TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technology

Maksim Maljutin
181931IVEM

Radiosonde and ground-station system for testing GNSS modules and TTÜ100 satellite sun sensor at high altitude

Master thesis

Supervisor: Rauno Gordon, Ph.D

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Maksim Maljutin
181931IVEM

Raadiosondi ja maajaama süsteem GNSS moodulite ja TTÜ100 satelliidi päikesesensori testimiseks suurtel kõrgustel

magistritöö

Juhendaja: Rauno Gordon, Ph.D

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Maksim Maljutin

12.05.2018

Abstract

This thesis is the final work for fulfilment of the Master of Science degree at the Thomas Johann Seebeck Department of electronics in Tallinn University of Technology.

The goal of this thesis is to create the full platform for testing GNSS modules from different producers and a sun sensor for TTÜ100 satellite at high altitude using weather balloons. The platform consists of balloon payload unit, ground-station module and UI for configuration and data output. The created payload unit works together with the ground-station module (GSM and LoRa communication) or can work as standalone with GSM communication only.

The system was tested with a weather balloon that flew to 6km altitude and to 15 km distance. Communication with LoRa lasted until the highest point reached during the flight. Communication with GSM lasted for 15 min after the launch and was available again for some time after the landing. The 3 GNSS modules onboard provided coordinates during the whole flight with exception for fall period.

This thesis is written in english language and is 65 pages long, including 35 figures and 4 tables.

Annotatsioon

Antud uurimistöö on esitatud magistrikraadi saamiseks Thomas Johann Seebeck'i elektroonikainstituudi juures Tallinna Tehnikaülikoolis.

Käesoleva uurimistöö eesmärk on platvormi loomine erinevate tootjate loodud GNSS moodulite ja TTÜ100 sateliidi päikese-sensori testimiseks suurtel kõrgustel. Platvorm koosneb lennu-moodulist, maajaama moodulist ja maajaama mooduli kasutajaliidesest. Kasutajaliidest kasutatakse süsteemi konfigureerimiseks ja andmete kuvamiseks. Loodud lennu-moodul töötab üheskoos maajaama mooduliga (GSM ja LoRa ühendus) või siis iseseisvalt ainult GSM'i ühendusega.

Antud süsteem sai testitud kasutades meterooloogilist õhupalli, mis lendas 6 km kõrgusele ja 15 km kaugusele. Ühendus LoRa'ga kestis kuni lennu maksimum-kõrguseni. Ühendus GSM'iga kestis 15 minutit pärast süsteemi käivitamist ning oli taas saadaval mõne aja jooksul pärast maandumist. Pardal olevad 3 GNSS moodulit salvestasid koordinaate kogu eksperimendi jooksul, välja arvatud kukkumise faasis.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 65 leheküljel, 35 joonist ja 4 tabelit.

List of abbreviations and terms

GNSS	Global Navigation Satellite System
GPS	Global Positioning System (by USA)
GLONASS	Global Navigation Satellite System (by Russian Federation)
UI	User interface
PCB	Printed circuit board
LoRa	Long range, low power wireless platform
IDE	Integrated development environment
GSM	Global System for Mobile Communication
SMS	Short Message Service
GPIO	General Purpose Input Output
NMEA	The National Marine Electronics Association
HAT	Add-on board for Raspberry PI 40 pin GPIO
SNR	Signal to Noise Ratio
RSSI	Received Signal Strength Indicator
PITS	Pi In The Sky (the Raspberry Pi telemetry board)
UART	Universal Asynchronous Receiver-Transmitter
Wi-Fi	Technology for wireless local area networking

Table of Contents

Table of Figures

List of tables

1 Introduction

My master thesis is about building a high altitude balloon infrastructure. It covers hardware design, hardware integration, PCB layout, and software for the balloon flight-module and ground-station. The last will consist of two parts ground-station itself and separate UI made for android tablets and PC. The motivation of the high altitude experiment system is testing the behavior of different GNSS devices and the TTÜ100 satellite sun sensor. On the flight-module will also be other devices like accelerometers, gyroscopes, pressure sensor, temperature sensor and communication modules.

1.1 Goals

The goal of this thesis is to create the full platform for testing GNSS modules from different producers, a sun sensor for TTÜ100 satellite and possibly other systems at high altitude using weather balloons. GNSS modules can be easily replaced with modules from other producers, so after the series of experiments it is planned to select the ones that will work at high altitudes over 20km. Besides positioning, test platform gives possibility to demonstrate that an affordable optical sensor regularly used in computer mouse can be successfully used for sensing satellite orientation towards the sun.

1.2 GNSS Accuracy and Reliability

Large part of our modern infrastructure relies on GNSS positioning. Affordable GNSS sensors are used everywhere – in small and cheap devices as well as in high risk applications. The GNSS system in general and the actual modules produces have limitations. One of the motivations of the high altitude testing system is to test various modules and determine their capabilities in high altitude and high speed applications.

1.2.1 Vertical and Horizontal accuracy

In balloon experiments both vertical and horizontal positions do make sense. As winds at different altitudes may differ too much it will affect the flight route. There is information about how accurate GNSS can be. According to [1] vertical and horizontal accuracy at 95% level of confidence is ca 4.7m and 3.3m.

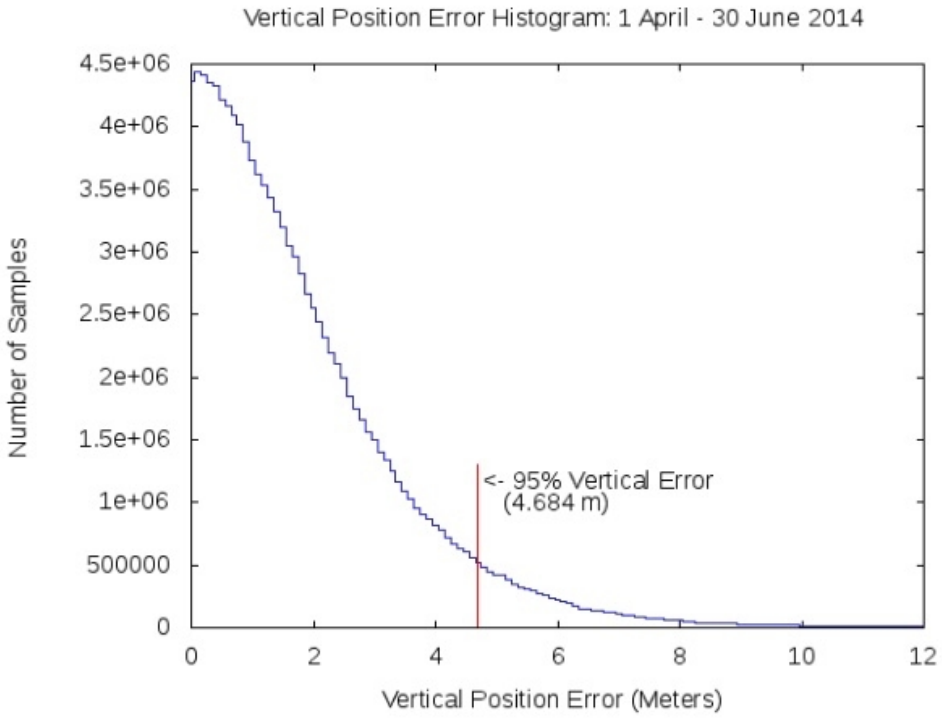

Figure 1: Vertical Position Error

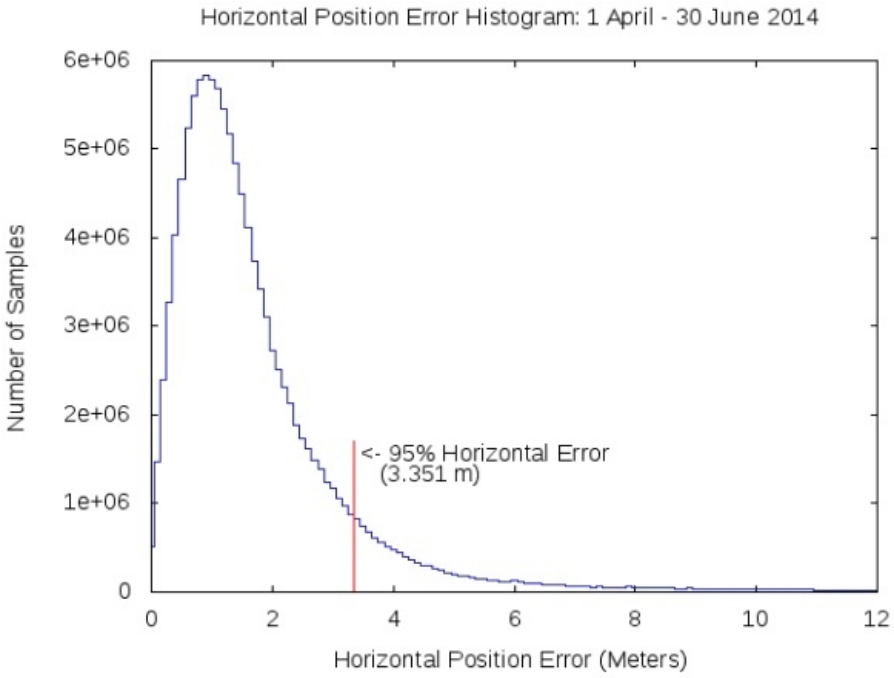

Figure 2: Horizontal Position Error

Those plots show, that even static object will have different data at each sample. There are some solutions to increase accuracy of GPS. For example, Differential Global Positioning System (DGPS). It uses well known positions to eliminate pseudo-range errors in real time signal. But it is important to note that it fixes position data, not the object speed. Sending three devices at the same time will give us a good possibility to compare accuracy at different altitudes and speeds. Even more, GNSS device altitude will be compared to altitude got from air-pressure sensor

1.2.2 Military vs Civilian GPS

The accuracy of the GPS signal in space is actually the same for both the civilian GPS service (SPS) and the military GPS service (PPS). However, SPS broadcasts on one frequency, while PPS uses two. This means military users can perform ionospheric correction, a technique that reduces radio degradation caused by the Earth's atmosphere. [1]

1.2.3 Reasons for accurate GNSS

Civilian GPS service has typical accuracy of about 3.5 m [1]. While it seems that 3.5 meters (or even 10 meters) accuracy is enough for some navigation purposes, it

might not be acceptable for autonomous vehicle navigation or scientific needs. For example, weather forecast system needs to have accurate wind speed, direction, temperate, humidity and air pressure data at different altitudes.

1.2.4 Limitation to commercial GPS

There are regulation regarding GPS technology. So named the "COCOM Limits" placed on GPS tracking devices [32]. According to those limits device must disable tracking possibilities when it is moving faster than 1900 km/h at an altitude at least 18000 m. Main idea of the limits is to prevent the use of GPS in missile-like applications.

At this point different manufactures do have their own vision about such limitations. Some of them implement AND while others implement OR logic. Therefore GNSS modules of some manufacturer will work at high altitudes, other will not.

1.2.5 Comparison example of GPS modules

There are some successful comparison tests in the web. For example, in [3] was examined six different modules. Experiment was held on ground level at walking speed.

Figure 3: Example of GPS comparison [3]

As one can see although all GNSS sensors show almost the same position it is still different for each device.

In practical part of my thesis I will try to reproduce similar experiments but with hardware modification and in different conditions. Idea is to test modules at different speed, altitude, air pressure and temperature. Accuracy from [1] may not be the same at altitudes. Therefore it is interesting to compare data from multiple devices.

2 State of the art – systems for testing at high altitude

Some systems exist that are created for hobby high altitude ballooning or can be useful for such experiments. They do not have all functions necessary for our 2 test-scenarios – comparison of different GNSS modules and the TTÜ100 satellite sun sensor.

2.1 ARHAB program

Amateur radio high-altitude ballooning is the application of amateur radio to weather balloon. It is a quite popular hobby for radio amateurs since the first flight that took place at 15 August 1987. Typical flight experiment consists of a balloon, descending parachute and some payload. In addition to the tracking systems payload may have different sensor, radio transmitters, cameras etc. Usually flight time is few hours with maximum reached altitude 25-35km. [36]

2.2 Project Loon

Project by X (formerly Google X) with the mission of providing high speed wireless networking to rural areas[34]. It uses high-altitude balloon at altitudes 18-25km. Each balloon is made from polyethylene with size about 250m^{2}[35]. Estimated flight time of such balloon is 100 days. The whole flight is controlled since launching till controlled descent. According to balloon position can be achieved by using only balloon's altitude. It has solar panels for daytime needs and rechargeable batteries for night work [35].

