Teacher Standardisation

Spring 2018

A-level Computer Science (7517)

Booklet 2

Contents

Analysis

Project Background

In the last few months, I have been getting into an advanced computing and electronic engineering hobby: high altitude ballooning. This involves sending a balloon with a payload attached containing a microcontroller (in my case, a Raspberry Pi Zero), a low power UHF radio, a GPS module and a camera. The Pi is programmed to take pictures at regular intervals and transmit both the images and the GPS data (telemetry) down in order to aid tracking and recovery of the payload. The software I developed and used for my first 3 flights can be seen here: https://github.com/Abrasam/SKIPI-Launch-1 and here: https://github.com/Abrasam/SKIPI2. Below is a data-flow diagram for the current system.

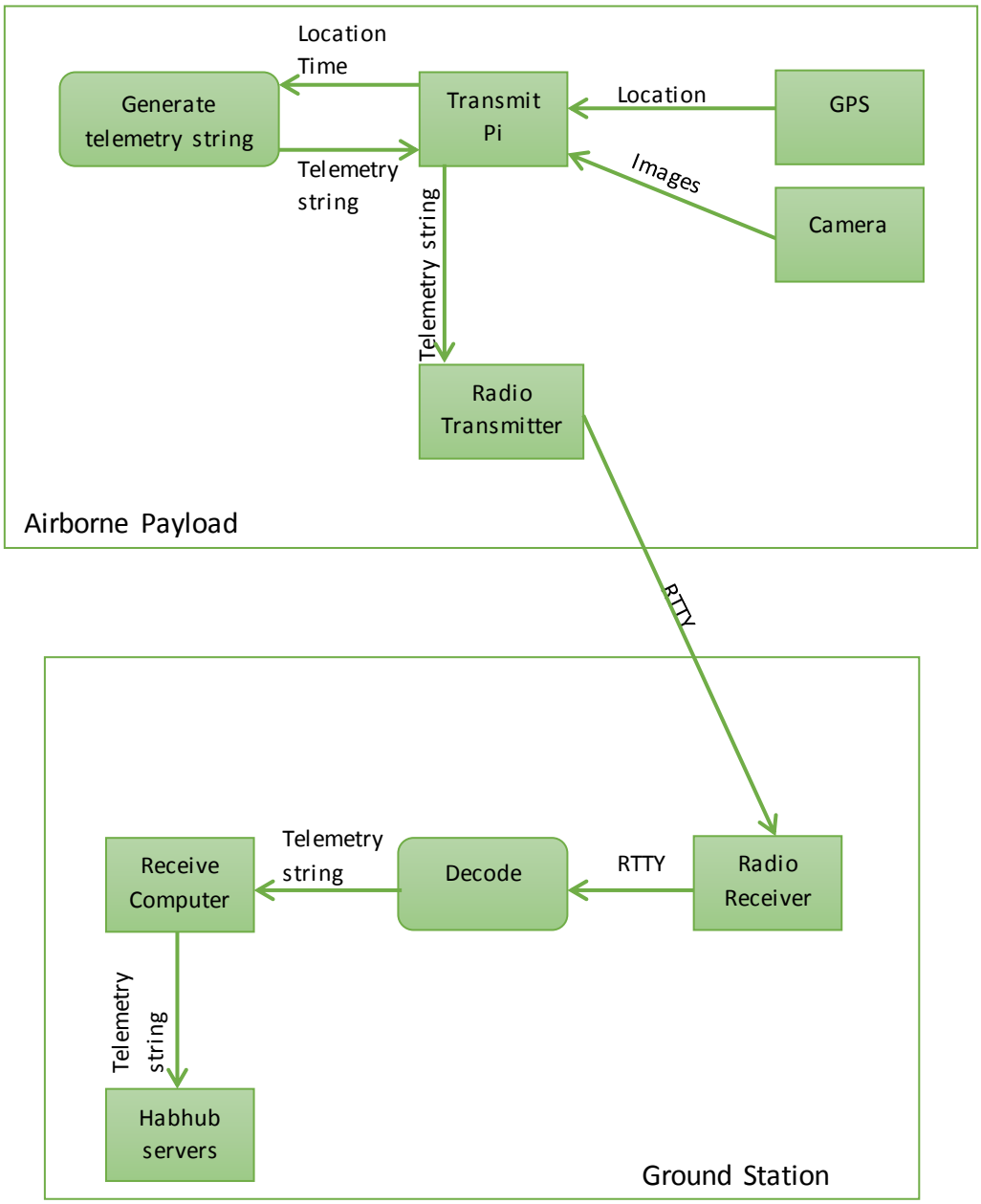


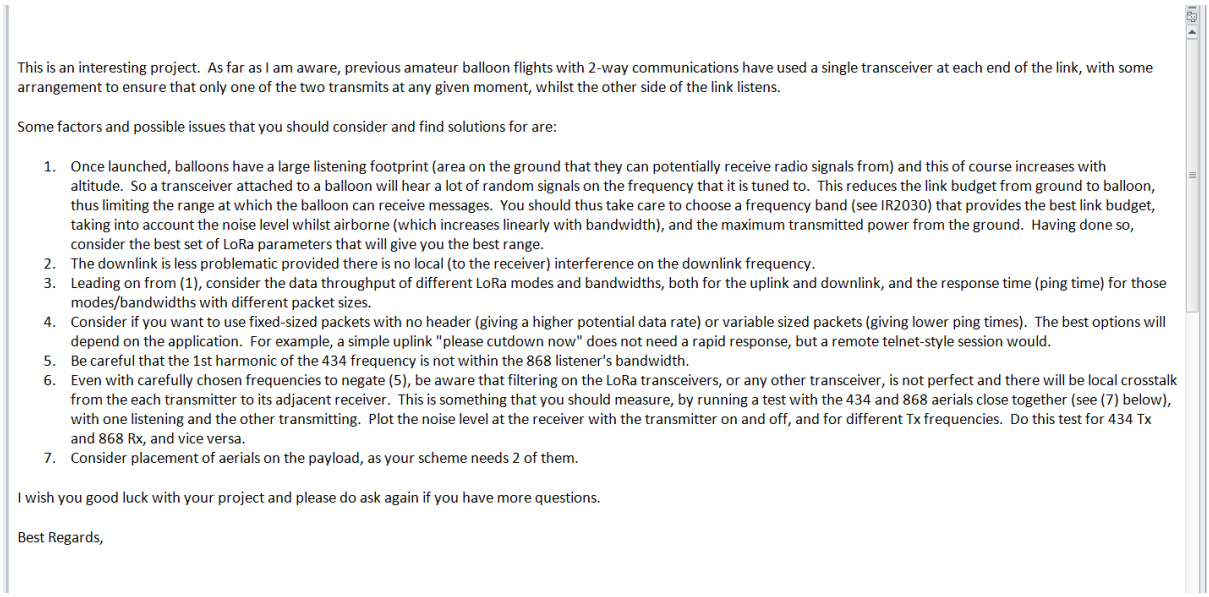
Figure 1 - DFD diagram for the current system, note the lone one-way RTTY link between the airborne payload and the ground-based station; there is no capacity for uplink.

Project Outline

As described previously, there is only one-way communication occurring between the payload and the receiver and if something goes wrong, then there is very little that can be done about it, furthermore, it means that the information which comes down from the payload is very specific and limited and all diagnostic data must be predefined when creating the payload's telemetry format. For my project, I want to create a prototype piece of software which allows two-way communication with and control of an airborne high altitude balloon payload for use by UKHAS (UK High Altitude Society) members, myself included. I have used two methods of communicating with the receiver on the ground, they are described in the research section. My clients for this project are members of the UKHAS who fly High Altitude Balloons (HABs) as a hobby. We have several pieces of software which are used by members, and this software is intended to be an addition to our suite of HAB utilities. I intend to use a Raspberry Pi as the microcontroller for my payload, I could use an Arduino, however, the Raspberry Pi camera module makes the Pi an attractive piece of hardware for HAB, as well, it gives me more freedom of programming language.

Client

As noted above, my intended clients are members of the UKHAS who will be the principle users of this software. One member of the UKHAS, ı, has agreed to answer questions via email and provide technical advice with regards to the hardware implementation.



This is an interesting project. As far as I am aware, previous amateur balloon flights with 2-way communications have used a single transceiver at each end of the link, with some arrangement to ensure that only one of the two transmits at any given moment, whilst the other side of the link listens.

Some factors and possible issues that you should consider and find solutions for are:

1. Once launched, balloons have a large listening footprint (area on the ground that they can potentially receive radio signals from) and this of course increases with altitude. So a transceiver attached to a balloon will hear a lot of random signals on the frequency that it is tuned to. This reduces the link budget from ground to balloon, thus limiting the range at which the balloon can receive messages. You should thus take care to choose a frequency band (see IR2030) that provides the best link budget, taking into account the noise level whilst airborne (which increases linearly with bandwidth), and the maximum transmitted power from the ground. Having done so, consider the best set of LoRa parameters that will give you the best range.
2. The downlink is less problematic provided there is no local (to the receiver) interference on the downlink frequency.
3. Leading on from (1), consider the data throughput of different LoRa modes and bandwidths, both for the uplink and downlink, and the response time (ping time) for those modes/bandwidths with different packet sizes.
4. Consider if you want to use fixed-sized packets with no header (giving a higher potential data rate) or variable sized packets (giving lower ping times). The best options will depend on the application. For example, a simple uplink "please cutdown now" does not need a rapid response, but a remote telnet-style session would.
5. Be careful that the 1st harmonic of the 434 frequency is not within the 868 listener's bandwidth.
6. Even with carefully chosen frequencies to negate (5), be aware that filtering on the LoRa transceivers, or any other transceiver, is not perfect and there will be local crosstalk from the each transmitter to its adjacent receiver. This is something that you should measure, by running a test with the 434 and 868 aerials close together (see (7) below), with one listening and the other transmitting. Plot the noise level at the receiver with the transmitter on and off, and for different Tx frequencies. Do this test for 434 Tx and 868 Rx, and vice versa.
7. Consider placement of aerials on the payload, as your scheme needs 2 of them.

I wish you good luck with your project and please do ask again if you have more questions.

Best Regards,

Figure 2 - Email from , member of the UKHAS, amateur radio operator and embedded systems programmer. IR2030 is the document published by Ofcom noting what radio frequency bands are available for license-free use.

comments on the fact that the payload will have a large listening footprint and as such will be receiving lots of other transmissions, meaning that our transmissions may require higher power in order to be received over the noise. I will discuss potential frequencies to use which are license-exempt in the next section. also suggests that I use a LoRa transceiver (see research for more information) and that I choose my parameters wisely to maximise range. points out that 434MHz and 868MHz are harmonically related (these

are the two frequencies frequently used by HAB enthusiasts and are the two frequencies which LoRa modules operate on). I will discuss points more in research.

Research

Radio Communications

The first method of communication I have used is RTTY or Radio Teletype which is an old protocol initially developed for the teleprinter, which works by simply shifting frequency up and down to correspond to binary 1 and binary 0, this is done by applying a small voltage to one of the pins on the radio. These voltage changes must be timed accurately, but as the Pi doesn't have a real time OS, the best way to achieve this was using the Pi's RS-232 (a standard for asynchronous serial communications) serial interface (and thus connecting the Pi's Tx (see fig. 3) pin to the radio's pin). This runs at 75-100 baud robustly and could be pushed to 300 but is then significantly susceptible to interference, so this means a maximum usable downlink bitrate of 300bps, as I have only one bit encoded per state change. This rate is unfeasible for two way communications and additionally, automatic detection and decoding of RTTY is difficult to achieve reliably, even when receiving transmissions during a normal flight I typically receive errors on at least 20% of packets, this is significantly too high to be useful for 2-way communications, particularly of telnet style communication is desired.

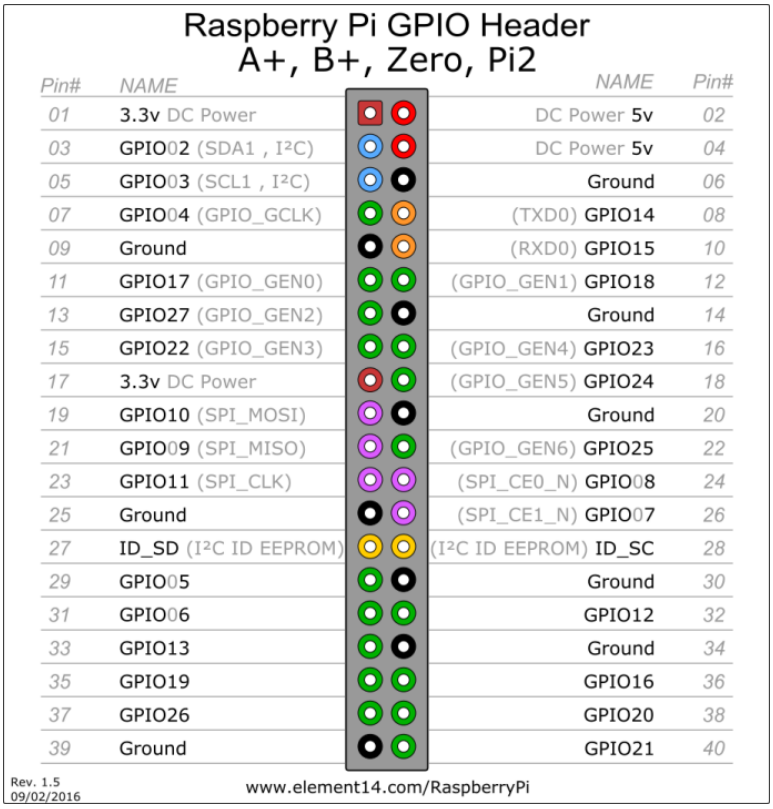


Figure 3 - Image showing the pin layout of a Raspberry Pi Zero, the TXD0 pin is used to communicate with the NTX2B (RTTY) radio as well as the GPS, the RXD0 is also used to communicate with the GPS. The SPI_MOSI, SPI_MISO, SPI_CLK and SPI_CEO_N pins are used for the LoRa radios, described in the paragraph below.

The second method of communication is LoRa which is a long range, low power, high data rate radio solution also running at a license exempt frequency. The LoRa modules are low cost and are controlled with an SPI (Serial Peripheral Interface, a synchronous serial communication protocol) interface and provide state change notifications via DIO (Digital Inout/Output, a simple protocol whereby a line can have either a HIGH or a LOW signal to denote binary flags). Their modulation and demodulation is handled internally as the modulation scheme is patented. The modules are capable of achieving an equivalent of 17,000 baud RTTY (see figure 4), making them ideal for long range 2-way communications. Their range is somewhere in the region of 60-100km, which is perfectly adequate for high altitude ballooning, this is somewhat lower than the RTTY which can reach 600km and more with perfect conditions, however, we do not need that extra range.

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

Figure 4 - Shows nominal bitrate vs bandwidth. Bitrate can be further optimised with modification to spreading factor and error coding rate. Taken from the LoRa module datasheet.

Thus far I have flown three flights, all of which have used the RTTY and two of which have used the LoRa radios, in order to use the RTTY I used a cheap Radiometrix NTX2B radio (see: http://www.radiometrix.com/files/additional/NTX2B.pdf) which shifts frequency as a result of a voltage applied to one of its pins, however, the Pi outputs 3.3v which would result in a frequency shift of about 7kHz, this is far too large as it is outside the range of typical SDR (software defined radio, a USB radio receiver) receivers, so a potential divider was needed to lower the voltage to about 0.2v-0.3v to acquire a shift of around 400-500Hz, a graph of frequency shift against voltage applied to TXD pin is shown in fig. 4. Furthermore, the LoRa modules were very successful, I used a Python library designed for a similar module which worked well, however, in this project, I will want to develop my own wrapper and API for the LoRa radios which will be more robust than the library I used before, and will handle the SPI and DIO interfaces itself, while providing me with a self-documenting API for use throughout the rest of the programming and possibly publication as a stand-alone LoRa API.

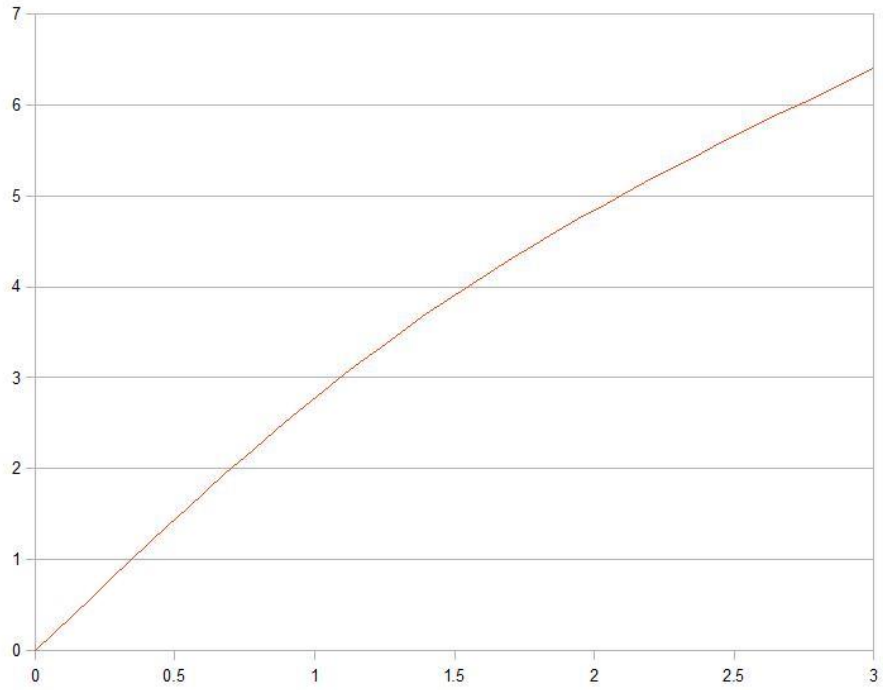


Figure 5 - The above graph shows the frequency shift (kHz, y-axis) vs the voltage applied to the radio's Tx pin (V, x-axis). Sourced from the UKHAS website.

Due to Ofcom regulations, I am restricted to specific power outputs and frequency bands as I do not hold an amateur or commercial radio license. Therefore, I must use license-exempt bands and adhere by any specific rules in those bands. There is a 434.04-434.79MHz band which I have used for RTTY, this has a power output limit of 10mW E.R.P. (effective radiated power – this takes into account the antenna gain (amplification by the antenna)); a band at 869.70-870.00MHz which has a power output limit of 5mW E.R.P. which I will use for the 868MHz LoRa downlink from the payload. These bands are useful due to their 100% duty cycle, meaning I can transmit continuously, and their lack of requirement for techniques to mitigate interference such as "Listen Before Talk" which could inhibit some of my communications. However, there is another band, 869.40-869.65MHz which allows 500mW transmissions with a duty cycle limit of 10%, this could be used for the uplink as I am only likely to be transmitting to the payload for a small amount of time compared to the time during which the payload will be transmitting to me and thus I could easily manage with 10% duty cycle, this would provide a much greater signal strength for transmissions to the airborne payload, it would be much less susceptible to interference. The table in figure 5 shows the available bands that are suitable for this project.

Frequency (MHz)	Power Limits (mW)	Other Requirements
869.40-869.65	500	Duty cycle limit of 10%.
869.70-870.00	5	None.
434.04-434.79	10	Channel spacing ≤ 25kHz

Figure 6 - Summary of frequency bands which could be useful for my project. Sourced from the Ofcom IR2030 table of license-exempt frequency bands in the radio spectrum, available at: https://www.ofcom.org.uk/__data/assets/pdf_file/0028/84970/ir_2030-june2014.pdf

The LoRa module requires a detailed understanding of its interface. The datasheet (http://www.hoperf.com/upload/rf/RFM95_96_97_98W.pdf) describes the SPI and DIO (Digital Input/Output) interface it uses. SPI is a synchronous serial communication interface used primarily in embedded systems; it has a master-slave architecture with a single master and can operate in full-duplex mode; it supports multiple slaves when separate slave-selection lines are used. The SPI is used to modify the registers on the radio, in order to configure the radio (e.g. frequency, spreading factor, operation mode etc.) or, indeed, to write the packet that is to be transmitted to the appropriate register or to read a received packet from the appropriate register. Modulation, demodulation, receiving and transmission of packets is handled internally as the methods used are patented. The DIO is used to notify the interfacing device, in my case a Raspberry Pi, when specific events occur (see fig. 6 for functionality of each pin), in my case I will be using the DIO0 and DIO5 pin, they function as follows: the DIO0 pin can be configured to change state to HIGH (1) when a transmission has finished sending or receiving (TxDone or RxDone), while the DIO5 pin can be configured to change state to HIGH (1) when the radio has changed operation mode (ModeReady, i.e. changing from transmit to standby mode). My software will need to use both SPI and DIO so I propose that I use a library such as WiringPi or Pi4J to do this. WiringPi has implementations in many languages and Pi4J is a Java based library, as the name 'Pi 4 Java' suggests; however, there are many other options such as spidev in Python which is an excellent library which I have experience developing with.

Operating Mode	DIOx Mapping	DIO5	DIO4	DIO3	DIO2	DIO1	DIO0
ALL	00	ModeReady	CadDetected	CadDone	FhssChangeChannel	RxTimeout	RxDone
	01	ClkOut	PllLock	ValidHeader	FhssChangeChannel	FhssChangeChannel	TxDone
	10	ClkOut	PllLock	PayloadCrcError	FhssChangeChannel	CadDetected	CadDone
	11	-	-	-	-	-	-

Figure 7 - Table showing the functions of the individual DIO pins. Taken from the datasheet.

The LoRa SPI interface works in the following way: each register on the LoRa module is assigned a unique address of up to 7 bits, to write to or read from a register you must send a sequence of bytes via SPI to the radio, the least significant bits of the first byte will be the register's address, the most significant bit is 1 when writing and 0 when reading. Then, if writing, you send the bytes you wish to write in the order you wish then to be written (if you send more bytes than the register can hold it will ignore the excess); if reading you must send the *n* 0x00 bytes where *n* is the number of bytes you wish to read from the specified register, again, if you specify more bytes than the register contains it'll just stop once it's read the whole register. This gives a sufficient knowledge of the SPI interface for this project, for more information see the aforementioned datasheet.

GPS

I will need a GPS module on my payload, however, due to the COCOM limits (see https://en.wikipedia.org/wiki/CoCom), which prevent a GPS from functioning if it is above 18,000m altitude or travelling faster than 1,000 knots in order to prevent GPS technology being used to guide inter-continental ballistic missiles (this limitation is an artefact of the Cold War), I have to purchase a specialist GPS module which instead requires both high

altitude and high speed to shut down rather than just one of the two. There are several options and all have a serial interface running at 96,000 baud by default, this will require usage of the Pi's RS232 serial connection to read continuously from the GPS. Alternatively, I could use an I^2C interface which many of the GPS modules provide; however, the Ublox GPS modules (which is the most popular type with HAB enthusiasts) perform clock stretching at arbitrary times and the Pi I^2C driver simply cannot handle this. I should note that the GPS also outputs several different types of location strings in a looping sequence, all the output types can be seen on http://www.gpsinformation.org/dale/nmea.htm but we are only interested in GGA strings, which give a 3D position (i.e. including altitude) and the number of satellites the GPS is currently in contact with. These give latitude and longitude in the form ddmm.mmmm where dd is the number of degrees and mm.mmmm is the number of arc minutes as a decimal. So, to convert to degrees correctly, which most mapping systems use you must use the following equation:

$$\text{coordinate degrees} = dd + \frac{mm.mmmm}{60}$$

Additionally, the GPS doesn't give the correct negative values, it instead gives another field saying whether the position given is North/South of the equator for latitude or West/East of the Greenwich Meridian for longitude so these fields will need to be checked and the correct negative sign will need to be applied to the latitude and longitude values.

I should also note that the GPS doesn't by default work at high altitudes, by default it adheres to the standard CoCom limits, a sequence of bytes must be sent to it in order to switch to 'Airborne Mode', the required bytes are as follows:

```
[0xB5, 0x62, 0x06, 0x24, 0x24, 0x00, 0xFF, 0xFF, 0x06, 0x03,
0x00, 0x00, 0x00, 0x00, 0x10, 0x27, 0x00, 0x00, 0x05, 0x00,
0xFA, 0x00, 0xFA, 0x00, 0x64, 0x00, 0x2C, 0x01, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x16, 0xDC]
```

Server and Slave Network

As I have stated, I have flown flights using both LoRa and RTTY transmission methods and have gained an understanding of their function and implementation through building and programming the payloads for these flights. I've also seen that when chasing a payload, it is quite easy to lose contact with it temporarily due to, for example, an inconvenient road route or a building breaking line of sight, and at these times I've previously relied on other enthusiasts (members of the UK High Altitude Society) to receive the transmissions using their high gain stationary antennas based around the country. In order for my 2-way communication system to function consistently, through signal dropouts with the payload, I suggest that I need to harness the many willing enthusiasts across the country who would be happy to assist in tracking. So, I propose the development of a slave tracking software that acts to rebroadcast transmissions that do not successfully reach the payload when transmitted by the main controller and to forward any transmissions received by the payload to a central server (the function of this server is described below). Figure 7 below shows a map of UKHAS listening stations recently active, as you can see there are many

receivers across the country, massively increasing listening and transmitting capacity if they can be harnessed! I will suggest this to members of the UKHAS.

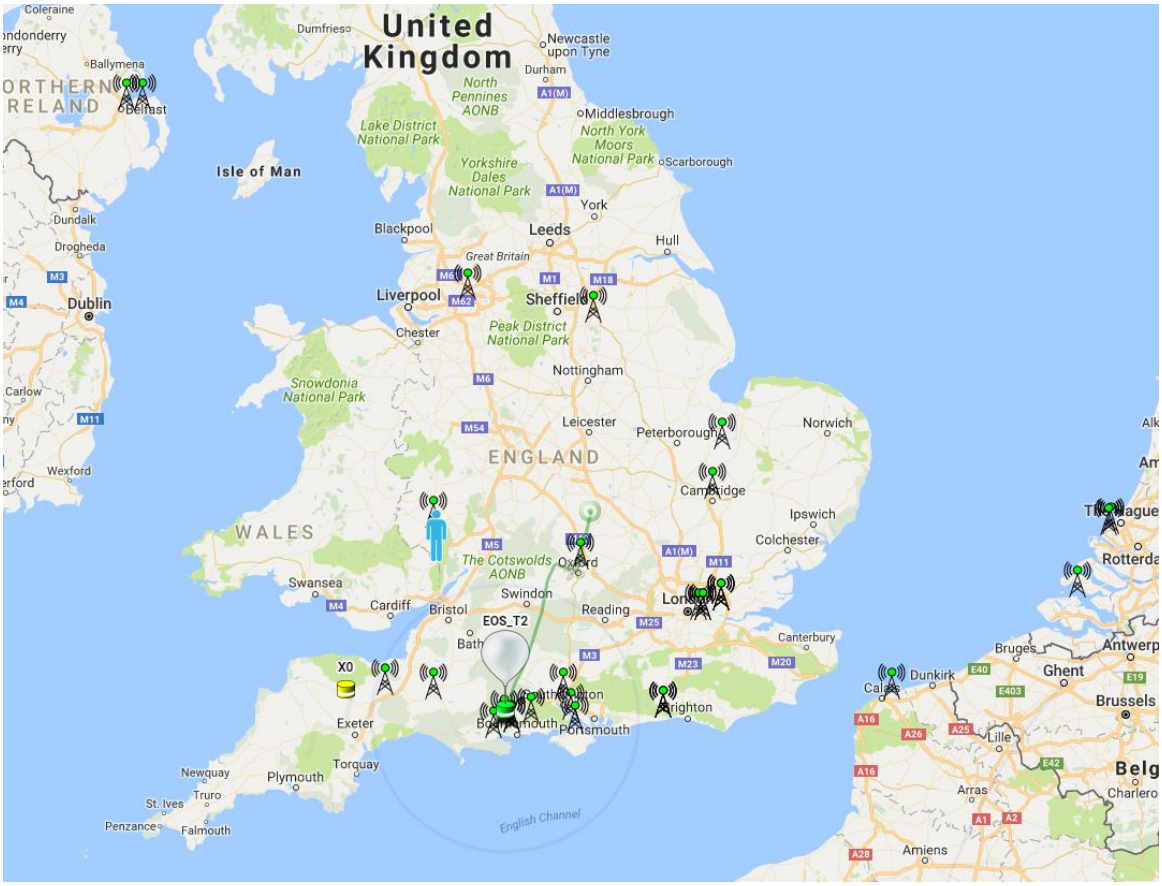


Figure 8 - Map of UKHAS receiving nodes active recently. Screenshot from the UKHAS tracker (https://tracker.habhub.org/).

Potential Radio Solutions

Parallel

My software could work my transmitting packets to the payload on one LoRa radio, and from the payload on another LoRa radio running at a different frequency. Having two radios operating simultaneously at different frequencies would allow us to maintain 100% duty cycle on both transmit and receive operations in order to maximize efficiency, rather than having to wait for a cycle to complete. Note that although in my implementation I would most likely be using 434MHz for uplink and 868MHz for downlink, the frequencies should be configurable by the end user in a configuration file, so if they wanted to use two 868MHz radios with lower bandwidths or one 868MHz and one 434MHz then they should be able to by modifying a configuration file. However, there are problems with having two radios operating simultaneously, particularly as the two frequencies available to the LoRa radios are harmonically related which would result in significant interference, potentially preventing any communication from functioning correctly.

Cycled

Alternatively, I could use only one transceiver and switch between receive and transmit regularly, this would reduce responsiveness for the 2-way communications, but make it

more robust and less likely to be affected by interference. Note that my implementation should be fully configurable allowing users to use different frequencies and different cycles of uplink and downlink. This would mean that I would need to configure a cycle where the airborne payload spends some time transmitting, then transmits a packet informing the ground-based device that it is now accepting packets, waits to receive packets, and then if no packets are received for some defined period of time, begin transmitting for a while again. The ground based receiver would need to be configured to receive packets and then upon receiving a packet that states that the transmitter is going into receive mode, transmit any packets that are queued for transmission.

Further Client Discussions

Radio Solutions

As mentioned the UK High Altitude Society (UKHAS) is a group of enthusiasts who fly high altitude balloons, of which I am a member. We have a suite of software called habitat which allows data received by numerous receivers to be forwarded to a central server and displayed on a map. I would like for location data received from my payload to be relayed to habhub. My project will be targeted at myself and other members of the UKHAS. Further, in order for the slave transceiver network to function correctly, I would need to develop my own central server software which coordinates the slave node network and logs flight data for my own records, this would be developed with the overall aim in mind for it to be eventually integrated into the habhub habitat software suite. The central server could also function to ensure that the main controller is forwarded any packets from the payload which it does not receive itself which are, however, received by the army of slaves.

I have been discussing this project with members of the UKHAS and have made many of the decisions noted in this section and below in the specification with their interests in mind, as they are my principle clients. I had some discussions with members of the UKHAS on the #highaltitude IRC channel on Freenode, I discussed the advantages and disadvantages of having two LoRa radios in parallel and of using just one with a cycle of transmit and receive. I have reached the conclusion that it would be more suitable to use just one radio with a cycle of transmit and receive, with both queuing packets for transmission while unable to transmit (as they're in receive mode). Several members of the UKHAS were concerned about the interference due to having the two radios simultaneously transmitting and receiving while adjacent to each other and most were unconcerned about the slight delay that would be caused by having to queue packets for transmitting while in receive mode. See fig. 9 for transcript of IRC.

```
[08:27] jakeio (~jakeio___@149.254.183.73) joined #highaltitude.
[08:29] <jakeio> Hello, my A-Level Computing project is to create a software suit that allows 2-way communications with airborne payloads.
It will use two LoRa radios in parallel to maintain 100% duty cycle transmit/receive. Can I ask what people think of this, and if they have
any suggestions for features. This is all for my 'research and analysis'.
[08:33] fab4space-f4hhv (~Fabrice@109.237.242.98) left irc: Ping timeout: 248 seconds
[08:34] fab4space-f4hhv (~Fabrice@109.237.242.98) joined #highaltitude.
[08:34] <Geoff-G8DHE-M> Will the receiver front end work happily when your transmitting (in the same  band ?) next too it ?
[08:36] <jakeio> I wasn't necessarily going to use the same band, as many people have one 434MHz LoRa and one 868MHz LoRa on one Pi I
thought I might use one of those for transmission and another for receiving. Should be less interference that way shouldn't there?
[08:36] <mfa298> some analysis of ir2030 might be useful to demonstrate you've looked at what's allowed (or not)
[08:37] <Geoff-G8DHE-M> Possibly, the front ends have little to no filtering so can still be swamped by other large signals.
[08:40] <jakeio> mfa298, I've commented on the main bands suitable for LoRa, aren't these the 434.04-434.79 and the two 868ish ones, one
with the 500mW limit but duty cycle limit of 10% or "techniques to mitigate interference" and the other being the 5mW one we usually use?
Geoff-G8DHE-M would you suggest it would be better to use just one transceiver to prevent this? Slightly reducing responsiveness in favour
of robustness?
[08:40] <SpacenearUS> New position from 03HIRFW-6 after 0313 hours silence - 12https://tracker.habhub.org/#!qm=All&q=HIRFW-6
[08:41] <Geoff-G8DHE-M> It certainly makes for a simpler and lightweight system.  Unless you really need the bandwidth/time then its a lot
simpler.
[08:42] <gonzo_> if you are planning those for flight, double check the power limits for airborne kit
[08:43] <Geoff-G8DHE-M> remember 434 and 868 are harmonicaly related so there is always going to be greater residual signals from one band
to the other.
[08:46] <mfa298> if you don't need 100% duty cycle on uplink then the higher power 868 option could be good for ground to balloon.
[08:47] <jakeio> Thanks. I'll probably do a table comparing both methods and then decide on the one radio one! And mfa298 thanks, I shall
include this in my write up!
[08:47] <Geoff-G8DHE-M> You might also want to read up and comment on intermodulation and mixing products of radio signals as its these that
normally limit such systems.
[08:48] <mfa298> A Level computing sounds a lot more interesting than when I did it.
```

Figure 9 - Transcript of IRC chat with members of the UKHAS.

Two Way Protocol

I discussed my concept for my protocol specification with the UKHAS on IRC, the log is in figure 10, we discussed the network of slave receivers and it was suggested that the system be coordinated so that only one node is transmitting at any given time, this is what I had suggested above in my research. When discussing the protocol, I suggested a system where the payload transmits to say it is entering into receive mode and then begins to listen, then the ground station begins transmitting and when the ground station has not transmitted anything for a given time the payload returns to transmit mode and then processes the received packets; members of the UKHAS suggested that I expand upon this with something similar to the sliding window protocol used in TCP, this is where each packet has a consecutive number and the receiver uses the numbers to put the packets in the correct order, detect duplicates and detect missing packets; the sliding window protocol puts a limit on the number of packets that can be transmitted in a given time by limiting the number of packets that are sent before waiting for an acknowledgement (source: https://en.wikipedia.org/wiki/Sliding_window_protocol). Although this will not exactly be used in my implementation, it would be suitable to limit to a set number of transmissions per transmit cycle to prevent any cycle continuing for too long. I do not think that sending NACK (negative-acknowledgement) packets or ACK (acknowledgement) packets will be appropriate for my implementation, ACK packets would be inappropriate because I am running on an unreliable and slow medium and this would result in having to send an extra packet for each packet, NACKs would still be inappropriate because the number of packets transmitted in a cycle will not necessarily be fixed.

Members of the UKHAS expressed concern that others could attempt to enact remote control operations on the payload, a proposed solution to this is described later (see 'Authentication').

