shown in Figure 3.4. Having slots allows multiple GBAS stations to coexist on the same frequency using Time Divided Multiple Access (TDMA) techniques if required.

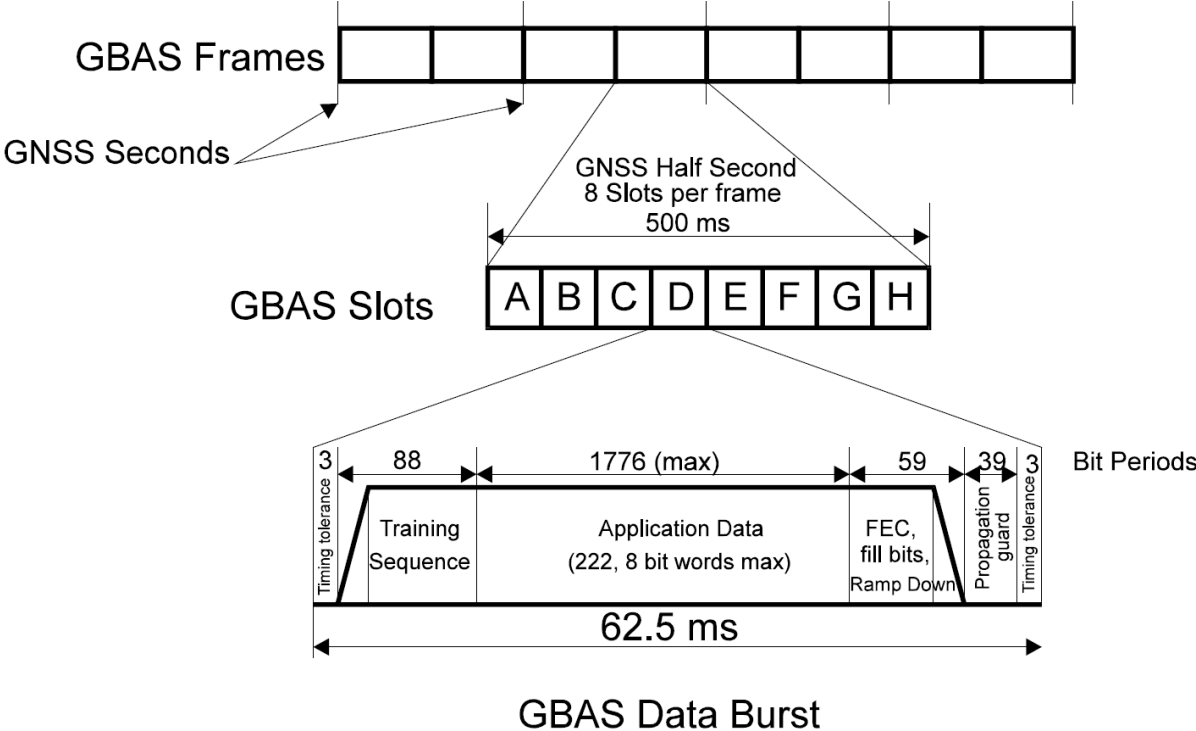


Figure 3.4: Visualization of how GBAS frames, slots and data bursts relate to each other. The three main parts of the data burst are also shown [2]

3.1.1 D8PSK modulation

Phase shift keying (PSK) is a digital modulation method where the data is encoded as the phase of a carrier frequency. PSK demodulation is complicated, as a reference of the carrier is required in the demodulator in order to compare the phase. In differential PSK (DPSK), the data is encoded as a change in phase rather than absolute phase. The change in phase can easily be determined by using the preceding sample as reference.

The number of symbols determines the size of the available 2π phase shift that each can be allocated. More symbols increase the data throughput, but the tighter spacing causes demodulation errors to be more prevalent. D8PSK can be thought of as having 8 distinct states, where the symbols are represented as phase shifts between the states. Figure 3.5 shows the relative phase shifts on a unit circle.

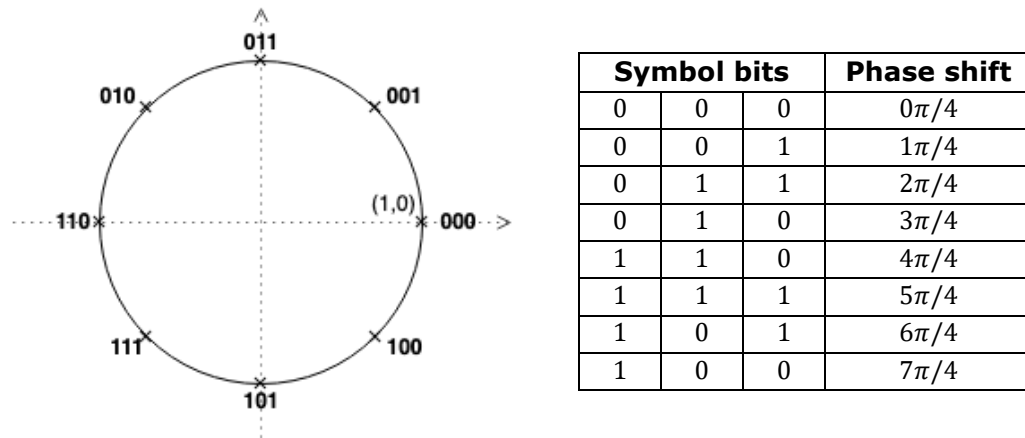


Figure 3.5: D8PSK phase shift mapping on a unit circle relative to (1,0)

The symbols are assigned to phase shifts according to grey code, where they are placed such that adjacent values are only 1 bit different. This property is cyclic, so that it also holds when rolling over from the largest to the smallest value. The result is that getting a symbol wrong by one phase step only causes a single bit error. Provided there are not too many, these can be easily corrected using forward error correction (FEC) algorithms.

3.1.2 The data burst

In each GBAS slot, a data burst can occur. Such a data burst follows a well-defined structure, of which the data fields are shown in Table 3.1.

Table 3.1: The GBAS data burst fields. The double line marks the start of scrambled data

Field	Contains	Bits
Power stabilization	-	15
Synchronisation & Ambiguity resolution	A fixed sequence of bits: 010 001 111 101 111 110 001 100 011 101 100 000 011 110 010 000	48
Station Slot Identifier	The GBAS station's first assigned slot as a value 0-7	3
Transmission Length	Number of bits in application data and FEC	17
Trainings Sequence FEC	Parity bits computer over SSID and length field	5
Application Data	One or more messages, see 3.1.3	Up to 1776
Application FEC	Reed-Solomon FEC bytes	48
Fill bits	Ensures any length message can be sent by 3 bit symbols	0 to 2

Scrambling

After the fixed sequence of bits that is used for synchronisation and ambiguity resolution of the signal, the data is scrambled. This is done to cause variation in the signal in order to avoid transmitting long sequences of the same-bit values.

In practice, this is achieved by an XOR operation between the data and the output from a pseudo-noise generator (shown in Figure 3.6). Due to the nature of the XOR function, scrambling an already scrambled sequence with the same pseudo-noise as was used to

generate it, brings back the original data: $[A \text{ xor } N] \text{ xor } N = A \text{ xor } [N \text{ xor } N] = A \text{ xor } 0 = A$. Therefore, the descrambler on the receiver side of the broadcast can be the same as the scrambler on the transmitter.

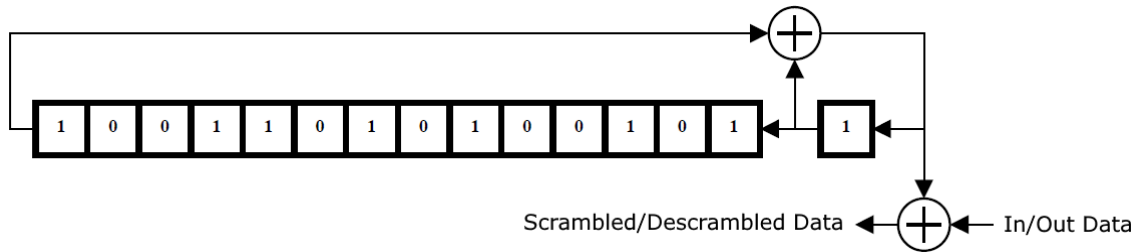


Figure 3.6: The pseudo-noise scrambler/descrambler shift register, showing its initial state. Descrambling can be done the same way due to the symmetry of the XOR function [2]

Training Sequence FEC

The training sequence code can, according to [3], correct any single bit errors and detect 75 double bit errors of 300 possible. The 5-bit parity FEC P is generated by parity matrix multiplication of the SSID and transmission length field with the 20x5 matrix H :

$$P = [SSID_1, SSID_2, SSID_3, TL_1, \dots, TL_{17}] H^T \quad (3.1)$$

$$H = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Application Data FEC

The FEC for the application data is a Reed-Solomon block correcting code with block length of 255 bytes and message length of 249. For messages shorter than 249 bytes, zero padding up to a full message length is required. Those padding bytes are only used for the FEC calculation and not transmitted in the GBAS broadcast. With 6 RS-FEC bytes, 6 byte errors can be detected, or 6/2 can be corrected.

3.1.3 The GBAS message

The application data of a GBAS burst will contain one or more messages, up to the max length allowed of 222 bytes (1776 bits). Table 3.2 sums up the main message types and what they pertain. The exact contents of the messages can be found in their respective tables in appendix A.

Table 3.2: The main GBAS message types. Additional special messages have not been considered here.

Message Type (MT)	Contents of message
1	Pseudorange corrections for 100 second smoothed data
2	Information on the GBAS ground station
3	Message of fill bits
4	Data for the landing paths supported by the station
11	Pseudorange corrections for 30 second smoothed data

A message is built up as per Table 3.3, where the message type (MT) is one of those indicated in Table 3.2 above. Received messages starting with hexadecimal 0xFF should not be used, as they indicate the GBAS station is not in operative mode. The GBAS ID field consists of four 6-bit characters, either capital letters or number. The bit encoding corresponds to the International Alphabet nr.5 (IA5) with bit 7 not used.

Table 3.3: Data fields in a GBAS message

Message Block	Fields	Value	Bits
Message Block Header	Message Block ID	0xAA – operational 0xFF – test	8
	GBAS ID	4 character name of the GBAS station	24
	Message Type	Number as by Table 3.2	8
	Message Length	Length of entire message in bytes	8
Message	<i>Depends on message type</i>	<i>Depends on message type</i>	Up to 1696
Message Block CRC	Message Block CRC		32

3.2 UAV application

The GBAS equipped UAV system was introduced in Figure 2.1. For pseudorange corrections, the data flow between the modules is shown in Figure 3.7. Combining GPS data and the GBAS corrections in the GBAS module allows full, unlimited control of the algorithm implementations. It also evades the potential dangers that could result from, for instance, interrupting the regular tasks of the autopilot with GBAS calculations.

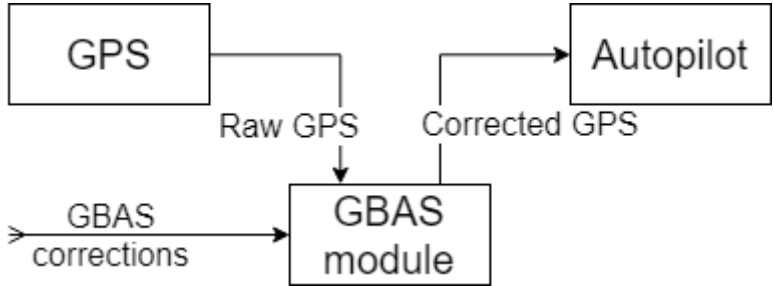


Figure 3.7: UAV avionics data flow for GBAS corrections during normal operation (modified from [1])

The messages that do not concern the positioning directly, (MT4 and parts of MT2) contain information that the UAV cannot easily use without pilot input. The contents of these messages can be forwarded for consideration by the pilot on the ground.

3.3 System design

The functionality of the GBAS module can be further divided into parts with specific functions. Figure 3.8 shows the GBAS module with both external and internal interfaces between the module's parts. The software for the GBAS module has been designed with a similar partitioning in mind.

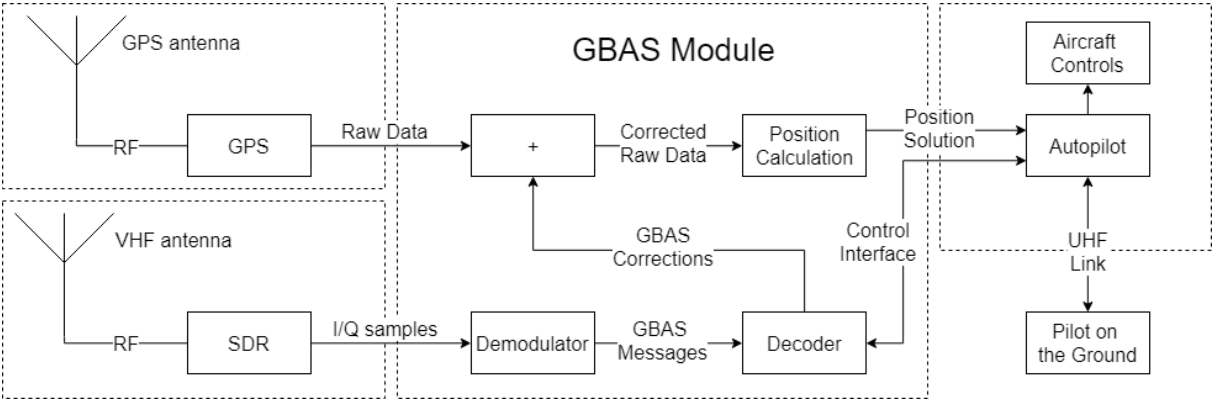


Figure 3.8: Detailed block module of the internal and external interfaces of the GBAS module (modified from [1])

In Figure 3.8, the SDR generates in-phase and quadrature (I/Q) samples. The demodulator uses these to determine phase shifts and in turn symbols, combining them into a complete GBAS data burst. The messages contained in the burst are passed along to the decoder, which will decode them and sort them on type.

MT3 can be simply discarded. MT4 and parts of MT2 are passed along the GBAS control interface into the autopilot for forwarding over the UHF link. MT1 and MT11 contain corrections while MT2 contains general GBAS conditions. These are sent into the "+" module, where they are combined with the raw data from the GPS receiver.

The resulting corrected pseudoranges can then be used to generate a more accurate position solution than would be possible with the raw pseudoranges alone, which the autopilot can use for navigation.

4 Avionics Software

This section details the work done pertaining to the development and verification of software aspects to the GBAS module. The chapter covers a variety of code topics, which have been grouped by area of application from the parts shown in Figure 3.8.

4.1 The AP–GPS interface

The interface between the Piccolo 2 autopilot and the NovAtel OEMV2 GPS receiver is pre-existing, and needs to be understood in order to wedge a custom GBAS module in between the two units.

The autopilot uses the GPS's interface as described in the GPS's documentation [8] in order to gather the data it needs for navigation. The interface functions using the concept of "commands" and "logs", which can be issued and received in three different formats:

- ASCII is an all-round format for any use. Easily understood by humans and machines alike.
- Abbreviated ASCII is a shorter form that strips away unnecessary sync characters and error checking abilities for a simpler user experience.
- Binary is an effective and compact messaging method. Not human-readable.

Commands are issued to the GPS receiver from the autopilot. They are used to configure every aspect of the GPS receiver, from satellite reception to port settings and data rates. The main command that is interesting for the GBAS software is the *log* command. This command can be used to request single logs, or schedule logs to be transmitted at chosen rates.

Logs are data packets generated by the GPS receiver. A large variety of possible logs can be generated, depending on the application requirements. Typically, position and velocity are of interest in GPS applications. Raw information on ephemeris and pseudorange measurements, which are required for the GBAS module, are also supported by the GPS and have their own logs. By default, the logs are in abbreviated ASCII, but ASCII or binary can be selected with a trailing "a" or "b" on the log name.

4.1.1 Error detection

Binary and ASCII messages sent from the Novatel GPS are protected by a 32-bit CRC. This allows the receiver of the messages to discern if bits of the message have been corrupted. A GBAS module will be required to both check incoming messages and create new CRC bits for outgoing messages.

Along with the source code for a CRC32 calculation written in C, the Novatel firmware reference manual [8] specifies the generator polynomial as 0xedb88320 in. In this value, bit n represents whether summand x^n is part of the polynomial. The value is least significant bit (LSB) first, such that 0xe = 1110 translates as $1 + x + x^2$. The summand x^{32} is always implicitly part of the equation, such that the entire polynomial is:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

4.1.2 Autopilot initialisation

In order to determine how the autopilot configures the GPS receiver, a splitter cable was made such that a PC could listen in on the communication. By learning how the GPS was configured, it has been determined what information the GBAS module needs to supply to the autopilot once the module is mounted in between the two units.

With the cable connected during power-up, the following start-up sequence was observed:

- 1) The autopilot issues a full reset to the GPS. This defaults all the ports to bit rate of 9600.
- 2) The autopilot configures the GPS port to a bit rate of 57600 (the highest supported by the autopilot) by sending the "com" command first at 9600b/s and then repeating the same command at 57600b/s. This functions as a simple form of handshaking for the GPS.
- 3) The autopilot clears the GPS's list of requested logs and removes any position fix that was previously made. It then requests a log of GPS receiver version in ASCII format, and waits. It was observed that, should the autopilot not get any reply in 5-10 seconds, the entire procedure would be repeated from step 1
- 4) The GPS's response indicates the type of receiver that is connected, its capabilities, its serial number, hardware and software revision as well as software compilation date.
- 5) The autopilot then finishes the setup by issuing log commands for a number of binary logs, where after the configuration is saved on the GPS.
- 6) The GPS starts generating the requested logs. No further communication from the autopilot is observed.

The final configuration of the receiver can be verified by connecting to the autopilot directly via an unused port (USB cable in this case), and request the log list, which shows all active scheduled logs. The response (Figure 4.1) shows that logs are being generated on 3 ports.

- USB2 is where the PC's terminal is connected and where the *loglist* command was issued.
- COM2 is generating *Align*® corrections. This is Novatel's system for relative positioning or determining heading from multiple receivers. This feature is not in use and the port is not connected to anything in the UAV's current configuration.
- COM1 is connected to the autopilot. These are the logs that are of most interest as the GBAS module will have to generate them.

```

<LOGLIST USB2 0 85.5 UNKNOWN 0 26.382 004c0000 c00c 7009
< 9
< COM1 PSRPOSB ONTIME 0.250000 0.000000 NOHOLD
< COM1 PSRVELB ONTIME 0.250000 0.000000 NOHOLD
< COM2 RTCAOBS3 ONTIME 0.100000 0.000000 NOHOLD
< COM2 RTCAREFEXT ONTIME 0.100000 0.000000 NOHOLD
< COM1 HEADING2B ONNEW 0.000000 0.000000 NOHOLD
< COM1 PSRDOPB ONTIME 3.000000 0.000000 NOHOLD
< COM1 SATVISB ONTIME 20.000000 0.000000 NOHOLD
< COM1 TRACKSTATB ONTIME 5.000000 1.500000 NOHOLD
< USB2 LOGLIST ONCE 0.000000 0.000000 NOHOLD

```

Figure 4.1: GPS response on the command “log loglist once”. Every line is a log, on the form [port] [name] [trigger] [period] [offset]. “Nohold” means the log will be removed by an “unlogall” command.

The logs that are transmitted to the Autopilot over COM1 as shown in Figure 4.1 are:

- GPS Position (PSRPOSB) at 4Hz, contains coordinates and height above sea level
- GPS Velocity (PSRVELB) at 4Hz, contains horizontal and vertical speed as well as track over ground
- Heading (HEADING2B), an *Align*® feature for finding the direction of the line between base and rover relative to north.
- GPS dilution of precision (PSRDOPB) every 3 seconds, a measure of how well geometrically spaced the satellites in the solution are
- Visible satellites (SATVISB) every 20 seconds, gives a quick overview of visible satellites and their apparent elevation and azimuth
- Tracking status (TRACKSTATB) of all GPS receiver channels

For the integration of the GBAS module, some additional logs from the GPS will be required. Some logs are not influenced by GBAS at all, and can simply be passed along to the autopilot. The computed position solution by the GPS is no longer needed.

