The Design of a Low-Latency Motor Control System for Precise Camera Pointing on a Balloon-Borne Stratospheric Telescope

Design Process and Outcomes Documentation: Iteration III

Prepared for the

Engineering Science ECE Capstone Design Course

Authors:

Michael Acquaviva

Joao Bicalho

Shafinul Haque

Robert Purcaru

Antoine Vilain

Regis Zhao

Client: Barth Netterfield, Balloon Astrophysics Group

Date: April 30, 2025

Contents

1. Introduction

Astronomers face a fundamental challenge when observing the universe by making observations through Earth's atmosphere. The atmosphere distorts and absorbs incoming light, limiting the clarity and range of ground-based observations. Atmospheric turbulence causes stars to twinkle and blurs images, a phenomenon known as astronomical seeing (1). Additionally, large portions of the electromagnetic spectrum, particularly ultraviolet (UV), X-ray, and infrared (IR) wavelengths, are absorbed readily by the atmosphere, restricting what can be observed from Earth (2). Furthermore, light pollution and weather conditions introduce additional spatial and temporal constraints on observations, further contributing to the difficulty in making astronomical observations from the surface of the Earth.

Ground-based telescopes, including the Very Large Telescope (VLT) and the upcoming Extremely Large Telescope (ELT), remain the most commonly used observational tools due to their accessibility and cost-effectiveness (3). These telescopes can support large mirrors, allowing for high-resolution imaging and deep-space observations. Unlike space telescopes, they can also be regularly serviced and upgraded, extending their usefulness over decades. However, atmospheric interference remains a fundamental issue. While modern adaptive optics systems can partially compensate for turbulence in real time (4), ground-based telescopes are ultimately limited by their location below the atmosphere.

Space-based telescopes avoid many of these issues by operating above Earth's atmosphere. The Hubble Space Telescope (HST) and the James Webb Space Telescope (JWST) have demonstrated the advantages of orbital telescopes, capturing high-resolution images without atmospheric distortion and accessing spectral bands that ground-based telescopes cannot (5; 6). Space telescopes are critical for observing distant galaxies, exoplanets, and cosmic background radiation, making them indispensable tools for astrophysics. However, their benefits come with significant costs. The complexity of launching and maintaining space telescopes means that projects often take decades to develop and require billions of dollars in funding. Unlike ground-based telescopes, which can be repaired and upgraded, most space telescopes are designed for one-time use, with failures often leading to complete mission loss. While Hubble was serviceable by astronauts, most modern space telescopes do not have that luxury.

Balloon-borne telescopes offer an alternative that balances some of the strengths and weaknesses of both ground and space-based observatories. First developed in the 1950s, these telescopes are carried into the stratosphere—above 99% of the Earth's atmosphere—by high-altitude balloons, reducing the impact of atmospheric turbulence while avoiding the extreme costs associated with space launches (7). Despite their lower cost, balloon-borne telescopes have historically struggled to achieve the pointing stability necessary for high-quality imaging. However, recent advancements, such as the SuperBIT telescope, have demonstrated that sub-arcsecond precision is possible from a balloon platform (8).

While balloon-borne telescopes offer a compelling alternative, they come with their own engineering challenges. Unlike ground-based telescopes, which can be fixed to stable mounts and use adaptive optics (9), or orbital telescopes, which rely on reaction control systems for stability (10), balloon-borne telescopes must actively counteract motion caused by the balloon's pendulations and drift through the atmosphere (11). The stability of the optical

assembly, and therefore the quality of the images produced, is fundamentally constrained by the latency of the control systems used to maintain pointing precision. Addressing this challenge is critical to unlocking the full potential of balloon-borne observatories, and recent technological advancements provide new opportunities to improve their performance.

GigaBIT is a next-generation balloon-borne telescope currently under development, following on SuperBIT's legacy. It features a larger primary lens and more ambitious science goals, demanding even tighter pointing stability. Achieving the required pointing precision for GigaBIT will necessitate an extremely responsive and stable attitude control system. In particular, one key improvement area identified is the latency of the motor control loop used to stabilize and point the telescope. High control loop latency can lead to sluggish responses to disturbances and tracking errors that blur images. Professor Barth Netterfield's team has tasked our capstone design group with developing an electronic control system for GigaBIT that minimizes this latency while reliably operating in the harsh stratospheric environment. The goal is to enable sub-millisecond reaction times in the pointing motors, thereby dramatically improving pointing accuracy. This document presents our progress toward that goal, including background context, stakeholder analysis, project scope and requirements, and the proposed system architecture for a low-latency motor control solution.

