Effective Ground Systems for Amateur and Research-Oriented Rocketry

Vraj Parikh*, Sebastien Martinez†, and Ken Shibata‡
Georgia Institute of Technology, Atlanta, Georgia, 30332

The introduction of dynamic ground systems to amateur and research-oriented aerospace projects allows for additional flexibility in monitoring launch data. Especially in rocketry, effective ground station software provides real-time telemetry, along with tools for live and post hoc data analysis. The presented system implements this through a lightweight cross-platform library, which can be set up to meet diverse requirements with minimal configuration. During flight, recorded data is maintained through a web application programming interface, supporting the provided multi-display graphical interface and third-party logging software. The inclusion of command-sending and message-logging functionality allows for remote debugging capabilities. Post-flight analysis is simplified through the saving of all data in an local structured query language (SQL) database. Utilizing a custom packet structure, the ground station maintained a 96.8% packet success rate during testing on an L2 class rocket launch with an apogee of more than 475 meters. During the launch, all corrupted and incomplete packets were successfully flagged by the software. Through its modularity and user-oriented interface, this ground station design serves as an effective tool to collect data during both the short burn time of a rocket and during larger, long-term tests. In doing so, the software enables the creation of larger-scale avionics projects with more complex objectives through sophisticated data collection and telemetry.

I. Introduction

Amateur and research-oriented rocketry projects have become increasingly complex with ongoing advancements in their technologies. The sophisticated avionics systems used in these projects often include technologies that require remote interactions with the rocket, such as preflight testing, real-time data logging, remote debugging, and recovery assistance. Ground systems offer an effective way to implement these remote technologies in a cohesive and unified manner.

For the past three years, the Guided Navigation and Control (GNC) project at the Georgia Institute of Technology has been working on various forms of active stabilization and thrust-vector control. The software-intensive nature of this work requires the use of ground systems to observe and verify functionality in a safe and effective manner. However, existing ground systems have limited usability. Many commercial models, including the *Featherweight GPS Ground Station* [1] and the *AltOS* by AltusMetrum [2], are non-customizable and are only intended to work with their respective products. As such, they are unsuitable for projects that utilize custom software or microprocessors. While searches for ground station software yields results by various hobby and collegiate groups [3, 4], these systems tend to be designed for individual projects and often lack good documentation or modular design, restricting their applicability in different scenarios.

The goal of this project is to create a reliable and efficient multi-purpose ground station. The ground station should support projects that involve both high data throughput in short periods and infrequent data collection over extended periods. Furthermore, the software should be fully configurable to adapt to diverse specifications, projects, and hardware. Configuration and setup should require minimal effort and be deployable on a wide range of hardware devices. In addition to rocket launches, the software should also be suitable for static firing, laboratory testing, and long-duration projects (including CubeSat and high altitude ballooning). The ground station software discussed in this paper meets these objectives, and has been used to great success in a live rocket launch as well as for software testing within the GNC project at Georgia Tech.

*Undergraduate Member, Ramblin' Rocket Club, Georgia Institute of Technology, AIAA Student Member 1811013
†Undergraduate Member, Ramblin' Rocket Club, Georgia Institute of Technology, AIAA Student Member 1810520
‡Undergraduate Member, Ramblin' Rocket Club, Georgia Institute of Technology, AIAA Student Member 1810846

II. Design

The ground station functionality is split across three devices. On the rocket, a small library interfaces with mission-specific code to generate packets. These packets can be transmitted to the ground station using any of a wide variety of interfaces. By default, the ground station assumes that a serial-based communication is used. For radio communication, the ground station uses a small C program that can be run for extended periods of time on lower-end hardware (a Raspberry Pi model 4B was used for testing). For live telemetry, the ground station maintains a web application programming interface (API) that applications can query. This allows separate devices connected to the ground station via Ethernet or USB to access recent data. The ground station comes bundled with its own graphical user interface (GUI) for this purpose, although third-party interfaces are also supported. This split design allows for the ground station to be placed in a suitable area in range of the rocket, while viewers can observe the process from a distance on their own devices. If this additional flexibility is not required for a given application, both the user interface and the ground station software can be run concurrently on the same device. The ground station software and GUI support Windows, macOS, and most popular Linux distributions, including Fedora, Ubuntu, Debian, Arch, NixOS, and openSUSE.

To provide a high degree of user-customizability, the software relies on templating to modify source code based on a provided specification. This allows for end users to quickly adapt the core software to fit their individual needs. The autogeneration process relies on a single JavaScript Object Notation (JSON) file, is well documented, and performs a number of internal tests to ensure valid inputs and functionality. Furthermore, the modular design of the software ensures that users can easily extend the application with further functionality. The same templating code also generates the lightweight shared C library that can be copied into rocket avionics. For simplicity, the GUI does not use templating, and instead parses the same JSON file on startup to handle configuration.

Rocket-to-ground communication is handled through a shared C library (denoted as the *Packet Protocol Library* in figure 1). This library contains code to both parse and generate communication packets (as byte arrays), and is intended to be run on both the rocket and as part of the ground station software. To accommodate different rocket avionics interfaces, the library does not require dynamic memory allocation or any third-party libraries (other than the standard C library) and is portable to little-endian architectures. (Porting to big-endian architectures only requires a minor change in the source code.) In particular, the library was compiled using STM32H7 (32bit ARM architecture), Raspberry Pi 3 (using both 32 and 64 bit operating systems on ARM architecture), ESP32 and ESP8266 (32-bit Xtensa architecture), and Arduino R3 (8-bit AVR architecture). The library manually generates packs payloads, avoiding issues regarding inconsistent packing between different hardware architectures.