2.3 Pi in the sky

Pi-in-the-Sky (PITS) is a working solution for tracking high altitude balloon position [6]. It is a Raspberry shield that can be easily configured for long distance communication at the amateur frequency band of 430 MHz. A separate software-defined-radio USB device has to be used for ground-station. PITS is using single GPS receiver that is tested to work at very high altitudes.

Figure 4: PITS+ Board for Raspberry A+, B+, V2 B From [6]

Figure 5: PITS Zero with Pi Zero. From [6]

The system has open source software (can be found at github.com) and has a possibility of adding extra I2C devices like air pressure sensor. Analog input is a big extension to the Raspberry Pi, which does not have it at all. But there is still the limitation of only one UART device which is already occupied by the MTX2 radio transmitter of the PITS. PITS has one-directional radio link which in some cases may not be enough. As there are no more UART ports it is impossible to make comparison of multiple GNSS devices (most of them are using serial port for data transfer). Yet another problem is absence of GSM module, which is very helpful while searching the system after landing. Currently additional GPS tracker is used for such purposes.

Pi in the sky is a popular system – the manufacturers claim over 70 high altitude flights have been made. But as almost all other Raspberry Pi expansion boards - it is quite expensive and does not provide 100% of the functionality we want from a high altitude testing system.

2.4 SPOT GEN3

Figure 6: Spot gen3. From [31]

This is an example of a GPS tracker that can be used together with the PITS or ARHAB system. The main disadvantage – it provides only positioning service. So, it should be used as helper device for some other system after the landing. A big bonus is messaging to communication satellites, so it will work even if no cellular network available. Another problem is weight. Additional 115g of weight adds costs for the whole system if very high altitudes are needed to reach (bigger balloon, more helium). In our case very high altitudes are needed for sun sensor tests but this device is proved for altitudes up to 6500m. The price of this device is about 150 eur.

There are number of other GPS tracker devices available on the market. For example, [33] gives a good comparison of GPS trackers. But all of have the same two disadvantages additional weight and limited functionality for positioning service only.

3 System description

All stages for the system were made from scratch, including hardware design, implementation and software development.

3.1 System overview

The system may be used as standalone module – just a weather balloon payload with logging system and (optionally) geofence configuration. However, it is much more convenient to use it together with ground-station module. In such scenario it is possible to track the payload unit and get data from sensors in real time.

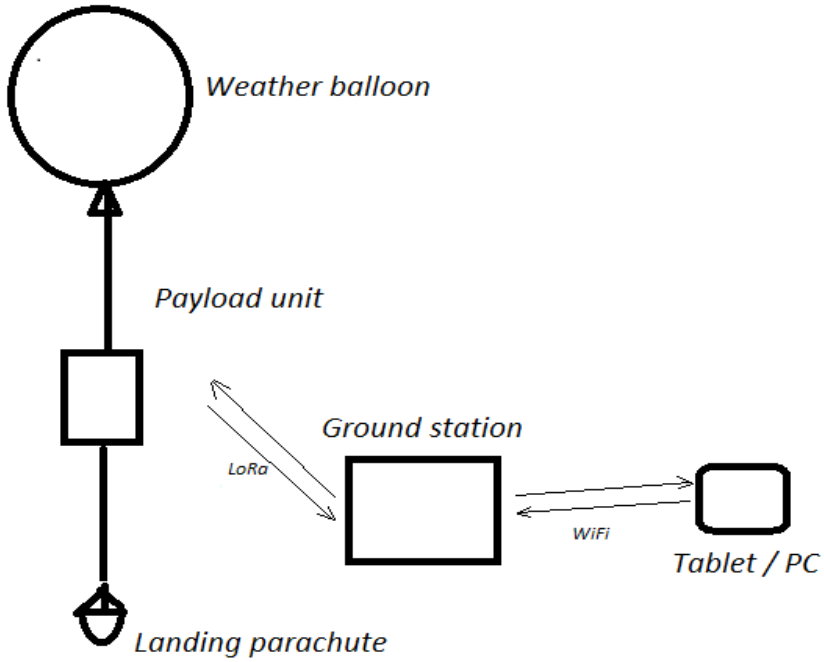

Figure 7: Whole system overview

Both balloon payload unit and ground-station are based on Raspberry Pi. The first one uses Raspberry Pi Zero (during development Zero W was used). The second is done on Raspberry Pi 3. Software for both modules is written in C/C++ language using Eclipse Mars IDE[10] and gnutoolchain cross-compiler[11]. Ground-station UI is

written on QML and C++ on QT framework [8]. It can be executed either on Android tablet or PC.

All created PCB-s were designed and traced using free version of Autodesk Eagle 8.3.2 . It is limited to 2 schematic sheets, 2 signal layers, and 80 cm2 board area.

3.2 Payload unit

Payload unit is the main part of whole system. In some scenarios it can be even used without other parts. The payload consists of control unit, sensors and communication modules.

3.2.1 PCB

Initial version of the payload unit PCB was done on prototype board. The final is double sided PCB 100x80mm and 1.6mm thick. Design and layout done in Autodesk Eagle 8.3.2 software[12]. Free version is limited to 2 schematic sheets, 2 signal layers, and 80 cm2 board area.

Raspberry Pi Zero, multiplexer, GSM, MPU and LoRa modules are mounted directly to the board. All other modules: 3x GNNS, Sun sensor, burner, beeper, led lightning, camera and power module are attachable via connectors.

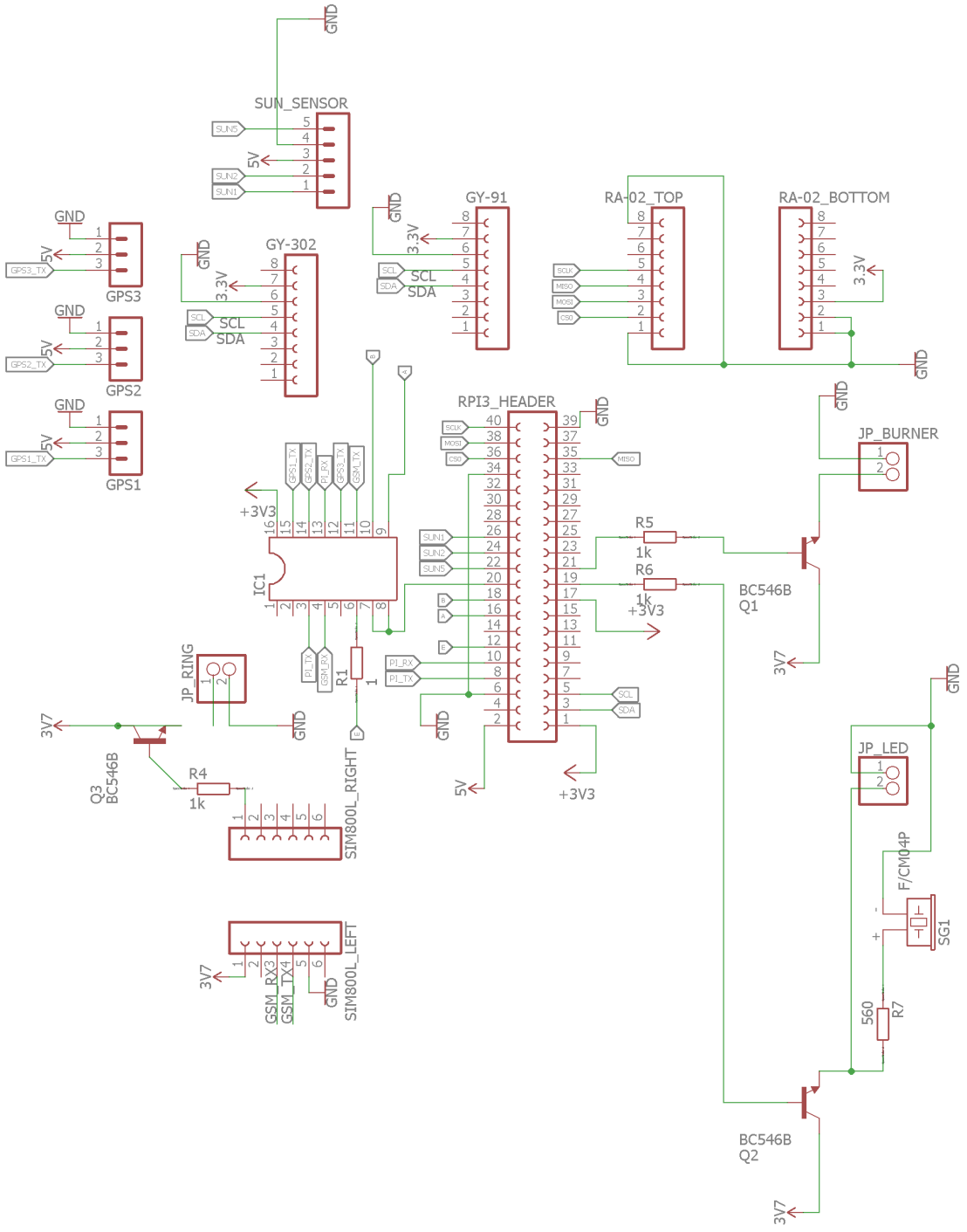

Figure 8: Payload unit schematics

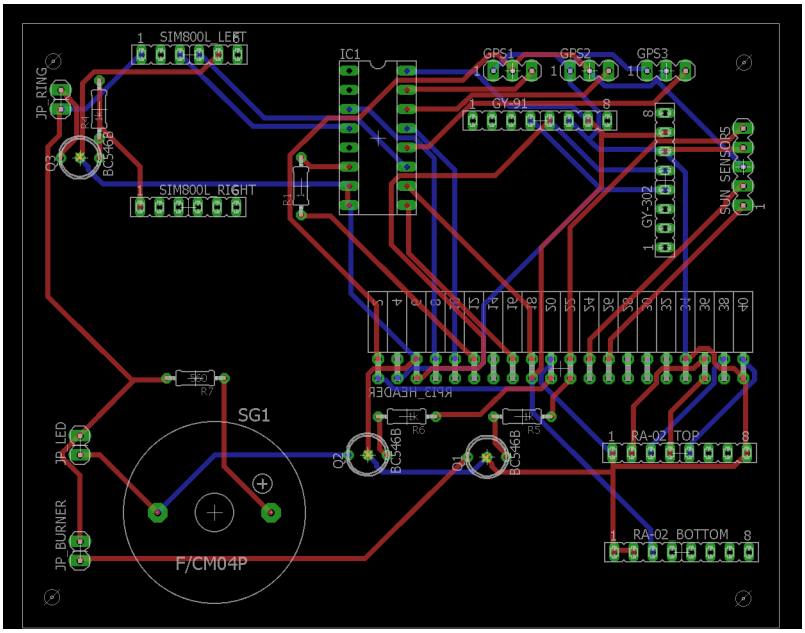

Figure 10: Payload unit PCB layout

PCB realization has been ordered from www.pcbway.com.