4.1.3 GBAS-required data

The GBAS corrections have to be applied before a position solution is calculated. This requires the raw pseudorange and phase measurements, as well as ephemeris information for position calculation to be available to the GBAS module.

By setting pseudorange output by the GPS to the same frequency as the autopilot’s required position solution, no timing complexity is added. Every pseudorange message received leads to a position solution output. Should the delay through the module be determined to be significant for the accuracy of the solution provided, a position prediction can be calculated instead, by extrapolating from position and velocity at time of measurement with an estimate of the input-output delay time:

$$P_{predicted} = P + V * t_{delay} \quad (4.1)$$

The required logs for the GBAS calculations are *rangecmp* (at 4Hz) for raw GPS ranges, and *rawephem* (whenever it changes) for ephemeris data. Binary logs are preferred as they are compact and easy to decode as the content follows standard bit patterns for the

variable type. After the autopilot has configured the GPS, these additional logs are added via the commands:

```
log com1 rangecmpb ontime 0.25
log com1 rawephemb onnew
saveconfig
```

A few logs no longer serve a purpose with the GBAS module in place. In particular, since new position and velocity logs are synthesised by the GBAS module, the ones output by the GPS have become obsolete.

These logs could potentially be given a new purpose in the form of a fail-safe. A problem introduced with the system configuration as in Figure 3.7, is that the autopilot no longer has direct access to the GPS. Should the GBAS module fail, the UAV would be left to navigate by dead reckoning, relying only on internal sensor data. This could be solved using the failsafe mechanism shown in Figure 4.2.

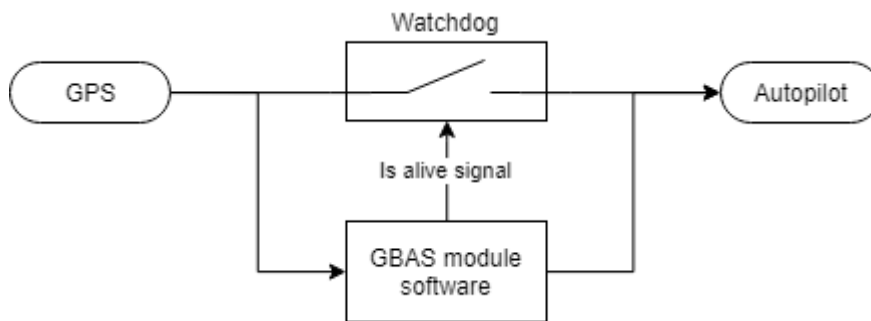


Figure 4.2: Possible failsafe mechanism for the GPS to autopilot interface

With all logs still output by the GPS, a short circuit between the GBAS input and output ports would cause the autopilot to have a position solution available, regardless of the functioning of the GBAS module. The short circuit can be made either in software or physically in hardware in the form of a relay. In addition to triggering by a lack of “OK” signal from the GBAS module, it could conceivably also be triggered via an override signal from the autopilot.

During normal GBAS operation, the GBAS module can simply discard the unneeded logs received. When simulating a watchdog trigger by sending in additional raw logs to the autopilot, no adverse effects were observed, and the autopilot did not attempt to unlog the raw data.

4.2 Demodulating the GBAS broadcast

This section details the work done on making a receiver and demodulator for a GBAS broadcast based on software-defined radio (SDR). The GBAS signal was discussed in chapter 3.1, and a block diagram of the demodulation steps are shown in Figure 4.3, highlighting the tasks typically performed in the SDR hardware. The required hardware will be discussed in chapter 5.

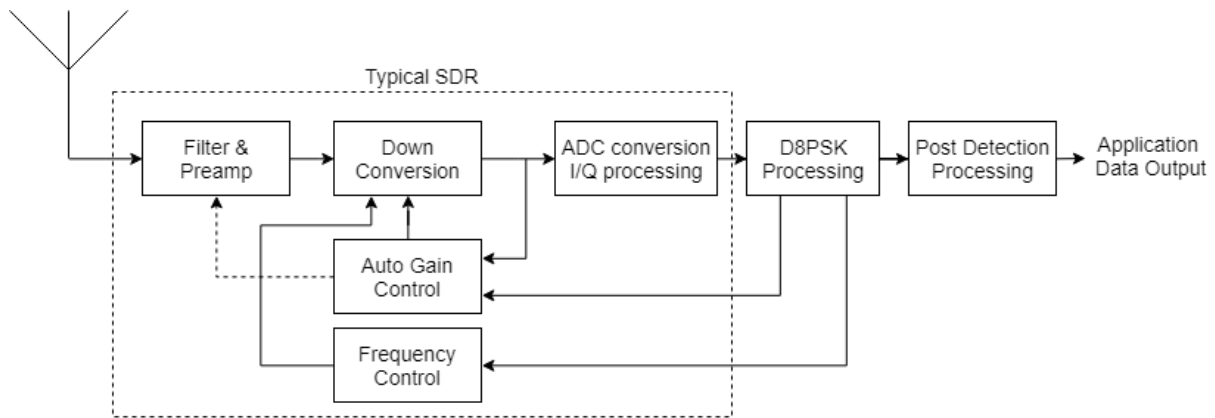


Figure 4.3: Block diagram of a D8PSK receiver based on an SDR (modified from [4])

An SDR supplies discrete samples of the signals in-phase (I) and quadrature (Q) components. Software must then further process these values into symbols and associated bit values. Figure 4.4 shows how a local oscillator is used to remove the carrier frequency from the received signal (Down conversion) and the resulting values sampled (ADC conversion).

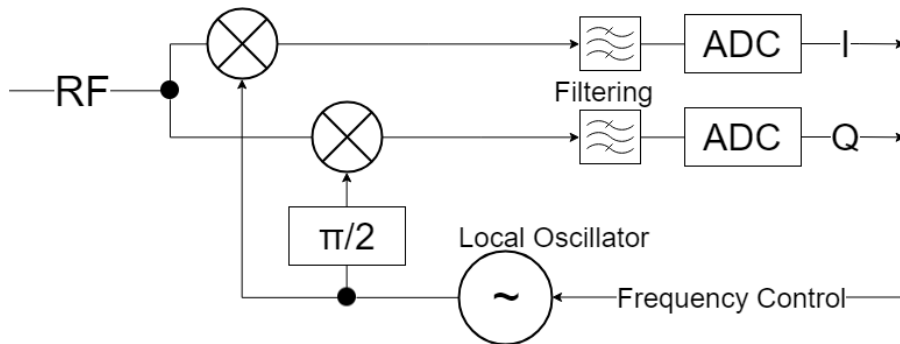


Figure 4.4: Down conversion and analog-digital conversion of an RF signal into discrete in-phase and quadrature values (modified from [1])

4.2.1 VHF Data Link Mode 2

VHF Data Link Mode 2 (VDLM2) is a way of communication between aircraft and ground stations. It consists of three layers, from the high-level application layer, through the link layer to the low-level physical layer.

Mode 2 is of interest to GBAS demodulation because a packet at the physical layer closely resembles the structure of a GBAS data burst. As can be seen when comparing Figure 4.5 and Figure 3.4, the main notable difference is that VDLM2 allows for longer messages by dividing them into data blocks and interleaving FECs.

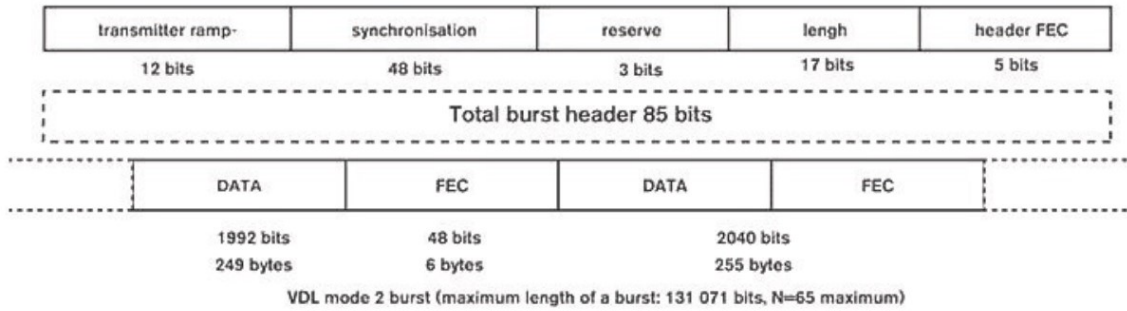


Figure 4.5: VHF Data Link Mode 2 data burst [17]

Mode 2 is also D8PSK modulated at 10500 symbols a second with a channel spacing of 25 kHz. Since it is transmitted in the communication band (117.975–137 MHz) with vertical polarization (versus horizontal in the navigation band for GBAS) this does not offer a problem for the GBAS reception.

Furthermore, VDLM2 has:

- Identical phase to character mapping as described in chapter 3.1.1
- Very similar burst headers as seen in Table 4.1
 - Identical sync sequence
 - Identical trainings sequence FEC algorithm and matrix
- Identical initial state in the bit scrambling algorithm
- Identical data block size
- Each data block is followed by a RS-FEC with the same polynomial.

That leaves only a few differences that must be accounted for.

Table 4.1: Comparison of the GBAS and VDLM2 data burst header fields

Data burst header fields			
GBAS	Bits	VDLM2	Bits
Power stabilization	15	Power ramp up	12
Synchronisation and ambiguity resolution	48	Synchronisation and ambiguity resolution	48
Station Slot Identifier (SSID)	3	Reserved symbol	3
Transmission length	17	Transmission length	17
Training sequence FEC	5	Header FEC	5

Reserved bits

The GBAS broadcast header contains the SSID field where Mode 2 has a reserved symbol. Since this symbol is used in the training sequence FEC calculations, it should not be removed or changed by the receiver.

Transmission length

This field can appear deceptive. While the field description is the same for GBAS and VDLM2, their respective definitions of what constitutes part of the “transmission” are not identical.

In a GBAS burst, this number represents the total number of bits of application data and FEC (up to 1776 bits and always 48 bits, respectively). A VDLM2 burst can be much longer, up to 131 071 bits. This is transmitted in blocks of up to 1992 bits, where each block is followed by the FEC for that block. Contrary to GBAS, the bits used for FECs are not counted in the transmission length field.

For a GBAS and VDLM2 burst of the same length, it thus follows that the GBAS transmission length field will indicate 48 bits more than that of the VDLM2 transmission. If unaccounted for, a VDLM2 decoder that receives a GBAS message will then read too many bytes and include the actual FEC bytes with the perceived data block.

Reed-Solomon FEC

In a VDLM2 burst, while six FEC bytes are always generated, the number of appended FEC bytes is dependent on the length of the contained data in the block. A GBAS broadcast always has a fixed FEC length of 6 bytes. In addition, compared to a VDLM2 burst, the FEC in a GBAS burst is appended in the exact opposite bit order.

Table 4.2: VDLM2 data length and the resulting number of FEC bytes appended

Length of data	FEC bytes transmitted / generated
Less than 3 bytes	0 / 6
3 to 30 bytes	2 / 6
31 to 67 bytes	4 / 6
More than 67 bytes	6 / 6

4.2.2 DumpVDL2

DumpVDL2 is a standalone VDL Mode 2 message decoder and protocol analyser developed by Tomasz Lemiech, available on github under the GNU General Public License. All work done here is based on version 1.6.0 of the software, released 19th of January 2019 [11]. Features that make it a particularly useful choice:

- Standalone: It is not dependent on other programs or software frameworks
- Well maintained: Bugs are being fixed and new features are added repeatedly
- SDR support: Build in support for a number of the most common SDRs
- I/Q file support: A log of I/Q samples can be decoded if there is no real time data

Compilation and installation instructions are well documented in the repository's *README.md*. During the work with this software, enabling DumpVDL2s Debug mode helped a considerably in observing what the software was decoding and allowing comparison to what a GBAS transmission should look like. In particular, the issue with transmission length was discovered by viewing the raw hexadecimal message and counting byte offsets.

Reserved bits

DumpVDL2 asserts that the reserved symbol is always transmitted as zero. For GBAS this is only the case for message blocks transmitted in slot 0. It does this in two stages, both that require some modification.

First, DumpVDL2 sets the reserved bits to zero and the FEC is performed. Since the FEC covers the reserved symbol as well, it will attempt to correct this symbol if the GBAS station has an SSID other than 0. As the FEC is only capable of correcting a single bit

error, it will fail for 4/8 SSIDs and use up all error correcting capabilities for an additional 3/8 SSIDs. It is sufficient to comment out the line that sets the reserved bits to zero for the FEC to perform normally, as shown in appendix B line 194.

Second, a sanity check of the reserved symbol is performed by comparing the header to a bitmask with reserved bits set to zero. This comparison will fail for 7/8 SSIDs, causing the message to be discarded. Since the symbol is not reserved in a GBAS burst, this check serves no purpose and is thus commented out in its entirety in line 200-205 of appendix B.

Transmission length

The problem with the difference in transmission length is observed in the following transmission that ends on a message of type 3, an empty message filled with the value 0x55:

```
[...]
aa cd e1 14 03 8a 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[...]
55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 ea de 17 d5
d9 b3 2e e1 28 b3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 ff 8d ba c1 6b 61
```

The message starts with a header (purple) and ends with the CRC32 bytes (green) for a total of 0x8a bytes, matching the value indicated by the header. The data block should have ended here, but 6 additional bytes are collected (orange) before interleaving zeroes and collecting the FEC (blue).

The “6 additional bytes” are in fact the actual FEC that is being grouped in with the data, while the receiver goes on to sample the transmitter ramp-down and noise as the FEC, inevitably causing the FEC algorithm to fail.

Correcting for the difference in data length is straightforward. The value found in the burst header is used to determine when enough bytes have been collected to make up the entire message. Subtracting the FEC length of 48 bits from this value, as shown in appendix B line 207, causes the data burst to be correctly interpreted as:

```
[...]
aa cd e1 14 03 8a 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[...]
55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 ea de 17 d5
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 d9 b3 2e e1 28 b3
```

Reed-Solomon FEC

While the FEC bytes are now correctly found and placed at the end of the burst, some additional changes are required for the implemented FEC algorithm to function correctly with GBAS data.

In order to determine how many bytes of FEC are included in the received transmission, DumpVDL2 has its own function in *decode.c*, *get_fec_octetcount*, which interprets the message according to Table 4.2. Modifying this function to always return an octet count of 6 is simply done by commenting out the main function body (Appendix B line 121 to 127).

In order to reuse RS-FEC the way it is implemented in DumpVDL2, it is required to swap the bit order. This is done in two stages. First, the order of the bytes are swapped around in the buffer (appendix B line 294 to 307). Then, the bit order in each byte is swapped (appendix B line 309 to 317) using a bit-shifting method by Anderson [18].

Getting data out

After the FEC is performed, the similarities between VDLm2 and GBAS end. We now want to transfer the received messages from DumpVDL2 to a program of our own choice for decoding. A form of inter-process communication (IPC) must be added in the appropriate location of the code. While this can be done in any number of ways, a method commonly known as a *named pipe* has been chosen for its simplicity in implementation. In practice, a named pipe is a file that works as a first in, first out (FIFO) buffer.

Setting up a named pipe only requires a few lines to be added in the main *dumpvdl2.c* file. These changes are shown in appendix B. After the pipe has been set up, data can be written to it. The required logic for outputting the GBAS messages over the pipe is added to *decode_vdl_frame* in *decode.c* after the FEC correction has been performed. The implementation shown in appendix B (line 334 to 345) writes the separate messages to the FIFO one at a time, instead of the entire data burst contents.

It uses the fact that every message should start on 0xAA to determine when all messages have been read from a burst. The end of each message is found by reading the message length byte, found in position 5 from the start of message. An added benefit of this method is that messages received from GBAS stations that are in test mode (0xFF) will not be passed along for further decoding.

4.3 Decoding the GBAS broadcast

The messages can be read from the FIFO named pipe in a different process. A FIFO reader and partial GBAS message decoder module are written in python. This is a threaded module, such that it will run concurrently with the other tasks of the GBAS module discussed in chapter 4.4. Refer to appendix C for the Python code of this module.

4.3.1 CRC32 redundancy check

Every GBAS message ends with a 4 byte (32 bit) Cyclical Redundancy Check (CRC) with the polynomial:

$$G(x) = x^{32} + x^{31} + x^{24} + x^{22} + x^{16} + x^{14} + x^8 + x^7 + x^5 + x^3 + x + 1$$

This can be represented as a LSB first binary number where bit n represents whether x^n is part of the polynomial, e.g. $1 + x + x^3$ as 1101 = 0xd. The x^{32} term is implicit in the definition of the CRC polynomial, resulting in the hexadecimal representation of $G(x)$ as 0xd5828281.

The same CRC32 implementation that was used for Novatel GPS data (chapter 4.1.1) can be used with the new polynomial for GBAS message checking.

Since CRC32 is a method for error detection but not correction, an incoming GBAS message that fails the check must be considered corrupted and is discarded.

4.3.2 Processing datafields

The message, having passed both the FEC in the receiver and the CRC32 check in the python decoder, is now assumed not to contain undetected errors and the binary contents can be split into variables for use in navigation solutions.

At this point, additional sanity checks and filtering can be performed. Data fields that are good candidates for this are the *GBAS ID* and *Message type* fields in the message header, as the name of the desired GBAS station and message types are typically known. Messages originating from other GBAS stations or messages that are not desired can be discarded, requiring no further processing power.

For the contained data in the message body, a distinction is made on the *rate* of data, concerning how it should be handled in the receiver. This distinction originates from the description of MT1, but is here generalized and applied across all messages.

Full rate data

Full rate data for a message type is data that is contained in every single message of that type, and carries no value after a new message of the same kind is received. An example of this are the PRC and RRC values in MT1, where the values for all satellites in view are contained in every message.

Low rate data

Low rate data is split over multiple messages. For the entire dataset, it is required to collect and keep multiple messages of the same type. The low rate data in MT1, from which this class of data borrows its name, is an example of this. Each MT1 message contains the low rate data pertaining to a single satellite, and a number of messages have to be collected to complete the data for all satellites in view.

4.4 GPS calculations

The raw GPS data can be augmented with the decoded GBAS messages. This is done in the block marked "+" in Figure 3.8, which represents the combination of information from the two sources. The required calculations to achieve this will be introduced in chapter 4.4.1. As a part of this operation, smoothing of the GPS pseudoranges is required, which is presented in chapter 4.4.2. The main aspects of the position solution calculation, the last step in the GPS data chain from Figure 3.8, aspects of which will be shown in chapter 0.

4.4.1 Pseudorange correction

The corrections that are applied to the pseudoranges can come from either GBAS messages MT1 (for 100 second smoothed pseudoranges), or MT11 (for 30 second smoothed pseudoranges), the difference is mainly how fast the values are varying.

Having pseudoranges smoothed over a longer period will mean less noise, but will also mean that erroneous measurements in the filter will influence the filter output for a longer period of time.

Preferably, both 100 and 30 seconds corrected pseudorange data is computed in parallel such that the position solution calculator can switch between the two. Figure 4.6 shows a flowchart over how the corrections are applied in the aircraft, using equation 4.2 derived from the GBAS avionics standard [4].