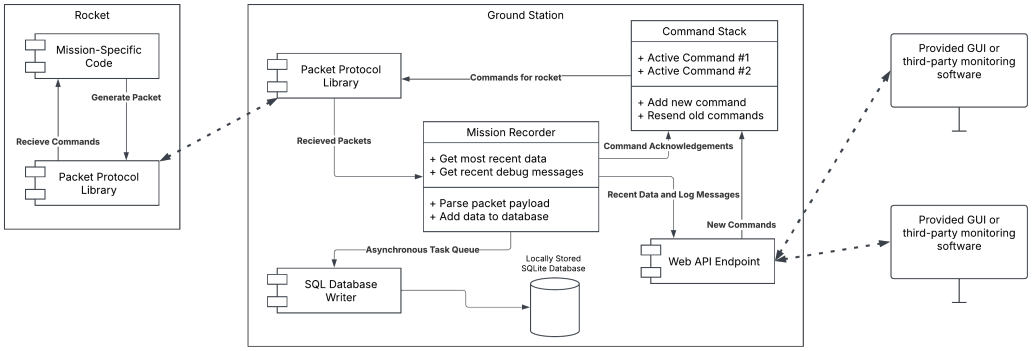

Figure 1. High-level ground station software module overview.

Once packets are parsed by the ground station packet protocol library, they are fed to the mission recorder module, which is responsible for organizing incoming data. This module stores received information and provides a wait-free, non-blocking interface where concurrently run sub-programs (or threads) can access recently-received information (thread-safety considerations are described in section III.A.1). All recorded data is also stored in an SQLite database. SQLite was chosen because it is a relatively fast and easy way to store data and because SQLite database files can be interpreted by most data analysis programs, including Microsoft Excel. To avoid slowing down other aspects of the ground station, database writes are handled via a data task queue, which automatically generates asynchronous batch updates in a separate thread to ensure high-speed data recording.

Recent data in the mission recorder module is read by the web API endpoint, which can be queried to access recent

data and logs. The API endpoint implements the WebSocket server protocol and can be queried by any software that supports this interface, including the provided graphical visualization software. Third party applications can also use the API to request that commands be sent to the rocket. Requested commands are stored in the command stack on the ground station, where they are periodically resent to the rocket until an acknowledgment message is received or the packet times out. The shared packet protocol library is equipped with functionality required to automatically recognize command packets and generate acknowledgment packets.

Generating mission-specific code is handled through the usage of the Jinja template engine. The user-specified JSON file is first parsed using a Python program, which then feeds parameters into a template to auto-generate C implementation and header files. Both the ground station code and the shared rocket code is created using templates. To prevent invalid configurations, certain checks are performed in Python when parsing the JSON file. Unit tests are also generated, and can be compiled separately to ensure that the code generation was successful. Furthermore, all generated code includes documentation, allowing for easy modification.

A. Packet Protocol

The ground system supports three primary communication categories: data, log messages, and rocket commands. Data packets can only be sent from the rocket to the ground system, and store key-value pairs of data. The user can specify 245 different types of data packets. It is assumed that a given data packet stores related information. However, this requirement is not actively enforced, and exists only for visualization purposes. For each packet, the keys and their respective data type must also be specified in advance. Each data packet also includes a timestamp of when the packet was transmitted from the rocket. These timestamps exist to combat potential latency delays in radio communication between the rocket and the ground station. The rocket can also send log messages to the ground station, which are internally stored as 16-bit unsigned integers to decrease packet size. The user must specify a list of all potential log messages in advance, with a maximum of 65,536 distinct messages.

For ground-to-rocket communication, the command interface is provided. Unlike the other two categories, command packets require an acknowledgment packet from the rocket. Until this acknowledgment packet is received, the ground station will continue to resend this command packet. Each command contains a unique id that ensures that rockets to do not execute the same resent command multiple times.

All packets are represented as an array of bytes. Numbers within each packet are ordered using little-endian (least-significant bytes come first). As such, the ground station supports any wireless or wired communication method that allows for direct byte transfer, including any serial communication-based method. Each packet contains the same 4-byte overhead, regardless of type. The first three bytes of the packet form the header, and consist of a zero-valued start byte (signaling the start of a new packet), a one-byte message ID, and a consistent overhead byte stuffing (COBS) encoding [5]. Immediately following this are the payload bytes, which differ based on the message type. Following the payload bytes is a 8-bit cyclic redundancy check (CRC-8), which is used to verify data integrity. Note that a packet size byte is not required, as the packet size is uniquely defined by the message ID.

No. bytes	Description
1	Always zero; allows detecting a start of a packet easily, as a valid does not contain a zero byte in any non-first byte
1	Message ID
1	COBS encoding byte
n	Payload (n is defined by the message ID)
1	CRC-8 checksum

Message ID	Description
0	Not used
1	Rocket pings
2	Commands
3	Command acknowledgments
4	Log messages
5-9	Reserved for future work
10-255	Data message packets

Figure 2. General packet structure and message ID descriptions. Packets are interpreted as byte arrays.