[15:51:28]	jakeio	My A-Level computing project is to develop a software suite for 2-way communications with airborne HAB payload.
[15:52:02]	jakeio	I've settled on a system with one LoRa radio on each end with a cycle of transmit and receive with packets being queued for transmission while in receive mode.
[15:52:46]	Ian_	Define A=Ground B=Air
[15:52:51]	jakeio	OK.
[15:53:35]	*	urchin_ has quit (Ping timeout: 252 seconds)
[15:53:43]	Ian_	What protocol, GPS time slots?
[15:54:10]	jakeio	Would it be useful to have a system where multiple users can contribute to the receiving and transmitting of packets to the payload, in a similar way to how we all contribute to tracking at present, however, with the added ability of helping with transmitting.
[15:54:12]	SpacenearUS	New vehicle on the map: KF6RFX-3 - https://tracker.habhub.org/#!qm=All&q=KF6RFX-3
[15:54:12]	SpacenearUS	New vehicle on the map: KF6RFX-4 - https://tracker.habhub.org/#!qm=All&q=KF6RFX-4
[15:54:15]	jakeio	And protocol will be...
[15:55:08]	Ian_	I would have a single A node. Any messages by internet to A for transmission or you risk collisions
[15:55:47]	jakeio	B sends a packet that says "I'm listening" and then switches to receive mode, A then transmits an acknowledgement packet and begins transmitting it's queued data, when A has finished and no packets are received for a short while by B then B returns to transmit mode and A receives.
[15:57:25]	jakeio	Ah, I've come up with a way to coordinate the multiple transceivers, in that the other transceivers will only be used instead of the main one if the main one fails to communicate with the payload for a given time.
[15:57:40]	*	pcal1 has quit (Ping timeout: 260 seconds)
[15:58:50]	Ian_	If you have a network of potential ground station txrs then I would be inclined to coordinate them via internet, with only one having the permission at any given time.
[15:59:36]	Ian_	So communication is node to node LoRa but communications management is via internet
[16:00:17]	Ian_	The airborne node merely needs to let you know that it has GPS lock to be in business of receiving and sending acks
[16:00:30]	jakeio	Yes, if A doesn't get through to B for two "cycles" then A sends packets to the server to distribute to other receivers.
[16:00:40]	jakeio	transceivers*
[16:00:49]	Ian_	Sounds sound!
[16:01:21]	jakeio	The reason I opted for a non-timed option, i.e. B has to initiate receive mode, is so as to have flexibility in how long those cycles last.
[16:01:33]	jakeio	This is in order to maximize duty cycle while airborne.
[16:01:40]	Geoff-G8DHE-Lap	Radio links are unreliable - using Ack's on unreliable links is bad, better to use packet numbering and then NAck's when sequence is missing a packet.
[16:01:55]	Ian_	That makes good sense
[16:02:25]	jakeio	OK, so each packet has an ID and if the sequence is interrupted then request retransmission?
[16:02:29]	jakeio	Or have I misunderstood.
[16:02:47]	*	Guest46323 is now known as richardeoin
[16:02:59]	Geoff-G8DHE-Lap	That's right, otherwise you can spends more time chasing missing Acks then sending useful data
[16:03:14]	jakeio	OK, that makes sense. Thanks Geoff-G8DHE-Lap.
[16:04:00]	Ian_	No, that's right. Personally I would have fixed time slots up and down to keep it simple. You can always refine that later otherwise you are mixing communications and communications management on an unreliable medium
[16:04:14]	Geoff-G8DHE-Lap	The TCP sliding window concept is a good idea for this
[16:05:17]	Geoff-G8DHE-Lap	https://en.wikipedia.org/wiki/Sliding_window_protocol
[16:06:38]	jakeio	OK, so the sliding window is used to prevent extremely large packet numbers?
[16:06:59]	Geoff-G8DHE-Lap	One aspect of it yes,
[16:07:25]	Ian_	It saves things getting too far out of real time I guess
[16:07:56]	Geoff-G8DHE-Lap	all the time there is data being exchanged you keep going, only go back and resend if NACK, but you can't send for ever as the other end may not be receiving anythign
[16:08:37]	Ian_	So telemetry on the down leg would be uncontrolled and optimistic
[16:12:23]	jakeio	Right, thanks for the info on the sliding window protocol.

Figure 10 - IRC log of discussions about the network of slave receivers and the 2-way protocol. Note that 'ACK' means acknowledgement packet and 'NACK' means negative-acknowledgement packet.

Character Set

When transmitting data via the LoRa radios, it needs to be encoded into bytes, so I will have to use a specific chosen character set. The chosen character set needs to be 8 bit because the SSDV program encodes data into packets of 256 bytes and I wish to transmit one packet per transmission. Additionally, the character set needs to be compatible with existing HAB software which use extended-ASCII. The character set I shall use will be ISO-8859-1 for it fulfils all the requirements and is included on most systems with most languages having built-in functions that can encode strings in it. This is effectively extended ASCII so is compatible with most existing HAB software.

DI-Fldigi Interface

Further discussions with members of the UKHAS led me to the conclusion that it would be a suitable extension to interface with the already commonly used (by UKHAS members) dl-fldigi software which is used to decode RTTY. This results in my software effectively being an all-round HAB toolkit, making it very useful to an enthusiast. I did, however, decide that this should only be an extension as most enthusiasts are happy to have both programs open simultaneously without an overall wrapper and this project is at its current stage a prototype.

Backup Tracker

Additionally, members of the UKHAS who I asked all wanted to have RTTY transmitted by the payload as well as an emergency backup tracking method due to it being so robust and reliable. Because of this, as the RTTY typically runs at 434MHz, I would thus need to use 868MHz for the LoRa. The IRC log for this is shown in figure 11. It might be sensible to disable the RTTY for the duration of the payload's receive cycle as although the RTTY is running at a different frequency than the LoRa the two frequencies are indeed harmonically related, resulting in possible interference. Having done some research and testing with an RTTY tracker running from the same payload, I have decided to instead run a separate payload separated by several metres from the 2-way payload in order to limit interference. For this I will use my well-tested tracker from my previous flights which I know to work well. Hence, the two-way payload software will not need to transmit RTTY as well as LoRa.

[16:04:03]	jakeio	Would you think it important to have a backup standard RTTY tracker setup also, in case the new 2-way LoRa system fails?
[16:04:30]	gonzo_	a backup is always going to be a good idea if weight will allow
[16:06:12]	gonzo_	if you are running sometjhing experimental, and you need to get it back, or need to know where it is for part of the experiment. Then a backup known reliable tracker is a good ide
[16:06:25]	gonzo_	a
[16:09:31]	mfa298	In terms of a backup transmitter I think today probably demonstrates why a backup is good!

Figure 11 - IRC log showing discussion of backup RTTY tracker. Note that the final comment by mfa298 is regarding a payload that was lost due to the failure of its single tracker.

Maximizing Link Budget

In the past, members of the UKHAS have managed only limited 2-way communications, they have successfully transmitted a packet to the payload in the event that image packets are missed requesting that they are retransmitted. These used high gain transmitting antennas and the software was written in Python. I spoke to , who wrote this software and he has confirmed that he has had reasonable success with it in the past using a relatively simple antenna as well as with a high gain directional antenna.

A screenshot of an email from is shown in figure 2, following this and discussion on the IRC I have concluded that I shall use the 500mW license-exempt band (see fig. 6) to transmit up to my payload as this will provide a greater link budget and this will make the 2-way communications much stronger, however, the LoRa module is only 100mW at its maximum so this will be my maximum output, however, the 500mW band is still the only band in which I can use this power output on the transmitter (using the 500mW band was suggested by a user on the IRC, see fig. 9). Furthermore, suggests in his email to optimise the LoRa parameters for best link budget, typically a spreading factor of 7 has been used with a 250kHz bandwidth for high data-rate long-range communications from HAB to payload, however, for payload to HAB I intend to use a lower bandwidth to maximize resistance to interference and increase link budget, a lower bandwidth means that the total power output is spread over a smaller section of the radio spectrum. This should help alleviate the issues created due to the large listening footprint of the airborne payload that mentions in his email.

Authentication

I will be sending transmissions from the ground station that could, if used incorrectly, sabotage my flight, for example, sending the console command *sudo halt* would achieve

that. Because of this, I feel that my packets should be some way secured, encrypting would be one method of doing this, however, I am not sending sensitive data so it seems unnecessary. An alternative would be using a sort of 'salt' like those used in salted hashes when storing passwords in databases, by this I mean that when configuring the payload the user would need to set a 'salt' or key which will be included in the calculation of the checksum but not actually transmitted, that way, on the receiving end the software could append the same key to the received packet and calculate the checksum, if the checksums match then we know both that the data is correct and has come from an authorised source.

Conclusion

In conclusion, as discussed, my target users are members of the UKHAS who would find it useful to see basic diagnostic information about an airborne payload such as battery voltage and internal temperature in order to detect issues, without having to include that in their standard telemetry format; members also expressed the desire to have the ability to reboot the entire Pi, making the point that the software should be configurable to start when the Pi boots, this is all in order to fix potential runtime issues; it was also pointed out that the payload software should not continue to wait if no packets are received from the ground as this could result in the payload going silent, it should time out, this is to prevent lost flights; several members also expressed the need to have multiple receivers and transmitters to maximize the range of the 2-way communications as discussed above, however, as this project is a prototype I think this should be considered as a future extension to the main project, simply achieving two way communications with the payload from a single ground station will be sufficient proof of concept. It was also noted that if there were no 2-way communications packets to send then the payload should just transmit standard telemetry strings following the standard UKHAS format (see figure 13 or for more information see: https://ukhas.org.uk/communication:protocol), 2-way packets should be in some way distinct from standard telemetry strings, perhaps a specific prefix should be used, rather than the standard $$ used by the UKHAS strings.

[16:14:09]	jakeio	What features would you find it useful to have when communicating with an airborne payload? And by this, I mean on a routine flight, non-repeater/cutdown or anything, just useful things to be able to do to a payload!
[16:16:25]	Geoff-G8DHE-Lap	Maybe remote reset, baettery power/current (just sensor as now)
[16:17:31]	Geoff-G8DHE-Lap	power control of Flying Tx might be handy
[16:19:50]	jakeio	OK, thanks Geoff-G8DHE-Lap. Other features I've suggested have been remote image download (full size images, however); diagnostic (e.g. the sensors we already use and CPU temp, etc.); telnet style sessions; change frequency.

Figure 12 - Showing a small snippet of my discussion with members of the UKHAS about potential features for the system.

```
XOR Checksum'd string:
$$A1,15254,15:36:34,52.145255,000.542061,00118,0000,03,3F4D3F2F,45*62
```

Figure 13 - Image showing UKHAS protocol standards.

Specification

Ground Station Controller

1. The transceiver controller should:
    a. Provide the user with a clear front-end.

    i. This need not be usable by a naïve user, as this software is aimed at experienced radio operators and high altitude balloon enthusiasts; this hobby requires a certain amount of technical knowledge on the subject matter.
    ii. The UI should function on a touchscreen, as the Pi used will be using a touchscreen for the interface.
    iii. The UI should provide the following features: current telemetry display; current image display; transmission log; remote console; control interface; configuration tab.
b. Provide a configuration file which allows the user to set the callsign, transmission frequency and bandwidth, the receiving frequency and bandwidth, the error coding rate, the spreading factor, the transmission power and whether the payload is using explicit packet headers. The user should also be able to set the key for 2-way packet authentication.
c. Allow normal LoRa receiving using an 868MHz module or a 434MHz module.
    i. This is done by awaiting the DIO0 pin to go high, then reading the contents of the LoRa FIFO.
    ii. LoRa modules are interfaced with SPI so this will require an SPI wrapper to be developed which handles all LoRa functions needed for the project.
d. Decode SSDV image packets and display on the graphical interface, also forward SSDV image packets to the habhub servers.
    i. SSDV can be decoded using a freely available library developed by a member of the high altitude ballooning community called fsphil.
    ii. SSDV is encoded by the library into 256 byte packets, these can be decoded individually so if a packet is lost most of the image can still be seen.
e. Parse telemetry, ignoring those with failed checksums or incomplete data and showing the relevant data on the telemetry display.
f. Log all received packets to a file with a timestamp.
g. Switch to transmit mode after receiving a packet from the payload stating that it is entering listening mode and send any 2-way communication packets that are queued.
    i. Packets will be queued by the user when they request that an action be completed by the payload.
h. Allow the user to queue packets for transmission by either clicking one of the command buttons in the control menu or by sending a remote console command.
i. Allow two-way communication with the airborne payload as described below.
    i. Two-way communication should be initialised by the sending of a packet by the payload stating that it has begun waiting for transmissions from the ground.

ii. The ground station should then switch to transmit mode and move to the transmit frequency and bandwidth.
iii. The ground station should then send a fixed number of its queued packets.
iv. The payload should return to transmit mode after not receiving any packets for a set duration of time, or after receiving the correct number of packets.
v. Each packet should have a consecutive ID; the ID will reset to zero at the start of each transmit cycle.
vi. All packets involved in two-way communications should be prefixed with an identifier so that they are not mistaken for standard telemetry or SSDV.
vii. The user should be able to control basic airborne operation such as toggling image transmission.
viii. The user should have remote shell/telnet style access and be able to reboot the payload in an emergency debugging attempt.
ix. The user should be able to request diagnostics of the payload e.g. number of pictures stored or output from sensors, there should also be a telemetry log, as telemetry data will be transmitted regularly as part of 2-way communications to maintain accurate location.
x. The ground station should return to the receiving frequency and bandwidth once the transmit cycle is over.

j. If the user has configured their payload on the web portal, the ground station should be able to, given the payload callsign, download the payload configuration and write the configuration file for the user. The user will, of course, have to enter the key themselves.
k. Allow the user to toggle uploading data to the server (effectively allow 'offline mode' for testing).

Server

2. The server should:
    a. Wait for telemetry and SSDV data to be received and then forward this data to habhub.
    b. Provide a means to export 2-way communications data in CSV format from the web portal.
    c. Allow users to view a live log of telemetry packets and 2-way packets received by the server, also indicating when an image packet is received.
    d. Provide a web portal to allow users to configure their payload and add it to the database.
        i. The web portal should have a similar graphical design to the habhub website.

Payload Software

3. The payload software should:

a. Transmit standard telemetry and SSDV on LoRa.
    i. In that the LoRa should transmit standard telemetry and SSDV for the duration of its transmit cycle which is not taken up by 2-way packets.
    ii. This is again going to require an SPI interface with the LoRa radio.
b. Have a configuration file functioning in the same way as that of the ground station.
c. Allow the 2-way communications to function as described in section 1, allowing the user to view diagnostics, access shell remotely, reboot, control transmission mode, etc.
    i. Remote mode is, as described, using LoRa in a cycle of transmit and receive.
    ii. A standardised packet format will be designed in the design section of the project.
    iii. The LoRa radio should be used to transmit for a given number of 2-way packets, followed by the telemetry and SSDV packets and ending with a packet stating that the payload is about to begin receiving instructing the ground station to transmit queued packets.
    iv. It should then wait to receive the fixed number of packets that will be send by the ground station. It should time out after a short while if it receives nothing.
    v. It should then return to transmission.
    vi. The payload should be able to handle any 2-way packets sent to it from the ground station, and it should queue appropriate responses to its transmit queue. For example, it should execute the command given in a shell command packet and transmit the resulting output.
d. Add all packets which are required to be transmitted to a queue so that they can be sent in required order.
e. Read from the GPS regularly, updating the current telemetry data so that the most up-to-date telemetry is transmitted each cycle. Additionally, it should clear the serial cache after reading a location fix from the radio as otherwise the buffer will fill up with old location fixes.
f. Take images at fixed intervals using the Raspberry Pi camera module.
g. Should be designed to continue functioning without failure under unforeseen circumstances.

So, to summarise, the project should encompass three modules: the controller module, responsible for both receiving standard telemetry from the payload and transmitting and receiving 2-way communication packets, as well as communicating with the central server to handle logging and packet forwarding (to habhub); the server which is responsible for logging all data and storing in a database configuration data for payloads; and the payload software responsible for operating the payload and also transmitting to and receiving from the ground station(s) as well as taking pictures and maintaining an accurate location fix for the telemetry.

Potential Programming Solutions:

Java

I could use Java with the Pi4J library which provides SPI, DIO and RS232 APIs. This would mean I only have to use a single library. Additionally, an Oracle Java Runtime Environment is in available in the Pi repositories by default (see: https://www.raspberrypi.org/blog/oracle-java-on-raspberry-pi/) and is installed by default on the latest version of Raspbian. Additionally, this allows me to use Java Swing or JavaFX for the development of the GUI which would provide excellent ease of development, though I am aware that there can be some issues with JavaFX on Raspbian. Additionally, an object-oriented approach will be sensible as I'm writing an interactive program with mutable states. Additionally, as habitat uses an HTTP interface, I could use any of the myriad of HTTP libraries available for Java, for example the Apache HTTP Client library or the built-in HTTPConnection library.

Python

I could use Java with spidev for the SPI interface, the integrated GPIO library for DIO and pyserial for the serial interface. These are both very easy to use, however, pyserial has a few known bugs which can result in a read operation hanging indefinitely. The 'Requests' (see: http://docs.python-requests.org/en/master/) library is an extremely easy to use HTTP library. Python is also a very easy programming language to use and is often far more flexible than other languages. Additionally I could use Tkinter to develop my GUI. Python is also the most used language on Raspberry Pis.

C

I could use C and use the WiringPi library to access the serial, DIO and SPI interfaces. Then I could use a library like cURL to for networking. Furthermore, I could use ncurses to develop a command-line GUI. C is a more complex programming language but would provide a significant performance gain. However, as this performance gain will be unnecessary using C would be unnecessary.

VB.NET

VB.NET would probably not be suitable for this project because although it is possible to access the I/O devices of the Pi, VB.NET is certainly not designed with these in mind.

Chosen Solution

I will use Java and Pi4J to develop my project because Java's in-built Java Swing graphics libraries will make the development of my GUI simpler; the Pi4J library provides all the I/O capabilities that I need for my project. I will not use Python due to the potential issues with pyserial; I will not use C as the only reason to do so would be for performance which is not necessary here and adds unnecessary complexity; VB.NET is, as I have mentioned, unsuitable for this project. Additionally, I should note that Java has easy built-in functions for converting strings to and from ISO-8859-1 byte arrays.

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:

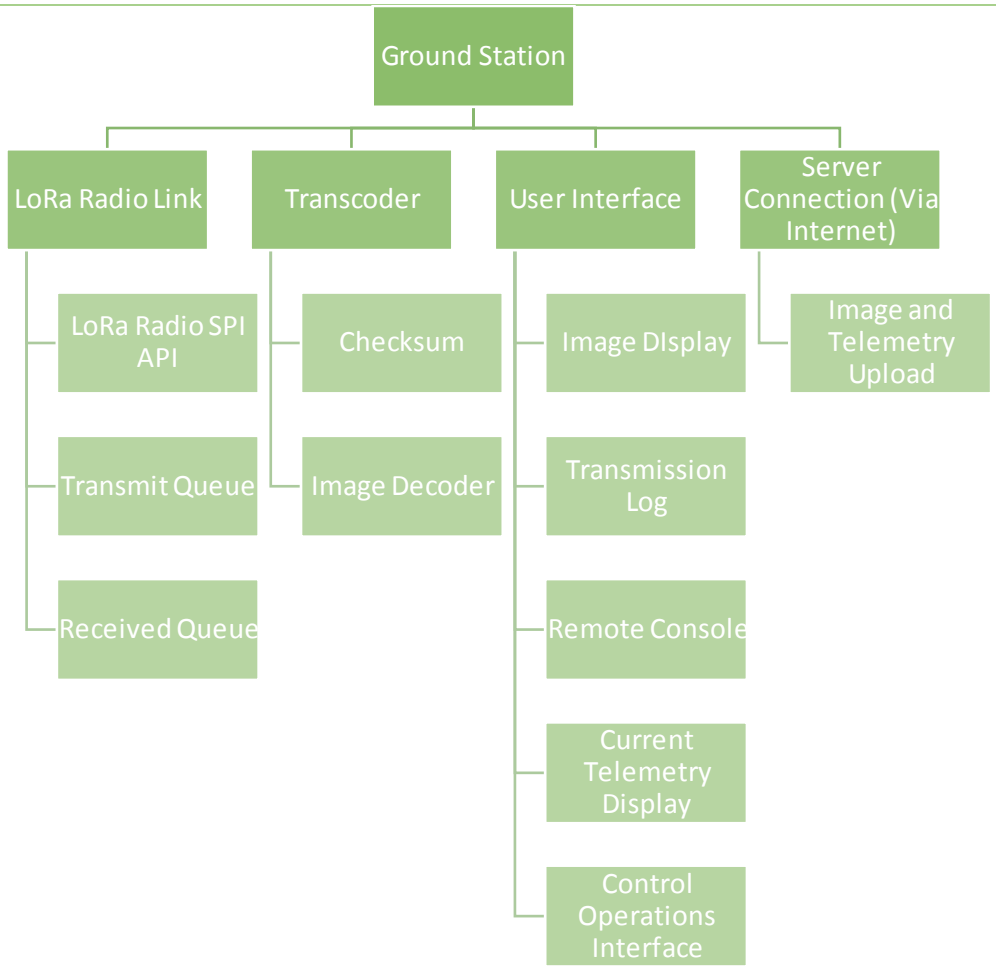
```
$$CALLSIGN,ID,TIME,LAT,LON,ALT,SATS*CSUM\n
```

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.

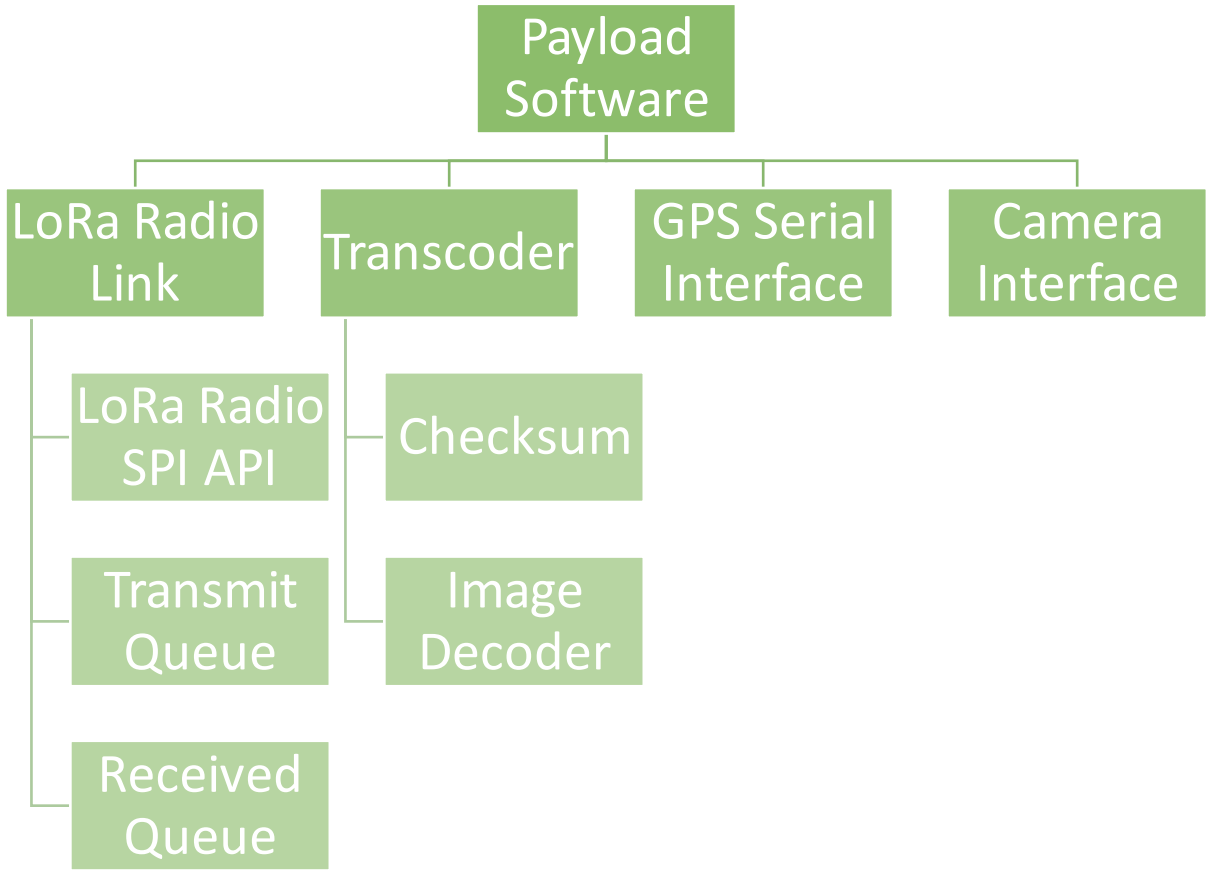
Ground Station



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.

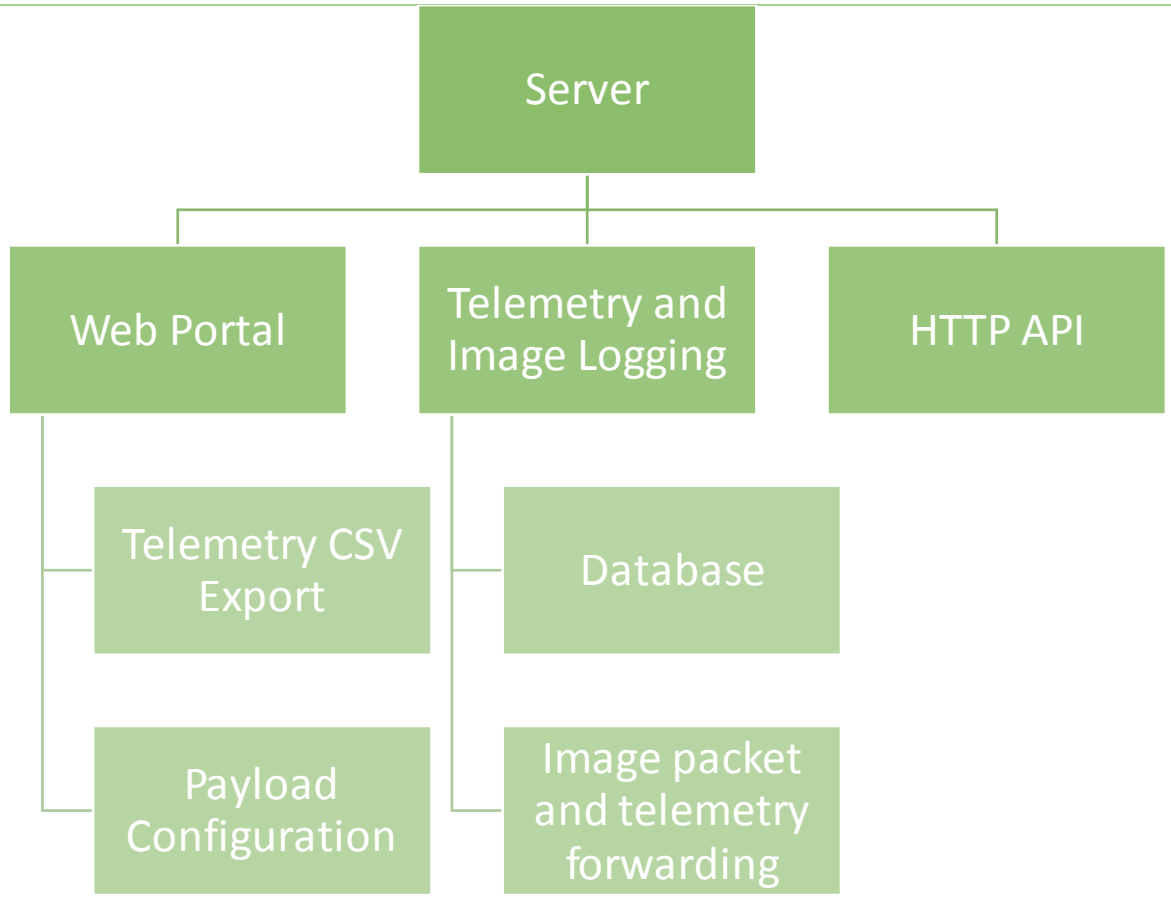
Payload Software



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.

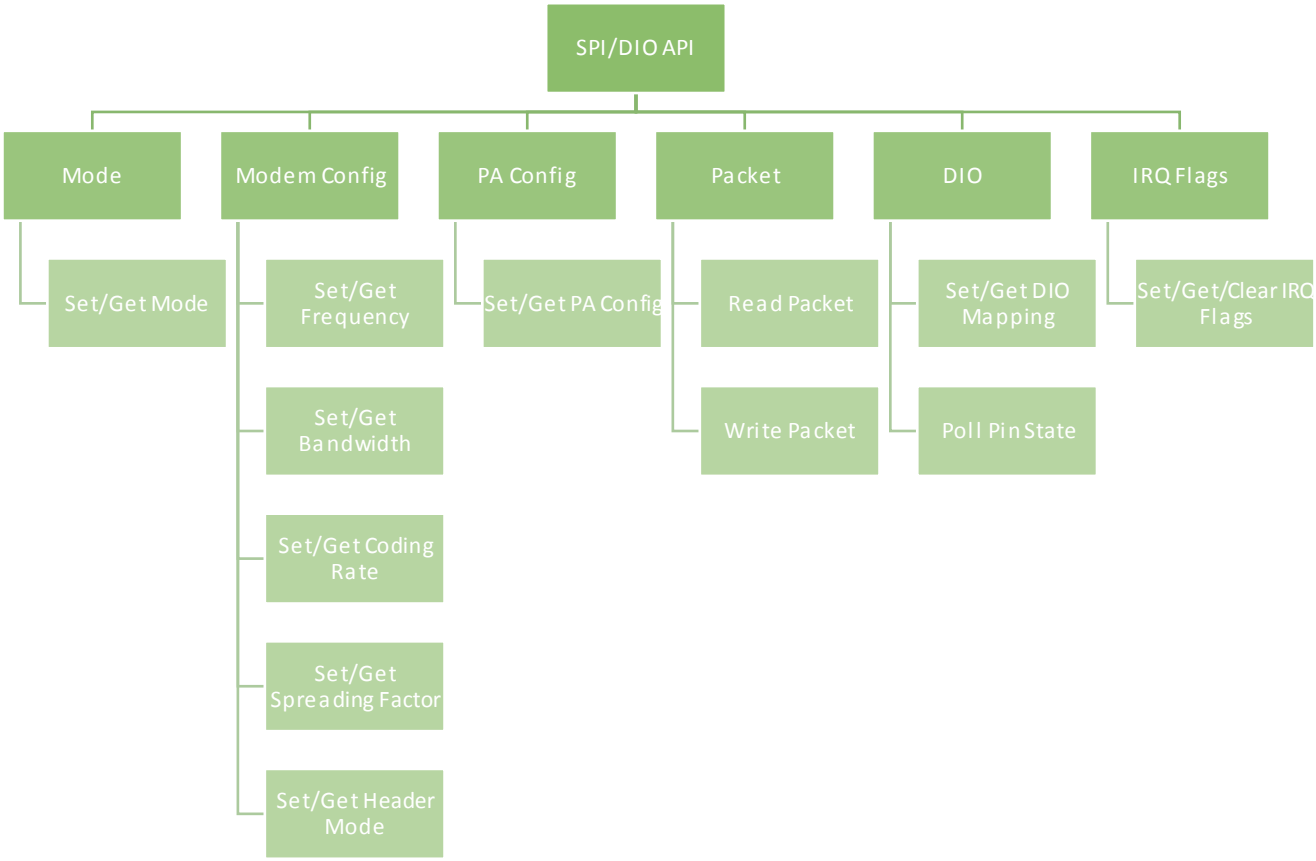
Required Operations

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.

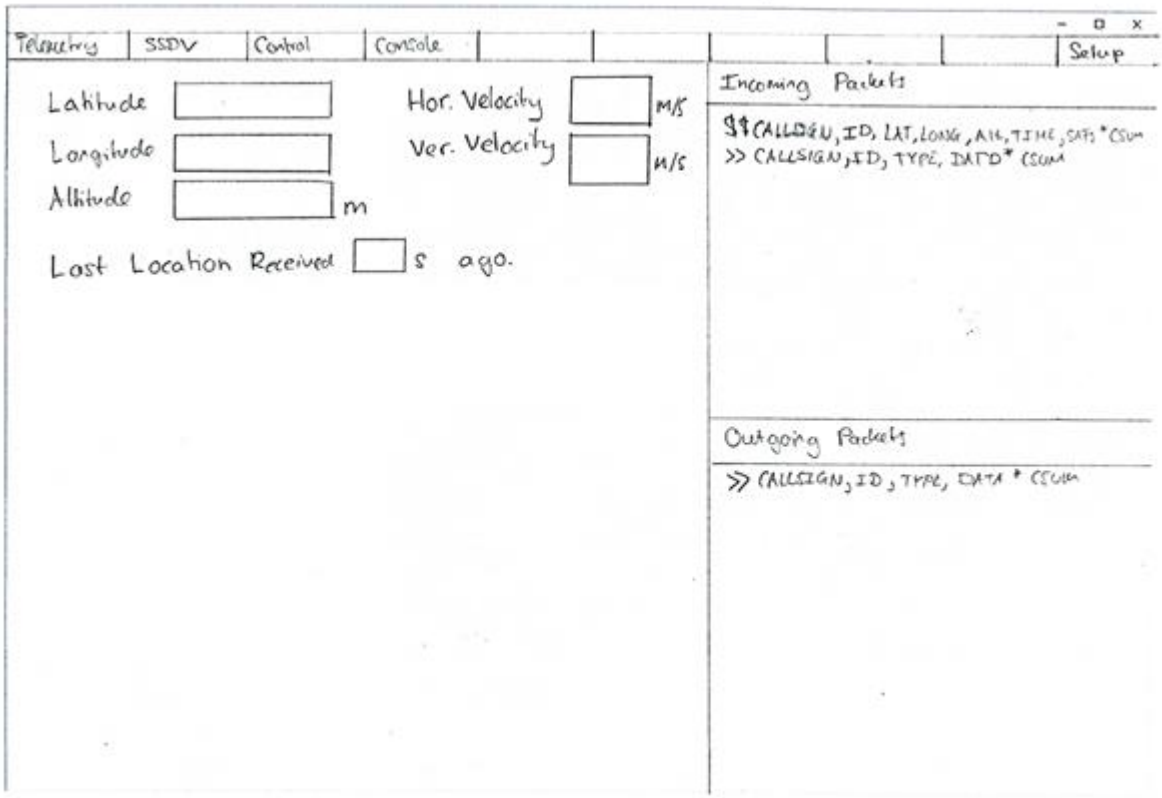
RegFifoRxBaseAddr	0x0F	Stores the base address of where a received packet is stored in the FIFO.
RegFifoRxCurrentAddr	0x10	Start address of last received packet.
RegIrqFlagsMask	0x11	Stores the IRQ flag mask settings, determining which flags can cause an interrupt.
RegIrqFlags	0x12	Stores the current 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.