2. Context and Prior Work

2.1 Pointing Precision

Balloon-borne telescopes operate in the stratosphere, high above most of the Earth's atmosphere. While this altitude dramatically reduces atmospheric distortion and enables access to a broader spectrum, it introduces new challenges in precision pointing and stability. The gondola is a suspended, dynamic platform, subject to drift, pendulation, and vibration. Achieving high-resolution imaging requires that the telescope maintain arcsecond-level stability despite these disturbances. This demands a control system capable of processing sensor feedback and commanding actuators at high frequency with low latency. In particular, with gyroscopes operating at 1 kHz, the control loop must match or exceed this rate to provide timely corrections. Without sub-millisecond response times, the system risks accumulating drift, degrading image quality, and missing the mission's science requirements.

2.2 SuperBIT Architecture

The SuperBIT project demonstrated how balloon-borne telescopes could achieve arcsecond-level pointing by combining multi-stage control loops, on-board computing, and a carefully coordinated software platform (11). Its main goal was to maintain tight pointing accuracy in the harsh and dynamic conditions of stratospheric flight. This required real-time sensor fusion, robust actuation strategies, and precise timing across multiple processors.

Layered Control Architecture

1. **Coarse gimbals** (stepper motors) slewed the telescope to within arcminutes of target coordinates.
2. **Reaction wheels and pivot** countered gondola pendulation with continuous torque.
3. **Inner-frame DC motors** on torsional flexures removed residual disturbances at tens of hertz.
4. **Piezo tip–tilt mirror** applied rapid corrections (> 100 Hz) to hold sub-arcsecond stability on the focal plane.

Computing and Software Stack

A distributed PC/104+ network handled control, data, and telemetry:

- **Master Control Computer (MCC)** – QNX RTOS, 1 kHz gyro loop, mission sequencing.
- **Star-Camera Computer (SCC)** – lost-in-space solving and centroid tracking.
- **Inner-Frame Computer (IFC)** – telemetry logging and packet routing to the ground.

Sensor Fusion and Latency Mitigation

SuperBIT's pointing performance relied on integrating high-rate fiber-optic gyroscope data with intermittent updates from star cameras. An extended Kalman filter corrected for gyro drift using absolute orientation references derived from camera imagery. To mitigate latency introduced by the star cameras' processing time, the system employed predictive forecasting—or dead-reckoning—so that the control loop could estimate interim centroid positions between frames. This reduced the control system's sensitivity to star camera delays and preserved pointing accuracy in real time.

Telemetry and Ground Support

A custom UDP-based networking protocol packaged telemetry and commands into discrete packets, simplifying transmission and synchronization. On the gondola, the MCC and IFC assembled sensor data, motor states, and images for downlink. FIFO buffering absorbed data bursts—such as when star camera frames coincided with sensor updates—ensuring reliable delivery. Ground operators received this data in near real time and could issue high-level directives. The IFC would interpret these commands and route them to appropriate subsystems without interfering with the real-time loops.

2.3 Challenges with the Existing Design

Complexity of Custom Hardware SuperBIT relied on custom interface boards and data acquisition modules, each tailored to specific sensors. This created a dense wiring architecture that required careful integration of daughter boards for signal conditioning and power distribution. The result was a fragile and intricate system prone to calibration issues and failure modes such as loose cables or connector faults—especially under high-altitude environmental stresses.

QNX OS Constraints QNX enabled deterministic scheduling and real-time guarantees, but it also introduced friction due to its proprietary nature. Licensing restrictions and limited third-party driver support made it difficult to adopt newer libraries or hardware. This led to development slowdowns, reliance on in-house tools, and compatibility issues with emerging standards.

Aging Architecture The system architecture, initially developed over a decade ago, depended heavily on PC/104+ boards that are now difficult to source or upgrade. Meanwhile, modern embedded platforms offer far greater performance and power efficiency, as well as broader compatibility with modern software ecosystems. Emerging tools and languages—such as Rust, with its strong safety guarantees—open new possibilities for more reliable, maintainable, and scalable flight software. These developments motivate a complete modernization of the control system design.

