Design

Overall System Summary

My project is, as described in analysis, aiming to create a software suite to allow 2-way radio communications with an airborne payload attached to a high altitude balloon. The project will take the form of 3 main pieces of software, there are as follows:

- The ground station software, this will be a standard LoRa receiver with the added capacity to transmit to the payload, providing a remote shell interface and remote request options. This software forwards all received packets to the payload. It will also display received telemetry, 2-way communication results and SSDV to the user.
- The server software responsible for logging all data or forwarding it to habhub, allowing a user to configure a payload and export 2-way packets logged in a database. This should provide a simple web portal allowing users to configure their payload and export telemetry as a CSV file post flight.
- The payload software which will run on the airborne Raspberry Pi and will transmit data to and receive data from the ground, while also operating the camera(s) and GPS module. It should manage the LoRa 2-way system as well as transmitting regular telemetry and image packets.

Notes

Authentication

The system comprises of several ground stations connected to the internet, each equipped with a LoRa transceivers; the airborne payload, also equipped with a LoRa transceiver; and the central server, which is also connected to the internet.

Transmissions are authorised using a key which is used along with the packet when calculating the checksum. This key will be set in the configuration file. This is required in order to ensure that 2-way communication packets are signed with a hash so only one authorised ground station can control the payload. The packets will already have a checksum appended to the end (UKHAS standard is CRC16-CCITT), if I add the key to the string before I calculate the checksum then I can verify if the packet has come from an authorised source when decoding it, as I will calculate the checksum again at the receiving end and add the key there and if they are the same then it means that the packet is intact and from an authorised source. This will be done both on packets sent by the payload and those sent by the ground station, this ensures that nobody can impersonate me or my payload.

There is still one security risk, which is impossible to mitigate, somebody could jam the frequency that I am using by transmitting noise on it at a high power, this is illegal, however.

Defensive Design

The payload software needs to be designed to work continuously without any possibility of software failure over a long period of time. In order to achieve this I shall be using a

technique called defensive programming whereby I shall be using zealous error handling to ensure the program runs correctly even in unforeseen circumstances and does not crash under any preventable circumstance. Of course, if hardware fails the program will be unable to continue functioning but should continue to attempt to so that if the hardware begins to work correctly again the software can continue operating. The payload should be able to run continuously, with no input from me, without a risk of it crashing for an indefinite period of time.

Default Telemetry

Unlike with the UKHAS system where users can define their own telemetry format, my system will have a standard default for simplicity, this will be:

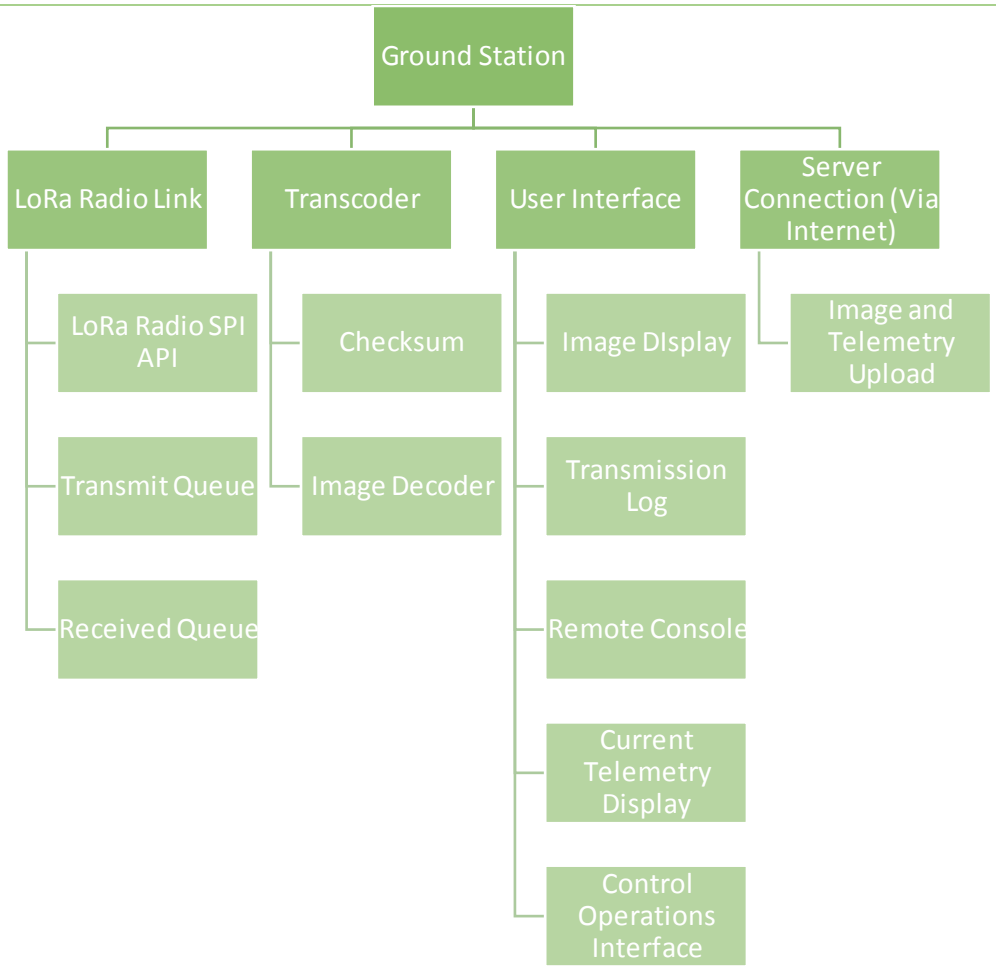
```
$$CALLSIGN,ID,TIME,LAT,LON,ALT,SATS*CSUM\n
```

This is the standard for normal UKHAS flights as noted in research. The checksum will always be CRC-16-CCITT as it is much more reliable than standard XOR checksums sometimes used in the past.

Hierarchy Charts of Each Component

Hierarchy charts for each component of the software are shown below along with a table explaining the function of each level 1 and 2 module.

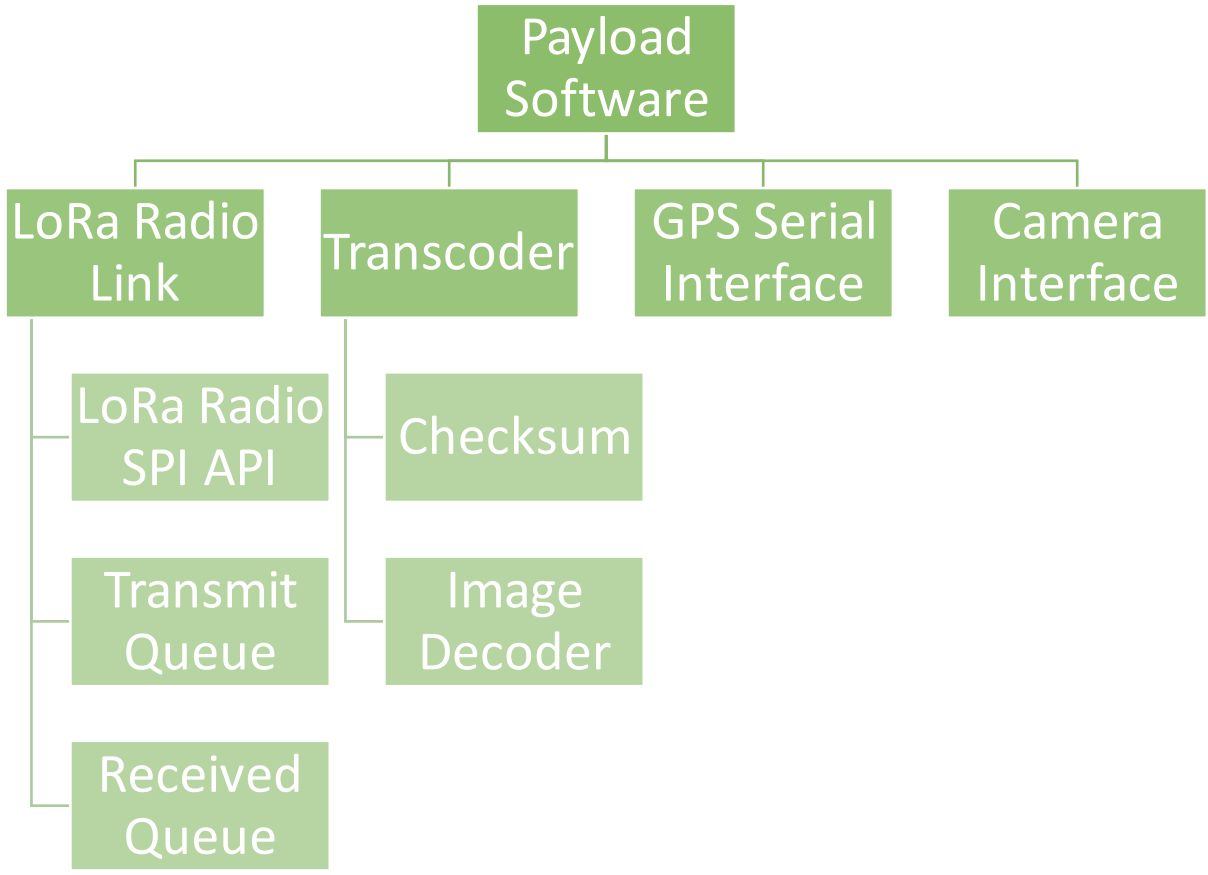
Ground Station



Module	Function
LoRa Radio Link	Manages the LoRa radio interface and transmission and receiving of packets.
LoRa Radio SPI API	The SPI and DIO wrapper that allows access to the radio module's registers and DIO pin flag states. This will be implemented as an encapsulated class, abstracting away most of the complexity of the radio interface from me throughout the rest of the program.
Transmit Queue	A queue of data that is to be transmitted in the next transmit cycle.
Receive Queue	A queue of received packet data that has been read from the radio. Queued here for processing.
Transcoder	Image decoding and checksum encoding.
Checksum	Checksum calculation, this will be a CRC16-CCITT checksum, there will be an authentication key added before the checksum is calculated as described in notes. This will require me to implement a hashing algorithm for CRC-16-CCITT, many pseudocode examples are available online.
Image Decoder	Uses the UKHAS image decoding library to decode image packets as they arrive.
User Interface	Provide the user with an easy to use interface.
Image Display	Display the packets of the current image that have been received from