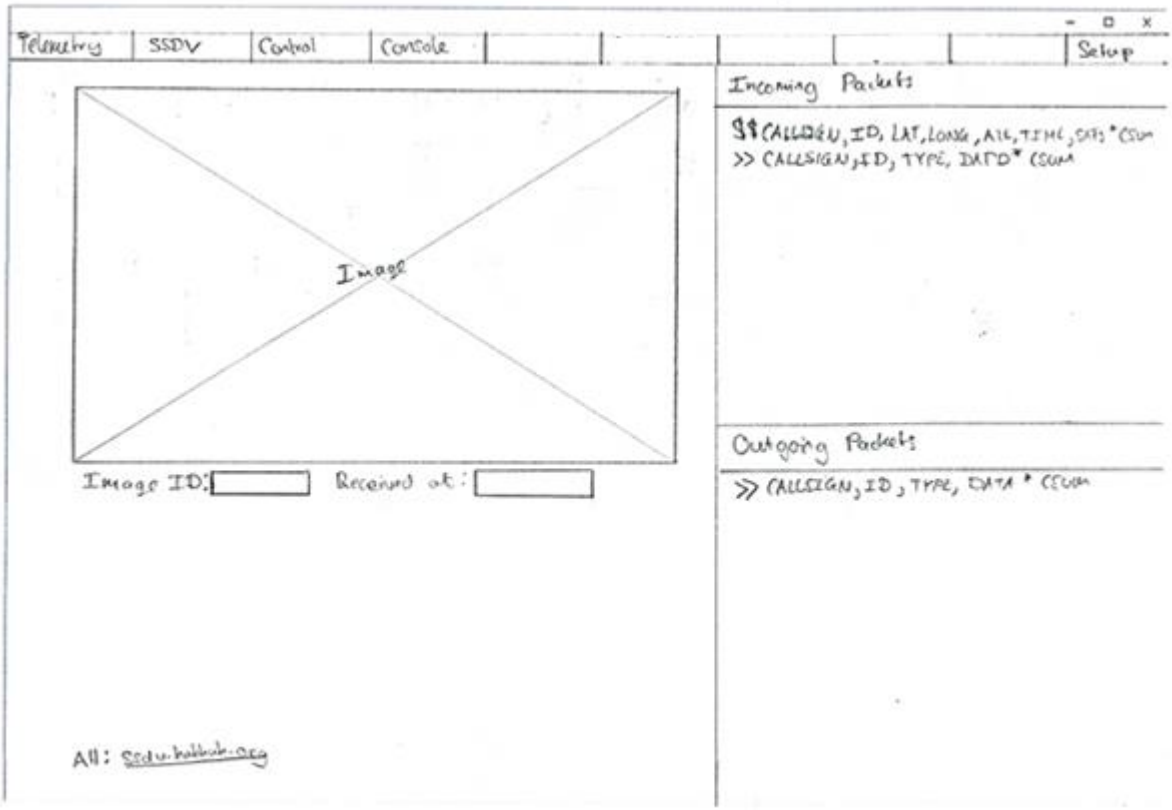
User Interface

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.

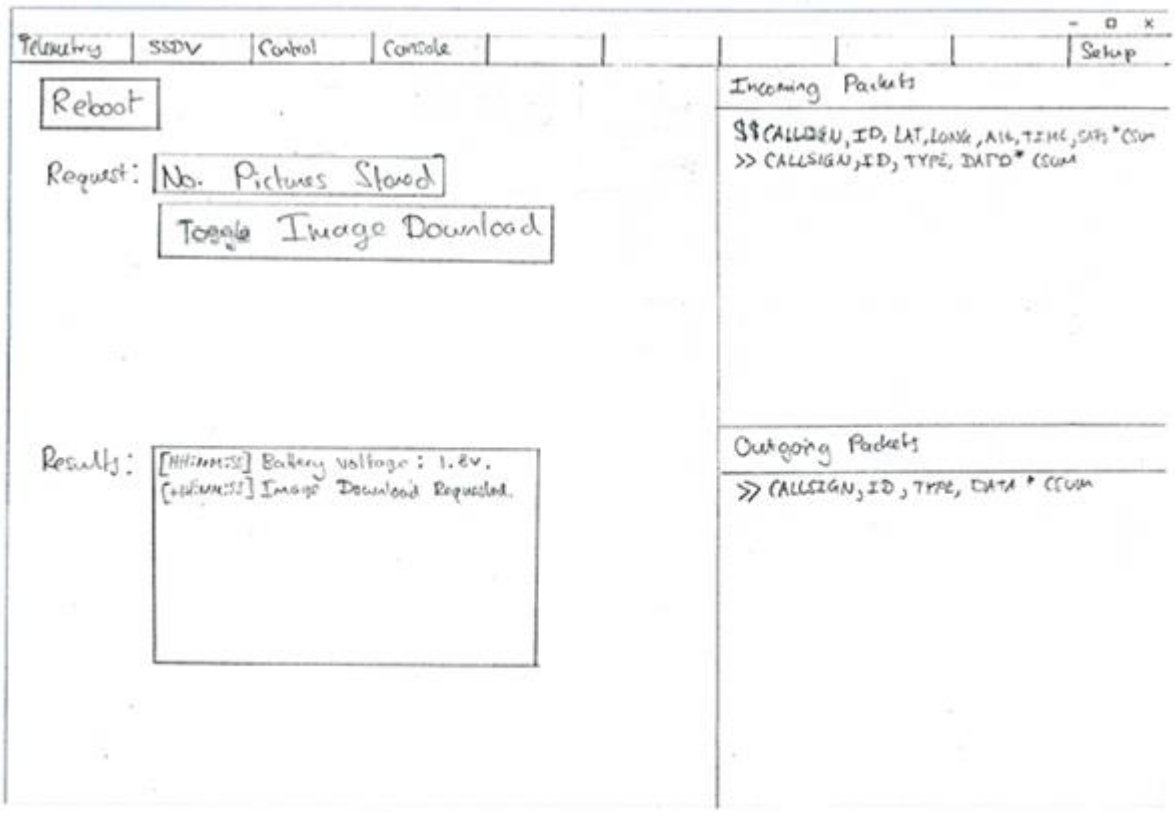
The first image below shows the telemetry screen, which screen you are currently viewing can be identified by looking at the toolbar, this toolbar will have a set of buttons, many of which take you to a specific display. The “Telemetry”, “SSDV”, “Control”, “Console” and “Setup” buttons all take you to 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.



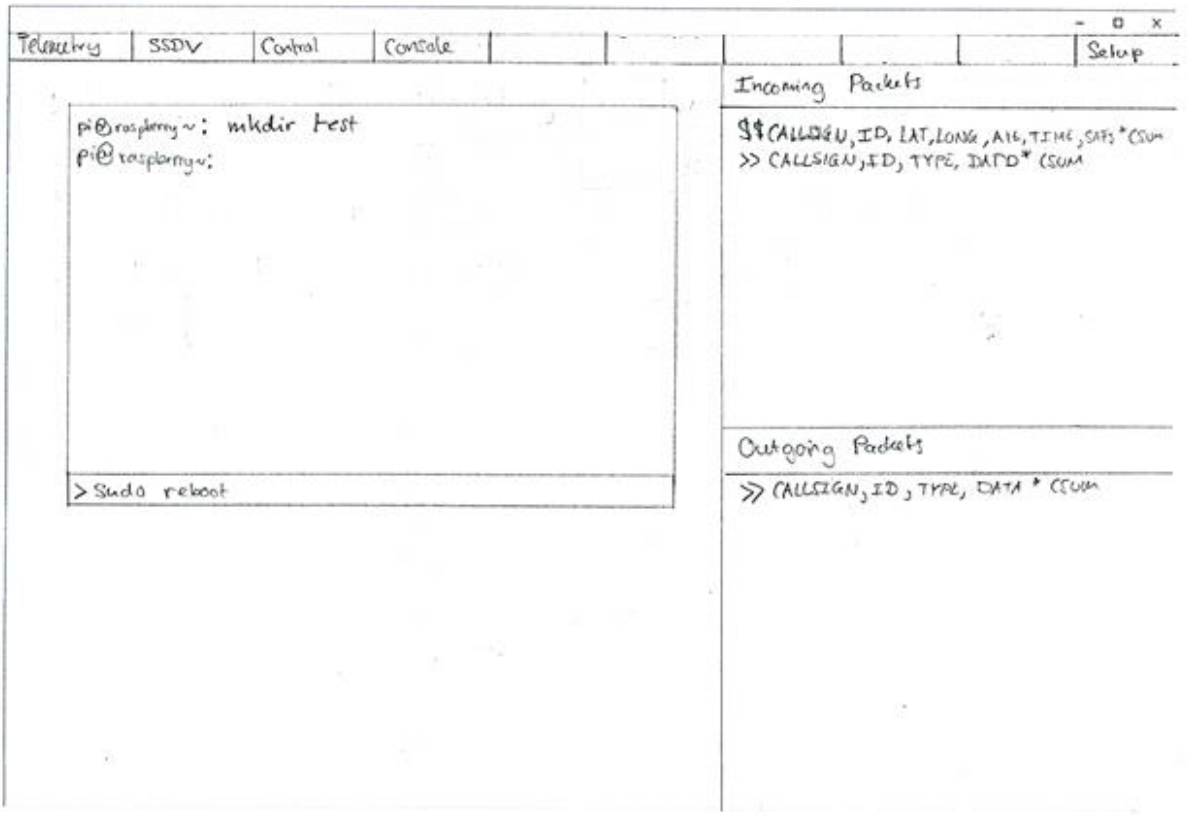
The telemetry screen above shows us the current latitude, longitude, altitude, horizontal and vertical velocities, time since last receive and the list of receivers which picked up the last packet. The white boxes will be text boxes containing non-static information.



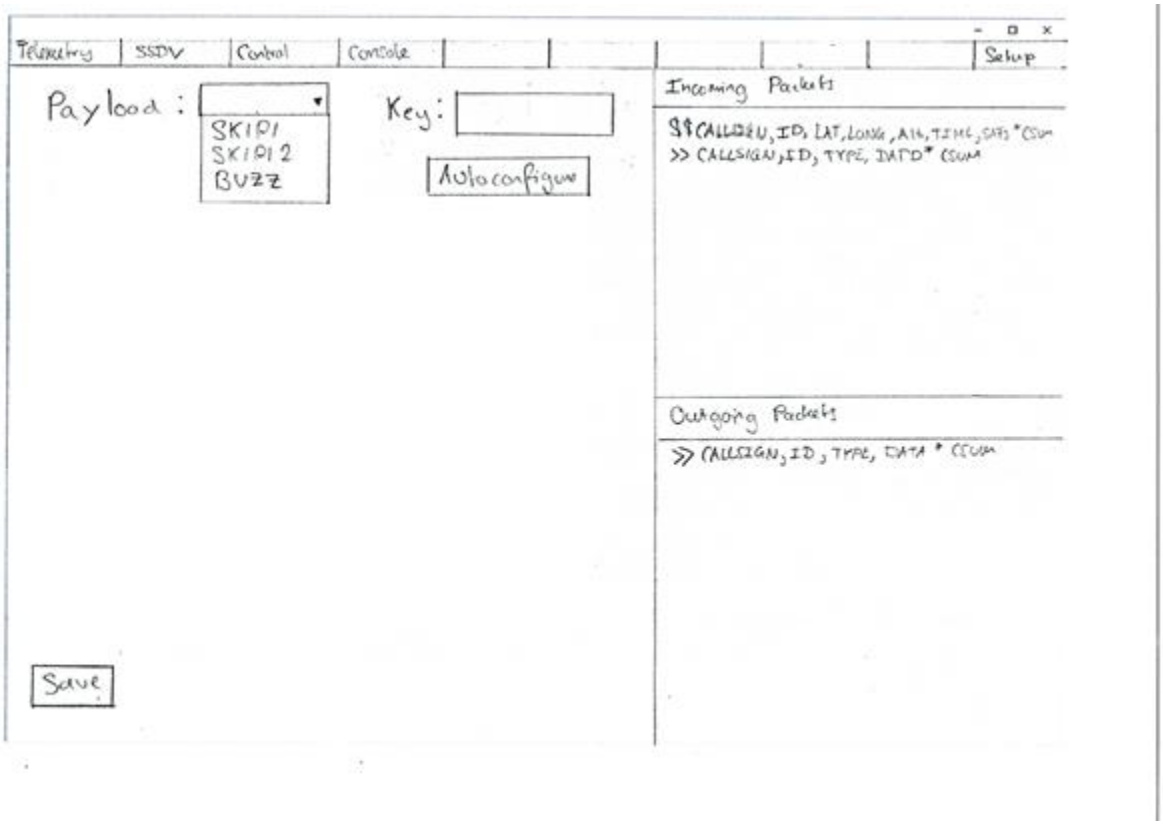
The SSDV screen shows the current image which is being downloaded from the payload in a large box and the 3 previous images in smaller boxes. The image ID and time at which the first packet was received are shown below the large image. Additionally, at the bottom of the page is a hyperlink which takes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

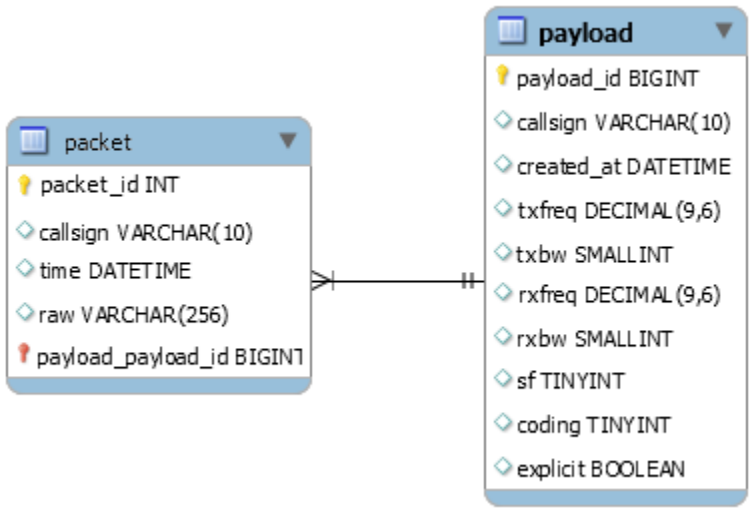


Figure 16 - Entity relationship diagram for simple 2-table database.

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.

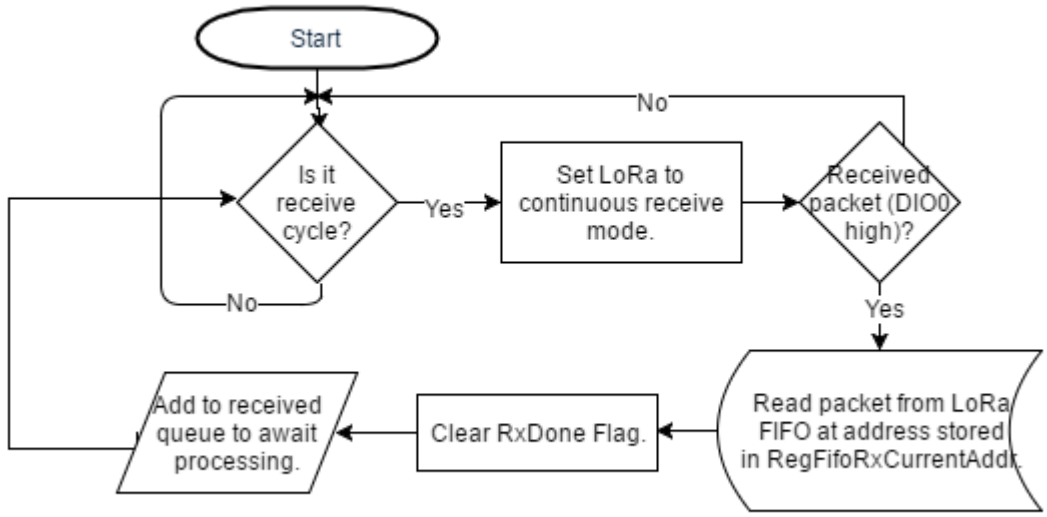
Receiving

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:



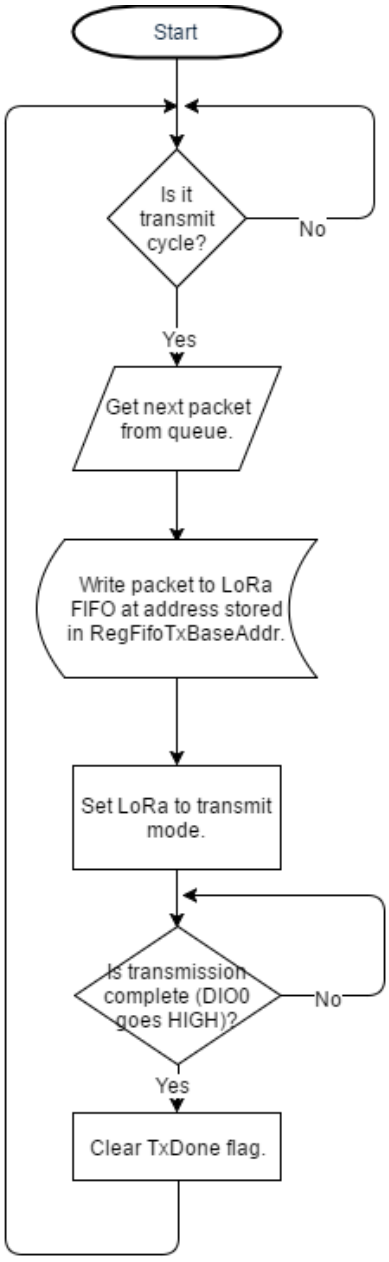
Transmission

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

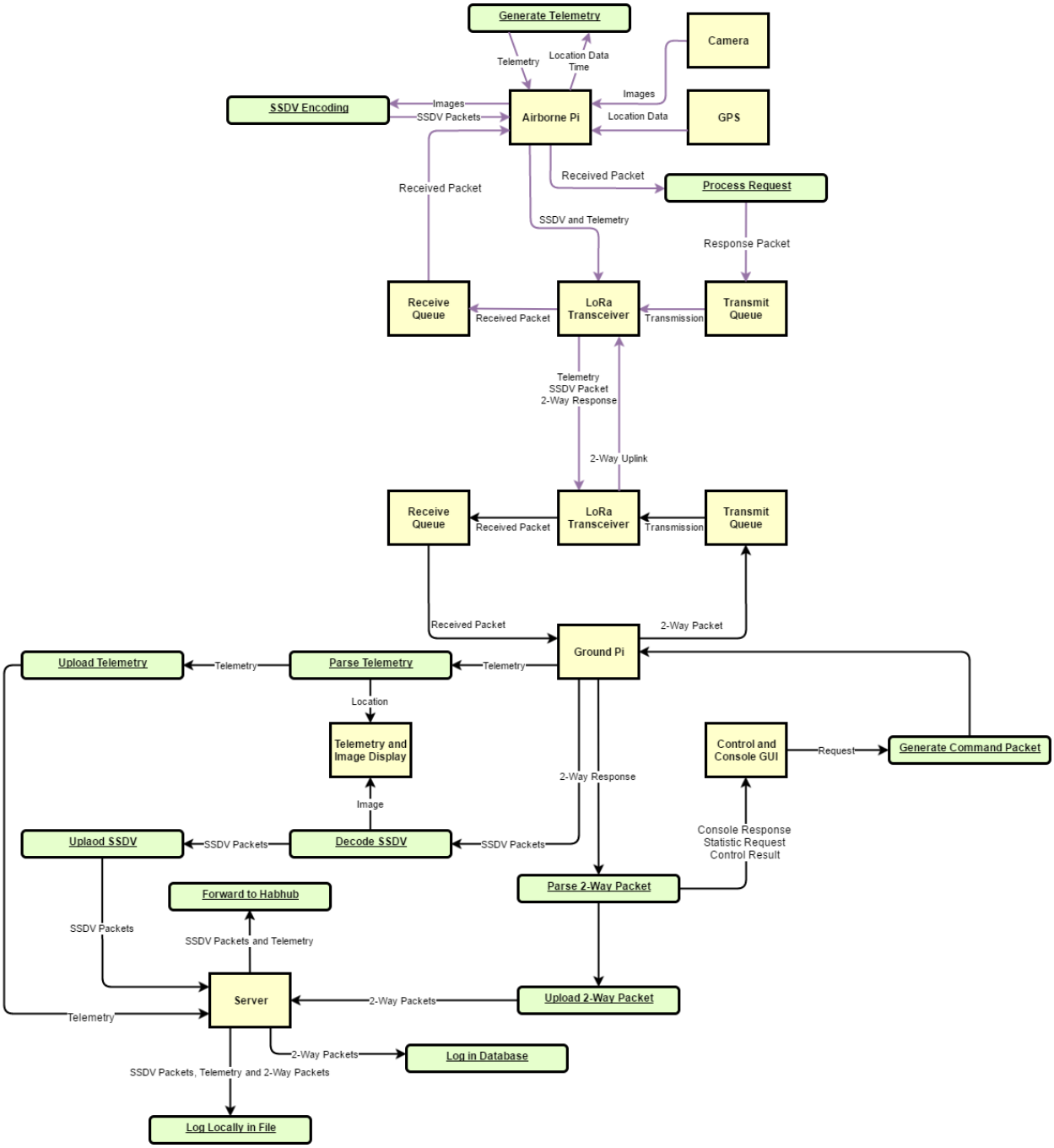


Figure 17 - Data Flow Diagram for Proposed System

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.

'Util' Module

Contains classes that are used by both the payload and the ground station, both the ground station and the payload modules depend on this module.

com.sam.hab.util.csum.CRC16CCITT.java

Contains a static method which uses hashing algorithm to generate the CRC-16-CCITT checksum of a byte array.

```
package com.sam.hab.util.csum;

public class CRC16CCITT {

    /**
     * Algorithm to generate a 16 bit Cyclic Redundancy Check checksum/hash. This
implementation uses polynomial 0x1012 and start value 0xFFFF.
     * @param val the byte array of data that a checksum is to be calculated for.
     * @return the checksum, in capital hexadecimal notation, i.e. EE56 would be a
possible checksum.
     */
    public static String calcCsum(byte[] val) {
        int crc = 0xFFFF;
        int poly = 0x1021;

        for (byte b : val) {
            for (int i = 0; i < 8; i++) {
                boolean bt = ((b >> (7-i) & 1) == 1);
                boolean c15 = ((crc >> 15 & 1) == 1);
                crc <<=1;
                if (c15 ^ bt) crc ^= poly;
            }
        }

        crc&=0xFFFF;
        return Integer.toHexString(crc).toUpperCase();
    }
}
```

com.sam.hab.util.lora.Config.java

A class which loads data from a YML file into a hash table data structure and then extracts the individual values and stores them in private fields. Also contains setter and getter methods to ensure the class is suitable encapsulated.

```
package com.sam.hab.util.lora;

import com.sam.hab.util.lora.Constants.*;
import com.sun.corba.se.impl.io.TypeMismatchException;
import org.yaml.snakeyaml.Yaml;

import java.io.*;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Config {

    private String callsign;
    private double freq;
    private double listen;
```

```
    private Bandwidth txbandwidth;
    private Bandwidth rxbandwidth;
    private short sf;
    private CodingRate codingRate;
    private String key;
    private boolean implicit;
    private byte power;

    /**
     * In the constructor the config.yml file is read and using the SnakeYAML
library converted into a HashMap.
     * The HashMap's contents are then extracted, cast to the correct types and
stored in the appropriate field.
     */
    public Config() {
        Yaml yaml = new Yaml();
        File f = new File("config.yml");
        try {
            if (!f.exists()) {
                f.createNewFile();
                FileWriter writer = new FileWriter(f);
                writer.write("callsign: TEST00\nkey: key123456\nfreq:
869.850\nlisten: 869.525\ntxbw: 250K\nrxbw: 62K5\nsf: 7\ncoding: 5\nimplicit:
false\npower: 5");
                writer.close();
            }
            BufferedReader reader = new BufferedReader(new FileReader(f));
            String s = "";
            String line = reader.readLine();
            while (line != null) {
                s += line + "\n";
                line = reader.readLine();
            }
            reader.close();
            Map<Object, Object> conf = (Map<Object, Object>)yaml.load(s);
            callsign = (String) conf.get("callsign");
            freq = (double) conf.get("freq");
            listen = (double)conf.get("listen");
            txbandwidth = Bandwidth.getBandwidth((String) conf.get("txbw"));
            rxbandwidth = Bandwidth.getBandwidth((String) conf.get("rxbw"));
            sf = (short) (int) conf.get("sf");
            codingRate = CodingRate.valueOf("CR4_" +
String.valueOf(conf.get("coding")));
            implicit = (boolean) conf.get("implicit");
            power = (byte)((int)conf.get("power"));
            key = (String)conf.get("key");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void setCallsign(String callsign) {
        this.callsign = callsign;
    }

    public void setFreq(double freq) {
        this.freq = freq;
    }

    public void setListen(double listen) {
        this.listen = listen;
    }

    public void setTxbandwidth(Bandwidth txbandwidth) {
        this.txbandwidth = txbandwidth;
    }

    public void setRxbandwidth(Bandwidth rxbandwidth) {
```

```
        this.rxbandwidth = rxbandwidth;
    }

    public void setSf(short sf) {
        this.sf = sf;
    }

    public void setCodingRate(CodingRate codingRate) {
        this.codingRate = codingRate;
    }

    public void setKey(String key) {
        this.key = key;
    }

    public void setImplicit(boolean implicit) {
        this.implicit = implicit;
    }

    /**
     * Stores all parameters in a HashMap and writes this HashMap to a file as
YAML.
     */
    public void save() {
        Yaml yaml = new Yaml();
        Map<Object, Object> conf = new HashMap<Object, Object>();
        conf.put("callsign", callsign);
        conf.put("freq", freq);
        conf.put("listen", listen);
        conf.put("txbw", Bandwidth.asString(txbandwidth));
        conf.put("rxbw", Bandwidth.asString(rxbandwidth));
        conf.put("sf", sf);
        conf.put("coding", codingRate.toString().replace("CR4_", ""));
        conf.put("implicit", implicit);
        conf.put("power", power);
        conf.put("key", key);
        File f = new File("config.yml");
        if (f.exists()) {
            f.delete();
        }
        try {
            f.createNewFile();
            FileWriter writer = new FileWriter(f);
            yaml.dump(conf,writer);
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public String getCallsign() {
        return callsign;
    }

    public double getFreq() {
        return freq;
    }

    public Bandwidth getReceiveBandwidth() {
        return rxbandwidth;
    }

    public Bandwidth getTransmitBandwidth() {
        return txbandwidth;
    }

    public short getSf() {
        return sf;
```

```
    }

    public CodingRate getCodingRate() {
        return codingRate;
    }

    public String getKey() {
        return key;
    }

    public boolean isImplicit() {
        return implicit;
    }

    public byte getPower() {
        return power;
    }

    public double getListen() {
        return listen;
    }

    public boolean getImplicit() {
        return implicit;
    }
}
```

com.sam.hab.util.lora.Constants.java

Contains all the LoRa constants, which are used when modifying/accessing registers on the radio module, as enumerations.

```
package com.sam.hab.util.lora;

/**
 * A utility class to store all me constants! These are used for the LoRa radio
operation, as it uses an SPI interface most data is sent in the form of a few
bytes.
 */
public class Constants {

    /**
     * Registers stored with their memory address.
     */
    public enum Register {
        FIFO(0x00),
        OPMODE(0x01),
        FRMSB(0x06),
        FRMID(0x07),
        FRLSB(0x08),
        PACONFIG(0x09),
        FIFOADDRPOINTER(0x0D),
        FIFOTXBASEADDR(0x0E),
        FIFORXBASEADDR(0x0F),
        FIFORXCURRENTADDR(0x10),
        IRQFLAGS(0x12),
        RXNBBYTES(0x13),
        MODEMCONFIG1(0x1D),
        MODEMCONFIG2(0x1E),
        PAYLOADLENGTH(0x22),
        DIOMAPPING1(0x40),
        DIOMAPPING2(0x41);

        public final byte addr;

        Register(int addr) {
            this.addr = (byte) addr;
```

```
        }
    }

    /**
     * Whether the DIO0 pin is configured to go HIGH (1) on TXDONE or RXDONE.
     */
    public enum DIOMode {
        TXDONE,
        RXDONE;
    }

    /**
     * The available modes and the value that the RegOpMode register needs to be
set to to set the radio to this mode.
     * Node, not all modes are here, some are not used in my program so were not
included, it would be trivial to add them of course.
     */
    public enum Mode {
        SLEEP(0x80),
        STDBY(0x81),
        TX(0x83),
        RX(0x85);

        public final byte val;

        Mode(int val) {
            this.val = (byte) val;
        }

        public static Mode lookup(byte val) {
            for (Mode mode : Mode.values()) {
                if (mode.val == val) {
                    return mode;
                }
            }
            return null;
        }

    }

    /**
     * All the available bandwidths that the radio can be set to along with the
value that must be sent to the radio to enable this bandwidth.
     */
    public enum Bandwidth {
        BW7_8(0),
        BW10_4(1),
        BW15_6(2),
        BW20_8(3),
        BW31_25(4),
        BW41_7(5),
        BW62_5(6),
        BW125(7),
        BW250(8),
        BW500(9);

        public final byte val;

        Bandwidth(int val) {
            this.val = (byte)val;
        }

        public static Bandwidth getBandwidth(String bw) {
            switch (bw) {
                case "7K8":
                    return BW7_8;
                case "10K4":
                    return BW10_4;
```

```
            case "15K6":
                return BW15_6;
            case "20K8":
                return BW20_8;
            case "21K25":
                return BW31_25;
            case "41K7":
                return BW41_7;
            case "62K5":
                return BW62_5;
            case "125K":
                return BW125;
            case "250K":
                return BW250;
            case "500K":
                return BW500;
        }
        return null;
    }

    public static String asString(Bandwidth bandwidth) {
        switch (bandwidth) {
            case BW7_8:
                return "7K8";
            case BW10_4:
                return "10K4";
            case BW15_6:
                return "15K6";
            case BW20_8:
                return "20K8";
            case BW31_25:
                return "21K25";
            case BW41_7:
                return "41K7";
            case BW62_5:
                return "62K5";
            case BW125:
                return "125K";
            case BW250:
                return "250K";
            case BW500:
                return "500K";
        }
        return null;
    }

    public static Bandwidth lookup(int txBw) {
        for (Bandwidth bw : Bandwidth.values()) {
            if (bw.val == txBw) {
                return bw;
            }
        }
        return null;
    }
}

/**
 * The error coding rate of the transmitted packet.
 */
public enum CodingRate {
    CR4_5(1),
    CR4_6(2),
    CR4_7(3),
    CR4_8(4);

    public final byte val;

    CodingRate(int val) {
```

```
            this.val = (byte)val;
        }
    }

    /**
     * The available types of packet that can be sent/received using my 2-way
packet protocol.
     */
    public enum PacketType {
        CMD(0),
        SHELL(1),
        DIAG(2),
        OTHER(5);

        public final int id;

        PacketType(int id) {
            this.id = id;
        }

        public static PacketType lookup(int i) {
            for (PacketType type : PacketType.values()) {
                if (type.id == i) {
                    return type;
                }
            }
            return null;
        }
    }
}
```