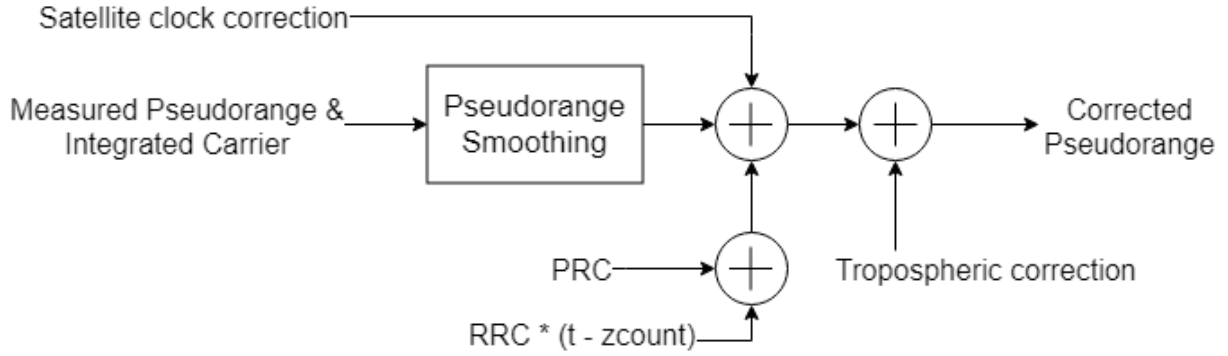


Figure 4.6: The corrections that are applied to the pseudoranges, visualized as flowchart (modified from [4])

$$P_{corrected} = \bar{P}_n + PRC + RRC * (t - t_{zcount}) + TC + c * (\Delta t_{sv})_{L1} \quad (4.2)$$

Equation 4.2 has to be applied to each ranging source. In this equation, $P_{corrected}$ is the pseudorange that is output to the position calculation, while \bar{P}_n is the filtered pseudorange as explained in chapter 4.4.2.

$(\Delta t_{sv})_{L1}$ is the clock correction broadcast by the satellite. Its influence on the range is found by multiplication with the speed of light (c), using equation 2.2.

PRC and RRC are the pseudorange and range-rate corrections from either MT1 or MT11. The range rate is projected ahead using the difference between current GPS time (t) and time of correction applicability (t_{zcount}). The z-count is reset every 20 minutes, while the time from the GPS receiver is typically given since the start of the week. The time formats can be related using modulo, as shown in equation 4.3 (with both time formats in seconds).

$$t_z = t_{week} \% 1200 \quad (4.3)$$

In equation 4.2, TC is the tropospheric correction, which compensates for the troposphere delay error. It is calculated based on the local conditions, and is influenced by the relative positioning of the aircraft, ground station and satellite. This is shown in equation 4.4, where both the aircraft altitude over GBAS station (Δh) and the satellite's elevation over the horizon (θ) are included. The refractivity index (N_R) and tropospheric scale height (h_0), are both received in MT2.

$$TC = N_R h_0 \frac{10^{-6}}{\sqrt{0.002 + \sin^2(\theta)}} (1 - e^{-\Delta h/h_0}) \quad (4.4)$$

Some of the variables in equation 4.4 are dependent on an already calculated aircraft position. In order to avoid a feedback loop, these variables should be taken from the uncorrected GPS position supplied by the GPS receiver. For instance, the perceived satellite elevation (θ) will not change significantly since the scale of corrections is small compared to the distance to the satellites.

4.4.2 Smoothing filter

The received pseudoranges from the GPS receiver are noisy. This receiver noise is smoothed using the measured phase in similar fashion to how it is done by the GBAS

ground station. This is contained in the “Pseudorange Smoothing” block in figure Figure 4.6.

The standard for GBAS avionics [4] shows the carrier smoothing as a two-step process shown in equation 4.5 and 4.6.

$$P_{proj} = \bar{P}_{n-1} + \frac{\lambda}{2\pi}(\phi_n - \phi_{n-1}) \quad (4.5)$$

$$\bar{P}_n = \alpha p_n + (1 - \alpha)P_{proj} \quad (4.6)$$

Initially, the smoothed range is the received pseudorange ($\bar{P}_1 = p_1$). For each measurement n , the two steps are performed. Equation 4.5 calculates the projected pseudorange (P_{proj}) based on the previous smoothed pseudorange (\bar{P}_{n-1}) and the change in phase since last sample ($\phi_n - \phi_{n-1}$), where λ is the GPS L1 carrier frequency of 1575.42 MHz.

Equation 4.6 is used to determine the new smoothed pseudorange (\bar{P}_n) based on a weighted sum of the new raw pseudorange sample (p_n) and the projected pseudorange (P_{proj}) from equation 4.5. The filter weighting (α) is determined based on the filter length and the frequency at which the samples are gathered:

$$\alpha = \frac{1}{t_{length} * f_{sample}} \quad (4.7)$$

For a 100 second filter as used for MT1 corrections, at a 4Hz rate matching the UAVs GPS configuration (chapter 4.1.2), this leads to a weighting of 1/400 raw pseudorange and 399/400 projected pseudorange based on phase (equation 4.6) for each iteration.

An implementation of the filter was made in python, and can be found in appendix C. Using a u-blox GPS evaluation module (based the LEA-6T chip), raw pseudorange samples were collected and filtered. Figure 4.7 shows the difference between the pseudorange and the filtered pseudorange for the same satellite. As can be seen, the high frequency noise is not passed on to the smoothed pseudorange.

While capturing data over a longer period of time, the jump shown in Figure 4.8 was observed. From sample 4439 to 4440, a 300-kilometre pseudorange jump occurred. This is exactly coherent with a 1ms clock correction.

As discussed in an article in *Inside GNSS* [19], clock corrections are something that can be observed with raw pseudoranges, but depends on the receiver used. For this particular receiver (u-blox LEA-6T), the error is thus observed to be limited to 1ms.

It is not known how the NovAtel receiver installed in the UAV handles this, though table 4 in the NovAtel firmware reference [8] mentions the time to be correct to the millisecond level. Being a high-end receiver it might very well use clock steering (clock is continuously adjusted to keep error to a minimum), though single millisecond clock corrections could also fit the description.

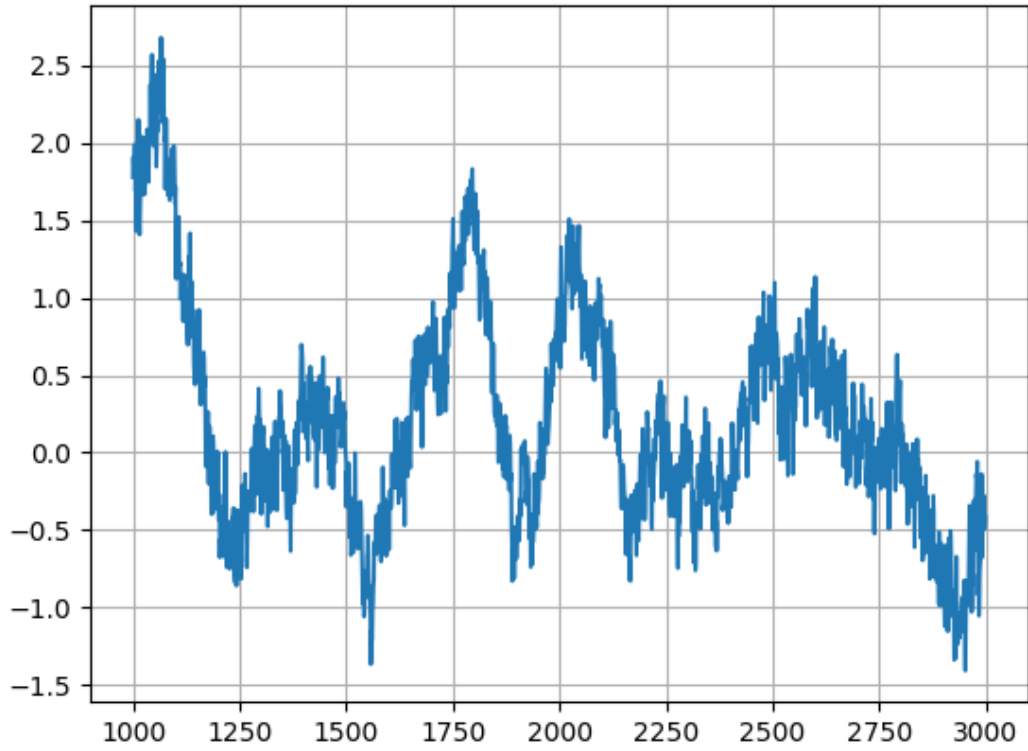


Figure 4.7: Difference between the received pseudorange and the 100 second smoothed pseudorange for a single satellite. Y-axis in meters, X-axis in samples at 4Hz

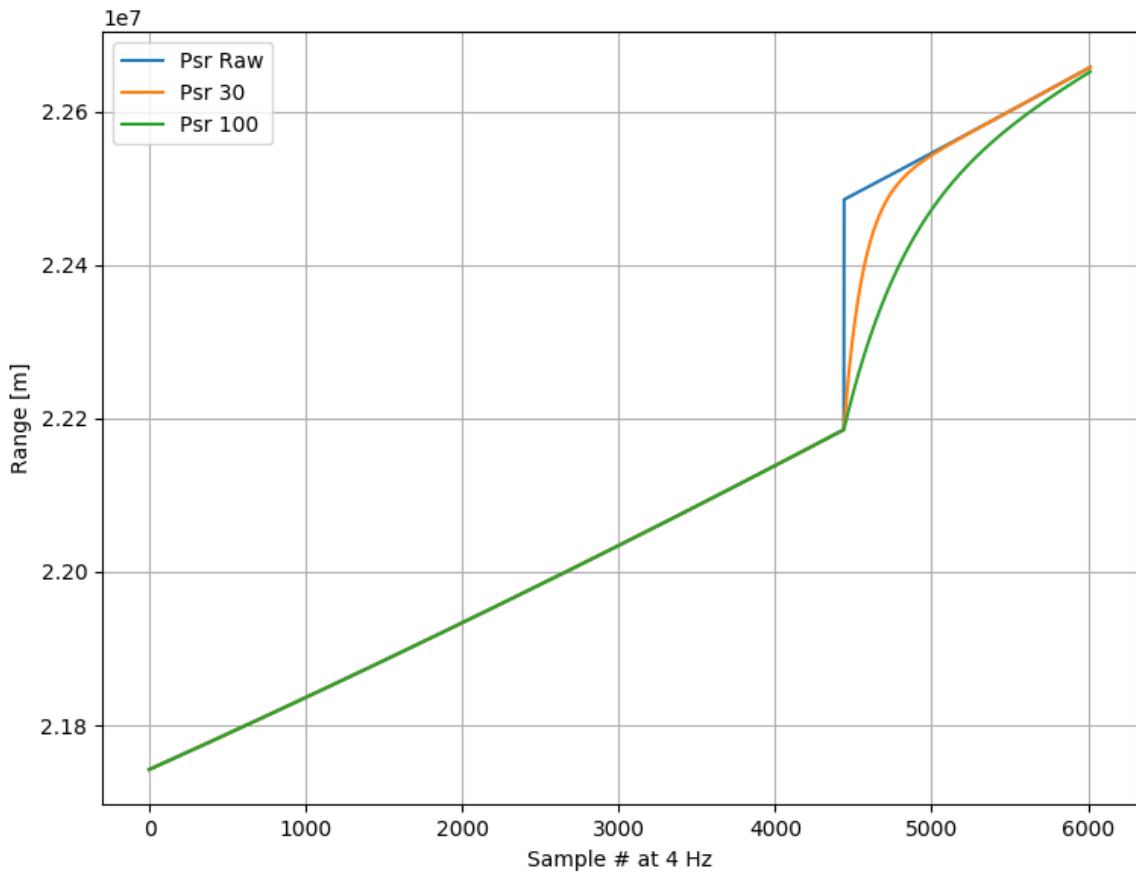


Figure 4.8: Log of pseudorange and filtered pseudorange, showing a 1ms clock jump

Since the clock corrections affect all pseudoranges the same way simultaneously, the change will cancel out in the position calculation and not cause any problem. This requires that these calculations do not involve previous pseudorange values.

The smoothing filter is dependent on previous values, by definition. In particular, it has problems with the discrepancy between a large jump in raw pseudorange and a lack of matching jump in phase. The error between the filtered value and raw pseudorange after a pseudorange jump is presented in Figure 4.9. Satellites tracked when the jump occurred will have filter errors in excess of 50 kilometers more than 100 seconds after the jump (sample 5000). A hypothetical new satellite being tracked from sample 4500, after the jump occurred, will have its 100 second filter in steady state by sample 5000 with errors similar to Figure 4.7 at less than 5 meters. This difference in satellite history cannot be detected by the positioning algorithm.

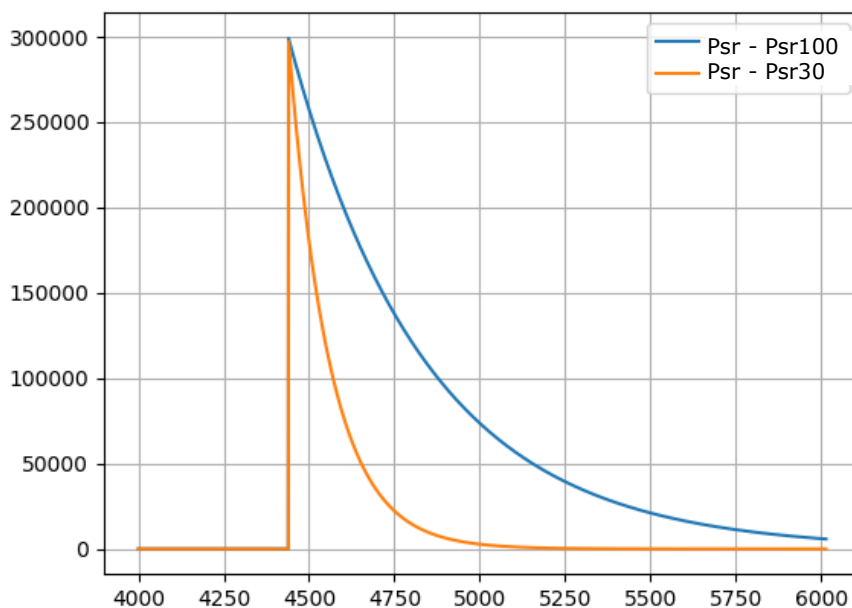


Figure 4.9: Error between the received pseudorange and the 30 and 100 second filtered pseudorange after a 1ms clock jump. Y-axis in meters, X-axis in samples at 4Hz

The airborne equipment standard [4] mentions, (as a note to measurement quality monitoring), that smoothed pseudorange should not be used if a pseudorange step has occurred. Seeing as the pseudorange jump in this case is an exact integer number of milliseconds, the jump can be added to the smoothed pseudorange before new samples are added, such that the abrupt step occurs the same across all pseudoranges and filtered pseudoranges. The smoothed pseudoranges can then be used continuously, and the position calculation will cancel out the jump without delay.

The occurrence of a jump is easily tested by comparing a new pseudorange to current filtered pseudorange. The smallest jump possible of 1ms is observed as about 300km change of distance, which is easy to detect.

While such jumps are generally not desired, the plot (Figure 4.9) shows how the 30-second and 100-second filters differ in response time.

4.4.3 Position solution calculation

In order to find the receiver position, four unknowns have to be determined; the location in three dimensions and the clock offset. With the corrected pseudoranges from at least

four satellites and the collected satellite ephemeris, the position of the receiver can then be calculated. Methods are extensively documented in books [14] and standards [4].

As the methods are standard, and multiple available implementations exist, it can be beneficial for the development time to base the software on a ready-made solution. RTKLIB [20] is such an open source program package written in C, that contains a number of algorithm implementations for the calculation of position-velocity-time (PVT) solutions. It is split in parts that can be included in other projects, and the licence allows its use in both commercial and non-commercial products. This is very suitable for the development of a GBAS module.

Two GPS receivers were used during the work with this thesis, the NovAtel OEMV2 in the UAV and a u-blox LEA-6T. For each receiver, a simple serial message receiver and decoder was written. RTKLIB includes message-decoding capabilities for a number of commonly used formats, including the two receivers used here. As such, RTKLIB will allow for a variety of GPS receivers to be used with the GBAS module.

5 Avionics Hardware

Two prototypes for the required UAV hardware for GBAS demodulation and decoding were designed, based on different single-board computers (SBCs).

The choice of software defined radio (SDR) was the first variable of hardware design that had to be solved. The choice fell on the RTL-SDR V.3, shown in Figure 5.1, as it fulfilled the key requirements of being supported by DumpVDL2, inexpensive and easily available. All further hardware design was made with this SDR in mind.



Figure 5.1: The RTL-SDR V.3, an inexpensive, easily available and much used SDR that is well suited for GBAS reception (photo rtl-sdr.com)

5.1 Hardware interfaces

The hardware interfaces required were determined based on existing hardware in the UAV and the requirements of the SDR receiver. These requirements for then informed the choice of single board computer (SBC) to be used.

5.1.1 Serial port

The serial port is a variant of the RS232 standard for serial communication. This port is generally considered obsolete in consumer products, but it is still commonly used in scientific, industrial and specialist equipment.

The Piccolo autopilot, being very much a specialist device, offers serial ports as the main method of interface with payloads and avionics. All communication between the autopilot and the Novatel GPS receiver occurs over a single serial connection.

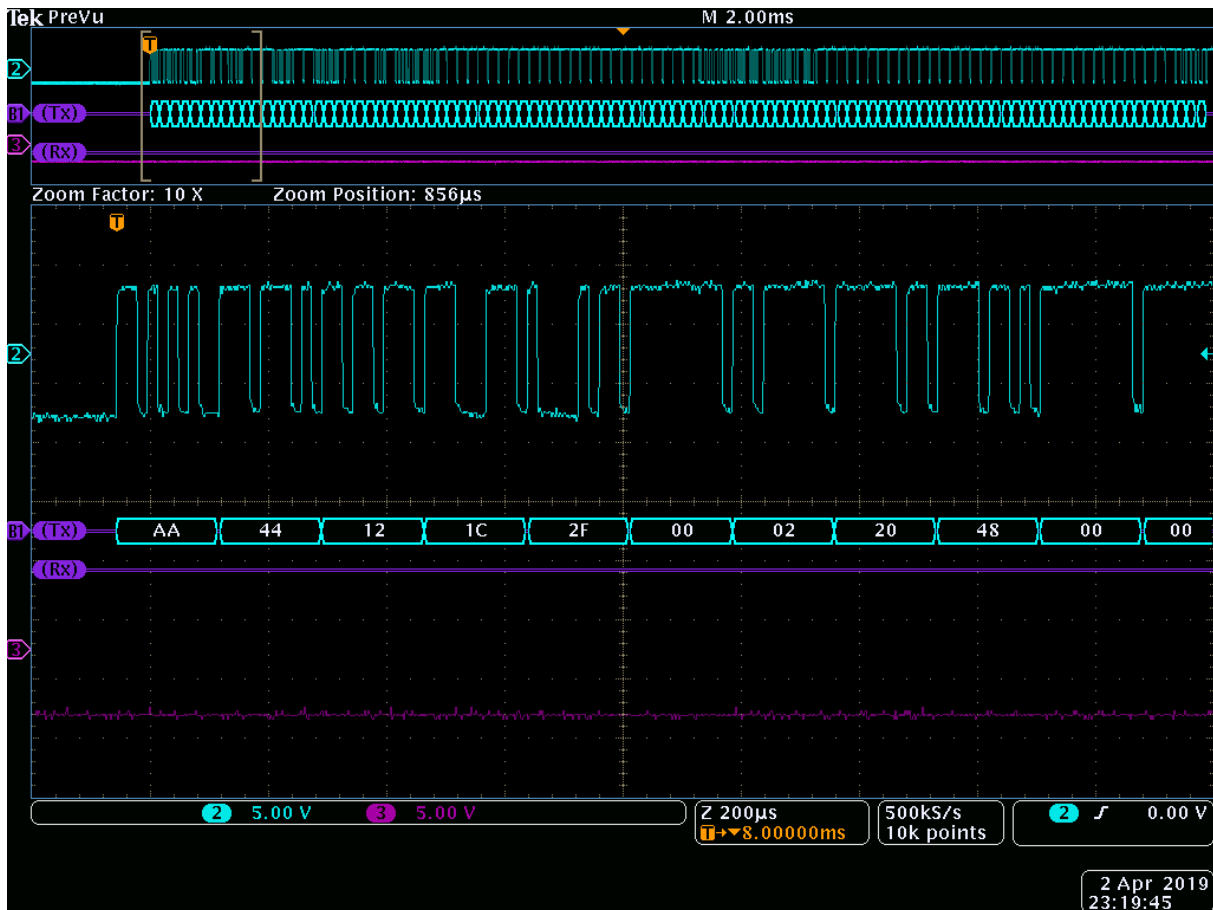


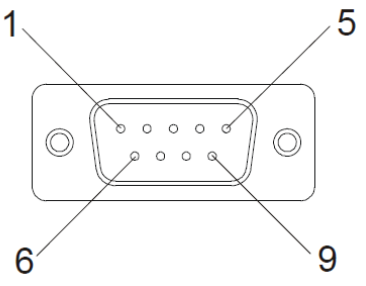
Figure 5.2: Oscilloscope trace of a binary message sent from the GPS receiver over the serial port. The signal switches between $\pm 5V$, and the live decoding of the data is shown.