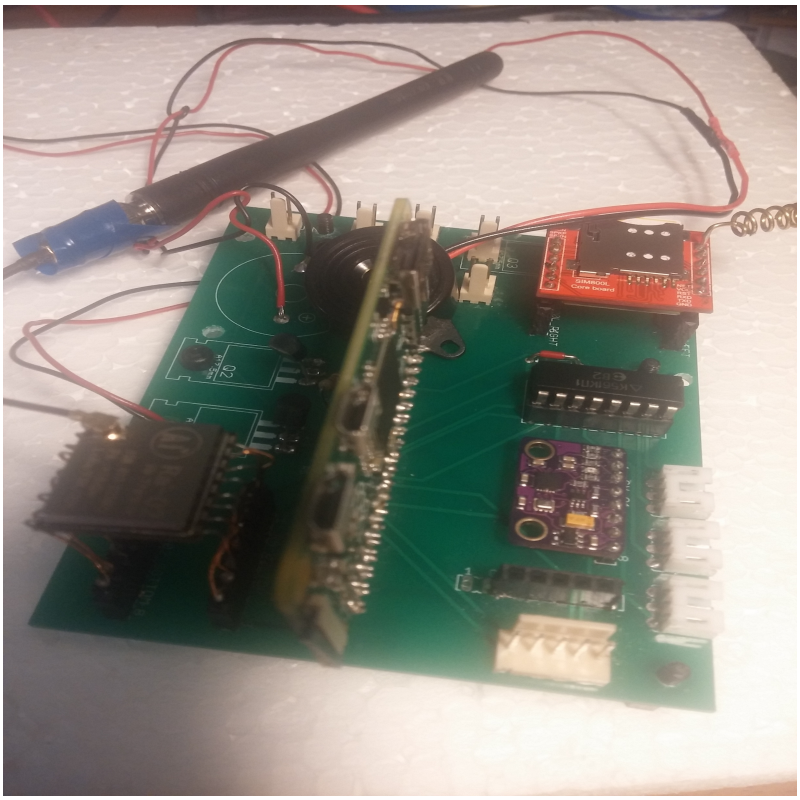

Figure 9: Payload unit board overview

3.2.2 Raspberry Pi Zero

3.2.2.1 Overview

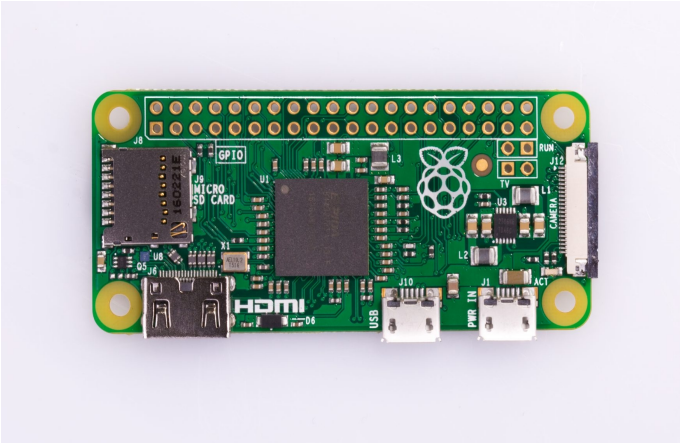

Figure 11: Raspberry Pi Zero

The brain of the Paylod unit is Raspberry Pi Zero. It is the smallest and the cheapest module from Raspberry Pi family. During development Raspberry Pi Zero W (similar product, but with WiFi on-board) was used. The main benefit of the module is its size and Linux based Raspbian OS. Thanks to the OS, software development is much easier comparing to bare metal programming.

3.2.2.2 Environment setup

1. Download Eclipse IDE to PC and extract it.
2. Setup toolchain according to the tutorial from [11].
3. In Eclipse File->New->C++ project. For cross-compiling Cross GCC compiler must be selected from toolchains list.
4. Raspberry. In terminal type: sudo raspi-config. Ensure that camera, ssh, spi, i2c and serial are enabled
5. Create folder where binaries will be stored.

3.2.2.3 Software

Payload unit software is written in C/C++. Each of the sensors has separate class. Entry point is main function inside main.cpp. Here are created instances for all classes, describing devices functionality and optionally registered to their callbacks. All work is done inside infinite loop.

Code from main.cpp:

```
GSM* pGSM;
Multiplexer* pMultiplexer;
...
void GSMCallback(int gsmRequest)
{
	switch (gsmRequest)
	{
	case GSM::GSM_REQUEST_POS:
		SendPosGSM();
		break;

	case GSM::GSM_REQUEST_BURN:
		Burn();
		break;

	case GSM::GSM_REQUEST_BEEP:
		pBeeper->Beep();
		break;

	default:
		break;
	}

}
...
void SendPosGSM()
{
	std::string strBase = "http://maps.google.com/maps?q=";
	std::stringstream ssPos;
	for (size_t idx = 0; idx < Multiplexer::GPS_COUNT; idx++)
	{
		ssPos.clear();
		ssPos << strBase;
		GPS* pGPS = pMultiplexer->GetGPS(idx);
		if (pGPS != NULL)
		{
			Coordinate c = pGPS->GetPos();
			double latGr = ((int)c.lat / 100);
			double latMi = (int)(1000 * (c.lat - latGr * 100) / 6);

			double lonGr = ((int)c.lon / 100);
			double lonMi = (int)(1000 * (c.lon - lonGr *100) / 6);

			ssPos << latGr << "." << latMi << ",";
			ssPos << lonGr << "." << lonMi << "\n";
		}
	}
	std::string strMessage = ssPos.str();
	std::cout << "GPS Data to sent: " << strMessage << std::endl;

	std::vector<std::string> vec;
	vec.push_back("SMS");
```

```
        vec.push_back(pGSM->GetRecipient());
        vec.push_back(strMessage);
        pGSM->AddCommand(vec);

}

int main()
{
...
        pGSM = pMultiplexer->GetGSM();
        pGSM->AddHandler(GSMCallback);
...
        int intervals[2] = {300, 900};
        int interval = intervals[0];

        gettimeofday(&atSMSTime, NULL);
...
        gettimeofday(&atStart, NULL);
...
        while(true)
        {
                gettimeofday(&now, NULL);

                if ((now.tv_sec - atStart.tv_sec) > 3600)
                {
                        interval = intervals[1];
                }

                if ((now.tv_sec - atSMSTime.tv_sec) > interval)
                {
                        gettimeofday(&atSMSTime, NULL);

                        SendPosGSM();
                }
...
        }
...
}
```

Here is an example how GSM module is handled inside code. First, code gets instance of GSM class created. Then registered callback function to GSM activities. When all initialization code inside main function is done infinite loop is started. Based on elapsed time some GSM activity is generated. During first hour SMS has to bee sent every 5 minutes and after every 15 minutes. This is decided based on the fact that GSM module will became unreachable due to high altitude after some time and there is no need to send SMS anymore. At the same time it will became needed again after landing to help searching the module. There are no network status checking logic so this is done using intervals.

3.2.3 UART Multiplexer

Raspberry Pi is a great platform for embedded systems, but it also has several limitations. Single hardware UART is one of them. Unfortunately, most of the GNSS sensors available on the market is using this type of communication. Besides them there

is GSM module, that is also controlled via UART. So, there is need for a solution that gives possibility to communicate with all the devices using only one hardware UART.

3.2.3.1 Module description

There are many multiplexing solutions on the market nowadays: for different channel count, single and dual direction, analog and digital, etc. Raspberry Pi uses UART for collecting data from three GNSS sensors and also communicates to GSM module. It means the system needs four channel multiplexer. While GNSS modules may be used in one-directional way, GSM module needs to have both TX and RX. So, I have selected CD4052B[14] 2-channel, 4:1 analog switch.

2-channel 4:1 analog switch means, that this chip can be switched to 1 of the 4 possibilities with 2 bidirectional channels. In my system both bidirectional channels will be used only for GSM modules. GNSS modules can be also talked to if there is a need to change some configuration parameter, but currently they are used in read only mode. All needed configuration are made before.

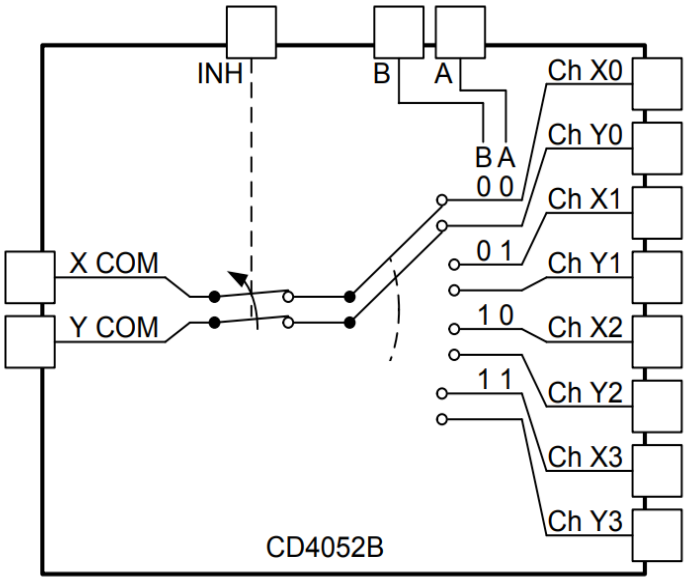

Figure 12: CD4052B functional diagram from [14]

3.2.3.2 Software

Multiplexing functionality is implemented inside Multiplexer class. It uses C library to manipulate Raspberry GPIO Broadcom BCM2835 chip. According to the schematics there are 3 control pins for the multiplexer, which are defined in code as:

```
#define E RPI_V2_GPIO_P1_12
#define A RPI_V2_GPIO_P1_16
#define B RPI_V2_GPIO_P1_18
```

Serial port will be opened only once and all 4 devices will share it. Each device has a pair combination to enable itself. Main logic of the class runs in separate thread, which has infinite loop where is implemented device selection logic.

```
void Multiplexer::OpenDeviceAB(const unsigned char a, const unsigned char b)
{
	bcm2835_gpio_write(A, a);
	bcm2835_gpio_write(B, b);
}

void Multiplexer::ThreadFuncSwitch(Multiplexer* pMultiplexer)
{
	unsigned char deviceIDX = 0;
	GSM* pGSM = pMultiplexer->GetGSM();
	SerialSensor* pDevice[GPS_COUNT + 1];

	pDevice[0] = pGSM;
	for (unsigned char idx = 0; idx < GPS_COUNT; idx++)
	{
		pDevice[idx + 1] = pMultiplexer->GetGPS(idx);
	}
	while(pMultiplexer->_started)
	{
		if(pDevice[deviceIDX]->Ready())
		{
			cout << "multiplexer before switch " << (int)deviceIDX << endl;
			switch(deviceIDX)
			{
			case 0:
				pMultiplexer->OpenDeviceAB(1, 1);
				pGSM->Pause(false);
				break;
			case 1:
				pGSM->Pause(true);
				pMultiplexer->OpenDeviceAB(0, 0);
				break;
			case 2:
				pGSM->Pause(true);
				pMultiplexer->OpenDeviceAB(1, 0);
				break;
			case 3:
				pGSM->Pause(true);
				pMultiplexer->OpenDeviceAB(0, 1);
				break;
			default:
				break;

			}
			cout << "multiplexer before process " << (int)deviceIDX << endl;
			pDevice[deviceIDX]->Process();
```

```
                cout << "multiplexer after process " << (int)deviceIDX << endl;
        }
        if (deviceIDX == 0)
        {
                usleep(10*1000*1000);
        }
        deviceIDX = (++deviceIDX) % (GPS_COUNT + 1);
        cout << "multiplexer device changed " << (int) deviceIDX << endl;

    }
}
```

3.2.4 GSM module

For additional communication purposes a GSM module is added to the system.

3.2.4.1 Module overview

Figure 13: SIM800L module from [16]

For communicating via cellular network SIM800L is used. It is one of the cheapest modules that can be found in local electronic stores. In balloon payload unit it is used for sending and receiving SMS messages but also making and receiving calls. Interaction between Raspberry Pi and GSM module is done via Multiplexer using AT commands. Specification for the commands can be found at [17]. Prior to use module is configured to communicate at 9600 baud, SMS is set to text mode.

3.2.4.2 Module functionality

GSM module functionality is implemented inside GSM class. During module setup there will be made a call to a phone-number written in configuration file. When the user receives the call to his/her mobile phone, it indicates that the GSM module is successful initialized and registered into cellular network. Main logic is done inside separate thread. When module is activated from Multiplexer class it checks if there are SMS-s. If yes, it tries to execute commands. Currently supported commands:

- GET POS

This command will notify core via callback that GNSS location SMS has to be sent

- BURN

This command will notify core via callback that rope-burning task has to be executed.

- BEEP

This command will notify core via callback that beeper has to be activated.

Module also checks if there is an incoming call. As a response to it SMS with the current GNSS position will be sent to the registered phone number.

3.2.5 GNSS

Three GNSS modules from different manufacturers are integrated into the payload unit for accuracy and reliability comparison purposes as one of the main goals of the thesis.

3.2.5.1 Modules description

These criteria were used for selecting GNSS modules:

- Affordability
- One uses only American GPS system
- One is GPS + GLONASS combination
- One module has external antenna

Figure 14: NEO6M u-blox 6 chip GPS with external antenna.

Bought from www.opood.ee

Figure 15: V.KEL VK16E SIRF III chip.

Bought from www.ebay.ie

Figure 16: GN-8013 u-blox8 chip. GPS + GLONASS version.

Bought from www.aliexpress.com

As multiplexing is used for GSM and 3xGNSS all the modules were pre-configured to operate at 9600 baud. Otherwise, during work UART port would be re-initialized each time the device is switched.

3.2.5.2 Code implementation

Each time GNSS module is activated it reads rows from serial port. Module is searching for specific NMEA commands.