com.sam.hab.util.lora.LoRa.java

This is an encapsulated class which provides an interface for a LoRa module, it contains all the methods for setting and getting data from the various registers and changing radio settings and operation mode. Most of these methods use complex bitwise logic and are commented in detail explaining what's going on.

```
package com.sam.hab.util.lora;

import com.pi4j.io.gpio.*;
import com.pi4j.io.spi.SpiChannel;
import com.pi4j.io.spi.SpiDevice;
import com.pi4j.io.spi.SpiFactory;

import com.pi4j.wiringpi.Gpio;
import com.sam.hab.util.lora.Constants.*;
import com.sam.hab.util.txrx.CycleManager;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.Arrays;

public class LoRa {

    private double frequency;
    private Bandwidth bandwidth;
    private short spreadingFactor;
    private CodingRate codingRate;
    private boolean explicitHeader;
    private Mode mode;
    private final GpioController gpio;
    private SpiDevice spi = null;
    private DIOMode dioMapping = DIOMode.RXDONE;

    /**
```

```
     * Constructor for the LoRa interface class. Parameters are the modem settings
for the radio.
     * This encapsulated class contains all the advanced LoRa register modification
functionality, providing me, as the developer, with a simpler interface elsewhere
in the program.
     * @param frequency frequency to set the radio to.
     * @param bandwidth bandwidth  to set the radio to.
     * @param spreadingFactor sf to set the radio to.
     * @param codingRate cr to set the radio to.
     * @param explicitHeader whether or not to use explicit headers (please use
explicit headers!).
     */
    public LoRa(double frequency, Bandwidth bandwidth, short spreadingFactor,
CodingRate codingRate, boolean explicitHeader) throws IOException {
        this.frequency = frequency;
        this.bandwidth = bandwidth;
        this.spreadingFactor = spreadingFactor;
        this.codingRate = codingRate;
        this.explicitHeader = explicitHeader;

        spi = SpiFactory.getInstance(SpiChannel.CS1, SpiDevice.DEFAULT_SPI_SPEED,
SpiDevice.DEFAULT_SPI_MODE);
        gpio = GpioFactory.getInstance();

        setMode(Mode.SLEEP);

        setDIOMapping(DIOMode.RXDONE);

        setFrequency(frequency);
        setModemConfig(bandwidth, spreadingFactor, codingRate, explicitHeader);
        setPAConfig((byte)0x08); //This is the default value which equates to
~10mW.
        clearIRQFlags();
    }

    /**
     * Called when packet is received.
     */
    public byte[] handlePacket() {
        try {
            byte[] payload = readPayload();
            resetRXPtr();
            setMode(Mode.RX);
            return payload;
        } catch (IOException e) {
        }
        return null;
    }

    /**
     * Used for all writing of registers via the SPI interface.
     * @param reg The register to write to.
     * @param values A byte array of values to write to the regsister.
     * @throws IOException if write operation fails, for example if the radio is
disconnected unexpectedly.
     */
    private void writeRegister(Register reg, byte... values) throws IOException {
        System.out.println(reg.toString());
        byte[] send = new byte[values.length + 1];
        send[0] = (byte)(reg.addr | 0x80);
        System.arraycopy(values, 0, send, 1, values.length);
        System.out.println("W:" + Arrays.toString(send));
        spi.write(send);
    }

    /**
     * Read the value held in a register.
     * @param reg The register to read from.
```

```
     * @param nbBytes The number of bytes to read.
     * @return The value read from the register (a byte array).
     * @throws IOException
     */
    private byte[] readRegister(Register reg, int nbBytes) throws IOException {
        System.out.println(reg.toString());
        byte[] send = new byte[nbBytes + 1];
        send[0] = (byte)reg.addr;
        byte[] out = spi.write(send);
        System.out.println("R:" + Arrays.toString(out));
        return out;
    }

    /**
     * Set Frequency in MHz.
     * @param frequency Frequency to set the radio to.
     * @throws IOException
     */
    public void setFrequency(double frequency) throws IOException {
        assert this.mode == Mode.SLEEP || this.mode == Mode.STDBY; //Assertion used
as these conditions should never not be true.
        int freq = (int)(frequency * (7110656 / 434));
        //This method separates the frequency into the 8 most significant bits, the
8 middle bits and the 8 least significant bits as they are stored in separate
registers.
        writeRegister(Register.FRMSB, (byte)((freq >> 16) & 0xFF));
        writeRegister(Register.FRMID, (byte)((freq >> 8) & 0xFF));
        writeRegister(Register.FRLSB, (byte)(freq & 0xFF));
    }

    /**
     * Set the operation mode of the LoRa radio.
     * @param mode Mode to change to.
     * @throws IOException
     */
    public void setMode(Mode mode) throws IOException {
        if (this.mode != mode) {
            this.mode = mode;
        }
        writeRegister(Register.OPMODE, mode.val);
        if (this.mode != Mode.SLEEP) {
            long time = System.currentTimeMillis();
            while (Gpio.digitalRead(26) != 1) {
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                }
                if (System.currentTimeMillis() - time > 500) {
                    return;
                }
            }
            clearIRQFlags();
        }
    }

    /**
     * Returns to the user the operation mode that the radio is currently operating
in.
     * @return The mode that the radio is currently operating in.
     * @throws IOException
     */
    public Mode getMode() throws IOException {
        byte reg = readRegister(Register.OPMODE, 1)[1];
        Mode mode = Mode.lookup(reg);
        if (mode != null) {
            this.mode = mode;
        }
        return mode;
```

```
    }

    /**
     * Used to configure modem settings, modifies the appropriate registers via the
SPI interface. Can be used to edit bandwidth, spreading factor, coding rate and
header mode.
     * @param bandwidth Bandwidth to set the radio to.
     * @param spreadingFactor New spreading factor.
     * @param codingRate New CRC error coding rate.
     * @param explicitHeader Whether to use explicit header mode.
     * @throws IOException
     */
    public void setModemConfig(Bandwidth bandwidth, short spreadingFactor,
CodingRate codingRate, boolean explicitHeader) throws IOException {
        byte val1 = (byte)((explicitHeader ? 0 : 1) | (codingRate.val << 1) |
bandwidth.val << 4);
        byte val2 = (byte)(0b00000000 | (spreadingFactor << 4));
        writeRegister(Register.MODEMCONFIG1, val1);
        writeRegister(Register.MODEMCONFIG2, val2);

    }

    /**
     * Used to write the packet to be sent to the LoRa FIFO.
     * @param str Payload to send. Must be a string encoded with ISO 8859-1.
     */
    public void writePayload(String str) throws IOException {
        byte[] payload = str.getBytes(StandardCharsets.ISO_8859_1);
        setMode(Mode.STDBY);
        if (payload.length < 256) {
            setPayloadLength((short)payload.length);
            byte baseAddr = readRegister(Register.FIFOTXBASEADDR, 1)[1];
            setFifoPointer(baseAddr);
            writeRegister(Register.FIFO, payload);
        }
    }

    /**
     * Set the pointer of the fifo to the given address.
     * @param ptr
     * @throws IOException
     */
    public void setFifoPointer(byte ptr) throws IOException {
        writeRegister(Register.FIFOADDRPOINTER, ptr);
    }

    /**
     * Used to set the packet length of the packet that is about to be written to
the LoRa FIFO.
     * Must not exceed 255.
     * @param length
     * @throws IOException
     */
    public void setPayloadLength(short length) throws IOException {
        writeRegister(Register.PAYLOADLENGTH, (byte)length);
    }

    /**
     * Set the power output configuration, see documentation on power output
limitations.
     * @param outputPower The output power register value.
     * @throws IOException
     */
    public void setPAConfig(byte outputPower) throws IOException {
        outputPower &= 0b00001111;
        //MSB is a 1 to set the PA_BOOST pin to HIGH. The 4 LSBs are the output
power value. The other 3 bits need to be 0 for my implementation.
        byte val = (byte)0b10000000;
```

```
        val |= outputPower;
        writeRegister(Register.PACONFIG, val);
    }

    /**
     * Read a received payload from the radio.
     * @return the payload.
     * @throws IOException
     */
    public byte[] readPayload() throws IOException {
        int nbBytes = 0xFF & readRegister(Register.RXNBBYTES, 1)[1];
        byte fifoRxCurrentAddr = readRegister(Register.FIFORXCURRENTADDR, 1)[1];
        setFifoPointer(fifoRxCurrentAddr);
        byte[] payload = Arrays.copyOfRange(readRegister(Register.FIFO, nbBytes),
1, nbBytes+1);
        return payload;
    }

    /**
     * See below.
     * @return Array of booleans for IRQ flags in this order: txdone, rxdone.
     * @throws IOException
     */
    public boolean[] getIRQFlags() throws IOException {
        byte val = readRegister(Register.IRQFLAGS, 1)[1];
        boolean[] flags = new boolean[2];
        flags[0] = ((val >> 7) & 0x01) == 1;
        flags[1] = ((val >> 3) * 0x01) == 1;
        return flags;
    }

    /**
     * Clear all IRQ flags. This also stops the DIO pins from being triggered.
     * @throws IOException
     */
    public void clearIRQFlags() throws IOException {
        writeRegister(Register.IRQFLAGS, (byte)0);
    }

    /**
     * Set the dio mapping, takes a byte[] of length 6.
     * @param mode Mapping.
     * @throws IOException
     */
    public void setDIOMapping(DIOMode mode) throws IOException {
        this.dioMapping = mode;
        if (mode == DIOMode.RXDONE) {
            writeRegister(Register.DIOMAPPING1, (byte)0x00);
        } else if (mode == DIOMode.TXDONE) {
            writeRegister(Register.DIOMAPPING1, (byte)0x40);
        }
        writeRegister(Register.DIOMAPPING2, (byte)0x00);
    }

    /**
     *  Reset the pointer in the FIFO where received packets are stored.
     * @throws IOException
     * @throws InterruptedException
     */
    public void resetRXPtr() throws IOException {
        setMode(Mode.SLEEP);
        byte baseAddr = readRegister(Register.FIFORXBASEADDR, 1)[1];
        setFifoPointer(baseAddr);
        //setMode(Mode.STDBY);
    }

    /**
     * Check the state of the DIO0 pin.
```

```
     * @return whether the pin is at HIGH (true) or LOW (false).
     */
    public boolean pollDIO0() {
        return Gpio.digitalRead(27) == 1;
    }

    /**
     * Check the state of the DIO5 pin.
     * @return whether the pin is at HIGH (true) or LOW (false).
     */
    public boolean pollDIO5() {
        return Gpio.digitalRead(26) == 1;
    }

    /**
     * Set the radio to begin transmission of some packets.
     * @param transmitList array of packets to transmit.
     * @throws IOException
     */
    public void send(String[] transmitList) throws IOException {
        setDIOMapping(DIOMode.TXDONE);
        int transmitPtr = 0;
        while (transmitPtr < transmitList.length) {
            if (transmitList[transmitPtr] != null) {
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                writePayload(transmitList[transmitPtr]);
                setMode(Mode.TX);
            }
            transmitPtr++;
            long time = System.currentTimeMillis();
            while (Gpio.digitalRead(27) != 1) {
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                }
                if (System.currentTimeMillis() - time > 500) {
                    break;
                }
            }
            clearIRQFlags();
        }
    }

    @Override
    public String toString() {
        return "LoRa Module Setup:\nFrequency: " + frequency + "\nBandwidth: " +
bandwidth.toString() + "\nSpreading Factor: " + spreadingFactor + "\nCoding Rate: "
+ codingRate.toString() + "\nExplicit Header: " + explicitHeader + "\nMode: " +
mode.toString();
    }
}
```

com.sam.hab.util.txrx.CycleManager.java

This is the manager class that oversees the radio's operation cycle, switching between transmit and receive mode. Note that this class is polymorphic and has some methods implemented differently in the ground station and the payload modules.

```
package com.sam.hab.util.txrx;

import com.sam.hab.util.csum.CRC16CCITT;
import com.sam.hab.util.lora.Constants.*;
import com.sam.hab.util.lora.LoRa;
```

```
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.*;

import static com.sam.hab.util.lora.Constants.Mode.TX;

public abstract class CycleManager {

    /*
     * This class manages the switching between transmission and receiving modes of
the radio and manages the handling of all received packets and the preparation for
sending for all packets.
     */

    //
    private Queue<String> transmitQueue = new LinkedList<String>();
    private Queue<String> receiveQueue = new LinkedList<String>();

    private final boolean payload;

    private final LoRa lora;

    private final double[] freq;
    private final Bandwidth[] bandwidth;
    private final short sf;
    private final CodingRate codingRate;
    private final boolean explicit;

    protected final String callSign;
    protected final String key;

    public CycleManager(boolean payload, String callSign, double[] frequency,
Bandwidth[] bandwidth, short sf, CodingRate codingRate, boolean explicit, byte
power, String key) { //Explaining frequency and bandwidth arrays: the 0 index is
for transmit, the 1 index is for receive.
        this.payload = payload;
        this.freq = frequency;
        this.bandwidth = bandwidth;
        this.sf = sf;
        this.codingRate = codingRate;
        this.explicit = explicit;
        this.callSign = callSign;
        this.key = key;
        try {
            lora = new LoRa(freq[0], bandwidth[0], sf, codingRate, explicit);
            lora.setPAConfig(power);
        } catch (IOException e) {
            throw new RuntimeException("LoRa module contact not established, check
your wiring perhaps?");
        }

        Thread packetThread = new Thread(new PacketHandler(this));
        packetThread.start();
    }

    /**
     * Add a packet to the transmit queue.
     * @param payload Packet to add, must be a string using only ISO 8859-1
characters.
     */
    public void addToTx(String payload) {
        transmitQueue.add(payload);
    }

    /**
     * Get the next packet from the received packet queue.
```

```
     * @return The packet, or null if there are none.
     */
    public String getNextReceived() {
        if (receiveQueue.size() > 0) {
            return receiveQueue.poll();
        }
        return null;
    }

    //Flag to determine when to transmit.
    private boolean transmit = false;

    /**
     * Sets the transmit flag to true so that the mainloop switches to transmit
mode on the next iteration, this is only actually used on the ground station.
     */
    public void txInterrupt() {
        transmit = true;
    }

    //Sets whether to transmit images, used only by the payload (obviously!) when
the ground station has requested an image transmission toggle.
    private boolean image = true;

    /**
     * Toggles image transmission for the payload.
     */
    public boolean toggleImage() {
        if (!payload) {
            return false;
        }
        image = !image;
        return image;
    }

    /**
     * This is the main loop for the program, switches between transmit mode and
receive mode as necessary.
     * The payload switches between transmit and receive in fixed intervals, the
ground station is always receiving unless instructed otherwise by the ground
payload.
     * @param startMode
     */
    public void mainLoop(Mode startMode) {
        Mode newMode = startMode;
        while (true) {
            try {
                if (newMode == TX) {
                    transmit();
                    newMode = Mode.RX;
                } else if (newMode == Mode.RX) {
                    receive();
                    if (transmit || payload) {
                        newMode = Mode.TX;
                    } else {
                        newMode = Mode.RX;
                    }
                }
            } catch (IOException e) {
                System.out.println("Unexpected IO exception while running main
loop.");
                System.out.println(lora);
            }
        }
    }

    /**
     * This method will receive packets until the transmit flag goes true or it
```

```
times out, there is a small timeout so that we do not have significant downtime not
transmitting.
     * Received packets are stored in the received packet queue using the addToRx()
method.
     * @throws IOException
     */
    private void receive() throws IOException {
        lora.setMode(Mode.STDBY);
        lora.setFrequency(freq[1]);
        lora.setModemConfig(bandwidth[1], sf, codingRate, explicit);
        lora.setDIOMapping(DIOMode.RXDONE);
        lora.setMode(Mode.RX);
        long timeout = System.currentTimeMillis() + 10000;
        while (System.currentTimeMillis() < timeout && !transmit) {
            while (!lora.pollDIO0() && !transmit && System.currentTimeMillis() <
timeout) {
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            if (lora.pollDIO0()) {
                lora.clearIRQFlags();
                byte[] payload = lora.handlePacket();
                if (payload != null) {
                    addToRx(payload);
                    timeout = timeout + 1000;
                }
            }
        }
    }

    /**
     * Acquires the next set of packets to transmit and sends them to the LoRa
radio to transmit.
     * If this is the payload it will transmit 10 2-way packets (if available, if
not available these will just be telemetry),
     * then 10 telemetry then if image transmission is enabled it will transmit 70
image packets.
     * If this is the ground station it will transmit up to 10 2-way packets,
however, once all 2-way packets are transmitted
     * it'll just stop so as to return to telemetry and image sending by the
payload quickly..
     * @throws IOException
     */
    private void transmit() throws IOException {
        transmit = false;
        lora.setMode(Mode.STDBY);
        lora.setFrequency(freq[0]);
        lora.setModemConfig(bandwidth[0], sf, codingRate, explicit);
        String[] transmit = new String[(payload ? 90 : 10)];
        for (int i = 0; i < 10; i++) {
            if (transmitQueue.size() <= 0) {
                if (payload) {
                    transmit[i] = getTelemetry();
                } else {
                    transmit[i] = null;
                }
            } else {
                transmit[i] = doPacket(transmitQueue.poll(), String.valueOf(i),
key);
            }
        }
        if (payload) {
            for (int i = 10; i < 20; i++) {
                transmit[i] = getTelemetry();
            }
```

```
            for (int i = 20; i < 89; i++) {
                if (image) {
                    transmit[i] = getImagePacket();
                } else {
                    transmit[i] = null;
                }
            }
            transmit[89] = doPacket(TwoWayPacketGenerator.generateCommand(callSign,
"TRA"), String.valueOf(89), key);
        }
        for (String pckt : transmit) {
            onSend(pckt);
        }
        lora.send(transmit);
    }

    /**
     * This method takes a 2-way packet and configures it for sending by
substituting its ID into place and then appending the checksum.
     * @param packet The packet to prepare, this is a string.
     * @param id The id of the packet.
     * @param key The encryption key set by user in config.
     * @return The prepared packet with the checksum and id substituted in.
     */
    private String doPacket(String packet, String id, String key) {
        packet = packet.replace(">>","");
        packet = packet.replaceFirst("%s",id);
        return ">>" + packet + "*" + CRC16CCITT.calcCsum((packet + key).getBytes())
+ "\n";
    }

    /**
     * Adds the given payload to the received queue.
     * @param payload Payload to add to the queue.
     */
    public void addToRx(byte[] payload) {
        receiveQueue.add(new String(payload, StandardCharsets.ISO_8859_1));
    }

    /**
     * Below are several abstract methods which are used to allow this class to be
polymorphic. These methods are implemented differently on the payload and the
ground station.
     * For example, the payload does not need to be able to handle received
telemetry or image packets, and the ground station doesn't need to be able to
generate telemetry or image packets.
     */

    public abstract void handleTelemetry(ReceivedTelemetry telem);

    public abstract void onSend(String sent);

    public abstract void handleImage(byte[] bytes, int iID, int pID);

    public abstract void handleTwoWay(ReceivedPacket packet);

    public abstract String getTelemetry();

    public abstract String getImagePacket();
}
```

com.sam.hab.util.txrx.PacketHandler.java

This class runs a continuous loop that waits for there to be packets in the received packet queue, pops that item from the queue and determines the packet type and then calls the

appropriate handling parsing from the PacketParser class and the appropriate handling method from the CycleManager.

```
package com.sam.hab.util.txrx;

import java.nio.charset.StandardCharsets;
import java.util.Calendar;

public class PacketHandler implements Runnable {

    private final CycleManager cm;

    /**
     * This is a simple loop which gets the latest packet from the received queue,
determines its type and then calls the relevant parse and handle methods.
     * @param cm The cycle manager currently in use, needed in order to get latest
received packet.
     */
    public PacketHandler(CycleManager cm) {
        this.cm = cm;
    }

    @Override
    public void run() {
        Calendar cal = Calendar.getInstance();
        while (true) {
            try {
                String packet = cm.getNextReceived();
                if (packet != null) {
                    byte[] bytes = packet.getBytes(StandardCharsets.ISO_8859_1);
                    if (packet.length() > 0) {
                        if (packet.charAt(0) == '>') {
                            ReceivedPacket pckt = PacketParser.parseTwoWay(packet,
cm.key);
                            cm.handleTwoWay(pckt);
                        } else if (packet.charAt(0) == '$') {
                            ReceivedTelemetry telem =
PacketParser.parseTelemetry(packet);
                            if (telem != null) {
                                cm.handleTelemetry(telem);
                            }
                        } else {
                            int[] res = PacketParser.parseSSDV(bytes);
                            try {
                                cm.handleImage(bytes, res[0], res[1]);
                            } catch (NullPointerException e) {
                                e.printStackTrace();
                            }
                        }
                    }
                }
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                }
            } catch (Exception e) {
                //e.printStackTrace();
            }
        }
    }
}
```

com.sam.hab.util.txrx.PacketParser.java

Parses SSDV, telemetry and 2-way packets to ensure the checksum is valid (and hence that the packet is from a trusted source) and that the data follows the correct format.

```
package com.sam.hab.util.txrx;

import com.sam.hab.util.lora.Constants.*;
import com.sam.hab.util.csum.CRC16CCITT;
import sun.misc.CRC16;

import java.io.*;
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

public class PacketParser {

    /**
     * Simple algorithm to parse a two way packet, will first check the checksum is
valid and that the correct key has been used then it will then return a new
ReceivedPacket object with this packet's data.
     * @param raw The string of the packet received from the radio.
     * @param key The encryption key, set in config.
     * @return The ReceivedPacket object for this packet. Or null if the packet
failed the checksum/key test.
     */
    public static ReceivedPacket parseTwoWay(String raw, String key) {
        String cSum = raw.split("\\*")[1].replace("\n", "");
        String packet = raw.split("\\*")[0].replace(">", "");;
        if (!CRC16CCITT.calcCsum((packet.replace(">", "") +
key).getBytes()).equals(cSum)) {
            return null;
        }
        String[] packetList = packet.split(",");
        PacketType packetType = PacketType.lookup(Integer.valueOf(packetList[2]));
        if (packetType == null) {
            return null;
        }
        return new ReceivedPacket(raw, packetList[3],
Integer.valueOf(packetList[1]), packetType);
    }

    /**
     * Parses telemetry by splitting by comma, this works for the standard UKHAS
format setup as $$CALLSIGN,ID,HH:MM:SS,LAT,LON,SATS*CSUM\n only.
     * @param raw The string of the telemetry taken from the radio.
     * @return The ReceivedTelemetry object for this telemetry string, this will
then be sent to the server and the display.
     */
    public static ReceivedTelemetry parseTelemetry(String raw) {
        String cSum = raw.split("\\*")[1].replace("\n", "");
        String packet = raw.split("\\*")[0].replace("$", "");
        if (!CRC16CCITT.calcCsum((packet).getBytes()).equals(cSum)) {
            return null;
        }
        String packetList[] = packet.split(",");
        return new ReceivedTelemetry(raw, Float.valueOf(packetList[3]),
Float.valueOf(packetList[4]), Float.valueOf(packetList[5]),
Long.valueOf(packetList[1]));
    }

    /**
     * This parses SSDV data, it first stores the bytes received in the appropriate
image file, the image number is the 7th byte in the array of each packet.
     * The ssdv program by fsphil is then run to decode the SSDV into a jpg image.
     * @param in The bytes of one SSDV packet.
     * @return The image number (i.e. 7th item in the input array) and the packet
ID, also derived from the packet data.
     */
    public static int[] parseSSDV(byte[] in) {
        byte[] bytes = new byte[in.length+1];
        bytes[0] = 0x55;
        System.arraycopy(in, 0, bytes, 1, in.length);
```

```
        int imageNo = (0xFF & bytes[6]);
        int packetNo = (0xFF & bytes[7]) * 256 + (0xFF & bytes[8]);
        FileOutputStream fos = null;
        File file = new File("images/image_" + String.valueOf(imageNo) + ".bin");
        if (!file.getParentFile().exists()) {
            file.getParentFile().mkdirs();
        }
        Runtime rt = Runtime.getRuntime();
        try {
            file.createNewFile();
            fos = new FileOutputStream(file, true);
            fos.write(bytes);
            fos.close();
            Process pr = rt.exec("./ssdv -d images/image_" +
String.valueOf(imageNo) + ".bin images/current.jpg");
            pr.waitFor();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {

        }
        return new int[] {imageNo, packetNo};
    }
}
```

com.sam.hab.util.txrx.ReceivedPacket.java

Simple class which contains data about a received packet, an instance of this is created every time a valid packet is received and is then passed around the program.

```
package com.sam.hab.util.txrx;

import com.sam.hab.util.lora.Constants.*;

public class ReceivedPacket {

    public final String raw;
    public final String data;
    public final int id;
    public final PacketType type;

    /**
     * Simple class, objects of which are generated automatically each time a 2-way
communications packet is received.
     * @param raw the raw string of the packet.
     * @param data the data section of the packet.
     * @param id the id of the packet.
     * @param type the type of the packet.
     */
    public ReceivedPacket(String raw, String data, int id, PacketType type) {
        this.raw = raw;
        this.data = data;
        this.id = id;
        this.type = type;
    }
}
```

com.sam.hab.util.txrx.ReceivedTelemetry.java

Simple class which contains data about a received telemetry sentence, an instance of this is created every time a valid telemetry sentence is received and is then passed around the program.

```
package com.sam.hab.util.txrx;

public class ReceivedTelemetry {

    public final String raw;
    public final float lat;
    public final float lon;
    public final float alt;
    public final long id;

    /**
     * Simple class, objects of which are generated automatically each time a
telemetry sentence is received.
     * @param raw the raw sentence.
     * @param lat the latitude which was included in the sentence.
     * @param lon the longitude which was included in the sentence.
     * @param alt the altitude which was in that sentence.
     * @param id the id of the sentence, should be number of seconds since 1/1/1970
at the time the packet was transmitted.
     */
    public ReceivedTelemetry(String raw, float lat, float lon, float alt, long id)
{
        this.raw = raw;
        this.lat = lat;
        this.lon = lon;
        this.alt = alt;
        this.id = id;
    }
}
```

com.sam.hab.util.txrx.TwoWayPacketGenerator.java

Simple class with several static method that generates the various different possible types of 2-way request packet. It doesn't include the packet ID or the checksum as these are calculated when the packet is prepared for transmission. This contains methods to generate shell packets, shell command packets, statistic packets and request/command packets.

```
package com.sam.hab.util.txrx;

public class TwoWayPacketGenerator {

    /**
     * Generates a shell response packet, used by the payload to encode the result
of a command executed in response to a remote shell command packet.
     * @param callsign the payload callsign.
     * @param response the result of executing the command.
     * @return an array of the packets which are to be transmitted to send the
response.
     */
    public static String[] generateShellPackets(String callsign, String[] response)
{
        for (int i = 0; i < response.length; i++) {
            String pckt = ">>" + callsign + ",%s,1," + response[i].replace('\n',
(char)0);
            response[i] = pckt;
        }
        return response;
    }

    /**
     * Generates a shell command packet to be sent to the payload to cause remote
execution of a shell command.
     * @param callsign payload callsign.
     * @param cmd the command to execute, cannot include asterisks.
     * @return the packet.
     */
```

```
    public static String generateShellCmdPacket(String callsign, String cmd) {
        return ">>" + callsign + ",%s,1," + cmd;
    }

    /**
     * Generates a stat packet, this is sent by the payload in response to a
statistic request by the ground station.
     * @param callsign payload callsign.
     * @param name the name of the stat/
     * @param stat the stat.
     * @return the packet.
     */
    public static String generateStatPacket(String callsign, String name, String
stat) {
        return ">>" + callsign + ",%s,2," + name + "/" + stat;
    }

    /**
     * Generates a command packet, this is a packet that it sent to the payload
requesting that it do something, this could be RBT (reboot) or many other things.
     * This is also used once by the payload to generate the TRA packet that
enables the ground station to begin transmitting.
     * @param callsign payload callsign.
     * @param cmd the command.
     * @return the packet.
     */
    public static String generateCommand(String callsign, String cmd) {
        return ">>" + callsign + ",%s,0," + cmd;
    }
}
```

'Ground' Module

This module contains all the classes that make up the ground station software. It depends on the Util module.

com.sam.hab.ground.gui.GUI.java

This class handles the GUI and hence has methods for dealing with button clicks, the console and displaying received and transmitted data. The GUI.form XML file contains the details of the GUI. This class also handles the logging of transmission data to a file and the auto configuration utility. This class makes use of the event based programming paradigm. This method initialises the CycleManager with the ground station's versions of CycleManager's polymorphic methods.

```
package com.sam.hab.ground.gui;

import com.sam.hab.ground.web.RequestHandler;
import com.sam.hab.util.lora.Config;
import com.sam.hab.util.lora.Constants;
import com.sam.hab.util.txrx.CycleManager;
import com.sam.hab.util.txrx.ReceivedPacket;
import com.sam.hab.util.txrx.ReceivedTelemetry;
import com.sam.hab.util.txrx.TwoWayPacketGenerator;

import javax.swing.*;
import javax.swing.text.DefaultCaret;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.io.*;
import java.nio.charset.StandardCharsets;
```

```
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

public class GUI {
    private JTabbedPane tabbedPane1;
    private JTextArea rxcon;
    private JTextArea txcon;
    private JTextField lat;
    private JTextField lon;
    private JTextField alt;
    private JTextField lastpckt;
    private JTextField velv;
    private JLabel img;
    private JPanel ssdvPanel;
    private JPanel panelMain;
    private JTextArea consoleOutput;
    private JTextField consoleInput;
    private JButton rebootButton;
    private JTextArea controlResults;
    private JButton imageTransmit;
    private JButton noPicsStored;
    private JTextField timeSince;
    private JCheckBox uploadTelemetryCheckBox;
    private JCheckBox uploadImagesCheckBox;
    private JTextField callsign;
    private JButton autoconf;
    private JFormattedTextField key;

    public final CycleManager cm;
    private final File log;

    float lastAlt = 0;
    long lastTime = System.currentTimeMillis();
    long lastID = -1;

    private boolean uploadImage = true;
    private boolean uploadTelem = true;

    public GUI(Config conf) {

        //Prepare log file. This is timestamped so each launch has a new log file.
        log = new File("logs/" + new SimpleDateFormat("yyyy-MM-dd-HH-mm-
ss").format(new Date()) + ".txt");
        try {
            if (!log.getParentFile().exists()) {
                log.getParentFile().mkdirs();
            }
            log.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Init request handler.
        RequestHandler requestHandler = new RequestHandler();

        //Prepare the reboot button so it completes the correct action.
        rebootButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {

cm.addToTx(TwoWayPacketGenerator.generateCommand(conf.getCallsign(), "RBT"));
            }
        });

        //Prepares the consoleInput so that when enter is pressed it sends the
given command.
```

```
        consoleInput.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                String cmd = consoleInput.getText();
                if (cmd.length() > 0 && cmd.length() < 255 - 14 -
conf.getCallsign().length() && !cmd.contains(",") && !cmd.contains("*")) {
//Ensures valid command.

cm.addToTx(TwoWayPacketGenerator.generateShellCmdPacket(conf.getCallsign(), cmd));
                    consoleInput.setBackground(Color.GREEN);
                    consoleInput.setText("");
                } else {
                    consoleInput.setBackground(Color.RED);
                }
            }
        });

        autoconf.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                String cs = callsign.getText();
                String ky = key.getText();
                String data = requestHandler.getPayloadData(cs);
                String[] payload = data.split(",");
                if (payload.length == 8) {
                    String callsgn = payload[0];
                    String txFreq = payload[1];
                    int txBw = Integer.valueOf(payload[2]);
                    String sf = payload[3];
                    String coding = payload[4];
                    boolean explicit = payload[5].equals("1");
                    String rxFreq = payload[6];
                    int rxBw = Integer.valueOf(payload[7]);
                    conf.setCallsign(callsgn);
                    conf.setListen(Double.valueOf(txFreq));
                    conf.setRxbandwidth(Constants.Bandwidth.lookup(txBw));
                    conf.setSf(Short.valueOf(sf));
                    conf.setCodingRate(Constants.CodingRate.valueOf("CR4_" +
coding));
                    conf.setImplicit(!explicit);
                    conf.setFreq(Double.valueOf(rxFreq));
                    conf.setTxbandwidth(Constants.Bandwidth.lookup(rxBw));
                    conf.setKey(ky);
                    conf.save();
                    JOptionPane.showMessageDialog(panelMain, "Success! Configured
for " + callsgn + ". Please restart the program for changes to take effect.");
                } else {
                    JOptionPane.showMessageDialog(panelMain, "Autoconfigure failed,
please edit config.yml and restart the program.");
                }
            }
        });

        //Prepares the image transmit toggle button so when clicked it prepares a
packet to send which toggles image transmission.
        imageTransmit.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {

cm.addToTx(TwoWayPacketGenerator.generateCommand(conf.getCallsign(), "IMG"));
            }
        });

        //Prepares this button so when clicked it prepares a packet which will ask
the payload to transmit the number of stored images.
        noPicsStored.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
```

```
cm.addToTx(TwoWayPacketGenerator.generateCommand(conf.getCallsign(), "IMGNO"));
            }
        });

        //Init cycle manager.
        cm = new CycleManager(false, conf.getCallsign(), new double[]
{conf.getFreq(), conf.getListen()}, new Constants.Bandwidth[]
{conf.getTransmitBandwidth(), conf.getReceiveBandwidth()}, conf.getSf(),
conf.getCodingRate(), !conf.getImplicit(), conf.getPower(), conf.getKey()) {
            @Override
            public void handleTelemetry(ReceivedTelemetry telem) {
                if (telem == null) {
                    return;
                }
                getAlt().setText(String.valueOf(telem.alt));
                getLat().setText(String.valueOf(toDeg(telem.lat)));
                getLon().setText(String.valueOf(toDeg(telem.lon)));
                writeRx(telem.raw);
                if (telem.id != lastID) {
                    getLastpckt().setText(new
SimpleDateFormat("HH:mm:ss").format(new Date()));
                    updateVelocities(telem.alt);
                    lastID = telem.id;
                    if (uploadTelem) {
                        requestHandler.sendTelemetry(telem.raw);
                    }
                }
            }

            @Override
            public void onSend(String sent) {
                if (sent != null) {
                    writeTx(sent);
                }
            }

            @Override
            public void handleImage(byte[] bytes, int iID, int pID) {
                writeRx("Image no. " + iID + " packet no. " + pID + "
received.\n");
                if (uploadImage) {
                    requestHandler.sendImage(new String(bytes,
StandardCharsets.ISO_8859_1));
                }
            }

            @Override
            public void handleTwoWay(ReceivedPacket packet) {
                if (packet == null) {
                    return;
                }
                writeRx(packet.raw);
                switch (packet.type) {
                    case CMD:
                        if (packet.data.equals("TRA")) {
                            this.txInterrupt();
                        }
                        break;
                    case SHELL:
                        getConsoleOutput().append(packet.data.replace((char)0,
'\n'));
                        break;
                    case DIAG:
                        String[] data = packet.data.split("/");
                        getControlResults().append(data[0] + ": " + data[1] +
"\n");
                        break;
```

```
                    case OTHER:
                        //Not really sure what to do here? How about you?
                }
                requestHandler.sendTwoWay(packet.raw);
            }

            @Override
            public String getTelemetry() {
                return null;
            }

            @Override
            public String getImagePacket() {
                return null;
            }
        };

        new Thread(new Runnable() {

            @Override
            public void run() {
                while (true) {
                    long diff = System.currentTimeMillis() - lastTime;
                    diff /= 1000f;
                    timeSince.setText(diff + "s");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).start();

        //Upload checboxes.
        uploadImagesCheckBox.addItemListener(new ItemListener() {
            @Override
            public void itemStateChanged(ItemEvent e) {
                if (e.getStateChange() == ItemEvent.SELECTED) {
                    uploadImage = true;
                } else if (e.getStateChange() == ItemEvent.DESELECTED){
                    uploadImage = false;
                }
            }
        });
        uploadTelemetryCheckBox.addItemListener(new ItemListener() {
            @Override
            public void itemStateChanged(ItemEvent e) {
                if (e.getStateChange() == ItemEvent.SELECTED) {
                    uploadTelem = true;
                } else if (e.getStateChange() == ItemEvent.DESELECTED){
                    uploadTelem = false;
                }
            }
        });
    }

    private static Double toDeg(float nmea) {
        boolean neg = false;
        if (nmea < 0) {
            neg = true;
            nmea = Math.abs(nmea);
        }
        String in = Float.toString(nmea);
        String[] data = in.split("\\.");
        return (int)(1000000*(Double.valueOf(data[0].substring(0,data[0].length()-
2)) +
                Double.valueOf(data[0].substring(data[0].length()-2) + "." +
```

```
                        data[1])/60d))/1000000d * (neg ? -1 : 1);
    }

    //Simple sets all the GUI elements to be formatted correctly.
    public void init() {
        rxcon.setLineWrap(true);
        txcon.setLineWrap(true);
        consoleOutput.setLineWrap(true);
        controlResults.setLineWrap(true);

((DefaultCaret)rxcon.getCaret()).setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE);

((DefaultCaret)txcon.getCaret()).setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE);

((DefaultCaret)consoleOutput.getCaret()).setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE
);

((DefaultCaret)controlResults.getCaret()).setUpdatePolicy(DefaultCaret.ALWAYS_UPDAT
E);
        consoleOutput.setFont(new Font("monospaced", Font.PLAIN, 12));
        consoleInput.setFont(new Font("monospaced", Font.PLAIN, 12));
        controlResults.setFont(new Font("monospaced", Font.PLAIN, 12));
        rxcon.setFont(new Font("monospaced", Font.PLAIN, 12));
        txcon.setFont(new Font("monospaced", Font.PLAIN, 12));
    }

    public JTextArea getControlResults() {
        return controlResults;
    }

    public JLabel getImg() {
        return img;
    }

    public JPanel getPanelMain() {
        return panelMain;
    }

    public JTextField getLat() {
        return lat;
    }

    public JTextField getLon() {
        return lon;
    }

    public JTextField getAlt() {
        return alt;
    }

    public JTextField getLastpckt() {
        return lastpckt;
    }

    public JPanel getSsdvPanel() {

        return ssdvPanel;
    }

    /**
     * Simple method which writes the received given data to the UI and the log
file.
     * @param write string to write.
     */
    public void writeRx(String write) {
        write = write.replace((char)0, '\n');
        rxcon.append("->: " + write);
        try {
```

```
            FileWriter writer = new FileWriter(log, true);
            DateFormat format = new SimpleDateFormat("HH:mm:ss");
            writer.write("RX [" + format.format(new Date()) +"]: " + write);
            writer.flush();
            writer.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Simple method which writes the transmitted given data to the UI and the log
file.
     * @param write string to write.
     */
    public void writeTx(String write) {
        write = write.replace((char)0, '\n');
        txcon.append("<-:" + write);
        try {
            FileWriter writer = new FileWriter(log, true);
            DateFormat format = new SimpleDateFormat("HH:mm:ss");
            writer.write("TX [" + format.format(new Date()) +"]: " + write);
            writer.flush();
            writer.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public JTextArea getConsoleOutput() {
        return consoleOutput;
    }

    /**
     * Updates the vertical velocity field. This is an estimate only.
     * @param alt new altitude.
     */
    private void updateVelocities(float alt) {
        float altSpeed = (alt - lastAlt)/((System.currentTimeMillis() -
lastTime)/1000f);
        altSpeed = (int)(altSpeed * 10) / 10f;
        velv.setText(String.valueOf(altSpeed) + " m/s");
        lastTime = System.currentTimeMillis();
        lastAlt = alt;
    }
}
```

com.sam.hab.ground.main.GroundMain.java

This is the 'main' class of the ground station, it contains the 'main' method which is the start point of the main application thread. In this class the configuration and GUI are initialised and the image decoding and displaying thread is initialised.

```
package com.sam.hab.ground.main;

import com.sam.hab.ground.gui.GUI;
import com.sam.hab.util.lora.Config;
import com.sam.hab.util.lora.Constants.*;

import javax.imageio.ImageIO;
import javax.swing.*;
import java.awt.*;
import java.awt.image.BufferedImage;
```

```
import java.io.File;
import java.io.FileInputStream;

public class GroundMain {

    public static void main(String[] args) throws InterruptedException {

        //Load configuration from config.yml (or create new if none exists).
        final Config conf = new Config();

        //Window setup.

        JFrame frame = new JFrame("Prototype 2-Way HAB Comms");
        GUI gui = new GUI(conf);
        gui.init();
        frame.setPreferredSize(new Dimension(1050, 720));
        frame.setSize(new Dimension(1050,720));
        frame.setContentPane(gui.getPanelMain());
        frame.setDefaultCloseOperation(JFrame.EXIT ON CLOSE);
        frame.pack();
        frame.setVisible(true);

        //This thread continuously updates the image being displayed on the SSDV
tab so that we always have the latest image, it also deletes old images (>30minutes
old).
        Thread imageThread = new Thread(new Runnable() {
            @Override
            public void run() {
                while (true) {
                    try {
                        BufferedImage pic = ImageIO.read(new FileInputStream(new
File("images/current.jpg")));
                        if (pic != null) {
                            Image resized = pic.getScaledInstance(512, 384, 0);
                            gui.getImg().setIcon(new ImageIcon(resized));
                        }
                        Thread.sleep(1000);
                        File imagesFolder = new File("images");
                        File[] files = imagesFolder.listFiles();
                        for (int i = 0; i < files.length; i++) {
                            if (System.currentTimeMillis() -
files[i].lastModified() > 1800000 && files[i].getName().matches("image_(.*).bin"))
{
                                files[i].delete();
                            }
                        }
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        });
        imageThread.start();

        //Ground station starts by transmitting.
        gui.cm.mainLoop(Mode.RX);
    }
}
```

com.sam.hab.ground.web.RequestHandler.java

This is a class which contains the various methods needed to interface with the server, these use HTTP PUT methods to upload received data to my server. As can be seen, data is handled differently for two way packets, telemetry and SSDV. There is also an HTTP GET method which acquires payload data for a given callsign.

```
package com.sam.hab.ground.web;

import java.io.*;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.nio.charset.StandardCharsets;

public class RequestHandler {

    String server = "http://212.250.101.219:8080/";

    /**
     * Sends telemetry to my server for forwarding to habitat tracker.
     * @param telem telemetry to send. Must be either ASCII or ISO 8859-1.
     */
    public void sendTelemetry(String telem) {
        try {
            URL url = new URL(server + "telemetryUpload");
            sendPut(url, telem);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }

    /**
     * Sends image packets to my server for forwarding to ssdv.habhub.org.
     * @param img image packet to send. Must be encoded in ISO 8859-1.
     */
    public void sendImage(String img) {
        try {
            URL url = new URL(server + "imageUpload");
            sendPut(url, img);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }

    /**
     * Simple method which sends a HTTP PUT request to the url supplied with the
data supplied.
     * @param url the url to send to.
     * @param data the data to send.
     */
    public void sendPut(URL url, String data) {
        try {
            HttpURLConnection connection = (HttpURLConnection)url.openConnection();
            connection.setRequestMethod("PUT");
            connection.setDoOutput(true);
            OutputStream out = connection.getOutputStream();
            out.write(data.getBytes(StandardCharsets.ISO_8859_1));
            out.flush();
            out.close();
            connection.getInputStream();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Uploads a 2-way packet to my server for logging in my marvellous database.
     * @param packet packet to log.
     */
    public void sendTwoWay(String packet) {
        try {
            URL url = new URL(server + "packetUpload");
            sendPut(url, packet);
        } catch (MalformedURLException e) {
```

```
            e.printStackTrace();
        }
    }

    public String getPayloadData(String callsign) {
        try {
            URL url = new URL(server + callsign);
            HttpURLConnection connection = (HttpURLConnection)url.openConnection();
            connection.setRequestMethod("GET");
            connection.setDoOutput(true);
            BufferedReader reader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
            String data = reader.readLine();
            reader.close();
            return data;
        } catch (IOException e) {
            e.printStackTrace();
        }
        return "";
    }
}
```