	the payload. Fills in any missed packets with suitable colour.
Transmission Log	Log of any packets received from the payload and any transmitted, this should be a simple CLI-style scrolling display of text.
Remote Console (Popup)	A console opens if the user actives the remote telnet-style console connection. This will act like a telnet console in that the user can send commands and outputs will be relayed back. Not available if acting as a slave.
Current Data Display	Current display of telemetry, this will be anything that the radio is by default set to transmit regularly and any data that has been requested specifically by the user.
Graphs of Data	Graphs of past data, this should all be on one graph, will show a history of altitude and any other statistics regularly received from the payload.
Control Operations Interface	Allows triggering of sending of packets to the payload, the user should be provided with a set of options to trigger any particular 2-way communications function. Not available if acting as a slave.
Server Connection	Connection via the internet to the central server.
Image and Telemetry Upload	Module which uploads all image packets and telemetry data received to my server for logging and forwarding to the habitat servers. Also upload any 2-way communications packets received from the payload.

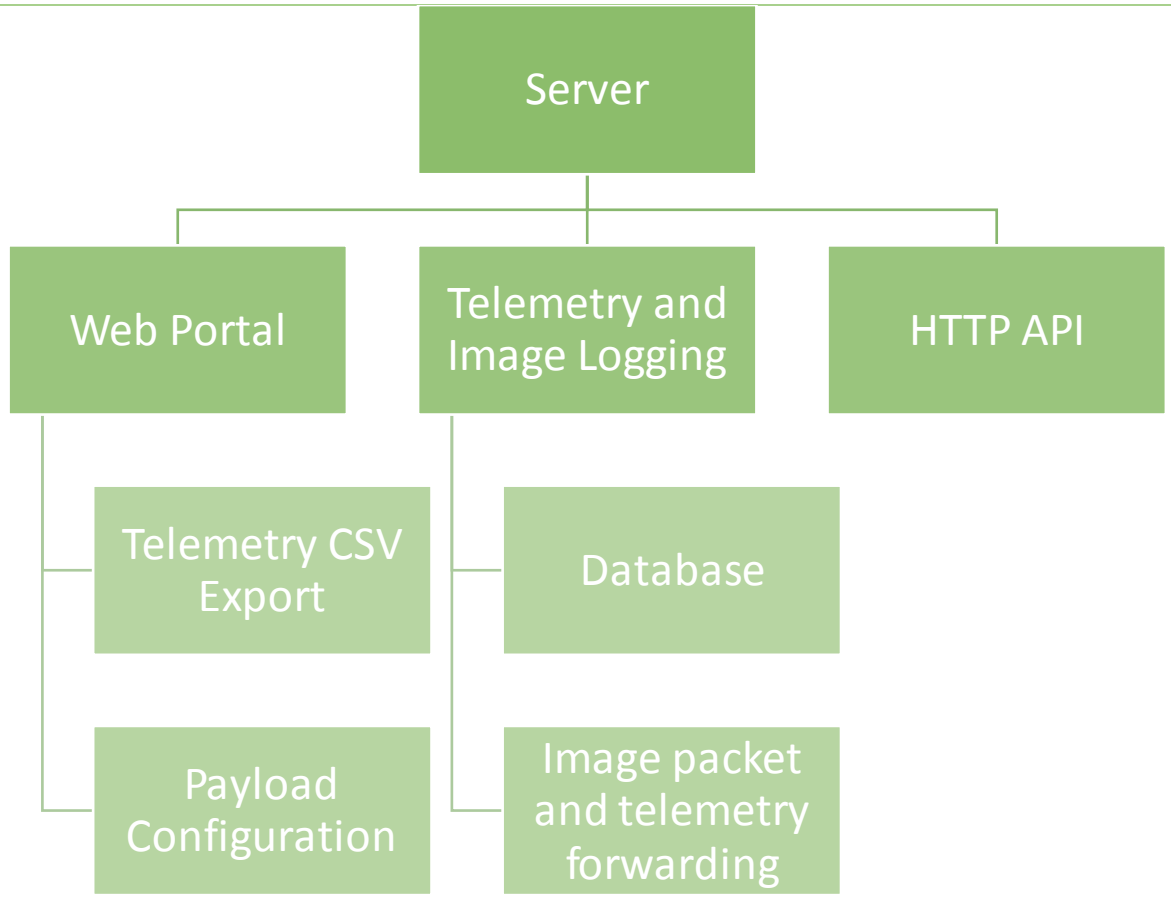
Payload Software



Module	Function
LoRa Radio Link	Manages the LoRa radio interface and transmission and receiving of

	packets.
LoRa Radio SPI API	The SPI and DIO wrapper that allows access to the radio module's registers and DIO pin flag states.
Transmit Queue	A queue of data that is to be transmitted in the next transmit cycle.
Receive Queue	A queue of received packet data that has been read from the radio. Queued here for processing.
Transcoder	Image decoding and checksum encoding.
Checksum	Checksum calculation, this will be a CRC-16-CCITT checksum, there will be an authentication key added before the checksum is calculated as described in notes. This will require me to implement a hashing algorithm for CRC-16-CCITT, many pseudocode examples are available online.
Image Decoder	Uses the UKHAS image decoding library to decode image packets for transmission in 256 byte packets.
GPS Serial Interface	Interfaces with the GPS via a serial connection, reads GGA strings and parses them for latitude, longitude, altitude and number of satellites being tracked.
Camera Interface	Interfaces with the Pi Camera, taking images at a configured interval and storing them on the Pi's micro SD card.

Server Software



Module	Function
HTTP API	Manages the networking between the server and ground station, this

	will just need to be a simple HTTP API allowing the ground station to upload data to the server for logging or forwarding. HTTP PUT will be used here, HTTP GET will be used to allow the ground station to query payload configuration information for the autoconfiguration.
Web Portal	Allows the user to configure their payload pre-flight and export telemetry post-flight as a CSV file.
Telemetry CSV Export	Allows the user to select their payload from a list of payloads registered and download the telemetry from their flight as a CSV file.
Payload Configuration	Allows the user to setup their payload, giving the frequency they will be running on and other LoRa parameters required (i.e. spreading factor, error coding rate and bandwidth) as well as a callsign.
Telemetry Logging	Any telemetry packet received by the ground stations should be logged in a database.
Database	Telemetry packet logs stored here.
Image packet and telemetry forwarding	Forward any image packets and telemetry packets received to the habhub servers.

2-Way Packet Protocol

As noted in analysis, the UKHAS already have a format for telemetry strings; this is demonstrated with an XOR checksum in figure 13. My strings will have to be formatted differently in order to be easily distinguished from UKHAS telemetry strings (which the payload will also transmit regularly). As can be seen in the example all UKHAS strings begin with "$$" (this was initially implemented to make it easier to locate a new packet amongst noise), in order to continue a similar theme, my packets will begin with ">>". The UKHAS strings then send the payload callsign and the packet ID, in the UKHAS protocol, the ID must always increment on successive packets. For my protocol, the callsign will follow, then as with the UKHAS the ID, however, my packet ID will be slightly different, this is discussed below. The next item will be the function or category of the packet; the possibilities are shown in the table below.

Packet Category	Function
0	Command or Status
1	Remote Shell
2	Diagnostic Data
5	Other (This could include status messages.)

NB: there is a gap between 2 and 5 because I may want to add other types of packet in the future.

Following the packet function, will be the actual data which could be the command to execute, the results of a command, the diagnostic data sent in response to a request, etc. and then the checksum is appended on the end, preceded by an asterisk (*). Note that as described in the notes section above, the checksum will be calculated by taking the entire string (from and including the ">>" to the character before the "*"), then appending the unique key generated by the server, then applying a CRC16-CCITT algorithm to this string, this will give a 4 digit hexadecimal number to be appended to the end of the message, followed finally by a newline character, as with the UKHAS protocol. Note that the entire

message must be of length 255 or less, as this is the maximum capacity of the register on the radio which takes the message that is to be sent.