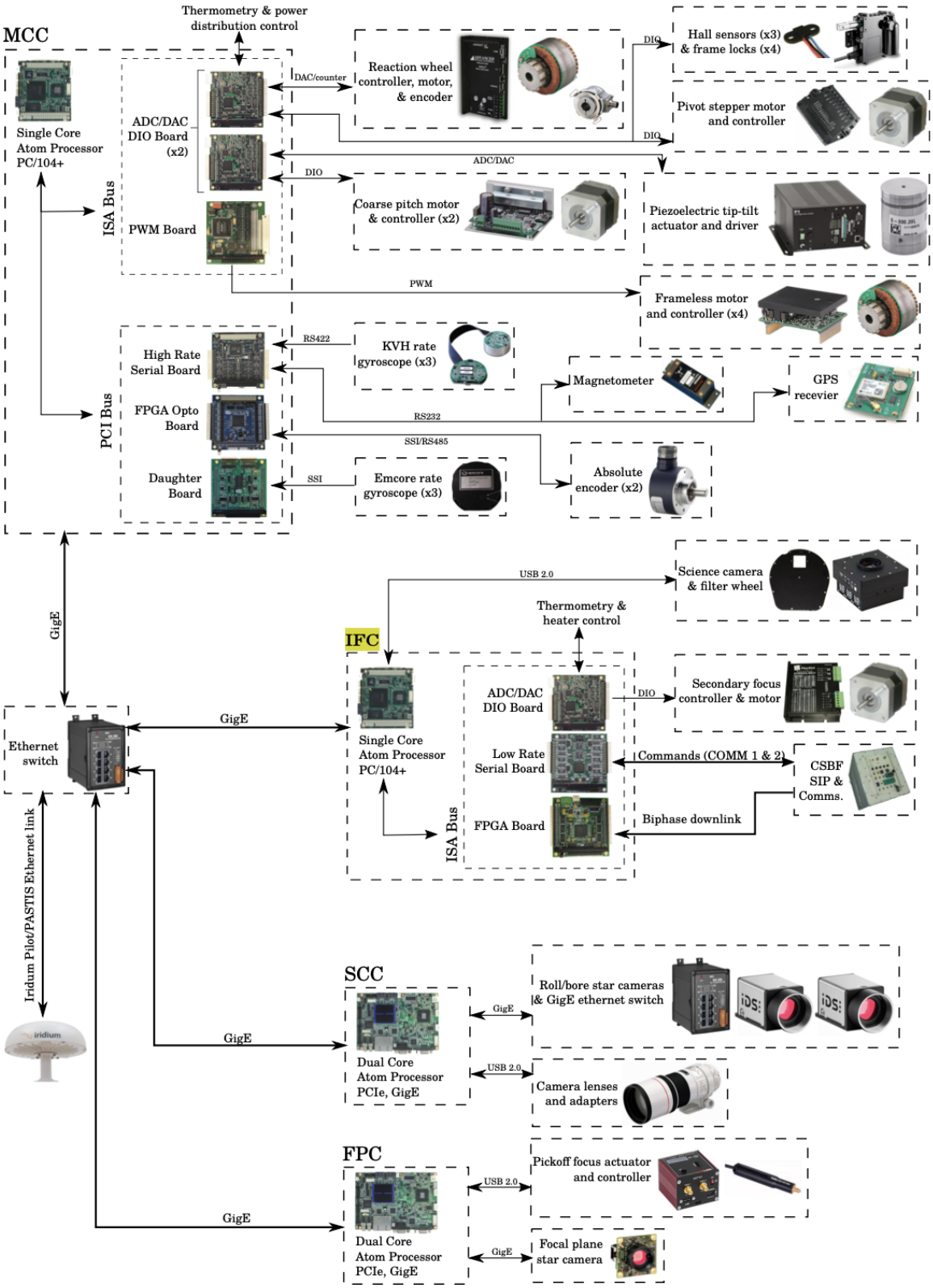

Figure 1: System architecture of the SuperBIT control system, featuring a centralized MCC connected to sensors, actuators, and subsystems via PCI and ISA buses. Ethernet links extend communication to additional processors for star camera and inner frame control.