'Payload' Module

This module contains the code for the Payload module, it depends on the Util module. This module is programmed defensively in order to minimise the possibility of terminal errors occurring.

com.sam.hab.payload.main.PayloadMain.java

This class contains the main method for the payload, this is the method that is called to start the main application thread. This method initialises the configuration data and the CycleManager, it also contains the payload's version of the polymorphic CycleManager methods. This method also contains the static method for generating telemetry using GPS data. Finally, the method initialises the GPS read loop.

```
package com.sam.hab.payload.main;

import com.sam.hab.payload.serial.GPSLoop;
import com.sam.hab.util.csum.CRC16CCITT;
import com.sam.hab.util.lora.Config;
import com.sam.hab.util.lora.Constants;
import com.sam.hab.util.txrx.CycleManager;
import com.sam.hab.util.txrx.ReceivedPacket;
import com.sam.hab.util.txrx.ReceivedTelemetry;
import com.sam.hab.util.txrx.TwoWayPacketGenerator;

import java.io.*;
import java.nio.charset.StandardCharsets;
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

public class PayloadMain {

    private static String currentTelemetry;
    private static String callsign;

    private static Config conf;
    private static CycleManager cm;
    private static ImageManager im;

    /**
     * Takes a GPS GGA string and generates a telemetry string using it, then
stores this in the static variable currentTelemetry.
```

```
     * @param gps the GGA string.
     */
    public static void generateTelemetry(String gps) {
        if (gps.startsWith("$GNGGA"))
        {
            String[] data = gps.split(",");
            if (data.length > 9)
            {
                try {
                    String lat = (data[3].equals("S") ? "-" : "") + data[2];
                    String lon = (data[5].equals("W") ? "-" : "") + data[4];
                    String time = data[1].substring(0, 2) + ":" +
data[1].substring(2, 4) + ":" + data[1].substring(4, 6);
                    String telemetry = callsign + "," +
String.valueOf(System.currentTimeMillis() / 1000) + "," + time + "," + lat + "," +
lon + "," + data[9] + "," + data[7];
                    String csum =
CRC16CCITT.calcCsum(telemetry.getBytes(StandardCharsets.ISO_8859_1));
                    telemetry = "$$" + telemetry + "*" + csum + "\n";
                    currentTelemetry = telemetry;
                } catch (StringIndexOutOfBoundsException e) {
                } catch (ArrayIndexOutOfBoundsException e) {
                }
            }
        }
    }

    public static void main(String[] args) {
        conf = new Config();
        callsign = conf.getCallsign();
        Thread gps = new Thread(new GPSLoop());
        gps.start();
        im = new ImageManager(conf.getCallsign());
        while (currentTelemetry == null) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        cm = new CycleManager(true, conf.getCallsign(), new double[]
{conf.getFreq(), conf.getListen()}, new Constants.Bandwidth[]
{conf.getTransmitBandwidth(), conf.getReceiveBandwidth()}, conf.getSf(),
conf.getCodingRate(), !conf.getImplicit(), conf.getPower(), conf.getKey()) {
            @Override
            public void handleTelemetry(ReceivedTelemetry telem) {
                return;
            }

            @Override
            public void onSend(String sent) {
                return;
            }

            @Override
            public void handleImage(byte[] bytes, int iID, int pID) {
                return;
            }

            @Override
            public void handleTwoWay(ReceivedPacket packet) {
                switch (packet.type) {
                    case CMD:
                        handleCommand(packet);
                        break;
                    case SHELL:
                        handleShell(packet);
                        break;
```

```
                    case OTHER:
                        System.err.println("Packet classed 'OTHER' received:");
                        System.err.println(packet.data);
                }
            }

            @Override
            public String getTelemetry() {
                return currentTelemetry;
            }

            @Override
            public String getImagePacket() {
                byte[] pckt = im.getImagePacket();
                if (pckt != null) {
                    return new String(pckt, StandardCharsets.ISO_8859_1);
                }
                return null;
            }
        };

        //Payloads begin by transmitting.
        cm.mainLoop(Constants.Mode.TX);
    }

    /**
     * Takes a received command packet and determines what action is to be taken,
then takes that action.
     * @param packet The packet to analyse.
     */
    public static void handleCommand(ReceivedPacket packet) {
        switch(packet.data) {
            case "RBT":
                Runtime rt = Runtime.getRuntime();
                try {
                    rt.exec("sudo reboot");
                } catch (IOException e) {
                    e.printStackTrace();
                }
                break;
            case "IMGNO":
                File f = new File("images/");
                int imageNo = 0;
                if (f.exists() && f.isDirectory()) {
                    imageNo = f.list().length;
                }

cm.addToTx(TwoWayPacketGenerator.generateStatPacket(conf.getCallsign(), "IMGNO",
String.valueOf(imageNo)));
                break;
            case "IMG":
                boolean sendImages = cm.toggleImage();

cm.addToTx(TwoWayPacketGenerator.generateStatPacket(conf.getCallsign(), "IMG",
String.valueOf(sendImages)));
                break;
        }
    }

    /**
     * Takes a shell packet and executes the given command, storing the result and
preparing it for sending, adds that result to the transmit queue.
     * @param packet the packet to analyse.
     */
    public static void handleShell(ReceivedPacket packet) {
        Runtime rt = Runtime.getRuntime();
        try {
            Process pr = rt.exec(packet.data);
```

```
            if (pr.waitFor(5, TimeUnit.SECONDS)) {
                InputStream stream = pr.getInputStream();
                BufferedReader reader = new BufferedReader(new
InputStreamReader(stream));
                String output = "";
                String line = reader.readLine();
                while (line != null) {
                    output += line + "\n";
                    line = reader.readLine();
                }
                int len = 255 - 14 - conf.getCallsign().length();
                String[] toSend = new String[(int)Math.ceil(output.length() /
(float)len)];
                if (output.length() > len) {
                    for (int i = 0; i < toSend.length-1; i++) {
                        toSend[i] = output.substring(0, (len > output.length() ?
output.length() -1 : len -1));
                        output = output.substring(len);
                    }
                }
                toSend[toSend.length -1] = output;
                String[] packets =
TwoWayPacketGenerator.generateShellPackets(conf.getCallsign(), toSend);
                for (String pckt : Arrays.asList(packets)) {
                    cm.addToTx(pckt);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

com.sam.hab.payload.main.ImageManager.java

This class initialises a separate thread which contains a loop which takes an image every ~1 minute; it then creates a low resolution copy of this image and encodes if for sending using fsphil's SSDV command line application. This makes use of the Java Runtime library to execute bash commands.

```
package com.sam.hab.payload.main;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Arrays;
import java.util.Calendar;
import java.util.LinkedList;
import java.util.Queue;

public class ImageManager {

    private Queue<byte[]> imageQueue = new LinkedList<byte[]>();

    private String latestImage = "";
    private boolean fullDownload = false;

    /**
     * This class creates a thread which loops continuously taking a picture every
30 seconds or so (the inaccuracy is because the time taken to execute the commands
is variable, particularly convert).
     * It also provides the means by which to get SSDV encoded image data in 256
byte packets.
```

```
     * @param callsign The payload callsign, this is used when encoding SSDV
images.
     */
    public ImageManager(String callsign) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                Calendar cal = Calendar.getInstance();
                int count = 0;
                Runtime rt = Runtime.getRuntime();
                try {
                    rt.exec("mkdir images").waitFor();
                    while (true) {
                        String name = String.valueOf(System.currentTimeMillis() /
1000) + ".jpg";
                        rt.exec("raspistill -o images/" + name).waitFor();
                        rt.exec("convert images/" + name + " -resize 768x576!
tmp.jpg").waitFor();
                        rt.exec("./ssdv -e -c " + callsign + " -i " +
String.valueOf(count) + " tmp.jpg out.bin").waitFor();
                        latestImage = name;
                        count++;
                        Thread.sleep(30000);
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }

    /**
     * Returns the next image packet that is ready to be sent by the payload.
     * @return A 256 byte array which equates to one packet.
     */
    public byte[] getImagePacket() {
        if (imageQueue.size() <= 1) {
            try {
                FileInputStream fis = new FileInputStream(new File("out.bin"));
                while (fis.available() > 0) {
                    fis.read();
                    byte[] packet = new byte[255];
                    for (int i = 0; i < 255; i++) {
                        packet[i] = (byte)fis.read();
                    }
                    imageQueue.add(packet);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }

        }
        byte[] packet = imageQueue.poll();
        return packet;
    }

}
```

com.sam.hab.payload.serial.GPSLoop.java

This class has two functions, firstly it sets the GPS module to airborne mode using a simple recursive method and then it runs a continually repeating loop which reads the GPS serial connection for GGA GPS data strings containing location fix data.

```
package com.sam.hab.payload.serial;

import com.pi4j.io.serial.Baud;
import com.pi4j.io.serial.Serial;
import com.pi4j.io.serial.SerialConfig;
import com.pi4j.io.serial.SerialFactory;
import com.pi4j.io.serial.SerialPort;
import com.sam.hab.payload.main.PayloadMain;

import java.io.IOException;
import java.sql.Time;
import java.util.Arrays;

public class GPSLoop implements Runnable {
    SerialConfig config;
    Serial serial;

    //This is the array of bytes that needs to be sent to the GPS to put it into
airborne mode.
    int[] airborneMode = new int[] {0xFF, 0xB5, 0x62, 0x06, 0x24, 0x24, 0x00, 0xFF,
0xFF, 0x06, 0x03, 0x00, 0x00, 0x00, 0x00, 0x10, 0x27, 0x00, 0x00, 0x05, 0x00, 0xFA,
0x00, 0xFA, 0x00, 0x64, 0x00, 0x2C, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x16, 0xDC};

    /**
     * Contains serial port initialisation, the GPS runs at 9600 baud with all
settings as default.
     */
    public GPSLoop() {
        this.config = new SerialConfig();
        this.serial = SerialFactory.createInstance();
        try {
            this.config.device(SerialPort.getDefaultPort()).baud(Baud._9600);
            this.serial.open(this.config);
            if (!setAirborneMode(0)) {
                System.exit(-1);
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    /**
     * My GPS needs to be in airborne mode to bypass the COCOM limits that disable
a GPS if it reaches a high altitude because it is assumed to be an ICBM.
     * This is a recursive algorithm that sets the GPS into airborne mode, it will
continuously try 100 times recursively but if it fails after the 100th try then the
program will exit.
     * This prevents me from starting a flight not in airborne mode.
     */
    private boolean setAirborneMode(int attempts) throws IOException,
InterruptedException {
        for (int i = 0; i < airborneMode.length; i++) {
            this.serial.write((byte)airborneMode[i]);
        }
        Thread.sleep(1000);
        byte[] read = this.serial.read();
        String s = "";
        for (byte b : read) {
            s += (char)(0xFF & b);
        }
        if (s.contains("µb\u0005\u0001\u0002\u0000\u0006$2[")) { //This is the
unicode representation of the sequence of bytes which the GPS will send if it has
successfully been put into airborne mode.
```

```
            return true;
        } else if (attempts > 99) {
            return false;
        } else {
            return setAirborneMode(attempts + 1);
        }
    }

    /**
     * Main loop for the GPS runnable, simply continuously reads from the GPS until
a newline character and then attempts to generate telemetry using that GPS data
(GPS data is terminated by newline).
     */
    public void run() {
        String received = "";
        while (true) {
            try {
                if (this.serial.available() > 0) {
                    char c = (char)(0xFF & this.serial.read(1)[0]);
                    received = received + c;
                    if (c == '\n') {
                        if (received.startsWith("$GNGGA")) {
                            PayloadMain.generateTelemetry(received);
                            this.serial.read();
                        }
                        received = "";
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Web Module

This module contains the server code, this consists of several HTML documents and PHP programs making up the web portal, it also contains the WebServer.py file which contains the interface for the ground station.

index.html

Homepage of the website.

```
<html>

<head>

    <title>Project Icarus Portal</title>

    <link rel="stylesheet" type="text/css" href="style.css">

</head>

<body>

    <div class="nav">

        <center>

            <h1><a href="/">Project Icarus</a></h1>
```

```
            <h5>A 2-way communications system for High Altitude Ballooning. Works
with Habitat.</h5>

            <h5>By Sam Sully (@Sullore or jakeio on #highaltitude).</h5>

        </center>

    </div>

        <div class="tablemain">

            <div class="content">

                <center>

                    <table>

                        <tr>

                            <td><a href="conf.html">Configure Payload</a><br>Setup
your callsign and LoRa parameters.</td>

                        </tr>

                        <tr>

                            <td><a href="export.html">Export Logs</a><br>Export 2-
way packet logs from flights in CSV format (more formats in future).</td>

                        </tr>

                        <tr>

                            <td><a href="logtail.html">View Logtail</a><br>View a
live log of packets being received by this server.</td>

                        </tr>

                    </table>

                </center>

            </div>

    </div>

</body>

</html>
```

conf.html

Payload configuration page. Contains an HTML form to input payload data.

```
<html>

    <head>

        <title>Configure Payload</title>

        <link rel="stylesheet" type="text/css" href="style.css">

    </head>

    <body style="font-family:Arial;">

        <div class="nav">

            <center>

                <h1><a href="/">Project Icarus</a></h1>
```

```
            <h5>A 2-way communications system for High Altitude Ballooning.
Works with Habitat.</h5>

            <h5>By Sam Sully (@Sullore or jakeio on #highaltitude).</h5>

        </center>

    </div>

    <!--Simple HTML website using a form to input all the data to configure a
payload for input to the database.-->

    <div class="content">

        <form action="conf.php" method="post">

            Callsign:<br>

            <input type="text" name="callsign"><br>

            Transmit Frequency (MHz):<br>

            <input type="text" name="txfrequency"><br>

         Receive Frequency (MHz):<br>

            <input type="text" name="rxfrequency"><br>

            Transmit Bandwidth:<br>

            <input type="radio" name="txbandwidth" value="9">500KHz

            <input type="radio" name="txbandwidth" value="8" checked>250KHz

            <input type="radio" name="txbandwidth" value="7">125KHz

            <input type="radio" name="txbandwidth" value="6">62.5KHz

            <input type="radio" name="txbandwidth" value="5">41.7KHz

            <input type="radio" name="txbandwidth" value="4">31.25KHz

            <input type="radio" name="txbandwidth" value="3">20.8KHz

            <input type="radio" name="txbandwidth" value="2">15.6KHz

            <input type="radio" name="txbandwidth" value="1">10.4KHz

            <input type="radio" name="txbandwidth" value="0">7.8KHz

            <br>

         Receive Bandwidth:<br>

            <input type="radio" name="rxbandwidth" value="9">500KHz

            <input type="radio" name="rxbandwidth" value="8" checked>250KHz

            <input type="radio" name="rxbandwidth" value="7">125KHz

            <input type="radio" name="rxbandwidth" value="6">62.5KHz

            <input type="radio" name="rxbandwidth" value="5">41.7KHz

            <input type="radio" name="rxbandwidth" value="4">31.25KHz

            <input type="radio" name="rxbandwidth" value="3">20.8KHz

            <input type="radio" name="rxbandwidth" value="2">15.6KHz

            <input type="radio" name="rxbandwidth" value="1">10.4KHz

            <input type="radio" name="rxbandwidth" value="0">7.8KHz
```

```
                <br>
                Spreading Factor (6-12):<br>
                <input type="text" name="spreading"><br>
                Error Coding Rate:<br>
                <input type="radio" name="coding" value="5" checked>4/5
                <input type="radio" name="coding" value="6">4/6
                <input type="radio" name="coding" value="7">4/7
                <input type="radio" name="coding" value="8">4/8
                <br>
                Explicit Header Mode:<br>
                <input type="radio" name="header" value="1" checked>Yes
                <input type="radio" name="header" value="0">No
                <br>
                <br>
                <input type="submit" value = "Save">
            </form>
        </div>
    </body>
</html>
```

export.html

Page for exporting 2-way packets from the database in CSV format. Uses an HTML form to get the payload callsign and start and stop dates for the export window.

```
<html>
    <head>
        <title>Export 2-Way Data</title>
        <link rel="stylesheet" type="text/css" href="style.css">
    </head>
    <body style="font-family:Arial;">
        <div class="nav">
            <center>
                <h1><a href="/">Project Icarus</a></h1>
                <h5>A 2-way communications system for High Altitude Ballooning.
Works with Habitat.</h5>
                <h5>By Sam Sully (@Sullore or jakeio on #highaltitude).</h5>
            </center>
        </div>
        <div class="content">
```

```
            <p>Use the form below to export 2-way packets transmitted by your
payload.
             <form action="export.php" method="post">
                 Callsign:<br>
                 <input type="text" name="callsign"><br>
                 Window start:<br>
               <input type="date" name="start"><br>
                 Window End:<br>
               <input type="date" name="end"><br>
                 <br>
                 <input type="submit" value = "Save">
             </form>
         </div>
    </body>
</html>
```

logtail.html

Displays the logtail.php web page in an iframe; hence, gives an automatically updating console without refreshing the whole page continually.

```
<html>
    <head>
        <title>View Logtail</title>
        <link rel="stylesheet" type="text/css" href="style.css">
    </head>
    <body style="font-family:Arial;">
        <div class="nav">
            <center>
                <h1><a href="/">Project Icarus</a></h1>
                <h5>A 2-way communications system for High Altitude Ballooning.
Works with Habitat.</h5>
                <h5>By Sam Sully (@Sullore or jakeio on #highaltitude).</h5>
            </center>
        </div>
        <div class="content">
           <iframe src="logtail.php" width=75% height=80%></iframe>
        </div>
    </body>
</html>
```

style.css

Contains the CSS for the website, it roughly matches the styling of the UKHAS website habhub.org.

```
body {
    margin:0;
}
.tablemain td {
    display:block;
    margin: 40px;
    padding:10px;
    padding-bottom:20px;
    padding-top:20px;
    border: 1px solid #eeeeee;
    text-decoration: none;
    text-align: center;
    font-family: Arial;
    font-size: 12px;
    color: #666;
}
.tablemain a {
    color: #00a3d3;
    font-size: 20px;
    text-decoration: none;
}
.tablemain td:hover {
    background-color: #f1f1f1;
}
.content {
    text-align: center;
    width: 95%;
    padding-top: 120px;
    margin: auto;
    background-color: #fcfcfc;
}
.nav {
    font-weight:bold;
    text-align:center;
```

```
    padding-bottom:20px;
}
.nav center {
    position:fixed;
    width:100%;
    color:#f2f2f2;
    background-color:#00a3d3;
    height:125px;
}
.nav h1 {
    font-family: Arial;
    margin-bottom: 0;
}
.nav h5 {
    margin:10px;
    font-family: Arial;
}
.nav a {
    text-decoration:none;
    color:#f2f2f2;
}
```

conf.php

Takes the POST data from the form on config.html and checks the data for validity before entering it into the database.

```
<html>
    <head>
        <title>Configure Payload</title>
        <link rel="stylesheet" type="text/css" href="style.css">
    </head>
    <body style="font-family:Arial;">
        <div class="nav">
            <center>
                <h1><a href="/">Project Icarus</a></h1>
                <h5>A 2-way communications system for High Altitude Ballooning.
Works with Habitat.</h5>
                <h5>By Sam Sully (@Sullore or jakeio on #highaltitude).</h5>
            </center>
```

```
        </div>
        <div class="content">
            <?php
            //Validation, ensure all data valid.
            if (count($_POST) < 8 || (is_null($_POST["callsign"]) ||
$_POST["txfrequency"] <= 0 || $_POST["rxfrequency"] <= 0 || $_POST["spreading"] >
12 || $_POST["spreading"] < 6 || strlen($_POST["callsign"]) > 6)) {
                //End if data invalid.
                die("Please supply all values within valid ranges.");
            }
            //New SQL connection.
            $conn = new mysqli("localhost","root","OlympiaRPG","icarus");
            if ($conn->connect_error) {
                die("Connection to MySQL server failed. Bad things a-happening!");
            }
            //Extract data from HTTP POST.
            $callsign = $_POST["callsign"];
            $txfreq = $_POST["txfrequency"];
           $rxfreq = $_POST["rxfrequency"];
           $txbandwidth = $_POST["txbandwidth"];
            $rxbandwidth = $_POST["rxbandwidth"];
            $sf = $_POST["spreading"];
            $coding = $_POST["coding"];
            $explicit = $_POST["header"];
            //Prepare SQL statement.
            $stmt = $conn->prepare("INSERT INTO payload
(callsign,txfrequency,txbandwidth,spreading_factor,coding,explicit,created_at,rxfre
quency,rxbandwidth) VALUES (?,?,?,?,?,?,NOW(),?,?)");
            $stmt-
>bind_param("sddiiidd",$callsign,$txfreq,$txbandwidth,$sf,$coding,$explicit,$rxfreq
,$rxbandwidth);
            $stmt->execute();
            //Insert and echo success message.
            echo "Successfully added to payload database!";
            $stmt->close();
            $conn->close();
            ?>
        </div>
    </body>
```

```
</html>
```

export.php

Takes the POST data from the export.html page and then exports the requested packets from the database, putting them in CSV format.

```
<?php

//Open new database connection.

$conn = new mysqli("localhost","root","OlympiaRPG","icarus");

if ($conn->connect_error) {

    //End program if database failed to load.

    die("Connection to MySQL server failed. Bad things a-happening!");

}

//Acquire data from HTTP POST.

$start = $_POST["start"];

$end = $_POST["end"];

$callsign = $_POST["callsign"];

//Prepare SQL statement, this is cross-table parametrised SQL using an INNER JOIN.

$stmt = $conn->prepare("SELECT packet.raw FROM packet INNER JOIN payload ON
payload.payload_id = packet.payload_id WHERE payload.callsign = ? AND packet.time <
? AND packet.time > ?");

$stmt->bind_param("sss",$callsign,$end,$start);

$stmt->bind_result($result);

$stmt->execute();

//Output CSV data.

printf("type,data<br>");

while ($stmt->fetch()) {

    $data = explode(",", explode("*", $result)[0]);

    printf($data[2] . "," . $data[3] . "<br>");

}

?>
```

logtail.php

Reads the log.txt file and displays its contents, has a meta tag that causes it to refresh every 2 seconds.

```
<html>

<head>

<!--The below tag makes the page refresh every 2 seconds. This ensures the log is
always up-to-date.-->

<meta http-equiv="refresh" content="2">

</head>
```

```
<body>

<?php

//Open the log file.

$file = fopen("log.txt", "r") or die("No file!");

//Read the log.

$data = fread($file, filesize("log.txt"));

fclose($file);

//Split the log by newline.

$exploded = explode("\n",$data);

//The below for loop outputs each line separately followed by a <br> (line break).

for ($i = count($exploded) -1; $i > count($exploded)-50; $i--) {

    if ($i < 0) {

        break;

    }

    echo $exploded[$i] . "<br>";

}

?>

</body>

</html>
```

WebServer.py

Handles the GET and PUT methods from the ground station. Takes the data for telemetry and SSDV and formats them in JSON before forwarding to the habhub server. Also logs 2-way packets received and provides the GET handler for getting payload configuration data from the database.

```
from http.server import HTTPServer,SimpleHTTPRequestHandler
from time import strftime
import requests as r
import base64,hashlib,MySQLdb,crcmod,datetime

checksum = crcmod.predefined.mkCrcFun('crc-ccitt-false')

db = MySQLdb.connect(host="localhost",user="root",passwd="OlympiaRPG",db="icarus")
cursor = db.cursor()

class TestHandler(SimpleHTTPRequestHandler):

    '''
    This function handles the HTTP GET request which can be sent by the ground
station in order ro request a payload's configuration details.
    It simply looks up in the database any payload with the given callsign and if
one is found it returns the configuration data as CSV.
    If not, it returns nothing.
    '''
    def do_GET(self):
        path = self.path[1:]
        cursor.execute("SELECT * FROM payload WHERE callsign=%s", (path,))
        result = cursor.fetchall()
        maxID = -1
```

```
        maxDate = datetime.datetime(1970, 1, 1)
        for row in result:
            if row[2] > maxDate:
                maxID = row[0]
                maxDate = row[2]
        if maxID == -1:
            return
        cursor.execute("SELECT
callsign,txfrequency,txbandwidth,spreading_factor,coding,explicit,rxfrequency,rxban
dwidth FROM payload WHERE payload_id=%s", (maxID,))
        result = cursor.fetchall()[0]
        self.send_response(200)
        self.send_header('Content-type','text/html')
        self.end_headers()
        out = result[0] + "," + str(result[1]) + "," + str(result[2]) + "," +
str(result[3]) + "," + str(result[4]) + "," + str(result[5]) + "," + str(result[6])
+ "," + str(result[7])
        self.wfile.write(out.encode("iso-8859-1"))


    '''
    This function handles the potential PUT requests to upload telemetry, SSDV or
packet data to the server from the ground station.
    The request type is determined by the URL used, /telemetryUpload, /imageUpload
and /packetUpload are self-explanatory.
    The appropriate function is then called to handle the request.
    '''
    def do_PUT(self):
        path = self.path
        length = int(self.headers['content-length'])
        data = self.rfile.read(length).decode("iso-8859-1")
        if path == "/telemetryUpload":
            handleTelem(data)
        elif path == "/imageUpload":
            handleSSDV(data)
        elif path == "/packetUpload":
            handlePacket(data)
        self.send_response(201)
        self.send_header('Content-type','text/html')
        self.end_headers()
        self.wfile.write("Received.".encode())

'''
This function handles telemetry data, this is forwarded to the habhub server for
logging and displaying on a map.
'''
def handleTelem(data):
    b64 = (base64.b64encode(data.encode()))
    sha256 = hashlib.sha256(b64).hexdigest()
    b64 = b64.decode()
    now = strftime("%Y-%0m-%0dT%H:%M:%SZ")
    json = "{\"data\": {\"_raw\": \"%s\"},\"receivers\": {\"%s\":
{\"time_created\": \"%s\",\"time_uploaded\": \"%s\"}}}" % (b64, "SAMPI", now, now)
    headers = {"Accept" : "application/json", "Content-Type" : "application/json",
"charsets" : "utf-8"}
    try:
        res =
r.put("http://habitat.habhub.org/habitat/_design/payload_telemetry/_update/add_list
ener/"+sha256, headers=headers, data=json)
        with open("log.txt", "a+") as f:
            f.write("[TELEM FWD] " + data)
    except:
        print("Unable to reach habitat.")

'''
This function handles SSDV data, it simply uploads it to the habhub servers for
displaying on ssdv.habhub.org.
'''
```

```
def handleSSDV(data):
    data = "U" + data
    b64 = base64.b64encode(bytearray(data.encode('iso-8859-1'))).decode('utf-8')
    headers = {"Accept" : "application/json", "Content-Type" : "application/json",
"charsets" : "utf-8"}
    now = strftime("%Y-%0m-%0dT%H:%M:%SZ")
    upload = "{\"type\": \"packet\", \"packet\": \"%s\", \"encoding\": \"base64\",
\"received\": \"%s\", \"receiver\": \"%s\"}" % (b64, now, "SAMPI")
    try:
        res = r.post("http://ssdv.habhub.org/api/v0/packets", headers=headers,
data=upload, timeout=2)
        with open("log.txt", "a+") as f:
            f.write("[IMG PCKT FWD]\n")
    except:
        print("Unable to reach habitat.")


'''
This function will input a 2-way packet into the database. The data is escaped
before it is put into the database.
'''
def handlePacket(raw):
    raw = raw.replace("\n", "\\n")
    data = raw.split("*")
    sentence = data[0].replace(">","")
    csum = data[1]
    packetData = sentence.split(",")
    callsign = packetData[0]
    print(packetData)
    cursor.execute("SELECT * FROM payload WHERE callsign=%s", (callsign,))
    result = cursor.fetchall()
    maxID = -1
    maxDate = datetime.datetime(1970, 1, 1)
    for row in result:
        if row[2] > maxDate:
            maxID = row[0]
            maxDate = row[2]
    if maxID == -1:
        return
    cursor.execute("INSERT INTO packet (payload_id,time,raw) VALUES (%s,NOW(),%s)",
(maxID,db.escape_string(raw),))
    db.commit()
    with open("log.txt", "a+") as f:
            f.write("[PCKT LOGGED]"  + raw + "\n")

server = HTTPServer(("",8080), TestHandler)

try:
    server.serve_forever()
except KeyboardInterrupt:
    server.server_close()
```

Testing

LoRa Radio Module Testing

I need to test thoroughly that my LoRa module is inducing the correct functionality of the LoRa radio. The LoRa radio's interface is complex and low level so without testing what the radio actually does when running I cannot be sure that it is running at the correct settings and that my register modification algorithms work. From tests later in this section it will be clear that transmission and receiving work correctly, however, I need to ensure that transmission is occurring on the correct frequency, bandwidth and with the spreading factor, etc.

Test ID	Description	Data/Action	Expected Result	Actual Result	Evidence
1.1	Test frequency of transmission	Set frequency to 869.850MHz and use radio spectrum analysing software to determine if transmission is indeed occurring at this frequency.	Should be transmitting at 869.860MHz.	Is transmitting at 869.850MHz as required.	See figure 28.
1.2	Test of bandwidth of transmission	Set bandwidth to 62.5kHz and use radio spectrum analysing software to determine if transmission is indeed occurring at this bandwidth.	Should be transmitting at 62.5kHz.	Is transmitting at 62.5kHz. The frequency at the left of the peak is roughly 60-65kHz from the right of the peak.	See figure 29.
1.3	Test of spreading factor, explicit header mode and other modem parameters	Attempt to transmit data from my software to a well-tested LoRa gateway software developed by [redacted] in C.	If my software handles modem parameters correctly, then his software should be able to receive data from mine when set to the same modem parameters.	As required, [redacted]s LoRa gateway receives data from my LoRa transmitter correctly when set to the same parameters.	None.

Payload Testing

The payload has no input whatsoever; it simply runs on a continuous loop. The only stimuli that need to be tested are those of the ground station and these will be tested in the '2-Way Communications Testing' section. A summary table is at the end of the section.

Telemetry and SSDV

The first specification point for the payload is that by default it transmits telemetry and SSDV continuously unless there are 2-way packets to transmit in the defined cycle of 20 telemetry and 70 SSDV packets. This has been successfully achieved as can be seen in the screenshot below.

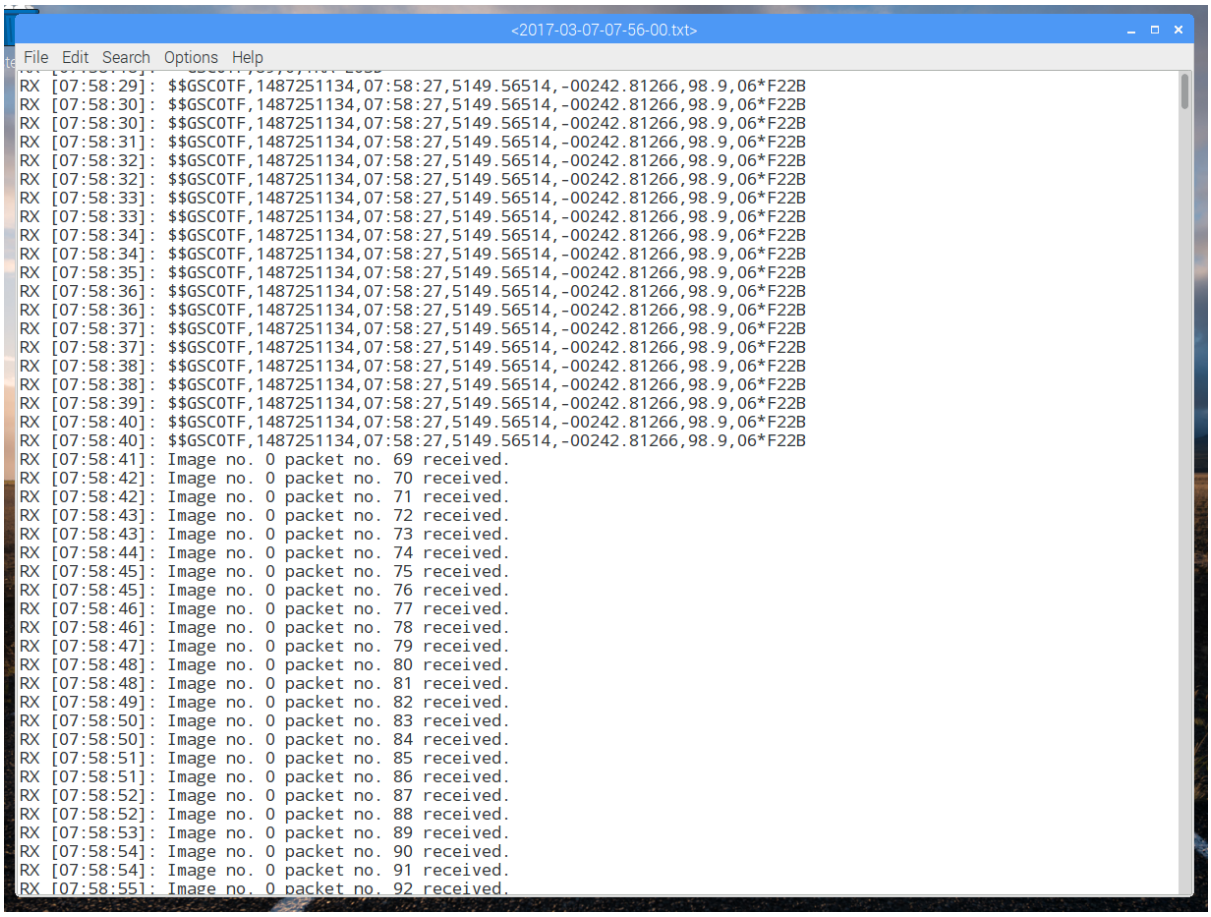


Figure 18 - Log of packets received during part of a 90-packet cycle; the packets are timestamped.

