Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Team Firebusters Report

FloatSat Laboratory
Chair of Aerospace Computer Science

Elisa Boccolari Khaireddine Essid Christoph Liebender
Kush Kumar Sharma Uma Parvathi Mallissery Manyan

13 March 2023

Contents

1 Mission

The change in the Earth's climate has caused many forest fires in recent years. Early detection of such fires enables local fire departments to react quickly, thus preventing substantial damage to the environment and its people.

This is the inspiration of team *Firebusters*: the design of a FloatSat that detects hot points in a scene, to simulate the detection of fires on earth. This report summarizes how we proceeded with this idea.

2 Mechanical

The conceptual idea behind the design of the FloatSat is based on the perspective of simplicity and sustainability. The 3D realization of the design was achieved using the CAD software SolidWorks. The main tasks under mechanical are as follows:

- Requirement Analysis
- Preliminary Design
- Detailed Design
- Manufacturing
- Assembly

2.1 Requirement Analysis

In accordance with the mission definition, the requirements of the FloatSat were defined at the component level. These requirements can be broadly classified into initial/qualitative requirements which are more generalized and were provided at the beginning of the project and mission-specific requirements which were derived from mission aims and objectives.

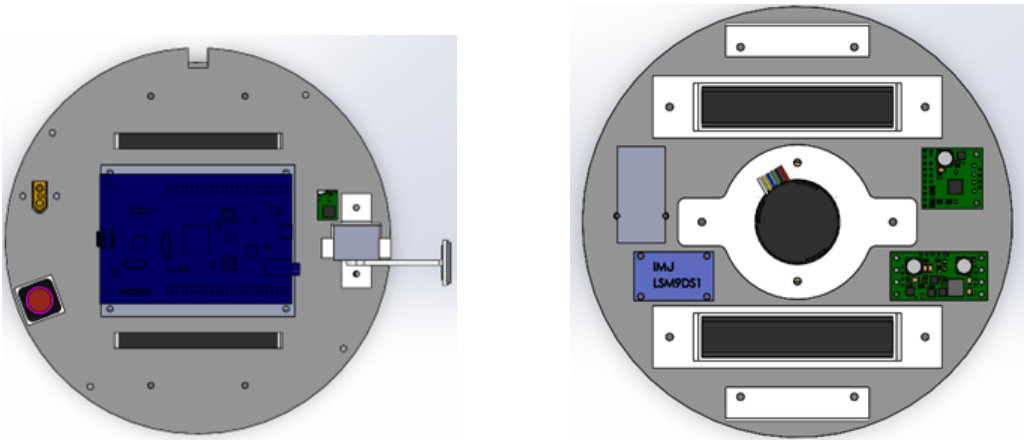
1. Initial/qualitative requirements:
    - Positioning of the satellite in the hemisphere
    - Balanced center of mass (COM) along the vertical axis (towards the bottom of the hemisphere)
    - Manufacturing from available tools and materials
    - Avoid wastage of material
2. Mission-specific requirements:
    - Mechanism and platform for the payload (IR sensor for fire detection)
    - Positioning of other essential components including sensors and actuators

2.2 Design Analysis

Based on the requirements, a list of desired components was prepared to determine their mass and dimensions. Accurate dimensional measurements were achieved using a vernier caliper. The necessary components for the concept of operation (ConOps) of the satellite were namely microcontroller, PCB, motor driver, voltage regulator, IMU, Bluetooth module, motor, and reaction wheel. The FloatSat was designed according to the limiting dimensions of the hemisphere with a diameter of 20 cm. The idea was to miniaturize the satellite to be able to fit in the hemisphere. Another important aspect of the design was to maintain symmetry throughout the process.

2.2.1 Preliminary design

The FloatSat has a two-plate structure screwed with two parallel-attached C-section columns separated at 3.3 cm. The upper plate with a diameter of 19 cm has the attached platform for mounting the servo and an arm for the payload (IR sensor). For better accessibility, the STM board connected to the PCB is placed right at the center of the top plate. In addition to it, on the other side, there is an XT-60 connector and a switch as shown in figure 1a. The lower plate with a diameter of 18 cm has a countersunk motor holder mounted from the top in the central position along with battery holders attached equidistant on both sides. The motor holder holds the attachment for the motor with the reaction wheel. The sensors including IMU, voltage regulator, Bluetooth, and motor driver are placed alongside as shown in figure 1b.



(a) Upper plate (b) Lower plate

Figure 1: Topviews of assembly

2.2.2 Detailed design

After the first review of the proposed design, there were several design iterations involving the implementation of industrial design standards used in manufacturing such as necessary clearance, tolerances, etc. The holes for the screws were standardized to M2.5 loose fit with a 3.1mm diameter. Additionally, extra holes were

provided considering passing electrical wires for connections. For the placement of the ON/OFF switch and the charging connector, specific holders were designed according to the requirements.