2.4 Gigabit Upgrades and New Constraints

GigaBIT is the successor to SuperBIT and introduces significant upgrades across the board: a larger primary lens, higher-rate gyroscopes, and more demanding science goals. These improvements come with stricter requirements on pointing precision, now targeting sub-arcsecond RMS error over long exposures.

From system profiling conducted by the lab group, it became clear that the 1 kHz gyroscope rate now defines the bottleneck for the control loop. Matching this sensor rate requires that the entire motor control loop—including sensor readout, computation, and actuation—operate at sub-millisecond intervals. The existing SuperBIT system had a latency of roughly 1.6 ms, which was adequate for past missions but no longer sufficient.

In addition to tighter latency, the GigaBIT redesign is motivated by:

- improved sensors and optics that demand higher control performance,
- the need to modernize and maintain the codebase using open and extensible technologies, and
- a shift toward modular development, enabling reusable test libraries and hardware abstraction layers that support general-purpose flight control.

2.5 Problem Statement

The science goals of the GigaBIT telescope require a motor control system capable of sub-millisecond latency. This target is set to align with the 1 kHz sampling rate of the gyroscopes and to ensure real-time responsiveness to dynamic disturbances during flight.

However, reducing control latency is not simply a matter of replacing a few hardware components. The existing system is tightly integrated, with interdependent communication, scheduling, and processing layers. Introducing faster sensors or processors has cascading effects across the architecture, including bandwidth limits, thermal constraints, and software coordination. Therefore, the redesign must take a holistic view of the system.

This project focuses on designing a next-generation control system that:

- achieves $<$1 ms end-to-end latency from sensor input to motor actuation,
- enables modular hardware and software components for future reuse, and
- replaces legacy hardware and proprietary software with modern, open-source, and maintainable alternatives.

A complete reevaluation of the architecture is required to meet these constraints without compromising power efficiency, maintainability, or fault tolerance. Each subsystem—whether for sensing, computation, or actuation—must be evaluated and integrated in a way that supports the broader performance goals of GigaBIT.

3. Stakeholders

3.1 Primary Stakeholders

3.1.1 University of Toronto Balloon Astronomy Group

Barth Netterfield's Balloon Astronomy Group is the pioneering force behind using balloon-borne platforms to achieve near-space imaging quality at a fraction of the cost. Their extensive experience in high-altitude operations and commitment to overcoming atmospheric challenges is integral to the project's success. They are one of our two main points of contact, providing insights from past experiments and lessons learned.

3.1.2 Starspec

Starspec, led by Javier, is a start-up spun out of the Balloon Astronomy Group at the conclusion of the SuperBIT project. By bringing cutting-edge technology and agile engineering practices, Starspec collaborates with us to enhance system performance and reduce latency—directly supporting our goal of sub-millisecond motor control.

3.1.3 Capstone Project Team

The Capstone team members gain valuable hands-on experience by tackling a challenging, real-world engineering problem that bridges theory with practical application. The project offers an exceptional opportunity to hone skills in real-time control systems, hardware integration, and collaborative design while contributing to breakthrough research in balloon-borne astronomy.

3.2 Secondary Stakeholders

3.2.1 NASA

NASA serves as a critical regulatory and launch partner, providing guidelines and performance benchmarks that ensure our design meets rigorous aerospace standards. They will ultimately launch the balloon for which we design the control architecture.

3.2.2 Canadian Space Agency (CSA)

The CSA helps shape regulatory and operational standards for Canadian aerospace projects and is a primary source of funding for the GigaBIT program.

3.2.3 NSERC

NSERC is a principal funding agency supporting the innovative research that drives this project forward.

3.2.4 University of Toronto

The University of Toronto supplies laboratory facilities, resources, and academic expertise that form the foundation of our research and development. Their support fosters an environment where advanced engineering solutions can be explored and refined.

3.2.5 Other Astronomy Groups

Other astronomy research teams can apply the improved control methodologies and system architectures we develop to enhance their own observational platforms, elevating the overall quality and precision of balloon-borne telescopes across the community.

3.2.6 Computational Physicists

Computational physicists may adapt our distributed real-time control algorithms and sensor-fusion methods to other high-precision, data-intensive applications, broadening the impact of our work beyond astronomy.