The payload had been left running at this point for 24 hours and as can be seen it is still running correctly as far as a black box approach is concerned no issues are visible. This sees that specification point 3.a is fulfilled.

GPS Data Integrity

My test plan indicates that the integrity of GPS data should be analysed to ensure that location data is up-to-date and that the location is accurate. Below is a telemetry string that was transmitted after the payload had been transmitting for several days:

```
RX [17:12:14]: $$GSCOTF,1487335472,17:12:11,5149.56749,-00242.81445,106.0,12*A367
```

Figure 19 - Telemetry string from after the payload had been transmitting for several days.

Looking at the timestamp of when the packet was received, this is 17:12:14 and the time given in the telemetry string which is extracted directly from the GPS string is 17:12:11. The small delay can be accounted for by error in the clock of the receiving Pi and by the time taken to transmit and parse the packet. While this test may seem pointless, it is important to demonstrate that the GPS data is up-to-date as there is a possibility of a queue building up in the serial buffer which would result in old GPS data being treated as current which would be useless when tracking the payload during a flight.

Another important test for the GPS is that it is correctly put into airborne mode and functions correctly at high altitudes above 9km (which is the default limit), this telemetry packet was transmitted during my flight in February:

```
$$GSCOTF,1486839039,14:03:32,5154.69075,-00324.92817,19293.3,12*B7AA
```

You can see that the altitude parameter (19293.3) is greater than 9km so clearly the GPS was indeed correctly put into airborne mode.

Image Taking

The payload is required to take images regularly. After leaving the payload online for about 30 minutes this is a screenshot of the ls command (list files) in the images directory on the payload.

```
pi@raspberrypi:~/images $ ls
1488997469.jpg  1488997805.jpg  1488998135.jpg  1488998462.jpg  1488998793.jpg
1488997518.jpg  1488997852.jpg  1488998181.jpg  1488998510.jpg  1488998840.jpg
1488997566.jpg  1488997899.jpg  1488998228.jpg  1488998557.jpg  1488998939.jpg
1488997614.jpg  1488997947.jpg  1488998275.jpg  1488998604.jpg
1488997662.jpg  1488997994.jpg  1488998322.jpg  1488998651.jpg
1488997709.jpg  1488998041.jpg  1488998368.jpg  1488998698.jpg
1488997757.jpg  1488998088.jpg  1488998415.jpg  1488998746.jpg
pi@raspberrypi:~/images $
```

Figure 20 - All files stored in images folder after 30 minutes of payload running.

As can be seen there were 31 images taken in this interval, so images are indeed being collected at a rate of 1 per minute.

Summary

A summary of the payload testing is shown in the below table.

Test ID	Description	Data/Action	Expected Result	Actual Result
2.1	Telemetry transmission	Examine payload transmit cycle to determine that telemetry is transmitted.	Payload is transmitted regularly with correct location fix data.	As required. Payload is transmitted correctly; all data is correct and in correct format.
2.2	SSDV	Check payload	Image should be	As required, image

	transmission	is transmitting image packets.	transmitted as part of the standard cycle in the ratio of 70 image packets to 20 telemetry packets.	transmitted in correct ratio.
2.3	Image taking	Check images folder after the payload has been on for 30 minutes.	There should be roughly 30 images in the images folder as the camera should've been taking pictures at a rate of 1 per minute.	There were 31 images in the folder.
2.4	GPS Airborne Mode	The GPS should be tested at a high altitude.	It should continue to transmit correct location data above 9km, indicating that the GPS is in airborne mode.	The GPS worked above 9km and was tested up to ~19km.
2.5	GPS Timing	The GPS timestamp should be analysed.	The timestamp sent on telemetry is the timestamp of when the GPS fix was acquired. We should test whether the GPS data is current to ensure a queue is not building up in the serial buffer.	The data was current, even after the payload was left running for several days.

2-Way Communications Testing

The 2-way communications should be tested to ensure that remote console works and appropriate validation is applied to remote packets, additionally, all remote control commands should work properly from reboot to image number requests.

Test ID	Description	Data/Action	Expected Result	Actual Result	Evidence
3.1	Remote Shell	The user should attempt to use the remote shell system from the ground	Commands should be decoded by the payload and executed as bash	The commands were all executed correctly on the payload and the outputs were	See figure 21 below.

		station. The commands: df -h, ls -la and echo TEST should be tested.	commands, the result should then be sent back.	transmitted in multiple packets if necessary. In the screenshot, you can see the console on the left and the transmission logs on the right.	
3.2	Remote Shell Length	The user should attempt to use the remote shell system again but with a command that will return a result that cannot fit in one packet.	The output should be split between several packets and transmitted separately.	This was achieved correctly, from the screenshot in the evidence we can see in the top right log that the response to ls -la was sent in multiple packets.	Figure 22.
3.3	Remote Command: Reboot	The remote reboot packet should be triggered on the ground station and transmitted.	The payload should reboot.	The payload reboots, after a short delay the payload begins transmitting again. All image IDs reset to 0 so we know the device has restarted.	None provided.
3.4	Remote Command: Image Number	The image number packet should be transmitted.	The payload should respond at the start of the next cycle with the number of pictures in the images folder.	The payload responds with the correct data listing the number of images in the images folder.	Figure 23.
3.5	Remote Command: Toggle Image Sending	The image toggle packet should be transmitted.	The payload should cease transmitting images from the next cycle onwards. When another is transmitted, images should	Image transmission is toggled, for successive cycles no image packets are transmitted until another IMG packet is transmitted	Figure 24 and 25.

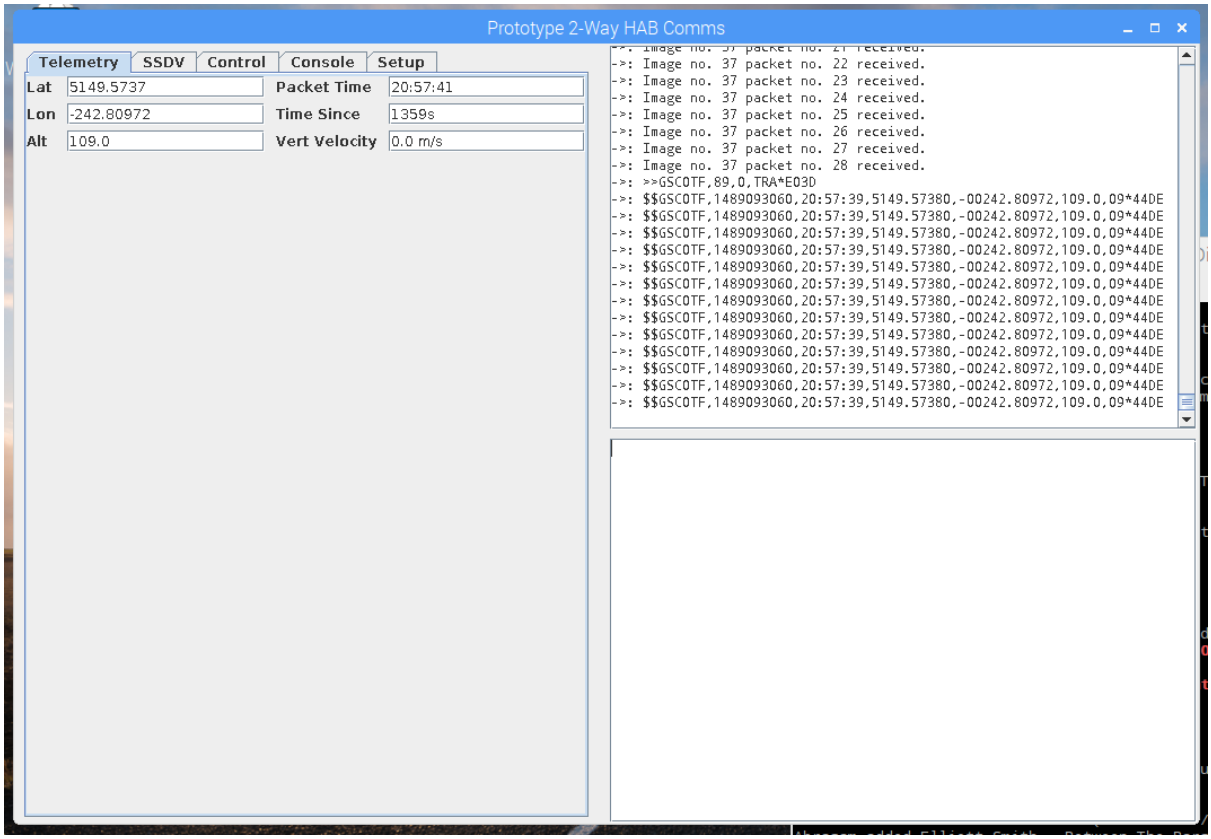
			resume from the next cycle.	toggling image sending again. You can see logs of several cycles where no images were transmitted during the time where image transmission was toggled off.	
3.6	Checksum authentication	Send a communication from the ground station where the key is set to a different value to that of the payload.	The payload should ignore the request.	As required, the payload ignores the packet because the checksum authentication has failed.	None provided.

Ground Station and Server Testing

The ground station needs to be tested to ensure that all received data is logged, forwarded and displayed correctly in that telemetry is displayed on the telemetry tab, SSDV is decoded and displayed on the SSDV tab, control results are displayed in the log on the control tab, console responses are displayed in the console tab and that all packets received or transmitted are logged to file and the two logs on the right of the screen. It also needs to be tested that packets are forwarded to the server for logging or forwarding to habhub and that the logging and forwarding occurs correctly.

Telemetry Display

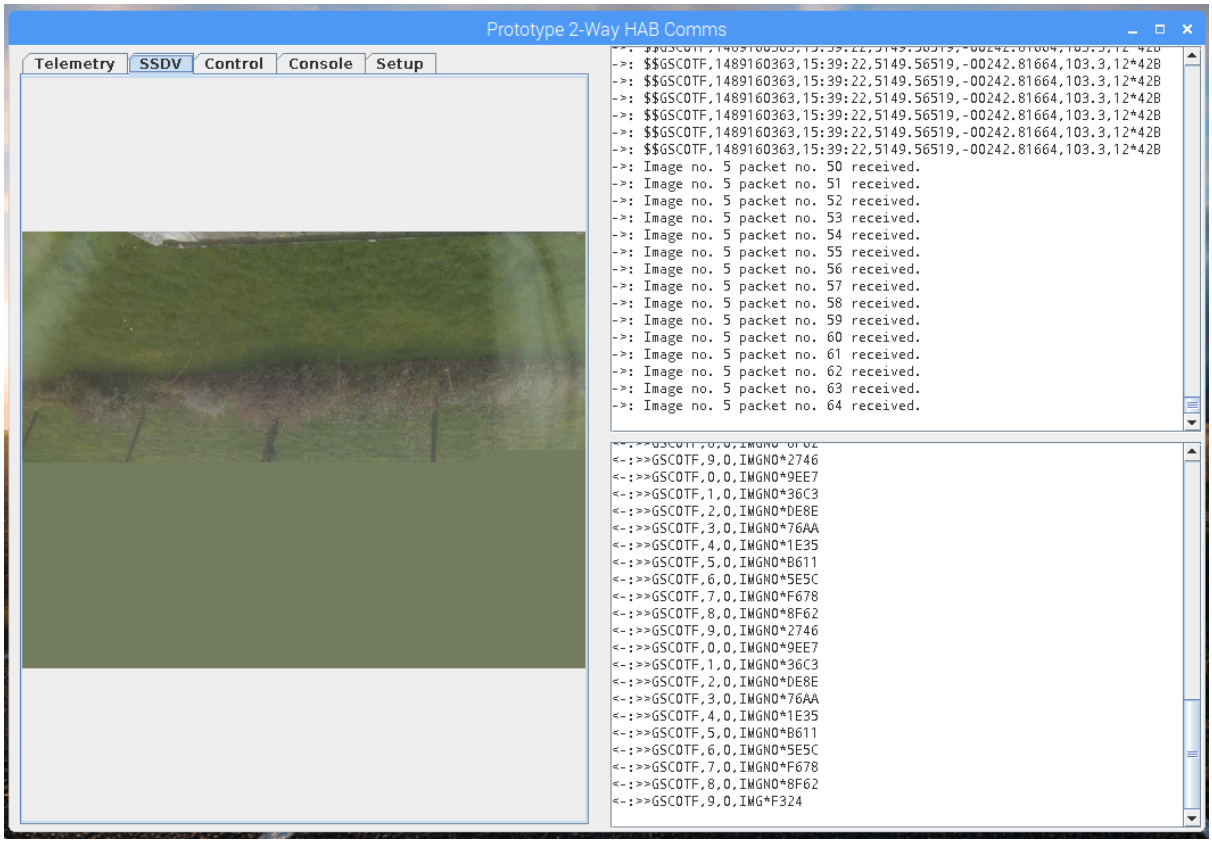
The ground station should display the data from each received telemetry packet along with time since the packet was received. It should display separately, the latitude, longitude, altitude and it should calculate the vertical velocity (estimate) based on time between packets.



As can be seen in the above screenshot all the data is correctly displayed, also, telemetry is displayed in the received log also. Additionally, velocity is calculated, as can be seen from my video demonstration, the velocity works correctly when in flight you can see the ascent velocity was roughly 6m/s. However, there is one issue, the latitude and longitude are given as NMEA data not correctly in degrees so I suggest that I update this to convert the NMEA data of the form ddmm.mmmm (d=degrees, m=minutes) to degrees in the form dd.dddddd as is used by most common mapping software, I already noted the conversion method in the design section.

SSDV Display

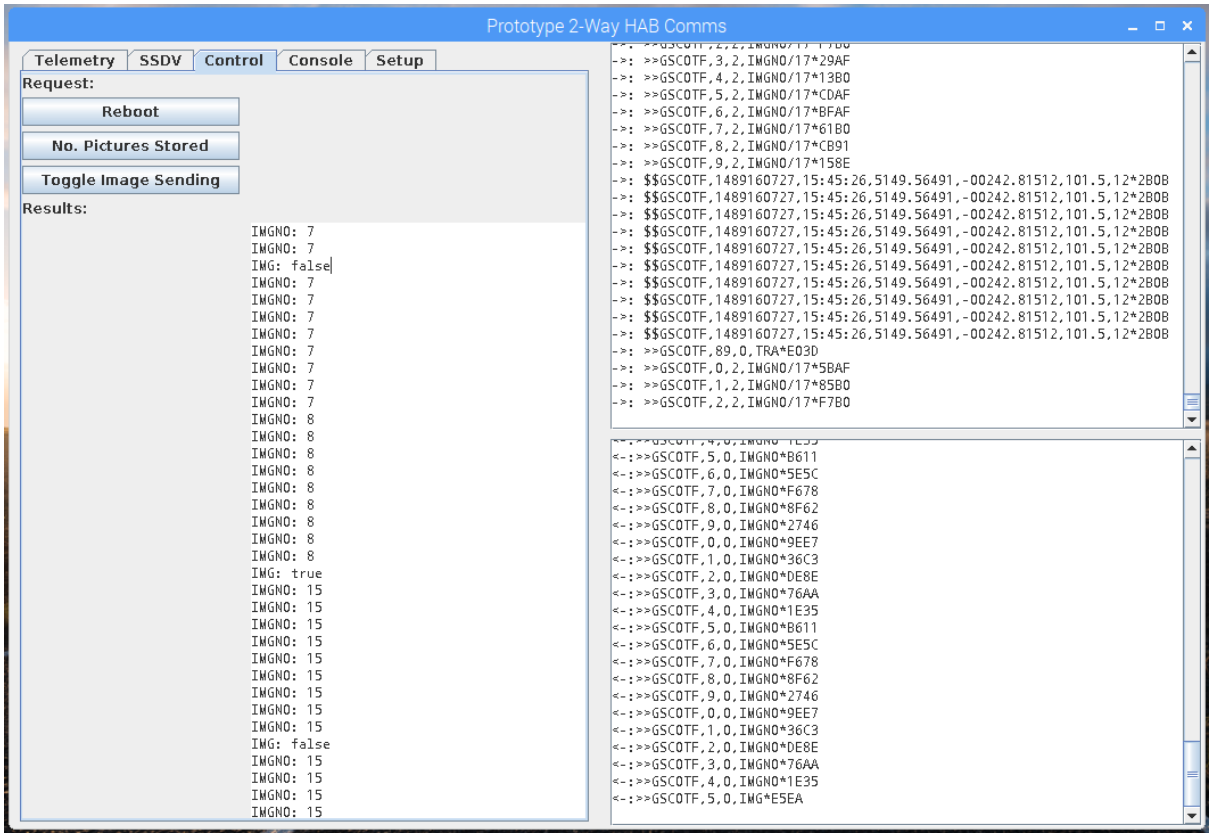
Additionally, the ground station needs to display the latest image that is currently being transmitted by the payload. It should display a partial image while the image has not been fully transmitted. As you can see from the image below, as image packets are transmitted the current version of the image is displayed correctly in the SSDV tab.



As you can see from the screenshot this is a partially transmitted image, so even partial images are displayed correctly as required. From the log on the right you can see that we are part way through receiving image number 5.

Control Results

When a remote control action is responded to by the payload, the result should be displayed in the log on the 'Control' tab, this could be the response to a statistic request or a remote control operation like toggling image sending. Having send many requests for the number of pictures stored and toggling image sending off, on and off again I get the following data displayed.



```
IMGNO: 7
IMGNO: 7
IMG: false
IMGNO: 7
IMGNO: 7
IMGNO: 7
IMGNO: 7
IMGNO: 7
IMGNO: 7
IMGNO: 7
IMGNO: 7
IMGNO: 8
IMGNO: 8
IMGNO: 8
IMGNO: 8
IMGNO: 8
IMGNO: 8
IMGNO: 8
IMGNO: 8
IMGNO: 8
IMG: true
IMGNO: 15
IMGNO: 15
IMGNO: 15
IMGNO: 15
IMGNO: 15
IMGNO: 15
IMGNO: 15
IMGNO: 15
IMGNO: 15
IMG: false
IMGNO: 15
IMGNO: 15
IMGNO: 15
IMGNO: 15
```

```
->: >>GSCOTF,3,2,IMGNO/17*29AF
->: >>GSCOTF,4,2,IMGNO/17*13B0
->: >>GSCOTF,5,2,IMGNO/17*CDAF
->: >>GSCOTF,6,2,IMGNO/17*BFAF
->: >>GSCOTF,7,2,IMGNO/17*61B0
->: >>GSCOTF,8,2,IMGNO/17*CB91
->: >>GSCOTF,9,2,IMGNO/17*158E
->: $$GSCOTF,1489160727,15:45:26,5149.56491,-00242.81512,101.5,12*2B0B
->: $$GSCOTF,1489160727,15:45:26,5149.56491,-00242.81512,101.5,12*2B0B
->: $$GSCOTF,1489160727,15:45:26,5149.56491,-00242.81512,101.5,12*2B0B
->: $$GSCOTF,1489160727,15:45:26,5149.56491,-00242.81512,101.5,12*2B0B
->: $$GSCOTF,1489160727,15:45:26,5149.56491,-00242.81512,101.5,12*2B0B
->: $$GSCOTF,1489160727,15:45:26,5149.56491,-00242.81512,101.5,12*2B0B
->: $$GSCOTF,1489160727,15:45:26,5149.56491,-00242.81512,101.5,12*2B0B
->: $$GSCOTF,1489160727,15:45:26,5149.56491,-00242.81512,101.5,12*2B0B
->: $$GSCOTF,1489160727,15:45:26,5149.56491,-00242.81512,101.5,12*2B0B
->: $$GSCOTF,1489160727,15:45:26,5149.56491,-00242.81512,101.5,12*2B0B
->: >>GSCOTF,89,0,TRA*E03D
->: >>GSCOTF,0,2,IMGNO/17*5BAF
->: >>GSCOTF,1,2,IMGNO/17*85B0
->: >>GSCOTF,2,2,IMGNO/17*F7B0
```

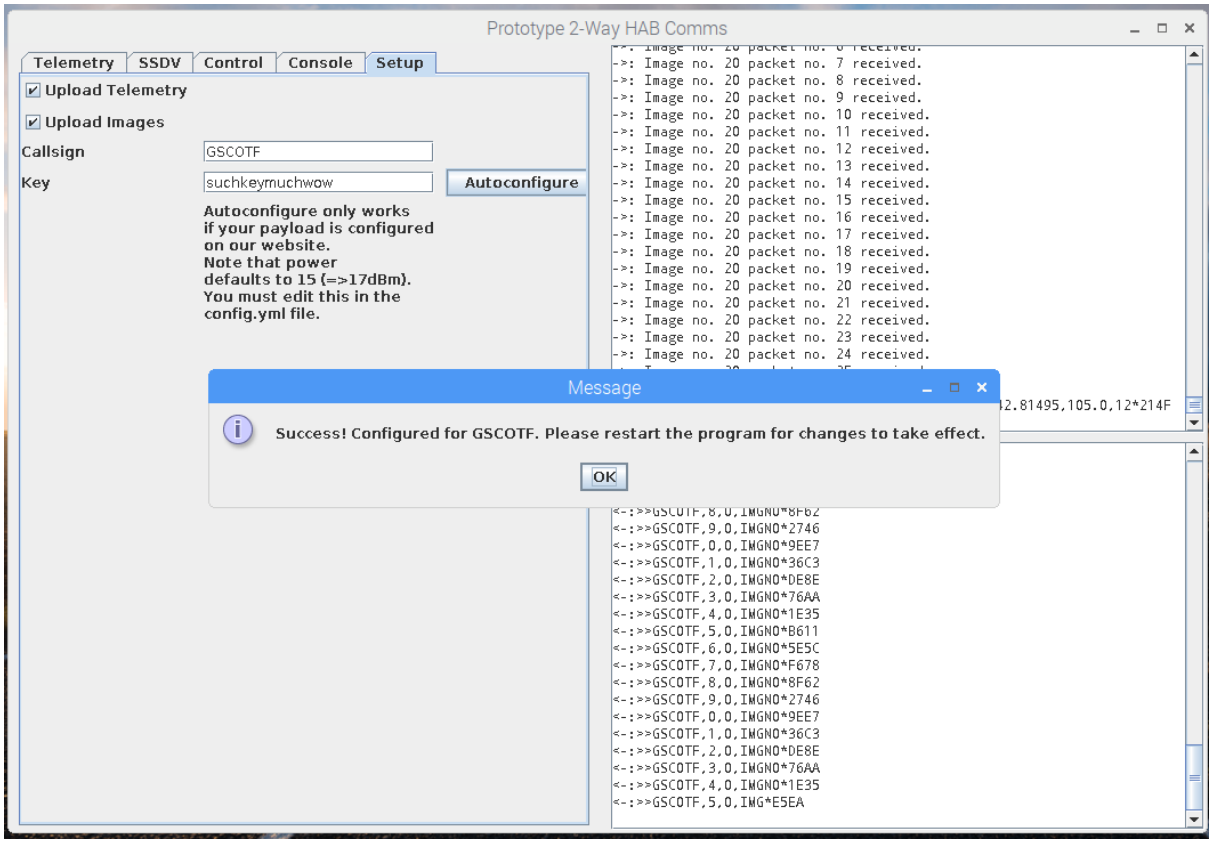
```
<-:>>GSCOTF,5,0,IMGNO*B611
<-:>>GSCOTF,6,0,IMGNO*5E5C
<-:>>GSCOTF,7,0,IMGNO*F678
<-:>>GSCOTF,8,0,IMGNO*8F62
<-:>>GSCOTF,9,0,IMGNO*2746
<-:>>GSCOTF,0,0,IMGNO*9EE7
<-:>>GSCOTF,1,0,IMGNO*36C3
<-:>>GSCOTF,2,0,IMGNO*DE8E
<-:>>GSCOTF,3,0,IMGNO*76AA
<-:>>GSCOTF,4,0,IMGNO*1E35
<-:>>GSCOTF,5,0,IMGNO*B611
<-:>>GSCOTF,6,0,IMGNO*5E5C
<-:>>GSCOTF,7,0,IMGNO*F678
<-:>>GSCOTF,8,0,IMGNO*8F62
<-:>>GSCOTF,9,0,IMGNO*2746
<-:>>GSCOTF,0,0,IMGNO*9EE7
<-:>>GSCOTF,1,0,IMGNO*36C3
<-:>>GSCOTF,2,0,IMGNO*DE8E
<-:>>GSCOTF,3,0,IMGNO*76AA
<-:>>GSCOTF,4,0,IMGNO*1E35
<-:>>GSCOTF,5,0,IMG*E5EA
```

As you can see, the log does indeed correctly display received responses from the payload, however, it does not scroll so once a certain amount of data has been received, the log is then useless.

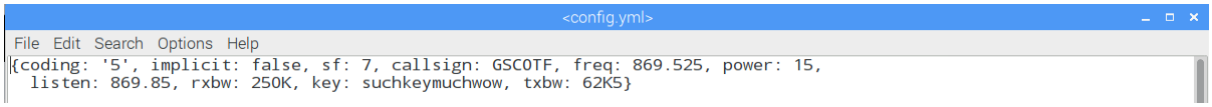
Automatic Configuration System

In the setup tab, the user can input the name of their payload and the encryption key in order to attempt to acquire the payload configuration data from the server and automatically load it into the config.yml file. Two tests must be performed on this, first a payload that exists in the database must be entered and then a payload that does not exist in the database.

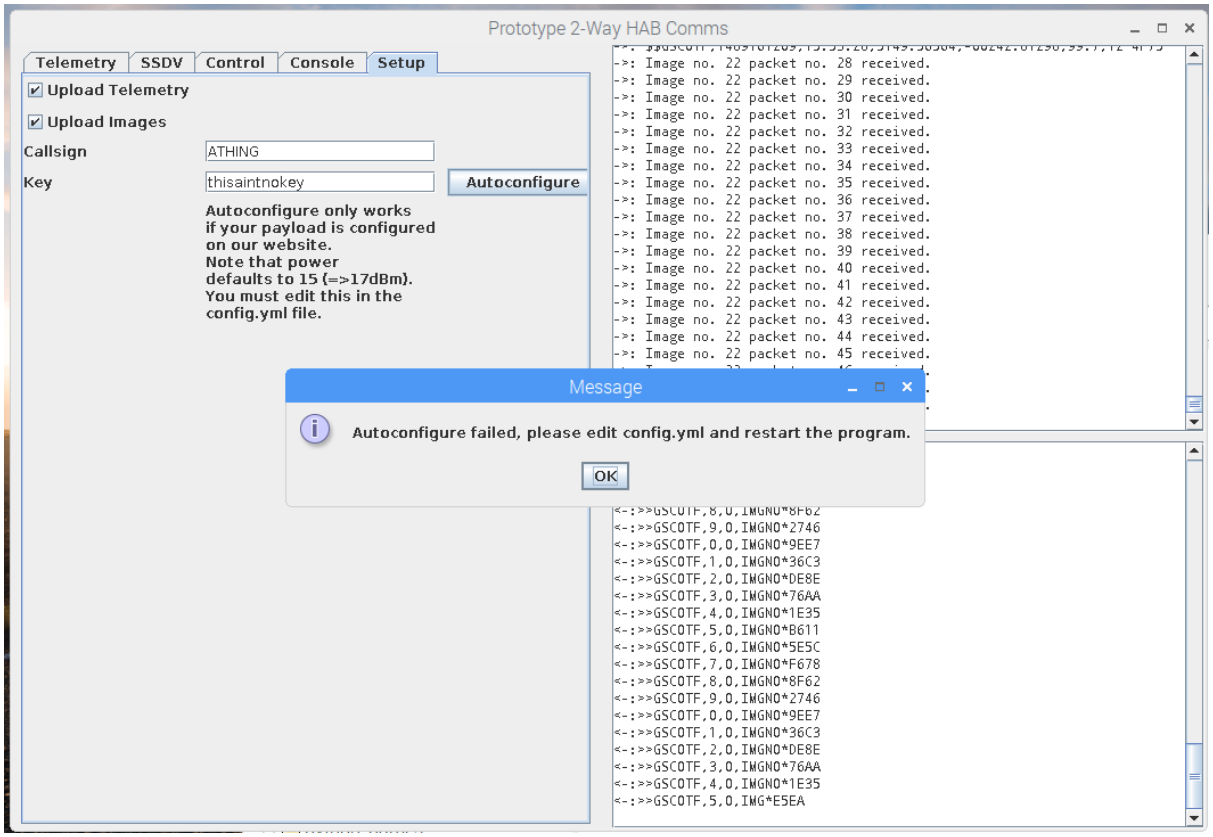
The following screenshot shows the response to entering the details of a payload that is in the database: in this case I used 'GSCOTF', my payload.



This suggests the connection to the server was successful. Additionally, the config.yml file has been updated correctly.

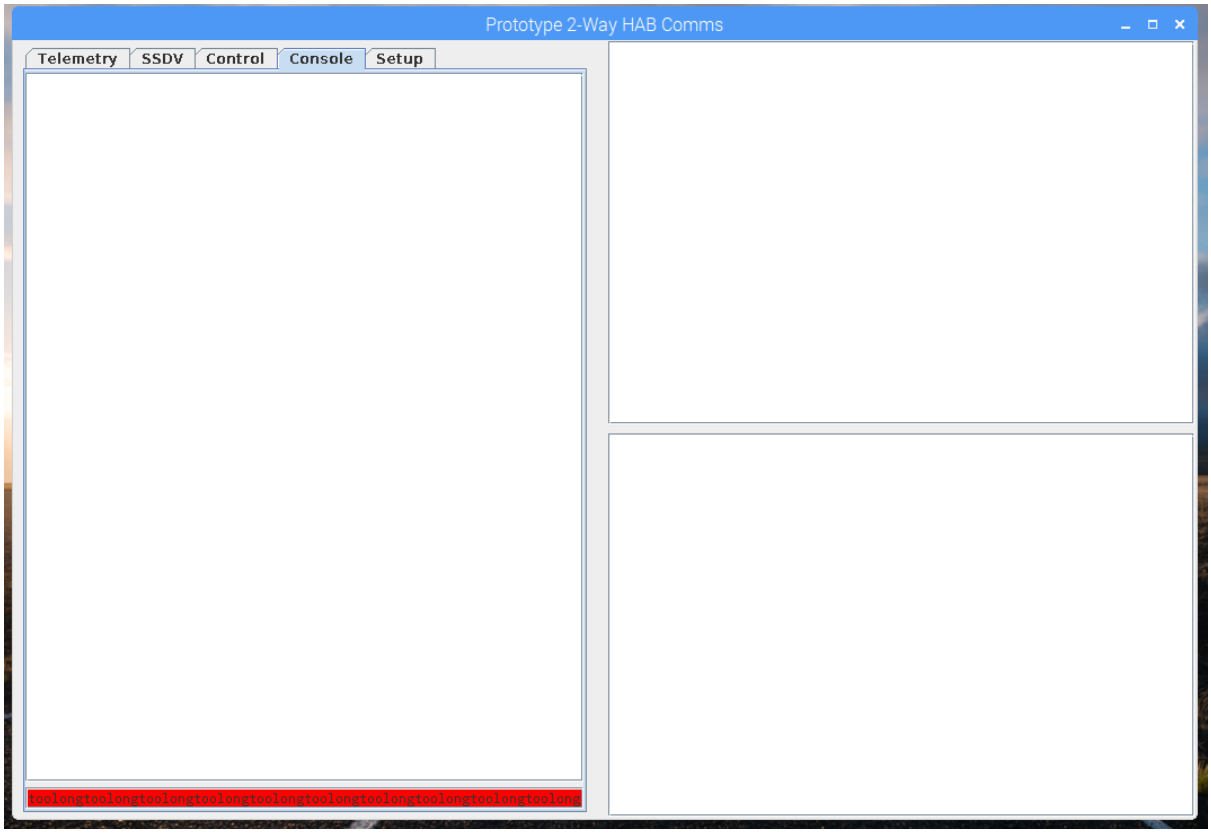


When an invalid payload is entered, the result displayed below is returned informing the user they should go and manually edit the config.yml file to configure their payload. The config.yml file is not modified in this case.

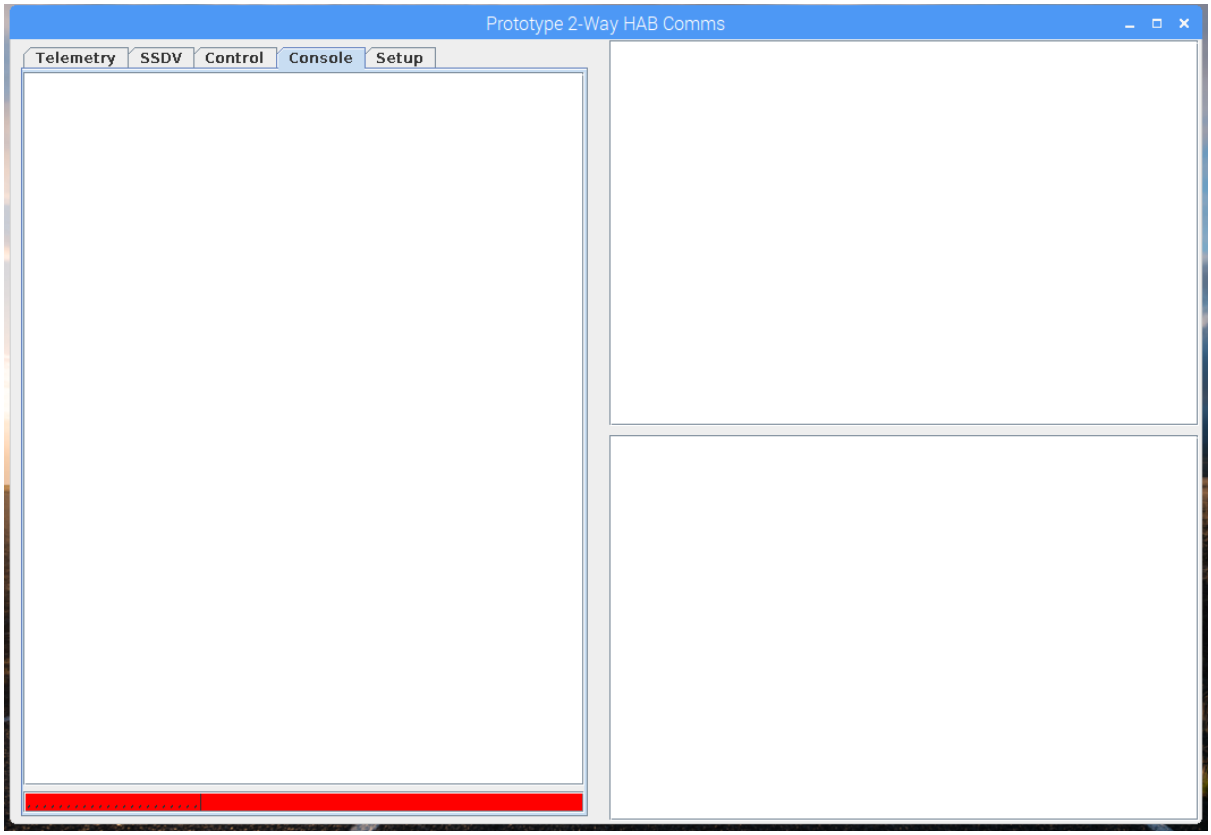


Console Validation

When entering commands to the remote console, the program should reject commands of length greater than the maximum length (see design), commands which contain an asterisk and commands which contain a comma. I shall be testing whether these are correctly rejected.