The COBS byte exists to fix the issue where a 0 byte may appear in the payload itself. This would otherwise cause issues when parsing packets, as the errant 0 byte would cause the parser to interpret the packet as two separate packets (with the second one starting with the errant 0 byte). To fix this, the COBS algorithm replaces each 0 byte with the distance (in bytes) to the next 0 byte. The COBS byte stores the distance to the first 0, and the final 0 stores the distance to the end of the packet. To recreate the original packet, it suffices to simply "jump" by the these distances and replace

all specified bytes with 0. To avoid overflow and keep implementation simple, packets are restricted to 255 bytes in size.

Note that the underlying XBee radio module (for which this software was originally designed) packetizes the data into multiple packets, each with 84 bytes of payload [6, p. 130]. Since the XBee conforms to IEEE 802.15.4, it employs an error detection algorithm on the physical (PHY) layer [7, p. 1175]. As such, the checksum is only required to ensure that these 84-byte-packets themselves are not reordered or dropped. If an alternate form of communication is used, a larger hash value like CRC-16 or CRC-32 may be preferable. As such, the desired hash size can be configured when generating the software module.

B. Hardware Specifications

For testing and implementation by the Georgia Tech GNC project, a Raspberry Pi was used as the ground station. Communication was handled by a Digi XBee Pro SB3 module powered by the Raspberry Pi and attached via UART. The Raspberry Pi was powered by an external battery and was connected via Ethernet to a laptop running the GUI. This setup was chosen because it allows the Raspberry Pi and XBee radio module to be mounted in a location within good radio connectivity of the rocket, including locations that would be unsafe for human observers. Given that most Ethernet cables are rated for up to 100m of connectivity, this provides significant flexibility in antenna positioning. For launches, the Raspberry Pi assembly was placed inside a case for ease of use.

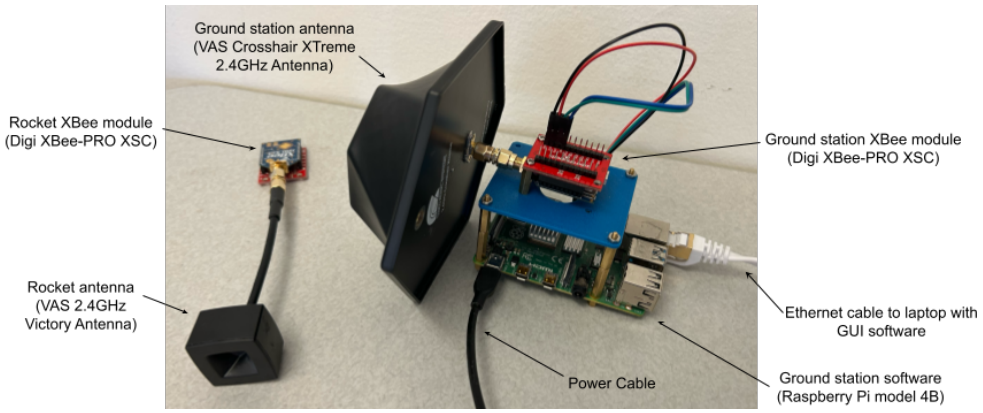

Figure 3. Ground Station Setup. For the launch, an external case was also used (not pictured).

Special consideration is required to ensure minimal data loss using the XBee radio module. The XBee SB3 is a half-duplex radio module - the radio cannot send and receive data simultaneously [6]. Instead, it must buffer data to send until it has finishing receiving, and vice versa. These buffers are internal to the radio module. The serial transmit buffer, which temporarily stores data prior to radio transmission, has a maximum capacity of 768 bytes. If the serial transmit buffer is full, the XBee cannot send new packets until existing packets on the buffer are transmitted. When the buffer has 17 bytes remaining, the clear-to-send (CTS) pin on the XBee module is set to high, indicating that the host should temporarily stop sending data while the buffer is emptied. Conversely, the receive buffer on the XBee can hold 1056 bytes and are transmitted via serial immediately upon being received (unless this behavior is manually overridden). As long as the baud rate on the receiving end is at least as high as the baud rate on the sending end, the change of the receive buffer overflowing should be negligible.

As mentioned previously, the ground station can be run on a personal computer or laptop, forgoing the need for a Raspberry Pi or an alternative device. However, this approach is not recommended for longer-duration tests, as the Raspberry Pi tends to be more reliable and energy efficient. For short launches, this can be a suitable, low-setup option if additional flexibility regarding receiver placement is not required.

Alternative radio modules and communication methods were also considered. In particular, the LoRa (Long Range) protocol and family of transceivers have properties suited for rocketry. LoRa uses a technique similar to chirp spread spectrum and has a typical range of 5-15 km [8]. However, LoRa has a relatively low bandwidth of 20 Kbps, which tends to decreases further as the distance between radio modules increases. This can be sufficient for many rocketry projects, especially when real-time control is not a primary concern. For the purposes of testing, XBee modules were ultimately chosen due to their higher bandwidth and better compatibility with the hardware used by the Georgia Tech GNC project.

III. Implementation

A. Software Specifications

The primary goals of the software implementation are for the system to be reliable, fast, and easy to use. The latter of these goals is achieved through the intuitive graphical user interface and the JSON configuration file. As mentioned earlier, much of the ground station software is auto-generated using templates. This allows the end user to provide a JSON file with the desired specifications without worrying about the underlying implementation details of the ground station.