GxGGA - essential fix data which provide 3D location and accuracy data.

GxGGA command example (from [18]):

*$GxGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47*

Where:
GGA Global Positioning System Fix Data
123519 Fix taken at 12:35:19 UTC
4807.038,N Latitude 48 deg 07.038' N
01131.000,E Longitude 11 deg 31.000' E
1 Fix quality:
0 = invalid
1 = GPS fix (SPS)
2 = DGPS fix
3 = PPS fix
4 = Real Time Kinematic

5 = Float RTK
6 = estimated (dead reckoning) (2.3 feature)
7 = Manual input mode
8 = Simulation mode
08 Number of satellites being tracked
0.9 Horizontal dilution of position
545.4,M Altitude, Meters, above mean sea level
46.9,M Height of geoid (mean sea level) above WGS84 ellipsoid
(empty field) time in seconds since last DGPS update
(empty field) DGPS station ID number
**47 the checksum data, always begins with **

Depending on device 'x' can be either 'P' for GPS device or 'N' for complex device.

GxGSV- Satellites in View shows data about the satellites that the unit might be able to find based on its viewing mask and almanac data.

GxGSV command example (from [18]):

*$GPGSV,2,1,08,01,40,083,46,02,17,308,41,12,07,344,39,14,22,228,45*75*

Where:
GSV Satellites in view
2 Number of sentences for full data
1 sentence 1 of 2
08 Number of satellites in view

01 Satellite PRN number
40 Elevation, degrees
083 Azimuth, degrees
46 SNR - higher is better
for up to 4 satellites per sentence
**75 the checksum data, always begins with **

The first command is used to get current position, while second is used for background monitoring to get aim about signal quality.

In case of complex device both GPGSV and GNGSV commands will present. In case of GPS only device only the first command will present.

If CRC for GGA command is correct module will calculate distance to predefined fence. Fence is a vector of points plus maximum allowed radius for the

point. Distance to the fence will be sent to the core module. If all of the GNSS are out of the fence burn command will be called.

The distance is calculated using haversine formula [19].

Distance calculation. Modification of [20]:

```
float GPS::Distance(Coordinate c1, Coordinate c2)
{

   float longitudeDelta = (c2.lon - c1.lon) * GradToRad;
   float latitudeDelta = (c2.lat - c1.lat) * GradToRad;
   float a = pow(sin(latitudeDelta / 2.0), 2) + cos(c1.lat * GradToRad) * cos(c2.lat *
GradToRad) * pow(sin(longitudeDelta / 2.0), 2);
   float c = 2 * atan2(sqrt(a), sqrt(1 - a));
   float d = 6367 * c;

   return d;
}
```

3.2.6 Multi-sensor module

Besides GPS coordinates the payload unit also needs to know and log information from additional devices: accelerometer, gyroscope, barometer, temperature.

Accelerometer

To detect free-fall. Is used to burn the rope to release the balloon and activate earlier drop down. Also used in opposite scenario, when burn command was send and system waits for successful burn.

Gyroscope

To show orientation of the payload unit. There are one additional sun-sensor module which must take pictures of the Sun. Gyroscope helps to select moments, when the Sun is opposite to the sun-sensor module. Also may be useful after landing to understand location. If payload unit has landed and is hanging from a tree – the numbers will change from time to time. If module has landed on the ground – the numbers will be constant.

Barometer

For pressure and altitude calculation. Both for collecting data and for comparing to GNSS sensors values.

Temperature

To check conditions inside payload unit. If temperature is too low the battery will be discharged faster and based on other data the burn command may be called. Temperature value is also used to correct pressure calculations.

3.2.6.1 Module selection

For such data collecting GY-91 module is used. It is so-called 10 dof module, which contains MPU9250 (MPU9255) + BMP280. MPU9250 has tri-axial accelerometer, gyroscope and magnetometer (not used in my system).

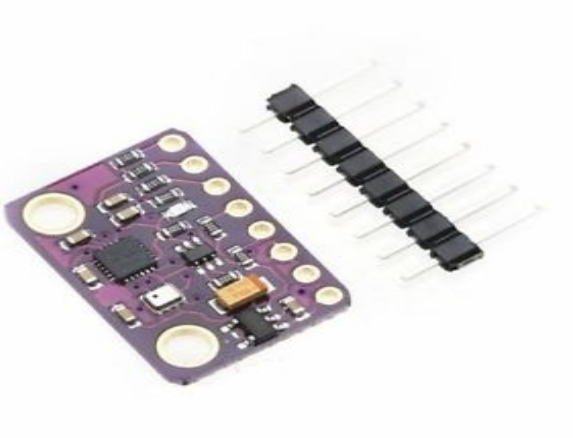

Figure 17: GY-91.

Bought from www.ebay.ie

3.2.6.2 Software implementation

Movement data.

Accelerometer and gyroscope functionality is implemented inside MPU class. The source is using library provided by InvenSense, which can be downloaded from [21]. Based on code from [22]. Code reads yaw, pitch, roll and acceleration data from sensor in separate thread and stores last 100 values in local queue. Done for future in order to test inertial positioning system. Current implementation just uses latest value. Core module is asking it for creating telemetry package. There is also notification which is sent to core module if free fall is detected.

Air pressure data

BMP280 is a pressure sensor made by Bosch and based on library provided by the company at [23]. Data is used to check internal temperature and calculate altitude. Logic implemented inside BMP class. Data collected in separate thread and asked by core module for creating telemetry package.

3.2.7 Sun sensor module

For testing the TTÜ100 satellite sun sensor at high altitude – a much more realistic usage scenario than laboratory tests – the sensor was added to the payload system.

3.2.7.1 Module description

This is an extra module used to gather data for TTÜ100 satellite development and future projects. Most modern optical computer mouse sensors uses USB interfaces and do not provide image data. Some older computer mice may also have PS/2 interface and there is a chance to get frame data from the image sensor. Based on collected data it is possible to calculate how this sensor is positioning against the sun. In this project ADNS-5020 module is used. It has 15x15 pixels.

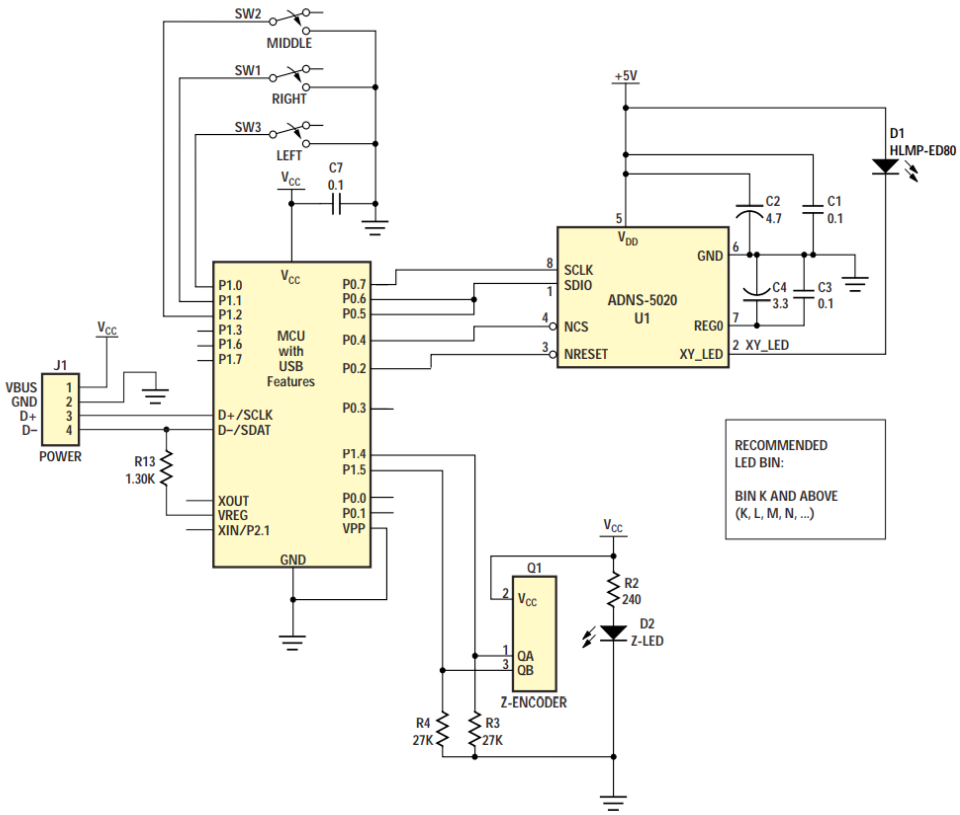

Figure 18: ADNS-5020 usage scenario. From [24]

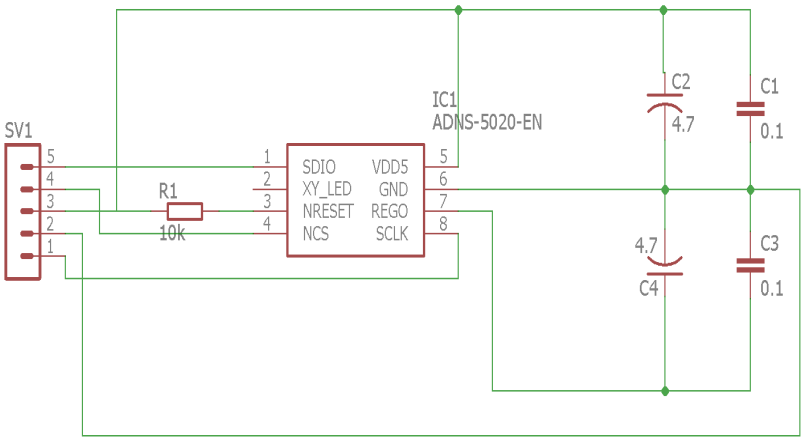

Figure 19: Sun sensor module schematics

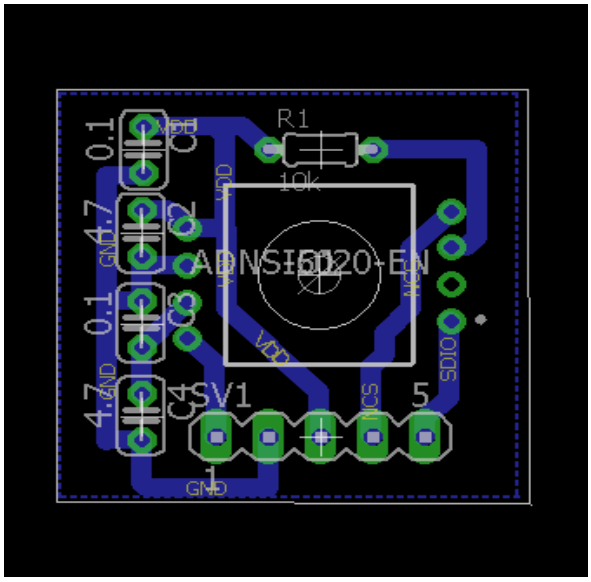

Figure 20: Sun sensor PCB

Sun-sensor PCB was fully made at home using CNC, citric acid and 3% hydrogen peroxide.

3.2.7.2 Software implementation

ADNS-5020 is optical mouse sensor. It's main functionality is providing mouse movement data. For my needs only initialization and frame related registers are needed. Here is overview of registers I use in software for sun-sensor.

Address	Register	Default value
0x00	Product_ID	0x12
0x01	Revision_ID	0x01

0x0b	Pixel_Grab	Any
0x0d	Mouse_control	0x00
0x3a	Chip_Reset	N/A
0x3f	Inv_Rev_ID	0xfe

Table 1: ADNS-5020 registers used for sun sensor module. Based on table from [24]

Pixel_Grab:

The pixel grabber captures 1 pixel per frame. If there is a valid pixel in the grabber when this register is read, the MSB will be set, an internal counter will incremented to capture the next pixel and the grabber will be armed to capture the next pixel. It will take 225 reads to upload the complete image. Any write to this register will reset and arm the grabber to grab pixel 0 on the next image. [24]

Chip_Reset:

write 0x5a to initiate chip RESET. [24]

ADNS-5020 uses singe wire SPI interface. Unfortunately, Raspberry Pi does not support such SPI type, so it was done via bit banging at software layer. It uses C library to manipulate Raspberry GPIO Broadcom BCM2835 chip. Code runs at separate thread asking for new frame each 10 seconds.