1. First detailed design iteration:
    - Improvising battery holder/motor holder design
    - Clearance for batteries (1.5 mm on each side)
    - Necessary filleting to avoid stress points
    - Standardizing the screw holes
    - Checks for COM
2. Second detailed design iteration:
    - Clearance for the required M2.5 nuts
    - Checks on the symmetrical alignment of components.
    - Fixing the servo holes alignment and chamfering the edges of the servo mount
    - Equidistant check for the Servo mount and connector/switch from the center of the top plate
    - Clearance checks for a) Batteries with hemisphere and b) reaction wheel with hemisphere
    - Checks for COM

2.3 Manufacturing

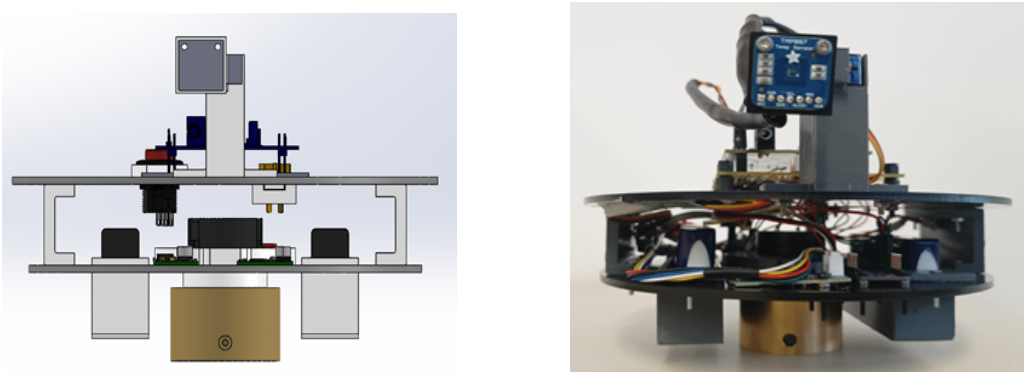
As per the availability, the materials used for the construction of the FloatSat are polylactic acid (PLA) and Plexiglas. A 3 mm thick Plexiglas was used for the top and bottom plates and given the desired shape using a laser cutter. Additional components such as the motor holder, connector holder, battery holder, switch base, C-section columns, arm, and servo mount were manufactured using 3D printing with PLA material.

2.4 Assembly

The mechanical design along with manufacturing was completed by the end of December. During the first week of January, all the components were placed according to the 3D design as shown in figure 2. In the end, the FloatSat was placed in the hemisphere on top of the air-bearing platform to check the balancing and it was found stable.

Parameter	Value
Mass of FloatSat	1106 g
Center of Mass	X = 0.16 mm Y = -36.12 mm Z = -1.02 mm
Principal moments of inertia	P_x = 2298441.90 $g \cdot mm^2$ P_y = 2503582.52 $g \cdot mm^2$ P_z = 3219369.18 $g \cdot mm^2$

Table 1: Mass Properties



(a) Complete assembly in Solidworks (b) Final assembly of FloatSat

Figure 2: Comparison between concept and final result

3 Electrical

The electrical design plays a significant role in this project. This section describes the electrical requirements and the design consideration for the project. The main tasks were to design a battery management circuit and the circuit for the satellite connecting all the components. Figure 3 shows the diagram of the electrical system of the satellite.

The power is supplied to the components by connecting two LiFePo4 (2100mAh, 7.2V each) batteries in parallel, supplying a total voltage of 7.2V to the system. This was connected through a switch to the voltage regulator to step down the voltage to 5v, which was the bus voltage and this was then provided to all the boards through the wires. The motor driver IC, servo motor and the STM32F4discovery boards were directly powered by the regulator and the Bluetooth. IMU and the temperature sensor were powered from the 3v and 5v pins of STM32f4discovery board.

The temperature sensor and IMU communicate to the board through I2C. The necessary PWM signals for the servo motor and driver IC were provided by the PWM pins in the board.