To ensure that the ground station is reliable, data is written to the device's disk immediately instead of at program completion. The SQLite library is run using the rollback journal mode, which relies on a 2-stage commit process using intermediate journal files. This ensures that the database can always be fully recovered in the case of crashes. Furthermore, efforts are taken to minimize dynamic memory allocation. The core ground station software relies solely on statically allocated memory, and the SQL library uses a single static-allocated buffer for all write operations. While the Mongoose Web Server library used to service web sockets does dynamically allocate memory, the allocations are largely limited to a single allocated buffer per web socket connection. Since Mongoose is heavily optimized for use on embedded systems, it is unlikely that this has any significant impact on reliability.

The final software challenge is ensuring thread-safety. During normal operation, the ground station needs to manage numerous tasks simultaneously, including reading serial data, servicing web API connections, and storing data. To handle these tasks efficiently, the ground station runs several threads concurrently. However, three key difficulties arise when designing algorithms to do these operations efficiently.

1) Many CPUs (especially ARM CPUs) reorder instructions, which may result in *theoretically* thread-safe code before unsafe upon execution.
2) Even when code is executed correctly, data that is modified in one thread may not appear to be modified in another thread until that value is eventually stored in shared memory (instead of local cache).
3) As has been shown by many authors [9–12], using the built-in sequential consistency model, which forces all operations to not be reordered and be pushed to shared memory instantly, can significantly impact performance.

Using atomic types (special data types designed to be shared by threads) does not inherently fix this issue, although it does prevent a separate issue called data tearing (when two threads try to write to the same memory at the same time). Instead, to circumvent the aforementioned difficulties, the C standard provides the less-stringent acquire/release consistency model. This model allows for manual control over which values can and cannot be reordered and over which values need to be pushed to shared memory immediately. This additional flexibility has been shown to provide a significant performance improvement over the sequential consistency model [11]. However, using this model requires careful design considerations to achieve thread safety.

1. Reader-Writer Consistency

To store the most recent rocket data, a reader-writer consistency mechanism was implemented. In particular, each type of data packet corresponds to its own reader-writer mechanism, allowing other threads to quickly view the last sent packet of each type. While this could be implemented by simply storing an atomic variable for each data value in each packet, this option was not chosen. Using atomics would restrict the supported data types to values that could be efficiently updated atomically and would allow for edge cases where a reader would query a data packet where half of the atomic variables had been updated to a new packet while the other half were stale. This first issue also ruled out the usage of sequence locks, the mechanism used by the Linux kernel for ensuring reader-writer consistency. As observed by Hans Boehm [13], it is not possible to create a sequence lock that is efficient on ARM architectures without restricting all data to values that can be updated atomically.

Instead, for each data packet, an list of identical data containers was created. The exact length of this array is arbitrary, although implementation details require it to be a power of 2. Each container contained all of the values specified by that data packets (which do not need to be atomic), along with an atomic counter of the number of readers. Lastly, an atomic value tracked the current active container. To perform a write operation, the writer accesses the container in front of the current active index, and stores the new values there. After writing the new values, the writer increments the active container tracker, so that it now points to the written container. Reader threads simply access the container referenced by the active index tracker to perform reads.

In theory, there exists an edge case where a writer may write n times (where n is the length of the array) before a reader completes its read. In this scenario, the reader may read values while the writer updates them, resulting in torn reads. To avoid this, prior to reading, the reader first increments the reader counter on the container being read, and then

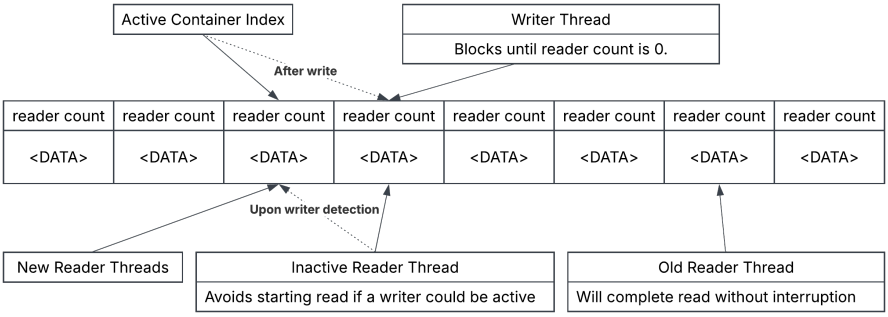

Figure 4. Data structure used to manage concurrent reads with a single writer.

checks if the current active container pointer is now one before the container it is reading. If it is not, the reader will read the current values, and if it is, the reader will decrement the reader counter and run the process again. When the writer attempts to write to a container, it will first block until the reader counter on this container is 0. This ordering guarantees thread safety because it ensures that a read operation will never start if a writer thread could be currently writing.

To ensure that this logic is not invalidated by operation reordering, the reads and writes to the active index tracker must use acquire and release memory models, respectively. However, the two pre-read operations (incrementing the reader counter and checking the current index) must be performed using sequential consistency to ensure that this order is observed by the writer thread as well.

2. SQL Task Queue

The SQL task queue is a standard multi-producer single-consumer task queue. Numerous wait-free options exist for this purpose, with various benefits and drawbacks. The final implementation was based on the multi-producer, multi-consumer queue designed by Eric Rigtorp [14], as it provided a lightweight and high-performance queue that ran on a bounded size array without requiring dynamic memory allocation. If dynamic memory allocation is not a concern, the unbounded queue designed by Michael and Scott [15] was found to be equally effective.