Code for getting frame data from ADNS-5020 optical mouse sensor:

```
void SunSensor::Process()
{
	_mutex.lock();
	//Reset frame
	bcm2835_gpio_write(NCS, LOW);
	Pushbyte(0x0b);
	Pushbyte(0x00);
	bcm2835_gpio_write(NCS, HIGH);


	int idx = 0;
	int errors = 0;
	while (idx < 225)
	{
		uint8_t val =  ReadLoc(0x0b);
```

```
            bool bValid = val & 0x80;
            if (bValid)
            {
                        _lastImage[idx] = val & 0x7f;
                        idx++;
            }
            else
            {
                  errors++;

            }

            bcm2835_delayMicroseconds(5);
          }
          _mutex.unlock();

          if (idx > 220)
          {
                  _logFile << currentDateTime() << "nr " << _lastImageNr++ << " Frame= ";

                  for(uint8_t idx = 0; idx < 225; idx++)
                  {
                          _logFile << (int)_lastImage[idx];

                          if (idx < 224)
                                  _logFile << ":";
                  }
                  _logFile << std::endl;
          }
}
```

Frame itself is stored to SD card. Storage is not optimal for size, but easy to use in Excel for investigation. Packed version of the frame (2 bits per pixel) data is transferred to the ground station via radio link. If there is direct sun the picture must be very contrast, nearly black and white. So, 2 bit per pixel may give quite good estimated picture at real time.

3.2.8 Burner

The stratospheric sonde needs to be able to start descending when needed. This module is responsible for such procedure when balloon gets out from predefined geofence or proper command received from ground-station.

3.2.8.1 Module description

For letting go of the balloon several electromechanical ideas were investigated. Electric magnet, servo and stepper motor. They do have disadvantages in weight and they also needs additional hardware.

So, it was decided to use 12 Ohm resistor to burn the rope between payload unit and the balloon. Special fishing line type rope needs to be used that has low melting temperature. In order to get needed power small burning board was created.

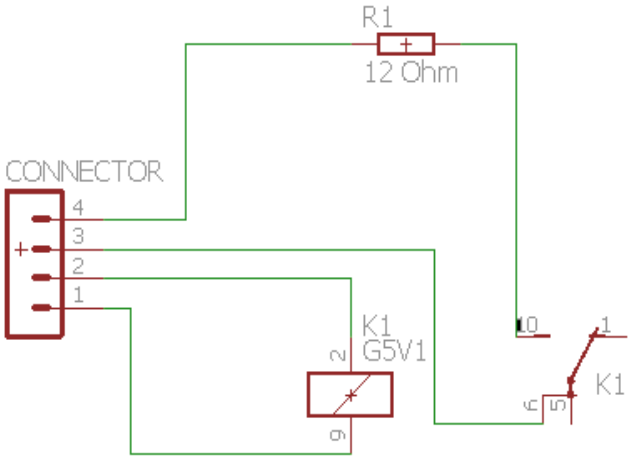

Figure 21: Burner module schematics

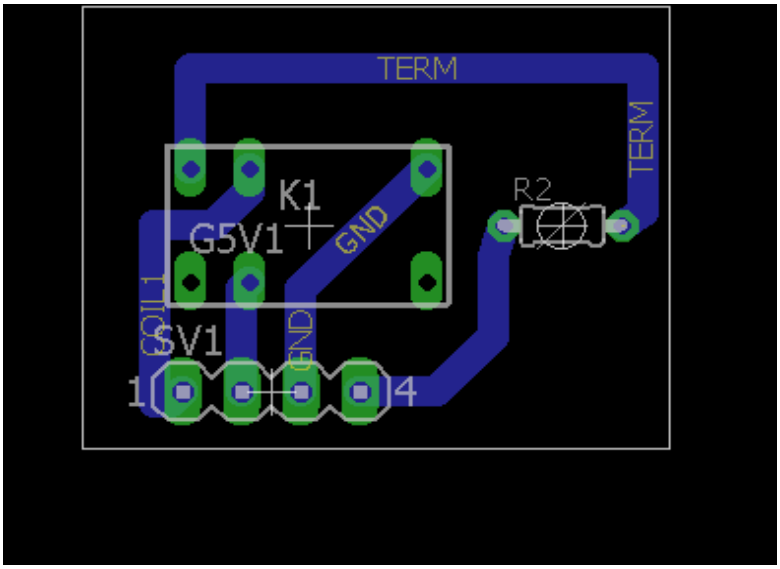

Figure 22: Burner module PCB layout

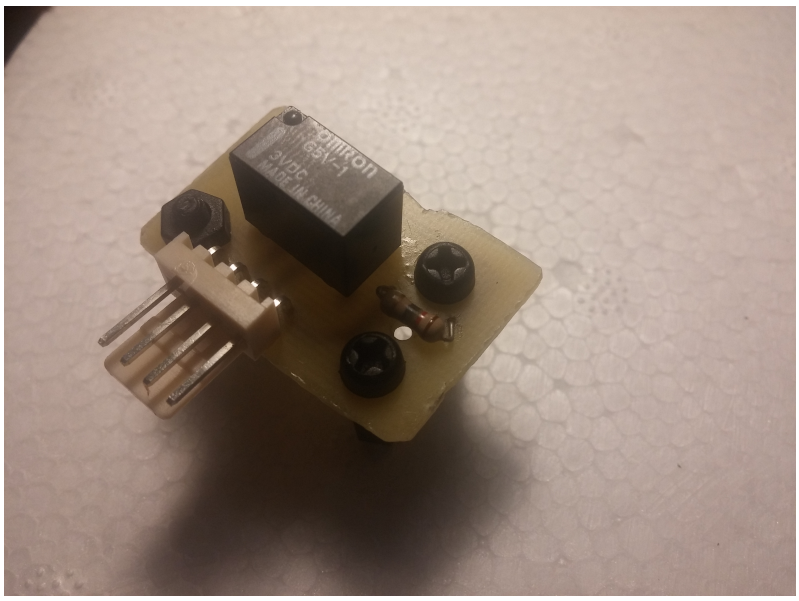

Figure 23: Burner module

Burner module PCB was fully made at home using CNC, citric acid and 3% hydrogen peroxide.

3.2.8.2 Software implementation

Software implementation is done in Burner class. The main functionality is in separate thread. When burning flag is active GPIO pin is toggled to HIGH for 30 seconds.

```
void Burner::BurnThread(int timeout)
{
        while (true)
        {

                if(_bActive)
                {
                        std::cout << "Burning started" << std::endl;

                        bcm2835_gpio_write(BURN_PIN, HIGH);

                        usleep(timeout * 1000 * 1000);

                        bcm2835_gpio_write(BURN_PIN, LOW);

                        std::cout << "Burning ended" << std::endl;
                        _bActive = false;

                }
                usleep(5* 1000* 1000);
        }
}
```

3.2.9 Beeper

Beeper is needed when the sonde has landed and users are close by to retrieve the sond. It can be hard to see in dense forest so sound signal helps to find the sonde. Beeper is just a 3V buzzer in parallel with 3 LEDs. Module produces sound and flashes during 1 minute. From code side it is quite similar to the Burner. The only difference is that GPIO toggles for 1 second during 1 minute.

3.2.10 Power module

Two separate batteries are used for the payload unit. The first one is 800mah 3.7v LiPo rechargeable battery for Syma X5 drone. The second one is 2xMR18650 4400mah Li Ion rechargeable battery. In order to get needed 5V for the Raspberry Pi DC-DC converter is used.

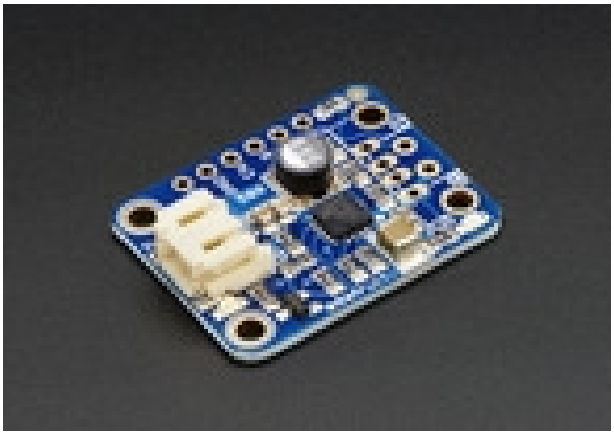

Figure 24: Adafruit PowerBoost 500 Basic [26]

This module can provide at least 1000mA from 3.7V Li Ion battery and has low battery pin. This pin is used in Battery class. When voltage drops below 3.2V rope burning process will be called.

The reason for the second small battery is that GSM module needs at least 3.5V in order to operate and currently there is no monitoring for battery level besides 3.2V from the PowerBoost module. As GSM itself does not use a much of power this small battery is also used for Burner and Beeper modules.

3.2.11 Radio link

Radio link is used for constant real-time monitoring of the flight experiment. GSM may get expensive with constant SMS messages. It does not work at high altitudes. Also GSM may not be available in all locations when sonde is low on the ground.

3.2.11.1 Module description

For radio communication LoRa SX1278 module by AI-Thinker is used.

Figure 25: Payload unit radio link module.

Bought from www.ebay.ie

LoRa specification:

- LoRa™ Modem 168dB maximum link budget
- +20dBm - 100mW constant RF output vs. V supply
- +14dBm high efficiency PA
- Programmable bit rate up to 300kbps
- High sensitivity: down to -148dBm
- Bullet-proof front end: IIP3 = -11dBm
- Excellent blocking immunity
- Low RX current of 9.9mA, 200nA register retention

- Fully integrated synthesizer with a resolution of 61Hz
- FSK, GFSK, MSK, GMSK, LoRa and OOK modulation
- Built-in bit synchronizer for clock recovery
- Preamble detection
- 127dB Dynamic Range RSSI
- Automatic RF Sense and CAD with ultra-fast AFC
- Packet engine up to 256 bytes with CRC
- Built-in temperature sensor and low battery indicator

Data from [27]

The LoRa™ spread spectrum modem is capable of achieving significantly longer range than existing systems based on FSK or OOK modulation. At maximum data rates of LoRa™ the sensitivity is 8dB better than FSK, but using a low cost bill of materials with a 20ppm XTAL LoRa™ can improve receiver sensitivity by more than 20dB compared to FSK. LoRa™ also provides significant advances in selectivity and blocking performance, further improving communication reliability. For maximum flexibility the user may decide on the spread spectrum modulation bandwidth (BW), spreading factor (SF) and error correction rate (CR). Another benefit of the spread modulation is that each spreading factor is orthogonal - thus multiple transmitted signals can occupy the same channel without interfering. [28]

There are successful Balloon experiments whit LoRa™ module that has results of over than 100km distance between transmitter and receiver [29].

3.2.11.2 Software implementation

Radio link functionality is implemented inside LoraClient class. The class is using modification of a SX1278 library for Arduino [30]. The library was initially tested using two Arduino Uno R3 boards and then converted to be used by Raspberry Pi. SX1278 LoRa™ modem has a number of parameters while setting up transceiver.

- **Spreading Factor**

The spread spectrum LoRa™ modulation is performed by representing each bit of payload information by multiple chips of information. The rate at which the spread information is sent is referred to as the symbol rate (Rs), the ratio between the nominal

symbol rate and chip rate is the spreading factor and represents the number of symbols sent per bit of information. [28]

OverheadSpreadingFactor	Spreading Factor (Chips / symbol)	LoRa Demodulator SNR
6	64	-5 dB
7	128	-7.5 dB
8	256	-10 dB
9	512	-12.5 dB
10	1024	-15 dB
11	2048	-17.5 dB
12	4096	-20 dB

Table 2: Range of Spreading Factor [28]

According to the table one can see, that thanks to the Spreading Factor technique it is possible to receive signal with negative SNR, which increases link budget.

- **Coding rate**

For improving link budget there is also implemented configurable forward error detection and correction. But as a result additional data overhead will present in transmission.

CodingRate	Cyclic Coding Rate	Overhead Ratio
1	4/5	1.25
2	4/6	1.5
3	4/7	1.75
4	4/8	2

Table 3: Cycling Coding Overhead[28]

- **Signal Bandwidth**

Higher bandwidth means higher effective data and shorter transmission time but it also reduces link sensitivity.

Bandwidth (kHz)	Spreading Factor	Coding rate	Nominal Rb (bps)
7.8	12	4/5	18
10.4	12	4/5	24
15.6	12	4/5	37
20.8	12	4/5	49
31.2	12	4/5	73
41.7	12	4/5	98
62.5	12	4/5	146
125	12	4/5	293
250	12	4/5	586
500	12	4/5	1172

Table 4: LoRa Bandwidth Options [28]

Current radio link realization has fixed values for those parameters: CR = 4/5, SF = 10, BW = 250 KHz. During my ground experiments I also tried to implement dynamically changing parameter functionality. But complexity of synchronization of bidirectional link with different packet length was too high.