4. Scoping the System

The overall goal of the team is to provide the Balloon Astronomy Group with a recommendation for a system which can control the various existing peripherals at the required latencies in order to achieve the required pointing precision for GigaBIT. Due to some existing constraints on the design space—which will be elaborated on in the next section—our design team comes into this project knowing that we most likely want some kind of central computer, either alone or supported by other computers and/or microcontrollers, to read the inputs of the various peripherals and output instructions to the various actuators via some kind of networking protocol.

Moreover, due to the success of SuperBIT, GigaBIT's predecessor, our team has access to a reference design which, although not perfect and although it does not meet the new required specifications for the upgraded telescope on GigaBIT, is known to have worked in the past and can therefore save a lot of time and testing effort if something similar can be made to work—both of which are important to our stakeholder. This basis for our design allowed us to begin by exploring our various options and led us to the conclusion that we wanted to test a selection of potentially viable components in order to make informed decisions on how best to implement the final design. By finding out the latencies and throughputs of these select candidate components and networking protocols, we hope to be able to later make an informed decision on how best to implement an overarching system which solves our design problem.

4.1 System Overview and Architectural Focus

To frame the subset of the system that our team is focusing on, we refer to the abstract control loop shown in Figure 2. This figure highlights the core logical stages involved in telescope attitude regulation: sensor data is acquired from gyroscopes and star cameras, passed

through an attitude estimation block, compared to a desired orientation (provided by user input), and then processed through a control algorithm which produces motor commands for coarse actuators.

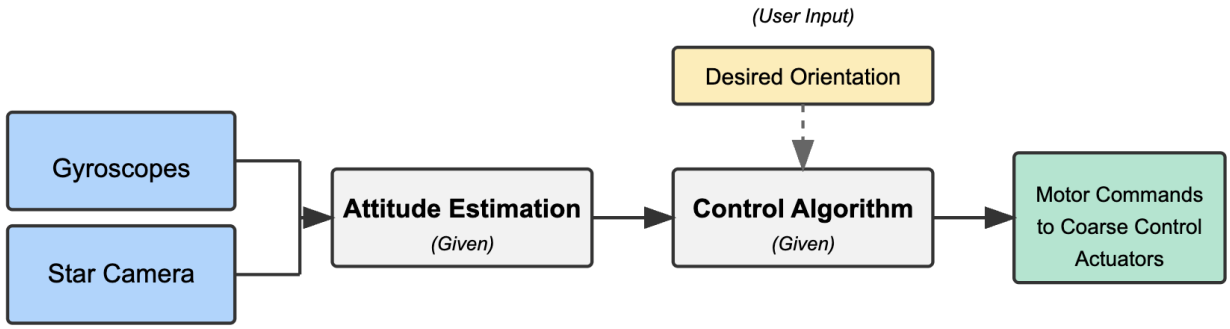

Figure 2: Abstracted control loop defining the scope of our system architecture.

Our responsibility is not to modify the estimation or control logic, which are assumed to be given, but rather to design the supporting architecture—both hardware and software—that enables these computations to run within strict real-time constraints. This includes selecting a central flight computer, identifying supporting processors (e.g., microcontrollers or co-processors), and defining how tasks such as sensor readout, data routing, computation, and actuation are communicated and performed across the system.

Specifically, our architectural focus includes:

- Selecting a suitable central computing unit and supporting processors (e.g., microcontrollers or co-processors).
- Defining how responsibilities are split between nodes, such as which processor handles sensor interfacing, control computation, and motor command generation.
- Designing data pathways and I/O interfaces that support low-latency, high-frequency data flow across the system.
- Choosing and integrating communication protocols (e.g., Ethernet, CAN, SPI) between sensors, processors, and actuators.

Although elements such as star camera firmware, estimation filters, or the control law are essential to the broader system, they are assumed to be provided and are not the focus of this project. Our task is to ensure that the surrounding system architecture can meet the timing, flexibility, and maintainability demands imposed by the GigaBIT mission requirements.