The connection consists of two lines, transmit (TX) and receive (RX). The voltage on the line signals the bit value, a negative voltage represents a 1 while a positive voltage represents 0 (see Table 5.2). The Data bits are sent over their respective wire at a selected rate, known as the Baud-rate.

The two lines operate independently from each other, such that sending and receiving can occur at the same time. However, RX from one device must go to TX of the other, and vice versa. Depending on the pinout of the connectors on the devices to be connected, this might mean a crossover cable can be required.

The GPS side of the connection uses a classic 9-pin D-sub connector typically used for serial ports. Of the 9 pins, only 3 are of interest, shown in Table 5.1. The remaining pins can be used for flow control signals, but these are not used by the autopilot.

Table 5.1: Pins of interest for serial communication on the GPS male D-sub 9 connector

Pin on GPS serial port	Function	
2	Receive, RX	
3	Transmit, TX	
5	Ground, GND	

As the connector is large and bulky and serial ports are no longer commonly used, no single board computers that offer a serial connection out of the box could be found. Converters from both UART and USB exist, so these have been used instead.

5.1.2 UART

Universal Asynchronous Receiver-transmitter (UART) is a protocol for data transport, very similar to the RS232 communication of the serial port, down to the RX/TX naming convention and baud-rate determined timing. The main differences are the voltage levels, which operate at the digital signal voltage level of the system. Being so similar, there are a large number of UART-signal translation chips available. These chips typically contain charge pumps so that they can generate the required positive and negative voltage themselves with only a few external capacitors.

Table 5.2: Voltage levels for RS232 and UART communication

Data bit	Serial (RS232) voltage	UART voltage
0	+3V to +15V	Low (0V)
1	-3V to -15V	High (3.3V or 5V)

5.1.3 USB

Universal Serial Bus (USB) is a modern interface for communication. It has replaced the serial port in most applications. In USB, data is sent in packets, over a differential pair of lines. Either the host PC or the device can send, one at a time, making this a half-duplex port. In order to use a USB device, the host computer needs to install the device's driver software.

USB also carries power in the form of a 5V line that can provide up to 500mA. This is what powers the RTL-SDR when it is connected, and allows active converters to be attached. Converters from USB to most of the common data ports are easily available, making USB a very flexible solution.

5.1.4 CAN bus

Control Area Network (CAN) is a message based differential bus where any node can send to any other node. With automatic message collision and error detection it is a robust data bus with a high level of security, commonly used in cars and vehicles in general.

It is therefore not a surprise that such features are supported in UAVs as well. The piccolo has a CAN bus that can be used to interface with avionics and payloads. Sending to a specific address allows data transfer to the ground station, which makes it a candidate for communication between pilot on the ground and the GBAS module.

Complete details of the CAN bus can be found in the official CAN Specification [21].

5.2 BeagleBone based system

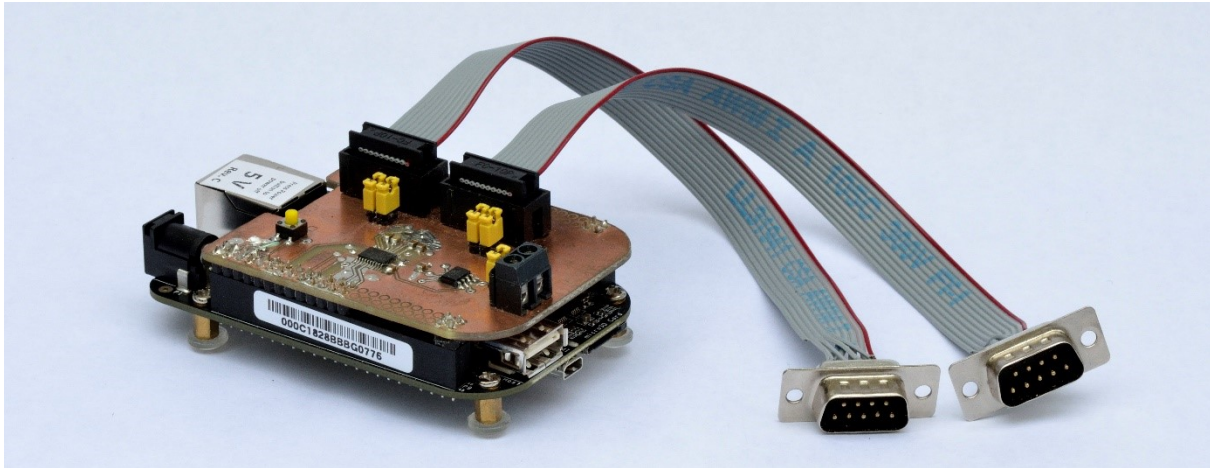


Figure 5.3: The BeagleBone Black rev. C single board computer, with the custom circuit board for serial ports and CAN bus attached.

Figure 5.3 shows the first hardware prototype, which is based on the BeagleBone Black rev. C. This is a single board computer based around Texas Instrument’s AM3358 processor, a single core 1GHz ARM cortex-A8. It has a host USB port available for the RTL-SDR to connect, and two 46-pin headers with a number of external interfaces.

Among these interfaces are four UARTs and two CAN ports. A circuit board was developed with the required transceivers that allow the BeagleBone to connect to two serial devices as well as a CAN bus.

The unit is powered by a 5V supply, via either a mini-USB connector or a barrel plug. When connecting it to a PC’s USB port it emulates a USB modem, which allows connecting to it over secure shell (SSH) at the fixed IP address *192.168.7.2* for configuration and programming.

5.2.1 BeagleBone design

The system is built up of the BeagleBone Black rev. C, running Debian 9, with a custom circuit board (known as a “cape” in the BeagleBone community) connected to the pin headers. A close-up of the cape is seen in Figure 5.4, and the design schematics are provided in appendix D. It offers two separate functions:

Serial ports

Two serial ports are required, in order to communicate with the Novatel GPS receiver and the Piccolo autopilot. This is achieved by using an ST3222 dual UART – RS232 bridge chip, which is able to generate RS232 voltages from the system 3.3V with its own built-in charge pump and only requires a number of external capacitors.

On the UART side, it is connected to the BeagleBone's UART1 and UART4. On the RS232 side, the RX/TX line pairs go to 2x2 headers with jumpers (see yellow jumpers on Figure 5.4). The signals are connected in such a way that rotating the jumpers 90 degrees swaps the RX and TX lines, which could be required in some connector configurations.

From the 2x2 headers, the signals lead into connection blocks for ribbon cables leading to D-sub 9 connectors. The connection is straight through, such that pin 1 on the connection block leads to pin 1 of the D-sub 9 plug.

CAN bus

The cape also provides an MCP2562 CAN transceiver. This is required in order to connect the pins of the BeagleBone to the physical CAN bus. It provides the differential transmit ability and gives protection of the BeagleBone hardware. This particular chip was chosen for its compatibility with 3.3V logic signals. A jumper allows the inclusion of a 120Ω bus-termination resistor. The CAN differential pair can be connected to the physical bus via a screw terminal block.

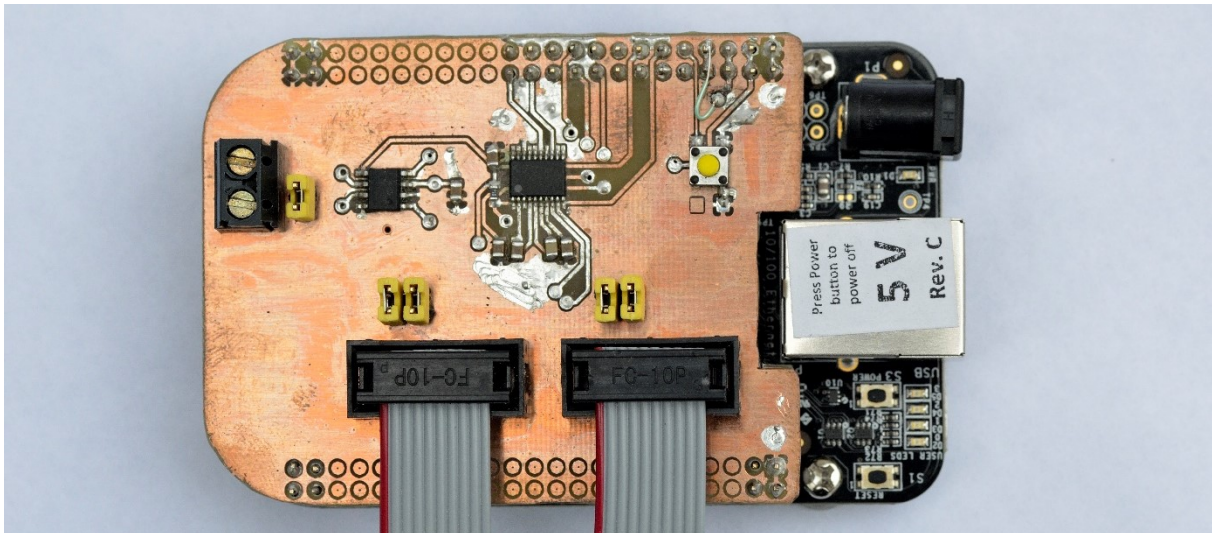


Figure 5.4: Top down view of the serial and CAN circuit board

The general-purpose input/output (GPIO) pins on the BeagleBone headers can have multiple possible functions. In the circuit board design, the two serial ports are connected to UART 1 and UART 4. In order to enable the UARTs, the pins have to be configured to UART mode for their respective RX and TX lines. Furthermore, the serial ports by default have "echo" mode enabled. Echo makes the serial port reply back any message it receives, which is an unwanted feature in this application. This is configured using the commands:

```
config-pin p9.11 uart
config-pin p9.13 uart
config-pin p9.24 uart
config-pin p9.26 uart

stty -F /dev/ttyO4 sane -echo
stty -F /dev/ttyO1 sane -echo
```

5.2.2 BeagleBone evaluation

The final system, as shown in Figure 5.3, weights 82 grams. This does not include the SDR receiver nor a protective case, so the weight cannot be directly compared to the Raspberry Pi system in chapter 5.3.

The operating system can be run either from a micro SecureDigital (microSD) card, or from the embedded MultiMediaCard (eMMC). The weaknesses of such storage devices, in particular their response to sudden power loss, is discussed in chapter 7.5.

The serial ports were tested by connecting them to serial-USB adapters plugged into a computer. Using serial terminal software, data could be sent to and received from the BeagleBone system.

In order to test the BeagleBone with the RTL-SDR, DumpVDL2 was run in its unmodified form (chapter 4.2.2). The BeagleBone was not able to keep up with the simultaneous collection of samples and decoding of data. This led to that not a single VDL2 packet could be received. It was verified that there was in fact VDL2 activity in the area by running the same RTL-SDR setup on a laptop both before and after the test.

Htop, a process viewer for Linux, showed a CPU usage of 100% continuously while the radio was on. It was thus concluded that the single 1GHz core had insufficient processing power for the application. This motivated the design of a new hardware prototype based on a more powerful computer.

5.3 Raspberry Pi based system



Figure 5.5: The Raspberry Pi 3 Model B+ single board computer, with the serial port modules as well as the SDR.

The second hardware prototype, based on the Raspberry Pi 3 Model B+ is shown in Figure 5.5. This single board computer is far more powerful than the BeagleBone from the first system, as it features a Broadcom BCM2837B0 processor, a quad core 1.4GHz ARM cortex-A53. It has four USB ports available for the RTL-SDR and other hardware to connect. A 40-pin header allows a few external interfaces; however, it does not have two UARTs available. The serial ports were therefore realised using USB converters.

The unit is powered by a 5V supply via a micro-USB connector. Having a far more powerful CPU also means additional power demand. Unlike the BeagleBone, the 500mA

from a computer is not enough to power it. A 2.5A phone charger or similar power supply is recommended.

Not having a USB modem interface to the device also makes configuring and programming somewhat more complex, as SSH is not enabled by default. This can be done using the *raspi-config* utility, which requires the connection of a screen and keyboard. The RPi can then be connected directly with an Ethernet cable to a laptop's own Ethernet port. This is desired as it allows connection to the RPi also in areas without a common Wi-Fi connection or Ethernet switch.

Unlike the BeagleBone, the RPi does not have a static IP address where the computer knows how to find it. This can be achieved by configuring static IPs manually on the RPi and laptop Ethernet adapter. An easier method is installing Apple's Bonjour Print Services. It will detect the RPi, such that SSH connections can be made to *raspberrypi.local* instead of a fixed IP address.

5.3.1 Raspberry Pi design

The system is built up of the Raspberry Pi 3 Model B+, running Debian 9. The required interfaces were realised using USB port converter modules, and the entire module was mounted in a custom enclosure, making for a single compact unit. A top-down view of the assembled module is seen in Figure 5.6.

Serial ports

The serial ports were made using two separate converter stages. First, UART ports are made using CP2102 USB-UART bridges. The devices are automatically detected by Debian, and no additional drivers have to be installed.

The UART signals are then passed to UART – serial bridges based on the SP3232 chip. These chips function the same way as the chip used on the BeagleBone cape. The modules are mounted to the RPi using plastic standoffs. By connecting this assembly to the RPi using short USB cables, the entire construction can be folded into a compact unit.

SDR

The SDR that was chosen for GBAS reception was the RTL-SDR V.3. As the casing of this SDR is quite bulky, it must be mounted with a USB extension cable in order to not block the other USB ports. The SDR is mounted to the baseplate alongside the RPi, and the shortest available extension cable was chosen. The large USB connectors themselves mean that even a 10cm cable takes a lot of space compared to the module size as a whole, as can be seen in Figure 5.6.

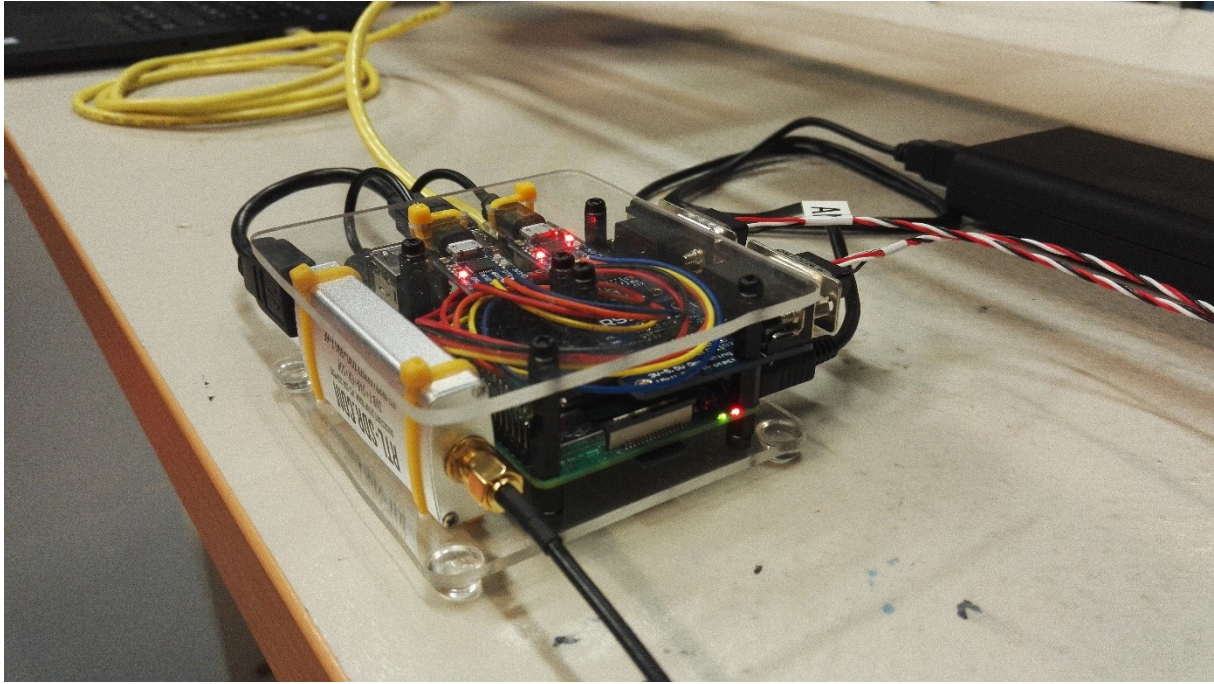


Figure 5.7: The RPi based system powered up and communicating with the GPS and autopilot in the UAV

6 UAV Implementation

The available room inside the UAV is divided into multiple compartments. The largest of these is by far the front of the aircraft, where mission payloads up to 20kg can be installed. The fuel tank takes a good part of space as well, supplying fuel for long missions of up to 8 hours. The remainder is left for the avionics hardware, its power system and batteries. The UAV's motor is mounted to the outside of the UAV behind the main wing, allowing the UAV to fly at nominal speeds of 60 to 70 knots. The motor also contains a generator, which will keep the batteries charged and vehicle powered during normal operation. The total UAV is approximately 3.3 meters long. The various compartments and their approximate relative size are visualised in Figure 6.1.

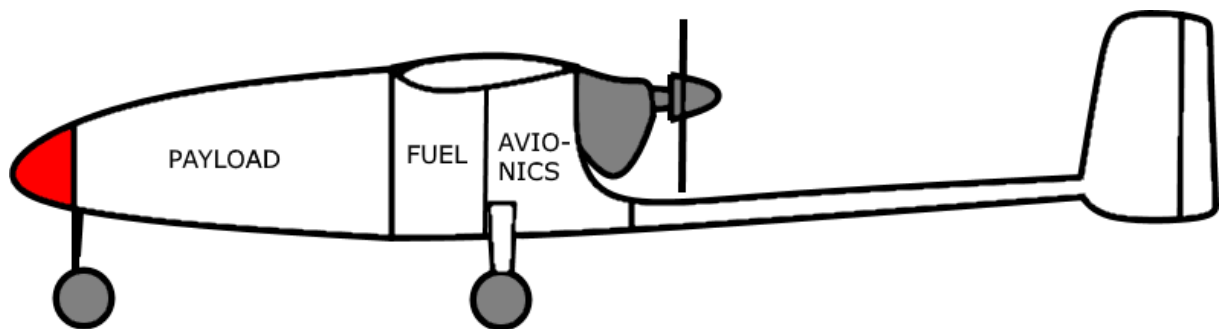


Figure 6.1: Illustration of the different compartments in the Cruiser 2 UAV

6.1 Avionics bay

The avionics bay can be accessed from the front by removing the payload bay and the fuel tank. Limited access from the top is also possible by removing the wing. The UAV is built in such a way that all parts are securely attached yet easily disassembled for transport and service.

Figure 6.2 shows the view of the avionics bay from the front, after the fuel tank is removed. The yellow fuel tubes can be seen hanging loose from the top right and left corners of the figure. The large metal box in the centre (C) is a protective, fireproof container for the battery packs. The NovAtel OEMV2 GPS receiver (A) is mounted on its side such that it fits next to the battery box. It is connected to the Cloud Cap Piccolo 2 autopilot (B) that is placed at the bottom of the assembly. On top of the autopilot is an Iridium satellite modem, which is used for low rate communication when UAV is operating outside of radio range of the ground station.

The autopilot (B) is placed square and centre as it also contains the gyros and accelerometers of the contained inertial measurement unit (IMU). The entire assembly is mounted on rubber vibration dampers (D) such that the IMU is not adversely influenced by vibrations and shocks of the airframe [6].