The above screenshot shows that the entry field goes red and the command is rejected if a command which is above the maximum length is entered.



The above screenshot shows that the entry field goes red and the command is rejected if a command which contains a comma is entered.

However, when a command containing an asterisk is entered it is accepted, this needs to be altered as a command containing an asterisk will not be decodable.

Summary

A summary of all tests carried out on the ground station.

Test ID	Description	Data/Action	Expected Result	Actual Result
3.1	Telemetry display test	Examine the telemetry display tab after telemetry has been received.	Individual telemetry components should be displayed in the correct box, as well as velocity calculated using this and previous telemetry fix.	As required except latitude and longitude are not displayed in degrees, they are as NMEA data of the form ddmm.mmmm, this will be rectified.
3.2	SSDV display test	Examine SSDV display after partial image transmission	Image should be displayed correctly with untransmuted sections as block colours.	As required.
3.3	Control display test	Examine the control tab after several 2-way operations have been enacted.	Results of control operations should be displayed in the console.	As required except the console does not scroll, this will be rectified.
3.4	Autoconfiguration test with valid data	Attempt to use autoconfiguration with a valid payload callsign.	Data should be downloaded from the server and written to the config.yml file, a success message should be displayed	As required.
3.5	Autoconfiguration test with invalid data	Attempt to use autoconfiguration with an invalid callsign.	An error message should be displayed.	As required.
3.6	Remote console length validation	Attempt to send a remote command that is too long.	Should be rejected.	As required.
3.7	Remote console	Attempt to send	Both should be	The command

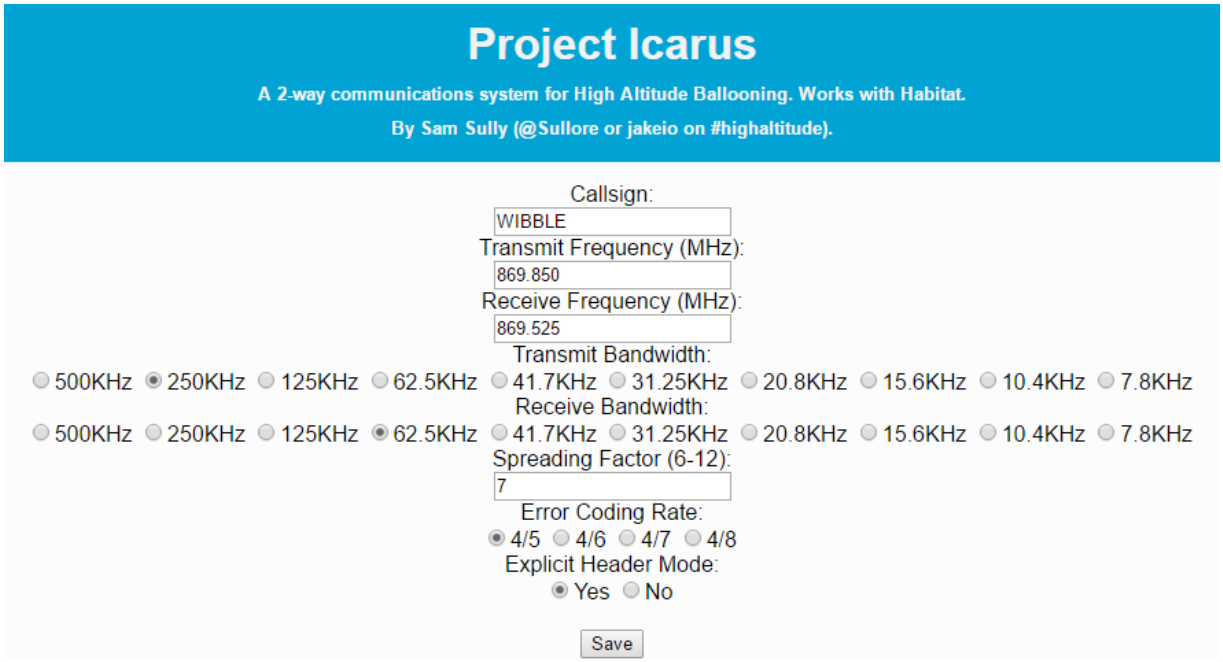
	command and asterisk validation	a command containing a comma and a command containing an asterisk.	rejected.	containing a comma is rejected, that containing an asterisk is not. This is an oversight by me and will be rectified.

Web Testing and Validation

The website has several inputs that need to be tested with erroneous and boundary data.

Configuration Page

The data input here needs to be valid for a payload and of the appropriate datatype for the database. Below is the input page with valid data in every field.

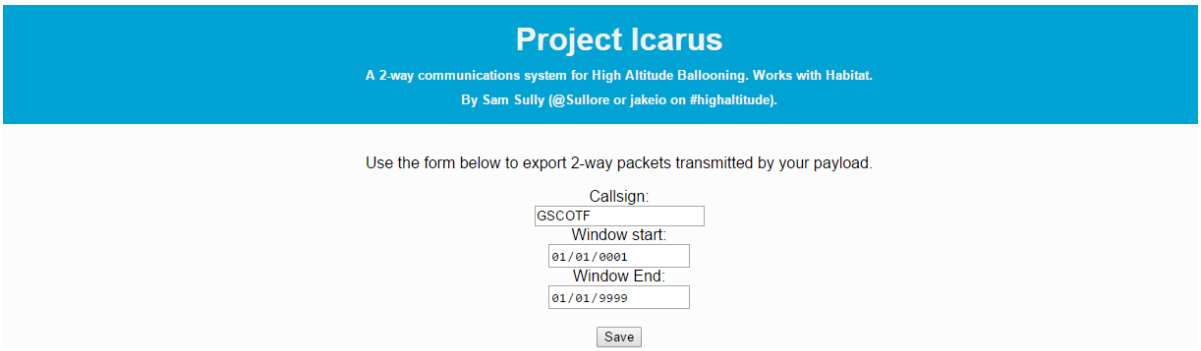


Test ID	Description	Data/Action	Expected Result	Actual Result	Evidence
4.1	Boundary Callsign	Callsign with length greater than 6 characters. Callsign WIBBLE222 was used.	Should be rejected.	Not rejected, accepted and input to database, this will be rectified.	None.
4.2	Erroneous frequency	Frequency that is not a number. The	Should be rejected.	As required. Message displayed saying input invalid.	See figure 26.

		frequency value of "KITTENS" was used.			
4.3	Boundary frequency	Enter a frequency that is less than 0.	Should be rejected.	Not rejected. This will be rectified.	None.
4.4	Boundary spreading factor	Put in spreading factor above 12 or below 6.	Should be rejected.	As required. Message displayed saying input invalid.	See figure 26.
4.5	Erroneous spreading factor	I used the string "KITTENS" instead of a valid integer.	Should be rejected.	As required. Message displayed saying input invalid.	See figure 26.
4.6	Valid data	Valid data in every field should be tested.	Should be accepted and put into the database.	As required.	See screenshot before this table for data used.

Export Page

On the export page of the website you can enter a payload callsign and two dates between which to extract data. Below is a screenshot of the page.

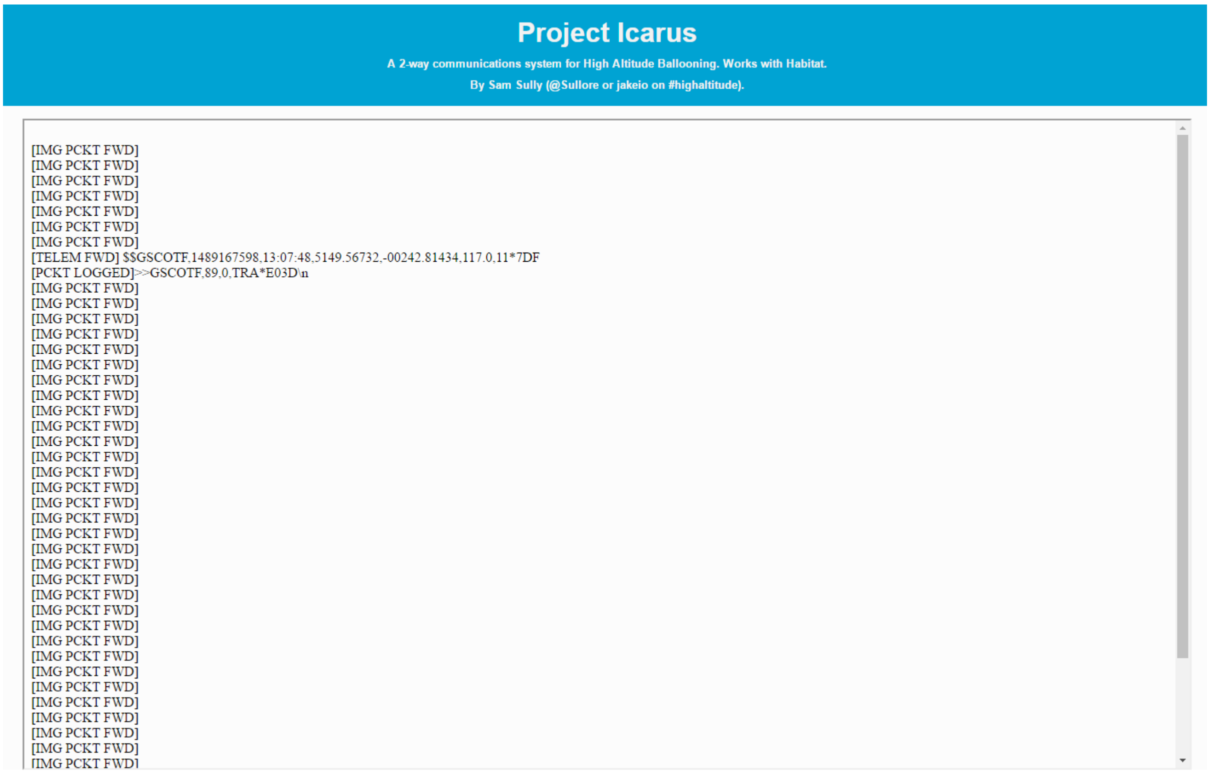


Test ID	Description	Data/Action	Expected Result	Actual Result	Evidence

5.1	Text exporting of 2-way packets	Attempt to export data from any time range. The query in the screenshot above was used in this example.	Data should be returned for that time range in CSV format.	Data returned correctly in CSV format.	Figure 27.
5.2	Test of dates.	Attempt to export data for just one day.	Data should be returned for just that day.	As required. Data returned for just the required day.	None.

Logtail Page

A log of packets received by the server should be displayed on the logtail page. From the below screenshot we can see that this is working correctly. I tested this page by watching the log while the payload was transmitting and all packets were correctly forwarded.



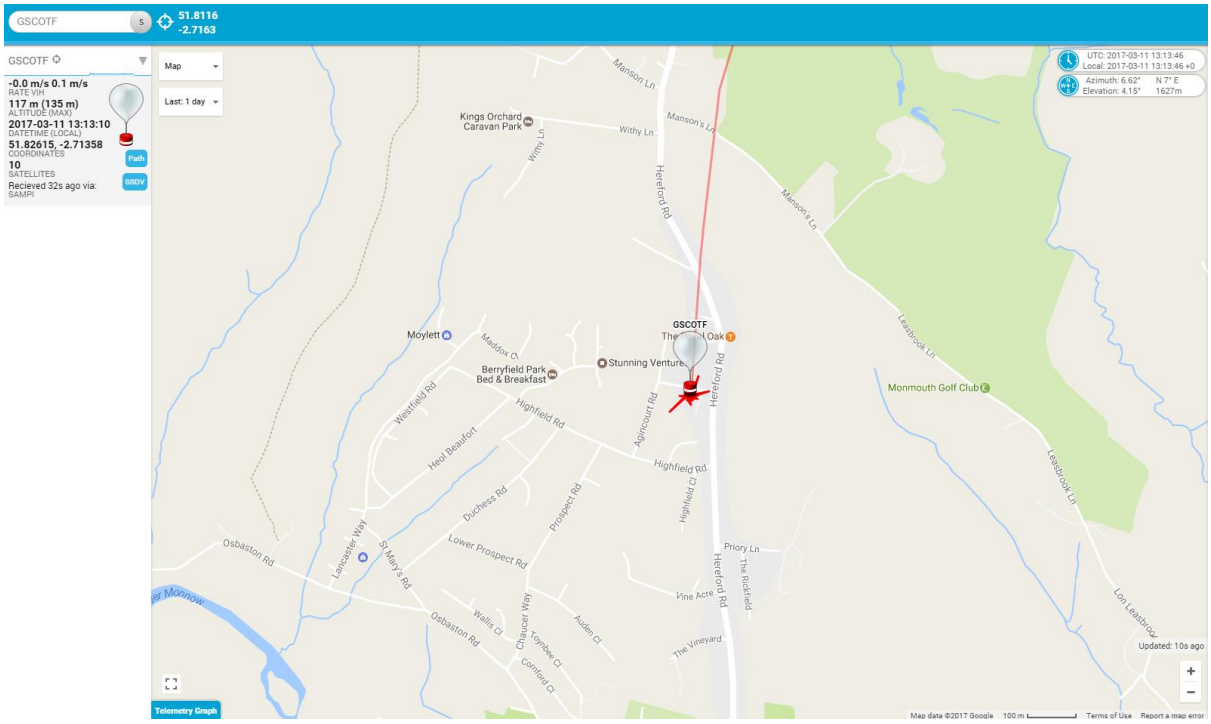
"IMG PCKT FWD" means an image packet was forwarded to habhub. This log could be improved by reducing the latency (perhaps by using AJAX).

Habhub Upload

It needs to be ensured that data is correctly forwarded to habhub.

Below is a screenshot of the habhub SSDV live images feed, you can see images from my payload are being displayed correctly. This shows that image packets are being consistently forwarded by my server to the habhub servers.

Below is a screenshot of the habhub tracker, you can see the GSCOTF payload (my payload) on the map being shown correctly. This demonstrates that telemetry data is being uploaded correctly by my server to the habhub servers.



Flight Test

I flew two test flights for my software both from Monmouth. For the first flight, I had the LoRa radio set to transmit from the payload on 869.850MHz at 250kHz bandwidth and to the payload on 869.525MHz at 250kHz bandwidth also. The launch was successful despite a

near collision with a tree and the flight quickly ascended away, we left the launch site and headed for the predicted landing site which was at the south of the Brecon Beacons. On the road, we had constant contact with the payload and were receiving location data and live images thanks to antennas mounted on the roof and I attempted several 2-way communication transmissions, however, only one was successful throughout the flight, at this point the payload was at around 10km altitude and we were using the rooftop antenna in the car not the high gain Yagi antenna. Nevertheless, we continued the chase and eventually found the payload in Sennybridge, quite far from the predicted landing site, however, easy to locate due to the payload being located on a hill meaning its signal carried further. It was clear from this flight that I would need to reduce the bandwidth for uplinks meaning the radio's power would be spread over a smaller band of frequencies hopefully increasing range. I made this and other adjustments to the payload software over the following day and continued with my second flight the day after that. Below you can see an image of and myself filling the first balloon and below that, the landing site from the first flight.

For the second flight we had an arrangement with from the UKHAS, he was going to go ahead towards the predicted landing site before us, giving me time to stay at the launch site and test my 2-way communications system from the ground using a high gain Yagi antenna (meaning improved range), the intention was to do this for 15 minutes after launch. As before we filled the balloon and launched, however, rather than rushing off to chase cars we got out the Yagi and a friend of mine kindly held it in roughly the direction the payload had gone, I then (using VNC to access my Pi via a tablet computer) operated the remote control functionality of my software, I successfully transmitted 2-way packets to the payload requesting number of pictures stored, toggling image transmission and most importantly I had several successful transmissions of remote console commands which were responded to, the commands I tested included df -h (checks status of disks) and ls -la (lists files in directory), these were transmitted and responded to in full (no packet loss). This was a huge success, demonstrating that my project works, enabling remote control of a payload while it is airborne. Our tests were done at significant range, between 5 and 10km, however, I had to stop and get on the road to chase the payload down but when I stopped testing I still had a strong link with the payload and calculations suggest I could've extended that range to 60km and upwards.

We then began our chase, we managed to receive telemetry and images from the payload almost continuously throughout the flight even though we were slowed down by a lorry who, it appeared, wasn't quite sure where they were going! We arrived at the landing site, had already arrived and had gained permission from the farmer who owned the land to retrieve the payload. I sent a few remote commands to the payload telling it to stop transmitting images and querying how many images had been taken, then we went and retrieved the payload from the field, overleaf there is an image of the landing site. The largest box is my payload GSCOTF, which was operating the 2-way communications software. The other two payloads are , one was a very reliable backup tracker running RTTY and the other was one of new projects using a BBC Microbit.

In conclusion, my project has been shown to work in the situation where it is designed to and can now be used by members of the UKHAS. The source is available on GitHub so hopefully members of the UKHAS will fork it and fly their own versions at some point in the near future. In fact, a group in Budapest have begun using my software themselves.

Testing Evidence

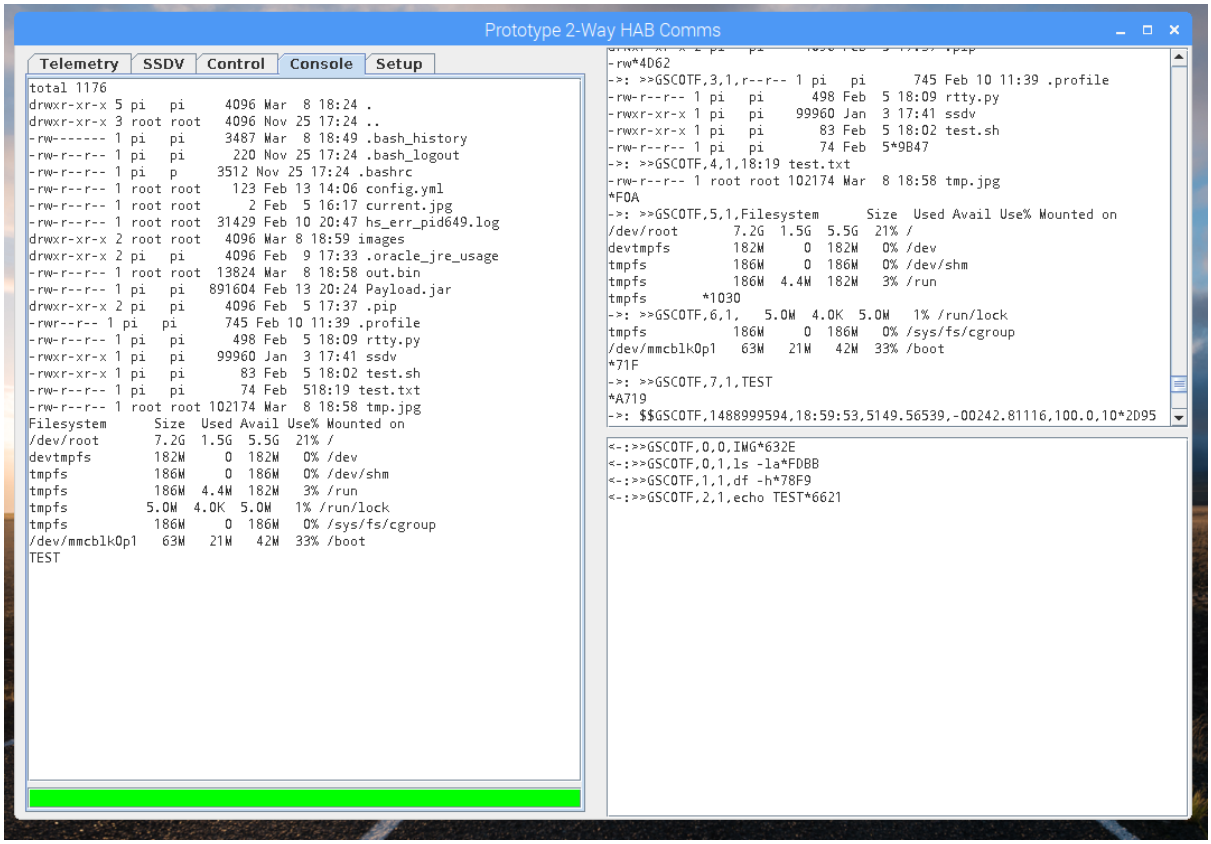


Figure 21- Testing evidence for the remote console.

```
->: >>GSCOTF,1,1,     3512 Nov 25 17:24 .bashrc
-rw-r--r-- 1 root root    123 Feb 13 14:06 config.yml
-rw-r--r-- 1 root root      2 Feb  5 16:17 current.jpg
-rw-r--r-- 1 root root  31429 Feb 10 20:47 hs_err_pid649.log
drwxr-xr-x 2 root root   4096 Mar*3FE4
->: >>GSCOTF,2,1, 8 18:59 images
drwxr-xr-x 2 pi   pi     4096 Feb  9 17:33 .oracle_jre_usage
-rw-r--r-- 1 root root  13824 Mar  8 18:58 out.bin
-rw-r--r-- 1 pi   pi   891604 Feb 13 20:24 Payload.jar
drwxr-xr-x 2 pi   pi     4096 Feb  5 17:37 .pip
-rw*4D62
->: >>GSCOTF,3,1,r--r-- 1 pi   pi      745 Feb 10 11:39 .profile
-rw-r--r-- 1 pi   pi      498 Feb  5 18:09 rtty.py
-rwxr-xr-x 1 pi   pi    99960 Jan  3 17:41 ssdv
-rwxr-xr-x 1 pi   pi       83 Feb  5 18:02 test.sh
-rw-r--r-- 1 pi   pi       74 Feb  5*9B47
->: >>GSCOTF,4,1,18:19 test.txt
```

Figure 22 - Log of received packets showing output of the ls -la command being split between several packets. A new packet can be seen to begin with each ">>".

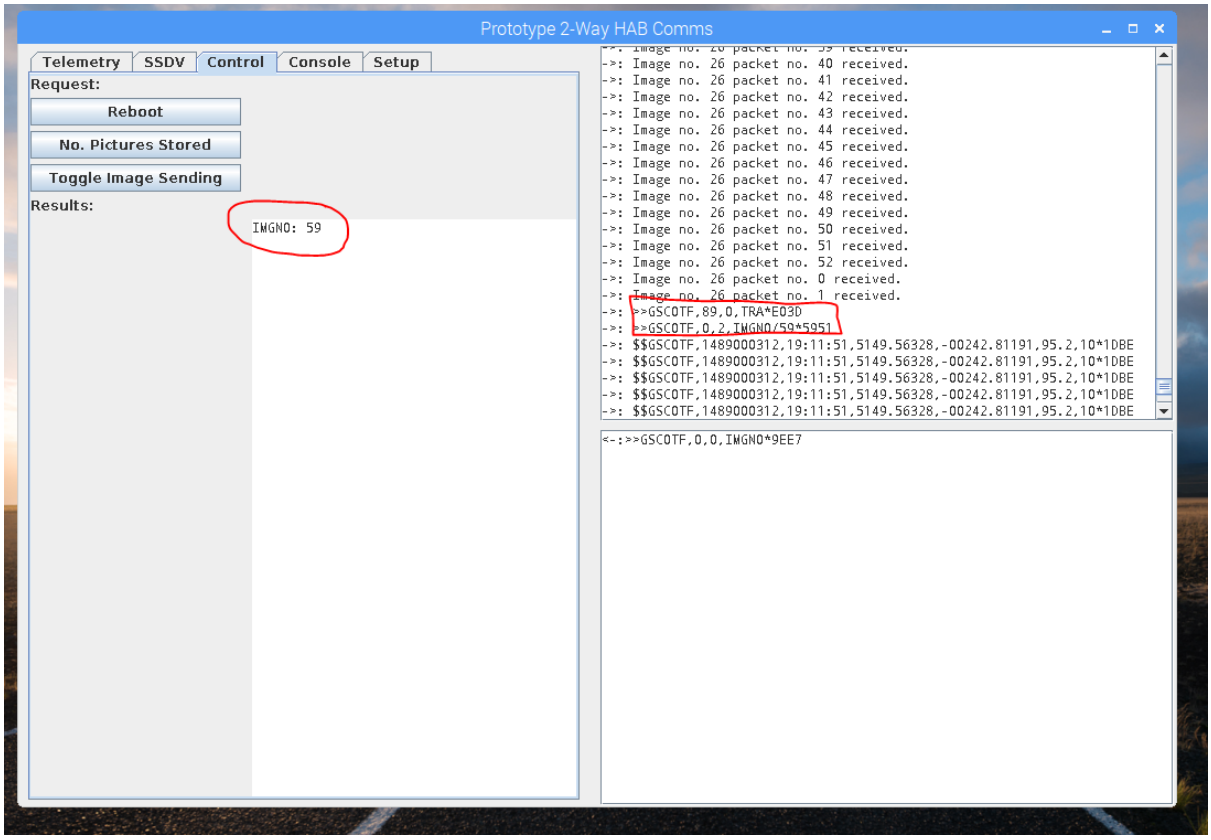


Figure 23 - Screenshot of the ground station software control tab after a the IMGNO packet was transmitted and responded to, you can see the result in the result column on the left and in the receive log on the right you can see the statistic packet was logged.

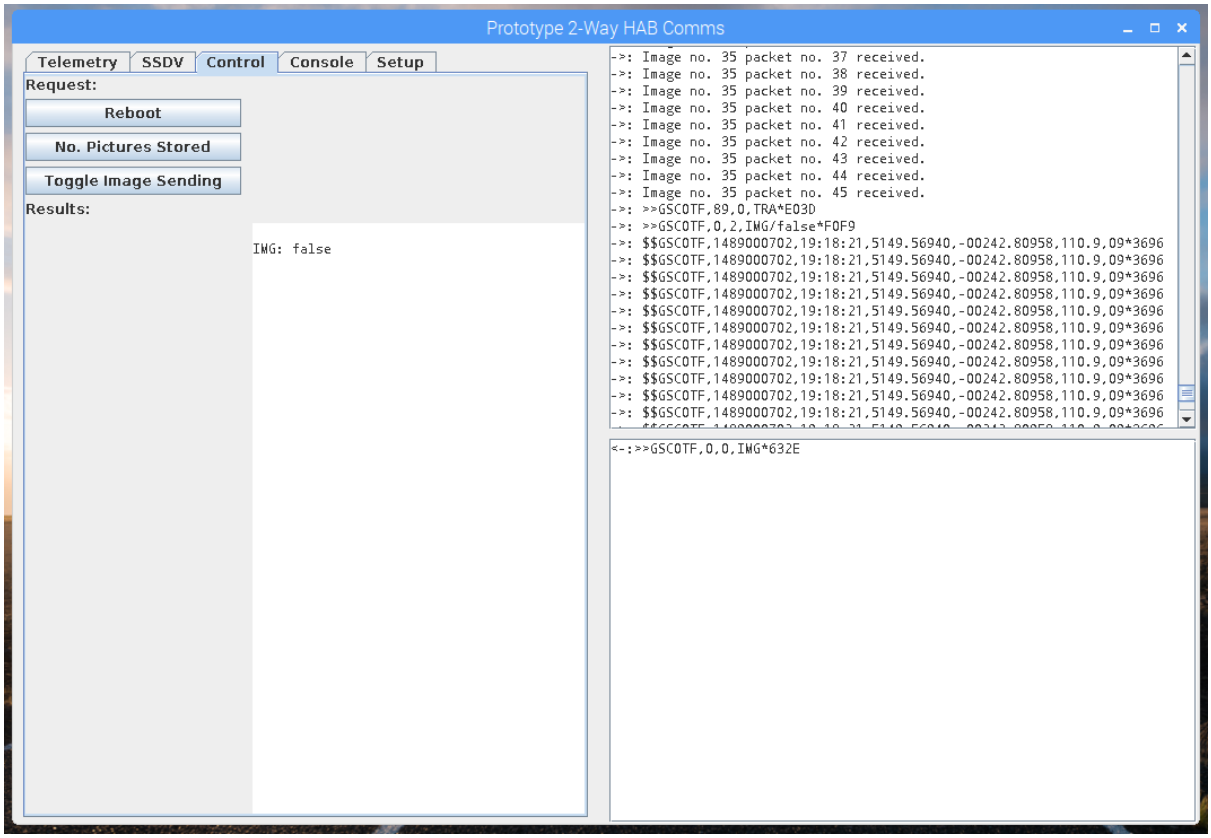


Figure 24 - Shows the transmission of the IMG packet to toggle image sending and the response 'false' shows that image transmission is now set to false.

```
RX [19:27:45]: $$GSC0TF,1489001254,19:27:33,5149.55825,-00242.81019,89.4,07*CD64
RX [19:27:46]: $$GSC0TF,1489001254,19:27:33,5149.55825,-00242.81019,89.4,07*CD64
RX [19:27:46]: $$GSC0TF,1489001254,19:27:33,5149.55825,-00242.81019,89.4,07*CD64
RX [19:27:47]: $$GSC0TF,1489001254,19:27:33,5149.55825,-00242.81019,89.4,07*CD64
RX [19:27:47]: >>GSC0TF,89,0,TRA*E03D
RX [19:27:58]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:27:59]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:27:59]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:00]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:00]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:01]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:02]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:02]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:03]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:03]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:04]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:04]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:05]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:06]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:06]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:07]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:07]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:08]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:08]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:09]: $$GSC0TF,1489001276,19:27:55,5149.55918,-00242.80753,88.8,07*8F32
RX [19:28:10]: >>GSC0TF,89,0,TRA*E03D
RX [19:28:20]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:21]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:21]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:22]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:22]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:23]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:24]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:24]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:25]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:25]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:26]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:26]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:27]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:28]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:28]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:29]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:29]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:30]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:30]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:31]: $$GSC0TF,1489001298,19:28:17,5149.56048,-00242.80761,91.1,07*7FF7
RX [19:28:32]: >>GSC0TF,89,0,TRA*E03D
```

Figure 25 - Logs showing multiple cycles where no images are being transmitted because image transmission is set to false.

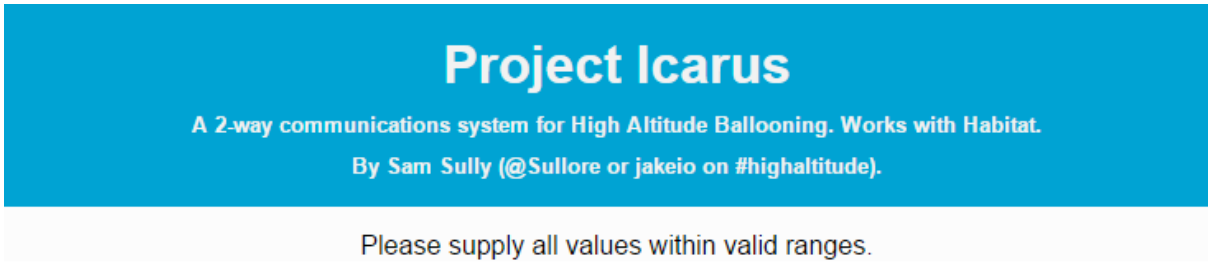


Figure 26 - Demonstrates the error message displayed when invalid data input to configuration page.

type,data
0,TRA
0,TRA
0,TRA
0,TRA
2,IMGNO/16
0,TRA
2,IMG/false
0,TRA
0,TRA
0,TRA
1,total 1304drwxr-xr-x 5 pi pi 4096 Feb 16 02:17 .drwxr-xr-x 3 root root 4096 Nov 25 17:24 ..-rw------- 1 pi pi 3382 Feb 13 23:10 .bash_history-rw-r--r-- 1 pi pi 220 Nov 25 17:24 .bash_logout-rw-r--r-- 1 pi p
1, 3512 Nov 25 17:24 .bashrc-rw-r--r-- 1 root root 123 Feb 13 14:06 config.yml-rw-r--r-- 1 root root 2 Feb 5 16:17 current.jpg-rw-r--r-- 1 root root 31429 Feb 10 20:47 hs_err_pid649.logdrwxr-xr-x 2 root root 4096 Feb
1,16 02:29 imagesdrwxr-xr-x 2 pi pi 4096 Feb 9 17:33 .oracle_jre_usage-rw-r--r-- 1 root root 20224 Feb 16 02:29 out.bin-rw-r--r-- 1 pi pi 891604 Feb 13 20:24 Payload.jardrwxr-xr-x 2 pi pi 4096 Feb 5 17:37 .pip-rw
1,r--r-- 1 pi pi 745 Feb 10 11:39 .profile-rw-r--r-- 1 pi pi 498 Feb 5 18:09 rtty.py-rwxr-xr-x 1 pi pi 99960 Jan 3 17:41 ssdv-rwxr-xr-x 1 pi pi 83 Feb 5 18:02 test.sh-rw-r--r-- 1 pi pi 74 Feb 5
1,18:19 test.txt-rw-r--r-- 1 root root 228739 Feb 16 02:29 tmp.jpg
0,TRA
0,TRA
0,TRA
0,TRA
0,TRA
0,TRA
0,TRA
2,IMG/false
0,TRA
0,TRA
0,TRA
0,TRA
0,TRA
0,TRA
0,TRA
0,TRA
0,TRA
0,TRA
2,IMG/true
0,TRA
2,IMG/false
0,TRA
0,TRA
0,TRA
0,TRA
0,TRA

Figure 27 - Data outputted by the export 2-way packets page. You can see the top row has the column names so this data can be exported into a CSV file and read by software like MS Excel.

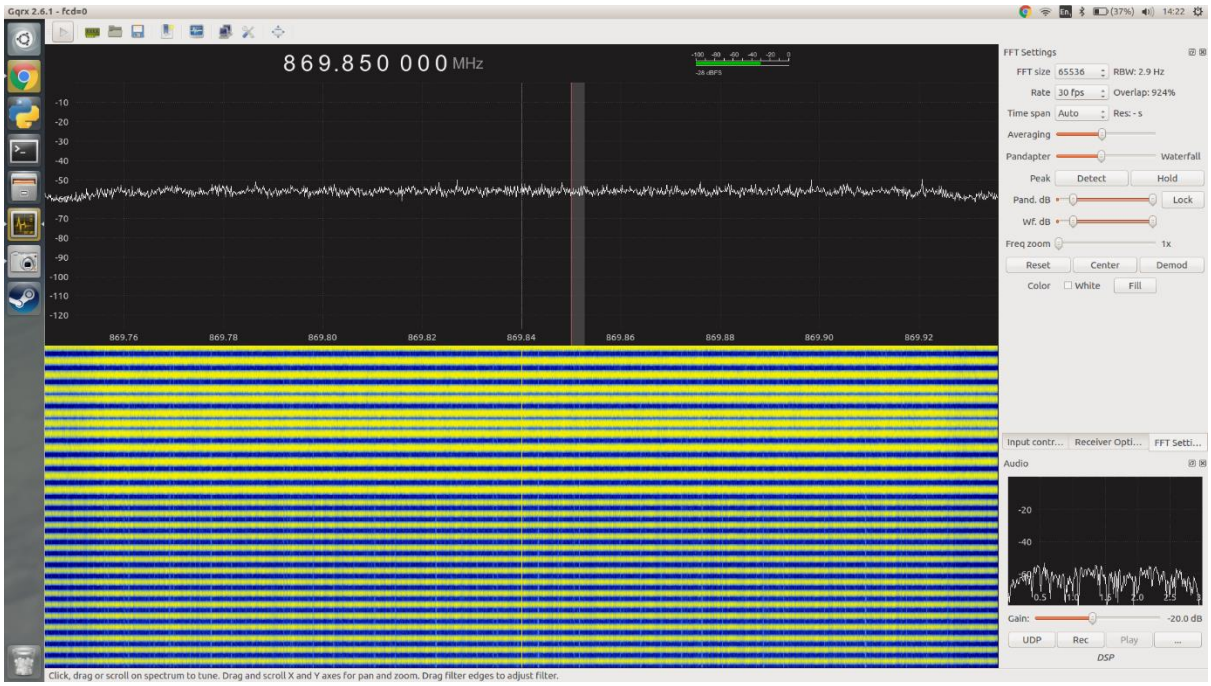


Figure 28 - Showing Gqrx, an SDR (Software Defined Radio) software, analysing what is being transmitted at 869.850MHz during testing. The yellow bars represent peaks in intensity (a transmission) and the blue represents background noise (no transmission).