4.2 Deliverables and Project Contributions

Within the defined architectural scope, our team aims to develop a complete, testable system design that can reliably meet the sub-millisecond latency requirement for motor control. Rather than producing a balloon-ready embedded system, our deliverable is a well-validated

architectural prototype: a benchtop system that captures the essential performance characteristics of a flight-ready control loop.

This work is subdivided into focused components that reflect the interdependent nature of system design.

- **Design Exploration of Components:** Based on research and subsystem analysis, we will propose candidate components for diverse system architectures. These will specify how tasks such as sensor interfacing, data processing, and packet transport can be executed to meet timing constraints.
- **Performance Benchmarking:** We will evaluate the real-world latency and throughput characteristics of selected computing platforms, peripheral buses, and networking protocols. This includes microcontrollers, embedded computers, and various communication stacks. These benchmarks will be used to assess the feasibility of achieving ¡1 ms loop performance under representative conditions.
- **Reusable Test Infrastructure:** We will design and implement a general-purpose benchmarking framework, including timing and profiling utilities. These tools will enable future developers to replicate our results and assess the performance of additional components or system variants.

Our focus throughout is not on any single component but on delivering system-level insights that guide future design. This includes practical measurements of real-time performance under load—going beyond manufacturer specifications—and producing design guidance informed by actual test results. These contributions will be useful not just for the immediate control system, but for any future development that relies on low-latency embedded control.

4.3 Stakeholder Value and Impact

The outcome of this project directly supports several key stakeholders in the balloon astronomy ecosystem:

- **The Balloon Astronomy Group (GigaBIT Project)** will benefit from the final system architecture, benchmarking methodology, and end-to-end latency validation we provide. Our work supplies concrete insights into how a distributed control approach could be integrated into their telescope systems and helps derisk potential hardware or architectural transitions.
- **Starspec** will gain from the modular design approach and supporting software infrastructure we develop. The control libraries, test harnesses, and interface patterns we build can accelerate integration into new platforms or future experiments.
- **The broader academic astronomy community** will benefit from access to the open-source benchmarking tools and analysis libraries created during this project. These resources provide a transparent way to evaluate embedded systems and control architectures in real-world scenarios, and offer our empirical performance data that may differ from vendor-provided specifications.

By conducting a system-wide reassessment rather than relying on incremental upgrades, this project aims to position GigaBIT with a control infrastructure that is performant and maintainable.

5. Requirements Model

This section outlines the objectives, evaluation metrics, design constraints, and fixed system specifications governing the development of our balloon-borne control system. These requirements reflect both the performance expectations of the GigaBIT project and critical limitations set by environmental, hardware, and architectural decisions.

5.1 Objectives

The primary objective of this project is to design, implement, and rigorously test a data acquisition and control system for a balloon-borne astronomical telescope. The system will:

- Perform real-time computations essential for precise attitude determination and control (ADC) of both the telescope and its gondola.
- Accurately and reliably read and process data from hundreds of analog and digital sensors distributed across the gondola.
- Efficiently control several dozen motors and actuators dispersed across the gondola to maintain sub-millisecond latency for critical tasks.
- Ensure system robustness and functionality in the challenging environment of stratospheric balloon flights, including extreme temperatures ($-40°$C to $35°$C) and low pressures ($\sim$3 mBar).
- Optimize the system for a distributed, modular architecture, significantly minimizing cabling complexity, especially between moving frames of the payload.

5.2 Performance Metrics

The project's success will be evaluated using the following metrics:

Latency:

- Critical sensors and actuators must operate with latencies below 1 ms.
- Analog outputs must maintain a maximum latency of 2.5 ms at update rates up to 500 Hz.
- Digital encoders and analog inputs requiring rapid updates (100 Hz) must remain under 5 ms latency.

Environmental Reliability:

- Demonstrated operational stability in controlled tests replicating stratospheric conditions ($-40°$C to $35°$C, 3 mBar).

Power Efficiency:

- Total system power consumption (excluding motors and heaters) must remain below 100W, with a preference for minimizing power use to ease thermal management challenges.

Communication Speed:

- Achieve Ethernet/Gigabit Ethernet latencies under 0.5 ms.
- Maintain a communication latency for tracking camera packets (50 bytes) below 1 ms, ensuring frame rates between 24 Hz (USB3) and 34 Hz (5GigE) are achievable.