So, overall, this is what my telemetry packet protocol looks like:

`>>CALLSIGN,ID,TYPE,DATA*CHECKSUM\n`

AS noted above, the ID will work differently to how it does in the UKHAS format, where it doesn't matter what the starting value is, just that the value increments for each successive packet, in my Python implementation I simply used the number of seconds since epoch (Unix uses 00:00 on 1st January 1970), however, as I noted in analysis, I will be sending the packets in fixed length windows so the ID will be the number of the packet in the window, starting at 0 up to the length of the window - 1. The length of the windows will be as follows: the payload will transmit 90 packets and the ground station will transmit 10 packets. This is because the ground station's frequency band has a 10% duty cycle limit.

The packet must be of length 255 or less as noted and this means that the data must have length of 255-(14 + callsign length).

The protocol needs to be extremely robust, prioritising the downlink from the radio over the uplink, so in order to do this the default mode will be for the payload to be transmitting, then every 90th packet will be a command packet telling the ground station to begin transmitting. The payload then listens for the packets from the ground station, after it has received 10 packets, or after it has not received a packet for 5 seconds, the payload should return to transmit mode and begin the cycle again. The ground station will only transmit immediately after it has received a command packet telling it that it can begin transmission, at all other times it should remain in receive mode.

In the below table is a description of each kind of packet that can be sent, the first column shows what category they are in (see table on page 22); note that the final column shows what would be the contents of the 'DATA' parameter in the packet description on page 22.

Packet Category	Packet Name	Description	Data Content
0	Transmit	Sent by the payload to the ground station as the final packet of its transmit cycle stating that it is ready to receive 10 packets from the ground.	The mnemonic 'TRA'.
0	Request Data	Sent by the ground station requesting a statistic to be transmitted down.	The name of the statistic requested, this could be IMGNO (number of images) or any other statistic.
0	Image Toggle	Sent by the ground station requesting the toggling of image transmission.	The mnemonic 'IMG'.

0	Reboot	Tells the payload to reboot.	The mnemonic 'RBT'.
1	Command	Sent by the ground station, it is a shell command which is to be executed by the payload. If the command is longer than the allocated length noted above, it should be rejected by the ground station.	The command. Note that if the command contains a comma or an asterisk it should be rejected as this will cause the packet to be parsed incorrectly when received.
1	Response	Sent by the payload after a shell command has been executed, this is the response of the shell command. If the response is longer than the allocated length noted above, then it should be sent in parts.	The command response, or part of the response (if it is being sent in multiple parts). Note that any newline characters in the response should be substituted for another character for transmission and replaced on receive (a control character could be used here).
2	Statistic	Sent by the payload in response to a command (category 0) packet requesting that statistic.	N/S where n is the name of the statistic (see "Request Data" packet) and S is the value of the statistic.

Note that normal telemetry will also be transmitted as will image packets, these will follow the standard UKHAS formats. These will be transmitted by the payload under the following conditions, up to 10 2-way packets are transmitted first during a cycle, if there are fewer than 10 packets to transmit (as there will be most of the time), the payload should transmit telemetry in place of these packets, the next 10 packets will be guaranteed telemetry packets, then the final 70 packets of the 90 packet window will be SSDV image packets, unless image sending has been toggled by a remote command.

Note that SSDV packets are 256 bytes, however, the first byte is always constant (0x55), so in order to meet the 255-byte maximum the payload removes this first byte and the ground station re-adds it before decoding.

Encoding and decoding of SSDV images will be done using the UKHAS SSDV utility as mentioned in analysis (see: https://ukhas.org.uk/guides:ssdv) this is a command line

application so I will use Java's Runtime library to run it, as I will do for taking pictures using the Pi camera.

SPI/DIO API Structure

As previously noted in the analysis section, it will be necessary to develop an API for handling communications with the LoRa radio modules. As noted, these function using SPI and DIO (for information on these protocols see the research section), the SPI is used to modify registers on the radio which control the radio's operation, the DIO is used for flagging particular state changes. My API will be required to be able to write a packet to the appropriate register for sending; read a received packet from the appropriate register; change mode in order to set transmit, receive, standby and sleep modes; and change the radio's settings by editing the appropriate registers.

As described in the linked datasheet for the HopeRF LoRa module, the SPI interface is described as follows: a transfer begins with one byte sent by the master (Raspberry Pi) down the MOSI line (MOSI is SPI Master Out Slave In; MISO is SPI Master In Slave Out) which determines the register address and whether it is to read or write data, the first bit determines whether to write or read, it is 1 for write and 0 for read, then there are 7 bites of address with the most significant bit first. This is followed by the bytes which are to be written if this is a write operation or, if reading, by *n* * zero byte, where *n* is the number of bytes that are to be read.

I shall use the Pi4J SPI module in order to interface with the radio. I will be using a Printed Circuit Board to ensure that my connections are uninterrupted.

Figure 14 – PCB which will be used to ensure reliable wiring.

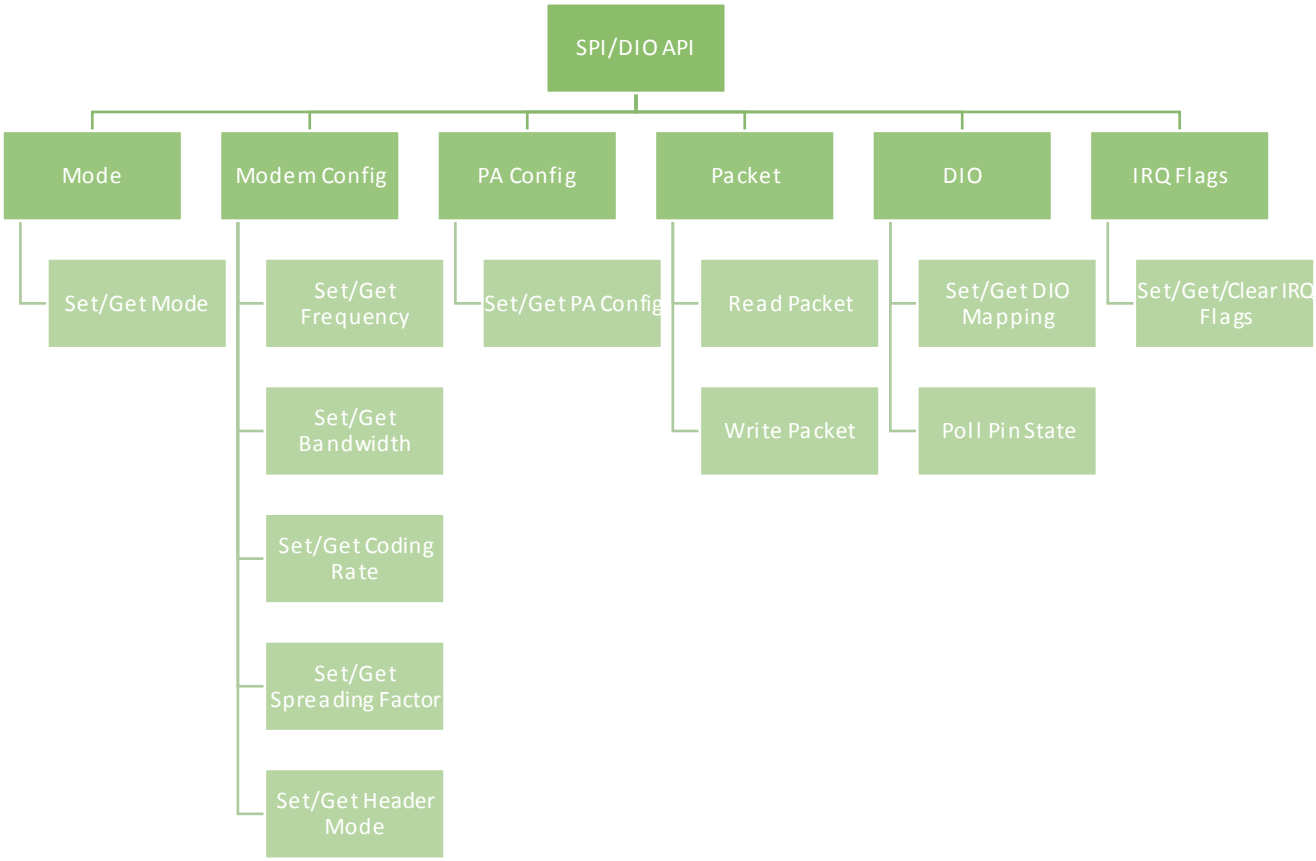
Required Operations

My API will need to complete the following operations:

- Change operation mode (available LoRa modes are standby, sleep, transmit, receive continuously, receive single transmission and channel activity detection). In reality, we will only be using standby, sleep, transmit and receive continuously.
- Get current operation mode.
- Modify and get frequency and other modem settings (bandwidth, error coding rate, header mode and spreading factor).
- Set and get power amplifier (PA) settings, this is output power, see analysis where I discuss power limitations.
- Write packet for transmission.
- Read received packet.
- Read and clear interrupt request flags (these are the flags which are set when a particular interrupt occurs, interrupts could be transmit done, receive done, bad CRC on received packet to name a few).
- Set DIO mapping (this determines which DIO pins will go to high logic level when particular IRQ flags become true).
- Poll state of DIO pins, in order to check whether the assigned flag has been set to true.