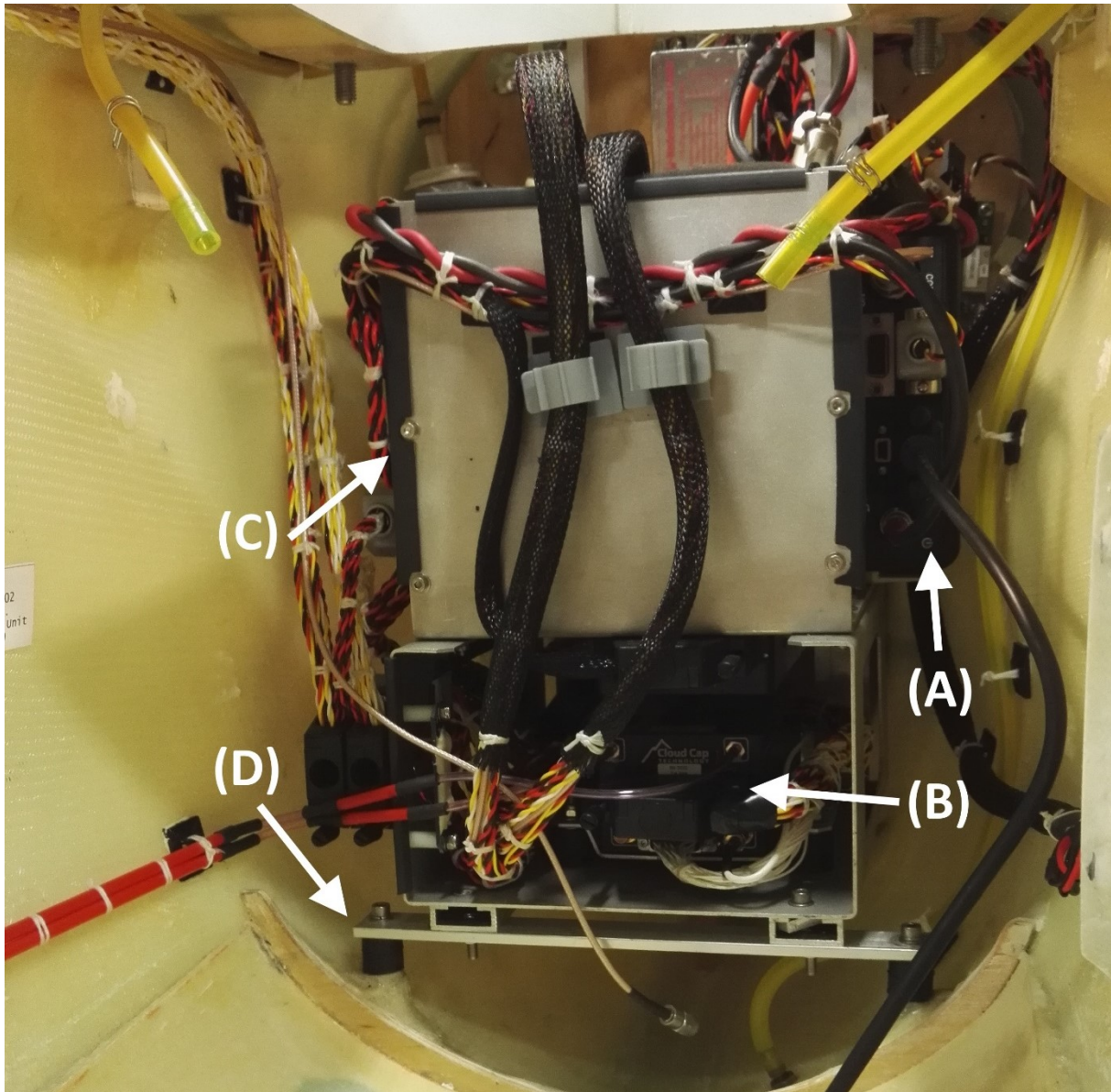


Figure 6.2: View of the Cruiser 2 avionics bay, as seen from the front with the fuel tank removed, clearly showing the GPS receiver (A), autopilot (B), battery compartment (C) and vibration dampeners (D)

There is not much space left in the avionics bay, and only two possible locations for the placement of the GBAS avionics module are available:

- 1) Upside-down underneath the autopilot holder
- 2) On its side, next to the autopilot and below the GPS receiver

The current prototype is not suitable for either position, as the connectors of the module point in multiple directions, making it hard to access. Ideally, all the GBAS module interfaces should be located next to each other, such that they are easily accessible from a single front plate in similar fashion to how it is done on the GPS.

Mounting option 2 is deemed the better of the two, as the module would be easier to access and be in close vicinity to the GPS module to which it must be connected.

6.2 Payload bay

The payload bay of the Cruiser 2 UAV is large and open, allowing for a variety of payload implementations to be mounted. During testing of GBAS in UAVs, the GBAS module can easily be placed here alongside other payloads.

Figure 6.3 shows the payload bay with the cover off, with a camera gimbal payload mounted. The Raspberry Pi based GBAS module prototype is placed on the workbench next to the UAV, showing the relative size of the available space.

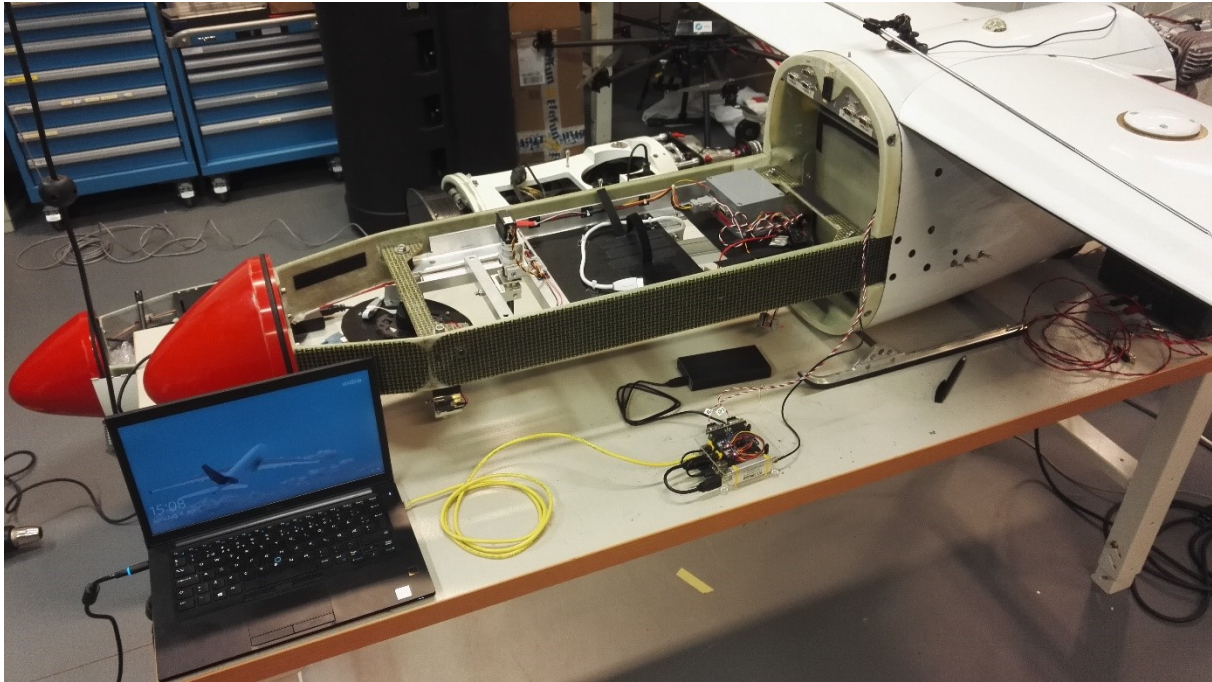


Figure 6.3: The Cruiser 2 payload bay. Raspberry Pi GBAS module on the workbench demonstrates the amount of space available

Right in front of the main wing, where the payload area begins, there is a plate with connectors mounted. These are pass-through interfaces, where payloads can send data to the autopilot, which will forward them to the pilot on the ground. One of these connectors can be used to send GBAS information for the pilot to consider.

However, the GPS-autopilot interface is not available here. In order to place the GBAS module in the payload bay, two additional serial cables have to be drawn from the avionics bay.

6.3 Antenna Placement

In order to receive the GBAS broadcast, a VHF antenna is required on board the UAV. The GBAS broadcast is horizontally polarised, so an antenna has to be mounted horizontally as well. This makes the wings good locations for antenna mounting.

Placing antenna along the tail boom has also been considered, but this would give the antenna low reception when flying directly at the station, which typically would happen during landing.

6.3.1 Current antennae

There are already antennae mounted on the wings. The additional GBAS antenna must not interfere with their functionality. The antennae mounted on the centre part of the main wing are shown in Figure 6.4, and can be seen in Figure 6.5.

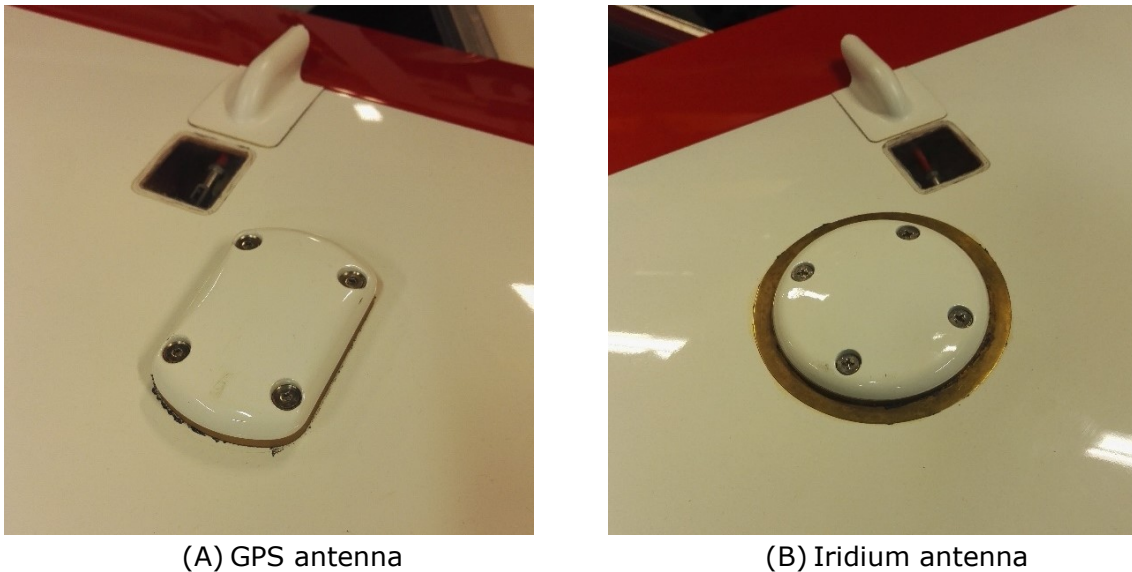


Figure 6.4: The GPS antenna (A) and Iridium antenna (B) that are mounted on the center part of the main wing

6.3.2 GBAS VHF antenna mounting

The wings as placement for the GBAS VHF antenna have been explored at ASC. The examples shown here are using a dipole antenna of 128cm, which is approximately half the wavelength of an average GBAS carrier wave frequency, and a suitable length for GBAS VHF reception.

Figure 6.5 shows the antenna mounted on top of the main wing. The GPS antenna and Iridium antenna can be clearly seen. It is a benefit to have the antenna mounted towards the front of the wing, as it reduces the possibility for the UAVs airframe to get in the line of radio reception during landing. At the same time, the main wing contains a carbon fibre beam for wing reinforcement. The carbon fibre will absorb the signal and negatively affect the signal strength.

Figure 6.6 shows the same antenna mounted on top of the tail wing. Being far back, a lot of the UAV can get in the way of the antenna during landing. This wing is not used for other antennae, and it does not contain any carbon fibre.

Since the broadcast that is to be received originates from the ground, the signal will be received from below for the main part of the flight. The antenna can therefore also be mounted on the underside of either wing, which could improve reception significantly, though blockage by the airframe as well as absorption by carbon fibre beams would remain influential.

In order to determine which antenna type and location is actually best suited for the UAV GBAS implementation, a more in-depth test would have to be performed where performance data for each location is collected and analysed.

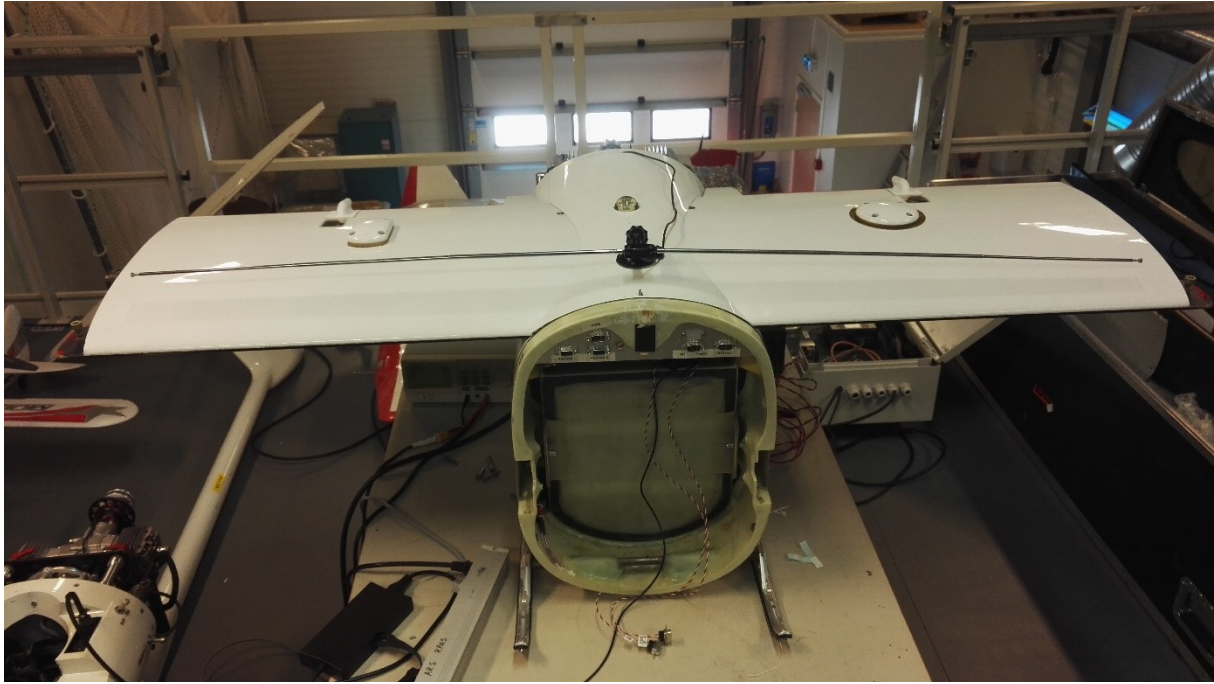


Figure 6.5: VHF antenna mounted on top of the main wing



Figure 6.6: VHF antenna mounted on top of the tail wing

6.4 Interface verification

In order to test if the interface between the GPS and autopilot was understood by the GBAS software, the laptop was connected in between the autopilot and the GPS. Messages on each port were collected, message contents checked with the CRC32, and output on the other port with a new generated checksum.

The test was successful, and demonstrated that the GBAS module software, running on the laptop, was able to identify messages and logs correctly on the serial port, and both collect and transmit them. Decoding and encoding of the CRC32 error detection bytes was also confirmed to be operational.

The same code was also tested on the Raspberry Pi GBAS module prototype, shown in Figure 5.7, with identical results, indicating that the serial interfaces added to the RPi were correctly working, and that both decoding and encoding of messages according to the NovAtel GPS format was possible.

After the indoor tests on the workbench were found to be successful, the avionics bay was moved outside so that the same test could be done while the GPS could receive actual satellite data.

Figure 6.7 shows the avionics bay of the UAV, with the center part of the main wing mounted, placed in the parking lot of ASC for the collection of GPS data. The red container is placed as a warning for cars driving around the corner not to crush the UAV.



Figure 6.7: Avionics compartment with center-part of wing placed outside in the parking lot at ASC, collecting GPS data.

7 Results and Discussion

The main results of the study of the system requirements and the technical details for the concept of GBAS integration in UAV systems are discussed in this chapter. The main focus is on the necessary software adaptation and the options for using off-the-shelf hardware. Ways to improve the current system and potential for future applications are also presented.

7.1 GBAS demodulation and decoding software

In chapter 4.2 it was demonstrated that the similarities between GBAS and VDLM2 signals in space allowed a software based VDLM2 receiver to be modified for GBAS broadcast demodulation. The GBAS messages contained in the data burst were put in a FIFO buffer, such that any program could read them out and process them further.

Using python, the messages were read from the buffer and selected datafields decoded. CRC32 checking is done to get the highest possible certainty no corrupt messages are accepted by the GBAS module.

7.2 Correcting raw GPS data

When GBAS messages are received, these can be used to augment the UAVs GPS data. Chapter 4.4 explored how the contents of MT1, MT2 and MT11 can be used to generate improved pseudoranges, which will provide a higher positional accuracy than can be achieved using regular GPS.

In order to minimize receiver noise, the pseudoranges observed by the UAV have to be smoothed in a similar fashion to how it is done by the GBAS ground station. This is done using the carrier phase. The smoothing was implemented in python, and a test was performed. This led to the identification of millisecond jumps in the pseudorange data.

Such a jump is detrimental to the filtered values if not caught and handled properly, as it presents a large discrepancy between the pseudorange and carrier phase. Furthermore, depending on which receiver is used, the jump can be of different size. Since it always is an integer number of milliseconds [19], the shortest jump is just under 300 kilometre, which is easy to detect.

The time jump can cause issues for the application of GPS corrections. The range-rate corrections (RRCs) are applied based on the difference between the time they were generated (supplied in the message) and the "current" time of application. The simplest way of acquiring the current time would be gathering the time from the GPS receiver directly, so that no feedback from the PVT solution calculation is required. The max value for the RRC is 32.767m/s (appendix A) such that a clock error limited to 1ms can at worst cause pseudorange errors of 3.3cm. Some receivers can have clock errors up to 100ms [19], which would cause unacceptable errors of up to 3.3m. As such, either the GPS receiver should be of a kind that is limited to small clock errors, or the implementation must use the time from the PVT solution.

Using RTKLIB is recommended for the PVT solution generation, as it has a large number of GNSS algorithms implemented. In addition, it supplies decoders for a large number of message types for GPS receivers from known brands.

7.3 Reception of GBAS using off-the-shelf components

As seen in chapter 5, the hardware required for GBAS demodulation and decoding is in practice limited to a computer and an SDR dongle. As multiple software tasks have to be performed in parallel (chapter 4), the system benefits greatly from a multi-core processor. This was evident when the BeagleBone's single CPU was overloaded by SDR sampling and demodulation alone.

The prototype based on the Raspberry Pi is in many ways quite similar to the BeagleBone, but the Pi features a far more powerful processor. This was noticeable, as GBAS reception and decoding used only about $\frac{1}{4}$ of available processing time.

The single board computers used are easy to get hold of, both from supplies of electronic components and certain computer chain stores. In the RPi module, both USB to UART and UART to serial converter boards were used. This made for a neat, compact design. However, direct USB to serial cable could also be used with the same result. Such cables are commonly found in computer hardware stores.

Due to the high volume production of the computers, converters and other components used, the total price of the assembled system is small when compared to the cost of a high-end GPS receiver or autopilot.

7.4 Installation of GBAS module in existing UAV systems

An abundance of available interface translation boards makes a general-purpose computer a flexible platform for controlling hardware interfaces of any kind. Equipping a single board computer with the required hardware for communicating with the existing UAV components can be as simple as plugging in a USB dongle.

With the evolution of complex autonomous systems and payloads in UAVs, it is reasonable to assume that modern, high-bandwidth interfaces will also become more prevalent. A development platform based on a single board computer is well suited for such a future, with gigabit Ethernet and high-speed USB available in the more recent models.

The resulting module is both light and small. While the avionics bay of the Cruiser 2 did not have room for the prototype module, it could easily be placed in the payload area of the Cruiser 2 without being a great hindrance to actual mission payloads.