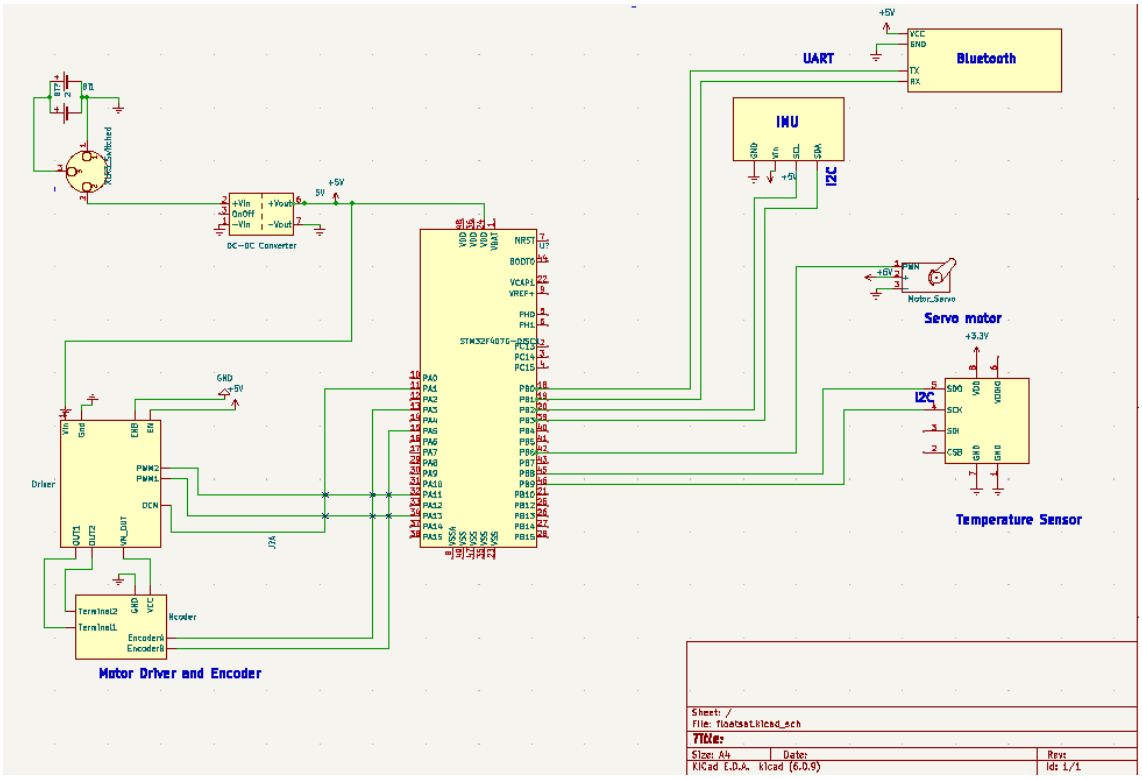


Figure 3: Schematic of the Satellite

3.1 Requirement Analysis

The main requirement of the electrical system is to generate, regulate and distribute power through the system. The electrical system is also responsible for the efficient wiring of the components with the board. One of the main requirements was also to design a battery management system this includes

- real-time power consumption measurement
- low voltage cut-off circuit

The STM32F4Discovery board was provided, so the task was to connect the reaction wheel, servo motor, sensor, etc to the board.

3.2 Battery Management System

The figure 4 depicts the battery management system schematics

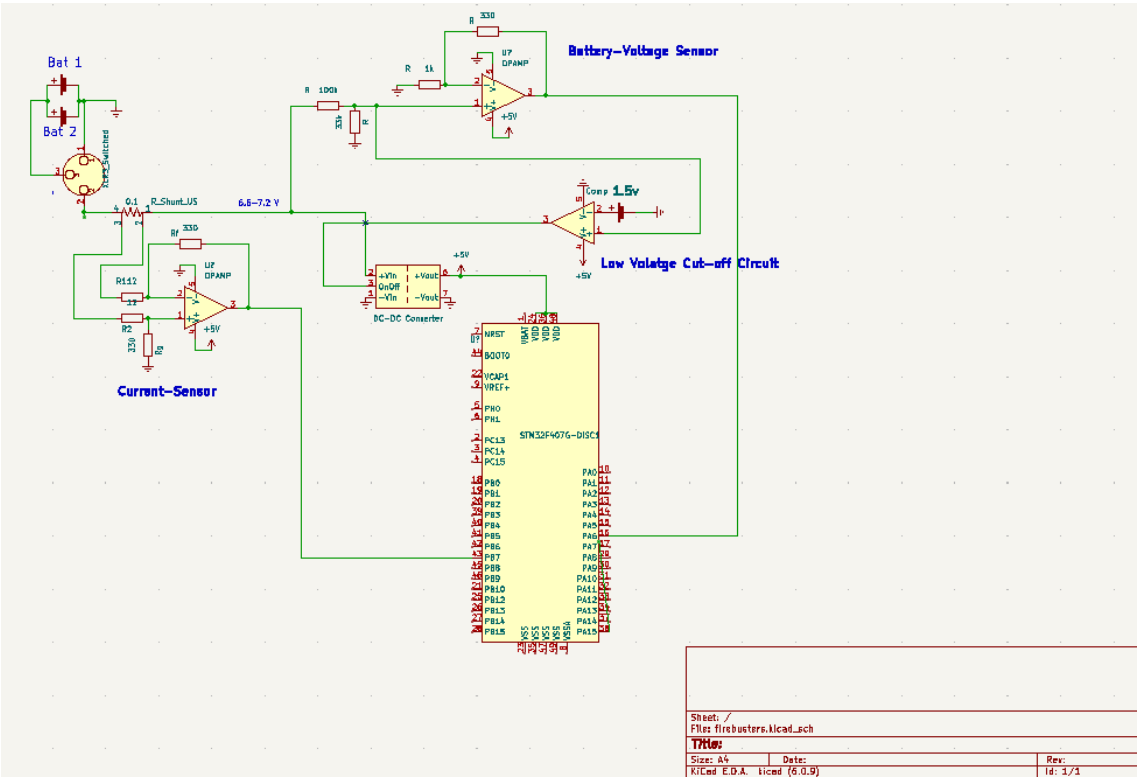


Figure 4: Schematic of battery management system

Battery management system was one of the requirements for the electrical design. This comprises a circuit for measuring the real-time power and a cut-off circuit for low voltage.

3.2.1 Real time power measurement

The main function is to measure the real-time voltage and current drawn by the circuit. The circuit was designed using a general-purpose operational amplifier MCP6022. The battery voltage was measured through a voltage divider circuit connected to the op-amp and the current was measured using a current sense resistor connected in series with load. The drop across this resistor was measured by the op-amp. The out of the op-amp was read by the ADC pin of the board.