I suggest that a standalone, encapsulated LoRa class should be developed to handle most of this functionality. A class which takes in its constructor the frequency and all other modem settings, power settings and DIO mapping, these should have default values if not supplied when calling the constructor. The class should handle all DIO and SPI interactions internally, abstracting away the complexities of the SPI interface away from me throughout most of the programming, and provide methods to complete the above tasks. The below hierarchy chart describes fully the API's requirements.

Structure of API



List of Required Registers

A list of registers that I will be using, their addresses on the LoRa module and their functions is below:

Register Name	Memory Address	Usage
RegFifo	0x00	Read and write access to the LoRa FIFO memory unit. This is where packets will be read from and written to.
RegOpMode	0x01	Change operation mode.
RegFrMsb	0x06	Setting the frequency, this stores the most significant bits of the frequency.
RegFrMid	0x07	Setting the frequency, this stores the middle bits of the frequency.
RegFrLsb	0x08	Setting the frequency, this stores the least significant bits of the frequency.
RegPaConfig	0x09	Configuring the power amplifier and thus output power.
RegFifoAddrPointer	0x0D	Stores the current address of the pointer in the FIFO.
RegFifoTxBaseAddr	0x0E	Stores the base address of where the packet is to be written to in the FIFO for transmission.

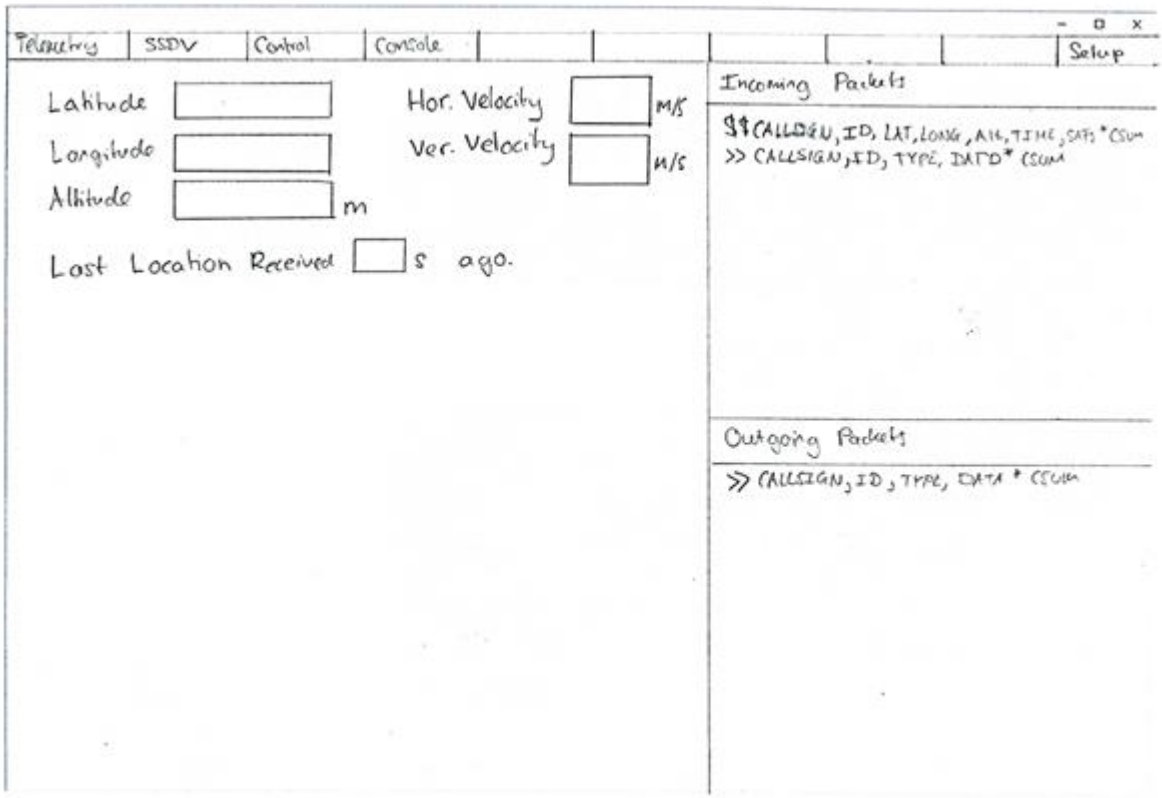
RegFifoRxBaseAddr	0x0F	Stores the base address of where a received packet is stored in the FIFO.
RegFifoRxCurrentAddr	0x10	Start address of last received packet.
RegIrqFlagsMask	0x11	Stores the IRQ flag mask settings, determining which flags can cause an interrupt.
RegIrqFlags	0x12	Stores the current state of the IRQ flags, will need to be polled and cleared.
RegRxNbBytes	0x13	Stores number of bytes received in latest packet.
RegModemConfig1	0x1D	Stores bandwidth, coding rate and header mode.
RegModemConfig2	0x1E	Stores spreading factor.
RegPayloadLength	0x22	Self-explanatory, stores the length of the current packet.
RegDioMapping1	0x40	DIO mapping of pins DIO0 to DIO3.
RegDioMapping2	0x41	DIO mapping of pins DIO4 and DIO5.

These can all be accessed, as described, through the SPI interface.

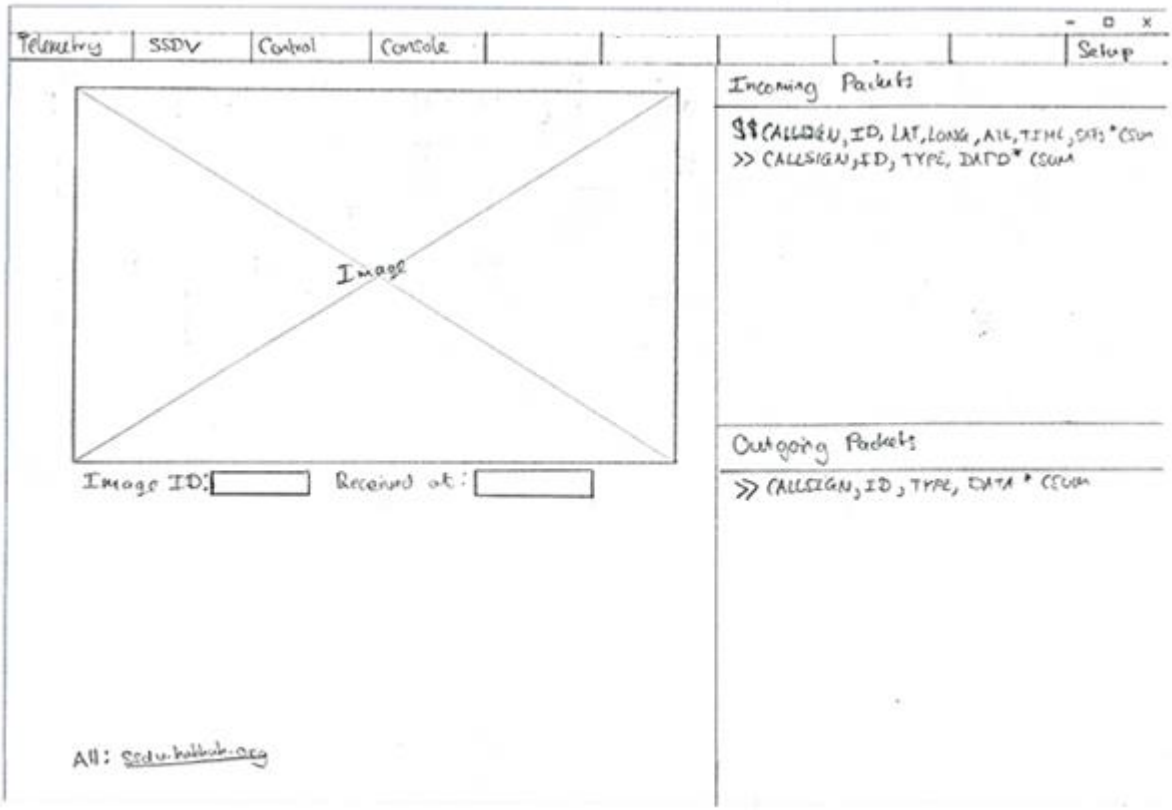
User Interface

My user interface is fairly simple to construct, abstraction of complexity is needed less in this software because of my clients. Some mock-ups of my interface are below with commentary.