However, there does exist a significant limitation to the general applicability of the UAV implementation as it is used throughout this thesis (Figure 3.7), specifically with regard to UAV integration. A core requirement is that the GPS and the autopilot of the UAV are separate modules, with a connection in between which the GBAS module can intercept. While a new GPS module can be added alongside the GBAS module in a UAV system, the autopilot must support GPS positioning by an external module in one way or another.

7.5 Selection of hardware

Hardware was partially selected based on availability. Commonly used single board computers are easily available, but are not designed for or originally intended for critical applications, such as an avionics module would be.

A challenge with single board computers such as the Raspberry Pi and the BeagleBone, which the prototypes in chapter 5 are based on, is that their storage is based on SD cards or eMMC chips. If power is abruptly lost, these can become corrupted. A complete formatting may be required.

Corruption typically happens when power is lost during a writing operation. In order to avoid this, there should either be enough power backup (e.g. from a chargeable battery) to finish the writing, or no writing should be done to minimize the risk.

The FIFO that is used for inter-process communication in the application presented in this thesis is a file on the drive, but no actual writing is done to the file. All data is passed between applications by memory alone. As such, the current system does not have a significant risk of corruption in its current state, but the risk is something that should be considered if new functionality is to be added. Writing log files would for instance pose a significant risk, due to the frequent write operations associated.

7.6 “DumpGBAS”

DumpVDL2 features far more functionality than is being used in the GBAS modification. It allows for collecting stats and logging multiple frequencies simultaneously, as well as decoding a number of different packaged data formats typically send over VDL2. These functions are not relevant to the GBAS receiver and demodulator, but are still present. This is not desired for a few reasons:

- It can waste processing power that would be better used elsewhere, in turn leading to additional power drawn by the overall system
- It increases the overall size of the software with functions and modules that are never called on. It also increases compilation time though that is mostly a nuisance during software development
- It introduces unnecessary dependencies to other software. Change in the software dependencies might entirely change how DumpVDL2 works, potentially introducing bugs or breaking the GBAS implementation in unexpected ways

This motivates the need for a variation, a GBAS-only build, which has here been given the nickname “DumpGBAS”. Generally, there are two possible approaches by which this can be achieved:

- Subtractive method, by trimming down DumpVDL2 and removing all parts that are not relevant to the GBAS demodulation. This requires good insight into the internal structure of the software so that desired functionality is not affected.
- Additive method, by creating an entirely new project and reusing GBAS-relevant functions, algorithms and code from DumpVDL2.

Of the two methods, the second is preferable as it allows for a transparent software design, which allows easier integration of new functionality. For instance, the currently ignored GBAS SSID could be used to filter out signals from unwanted stations. It is also desirable to eliminate the FIFO buffer as it adds delay. Currently it is needed for inter-

process communication, but by combining demodulation and decoding in the same program, it could be made obsolete.

RTKLIB, mentioned in chapter 0, could also be added along with the correction algorithms such that all functionality is contained in a single program. This will make for a more robust GBAS module.

7.7 Potential implications of GBAS navigation in UAVs

A GBAS navigation platform for UAVs has the advantage of increasing positioning accuracy over regular GPS. The GBAS system is based on international standards that assure high system integrity regardless of environmental factors.

Weather related challenges have been limiting UAV operations, not the least in the proximity of inhabited areas. GBAS allows safe landing of aircraft under low- to no-sight conditions. Aurora interference on airfields in the northern regions can also cause problems for the of GNSS signals. GBAS can detect these critical situations, provide integrity support and thereby improve operational safety.

In terms of pure position accuracy, pseudorange-based GBAS system is not able to outcompete phased-based RTK position solutions. However, being standardised, GBAS can enable remote UAV landing at any GBAS enabled airfield. This will allow for long-range unmanned flights, without requiring a pilot on location for landing, refuelling or maintenance.

The tested GBAS system poses limited constraints with regard to cost, weight and size. Future avionics developers might therefore be interested in adding this capability to their new avionics products.

The benefit of increased safety and greater reliability is especially of interest for operators of larger UAVs. This will allow for extended capabilities for unmanned operations in the future.

8 Conclusion

The study of the system requirements for integrating GBAS in UAV avionics showed that it was possible to design a system that could be integrated neatly with existing avionics. No significant modification of the existing UAV hardware was required.

The system was test fitted in an existing UAV system at ASC. This reduced the available payload capacity minimally. Possible locations for the required additional VHF antenna on the UAV were explored. Mounting the antenna was found to be possible with only minimal modification of the airframe.

In order to demodulate the GBAS broadcast, open-source software for the reception of VDLM2 signals was successfully modified. The program is able to collect GBAS data bursts, perform error correction and output the contained messages for further processing. The content of the broadcast is decoded and the contained data are made available for correcting the GPS pseudoranges.

Raw pseudoranges from a GPS receiver were smoothed using the carrier phase. The implications of pseudorange jumps in the data were discussed, both in the context of the smoothing filter and the pseudorange corrections. The method by which information from the messages in the GBAS broadcast can be combined with the smoothed pseudoranges for generating corrected pseudoranges was presented.

Two different hardware prototypes have been developed, each system based on different off-the-shelf single-board computers. Testing showed that the GBAS software was quite CPU intensive, such that one of the systems was not able to keep up with data flow. The Raspberry Pi based system was able to handle the radio samples and communicate with the UAV avionics at ASC.

GBAS will allow for extended UAV capabilities, since it can detect critical GNSS situations and provide integrity support. This makes it possible to operate UAVs under low- to no-sight conditions. In addition, it enables remote UAV landing at any GBAS enabled airfield. This will extend the capabilities of unmanned operations in the future.

In conclusion, it has been possible to develop a low cost system that allows the integration of GBAS corrections with UAV avionics, which will improve its autonomous landing capabilities of UAVs, independent of airfield and weather conditions.

References

- [1] P. D. Breedveld, Landing Unmanned Aerial Vehicles using a Ground Based Augmentation System, Trondheim: NTNU, 2018.
- [2] EUROCAE, ED-114B MOPS For Global Navigation Satellite Ground Based Augmentation System Ground Equipment To Support Category I Operations, EUROCAE WG-28, 2018.
- [3] SC-159 RTCA, Inc., RTCA DO-246E GNSS-Based Precision Approach Local Area Augmentation System (LAAS) Signal-in-Space Interface Control Document (ICD), Washington: RTCA, Inc., 2017.
- [4] SC-159 RTCA, Inc., RTCA DO-253D Minimum Operational Performance Standards for GPS Local Area Augmentation System Airborne Equipment, Washington: RTCA, Inc., 2017.
- [5] B. Alex, B. Dan, M. Van and V. Bill, PCC User's Guide v.2.2.1, Cloud Cap Technology, 2013.
- [6] J. Hammitt and D. Miley, Piccolo Vehicle Integration Guide, Cloud Cap Technology, 2011.
- [7] Magline, Cruiser 2 Pilot's operating handbook, Spain: Magline Composites y Sistemas, 2018.
- [8] NovAtel Inc., OM-20000094 Rev 8, OEMV® Family Firmware Reference Manual, Calgary: NovAtel Inc., 2010.
- [9] B. Vaglianti, Piccolo Communications v2.1.4.f, Cloud Cap Technology, 2012.
- [10] M. Zanmiller and D. Miley, Piccolo Setup Guide, Cloud Cap Technology, 2011.
- [11] T. Lemiech, "dumpvdl2 v1.6.0," 19 January 2019. [Online]. Available: <https://github.com/szpajder/dumpvdl2>. [Accessed 20 February 2019].
- [12] The National Coordination Office for Space-Based Positioning, Navigation, and Timing, "Official U.S. government information about the Global Positioning System (GPS) and related topics," National Coordination Office for Space-Based Positioning, Navigation, and Timing, 5 November 2018. [Online]. Available: gps.gov/. [Accessed 20 March 2019].
- [13] Global Positioning System Directorate, IS-GPS-200 Navstar GPS Space Segment/Navigation User Interfaces, Global Positioning System Directorate, 2018.
- [14] P. Misra and P. Enge, Global Positioning System, Signals, Measurements, and Performance, Massachusetts: Ganga-Jamuna Press, 2012.

- [15] T. Murphy and T. Imrich, "Implementation and Operations Use of Ground-Based Augmentation Systems (GBASs) - A component of the Future Air Traffic Management System," *Proceedings of the IEEE*, vol. 96, no. 12, pp. 1939-1957, 2008.
- [16] Samferdselsdepartementet, "Forskrift om luftfartøy som ikke har fører om bord," 30 11 2015. [Online]. Available: <https://lovdata.no/dokument/SF/forskrift/2015-11-30-1404>. [Accessed 10 5 2019].
- [17] WAVECOM, VDL-M2 Aeronautical data link - advanced protocols, Switzerland: Wavecom elektronik AG, 2018.
- [18] S. E. Anderson, "Bit Twiddling Hacks," 4 February 2011. [Online]. Available: <https://graphics.stanford.edu/~seander/bithacks.html>. [Accessed 1 April 2019].
- [19] M. Petovello, "Inside GNSS - Are there special considerations for dealing with raw GNSS data," May 2015. [Online]. Available: <https://insidegnss.com/are-there-special-considerations-for-dealing-with-raw-gnss-data/>. [Accessed 1 May 2019].
- [20] T. Takasu, "RTKLIB: An Open Source Program Package for GNSS Positioning," 13 May 2019. [Online]. Available: <http://www.rtklib.com/>. [Accessed 1 June 2019].
- [21] Robert Bosch GmbH, CAN Specification Version 2.0, Stuttgart: Robert Bosch GmbH, 1991.

Appendices

Appendix A: GBAS message content by type number

Appendix B: DumpVDL2 source code modifications

Appendix C: Python modules for GBAS decoding and use

Appendix D: Schematics and design files for BeagleBone RS232 cape

Appendix A: GBAS message content by type number

This appendix contains listings of the data fields contained in each GBAS message type. The tables for each are close-to-exact copies of similar tables found in the GBAS standards [1] [2], and are provided here for reference when the messages and their data fields are discussed.

MT1:

Message type 1 contains pseudorange corrections for 100-second smoothed pseudorange data. Table A.1 shows all the contained data fields. The table can be found as 3.7-2 in [1] and 2-12 in [2]. Reference these sources for full details on all data fields.

Table A.1: Data content of message type 1 [1][2]

Data Content	Bits	Range of Values	Resolution
Modified Z-count	14	0 to 1199.9 sec	0.1 sec
Additional Message Flag	2	0 to 3	1
Number of measurements (N)	5	0 to 18	1
Measurement Type	3	0 to 7	1
Ephemeris Decorrelation Parameter	8	0 to $1.275 * 10^{-3}$ m/m	$5 * 10^{-6}$ m/m
Ephemeris CRC	16	-	-
Source Availability Duration	8	0 to 2540 s	10 s
For N measurement blocks:			
Range Source ID	8	1 to 255	1
Issue of Data	8	0 to 255	1
Pseudo-range Correction (PRC)	16	± 327.67 m	0.01 m
Range Rate Correction (RRC)	16	± 32.767 m/s	0.001 m/s
σ_{pr_gnd}	8	0 to 5.08 m	0.02 m
B_1	8	± 6.35 m	0.05 m
B_2	8	± 6.35 m	0.05 m
B_3	8	± 6.35 m	0.05 m
B_4	8	± 6.35 m	0.05 m

MT2:

Message type 2 contains information on the GBAS ground station, both its location and configuration. Table A.2 shows all the contained data fields. The table can be found as 3.7-4 in [1] and 2-14 in [2]. Reference these sources for full details on all data fields, as well as the additional data blocks.

Table A.2: Data content of message type 2 [1][2]

Data Content	Bits	Range of Values	Resolution
GBAS Reference Receivers	2	2 to 4	-
Ground Accuracy Designator	2	-	-
<i>Spare</i>	1	-	-
GBAS Continuity/Integrity Designator	3	-	-
Local Magnetic Variation	11	$\pm 180^\circ$	0.25°
<i>Spare</i>	5	-	-
$\sigma_{vert_ino_gradient}$	8	0 to $25.5 * 10^{-6}$ m/m	$0.1 * 10^{-6}$ m/m
Refractivity Index (N_R)	8	16 to 781	3
Scale Height (h_0)	8	0 to 25500 m	100 m
Refractivity Uncertainty	8	0 to 255	1
GBAS Reference Point Latitude	32	$\pm 90^\circ$	0.0005 arcsec
GBAS Reference Point Longitude	32	$\pm 180^\circ$	0.0005 arcsec
GBAS Reference Point Height	24	± 83886.07 m	0.01 m
<i>Additional data blocks may be provided</i>			

MT3:

Message type 3 contains only filler data. Table A.3 shows the contained data field, *filler*, which contains a sequence of bits alternating between 0 and 1 [2]. The table can be found as 3.7-7 in [1] and 2-17 in [2].

Table A.3: Data content of message type 3 [1][2]

Data Content	Bits	Range of Values	Resolution
Filler	Variable	-	-

MT4:

Message type 4 contains sets of approach data and associated alarm limits. Table A.4 shows all the contained data fields. The table can be found as 3.7-5 in [1] and 2-18 in [2]. Reference these sources for full details on all data fields.

Table A.4: Data content of message type 4 [1][2]

Data Content	Bits	Range of Values	Resolution
For N data sets:			
Data set length	8	2 to 212 bytes	1 byte
FAS data block	304	-	-
FAS vertical alert limit/approach status	8	0 to 25.4 m	0.1 m
FAS horizontal alert limit/approach status	3	0 to 50.8 m	0.2 m

MT11:

Message type 11 contains pseudorange corrections for 30-second smoothed pseudorange data. Table A.5 shows all the contained data fields. The table can be found as 3.7-9 in [1] and 2-27 in [2]. Reference these sources for full details on all data fields.

Table A.5: Data content of message type 11 [1][2]

Data Content	Bits	Range of Values	Resolution
Modified Z-count	14	0 to 1199.9 sec	0.1 sec
Additional Message Flag	2	0 to 3	1
Number of measurements (N)	5	0 to 18	1
Measurement Type	3	0 to 7	1
Ephemeris Decorrelation Parameter	8	0 to $1.275 * 10^{-3}$ m/m	$5 * 10^{-6}$ m/m
For N measurement blocks:			
Range Source ID	8	1 to 255	1
Pseudo-range Correction (PRC_{30})	16	± 327.67 m	0.01 m
Range Rate Correction (RRC_{30})	16	± 32.767 m/s	0.001 m/s
$\sigma_{pr_gnd_D}$	8	0 to 5.08 m	0.02 m
$\sigma_{pr_gnd_30}$	8	0 to 5.08 m	0.02 m

References:

- [1] EUROCAE, ED-114B MOPS For Global Navigation Satellite Ground Based Augmentation System Ground Equipment To Support Category I Operations, EUROCAE WG-28, 2018.
- [2] SC-159 RTCA, Inc., RTCA DO-246E GNSS-Based Precision Approach Local Area Augmentation System (LAAS) Signal-in-Space Interface Control Document (ICD), Washington: RTCA, Inc., 2017.

Appendix B: DumpVDL2 source code modifications

This appendix summarises all changes added to the source code of DumpVDL2. The changes are sorted by the file in which they apply.

dumpvdl2.h

The definition of the GBAS FIFO location and name of this file has been placed in *dumpvdl2.h*, since it is included in all source files. This allows all files have access to its definition:

```
#define FIFO "/tmp/gbasfifo"
```

dumpvdl.c

The FIFO has to be enabled at the start of the program. This requires two system library files to be included in *dumpvdl2.c*:

```
//For named pipes
#include <sys/stat.h>
#include <sys/types.h>
```

Where after the FIFO is enabled at the start of the *main* function by the following line, where the number 0666 represents the required file permissions to use the FIFO:

```
mkfifo(FIFO, 0666);
```

decode.c

The changes made to the *decode.c* have been tracked in the following table, and full source code for the modified file is provided.

Line	Change	Effect
36 - 38	Added	Library imports for data export over named pipe (FIFO)
121 - 127	Commented	Function should always return 6
194	Commented	Bits should not be forced to 0
200 - 205	Commented	Sanity check not applicable to GBAS
207	Modified	Subtracted 48 from value
294 - 307	Added	Swap RS-FEC byte order in buffer
309 - 317	Added	Changed bit order in RS-FEC bytes
334 - 343	Added	Pipe out messages over FIFO
344	Added	Bypass further processing

```
1. /* decode.c
2.  * dumpvdl2 - a VDL Mode 2 message decoder and protocol analyzer
3.  * Copyright (c) 2017-2019 Tomasz Lemiech <szpajder@gmail.com>
4.  *
5.  * Modified for GBAS broadcast demodulation
6.  * Changed 2019 Petter Breedveld <petter.breedveld@gmail.com>
7.  *
8.  * This program is free software: you can redistribute it and/or modify
9.  * it under the terms of the GNU General Public License as published by
10. * the Free Software Foundation, either version 3 of the License, or
11. * (at your option) any later version.
```



```

12. *
13. * This program is distributed in the hope that it will be useful,
14. * but WITHOUT ANY WARRANTY; without even the implied warranty of
15. * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
16. * GNU General Public License for more details.
17. *
18. * You should have received a copy of the GNU General Public License
19. * along with this program. If not, see <http://www.gnu.org/licenses/>.
20. */
21. #define _GNU_SOURCE
22. #include <stdio.h>
23. #include <stdint.h>
24. #include <stdlib.h>
25. #include <string.h>
26. #include <limits.h>
27. #include <unistd.h>
28. #include <glib.h>
29. #include "config.h"
30. #ifdef WITH_STATSD
31. #include <sys/time.h>
32. #endif
33. #include "dumpvdl2.h"
34. #include "avlc.h" // avlc_frame_qentry_t, frame_queue
35.
36. #include <fcntl.h>
37. #include <sys/stat.h>
38. #include <sys/types.h>
39.
40. // Reasonable limits for transmission lengths in bits
41. // This is to avoid blocking the decoder in DEC_DATA for a long time
42. // in case when the transmission length field in the header gets
43. // decoded wrongly.
44. // This applies when header decoded OK without error corrections
45. #define MAX_FRAME_LENGTH 0x3FFF
46. // This applies when there were some bits corrected
47. #define MAX_FRAME_LENGTH_CORRECTED 0x1FFF
48.
49. #define LFSR_IV 0x6959u
50.
51. static uint32_t const H[HDRFECLLEN] = {
52.     0b000000001111111111110000,
53.     0b0011111100001111111101000,
54.     0b1100011100110000111100100,
55.     0b1101101101010011001100010,
56.     0b01101001111100101010100001
57. };
58.
59. static uint32_t const syndtable[1<<HDRFECLLEN] = {
60.     0b00000000000000000000000000000000,
61.     0b00000000000000000000000000000001,
62.     0b00000000000000000000000000000010,
63.     0b01000000000000000000000000000100,
64.     0b00000000000000000000000000000100,
65.     0b01000000000000000000000000000100,
66.     0b10000000000000000000000000000000,
67.     0b01000000000000000000000000000000,
68.     0b000000000000000000000000000001000,
69.     0b00100000000000000000000000000000,
70.     0b00010000000000000000000000000000,