3.2.2 Low voltage cut-off circuit

The main objective of this circuit is to prevent the battery from draining out completely, by cutting off the circuit when the battery voltage drops beyond a certain level. The circuit was designed using a TLV3702 comparator and a 1.5v button cell for reference. The battery voltage was compared with the reference and the output was given to the enable pin of the voltage regulator. Using this circuit when the comparator output is zero the regulator will stop working and the circuit will turn off. with this circuit design, the system stops operating at the voltage of 6.1 V.

4 Control

4.1 Requirements and Modes

The project's mission was to scan the space around the FloatSat to detect potential fires. For this purpose, the controller design had to be able to control the attitude of the FloatSat on the z-axis, its position, and speed. To achieve that, the controller consists of three different modes. The motor speed controller is composed of a PI part, that gives as output the duty cycle. With the duty cycle, a voltage of $\pm$ 5V is applied to the motor. Due to the presence of this limit, an Anti-Wind-Up function with clamping was added to avoid instabilities due to the cumulative saturation of an integral part. Clamping stops the integration in case the output of the controller is limited. The motor speed controller is used as an inner loop of two independent cascade controllers, one for the position and one for the speed of the FloatSat. The RW velocity control includes a PI block that returns as output the duty cycle which becomes the reference input of the motor speed control. A similar one has been developed for the position control, this time with a PID, with the differential part necessary to try to attenuate the inevitable overshooting. Two different mission modes were implemented:

- Scanning with constant velocity
- Scanning with fixed angles.

4.1.1 Scanning with constant velocity

When the "scanning with constant velocity" is activated a velocity in rpm is given as reference and the satellite tries to reach and maintain that defined speed while the servo motor moves the IR sensor vertically, allowing a scan in both azimuth and elevation. The inputs to this mode are the desired velocity, the time the servo takes to complete a full scan, and the number of times it stops.

4.1.2 Scanning with fixed angles

With the "scanning with fixed angles", the user defines an angle in degree after which the FloatSat should stop and scan. The servo starts scanning whenever the position is stable and a new reference position is set, summing the angle to the previous one. The criteria used to decide when it is time to scan does not depend on the error between the reference and the value measured with the IMU, but only on a threshold on the variance of consecutive measurements. The inputs, in this case, are the angular position in degree along with the time and steps of the servo.

4.2 Simulink Modeling

To begin with, in order to be able to simulate the behavior of the FloatSat in Simulink the values of the parameters of the DC motor had to be determined. Resistance and inductance were measured with a multimeter. Then, those values were used in the following equations to calculate the other parameters, considering a situation of

steady state (which means the derivatives are zero) and that the moment of inertia of the wheel is small enough compared to the one of the FloatSat and can be neglected ($J_w \gg J_m$).

$$V = L\frac{\mathrm{d}i}{\mathrm{d}t} + Ri + K\omega_m$$

$$Ki = J_w\frac{\mathrm{d}\omega}{\mathrm{d}t} + b\omega_m$$

Table 2 shows the motor parameters:

Parameter	**Value**
Moment of inertia of the FloatSat (J_s)	$0.0032 \cdot 10^4\ Kg \cdot m^2$
Moment of inertia of the reaction wheel (J_w)	$1.175 \cdot 10^4\ Kg \cdot m^2$
Motor viscous friction constant (b)	$1\ Nm/rad/s$
Motor resistance (R)	$1\ \Omega$
Motor inductance (L)	$0.0017\ H$
Constant factor (K)	$0.0085\ V/rad/s$ (or Nm/A)

Table 2: Model Parameters

With the motor parameters, the transfer functions that represent the plant of the model of the FloatSat in Simulink could be determined.
For the motor speed:

$$T_1 = \frac{W_m(s)}{V(s)} = \frac{K}{(b + sJ_w)(R + sL) + K^2}$$

For the satellite speed:

$$T_1 = \frac{W_s(s)}{W_m(s)}\frac{W_m(s)}{V(s)} = -\frac{J_w}{J_s}\frac{K}{(b + sJ_w)(R + sL) + K^2}$$

For the satellite position:

$$T_1 = \frac{\theta_s(s)}{W_m(s)}\frac{W_m(s)}{V(s)} = -\frac{J_w}{J_s \cdot s}\frac{K}{(b + sJ_w)(R + sL) + K^2}$$

In Figure 5 the two control loop models are shown.

The simulations were run to determine the optimal values for the controller's parameters, i.e. to minimize the steady state error and reduce as much as possible the settling time (ideally less than 10 sec). Although the results obtained with Simulink seemed quite good, the behavior when implemented on the hardware was very different. So, it resulted in the necessity to re-tune the controllers empirically, with a series of tests. In Table 3, the ultimate values are displayed. The software implementation of the controllers is described in detail in 5.4.