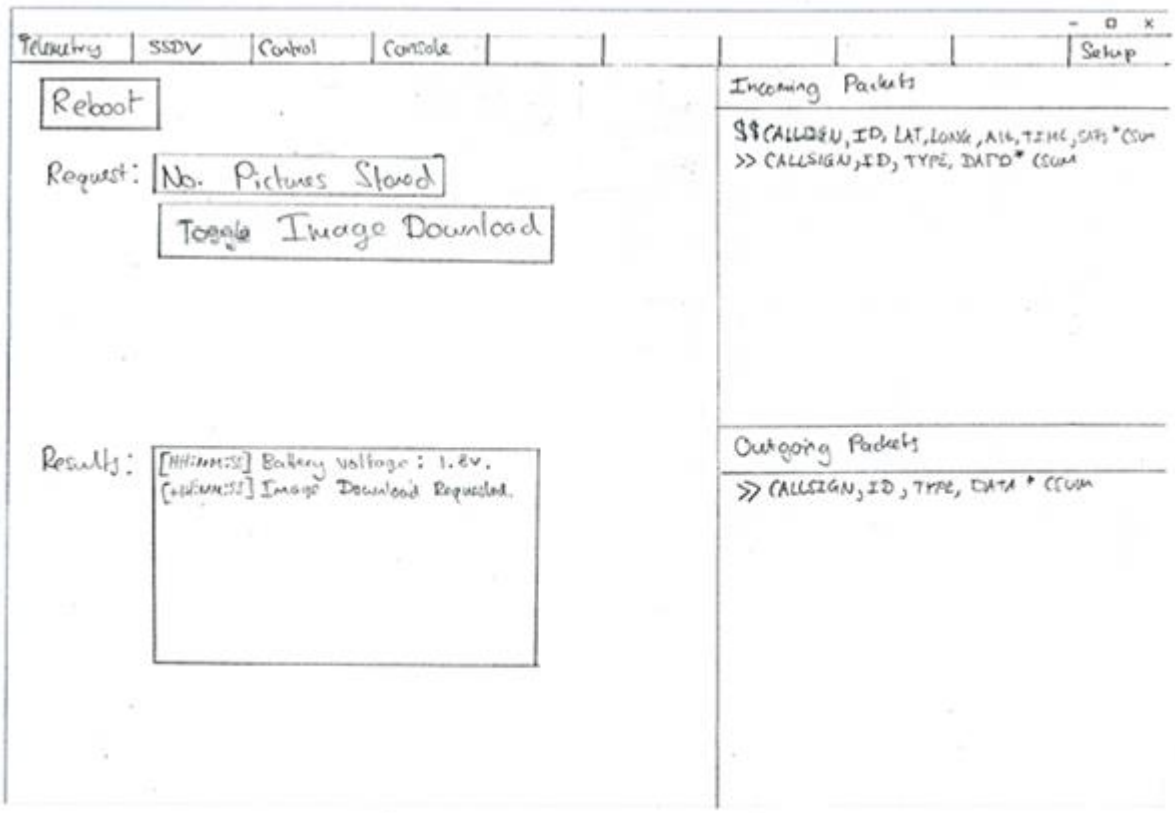
The first image below shows the telemetry screen, which screen you are currently viewing can be identified by looking at the toolbar, this toolbar will have a set of buttons, many of which take you to a specific display. The “Telemetry”, “SSDV”, “Control”, “Console” and “Setup” buttons all take you to a relevant display screen while the “Online” button toggles whether to upload information to the server (green when uploading, red when not), it will also go red when the internet is unreachable. The remote consoles should display only successfully transmitted/received packets; packets with invalid checksums should not be displayed.



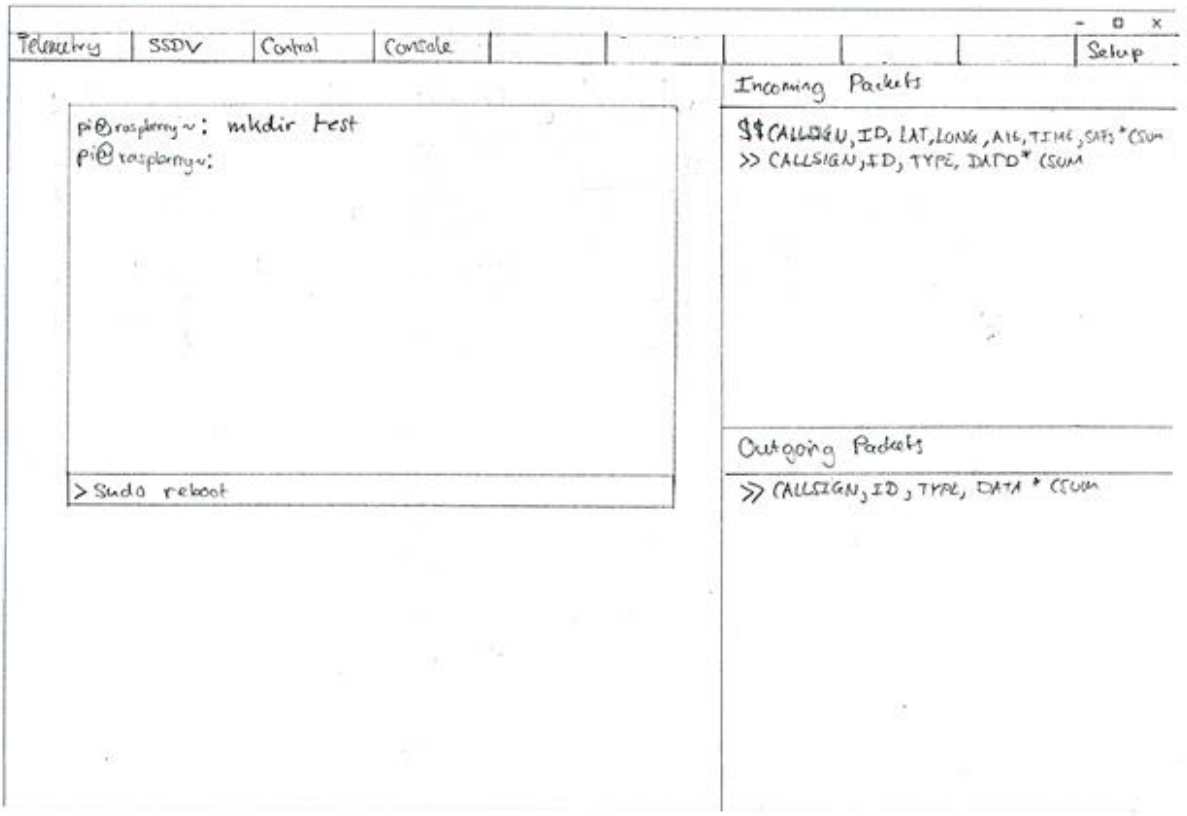
The telemetry screen above shows us the current latitude, longitude, altitude, horizontal and vertical velocities, time since last receive and the list of receivers which picked up the last packet. The white boxes will be text boxes containing non-static information.



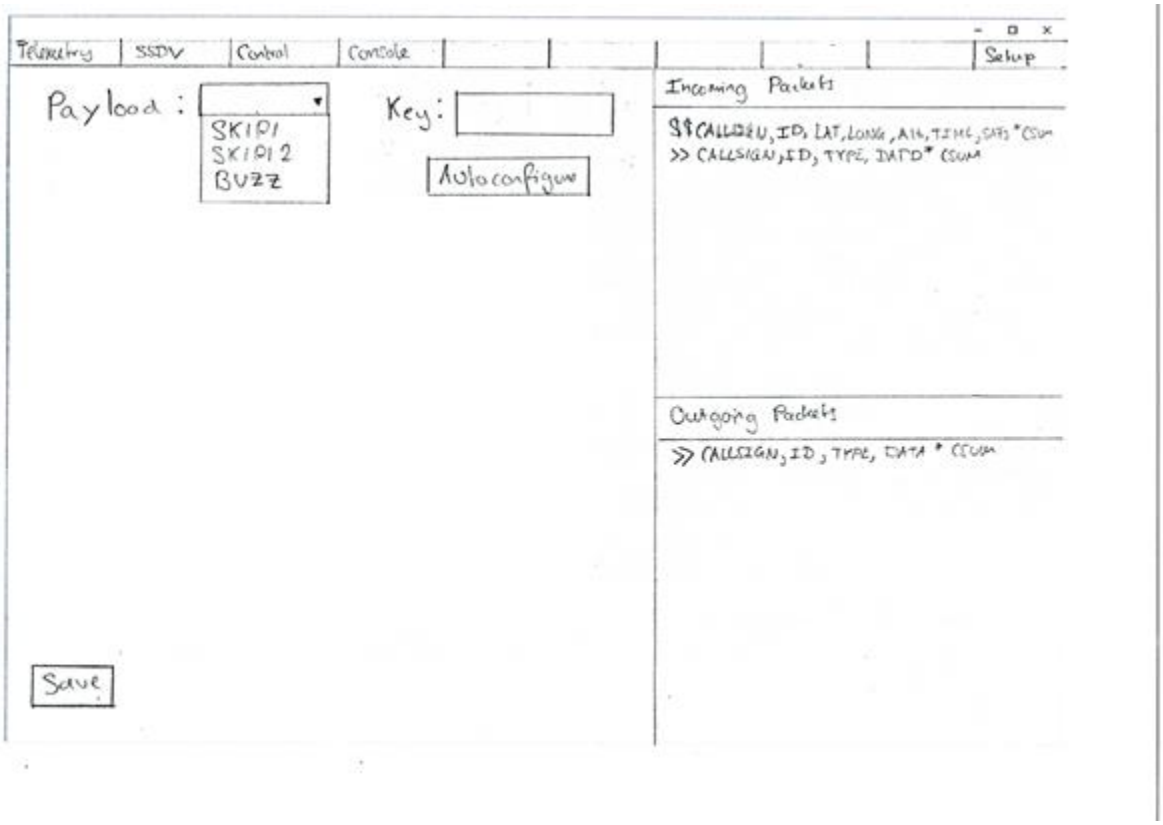
The SSDV screen shows the current image which is being downloaded from the payload in a large box and the 3 previous images in smaller boxes. The image ID and time at which the first packet was received are shown below the large image. Additionally, at the bottom of the page is a hyperlink which takes the user to the ssdv.habhub.org webpage which has a log of all the images from the flight.