5.3 Design Constraints

The project must comply with several critical constraints:

Environmental Constraints:

- No fan-cooled systems due to ineffectiveness at low pressure.
- All cables must remain flexible and reliable at temperatures as low as $-60°$C.

Real-Time Performance Constraints:

- Maintain sub-ms latency on all critical sensor and actuator control loops.
- Motor PWM control must reliably operate at frequencies of 10 kHz with a maximum latency of 1 ms.

Power Constraints:

- Limit the total electrical consumption of the control system to 100W or less.
- Investigate the use of Power-over-Ethernet (PoE) as a potential power distribution strategy to minimize cabling.

Communication and Interface Constraints:

- Ethernet (GigE, 5GigE) is mandatory for camera interfacing due to vendor driver support on Linux/AMD64 systems.
- Minimize cable length and quantity, emphasizing distributed architecture over Ethernet.
- Use of CAN, SSI, RS422, RS232, USB2/USB3 interfaces based on sensor or actuator latency requirements (e.g., RS422 latency must remain ≤ 0.5 ms, RS232 ≤ 10 ms).

Software Development Constraints:

- The preferred programming language is Rust due to reliability and maintainability, though C is acceptable where Rust compatibility is not feasible.
- Avoid languages with extensive runtime environments (e.g., Python, Java).

5.4 Low-Level Input and Output Requirements

Low-Level Inputs

Table 1 summarizes the low-level sensor inputs required by the system, including analog and digital signals used in control and telemetry loops.

Table 1: Low-level input requirements

Type	**Number**	**Rate**	**Latency to Compute Loop**
Analog Input (12 bit)	100	5 Hz	–
Analog Input (16 bit)	10	100 Hz	5 ms + filter lag
Digital Level Inputs	10	5 Hz	–
A/B/Z Incremental Encoder (1 μs/pulse)	10	100 Hz	5 ms

Low-Level Outputs

Table 2 summarizes the low-level output devices and actuators that must be controlled by the system, along with their required update rates and latency constraints.

Table 2: Low-level output requirements

Type	**Number**	**Current**	**Update Rate**	**Latency from Compute Loop**
Analog Output (16 bit)	5	–	500 Hz	2.5 ms
Digital Level Outputs	50	60 mA or 300 mW open collector	5 Hz	–
Heater PWM	100	2 A @ 40–58.4 V	8-bit at 1 Hz min	–
Motor PWM (10 bit @ 10 kHz)	10	5 mA	1000 Hz	1 ms
Pulse/Direction Stepper Control	20	5 mA	100 kHz pulse rate, 100 Hz update	5 ms

I/O Buses

Table 3 provides a list of all communication interfaces that must be supported by the system, with expected update frequencies and allowable latencies.

Table 3: Other I/O bus requirements

Bus Type	**Number**	**Update Rate**	**Latency from Compute Loop**
CAN	5	5 Hz	–
SSI	10	1000 Hz	1 ms
RS232 (UART)	20	115 kbps	10 ms
RS422 (UART)	10	1 Mbps	0.5 ms
USB3	10	–	0.5 ms
USB2+	20	–	10 ms
Ethernet (internal)	As needed	–	–
GigEthernet	8	–	0.5 ms
Ethernet (telemetry)	4	–	–
SSD Interface	For 10 TB storage	Minimum USB3	–

6. Design Exploration

The goal of this design–exploration phase is to translate the high-level performance and environmental requirements of the GigaBIT gondola into a coherent set of hardware choices. In practical terms the task is to identify and characterize

1. **Main flight computers**,
2. **Family of microcontrollers**,
3. **Motor-control interfaces**, and
4. **Deterministic network / bus fabric**,

Because many electro-optical and mechanical subsystems have already been frozen by earlier SuperBIT flights, the new electronics must interface with an extensive legacy I/O footprint and must do so with minimal re-engineering of proprietary drivers originally written for x86 Linux.

Our strategy is therefore to cast a wide net over commercially available, industrial-grade components, evaluate each candidate against a ranked list of criteria—processing head-room, deterministic timing, operating-temperature range, fanless cooling, and breadth of I/O—and then converge on the smallest set that collectively meets or exceeds every requirement. Where multiple candidates appear viable, we deliberately favour the highest-performance option that fits inside the thermal and power envelopes; overspecification at this stage is inexpensive compared with the cost of late design churn or in-flight failure. The following subsections document this process, starting with the selection of the main flight computer, proceeding through microcontroller and motor-driver choices, and concluding with the bus and network protocols that tie the distributed architecture together.