```

```

71.     0b00001000000000000000000000000000,
72.     0b00000100000000000000000000000000,
73.     0b10001000000000000000000000000000,
74.     0b00000010000000000000000000000000,
75.     0b00000001000000000000000000000000,
76.     0b000000001000000000000000000010000,
77.     0b00000000010000000000000000000000,
78.     0b01000000001000000000000000000000,
79.     0b00000000010000000000000000000000,
80.     0b01000000010000000000000000000000,
81.     0b00000000010000000000000000000000,
82.     0b00000000001000000000000000000000,
83.     0b10000000100000000000000000000000,
84.     0b00000000000100000000000000000000,
85.     0b00000000000010000000000000000000,
86.     0b00000000000001000000000000000000,
87.     0b00000000000000100000000000000000,
88.     0b00000000000000010000000000000000,
89.     0b00000000000000001000000000000000,
90.     0b00000000000000000100000000000000,
91.     0b00000000000000000010000000000000,
92. };
93.
94. static uint32_t const synd_weight[1<<HDRFECLLEN] = {
95.     0, 1, 1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 2, 1, 1, 2, 1, 1
96.     , 1, 1, 1, 1, 1, 1
97. };
98. uint32_t parity(uint32_t v) {
99.     uint32_t parity = 0;
100.     while (v) {
101.         parity = !parity;
102.         v = v & (v - 1);
103.     }
104.     return parity;
105. }
106.
107. uint32_t decode_header(uint32_t * const r) {
108.     uint32_t syndrome = 0u, row = 0u;
109.     int i;
110.     for(i = 0; i < HDRFECLLEN; i++) {
111.         row = *r & H[i];
112.         syndrome |= (parity(row)) << (HDRFECLLEN - 1 - i);
113.     }
114.     debug_print("received: 0x%x syndrome: 0x%x error: 0x%x, decoded: 0x%x\n",
115.         *r, syndrome, syndtable[syndrome], *r ^ syndtable[syndrome]);
116.     *r ^= syndtable[syndrome];
117.     return syndrome;
118. }
119.
120. int get_fec_octetcount(uint32_t len) {
121.     /*if(len < 3)
122.         return 0;
123.     else if(len < 31)
124.         return 2;
125.     else if(len < 68)
126.         return 4;
127.     else*/
128.     return 6;

```

```

129.     }
130.
131.     static int deinterleave(uint8_t *in, uint32_t len, uint32_t rows, uint32_t cols,
132.         uint8_t out[][cols], uint32_t fillwidth, uint32_t offset) {
133.         if(rows == 0 || cols == 0 || fillwidth == 0)
134.             return -1;
135.         uint32_t last_row_len = len % fillwidth;
136.         if(last_row_len == 0) last_row_len = fillwidth;
137.         if(fillwidth + offset > cols) // fillwidth or offset too large
138.             return -2;
139.         if(len > rows * fillwidth) // result won't fit
140.             return -3;
141.         if(rows > 1 && len - last_row_len < (rows - 1) * fillwidth) // not enough data to fill requested width
142.             return -4;
143.         if(last_row_len == 0 && len / fillwidth < rows) // not enough data to fill requested number of rows
144.             return -5;
145.         uint32_t row = 0, col = offset;
146.         last_row_len += offset;
147.         for(uint32_t i = 0; i < len; i++) {
148.             if(row == rows - 1 && col >= last_row_len) {
149.                 out[row][col] = 0x00;
150.                 row = 0;
151.                 col++;
152.             }
153.             out[row][col] = in[i];
154.             if(row == rows) {
155.                 row = 0;
156.                 col++;
157.             }
158.         }
159.         return 0;
160.     }
161.
162.     static void enqueue_frame(vdl2_channel_t const * const v, int const frame_num,
163.         uint8_t *buf, size_t const len) {
164.         avlc_frame_gentry_t *gentry = XCALLOC(1, sizeof(avlc_frame_gentry_t));
165.         gentry->buf = XCALLOC(len, sizeof(uint8_t));
166.         memcpy(gentry->buf, buf, len);
167.         gentry->len = len;
168.         gentry->freq = v->freq;
169.         gentry->frame_pwr = v->frame_pwr;
170.         gentry->mag_nf = v->mag_nf;
171.         gentry->ppm_error = v->ppm_error;
172.         gentry->burst_timestamp.tv_sec = v->burst_timestamp.tv_sec;
173.         gentry->burst_timestamp.tv_usec = v->burst_timestamp.tv_usec;
174.         if(extended_header) {
175.             gentry->datalen_octets = v->datalen_octets;
176.             gentry->synd_weight = synd_weight[v->syndrome];
177.             gentry->num_fec_corrections = v->num_fec_corrections;
178.             gentry->idx = frame_num;
179.         }
180.         g_async_queue_push(frame_queue, gentry);
181.     }
182.
183.     void decode_vdl_frame(vdl2_channel_t *v) {
184.         switch(v->decoder_state) {
185.             case DEC_HEADER:

```

```

184.         v->lfsr = LFSR_IV;
185.         bitstream_descramble(v->bs, &v->lfsr);
186.         uint32_t header;
187.         if(bitstream_read_word_msbfirst(v->bs, &header, HEADER_LEN) < 0) {
188.             debug_print("%s", "Could not read header from bitstream\n");
189.             statsd_increment(v->freq, "decoder.errors.no_header");
190.             v->decoder_state = DEC_IDLE;
191.             return;
192.         }
193.         // force bits of reserved symbol to 0 to improve chances of successful decode

194.         //header &= ONES(TRLEN+HDRFECLLEN);
195.         v->syndrome = decode_header(&header);
196.         if(v->syndrome == 0) {
197.             statsd_increment(v->freq, "decoder.crc.good");
198.         }
199.         // sanity check - reserved symbol bits shall still be set to 0
200.         /* if((header & ONES(TRLEN+HDRFECLLEN)) != header) {
201.             debug_print("%s", "Rejecting decoded header with non-
202. zero reserved bits\n");
203.             statsd_increment(v->freq, "decoder.crc.bad");
204.             v->decoder_state = DEC_IDLE;
205.             return;
206.         }*/
207.         header >>= HDRFECLLEN;
208.         v->datalen = reverse(header & ONES(TRLEN), TRLEN) - 48;
209.         // Reject payloads with unreasonably large length (in theory longer frames ar
210. e allowed but in practice
211. // it does not happen - usually it means we've locked on something which is n
212. ot a preamble. It's safer
213. // to reject it rather than to block the decoder in DEC_DATA state and readin
214. g garbage for a long time,
215. // possibly overlooking valid frames.
216.         if((v->syndrome != 0 && v-
217. >datalen > MAX_FRAME_LENGTH_CORRECTED) || v->datalen > MAX_FRAME_LENGTH) {
218.             debug_print("v->datalen=%u v->syndrome=%u - frame rejected\n", v-
219. >datalen, v->syndrome);
220.             statsd_increment(v->freq, "decoder.errors.too_long");
221.             v->decoder_state = DEC_IDLE;
222.             return;
223.         }
224.         v->datalen_octets = v->datalen / 8;
225.         if(v->datalen % 8 != 0)
226.             v->datalen_octets++;
227.         v->num_blocks = v->datalen_octets / RS_K;
228.         v->fec_octets = v->num_blocks * (RS_N - RS_K);
229.         v->last_block_len_octets = v->datalen_octets % RS_K;
230.         if(v->last_block_len_octets != 0)
231.             v->num_blocks++;
232.
233.         v->fec_octets += get_fec_octetcount(v->last_block_len_octets);
234.
235.         debug_print("Data length: %u (0x%x) bits (%u octets), num_blocks=%u,
236. last_block_len_octets=%u fec_octets=%u\n",
237. v->datalen, v->datalen, v->datalen_octets, v->num_blocks, v-
238. >last_block_len_octets, v->fec_octets);
239.
240.         if(v->fec_octets == 0) {

```

```

233.         debug_print("%s", "fec_octets is 0 which means the frame is unrea
          sonably short\n");
234.         statsd_increment(v->freq, "decoder.errors.no_fec");
235.         v->decoder_state = DEC_IDLE;
236.         return;
237.     }
238.     v->requested_bits = 8 * (v->datalen_octets + v->fec_octets);
239.     v->decoder_state = DEC_DATA;
240.     return;
241.     case DEC_DATA:
242.     #ifdef WITH_STATSD
243.         gettimeofday(&v->tstart, NULL);
244.     #endif
245.         bitstream_descramble(v->bs, &v->lfsr);
246.         uint8_t *data = XCALLOC(v->datalen_octets, sizeof(uint8_t));
247.         uint8_t *fec = XCALLOC(v->fec_octets, sizeof(uint8_t));
248.         if(bitstream_read_lsbfirst(v->bs, data, v-
          >datalen_octets, 8) < 0) {
249.             debug_print("%s", "Frame data truncated\n");
250.             statsd_increment(v->freq, "decoder.errors.data_truncated");
251.             goto cleanup;
252.         }
253.         if(bitstream_read_lsbfirst(v->bs, fec, v->fec_octets, 8) < 0) {
254.             debug_print("%s", "FEC data truncated\n");
255.             statsd_increment(v->freq, "decoder.errors.fec_truncated");
256.             goto cleanup;
257.         }
258.         debug_print_buf_hex(data, v->datalen_octets, "%s", "Data:\n");
259.         debug_print_buf_hex(fec, v->fec_octets, "%s", "FEC:\n");
260.         {
261.             uint8_t rs_tab[v->num_blocks][RS_N];
262.             memset(rs_tab, 0, sizeof(uint8_t[v->num_blocks][RS_N]));
263.             int ret;
264.             if((ret = deinterleave(data, v->datalen_octets, v-
          >num_blocks, RS_N, rs_tab, RS_K, 0)) < 0) {
265.                 debug_print("Deinterleaver failed with error %d\n", ret);
266.                 statsd_increment(v-
          >freq, "decoder.errors.deinterleave_data");
267.                 goto cleanup;
268.             }
269.
270.             // if last block is < 3 bytes long, no FEC is done on it, so we should not wr
          ite FEC bytes into the last row
271.             uint32_t fec_rows = v->num_blocks;
272.             if(get_fec_octetcount(v->last_block_len_octets) == 0)
273.                 fec_rows--;
274.
275.             if((ret = deinterleave(fec, v-
          >fec_octets, fec_rows, RS_N, rs_tab, RS_N - RS_K, RS_K)) < 0) {
276.                 debug_print("Deinterleaver failed with error %d\n", ret);
277.                 statsd_increment(v-
          >freq, "decoder.errors.deinterleave_fec");
278.                 goto cleanup;
279.             }
280.     #ifdef DEBUG
281.         debug_print("%s", "Deinterleaved blocks:\n");
282.         for(uint32_t r = 0; r < v->num_blocks; r++) {
283.             debug_print_buf_hex(rs_tab[r], RS_N, "Block %d:\n", r);
284.         }

```

```

285.     #endif
286.         bitstream_reset(v->bs);
287.         for(uint32_t r = 0; r < v->num_blocks; r++) {
288.             statsd_increment(v->freq, "decoder.blocks.processed");
289.             int num_fec_octets = RS_N - RS_K; // full block
290.             if(r == v->num_blocks - 1) { // final, partial block
291.                 num_fec_octets = get_fec_octetcount(v-
>last_block_len_octets);
292.             }
293.
294.             //-----Swap byte endiannes
295.             uint8_t tmp;
296.             tmp = rs_tab[r][RS_N-6];
297.             rs_tab[r][RS_N-6] = rs_tab[r][RS_N-1];
298.             rs_tab[r][RS_N-1] = tmp;
299.
300.             tmp = rs_tab[r][RS_N-5];
301.             rs_tab[r][RS_N-5] = rs_tab[r][RS_N-2];
302.             rs_tab[r][RS_N-2] = tmp;
303.
304.             tmp = rs_tab[r][RS_N-4];
305.             rs_tab[r][RS_N-4] = rs_tab[r][RS_N-3];
306.             rs_tab[r][RS_N-3] = tmp;
307.             //-----
308.
309.             //-----Swap bit endianness
310.             for(int i = RS_N-6; i<RS_N; i++){
311.                 tmp = rs_tab[r][i];
312.                 tmp = (tmp >> 4) & 0x0F | (tmp & 0x0F) << 4;
313.                 tmp = (tmp >> 2) & 0x33 | (tmp & 0x33) << 2;
314.                 tmp = (tmp >> 1) & 0x55 | (tmp & 0x55) << 1;
315.                 rs_tab[r][i] = tmp;
316.             }
317.             //-----
318.
319.             ret = rs_verify((uint8_t *)&rs_tab[r], num_fec_octets);
320.             debug_print("Block %d FEC: %d\n", r, ret);
321.             if(ret < 0) {
322.                 debug_print("%s", "FEC check failed\n");
323.                 statsd_increment(v->freq, "decoder.errors.fec_bad");
324.                 goto cleanup;
325.             } else {
326.                 statsd_increment(v->freq, "decoder.blocks.fec_ok");
327.                 if(ret > 0) {
328.                     debug_print_buf_hex(rs_tab[r], RS_N, "Corrected block
%d:\n", r);
329.                     // count corrected octets, excluding intended erasures
330.                     v-
>num_fec_corrections += ret - (RS_N - RS_K - num_fec_octets);
331.                 }
332.             }
333.
334.             //-----GBAS data pipe out
335.             int gbaspipe;
336.             //write(gbaspipe, rs_tab[r], rs_tab[r][5]);
337.             int offset = 0;
338.             while(rs_tab[r][offset] == 0xaa){
339.                 gbaspipe = open(FIFO, O_WRONLY);

```

```

340.             write(gbaspipe, &rs_tab[r][offset], rs_tab[r][offset+5]);
341.             offset = rs_tab[r][offset+5];
342.             close(gbaspipe);
343.         }
344.         goto cleanup; //Short-circuit
345.     //-----
346.
347.         if(r != v->num_blocks - 1)
348.             ret = bitstream_append_lsbfirst(v-
>bs, (uint8_t *)&rs_tab[r], RS_K, 8);
349.         else
350.             ret = bitstream_append_lsbfirst(v-
>bs, (uint8_t *)&rs_tab[r], v->last_block_len_octets, 8);
351.         if(ret < 0) {
352.             debug_print("%s", "bitstream_append_lsbfirst failed\n");
353.             statsd_increment(v->freq, "decoder.errors.bitstream");
354.             goto cleanup;
355.         }
356.     }
357. }
358. // bitstream_append_lsbfirst() reads whole bytes, but datalen usually isn't a
multiple of 8 due to bit stuffing.
359. // So we need to truncate the padding bits from the end of the bit stream.
360.     if(v->datalen < v->bs->end - v->bs->start) {
361.         debug_print("Cut last %u bits from bitstream, bs-
>end was %u now is %u\n",
362.             v->bs->end - v->bs->start - v->datalen, v->bs->end, v-
>datalen);
363.         v->bs->end = v->datalen;
364.     }
365.     int ret;
366.     int frame_cnt = 0;
367.     while((ret = bitstream_copy_next_frame(v->bs, v->frame_bs)) >= 0) {
368.         if((v->frame_bs->end - v->frame_bs->start) % 8 != 0) {
369.             debug_print("Frame %d: Bit stream error: does not end on a by
te boundary\n", frame_cnt);
370.             statsd_increment(v-
>freq, "decoder.errors.truncated_octets");
371.             goto cleanup;
372.         }
373.         debug_print("Frame %d: Stream OK after unstuffing, length is %u o
ctets\n",
374.             frame_cnt, (v->frame_bs->end - v->frame_bs->start) / 8);
375.         uint32_t frame_len_octets = (v->frame_bs->end - v->frame_bs-
>start) / 8;
376.         memset(data, 0, frame_len_octets * sizeof(uint8_t));
377.         if(bitstream_read_lsbfirst(v-
>frame_bs, data, frame_len_octets, 8) < 0) {
378.             debug_print("Frame %d: bitstream_read_lsbfirst failed\n", fra
me_cnt);
379.             statsd_increment(v->freq, "decoder.errors.bitstream");
380.             goto cleanup;
381.         }
382.         statsd_increment(v->freq, "decoder.msg.good");
383.         enqueue_frame(v, frame_cnt, data, frame_len_octets);
384.         frame_cnt++;
385.         if(ret == 0) break; // this was the last frame in this burst

```

```
386.     }
387.     if(ret < 0) {
388.         statsd_increment(v->freq, "decoder.errors.unstuff");
389.         goto cleanup;
390.     }
391.     statsd_timing_delta(v->freq, "decoder.msg.processing_time", &v-
>tstart);
392. cleanup:
393.     XFREE(data);
394.     XFREE(fec);
395.     v->decoder_state = DEC_IDLE;
396.     debug_print("%s", "DEC_IDLE\n");
397.     return;
398. case DEC_IDLE:
399.     return;
400. }
401. }
```


Appendix C: Python modules for GBAS decoding and use

GBAS FIFO reading

This python module handles the collection of GBAS messages from the FIFO buffer, as well as some initial decoding. CRC32 is verified and the GBAS station name is checked against a whitelist. The module is based on the threading class, so that the fetching and decoding of data is performed in a separate thread. The most recent values received of each kind are stored in the module, so that they may be used for corrections.

```
from crc32 import checkcrcGBAS as checkcrc
import InternationalAlphabet as ia
import os
import errno
import threading
import numpy as np

class gbas(threading.Thread):
    def __init__(self, fifoname):
        super().__init__()
        self.dorun = True
        self.FIFO = fifoname #self.FIFO = '/tmp/gbasfifo'
        try:
            os.mkfifo(self.FIFO)
        except OSError as oe:
            if oe.errno != errno.EEXIST:
                raise

        self.gbas_stations = ["ENGM"]
        self.gbas_messages = [1,2,4,11]

        #Message 1
        self.prc_100 = np.full(256,np.NaN)
        self.rrc_100 = np.full(256,np.NaN)
        self.zcount_100 = np.NaN

        #Message 2
        self.N_R = np.NaN
        self.h_0 = np.NaN
        self.station_height = np.NaN

        #Message 4

        #Message 11
        self.prc_30 = np.full(256,np.NaN)
        self.rrc_30 = np.full(256,np.NaN)
        self.zcount_30 = np.NaN

    def __enter__(self):
        self.start()
        return self

    def __exit__(self, *args):
        self.dorun = False

    def run(self):
        while self.dorun:
```

```

#Open FIFO
with open(self.FIFO, "rb") as fifo:
    while True:
        data = fifo.read()
        if len(data) == 0:
            #EOF
            break
        self.parse(data)

def parse(self, rawmsg):

    if not checkcrc(rawmsg):
        return -1, "CRC failed"
    station_name = ia.gbasid(message[1:4])
    if self.gbas_stations and station_name not in
self.gbas_stations:
        return -1, "Unknown station", station_name
    msg_type = message[4]
    if msg_type not in self.gbas_messages:
        return -1, "Unknown message", msg_type

    if msg_type == 1:
        #self.zcount_100 =
        N = rawmsg[8] & 0x1F
        for i in range(N):
            #Collect PRC and RRC from each message block
            pass

    if msg_type == 2:
        self.N_R = (rawmsg[10] * 3) + 16
        self.h_0 = rawmsg[11] * 100
        self.station_height = (rawmsg[23]<<16 + rawmsg[22]<<8 +
rawmsg[21])*0.01

    if msg_type == 4:
        pass

    if msg_type == 11:
        #self.zcount_30 =
        N = rawmsg[8] & 0x1F
        for i in range(N):
            #Collect PRC_30 and RRC_30 from each message block
            pass

```

International Alphabet 5

The GBAS station name is encoded as four 6-bit characters from the international alphabet nr.5, placed in 3 bytes. The function *gbasid* takes in the 3-byte array and returns the decoded station name as a string.