The control page has the ability to initiate transmissions to the payload it can request battery voltage, internal temperature, CPU temperature, the number of pictures stored and it can request a download of the latest image which was transmitted in full resolution. The results are shown in a small log at the bottom with timestamps. The user can also request a remote reboot - this is to be used in emergencies.



The console screen is shown above. The remote console has a simple entry box at the bottom with a display of results in a PuTTY-style console. There are a few limitations placed upon the console, if a command takes more than 5 seconds to execute then it will be terminated (this is to prevent programs like “top” being called which provide continuous output), additionally, the command cannot take more than 1 packet to send, the responses can take any number of packets to send, however. This is to prevent issues occurring when the payload receives part of a command to execute but not the complete command, additionally, very few commands will take more a few characters.



This is the configuration screen, allowing the user to select the payload from a list gathered from the server, if they enter their key and click "Autoconfigure" then the settings will be fetched from the server and filled in for them providing the key is correct; if they need to (i.e. they're offline) they can enter the information manually in the config.yml file.

Configuration Design

I will be using YAML (the recursive acronym stands for YAML Ain't Markup Language) for my configuration file on the payload, this is a very simple configuration language. They simply need to be able to set their callsign, key, transmit and receive frequency, transmit and receive bandwidth, spreading factor, error coding rate, whether they are in implicit mode and what output power is being used. I will use a simple Java library called SnakeYAML which will parse the YAML file and output the data into a hash table; I will then extract the data from the hash table into the appropriate variables.

```yaml
#Error coding rate, options are 5, 6, 7 and 8 (they correspond to 4/5, 4/6, 4/7, 4/8).
coding: '5'
#Whether to use implicit headers. This is a boolean.
implicit: false
#Spreading factor, options are integers from 6-12.
sf: 7
#Callsign, must be 6 characters or less, identifies the payload.
callsign: GSCOTF
#Transmission frequency.
freq: 869.525
#Transmission power.
power: 5
#Listening frequency.
listen: 869.85
#Listening bandwidth.
rxbw: 250K
#Authentication key.
key: notakey
#Transmission bandwidth.
txbw: 62K5
```

Figure 15 - Example config.yml file.

Above is an exemplar configuration file, it allows the user to set their settings easily and gives a brief explanation of the options. The file will sit in the same directory as the .jar executable of the main program and will be called *config.yml*.

GPS Serial Interface

The payload software will need to interface with the GPS via a serial port, as noted in the analysis the GPS outputs several different types of location string in a looping sequence. As noted we are only interested in GGA strings, these look like this:

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

This can be parsed to extract data as follows: the items are separated by commas, the first item is the type of string, this is GGA so this is the correct string to be parsing; the second is the time at which the GPS fix was acquired, in this case 123519 refers to 12:35:19; the third is the latitude, this is slightly harder to decode but to convert to degrees you must take the first two numbers, these refer to the number of degrees, then the following numbers refer to the number of minutes, the N shows that this is in the northern hemisphere as opposed to the southern hemisphere, henceforth, the value of 4807.038,N refers to 48° 07.038' North. The same applies to the next number except that it refers to longitude so can be West (W) or East (E), in this case 01131.000 refers to 11° 31.000’ East, note that 3 digits are dedicated to the number of degrees for longitude as opposed to 2 being dedicated to the number of degrees for latitude, this is because latitude has a maximum value of 90, but longitude has a maximum value of 180; the next number is not used but it refers to the fix quality, the following, 08, refers to the number of satellites being tracked; the following number is again unused in my implementation; the next number, 545.4 refers to the altitude so this is 545.4m above sea level. None of the other pieces of data are used in my implementation. It should be noted that when transmitting location data, if the latitude is

North, then it should be transmitted as positive value, if it is South, the it should be negative. For longitude, West is negative and East is positive.

The GPS data will be needed by the 2-way LoRa software while both are running asynchronously. All will be running from the same Java program so I shall simply launch a separate thread which will loop a read operation from the GPS and each time it receives a GGA string, it will parse it and update a variable in the main thread to contain the current location data. The GPS operates at 96,000 baud and as noted uses a serial interface, so I shall use the Pi4J serial library.

Because the GPS outputs data constantly, this data just builds up in the serial buffer on the Pi, as a result, one possible issue which could arise is that because I will be reading GPS strings at a lower rate than the GPS will be outputting them a queue of GPS strings will build up in the buffer and the ones I am reading will become outdated resulting in useless position information being transmitted. In order to fix this, I should simply clear the buffer (this is done by reading and discarding all available bytes) after each GGA string has been read.

Server

As noted previously, the server will allow users to configure their payload on a web portal by providing their callsign, modem settings (frequency, bandwidth, etc.) and it will also allow them to export telemetry as a CSV file. The server will also communicate with the ground stations in order to: share payload details when configuring a ground station; 2-way packets, receive telemetry strings and image packets from the ground stations and log them or forward them to habhub.

I will use the Java HTTPConnection API to handle my HTTP requests, I will be using HTTP PUT to upload information to the server and HTTP GET to acquire payload configuration information when setting up the ground station.

When forwarding to the habhub servers they have a very specific prococol. When uploading telemetry datashould be sent to the server in base 64 format, the url has to be http://habitat.habhub.org/habitat/_design/payload_telemetry/_update/add_listener/ followed by the sha256 hash of the base 64 data. The HTTP headers should be JSON format as below:

{"Accept" : "application/json", "Content-Type" : "application/json", "charsets" : "utf-8"}.

The data sent should a json string containing the base 64 string, the receiver name and the time the packet was decoded. The request type will be HTTP PUT.

When uploading SSDV a PUT request will be sent to http://ssdv.habhub.org/api/v0/packets where the data includes the raw base 64 of the data, the receiver name and the time that the packet was received. The headers should be JSON, identical to that above for telemetry. The request method will be HTTP POST.

Needless to say the character encoding used will be UTF-8.

Database Design

On the server I will make use of a database to store information about each payload. Each payload will need to have its callsign, settings and encryption key stored, as well as a log of 2-way packets sent by it with the time that that packet was decoded. My database will, of course, be in Third Normal Form (3NF). Below are tables demonstrating the two entities in my database and their attributes.

Database Table: Payload

Stores information about a payload. Note that this table has a composite key because payloads can have the same callsign so to distinguish these the time and date created is also part of the key.

Name	Type	Size	Purpose	Example
payload_id	String	8 bytes	Uniquely identifies a tuple. This is an auto incrementing value generated dynamically by the database each time a value is added. Primary key.	42
callsign	String	10 bytes	Identifies each payload and thus each tuple in this table.	SKIPI2
created_at	Date-time	-	Date and time at which this payload was configured.	<hash>
txfreq	Decimal	9,6 (see purpose)	(Length 9, 6 of which are after decimal place). Stores the frequency at which the payload radio will transmit.	868.850000
rxfreq	Decimal	9,6 (see purpose)	(Length 9, 6 of which are after decimal place). Stores the frequency that the payload radio will listen on.	868.850000
txbw	Integer	2 bytes	Stores the bandwidth that the payload radio will be using to transmit.	250
rxbw	Integer	2 bytes	Stores the bandwidth that the payload radio will listen on.	250
sf	Integer	1 byte	Stores the spreading factor that the payload radio will be using.	7
coding	Integer	1 byte	Stores the error coding rate that the payload radio will be using.	5 (would correspond to 4/5)
explicit	Boolean	-	Stores whether the payload radio is using explicit header mode.	TRUE
key	String	16 bytes	Stores the key that is used to determine authenticity of a packet. This is generated by hashing the callsign and the created_at hash.	52a05b30ffbc7de3

Database Table: Packet

Stores a log of two-way communication packets transmitted to or received by a payload during a flight.

Name	Type	Size	Purpose	Example
packet_id	Integer	8 bytes	Uniquely identifies a tuple. This is an auto incrementing value generated dynamically by the database each time a value is added. Primary key.	42
payload_id	Integer	8 bytes	Identifies which payload this packet was transmitted by.	42
time	Date-time	-	Database generated hash storing the date and time of when this particular packet entry was first received on the server.	<hash>
raw	String	256 bytes	Stores the raw string of the packet.	$$SKIPI,0,123,456,789,0*55/n

Entity Relationship Diagram

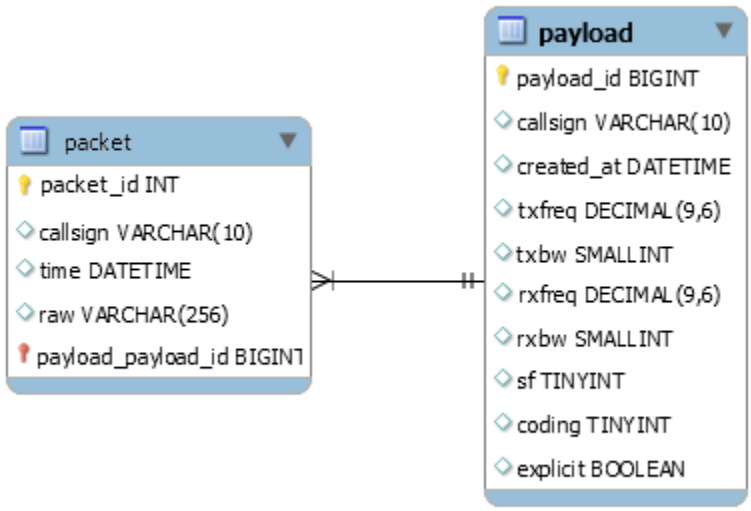


Figure 16 - Entity relationship diagram for simple 2-table database.