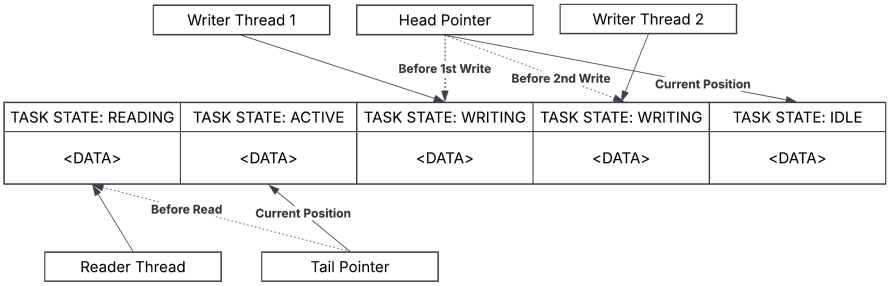

Figure 5. Task queue used to store pending SQLite database operations.

The queue consists of a fixed-length array, where each array element represents a task in the queue. An atomic integers tracks the head index, while a normal integer tracks the tail index, and a second atomic enumerated value tracks if an element has been written to. This enum is a departure from the turn counter used in Rigtorp's queue, which bypasses the issue of a full task array by blocking until the the desired rotation (or turn) of the queue list. However, the enqueue operation used by the ground system avoids blocking the writer if the queue is full, and simply discards the saved data. The rationale behind this decision is the fact that blocking the input thread will result in other information being lost *while also preventing the API from updating this data*. Discarding instead of blocking ensures that at the very least, log messages and recent data will still be made available to the API instead of being lost. As detailed in section 5.1, measures have been taken to ensure that the queue does not reach max capacity. In practice, the chance of filling the queue should be negligible. The ground station will also provide warning messages prior to filling up fully.

All atomic reads and writes for the queue can be run using only acquire and release memory models. In this scenario, sequentially consistent reads and writes are not required because readers and writers act based solely on the value of the atomic enumerated state in each array element. These enums take advantage of atomic compare-and-swap operations

ensures that updates to this individual value are observed by all other threads before values within that array element are accessed. Since no other atomic operations are required to determine read/write privileges, the CPU may freely reorder other operations without issue.

As a small implementation detail, note that both the task queue and the concurrent reader-writer mechanism pad array elements and other atomic counters to a multiple of the CPU's hardware interference size. This prevents the issue of false sharing of cache lines between threads. On most ARM and x86 architectures, the cache line size is 64 bytes. In cases where a different hardware interference size is used, this value may need to be modified manually.

3. Other thread safety mechanisms

Lastly, both the command tracking mechanism and the debug message queue rely on a simple list of atomic variables. Commands are stored in an array where the i-th element represents the status of the i-th command. Each tracked command stores the last resent time (4 bytes) and the number of resend attempts (1 byte), and the command unique id (one byte). This is padded and stored as a 64 bit atomic value.

For debug messages, an array of size n stores the last n debug messages. Each element in the array stores the message itself (1 byte), the message parameter (4 bytes), and the revolution (2 bytes). The i-th debug message is stored at index $i \pmod n$ with a revolution of $\lfloor i/n \rfloor$. This allows for accesses to debug messages to quickly check a single index's revolution to determine if a message is present.

Both structures were padded into a 64 byte atomic type. If the hardware interference size is more than 64 bytes, than each atomic array element should also be padded to the hardware interference size to prevent false sharing (on most architectures, this should not be required). Since all data relevant to a single command or message is stored in a single atomic value, and no other operations are required, all methods can be implemented using atomic compare-and-swap operations with a fully relaxed memory model. No sequentially consistent or acquisition/release-based semantics are required.

B. Graphical User Interface

The ground system's GUI utilizes Dear PyGui, a Python library with GPU-accelerated rendering, making it ideal for real-time data visualization. Furthermore, its drag-and-drop interfaces allow users to fully configure the viewer to display information of interest.

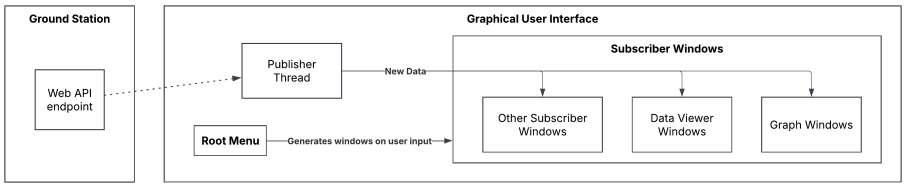

Figure 6. High level overview of the subscriber-publisher model used to update GUI windows.

The GUI handles data visualization based on the idea that data can be grouped into categories based on the data packet it is associated with. Based on these user-defined configurations, the GUI defines one window type per data packet, in which data from the most recent packet of this type is displayed. Since data fields in packets are intended to be grouped by category anyway, this serves to split data reporting into intuitive categories. These windows can be individually opened from the root menu bar, allowing full user control over what information is displayed.

Within data-viewer windows, the GUI supports a number of dynamic container types. Aside from primitive integer and floating point values, the GUI is also capable of formatting vectors, quaternions, and euler angles. Implicit unit conversions, cartesian to polar vector conversions, and quaternion to euler angle conversions are also supported. End users have full control over viewport value types.

For commands and debug messages, the GUI provides a log window where users can see debug messages received from the rocket and any ground station status messages. The GUI can automatically convert the integer representation of debug messages into human-readable logs. Filtering and searching functionality is also provided within the log. As with data viewer windows, the log window can be opened from the root menu bar.