Waypoint names from the MT4 use 5-bit characters. Via the function *waypointname*, these can also be decoded.

```
IA5 = {
    0x00: '', #empty, not used for GBAS message field
    0x01: 'A',
    0x02: 'B',
    0x03: 'C',
    0x04: 'D',
    0x05: 'E',
    0x06: 'F',
    0x07: 'G',
    0x08: 'H',
    0x09: 'I', #not used for Route Indicator field
    0x0a: 'J',
    0x0b: 'K',
    0x0c: 'L',
    0x0d: 'M',
    0x0e: 'N',
    0x0f: 'O', #not used for Route Indicator field
    0x10: 'P',
    0x11: 'Q',
    0x12: 'R',
    0x13: 'S',
    0x14: 'T',
    0x15: 'U',
    0x16: 'V',
    0x17: 'W',
    0x18: 'X',
    0x19: 'Y',
    0x1a: 'Z',
    #not used for GBAS message field
    0x20: ' ', #space
    #not used for GBAS message field
    0x30: '0', #not used for Route Indicator field
    0x31: '1', #not used for Route Indicator field
    0x32: '2', #not used for Route Indicator field
    0x33: '3', #not used for Route Indicator field
    0x34: '4', #not used for Route Indicator field
    0x35: '5', #not used for Route Indicator field
    0x36: '6', #not used for Route Indicator field
    0x37: '7', #not used for Route Indicator field
    0x38: '8', #not used for Route Indicator field
    0x39: '9', #not used for Route Indicator field
    #not used for GBAS message field
}
```

```
def gbasid(barr):
    b = 0
    gid = '' #GBAS ID
    for i in range(3):
        b = b | (barr[i]<<i*8)
    for i in range(4):
        try:
```

```

        gid = IA5[(b >> i*6)& 0x3f] + gid
    except KeyError:
        pass
    return gid

def waypointname(barr):
    b = 0
    wpn = '' #waypoint name
    for i in range(4):
        b = (b << 8) | barr[i]
    b = (b >> 2) & 0x3fffffff #Tim to 30 bit
    for i in range(6):
        try:
            wpn = IA5[(b >> i*5)& 0x1f] + wpn
        except KeyError:
            pass
    return wpn

```

Smoothing filter

The carrier phase smoothing filter is created by supplying filter length in seconds, and sample frequency in hertz:

```

filter100 = npSmoothingFilter(100, 4)
filter30  = npSmoothingFilter(30, 4)

filteredpsr100 = filter100.propagateFilter(psr, phi, rst)
filteredpsr30  = filter30.propagateFilter(psr, phi, rst)

```

The filters are propagated each sample by supplying Numpy arrays for pseudorange, phase and whether the filter must be reset (this allows the filters to be reset externally if an error is detected). The function call returns the current filtered values and whether the filters have reached steady state.

```

import numpy as np

class npSmoothingFilter:
    def __init__(self, filterlenght, samplefreq):

        self.alpha = 1/(filterlenght * samplefreq)
        self.wavelen = 299792458 / 157542000 #c / F_L1 =~ 0.190m

        self.p = np.full(32,np.NaN)
        self.prev_phi = np.full(32,np.NaN)
        self.samplecount = np.full(32,0)

    def propagateFilter(self, psr, phi, rst):

        self.p[rst>0] = np.NaN #Reset filter if rst is set
        self.samplecount[rst>0] = 0

        p_proj = self.p + self.wavelen * (phi - self.prev_phi)
        np.nan_to_num(p_proj,False) #Replace NaN with 0

```

```

        alpha_weight = np.isnan(self.p).astype(float) #New sats
weighted 1
        alpha_weight[alpha_weight == 0] = self.alpha #Existing sats
weighted alpha

        self.p = alpha_weight * psr + (1-alpha_weight) * p_proj
        self.prev_phi = phi

        sample_mask = np.isfinite(psr).astype(int)
        self.samplecount = (self.samplecount * sample_mask) +
sample_mask

        #Returns filtered pseudorange and if filter is in steady
state
        return self.p, self.samplecount > 1/self.alpha

```

UBlox driver

In order to test the smoothing filter, a simple serial driver for the u-blox GPS was needed. The implementation shown here is based on the threading class, such that the serial port is read by its own dedicated thread. A queue object is used to output received raw messages to a decoder application. The module can be used with the *with* keyword:

```
with ublox.ublox(gpsqueue, "COM25", 57600) as gps:
```

This is the recommended use of the module, as it makes sure the serial port is correctly closed before the program ends.

```

import serial
import threading
import struct
import time

class ublox(threading.Thread):
    def __init__(self, queue, portname, baud):
        super().__init__()

        self.sync = b'\xb5\x62'
        self.queue = queue #the received data is put in a queue
        self.buffer = bytearray()
        self.state = 0 #0=search for sync, 1=collect header, 2=
collect data+crc
        self.lenght = 0

        self.dorun = True
        self.ser = serial.Serial(port = portname, baudrate=baud)
        #time.sleep(0.1) #Serial returns before port is fully started

    def __enter__(self):
        self.start()
        return self

    def __exit__(self, *args):
        self.stop()

```

```

def run(self):
    while self.dorun:
        self.buffer += self.ser.read(self.ser.inWaiting() or 1)
#read all char in buffer

        if self.state == 0 and self.sync in self.buffer:
            self.state = 1
            self.buffer =
self.buffer[self.buffer.find(self.sync):] #Discard up to sync

            if self.state == 1 and len(self.buffer)>=6: #Entire
header collected
                self.state = 2
                payloadlength = struct.unpack_from('<H', self.buffer,
4)[0]
                self.length = payloadlength + 6 + 2 #header +
checksum

                if self.state == 2 and len(self.buffer)>=self.length:
#Entire message collected
                    self.state = 0
                    msg, self.buffer = self.buffer[:self.length],
self.buffer[self.length:]

                    if(self.checksum(msg[2:-2]) == msg[-2:]): #Checksum
OK
                        self.queue.put(msg[2:-2])
                    else:
                        pass #Message discarded

def stop(self):
    self.dorun = False
    time.sleep(0.1)
    self.ser.close()
    super().join()

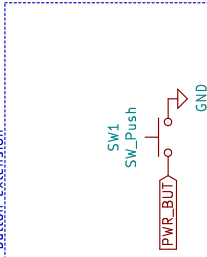
def checksum(self, buffer): #buffer containing all bytes over
which checksum is calculated
    ck_a = 0
    ck_b = 0
    for b in buffer:
        ck_a = (ck_a + b) & 0xff
        ck_b = (ck_b + ck_a) & 0xff
    return bytearray([ck_a, ck_b])

def send(self, data):
    self.ser.write(self.sync + data + self.checksum(data))

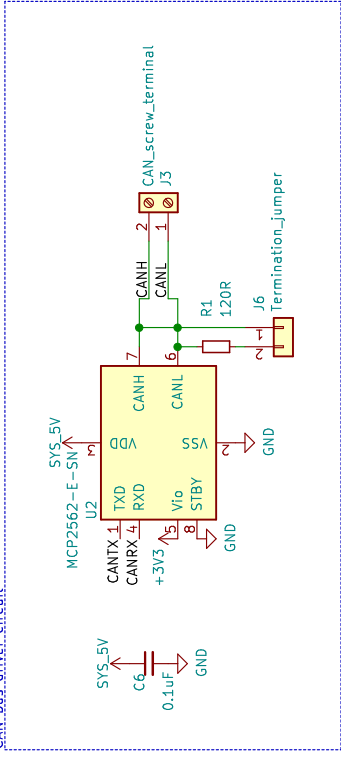
```

Appendix D: Schematics and design files for BeagleBone RS232 cape

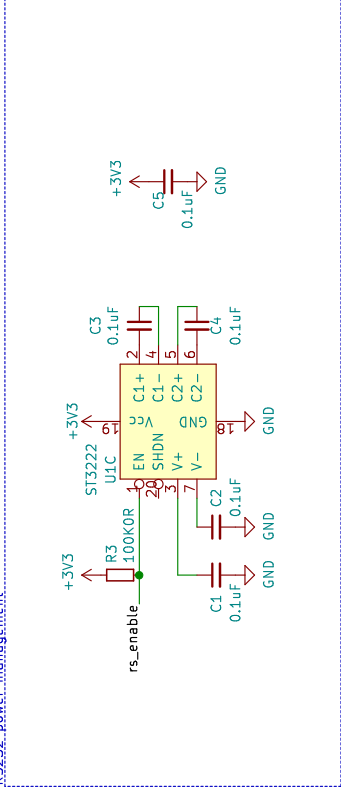
Power_button_extension



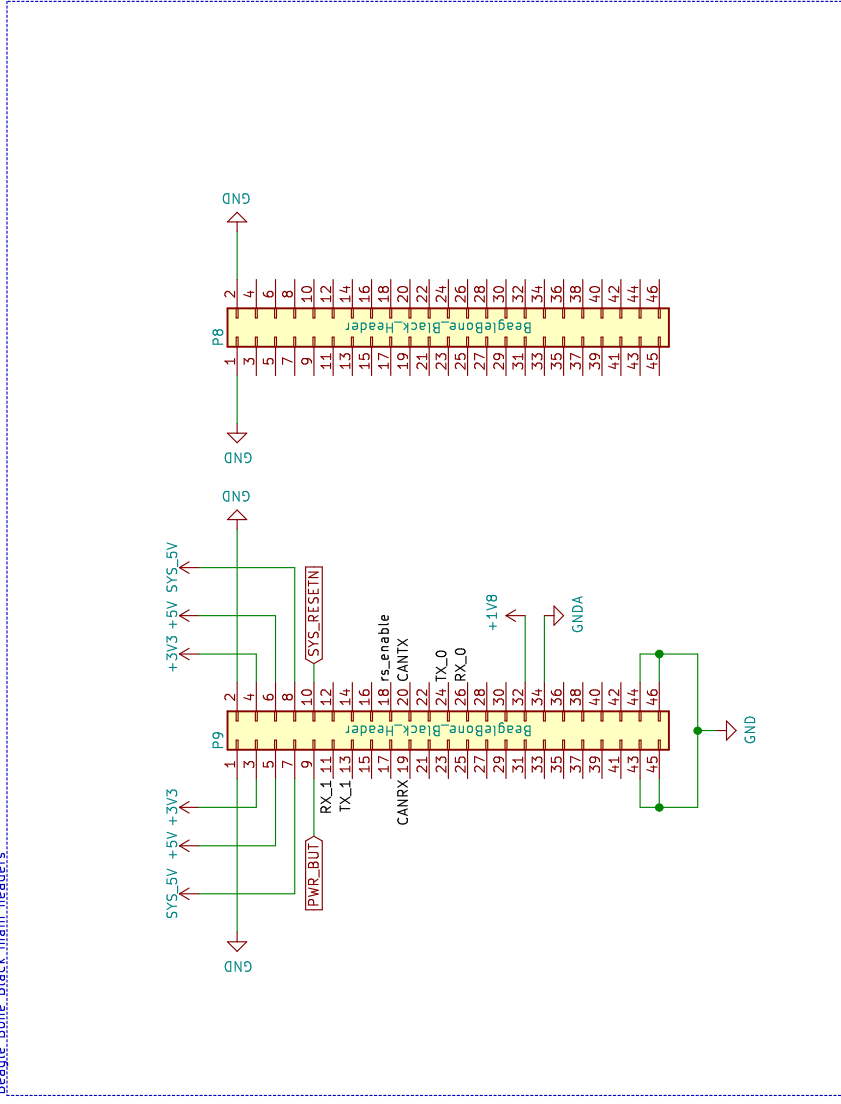
CAN_bus_driver_circuit



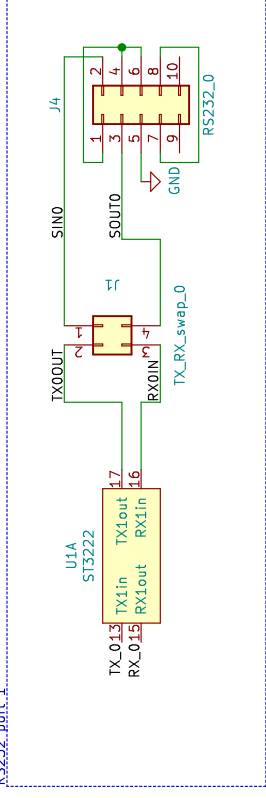
RS232_power_management



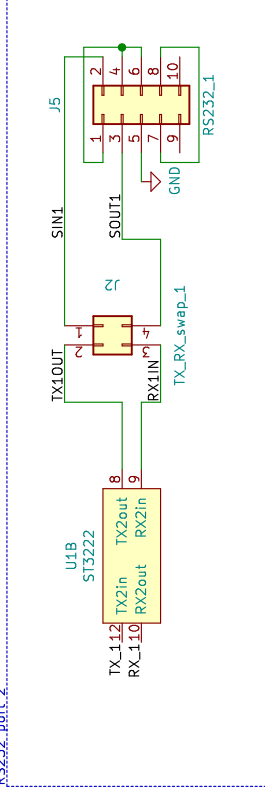
Beagle_Bone_Black_main_headers



RS232_port_1



RS232_port_2



Sheet: /
File: RS232_CAN_cape.sch

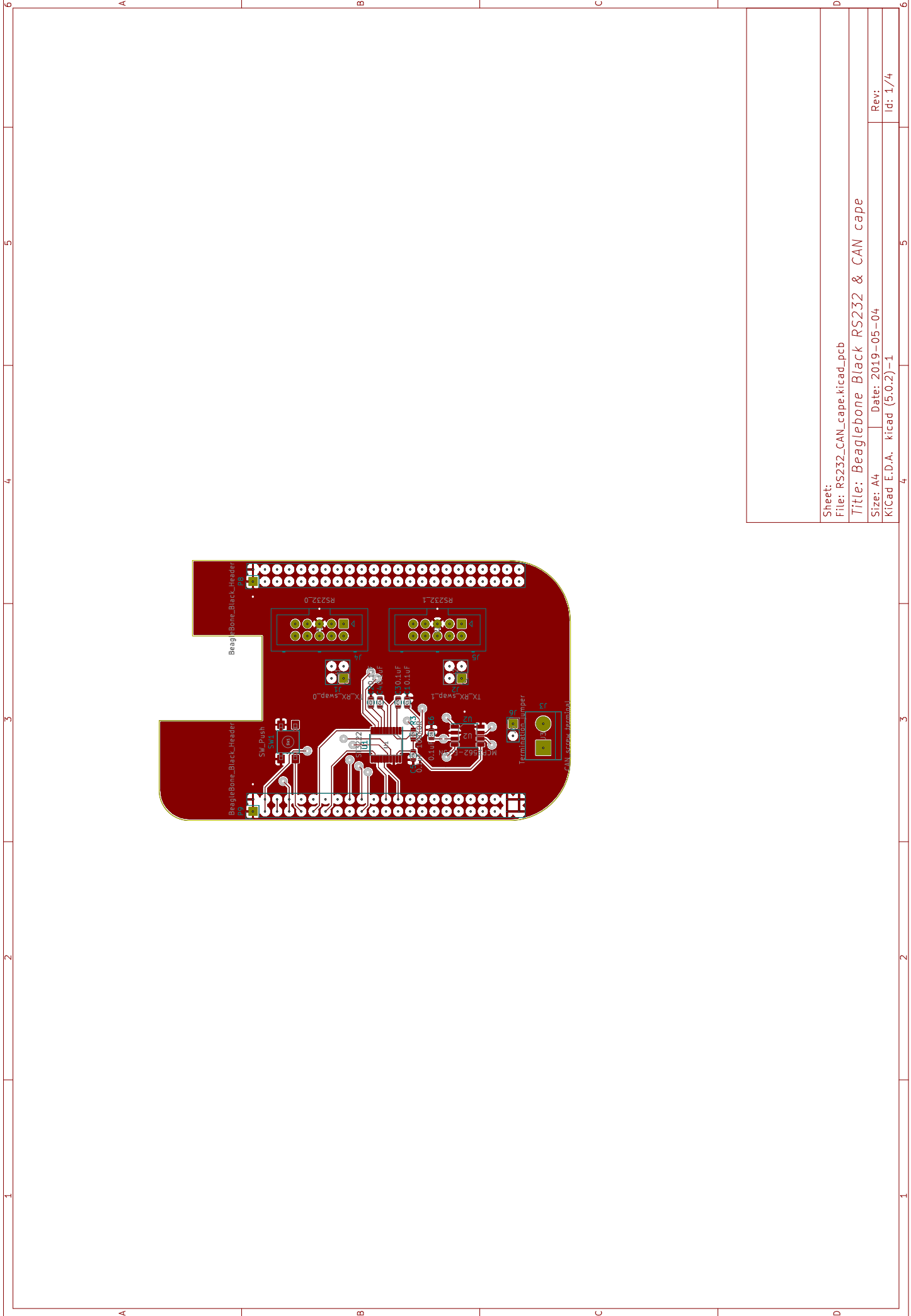
Title: Beaglebone Black RS232 & CAN cape

Size: A4 | Date: 2019-05-03

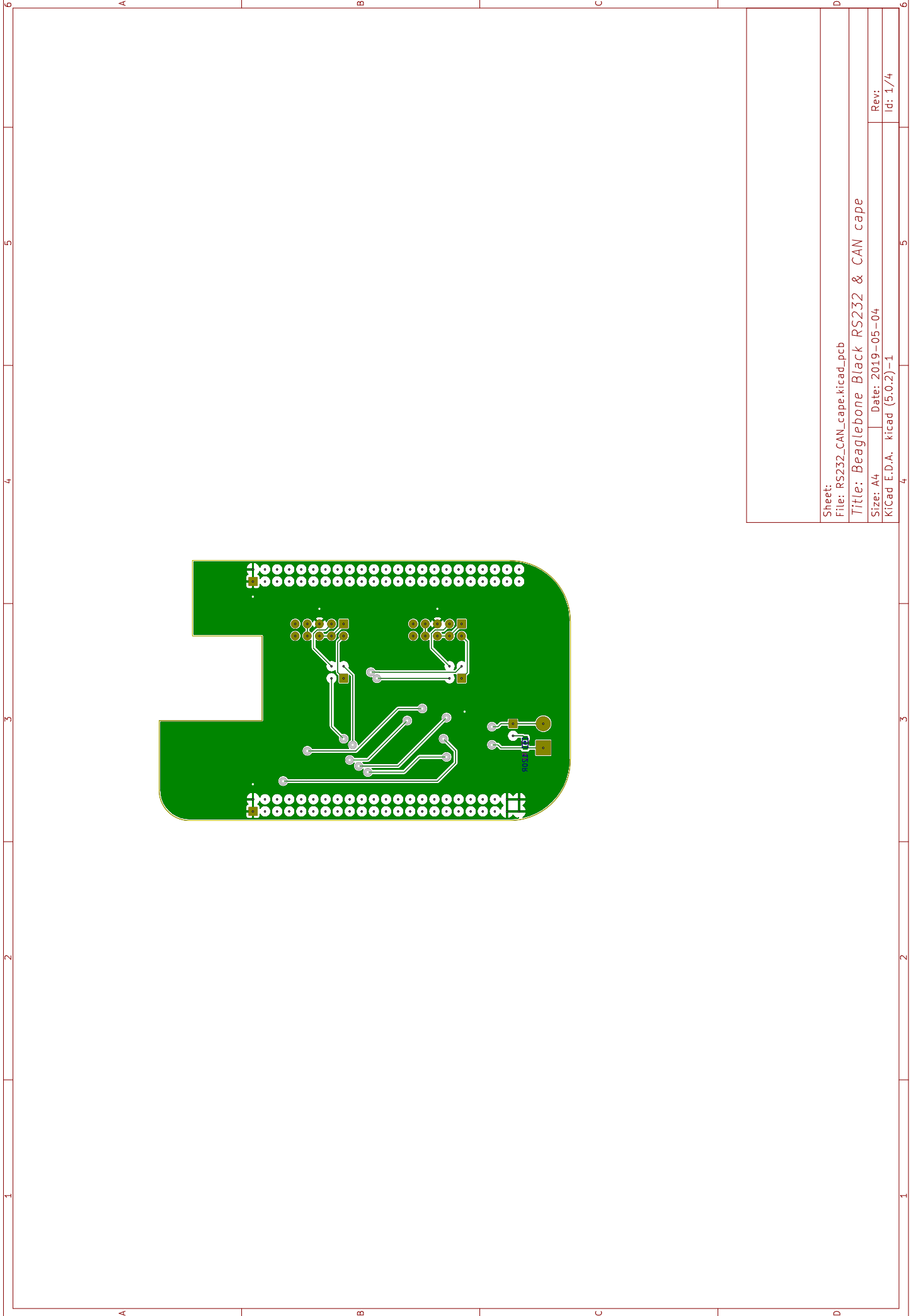
KiCad E.D.A. kicad (5.0.2)-1

Rev: 1/1

Id: 1/1



Sheet:		File: RS232_CAN_cape.kicad_pcb	
Size: A4		Date: 2019-05-04	
KICad: E.D.A. kicad (5:0:2)-1		Rev: 1/4	
Id: 1/4		Title: Beaglebone Black RS232 & CAN cape	



Sheet:		File: RS232_CAN_cape.kicad_pcb	
Title: Beaglebone Black RS232 & CAN cape			
Size: A4	Date: 2019-05-04	Rev:	
KiCad: E.D.A.	kicad (5:0:2)-1	Id: 1/4	