The above ER diagram also includes the MySQL data types of each attribute; I will be using MySQL for my database. As you can see, there is a 'one to many' relationship between the payload and telemetry.

Interrupt Driven Cycle

As noted previously, the payload will follow a rigid cycle of transmitting 90 packets and then receiving (up to) 10 while the ground station will need to continuously receive unless told to do otherwise. In order to achieve this, I intend to implement an interrupt-driven design whereby the ground station receives continually unless a transmit flag is set to true, which

will be done when a transmit packet has been received. When this flag is set to true, the ground station should immediately terminate receiving and immediately hand control over to the transmission code allowing queued packets to be sent to the radio via SPI for transmission. The need for speed here is because the payload will timeout listening for packets after a few seconds.

Image Taking

The payload will need to take images at regular intervals, as I will, by default, be using a Raspberry Pi camera, I shall use raspistill, a command-line application, to take the images, I shall use the Runtime Java library to execute *raspistill -o <outputfilename>.jpg*. When images are transmitted, however, they are not transmitted at full resolution, so I will need to convert them to a smaller size, this could be done easily with Java's image handling libraries, however, an easier solution would be to use ImageMagick, a free command-line library which will resize images among other things.

Algorithms

In this section I detail the import algorithms involved in my project as pseudocode, flow charts and/or plain English.

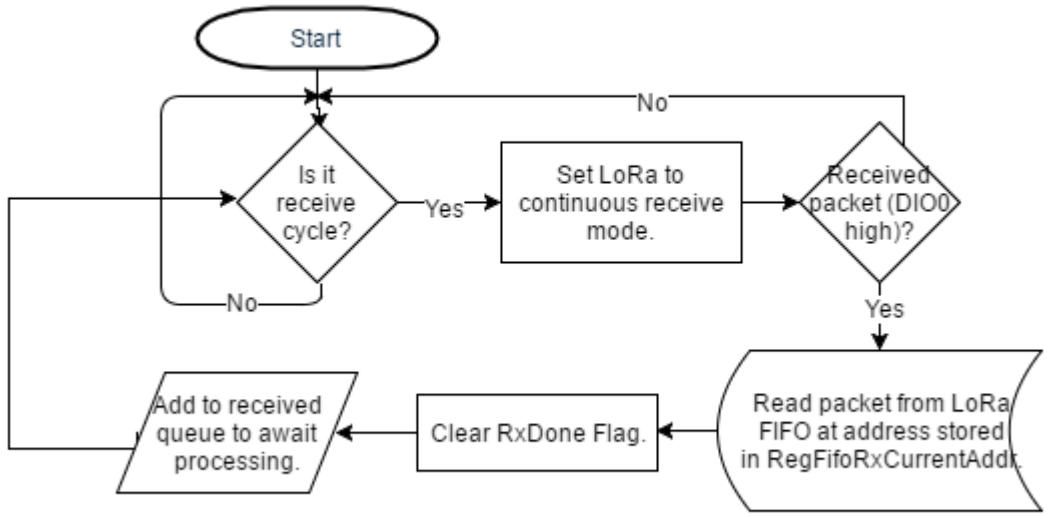
Receiving

The below flowchart and pseudocode denote how a packet is received when the station is in receive mode.

Pseudocode:

```
While True:
     If receive cycle:
          Set LoRa radio mode to continuous receive mode;
          If Packet is waiting (DIO0 HIGH):
               Packet <- read from FIFO (address of packet in
RegFifoRxCurrentAddr);
               Clear RxDone IRQ flag;
               Add packet to received packet queue to await
          processing;
          End;
     End;
End;
```

Flowchart:



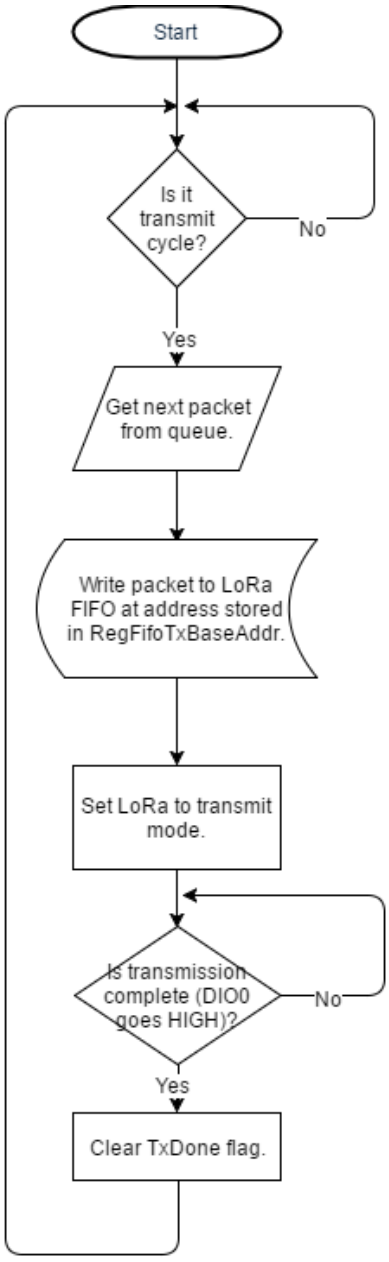
Transmission

The below flowchart and pseudocode denote how a packet is actually transmitted.

Pseudocode:

```
While True:
     If transmit cycle:
          If transmit queue is not empty:
               Packet <- next packet from queue;
               Write packet to FIFO on radio using SPI
interface;
               Set LoRa radio mode to transmit mode;
               While transmission is not complete (DIO0 not
HIGH):
                    Wait;
               End;
               Clear TxDone IRQ flag;
          End;
     End;
End;
```

Flowchart:



Modification of Specific Registers Using SPI Interface

The pseudocode below gives a demonstration of the general method for writing a register:

```
func writeRegister(byte addr, bytearray data):
      toSend = new bytearray(length = data.length + 1);
      toSend[0] = addr & 0x80;
      toSend[1…] = data;
      spi.write(data);
endFunc;
```

The pseudocode below gives a demonstration of the general method of reading a register:

```
func  readRegister(byte addr, int numberOfBytes):
     toSend = new bytearray(length=(1+numberOfBytes));
     data = spi.read(toSend); //returns a bytearray.
     return data;
endFunc;
```

Some values require more complex setting and getting methods due to, for example, data being split between multiple registers. One such situation is setting frequency, the most significant, mid significance and least significant 8 bits of the frequency are stored in separate registers. An algorithm for this is shown below:

```
func setFrequency(double frequency):
     check opmode is sleep or standby, if not exit;
     int freq = frequency * 2^14;
     writeRegister(0x06, (freq >> 16) & 0xFF);
     writeRegister(0x07, (freq >> 8) & 0xFF);
     writeRegister(0x08, freq * 0xFF);
endFunc;
```

CRC-16-CCITT

The pseudocode below shows the hashing algorithm for the calculation of a CRC-16-CCITT checksum for a byte array representing a string, returning a 4 digit hexadecimal number as a string.

```
func calcCsum(bytearray data):
     int crc = 0xFFFF;
     int poly = 0x1021;
     for each byte as b in data:
          for i in 0 to 8:
               boolean bt = ((b >> (7-i) & 1) == 1;
               boolean c15 = ((crc >> 16 & 1) == 1;
               crc << 1;
               if (c15 ^ bt):
                    crc ^= poly
               End;
          End;
```

```
    End;
    crc = crc & 0xFFFF;
    return toHexString(crc);
endFunc;
```

Reading From the GPS

Loop to read from the GPS and clear the buffer after each successfully received GGA string.

```
String received = "";
While true:
    if serial.hasBytes():
        byte b = serial.read();
        received += toASCIIChar(b);
        if c == "\n": //"\n" is a newline character.
            If received.substring(0,6) == "$GNGGA":
                generateTelemetry(received);
                serial.clearBuffer();
            received = "";
        End;
    End;
End;
```

Generating Telemetry Strings

Function called when a new GPS GGA string is received to generate a telemetry sentence ready for transmission.

```
func generateTelemetry(String gpsData, String callsign):
    Stringarray data = gpsData.split(",");
    If length of data >9:
        String lat = data[2];
        String lon = data[4];
        If data[3] == "S":
            lat = "-" + lat;
        End;
        if data[5] = "W":
```

```
                lon = "-" + lon;
            End;
            data[1].substring(0, 2) + ":" + data[1].substring(2,
4) + ":" + data[1].substring(4, 6);
            String telem = callsign + "," +
String.valueOf(getUnixTime()) + "," + time + "," + lat + "," +
lon + "," + data[9] + "," + data[7];
            String csum = calcCsum(telem);
            telem  = "$$" + telem + "*" + csum + "\n";
            return telem;
        End;
        return nothing;
endFunc;
```