LoraClient class is responsible for the LoRa modem setup. It works with modem registers and communicates to the core module via registered callbacks. The main logic runs in separate thread:

```
void LoraClient::LoraThread(LoraClient* pClient)
{
 SX1278* pLora = pClient->GetLora();
 uint8_t errorCnt = 0;
 while(pClient->IsStarted())
 {
    //in  milliseconds
    uint8_t result = pLora->receivePacketTimeout(10 * 1000);

    cout << "LoraClient receivePacketTimeout result = " << (int)result << endl;
    if (result == 0)
    {
     cout << "packet: [" << pLora->packet_received.data << "]" << endl;
     pClient->Message(pLora->packet_received.data);
         errorCnt = 0;
    }
    else
    {
     errorCnt++;
         if (errorCnt > 5)
         {
          pClient->Setup();
          errorCnt = 0;
```

```
            continue;
          }
      }

          usleep(2 * 1000 * 1000);

          Packet packet = pClient->GetPacket();

          if (packet.ReceiverID > 0)
          {
        uint8_t* message = packet.Data.data();
            uint8_t result = pLora->sendPacketTimeout(packet.ReceiverID, message,
packet.Data.size());

            cout << "LoraClient sendPacketTimeout result = " << (int)result << endl;
          }

    usleep(1 * 1000 * 1000);
  }
}
```

The estimated packet transmission is about 3 seconds. For stable synchronization 3 times longer timeout is used while waiting for receive packet. Similar logic is implemented at the ground station.

3.3 Ground station

The idea of ground-station was small handheld device with autonomous power. Besides monitoring data during the flight experiment such lightweight device may be also used while searching the payload unit at the ground.

3.3.1 Hardware

Ground station has a Raspberry Pi 3 as the main module, LoRa HAT and 10000mah power bank.

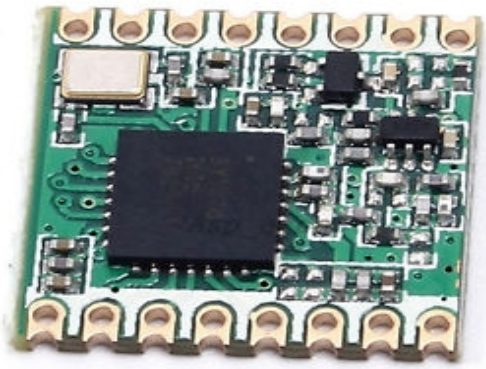

Figure 26: LoRa ground station chip

Ground-station talks to the balloon payload unit via LoRa radio-link. It is used to setup some configurable parameters like geofencing and GSM client phone number prior to the flight experiment. During the flight it is mainly used for collecting data. UI part can be executed from either Android tablet or PC. Communication between UI and ground-station works over Wi-Fi.

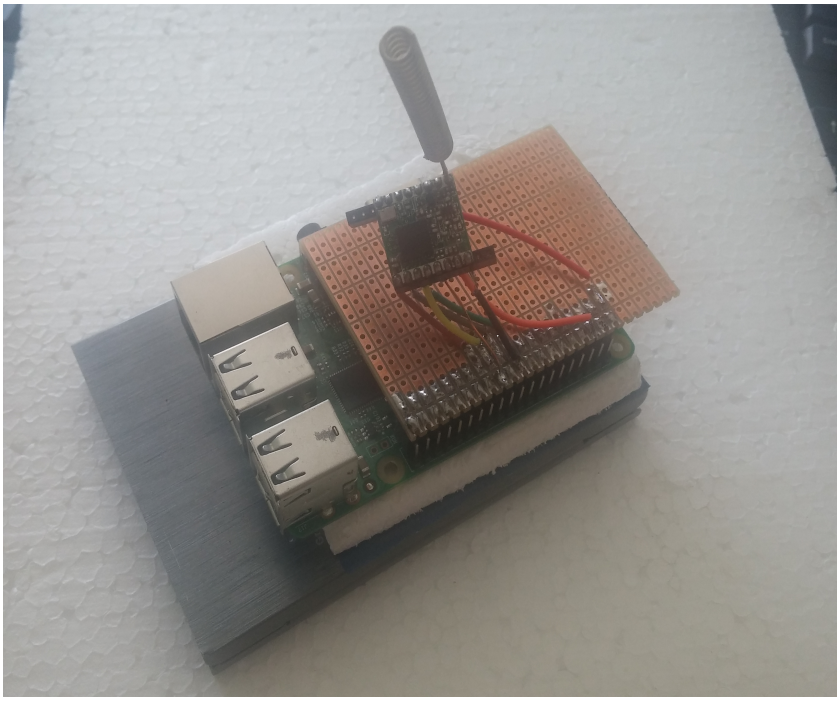

Figure 27: Ground station prototype.

3.3.2 Software

The ground station software consists of 3 modules: LoraClient, Socket and Core. Each module is discussed briefly as follows.

- LoraClient

  Radio link implementation. Talks to balloon payload unit

  Protocol between payload unit and ground-station

  Payload unit packets

  SUNP[data0, data1, .. , data57] total 62 bytes
  SUNP – packet identifier
  Packet sun sensor data. Each byte has information for 4 pixels, 2 bits per pixel

  TELE[data0, data1, … ,data85] total 89 bytes
  TELE – telemetry package identifier
  1 byte last received command number

  3x16
  4 bytes GPS latitude
  4 bytes GPS longitude
  4 bytes GPS altitude

  4 bytes yaw
  4 bytes pitch
  4 bytes roll

  4 bytes temperature
  4 bytes air pressure
  4 bytes altitude based on pressure

  FENG[data0, data1...dataN] variable size
  FENG – fence packet identifier

  Mx12 bytes for each fence point
  4 bytes fence point latitude
  4 bytes fence point longitude
  4 bytes fence point radius

  Ground-station packets

  Currently ground-station packets are sent as text with "|" as delimiter between parameters. All commands are formatted next way:

cmdNr | cmdID | par0 | .. | parN |

cmdNr 1 byte from 0 to 255. This number will be sent back from payload unit in TELE command to indicate last successfully handled packet.

cmdID 1 byte.
Possible values:
1: Burn. No parameters.
Ground-station call to perform the rope burning procedure

2: Beep. No parameters.
Ground-station call to perform beeping and led flashing

3: SetFence. Parameters: 3xN latitude | longitude | radius
Ground-station call to setup geofence.

4: GetFence. No parametes
Ground-station request for currently used geofence

5: SetPhone. Parameter: phone number
Ground-station call to setup GSM module recipient phone number

6: GetSun.No parameters.
Ground-station request for the latest sun-sensor image.

- Socket

Used for bidirectional communication between ground station and UI application. Transfers parsed packed from payload unit and handles commands in opposite direction.

Protocol between ground-station and UI application

Ground-station packets

SetPos | GNSS_ID | latitude | longitude | distance_to_fence | altitude
Sends positioning data for specific GNSS sensor.

SetYpr|yaw|pitch|roll
Sends payload unit orientation data.

SetAccel | gx | gy | gz
Sends payload unit accelerometer data.

SetBmp | temperature | pressure | altitude
Sends payload unit barometric data.

SetFence | 3xN : latitude | longitude | radius
Sends geofence currently used by payload unit

SUNP[binary data]
Sends packed data from last received sun-sensor image.

UI packets

UI sends packet in string format

cmdID | par0 | .. | parN |

Command and parameters description is the same as above in LoRa protocol.

- Core

Entering point, creating and handling LoraClient and Socket modules. All intermediate transformations of LoRa and Client application data is done here.

3.3.3 User interface

Ground station UI is done in Qt Creator IDE using C++ Qt framework and QML. Application can be launched either at PC or Android tablet. UI consists of two views. The first one is done for data visualization. It is used for showing positioning data and to monitor connection status. The second view is used for configuration and sending commands to the payload unit.

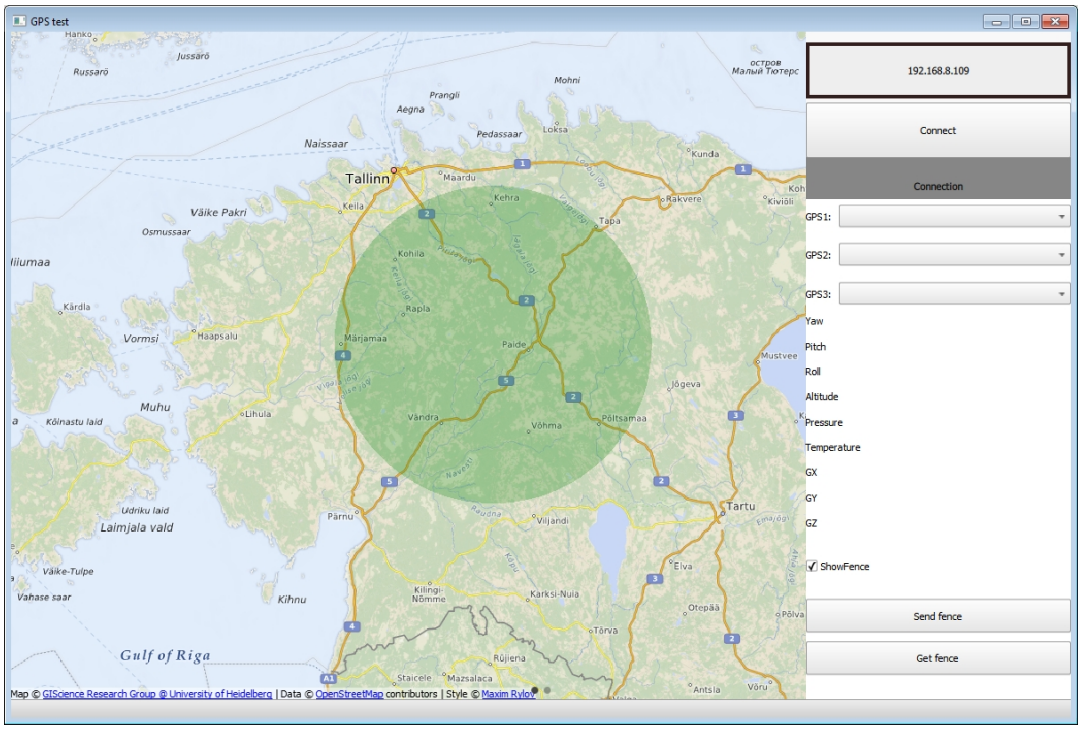

Figure 28: Ground station UI. Telemetry tab view with active geofence

This view is used for telemetry data presentation but also for creating and sending geofence to the balloon payload unit. When there is active connection – the Connection label will became green. Packet receive time and RSSI will be displayed there. If there is a case of receive packet timeout – the background will became red. Three GPS comboboxes store all coordinates sent by the balloon payload unit. Selecting an item will move the map so, that selected position will be at the centre of it. Status bar shows selected position coordinate and altitude based on NMEA GPGGA sentence. There is also data from motion and air pressure sensor. First fence button is used to create and send fence data to the payload unit. The second button will ask module to transfer its currently used fence data to the ground station.