The root menu bar also provides a number of convenience utilities for further data analysis capabilities. This includes the ability to graph any data value (or a group of three or less data values) on a time series plot and the ability to visualize live GPS data on a map. Map images are taken from OpenStreetView and can be downloaded in advance for

offline usage.

GUI updates follow the publisher-subscriber model. A single "publisher" thread queries the web API at fixed intervals to access recent data. When windows are created via the root menu bar, windows automatically subscribe to relevant data. Then, on data updates, the publisher thread notifies all components subscribed to a particular data value of that value's change. This design framework makes it easy for users to extend the GUI with new functionalities. Adding new windows consists of fetching the desired data by "subscribing" to the appropriate categories and registering a module with the GUI's window manager, which will automatically cause it to appear in the home screen.

Since the GUI allows users to freely open, move, and transform windows, setup can require some time. To mitigate this, the GUI also has methods to save the current state of the GUI for future sessions. Users can also save multiple layouts in advance, allowing them to quickly switch between them in different scenarios.

IV. Testing

Due to its role as a real-time data collection mechanism, it is crucial that the ground station operate fast enough to keep up with inputted data. The single biggest bottleneck in meeting this goal is the SQL database itself, as much of the other sections of the code contain relatively minimal logic, use wait-free (or mostly wait-free) operations, and avoid writing to disk.

To test SQL speeds, 5 different entry types were used, each of which were 32 bytes long in total. A variety of floating point and integer types were used. Rather than feeding data via serial input, a generating function was used to quickly generate ordered data for each test. The original behavior consisted of using string formatting to manually generate and parse (using the SQLite execute interface) an SQL command for each write. As an attempt to optimize this process, the code was then refactored to instead rely on a prepared SQL statement, which could be compiled once at the start of execution and then used repeatedly by binding values. In total, each test consisted of generating/writing 50,000 entries per entry type. All entries were successfully verified after the test was complete.

A common way of optimizing SQL write speeds is the use of transactions. Internally, SQLite begins a transaction at the start of each command, and ends it after the command. Information is written to disk only when a transaction ends. By manually creating and ending a transaction, multiple write operations can be combined into a single batched disk write, resulting in potential speedups. However, this performance increase comes at the cost of decreased reliability, as sudden crashes will cause any data stored in an incomplete transaction to be lost. As such, wrapping all writes into a single transaction is unideal because it would result in total data loss in the event of a crash.

Instead, the program takes advantage of the asynchronous task queue to process SQL write operations in process SQL operations in batches. The SQL database writer thread begins a transaction when the task queue is non empty, and then adds entries to the transaction. The transaction is completed when the queue is empty or the desired batch size is met, whichever comes first. This strikes a balance between throughput speed and reliability, as it creates an upper bound on the amount of data lost in a crash. Testing of the impact of batching was conducted using the same setup as the first trial. Different batch sizes were used to observe a relationship between batch size and performance.

To then verify full-program latency, packet data was fed in via spoofed serial ports with configurable packet frequencies. The packet frequency was set to 95% of the theoretical max found in the previous test. After running the setup for 4 hours, the database was inspected to ensure that all sent packets were accurately stored. During these tests, both size-bounded thread safety mechanisms (the task queue and the reader-writer concurrency mechanism) were monitored for overflow. Reading was simulated by a single web socket connection, which read all information at a frequency of 15Hz, the default used by the GUI.

Testing Component	Model
Ground Station Software	Raspberry Pi 4B
Communication Module (only on rocket test)	Digi XBee Pro SB3 Radio Module
Rocket GPS	ZED-F9P GPS
Voltmeter (not used for rocket test)	Klein Tools ET-920 USB-A and USB-C Digital Meter
Ground Station Antenna (only on rocket test)	VAS Crosshair XTreme 2.4Ghz Antenna
Rocket Antenna	VAS 2.4GHz Victory Antenna

Figure 7. Hardware used for ground station testing.

For all three tests, the Raspberry Pi's power source was run through a USB voltmeter. This allowed for the energy

consumption of the Pi to also be monitored.

The ground station was also tested using an L2 class rocket launch. The rocket was equipped with a custom power management and avionics system, and reached an apogee of 502m. Hardware specifications are detailed in figure 7. Packets were sent from the rocket to the ground station at a frequency of 60 packets/second, and were logged on the rocket's data recorder. Following the launch, the data on the recovered rocket card was compared to the data recorded in the SQL database on the ground station.

The ground station was placed at a distance of approximately 50 meters from the rocket, as reported by the launch site. Prior to launch, a number of commands were sent to the rocket to initialize software tests and preflight sensor calibrations. Once sensor calibration was complete, as verified by the convergence of the onboard extended Kalman filter had converged, this information was communicated by the rocket to the ground station to confirm launch readiness. The rocket was programmed to maintain telemetry throughout the flight, and provide additional status messages when entering free fall, deploying parachutes, and hitting the ground.

V. Results and Discussion

Figure 8 presents the results of the first SQL performance test. As a baseline, the *no writes* category counts the theoretical max throughput of the generating function itself, without any SQL operations. The *execute* category measures the average performance when using string formatting to manually generate and parse (using the SQLite execute interface) an SQL command for each write. Finally, the *prepared* category measures average performance when relying on pre-compiled SQL statements.