Image Loop

Simple loop for image capture every 30 seconds.

```
While true:
    Take picture using command line operation 'raspistill -o
"filename.jpg"';
    Make a low resolution copy of this image using
ImageMagick;
    Encode this image for transmission with fsphil's SSDV
utility store as out.bin;
    Sleep(30s);
End;
```

Whole System Data Flow Diagram

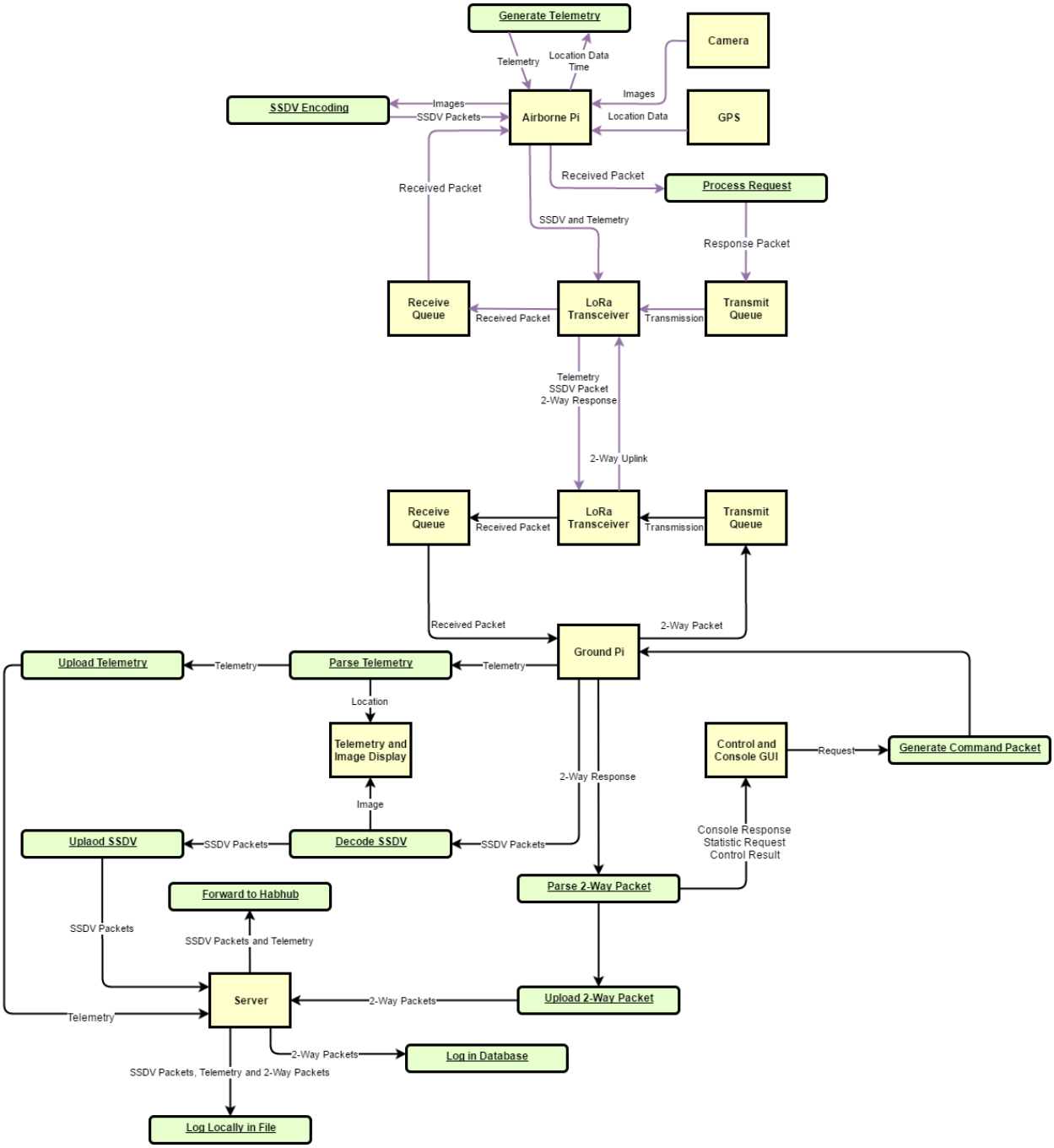


Figure 17 - Data Flow Diagram for Proposed System

Validation Required

There are very few instances in the program where the user has to provide direct input to the program with most processes being automated. However, there are a few sections where the user's input will need to be validated. As noted formerly, user input to the console should be validated to ensure that it would not take up more than 1 packet. Additionally, command packets should not contain an asterisk or a comma as this will cause the parser to fail on the receiving end. Also, command response packets should not contain newlines, instead, newlines should be substituted for an unused control character when

sending and replaced on receiving, this is because the newline will also cause parsing to fail as a newline denotes the end of a packet. Of course, as noted before, packet sizes will need to be limited to 255 characters and thus long console commands will not be possible.

Further validation required would be ensuring that all characters transmitted by either ground station or payload are ASCII characters. Additionally, packets received should have their checksum checked both for integrity and authenticity, as described in the notes section at the beginning of the design. Each packet has a CRC checksum automatically applied by the radio, however, this will not be used, instead we will use the CRC-16-CCITT checksum that we will have appended to each packet; this is because the LoRa CRC checksum has been shown to be unreliable by testing by other members of the UKHAS and because our checksum will take into account the key and thus also check for authenticity.

Test Strategy

I will be testing the software mainly with standard black box testing, I will test that the software works correctly in a controlled environment where the payload and ground station are separated physically and I have no control over the payload except through the ground station. I will test the 2-way console, remote requests, telemetry transmission, SSDV transmission and I will test the integrity of the location data acquired from the GPS. Additionally, I will need to test the authentication system that I have discussed. I will of course, also test carefully where user input is required to ensure that boundary, erroneous and normal data are dealt with correctly. I will also need to ensure that data is correctly uploaded to the server and that the server handles this data correctly, I will also need to be sure that SQL injection attacks are not possible on my website, this is easily achieved with correct use of prepared statements and escaped strings. I will also need to test the functionality of my LoRa API, ensuring that when, for example, I set a frequency, the radio does indeed get set to that frequency.

However, while the above tests are the core of my testing, I will also be testing the ability of my software to perform in a real flight. In order to do this, I will first do 'endurance' testing, whereby I will leave the payload software running for a long period of time and check that it is still working correctly afterwards, this is important because once the payload is airborne I have little ability to fix problems that occur. I will then be flying two test flights to test the 2-way communications in a real flight. I will be documenting these flights with a short video. The first flight is expected to have some issues which will hopefully be rectified before the second flight. I have received permission from the Civil Aviation Authority to conduct these flights on Sunday the 12th and Tuesday the 14th of February.

To summarise, I will be testing:

1. **Integrity of transmitted data:** I will need to ensure that data transmitted is valid, up-to-date and encoded correctly. Telemetry should be up-to-date, with valid location fix and encoded using ISO-8859-1, SSDV should be encoded with fsphil's library and have the first byte (always 0x55) removed.
2. **Data displays on ground station:** I will need to test the received data is handled correctly by the ground station. Telemetry should have its checksum checked and

then if valid, the relevant data should be displayed as latitude (degrees), longitude (degrees) and alaitude on the telemetry display tab. The currently transmitting image should be displayed on the SSDV tab, with missing packets showing up as block colour. Two-way data should be checked for valid checksum and then console results should be displayed in the console and request results should be shown in the transmission log. Additionally, all outgoing and incoming packets should be displayed on the tx/rx logs on the left of the screen.

3. **Authentication:** I will need to ensure that the authentication system correctly identifies only the authorised receivers. I will test this simply by setting the payload and the ground station to use different keys and ensuring that they ignore eachother's two way communications, while otherwise they should handle eachother's two way communications as required.
4. **Remote control:** I will test all possible remote control operations with a ground station and payload set to use the same key. I will send remote console commands, I will request a remote reboot, I will toggle image sending and I will request the number of pictures stored. All these should produce the correct results.
5. **Web Server:** The web server needs to be shown to be able to handle invalid input on the payload configuration page, rejecting inputs which would simply not work. Additionally, it needs to be ensured that the server handles HTTP requests from the ground station correctly, forwarding telemetry to the habhub servers and logging 2-way packets in my database.
6. **LoRa API:** I need to ensure that my LoRa API works as required; it should be able to complete all the functions that are set out in the design. I will need to ensure that it operates as expected, in that, when I set it to run at a particular frequency, it does indeed run at that frequency, I will also check bandwidth is correct. This is to ensure that my LoRa API is without error and that my algorithms for register modification are indeed correct. As a final test of this, I will attempt to receive transmissions from LoRa receiver software (written in C, known to function correctly, thoroughly tested by the HAB community) to ensure that my radio is indeed running on the settings that my API supposedly set it to.