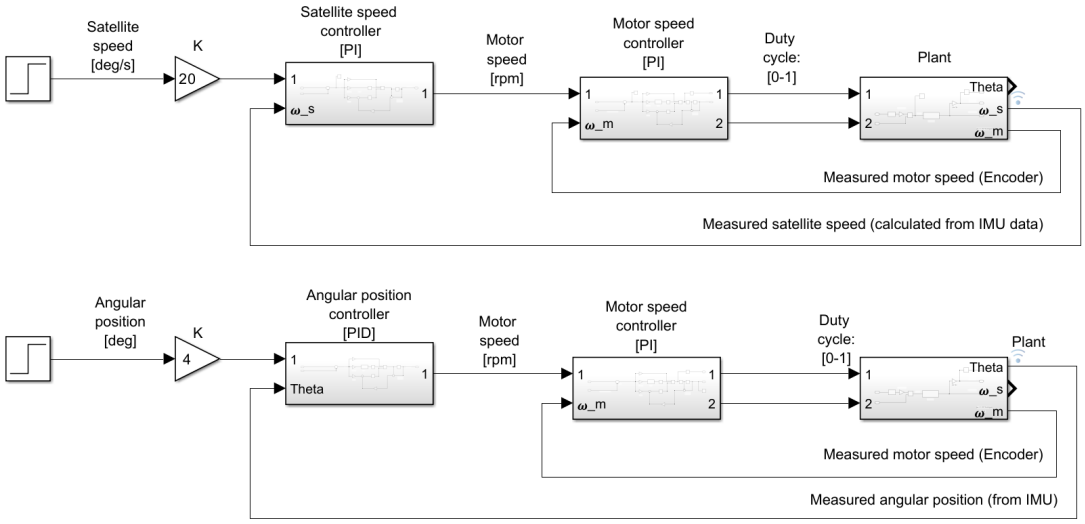


Figure 5: Simulink model of the Speed and Position Controller

Controller	Gain	Value
Motor speed controller	K_P	0.5
	K_I	0.005
Velocity controller	K_P	150
	K_I	1
Position controller	K_P	100
	K_I	0.5
	K_D	50

Table 3: Control Parameters

5 Embedded Software

To program the satellite, the real-time operating system RODOS was used with the language `C++`. As a toolchain, the shell scripts bundled with RODOS in its GitLab repository were preferred over the provided Eclipse workspace. [2] The main principles of the space segment are:

1. No unnecessary heap-allocations ⇒ stack-oriented programming
2. Preferring `float` over `double` computations (32-bit FPU of STM32F4)
3. Easy configuration of parameters through telecommands
4. Making use of RODOS' thread-scheduler where possible.

5.1 Thread structure

In total, there are five threads scheduled to repeatedly run on the satellite:

1. Telecommand
2. Telemetry

3. ADCS
4. Mission
5. Housekeeping.

Only numbers 3-5 run at fixed intervals. The TC thread may interrupt the TM thread upon arrival of a frame. Otherwise, the RODOS-Scheduler is utilized by assigning priorities to the threads, shown in Table 4.

Thread	TC	TM	ADCS	Mission	Housekeeping
Priority	151	150	153	152	150

Table 4: Thread priorities. Higher values get served first.

The Housekeeping thread is not mentioned in the following sections, since its only task is reporting aggregated values from other threads. The communication between these threads is achieved by using the topic middleware provided by RODOS.

5.2 Telecommand

In order to send commands to the satellite, it is necessary to define a format that encodes the information that is to be sent. In this mission, this information is either a command on its own or a command with additional payload. The telecommand thread parses the command and its payload, publishing the decoded command to the respective topic of the other threads.

5.2.1 Format

The format of one telecommand frame consists of the following tokens, without whitespace:

```
START_BYTE IDENTIFIER [ARGUMENT DELIMITER ...] STOP_BYTE
```

where `[ARGUMENT DELMITER ...]` can be a repeating pattern of arguments, where the last delimiter is omitted. Possible arguments may be either integers or floating point numbers with a "." denoting the decimal point. To increase the usability of the satellite with a serial terminal, the encoding was chosen to be ASCII. This results in all tokens being representable as readable characters.

It follows a short excerpt of telecommands, to give an example:

```
<A>      // ping-pong, check connection
<B>      // start calibration of onboard accelerometer
<B123|456|789> // provide values for xccl-calibration
<P1.0|0.1|0.5> // set PID-values for yaw-control
```

`IDENTIFIER` is a single, unique character corresponding to a telecommand. In practice, uppercase letters were used. `START_BYTE` and `STOP_BYTE` were chosen as `<` and `>` respectively. `DELIMITER` may be any character that is not a number or a decimal point. Not all telecommands require parameters, and for some they are optional; The telecommands used for calibration may attach values that are used for calibration. If those are missing, the satellite would continue to determine these values by itself based on its current environment.

5.2.2 Parsing

To check for valid data, the telecommand thread requires an incoming command to have both start- and stop bytes. If this is not the case, the data is discarded. If a valid command has arrived, the identifier is checked and, based on the determined telecommand, possible arguments are parsed using regular expressions:

```
    -?\d+       // integer with optional sign
    -?\d+.\d+   // floating point number with optional sign
```

This ensures that a telecommand carries valid data, since commands expecting floating-point arguments cannot be sent with integers, and vice versa. As a regex-library, *tiny-regex-c* was used for a small memory footprint. [1]