		Writes/Second			Bytes/Second		
		No Writes	Execute	Prepared	No Writes	Execute	Prepared
Entry Types	4 Floats + 4 Ints	7.871×10^7	57.724	55.250	2.519×10^9	1847.183	1768.000
	8 Floats	1.270×10^8	58.224	57.187	4.063×10^9	1863.170	1829.979
	8 Ints	8.543×10^7	56.397	56.421	2.734×10^9	1804.697	1805.499
	4 Doubles	1.490×10^8	56.105	57.904	4.767×10^9	1795.389	1852.929
	4 Long Ints	1.602×10^8	56.917	57.690	5.128×10^9	1821.350	1846.079

Figure 8. Comparing performance between using SQLite3 prepared statements and using SQLite3 execute mode with string format on a Raspberry Pi 4.

As can be seen, using SQL in this manner does not yield desirable performance. On average, generating and executing new commands repeatedly resulted in an average throughput of 1826.4 bytes or 57.1 entries written per second. This throughput would be just enough to read continuously from a serial line with a baud rate of 9600 bps, but would be completely unable to read continuously from a baud rate of 38,500 bps or higher. Since many devices use a baud rate of 115,200bps, this method would be far too slow. Surprisingly, using a prepared statement had no significant impact on the results.

The results of the second SQL performance test are detailed in figure 9. For this test, write operations were combined into transactional batches of varying sizes. Each trial consisted of 125,000 entries. For all trials, prepared statements were used instead of re-generating and executing each command separately. While this decision seemingly has no notable impact on performance, it was made to conform to the recommended design practices given by SQLite.

These results indicate that with batching, an SQLite instance is more than capable of supporting a reasonable data throughput. With a batch size of 50 entries, writes were completed with an average throughput of 96,745.1 bytes or 3023.3 entries per second. This is enough to support continuous input from more than 6 concurrent serial connections at a baud rate of 115,200 bps. For most applications, this throughput was determined to be more than sufficient. If better performance is required, a higher batch size can also be used. At a batch size of 500, the software can support a throughput of 9.32×10^6 bits per second. A throughput this high is enough to support almost 81 concurrent serial connections at a 115,200bps baud rate.

For full-system testing, two tests were ran: one with a batch size of 50 and one with a batch size of 500. For the first test, the packet rate was set to 2,857 packets per second. For the second test, the packet rate was instead 34,586 packets per second. Both values are approximately 95% of the proposed maximum. The results of these tests are displayed in figure 10.

In both tests, the ground station maintained a 100% success rate with keeping up with the provided data throughput.

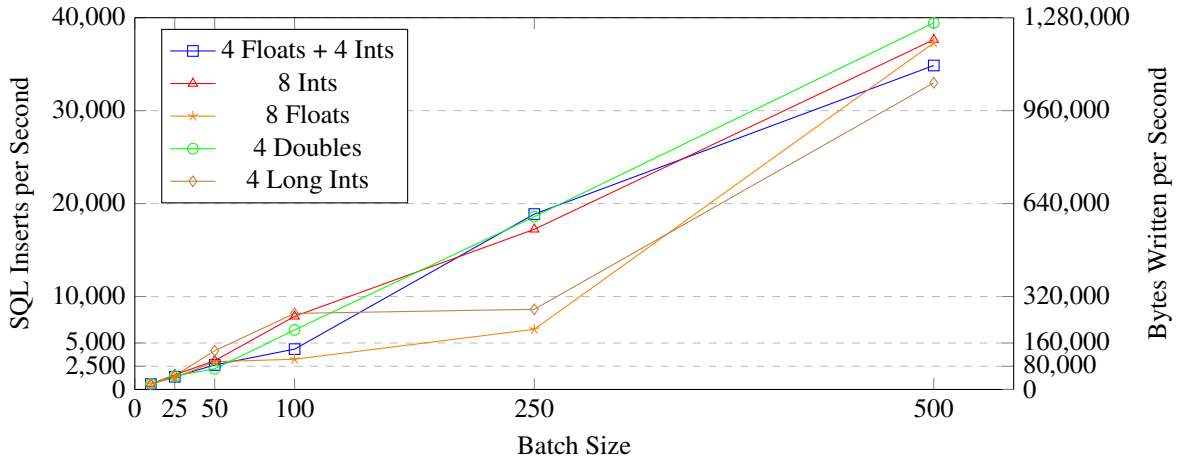

Figure 9. SQLite3 performance on a Raspberry Pi 4b using prepared statements when batching multiple writes into transactions.

In both tests, the SQL max queue size fairly consistently remained within twice the batch size. For the reader-writer concurrency structure, it was found that with an array size of 8 the writing thread was never blocked by a reader, and that the structure was effectively wait-free in practice. If the frequency of write operations is low, the array size can be lowered to 4 or 2 without issue.

Configuration	Packet Success Rate	Max SQL Queue Size
Batch Size 50 + 2,857.2 packets per second	100%	71
Batch Size 500 + 34,586 packets per second	100%	789

Figure 10. Results of the full-software throughput test.

For all tests, the power consumption on the Raspberry Pi remained very consistent. The average power consumption was 4.22W, and peaked at 5.04W. RAM consumption was also fairly consistent, and displayed little variation over the course of the longer tests. In particular, the RAM usage never exceeded the 60% of the total device RAM (2gb). These results indicate that this software would be more than suitable for long-duration missions.