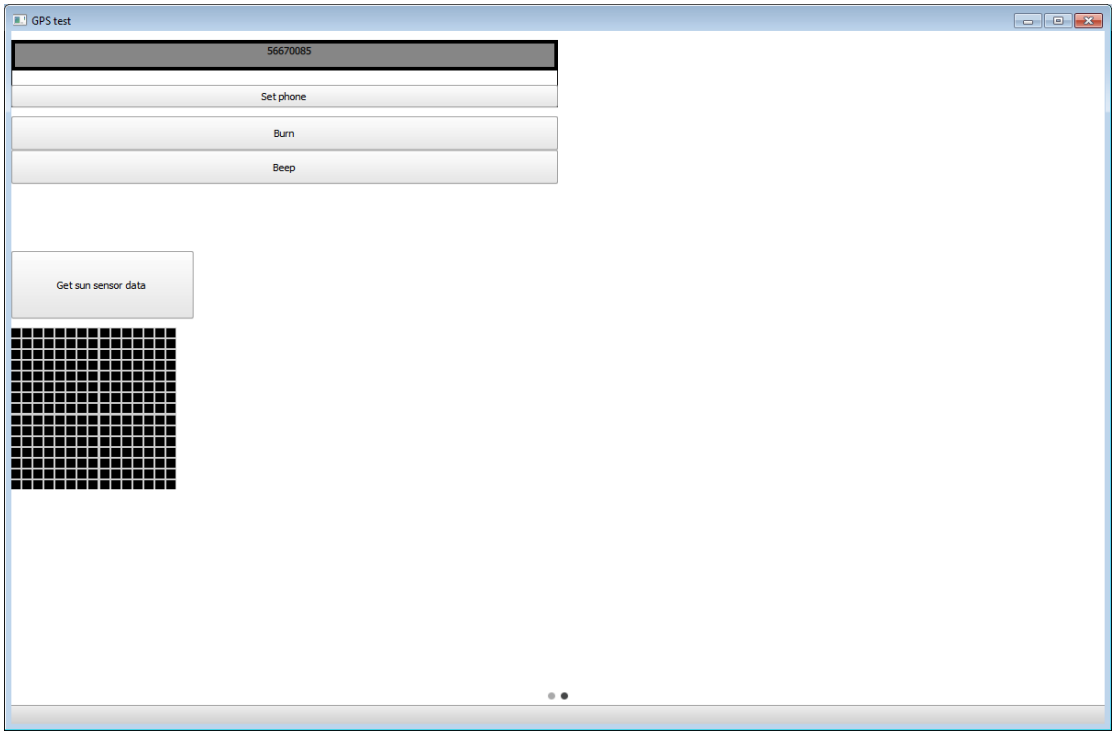

Figure 29: Ground station UI. Configuration tab

This view contains configuration and interaction functionality. “Set phone” will send to the payload unit phone number to interact with. This is the number payload unit will send SMS to. “Burn” will notify that payload unit must burn the rope. “Beep” will activate Beeper at payload unit.

There is also region representing sun-sensor data. It contains 15x15 rectangles, each representing pixel state from the latest sun-sensor image This region is updated once in a minute or as a response on “Get sun sensor data” button click. This is a compressed frame data with 4 shades of grey (2 bits per pixel).

4 Results

4.1 Tests

4.1.1 MPU Test

Server application for Raspberry Pi Zero was written in C++. Client application was written in C++ using QT Framework. Server asks data from MPU9250 20 times in a second. After each successful data read application sends yaw, pitch and roll values over socket connection to client. The cube was moving according to board orientation.

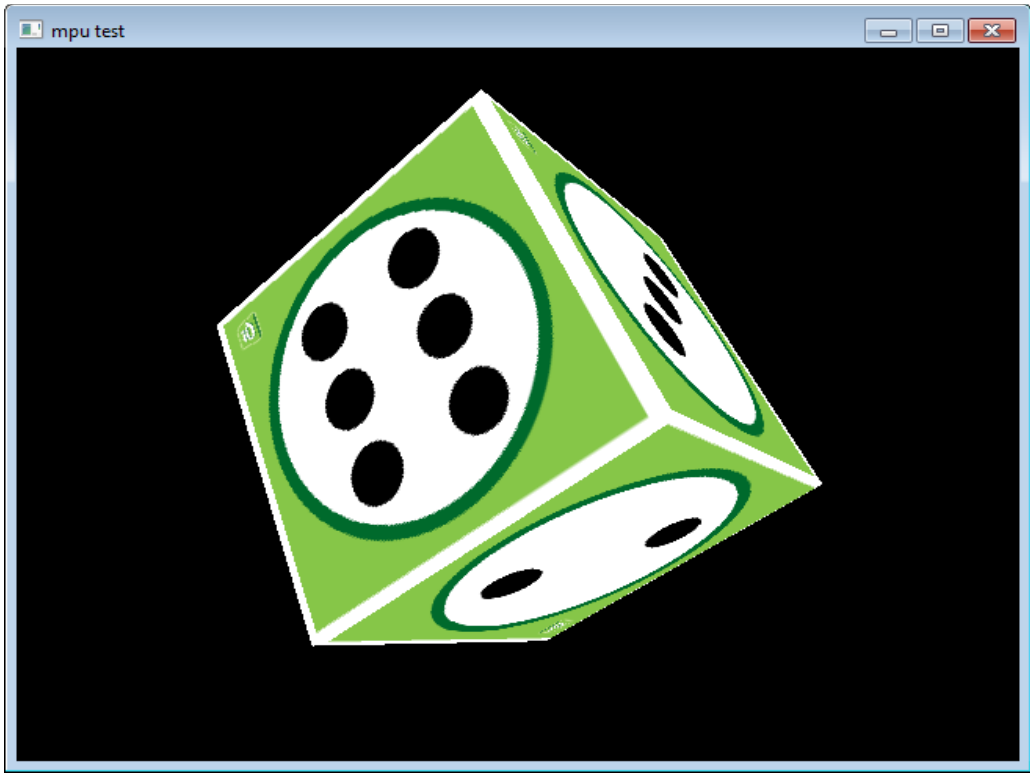

Figure 30: MPU9250 test

4.1.2 LoRa Test

Open air direct vision test was performed at Lohusalu Bay 7.4.2018.

Modules tested: 2x Ra02 SX1278 with a spring antennas
Server: Arduino Uno R3

Client:Raspberry Pi 3

Test: one-directional communication test

Experiment: Server was left in my car at Lohusalu Harbour. I was walking along the beach line with the client trying to reach point where signal will be lost.

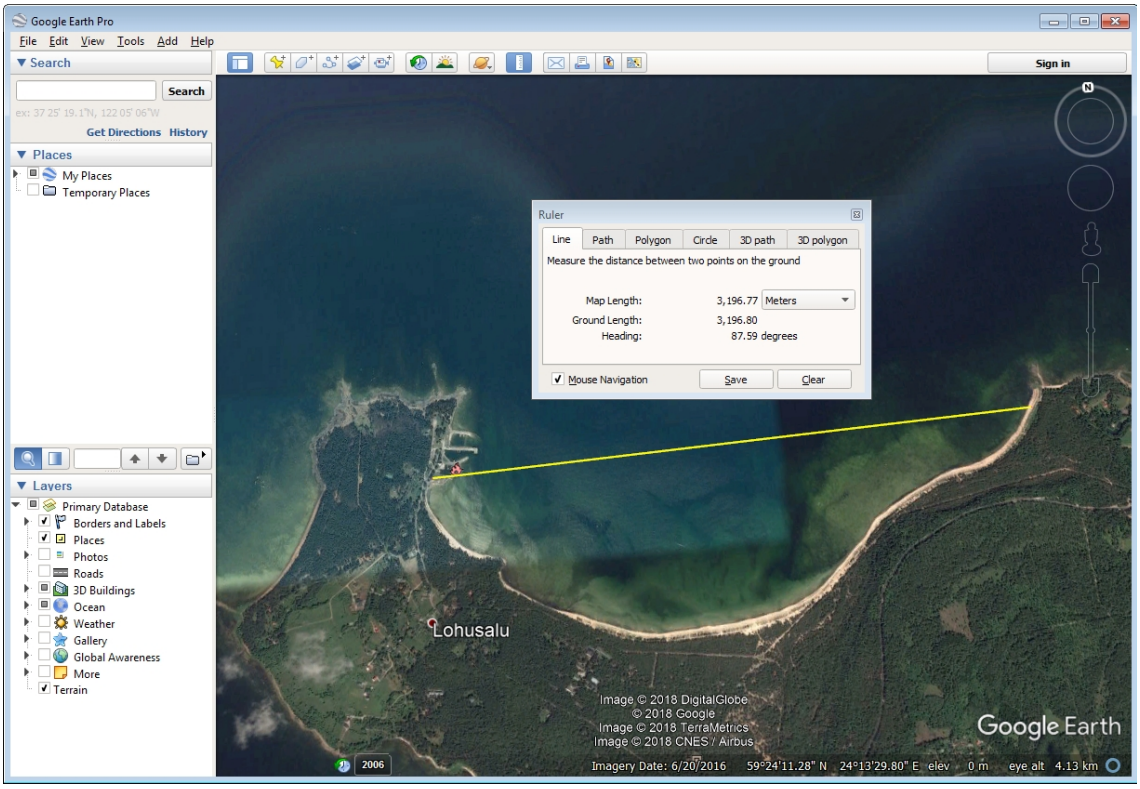

Figure 31: LoRa Direct vision test

Server was sending "000Lavandysh!!!", where first three bytes were incremented after each successful message send. After each message there was 4 second delay. After 10 message LoRa modem settings were changed.

Client was waiting for messages and switched into new mode if third byte % 10 == 0

Modes in use:

Mode	Coding rate	Spread Factor	Bandwidth (KHz)
0	4/8	12	7.8
1	4/5	12	125
2	4/5	12	250
3	4/5	10	125
4	4/5	12	500
5	4/5	10	250
6	4/5	11	500
7	4/5	9	250

8	4/5	9	500
9	4/5	8	500
10	4/5	7	500
11	4/8	10	62.5
12	4/8	10	41.7
13	4/8	10	31.2
14	4/8	10	20.8
15	4/8	10	15.6
16	4/8	7	10.4
17	4/8	10	7.8

Result:

Modes 1 – 15 were stable till the end.

Modes 0 and 17 did not worked at all

Mode 16 worked in 20% no matter what distance it was

Communication works for the whole possible direct view distance at this place, which is 3196 meters, according to the Google Earth software.

4.1.3 Sun sensor test

I have written test application for Raspberry Pi Zero that asks the optical sun-sensor for picture data 10 times in a second. After each successful frame read the application sends it over socket connection to client. Client application works at PC and is written in C++ using QT Framework. For a quick test a pocket flash-light was directed to the sun sensor, results can be seen in Figure 33.

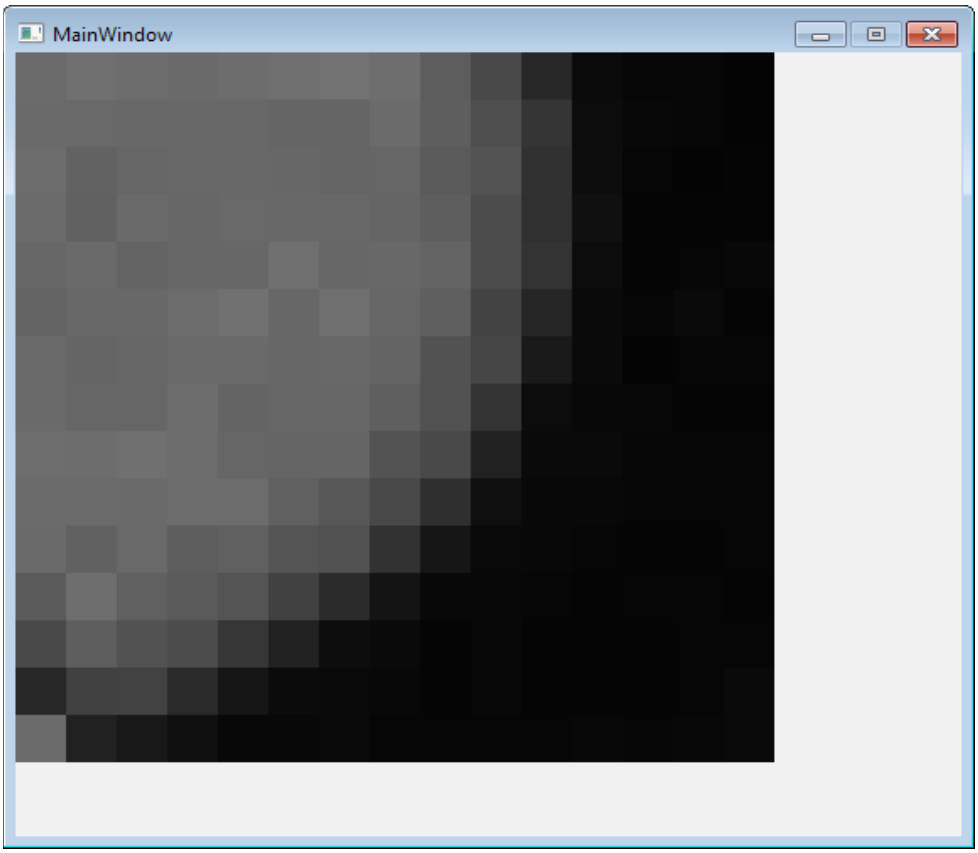

Figure 32: Sun sensor test

4.2 Flight experiment

Flight experiment was held 11.05.2018 and launch-site was at Väätsa school stadium. Balloon payload was composed of the system described in this thesis, “Pi in the sky” system and additional action camera. For safety reasons the system was also equipped with a GPS-GSM tracker. Expected result was 20km altitude.