5.3 Telemetry

The main element of the telemetry thread is its FIFO buffer. This buffer is filled with messages that are created by other threads. Once all other threads are suspended, the telemetry thread sends these messages over the UART channel while maintaining the initial order. The FIFO buffer contains structs that encapsulate the following data:

```
struct Telemetry_t { char *msg; size_t len; };
```

where `msg` is a pointer into heap-memory, denoting the start of the message's content with a size of `len` bytes. This message-sharing between threads is one of the few parts of the program that allocates memory. Since the buffer's size is not dynamic, messages that can not be enqueued into a full buffer would be lost, resulting in memory leaks. To mitigate this, an overridden `RODOS::Subscriber` of the telemetry topic is used that calls `free()` on the message of a telemetry frame, if there is no space left in the buffer. As a result, telemetry messages (in this case: housekeeping data) accumulate in the ground segment as follows:

```
[3.231][ATT] h( 23.33 ) dh( 5.97 ) whl duty/rpm/mA( 100 | 523 | 153
    ) batt v( 6.76 ) bus( 10.48 )
[3.231],[ATT],1.22,3.01,23.33,5.97,523,153,6.76,10.48
```

with the former being a human-readable format displaying information important for debugging, while the latter is a more complete, machine-readable `.csv` format for the ground segment to parse. From left to right, the displayed tokens are the mission time in seconds, the thread-ID sending the telemetry and attitude information (`h`: heading [°], `dh`: delta-heading [$^\circ/s$]) as well as health parameters of the satellite.

5.4 Attitude Determination and Control

The ADCS thread runs at a cycle time of 20ms and repeatedly filters the satellite's attitude, followed by setting the flywheel's speed computed by the PID controllers.

5.4.1 Filtering

A complementary filter is used to filter the erroneous data produced by the IMU. The computations necessary for attitude filtering were implemented with the `math.h`

library provided with RODOS. For increased robustness, the following additional modifications were made:

Heading singularity When the satellite reaches a heading θ close to $\pm\pi$, it was observed that the output of the complementary filter resulted in heading values close to zero. This is a consequence of the last step in the complementary filter:

$$\mathrm{YPR} = \alpha \cdot \mathrm{YPR}_{\mathrm{gyr}} + (1 - \alpha) \cdot \mathrm{YPR}_{\mathrm{mag,\ xccl}}$$

where the resulting attitude is a weighted average of both gyroscope and magnetometer/accelerometer measurements. In the case that θ reaches this singularity, both measurements may report values that are close to $\mp\pi$ for one measurement, and $\pm\pi$ for the other. To mitigate this issue, the computation was changed to:

$$\mathrm{YPR} = \begin{cases} \alpha \cdot \mathrm{YPR}_{\mathrm{gyr}} + (1 - \alpha) \cdot \mathrm{YPR}_{\mathrm{mag,\ xccl}}, & |\theta_{\mathrm{gyr}} - \theta_{\mathrm{mag,\ xccl}}| \leq \pi \\ \mathrm{YPR}_{\mathrm{mag,\ xccl}}, & \text{else.} \end{cases}$$

Running median In order to further reduce the variation in the heading-output of the filter, a running median was implemented that takes the median attitude based on θ. For this, a continuously updating ring buffer with the size of $N = 5$ was chosen. The variation decreases with increasing N, albeit reducing the responsiveness of the filter.

5.4.2 Control implementation

The aforementioned cascaded control was implemented such that only one of the outer loops is active. This means that telecommands enabling speed (heading) control, while heading (speed) control is already in progress, are rejected.

5.5 Mission Thread

This thread executes the mission of the satellite – consisting of the following subtasks:

- Handling scanning-related telecommands
- Setting control parameters of the satellite, based on the chosen scan program:
  - Heading position or angular velocity
  - Scan servo duty cycle
- Reading the IR Sensor temperature
- Publishing telemetry with measurements.

The scan programs described in section 4 are implemented to run indefinitely until an interrupt-telecommand is received. The variance of the step program mentioned in section 4.1.2 is thresholded to determine satellite stability. This threshold is variable through telecommands. For each of the samples taken from the IR-Sensor, one telemetry package is published with the heading and duty cycle of the servo motor. This is necessary for the representation in the ground segment.

5.6 Satellite payload

For the satellite payload, the IR sensor TMP006 from Adafruit Industries has been used. An IR sensor is able to measure the temperature of objects without physical contact. It employs a thermophile Sensor to absorb the infrared energy emitted from the object being measured and uses the corresponding change in thermophile voltage to determine the object's temperature.

The interfacing with the sensor was achieved with the I2C protocol. The temperature which was read from the voltage address is in kelvin and had to be converted to degrees celsius.

6 Software - Ground Segment

The ground station aims to receive the telemetry package from the satellite and visualize the sensor's data in real time in several widgets. Moreover, it will be used to send telecommands to the satellite to control the mission. It has been implemented with PyQt5 and can be executed on Windows and Linux. It contains 9 Groupboxes for the data acquisition, telecommands, map, ground track and database. In the background, a call server has been implemented with the telecommunications and cloud communications platform Sinch.