Performance during the test launch largely matched theoretical expectations. Before launch, the ground station was tested to ensure that the connection was stable. While minor interference issues were present with the original location of the antenna, this was due to the presence of a large, unrelated omnidirectional antenna between the ground station and the rocket. Furthermore, when bystanders walked between the rocket and the ground station, packet loss spiked up significantly. Moving the ground station away from other broadcasting antennas and to a location with no foot traffic largely resolved these issues. After making this change, a comparison between recorded packets on the ground station and on the rocket data log revealed a packet loss of less than 0.2%. During the launch, the ground station was able to receive almost all packets from the rocket, with approximately 3.2% of packets lost during the launch, prior to ground contact. Following ground contact, connection seemed to drop. Since the rocket stopped transmitting after landing, the exact packet loss rate is unknown. However, attempts to ping the rocket were unsuccessful. Ultimately, it was revealed that this was due to the rocket antenna being slightly buried into the ground at landing. As soon as the rocket was removed from the ground, the rocket could again be pinged successfully.

VI. Conclusion

The presented ground station software is a robust and efficient option for many amateur and research-focused rocketry projects. It allows for easy configuration and customization of recorded data. It is also easily deployable on a wide range of applications and can be adapted to use different communication methods. The provided GUI package allows for real-time data visualization with minimal impact to data throughput. Due to the separation of the GUI and core software, the GUI can be freely opened and closed on the same device or across multiple devices without affecting ground station functionality. The generated SQL database file maintains a timestamped record of all rocket interactions, allowing for effective post-mission analysis.

This ground station has the potential to be suitable for a wide range of projects. This includes both short-duration, high-throughput use-cases, including rocket launches, static fires, and software testing, and long-duration, low-packet frequency projects like high-altitude ballooning and CubeSat missions. When considering novel applications, additional testing regarding power consumption and reliability optimizations over long durations could be beneficial. Furthermore, using an alternative to the interference-susceptible XBee modules, including LoRa, could reduce connectivity issues. The best ground station is transparent, reliable, and extensible. By combining these characteristics with a suite of customization features, this ground station software sets itself up as an effective contribution to a wide range of future engineered systems with telemetry needs.

References

[1] Featherweight Altimeters, "Featherweight GPS Tracker," , 2021. URL `https://www.featherweightaltimeters.com/store/p22/Featherweight_GPS_Tracker_%28upd%29.html`.

[2] Altus Metrum, "AltOS," , 2024. URL `https://altusmetrum.org/AltOS/`.

[3] University of Victoria Rocketry, "UVic Rocketry Ground Station Software (Proof-of-Concept)," , 2018. URL `https://github.com/UVicRocketry/groundstation`.

[4] Cornell University Rocketry Team, "Ground Software," , 2025. URL `https://github.com/cornellrocketryteam/Ground-Software`.

[5] Cheshire, S., and Baker, M., "Consistent overhead byte stuffing," *SIGCOMM Comput. Commun. Rev.*, Vol. 27, No. 4, 1997, p. 209–220. https://doi.org/10.1145/263109.263168, URL https://doi.org/10.1145/263109.263168.

[6] *Digi XBee® 3 ZigBee® RF Module*, Digi International, December 2023. Available at https://www.digi.com/resources/documentation/digidocs/pdfs/90001539.pdf.

[7] Yu, K., Baracć, F., Gidlund, M., Åkerberg, J., and Björkman, M., "A flexible error correction scheme for IEEE 802.15.4-based industrial Wireless Sensor Networks," *2012 IEEE International Symposium on Industrial Electronics*, 2012, pp. 1172–1177. https://doi.org/10.1109/ISIE.2012.6237255.

[8] Shanmuga Sundaram, J. P., Du, W., and Zhao, Z., "A Survey on LoRa Networking: Research Problems, Current Solutions, and Open Issues," *IEEE Communications Surveys & Tutorials*, Vol. 22, No. 1, 2020, pp. 371–388. https://doi.org/10.1109/COMST.2019.2949598.

[9] Wrenger, L., Töllner, D., and Lohmann, D., "Analyzing the memory ordering models of the Apple M1," *Journal of Systems Architecture*, Vol. 149, 2024, p. 103102. https://doi.org/10.1016/j.sysarc.2024.103102.

[10] Chou, Y., Spracklen, L., and Abraham, S. G., "Store Memory-Level Parallelism Optimizations for Commercial Applications," *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, USA, 2005, p. 183–196. https://doi.org/10.1109/MICRO.2005.31.

[11] Ranganathan, P., Pai, V., and Adve, S., "Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models," *Proceedings of the 1997 9th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA ; Conference date: 22-06-1997 Through 25-06-1997*, 1997, pp. 199–210. https://doi.org/10.1145/258492.258512.

[12] Blundell, C., Martin, M. M., and Wenisch, T. F., "InvisiFence: performance-transparent memory ordering in conventional multiprocessors," *Proceedings of the 36th Annual International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, USA, 2009, p. 233–244. https://doi.org/10.1145/1555754.1555785.

[13] Boehm, H.-J., "Can seqlocks get along with programming language memory models?" *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, Association for Computing Machinery, New York, NY, USA, 2012, p. 12–20. https://doi.org/10.1145/2247684.2247688.

[14] Rigtorm, E., "A Bounded Multi-Producer Multi-Consumer Concurrent Queue," , 2016. URL https://github.com/rigtorp/MPMCQueue.

[15] Michael, M. M., and Scott, M. L., "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, Association for Computing Machinery, New York, NY, USA, 1996, p. 267–275. https://doi.org/10.1145/248052.248106.