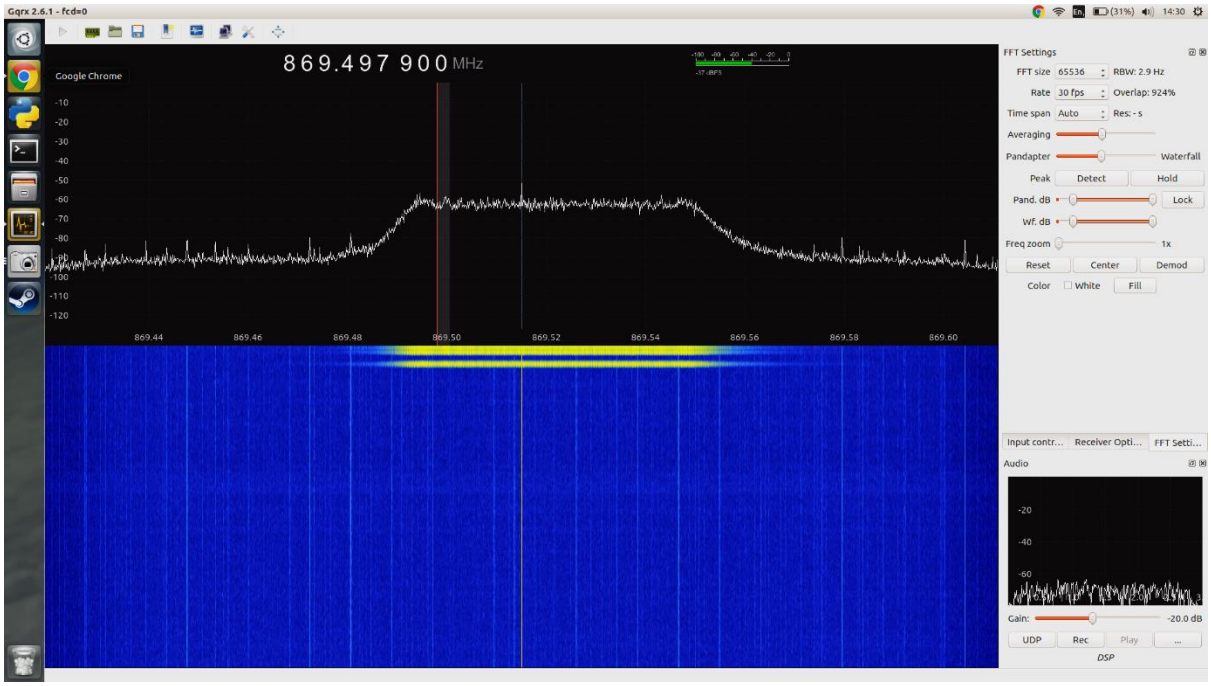


Figure 29 - Showing Gqrx, an SDR (Software Defined Radio) software, analysing what is being transmitted at 869.525MHz during testing. The difference between the frequency of the left of the peak and the frequency of the right of the peak is roughly 60-65kHz (depending on where you determine the edges to be).

Changes

I have noted throughout the testing section anything that should be changed. I have produced a table below demonstrating my changes.

Relevant Test	Change Made	New Test Result	Evidence
3.1 – Testing telemetry display	Instead of just displaying the raw data from the	Data displayed is in latitude and longitude degrees rather than NMEA data.	See figure 30.
3.3 – Testing remote control results display	Added a JScrollPane around the console.	Scrolls when full now.	See figure 31.
3.7 – Testing remote console command validation	Added a check to ensure asterisks aren't in remote command.	Asterisks are rejected.	See figure 32.
4.1 – Testing web configuration page	Added a length check to ensure callsign is length ≤ 6.	Rejects invalid callsigns as required.	None.
4.3 – Testing web configuration page	Added a check to ensure frequency cannot be negative.	Rejects invalid frequencies.	None.

Code Changes

The method below was created to convert NMEA data into degrees as required by the first test.

```
private static Double toDeg(float nmea) {
    boolean neg = false;
    if (nmea < 0) {
        neg = true;
        nmea = Math.abs(nmea);
    }
    String in = Float.toString(nmea);
    String[] data = in.split( regex: "\\.");
    return (int)(1000000*(Double.valueOf(data[0].substring(0,data[0].length()-2)) +
            Double.valueOf(data[0].substring(data[0].length()-2) + "." +
                    data[1])/60d))/1000000d * (neg ? -1 : 1);
}
```

The line below was modified to ensure that commands containing asterisks are rejected in that the final AND clause was added to the if statement.

```
if (cmd.length() > 0 && cmd.length() < 255 - 14 - conf.getCallsign().length() && !cmd.contains(",") && !cmd.contains("*")) {
```

To ensure that frequencies cannot be set to negative numbers and that callsigns must be of length at most 6, this block of code was modified.

```
if (count($_POST) < 8 || (is_null($_POST["callsign"]) || $_POST[
"txfrequency"] <= 0 || $_POST["rxfrequency"] <= 0 || $_POST[
"spreading"] > 12 || $_POST["spreading"] < 6 || strlen($POST[
"callsign"]) > 6)) {
```

Originally, the frequency checks only checked if the frequency was equal to zero, I changed this to equal to or less than. Additionally, I added the final clause checking if the length of callsign is larger than 6.

Correction Evidence

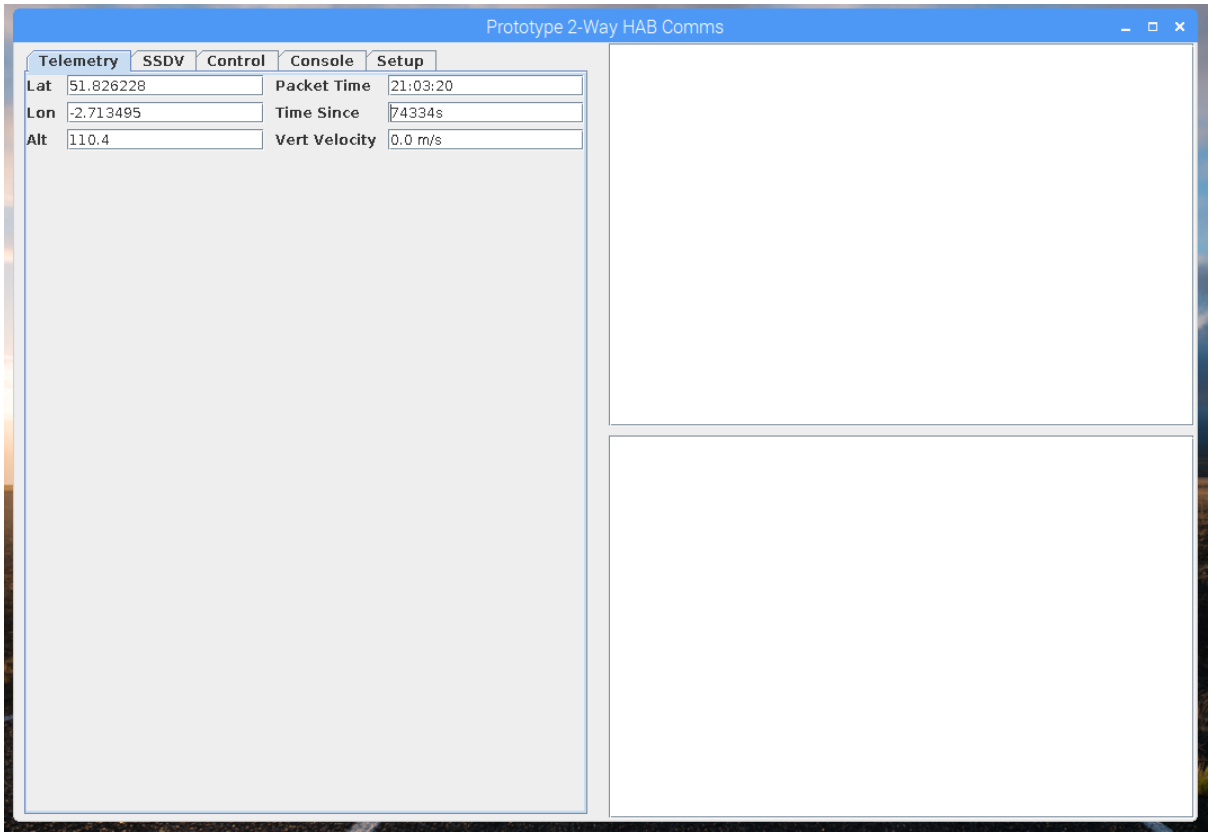


Figure 30 - Screenshot showing location data being displayed correctly in degrees.

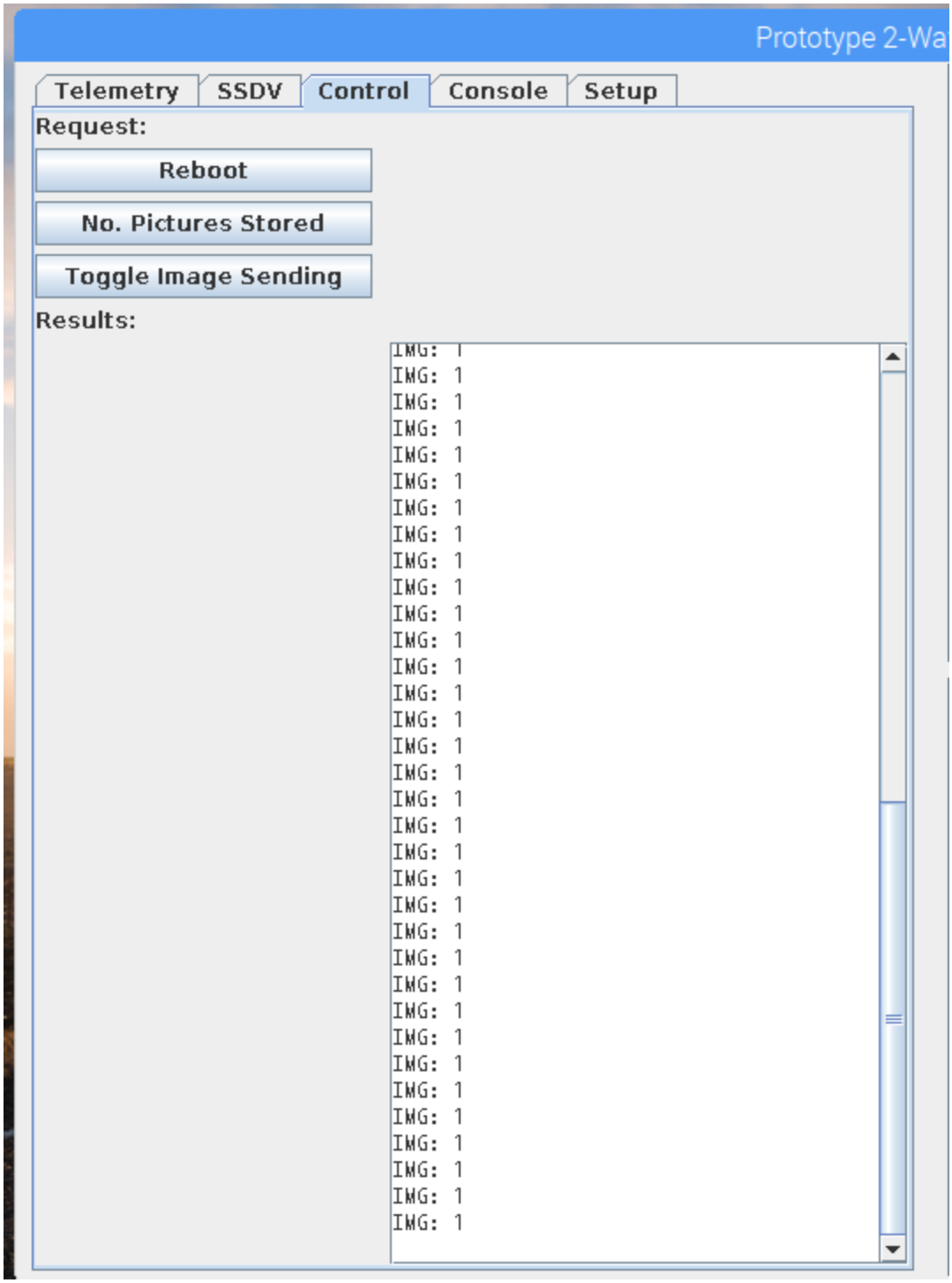


Figure 31 - Shows the remote control results console with a scroll bar.

Evaluation

Achievement of Objectives

I have quoted below my objectives from the analysis. I have only quoted the main headings not the detailed descriptions. Below each point I will evaluate how successfully I have achieved it (in *italics*).

Ground Station

1. The transceiver controller should:
    a. Provide the user with a clear front-end.
    The interface clearly shows in labelled dialogue boxes data which has been received from the payload; as well, all tabs are clearly labelled with self-documenting identifiers. However, there are some areas where the purpose of an input field may be unclear, for example, the console input field is not clearly labelled. Additionally, the transmission and receiving console are not labelled, although in this case the direction of the arrow denotes the direction of the transmission.
    b. Provide a configuration file which allows the user to set the callsign, transmission frequency and bandwidth, the receiving frequency and bandwidth, the error coding rate, the spreading factor, the transmission power and whether the payload is using explicit packet headers. The user should also be able to set the key for 2-way packet authentication.
    This has been achieved clearly as shown in testing, the config.yml file is produced by the ground station and allows the user to modify parameters easily (YAML is a very straightforward configuration system). Changes to parameters are only taken into account when the program restarts, this could be improved in future to allow changes to radio parameters on the fly.
    c. Allow normal LoRa receiving using an 868MHz module or a 434MHz module.
    The ground station can receive normal telemetry and SSDV and it will display this data to the user. So this software could be used to receive data from any LoRa payload transmitting UKHAS telemetry and SSDV.
    d. Decode SSDV image packets and display on the graphical interface, also forward SSDV image packets to the habhub servers.
    As shown in testing, SSDV images are decoded as they are transmitted and are displayed correctly in the SSDV tab, this includes partially transmitted images and images with missing packets, all are displayed correctly (with block colour replacing missing/non-transmitted packets). Additionally, as shown in testing, all SSDV packets are forwarded to the habhub server and displayed correctly on the ssdv.habhub.org website.
    e. Parse telemetry, ignoring those with failed checksums or incomplete data and showing the relevant data on the telemetry display.
    As shown in testing, telemetry is parsed and displayed, with each component

in a separate field, in the telemetry tab. All packets with invalid checksums are ignored and all packets with no data are ignored.

f. All received packets should be logged to a file with a timestamp.
When the program is launched a log file, named YYYY-MM-DD-HH-MM-SS.txt, is created (i.e. the name is the timestamp) and every transmission both to or from the payload that is received/transmitted by this ground station is logged to this file. This actually goes beyond the requirement.

g. Switch to transmit mode after receiving a packet from the payload stating that it is entering listening mode and send any 2-way communication packets that are queued.
As required, when the TRA packet is received the ground station transmits 10 queued 2-way packets (if available).

h. Allow the user to queue packets for transmission by either clicking one of the command buttons in the control menu or by sending a remote console command.
This works as required, users can click one of the request buttons on the control tab to queue a packet or enter a (valid) command in the console. One area for improvement here would be perhaps to display a live graphic or textbox showing the current contents of the transmit queue, perhaps even providing an option to clear the queue.

i. Allow two-way communication with the airborne payload as described.
This works as required, as shown by my flight test the system clearly worked at distance and allowed me to effectively have complete remote control over my payload. Although theoretically any action can be achieved through the remote console, it would be a suitable extension in the future to add more request options in the control tab, it might be useful to be able to adjust radio settings remotely and it might be useful to add more sensors to the payload and then give an option to get data from those sensors.

j. If the user has configured their payload on the web portal, the ground station should be able to, given the payload callsign, download the payload configuration and write the configuration file for the user. The user will, of course, have to enter the key themselves.
This was demonstrated to work completely correctly in testing, though if invalid parameters are downloaded from the database or input into the config.yml file the radio will function unpredictably. Theoretically, however, it should be impossible to input invalid parameters into the website, except setting frequency to a value that the particular radio does not support.

k. Allow the user to toggle uploading data to the server (effectively allow 'offline mode' for testing).
This is achieved correctly, there are two tick boxes in the setup tab which allow the user to turn uploading of data on and off by simply toggling a bool flag.

Server

2. The server should:
    a. Wait for telemetry and SSDV data to be received and then forward this data to habhub.
    Data is forwarded to habhub and is displayed on their servers correctly as it is for all other payloads registered on habhub.
    b. Provide a means to export 2-way communications data in CSV format from the web portal.
    This works as intended, allowing exporting of the packet type and data from each packet. This could definitely be improved, by allowing users perhaps to export each packet with a timestamp of when it was logged. This would be a suitable improvement.
    c. Allow users to view a live log of telemetry packets and 2-way packets received by the server, also indicating when an image packet is received.
    This works adequately, however, the interface is clunky as it is simply a web-page set to refresh every 2 seconds. Additionally, it might be good if image packets could give more information than just 'IMG PACKET', perhaps the payload callsign, image ID and packet ID could be displayed here also.
    d. Provide a web portal to allow users to configure their payload in the database.
    This works as intended and appropriate validation is in place to prevent obviously invalid inputs. It is impossible to prevent invalid inputs entirely, however, because the software cannot know the exact range of available frequencies to the LoRa module being used.

Payload Software

3. The payload software should:
    a. Transmit standard telemetry and SSDV on LoRa.
    This is indeed achieved, standard telemetry and SSDV are transmitted on the LoRa for the majority of the time, telemetry and SSDV are transmitted on every cycle (technically, SSDV can be toggled). There is no potential improvement here except that potentially the rate of packet transmission could be improved by reducing the time gap between transmissions. Detailed testing would be needed to ensure the minimum time gap before which packet loss occurs because the ground station cannot keep up.
    b. Have a configuration file functioning in the same way as that of the ground station.
    This is achieved correctly. The same code was used as is used for the ground station configuration.
    c. Allow the 2-way communications to function as described in section 1, allowing the user to view diagnostics, access shell remotely, reboot, control transmission mode, etc.
    As described in the evaluation of ground station objectives, this functions perfectly though could be improved with additional features. The payload

correctly responds to all request packets as required and executes shell commands and responds with the results. This functions perfectly and as intended with no known issues.

d. Add all packets which are required to be transmitted to a queue so that they can be sent in required order.
   Packets are always transmitted in the correct order as a queue is used as required.
e. Read from the GPS regularly, updating the current telemetry data so that the most up-to-date telemetry is transmitted each cycle. Additionally, it should clear the serial cache after reading a location fix from the radio as otherwise the buffer will fill up with old location fixes.
   This is achieved as required, the current location fix is updated regularly as is the current telemetry string, meaning a current location is always transmitted in telemetry making tracking and locating the payload significantly easier. As required, the serial buffer is regularly cleared so a queue of location data does not build up.
f. Take images at fixed intervals using the Raspberry Pi camera module.
   Images are taken at fixed intervals and are stored in a directory called 'images' in the same directory as the jar file. The images are stored with a timestamp making it easy to work out how far along the flight they were taken.
g. Should be designed to continue functioning without failure under unforeseen circumstances.
   In the event of an exception, most should be caught and handled appropriately, in the event that an exception is not caught there are several catch-all clauses which catch the generic error: 'Exception'. I've tested the payload and it runs continuously without issue. The only issue that does occur after an extremely long runtime (this would be several days) is that the disk fills up and images can no longer be taken, however, the payload continues to transmit telemetry and image packets (of the last image taken) even in this state.

Client Feedback

I have published my project on GitHub with detailed installation instructions for members of the UKHAS to use and hopefully modify to suit their projects. The prototype that I've built hopefully provides a good framework for members of the UKHAS to build other projects which involve 2-way communications with HAB payloads. , as noted above, also joined me for both my launches and gave some feedback on the functionality of the software. I s comments are below, sent me this as a pastebin over IRC.

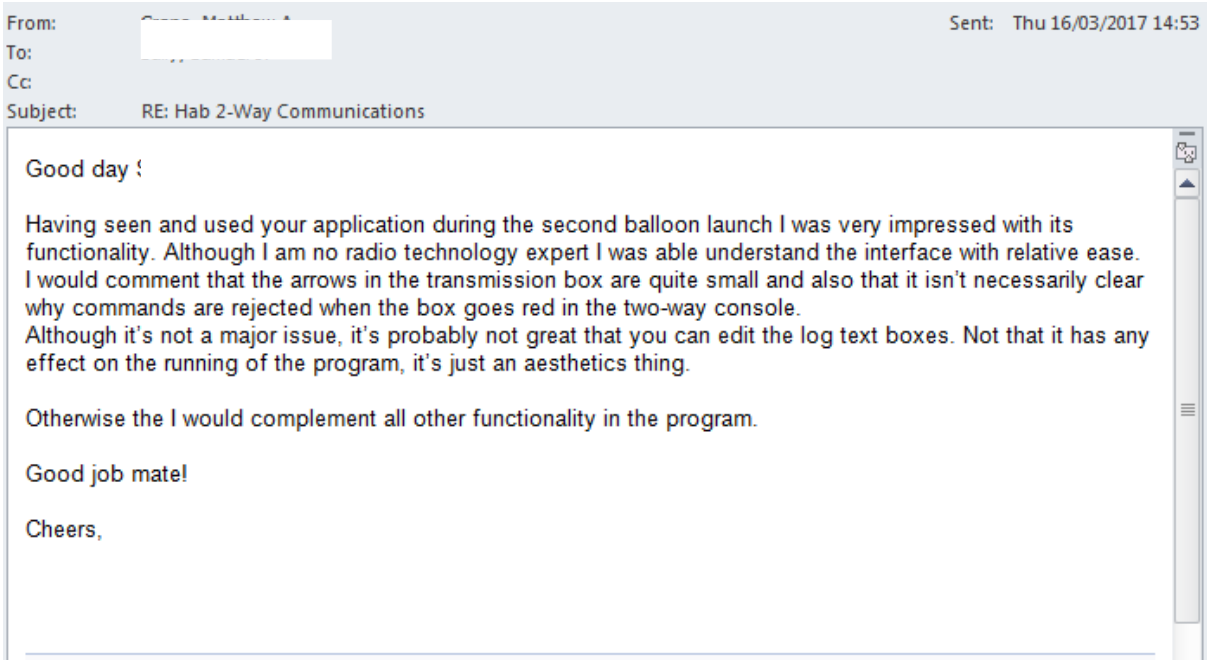
Background: I'm a software and hardware engineer, and have been involved in HAB (high altitude ballooning) for about 6 years as a hobby, and through training teachers and students how to run their own HAB projects. I gave Sam some advise through his project however, unlike the vast majority of balloonists, he designed and built his own electronics and designed and coded his own software.

Every high altitude balloon transmits its own position, and many transmit image data also. However Sam has gone further by providing an uplink from the ground to the balloon, so his flights have a true two-way link. I saw this in operation myself at his launches. Only a handful of people are doing this with balloons, which is a shame as it opens up a world of possibilities, but I'm sure that this will increase now that people like Sam are demonstrating what can be done.

Further, Sam has not only written the tracker software, but also the ground-station software with telemetry decoding, image decoding and display, the uplink of commands and showing the status of both uplinks and downlinks. I can think of only 3 people (myself included) who are doing this currently, and for a school student to do it is truly remarkable.

As you can see was impressed with my project, achieving two-way communications is something that only a handful of HAB enthusiasts have done as of yet. On the launch day was interested to see my software in use. Clearly I've successfully produced a piece of software which can be used by members of the UKHAS. Furthermore, a team of radio enthusiasts from Budapest have begun using my software for prototyping of a WAN.

Additionally, I run a club at my school which aims to get other students into HAB and asked for some feedback from two members of this club, , an amateur radio operator who's going to study Electronic Engineering at Swansea University; and , who's going to study Computer Science at Exeter University.



From:
To:
Cc:
Subject: RE: Hab 2-Way Communications
Sent: Thu 16/03/2017 14:53

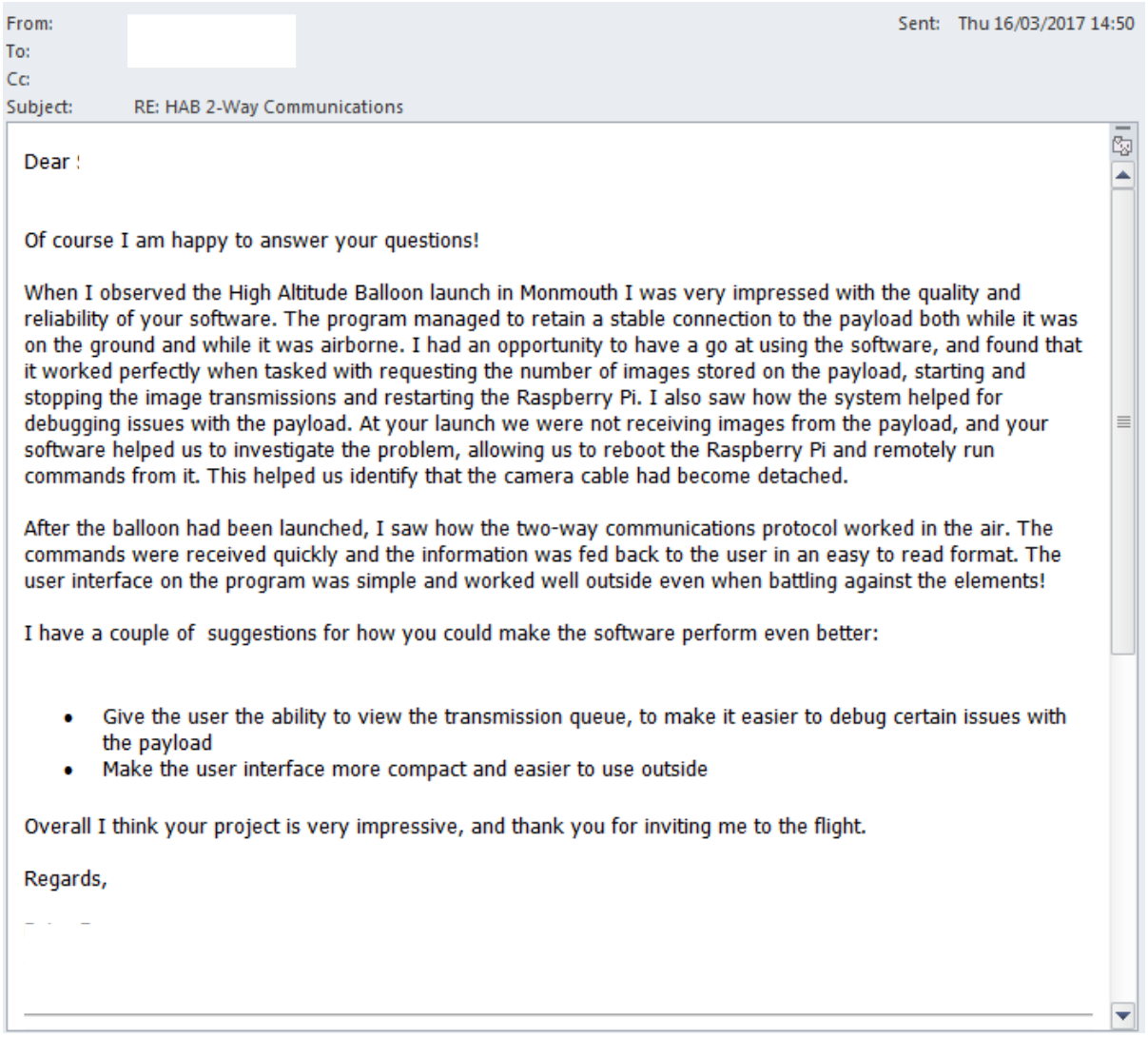
Good day

Having seen and used your application during the second balloon launch I was very impressed with its functionality. Although I am no radio technology expert I was able understand the interface with relative ease. I would comment that the arrows in the transmission box are quite small and also that it isn't necessarily clear why commands are rejected when the box goes red in the two-way console.
Although it's not a major issue, it's probably not great that you can edit the log text boxes. Not that it has any effect on the running of the program, it's just an aesthetics thing.

Otherwise the I would complement all other functionality in the program.

Good job mate!

Cheers,



From:
To:
Cc:
Subject: RE: HAB 2-Way Communications

Sent: Thu 16/03/2017 14:50

Dear

Of course I am happy to answer your questions!

When I observed the High Altitude Balloon launch in Monmouth I was very impressed with the quality and reliability of your software. The program managed to retain a stable connection to the payload both while it was on the ground and while it was airborne. I had an opportunity to have a go at using the software, and found that it worked perfectly when tasked with requesting the number of images stored on the payload, starting and stopping the image transmissions and restarting the Raspberry Pi. I also saw how the system helped for debugging issues with the payload. At your launch we were not receiving images from the payload, and your software helped us to investigate the problem, allowing us to reboot the Raspberry Pi and remotely run commands from it. This helped us identify that the camera cable had become detached.

After the balloon had been launched, I saw how the two-way communications protocol worked in the air. The commands were received quickly and the information was fed back to the user in an easy to read format. The user interface on the program was simple and worked well outside even when battling against the elements!

I have a couple of suggestions for how you could make the software perform even better:

- Give the user the ability to view the transmission queue, to make it easier to debug certain issues with the payload
- Make the user interface more compact and easier to use outside

Overall I think your project is very impressive, and thank you for inviting me to the flight.

Regards,

As you can see in both cases, the impression is that the software works very well and achieves its primary goals of enabling reliable 2-way communications with an airborne HAB.

· commented on a particular aspect of one of my launches that demonstrates the utility of my software, just before launch, images stopped being transmitted and I couldn't work out why, however, using the remote console I was able to ascertain that the camera cable had become detached, which was an easy fix. Normally, I would've had to open up the payload, and plugin HDMI and such for this. He also comments on its stability throughout the flight and the capacity to control and monitor the payload remotely while it was airborne, he comments that requesting image number, toggling image sending and rebooting the Pi worked as required when he used the software.

· commented that the user interface was fairly easy to understand. He did give some suggestions, such that the transmission queue should be made visible, this is a feature I think would be a good addition to the software, it would certainly be useful to be able to see all the packets that have been prepared for transmission, perhaps even with the option of removing some if you've changed your mind about a particular command, or perhaps made a typo! also suggested that the user-interface be made more compact for use outside, the software was being run off a Pi which was being accessed remotely on a tablet,

this did make it a little clunky when attempting to interact, and I remember that did actually trigger the payload to reboot twice when meaning to request number of pictures stored! So perhaps I've not achieved my aim of making the software function well on a touchscreen, it might be useful to redesign the UI with a phone/tablet in mind, not using Java Swing as this is fairly constraining!

commented that, again the software was easy to use, though he commented that when the remote console command box goes red, although it is clear that the command was rejected, he had no idea why they were being rejected. Additionally, he commented that it was a little annoying that you can edit the text of the transmission logs, which is easily rectified!

Potential Extensions and Improvements

I have a few suggestions for potential extensions and improvements to this project, some of which have been informed by the comments received in my client feedback.

1. **Ergonomic and Aesthetic Design -** There is no denying that my interface is slightly clunky, while this is fit for purpose for this is not targeted at naïve users, it's always nice when an application has pleasing aesthetic design, while not a priority in my prototype it is now something that I can turn to. One suggested method of making use of the software ergonomic would be perhaps creating a web interface that can be accessed by any device on a local network, meaning all local devices, from mobiles to tablets to laptops, could access the control system with ease. This would make the software easier to use and would be easier to design a good UI for HTML/CSS are better tools for this than Java Swing. Obviously, the local network would require password authentication! Another suggestion along the same lines of this was perhaps an app that connects (again, via local network) to the ground station software on the Raspberry Pi, this would be equally easy to use and would provide easier portability.
2. **Transmission Queue –** following on from comments, it would be very nice to have access to the transmission queue to see what commands had been queued for transmission on the ground station, perhaps even allowing on-the-fly editing of the commands or removal of commands. Further on this point, it might be nice to implement the queue as a priority queue rather than a normal queue as this would allow important commands to be prioritised over others.
3. **Slave Network –** in my research, I discussed a slave network being created to allow other members of the UKHAS to setup ground stations that would contribute both to transmission to and receiving from a payload while still giving full control over what is transmitted to the owner of the payload. While I decided that this was not necessary in this prototype, it might be a nice thing to start working on now, as it would certainly increase the range of the 2-way communications, perhaps even allowing for flights to be sent around the world, with different stations gaining contact with it as it flies.
4. **Further Request Features –** while the two way system now provides a functioning framework that allows complete remote control of a payload, it would be nice to add

support for more remote requests, perhaps CPU temperature and possibly the integration of some sensors on the payload which could then have their state polled remotely via the 2-way system. For this, however, I would need to add a hat to my Pi with some sensors on it. Perhaps the development of a standard board that includes all the required components for this would be suitable (has created such a board, called the PITS (Pi-In-The-Sky) board).

5. **Custom Telemetry Format –** at the moment, all telemetry transmitted from the payload follows the standard format of $$CALLSIGN,ID,TIME,LAT,LON,ALT,SATS*CSUM\n, however, it might be a suitable addition to allow users to define their own telemetry format, perhaps to include data from the aforementioned sensors which could be added to the project.
6. **Image History –** at present, only the current image that is being transmitted is displayed to the user, all others are discarded; it might be a good idea to have the last *n* images available for the user to see perhaps in a grid-like display on a popup window.
7. **Prediction Integration –** it might be an interesting feature if the latest data from the payload was used to predict it's landing location, this could be achieved by using the CUSF landing predictor which is used by the UKHAS generally and is known to be very accurate.
8. **Multiple Radios –** it might be interesting to try to experiment with the parallel radio system that I suggested in my research where we have 2 radios running continuously in parallel at different frequencies allowing 100% duty cycle on both uplink and downlink. It might be inhibited by interference but further experimentation is required to determine feasibility.
9. **Live Videos –** something that might be feasible with this would be to, on request from the ground station, transmit live analogue video (not high quality of course) for a few seconds, albeit the LoRa radios would not be suitable for this as their modulation is handled internally, however, this would be a very interesting and exciting feature if it worked!

Final Conclusions

Overall, I feel that my project has met the requirements set out in my objectives and has been well received by HAB enthusiasts in the UKHAS. It will hopefully be used in future by members of the UKHAS in their own flights, perhaps even forking my GitHub repository to modify the code themselves (perhaps implementing sensors). I intend to continue development of this beyond A-Level, perhaps creating a more user-friendly design, as well as attempting to achieve some of my other extensions.

The software has been shown to work reliably in my test flight and my testing has verified that is functions to significant tolerance and is resistant to errors and that the payload's defensive programming means that it is unlikely to fall over (it hasn't done so yet, and it's been running non-stop for several days as I write this).