6.1 Ground Station Architecture

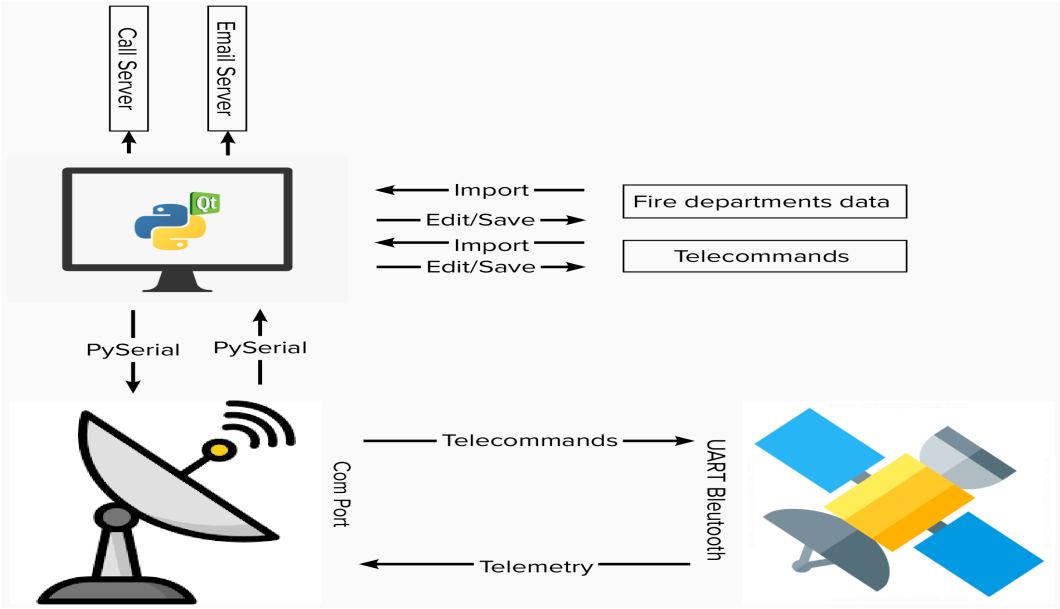


Figure 6: The architecture of the ground station

As illustrated in Figure 6, the connection with the satellite is made with the UART Bluetooth module. The user can search and select from the GUI what port is available and connect the port to the satellite. After the communication with the satellite is established and the calibration process is finished, the GUI starts

the acquisition of the data from the satellite and visualizes every value on the corresponding widget. During the connection with the satellite, the user may import and record data from the database, send e-mails and make a call if necessary during or after the mission. More details will be provided in future sections.

6.2 Ground Station Group-Boxes

As shown in Figure 7, the GUI contains 9 group boxes:

- MenuBar: for the login in the ground station, find the available ports, see the mission reports and make a call.
- Connection: to connect to- and disconnect from the satellite.
- Satellite Battery to show the available voltage in the battery, it was implemented with the class QLCDNumber().
- Satellite Rotation (Longitude in the scenario) will be shown in the compass which has been implemented and drawn with QPainter, the needle rotates after every update and shows values between 0 deg and 360 deg.
- Payload Elevation (Attitude in the Scenario) has been implemented with the same class as the rotation and will be updated only during the mission by receiving the data from the scan thread.
- Telemetry Package and has tabs for:
  - Telemetry overview in the form of a table, a new row will be added on the top of the table by receiving the telemetry package.
  - Mission overview has also the form of the table and will be updated during the mission.
  - Real-time plot graphs for the battery voltage, satellite rotation, payload elevation and IR temperature with Pyqtgraph.
  - Serialized Telemetry(String from com port without parsing).
- Telecommands: to send telecommands to satellite, see the telecommands history and manage the database. Every sent telecommand will be stored on the top of the history table, if the user sends a false telecommand, it will be stored in the history with a red color and status error.
- Ground Track to follow the satellite rotation and Payload elevation and see the fire area.
- Fire departments:
  - Fire departments map: to see the location of departments as a marker on the map which is a Java script tool and has been embedded in the GUI using web-engine.

– Fire department database enables the user to see the coordinates of the department and their contact data, it will be used to see which department must be called by detecting a fire.