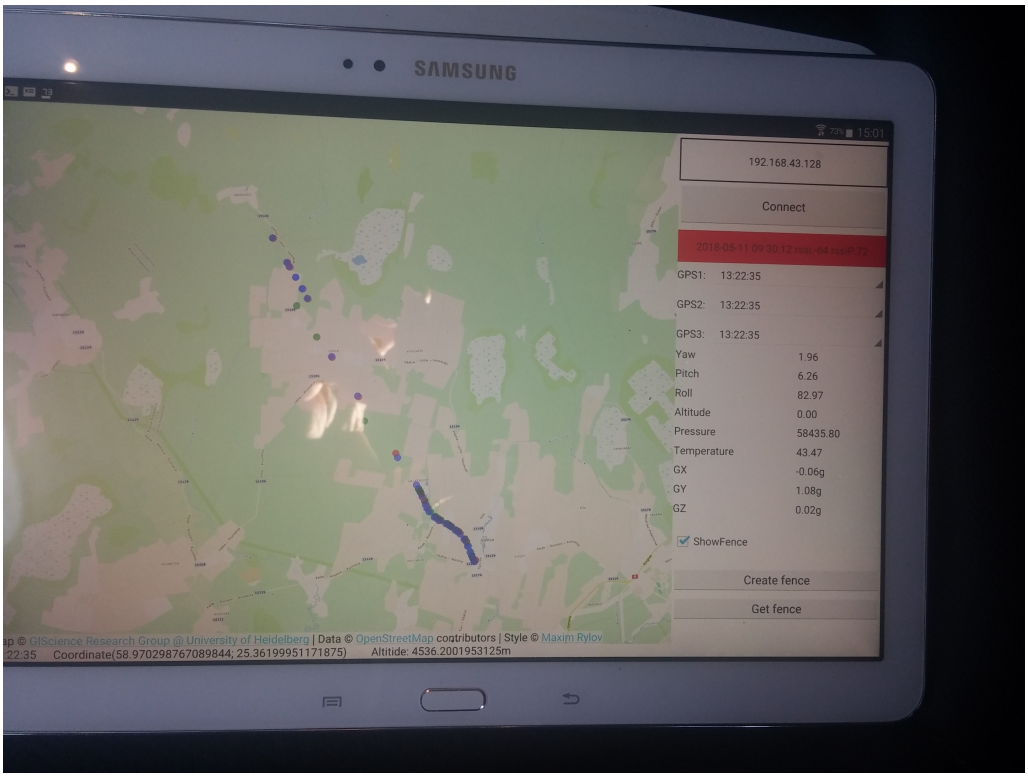

Figure 33: Experiment results in UI

The balloon was launched at 12.55. GSM module has sent two messages:

12:57
http://maps.google.com/maps?q=58.8914,25.4471
http://maps.google.com/maps?q=58.8905,25.4480
http://maps.google.com/maps?q=58.8907,25.4474

13.07
http://maps.google.com/maps?q=58.9253,25.4039
http://maps.google.com/maps?q=58.9198,25.4062
http://maps.google.com/maps?q=58.9254,25.4037

During first 20 minutes telemetry was stable. After that, client was able to receive data from payload unit after making additional synchronization tasks, including LoRa modem restart. Last successful packet received at 13.22. Last position sent from payload module was 58.9703N 25.3620E. After that connection was lost. Balloon payload module was found at 59.0487N, 25.2933E. When module was found the rope was burned and battery for the rope burning resistor was nearly empty. This could happen if balloon would gets out the geofence and will stay there for a long time. Looks like some calculation error in distance measurements happened because geofence was set to Väätsa school stadium with safety radius of 53 km.

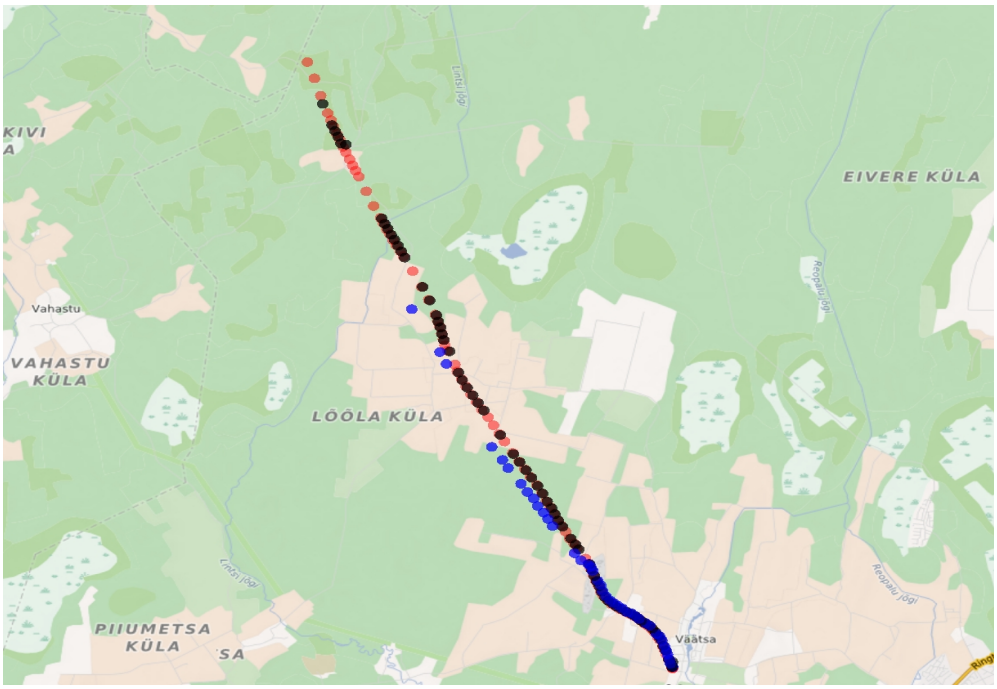

Figure 34: GPS tracks from logs

Red dot GN-8013
Black dot NEO6M
Blue dot V.KEL VK16E

Tracking reconstruction based on the logs from payload unit. The picture is a bit different comparing to Figure 34. The reason is that ground-station UI show only data sent by telemetry command. The command comes every 15 seconds and there are no intermediate points. But the payload stores all the points it got from the GNSS sensors. Although the flight time was not too long, some conclusions can be done. The number of blue dots is smaller comparing to black and red dots. Also, positions got from the blue GNSS sensor a bit different. As the dots radius is 100m the difference is about 250m to the West, while two other sensors shows almost the same position.

Because of very low altitude there were no required conditions for proper sun sensor test. There is too much ambient light at low altitudes.

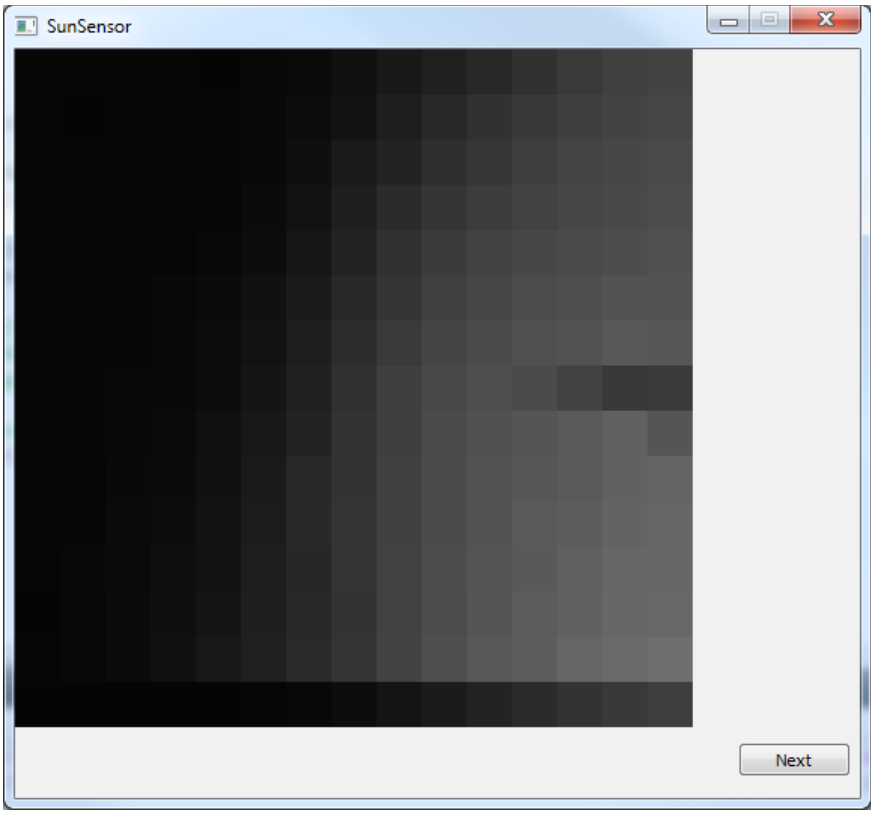

Figure 35: Sun sensor data output

Here is one example of collected data from sun sensor. The picture does not have enough contrast and the sun circle radius is too big (probably due to improper pinhole size).

5 Conclusions

The work was divided into two main parts: creating hardware design with realization and software development to control the whole system. The first part took more time than was initially expected. But the final PCB version is more stable comparing to prototype.

The system made for this thesis has passed the first flight experiment. All system parts were working as expected. The LoRa communication works till the fall time. The connection was tested when payload unit was found. The signal was present, but due to empty battery it was unable to perform sound and led alarm signals. The GNSS sensors have worked similar way – quite stable till falling time, with no data during fall. GSM module was working till some altitude and it was available again after the landing till battery became low.

The main goal of the thesis is reached. The first version of working system exists. It can be re-used in new high altitude experiments.

6 Discussion

There are number of things to be improved. Ground station needs to be done on proper PCB instead of prototype board. Directed antenna instead of spring antenna should be used to increase the radio link distance. GSM module needs to be added to the ground-station.

There was software bug while converting position to decimal representation. This bug caused shorter flight, cause incorrect calculation has moved the real position ca 50km to the North. Some other software changes to be done: additional logging, parameters allowing to change configuration during flight mode.

References

[1] http://www.gps.gov/systems/gps/performance/accuracy

[2] http://www.slideshare.net/VineethSundar1/weather-balloon-43179528

[3] https://www.sparkfun.com/tutorials/169

[4] http://www.wikipedia.org

[5] https://racelogic.support/01VBOX_Automotive

[6] http://www.pi-in-the-sky.com/

[7] http://www.raspberry.org

[8] https://www.qt.io

[9] https://developer.android.com/

[10] https://www.eclipse.org

[11] http://gnutoolchains.com/raspberry/

[12] https://www.autodesk.com

[13] https://www.semtech.com

[14] http://www.ti.com/product/CD4052B

[15] http://www.airspayce.com/mikem/bcm2835/

[16] http://www.ittgroup.ee

[17] http://simcom.ee/modules/gsm-gprs/sim800/

[18] http://www.gpsinformation.org/dale/nmea.htm

[19] https://en.wikipedia.org/wiki/Haversine_formula

[20] https://stackoverflow.com/questions/365826/calculate-distance-between-2-gps-coordinates

[21] https://www.invensense.com/developers/software-downloads/

[22] https://github.com/rpicopter/MotionSensorExample

[23] https://github.com/BoschSensortec/BMP280_driver

[24] http://www.alldatasheet.com/datasheet-pdf/pdf/203509/AVAGO/ADNS-5020-EN.html

[25] http://frenki.net/2013/12/convert-optical-mouse-into-arduino-web-camera

[26] https://www.adafruit.com/product/1903

[27] https://www.semtech.com/products/wireless-rf/lora-transceivers/SX1278

[28] https://www.semtech.com/uploads/documents/DS_SX1276-7-8-9_W_APP_V5.pdf

[29] https://www.thethingsnetwork.org/article/ground-breaking-world-record-lorawan-packet-received-at-702-km-436-miles-distance

[30] https://github.com/wirelessopensource/lora_shield_arduino

[31] https://www.findmespot.com/en/index.php?cid=100

[32] http://www.eugenewei.com/blog/2015/8/3/cocom-limits

[33] http://www.toptenreviews.com/electronics/gps/best-gps-trackers/

[34] https://en.wikipedia.org/wiki/Project_Loon
[35] https://x.company/loon/technology
[36] https://en.wikipedia.org/wiki/High-altitude_balloon