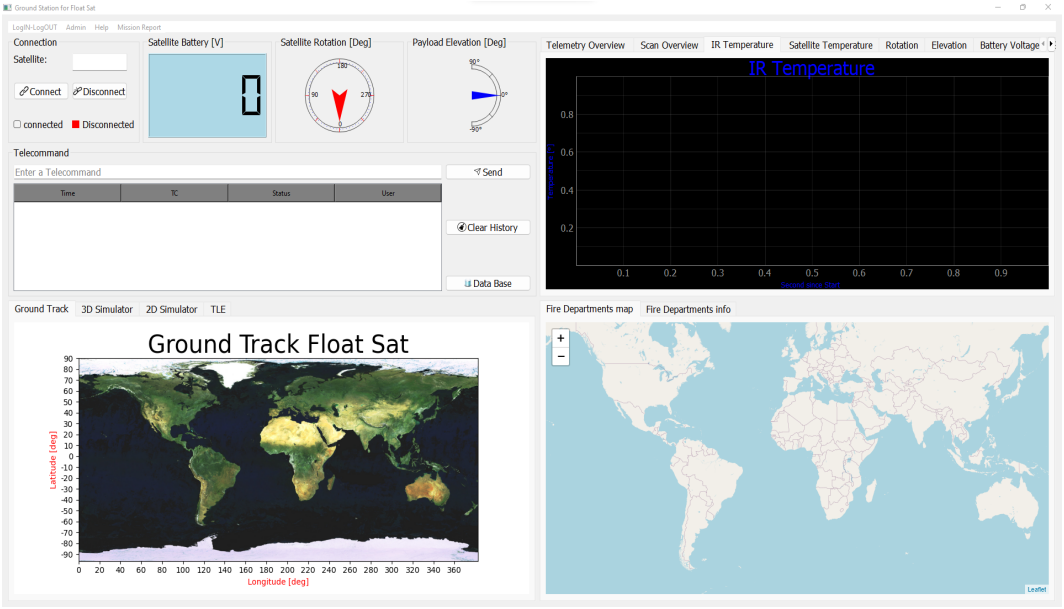


Figure 7: Ground Station GUI with PyQt5

6.3 Data Acquisition and parsing

The GUI will be updated every 100 ms from the PySerial with a variable of type String. 2 threads will be interpreted inside the GUI, telemetry thread([ATT] in the satellite code) and scan thread([SCAN] in the satellite code), the important data of ATT thread are:

- Satellite rotation: heading from the IMU.
- Satellite bus temperature from the IMU.
- Satellite battery value [V].

and the data of the [SCAN] thread which corresponds to the data from the mission are:

- Longitude[deg].
- Latitude [deg].
- IR temperature corresponds to satellite payload data.

The string has the Comma-separated values format, the first step is to remove the suffix (”\r \n”), then to split the string by (,) and to store the values individually inside a list, the first value in the list corresponds to the thread if the Value is [ATT] so the telemetry's overview will be updated if the value is [SCAN] the scan's overview will be updated.

6.4 From mission scenario to ground station

The scenario of the mission is to release the satellite into Earth's orbit so that the satellite changes its position compared with the Earth's rotation during the mission and in order to increase the scan area we got an IR sensor with a bigger opening angle for better security. The rotation of the satellite got from the IMU corresponds to the longitude and the payload elevation got from the servo motor duty cycle corresponds to the latitude. These 2 values (longitude and latitude) will be transformed to a point (x,y) using a coordinate transformation system and plotted on the ground track as a blue cross. As soon as the fire has been detected an alarm inside the GUI will be released and the fire area will be plotted as a red point. A shortest distance calculator algorithm will search for the nearest fire department in the map and changes the color of the fire department in the database to red. In this case, the user can see which fire department must be called and informed. In the Menu Bar in the GUI, the user can open the call server and make a call.

6.5 Call server

For the use of this call server, an account by the company Sinch is required and must be upgraded to get the full features and access to the API. Moreover, the library requests must be imported in Python code. The call server was implemented in a function called make_call() and this function can be called in the MenuBar under the action admin.

```
def make_a_call():
    key = "64daea22-ec03-4b19-a702-b3bd9bd26009"
    secret = "cWzYXk4hXEaRRDCy9VRyxw=="
    fromNumber = "+447520651649"
    to = "+4915901343167"
    locale = "en-US"
    url = "https://calling.api.sinch.com/calling/v1/callouts"
    payload = {
        "method": "ttsCallout",
        "ttsCallout": {
            "cli": fromNumber,
            "destination": {
                "type": "number",
                "endpoint": to
            },
            "locale": locale,
            "text": "  Hello, We are the Team From Float Sat at the
    University of Wurzburg,   we get a fire detection in your area,
    pls check your Mail"
        }
    }
    headers = {"Content-Type": "application/json"}
    response = requests.post(url, json=payload, headers=headers,
   auth=(key, secret))
```

As we can see in the Python code, some key variables must be configured to enable the call server:

Sinch configuration	
Parameter	**Value**
key	got from the dashboard of the account
fromNumber	the number which has been bought from the server Sinch
locale	The language and locale you want to use for the text-to-speech call
URL	Web address of the server
text	the text which will be converted to voice and said if the call will be accepted

7 Review

The team managed to complete in time a good design for our mission and all the sensors were working. The only problem that has been encountered is in regard to the control of the FloatSat. We managed to send commands to the satellite to perform scanning tasks and to require a specific position in its workspace, but in the end, it always evolved towards unstable conditions. The reason for this relies, probably, on a failure in deriving an accurate model of the system. Thus, we realized that the parameters for the controllers that we derived from the simulation performed on Simulink did not give the same results during the testing phase, and we resolved to tune them empirically, using trial and error. The final result, although far from optimal, was a good control of the FloatSat speed and a decent control of the position, considering that, provided good calibration, the satellite was able to reach the desired orientation and keep it for the necessary time to perform the mission. On the downside, the long settling time and, in particular, the large overshoot could have caused a complete loss of control, which, luckily did not happen. In order to obtain a better performance and especially to improve the reliability, a refinement of the control system would have been necessary. Unfortunately, due to the lack of time, we did not have the possibility to improve it. In conclusion, it was possible to carry out the mission in its entirety and the aforementioned problems did not prevent us from obtaining the desired results.

References

[1] kokke. *tiny-regex-c*. `https://github.com/kokke/tiny-regex-c`.

[2] Sergio Montenegro and Frank Dannemann. "RODOS - real time kernel design for dependability". In: *DASIA 2009 - DAta Systems in Aerospace* 669 (2009), p. 66.