6.1 Main Computer Selection

When selecting the main computer for the GigaBIT system, we evaluated several key requirements based on the specific operational conditions of stratospheric balloon missions:

- **x86 Architecture** (Very strongly preferred): Some existing I/O components ship with proprietary drivers which only run on x86 systems. Since they are proprietary, we do not have access to the code to change them, so using any other architecture would require redesigning the drivers from scratch. The team has deemed this highly difficult and impractical, although not impossible.
- **Real-Time Linux Support** (Preferred): If the required latencies can be achieved without a real-time operating system, this may not be necessary. However, it is currently unknown whether such latencies can be reliably obtained without a real-time OS. Linux is preferred because certain drivers are designed specifically for x86 Linux.
- **Temperature Resistance** (Preferred): The balloon does have integrated heating systems to maintain electronic components at appropriate temperatures. At steady state near the top of the stratosphere, solar panels work at full capacity during daylight hours while ambient temperatures are around -15°C, which combined with integrated heaters should maintain operational temperatures. Despite Prof. Netterfield's lab never experiencing computer failures due to temperature issues, temperature resistance remains a concern because the balloon must pass through the coldest parts of the stratosphere (around -50°C) while ascending to the destination altitude.
- **Fanless Design** (Required): The computer cannot rely on fans for cooling since there is very little air in the upper stratosphere. Alternative cooling methods must be employed.
- **Processing Power** (Very strongly preferred): The previous SuperBIT system with older generation technology could not achieve the low latencies required for the higher precision desired in the GigaBIT system's larger telescope lens. It would be extremely difficult to achieve lower latencies with a slow computer, regardless of optimizations elsewhere in the system.
- **Extensive I/O Connectivity** (Strongly preferred): The current design involves a widespread distributed system requiring numerous connections. A computer with limited I/O would necessitate significant changes to the rest of the system, such as condensing I/O or implementing multiple kernels. Since much of the balloon has already been designed, reducing I/O requirements is not feasible, though some functions could potentially be offloaded to microcontrollers.

Based on these requirements, we initially investigated five categories of computing solutions:

- Industrial Single Board Computers (SBCs)
- COM Express Modules

- High-Performance Mini PCs
- Next Unit of Computing (NUC)
- PXI and VPX-based Embedded Systems

After thorough research into these categories, we converged on Industrial Single Board Computers as the optimal solution. PXI and VPX Embedded Systems proved to be more expensive and relied on legacy technology. NUC and High-Performance Mini PCs met our fanless, x86, and performance requirements but lacked sufficient I/O options and temperature resistance for our application. COM Express Modules offered good I/O flexibility but compromised on temperature resistance and high-performance capabilities. For these reasons, Industrial SBCs emerged as the most suitable choice for our main computer.

We then examined offerings from leading Industrial SBC manufacturers:

- Advantech (provider of previous SuperBIT computer)
- Kontron
- OnLogic
- Congatec
- AAEON

SBC Model	**Processor**	**Features**	**Suitability for GigaBIT**
Advantech AIMB-285	Intel 11th Gen i5/i7	Multiple PCIe, industrial-grade	High-performance and expandable
AAEON UP Xtreme i11	Intel i5/i7 11th Gen	AI acceleration, ruggedized	Small, low-power, high computing
Kontron 3.5"-SBC-VR1000	AMD Ryzen V1000	High GPU performance, multiple I/O	Good for sensor-heavy applications
OnLogic Karbon 801	Intel i9 24-core	Fanless, ruggedized	Good for extreme environments + 6xGbE
Congatec conga-JC370	Intel i5/i7	Thin Mini-ITX, long lifecycle	Reliable for aerospace applications
IEI WAFER-TGL	Intel Tiger Lake	Compact, AI-ready	Efficient, real-time performance

Table 4: Comparison of Industrial SBC Options

From this list, many computers were eliminated due to insufficient temperature resistance, as they could only operate down to 0°C or -20°C, which we deemed inadequate for our application. Our previous reference was the Advantech ARK-1250L used in the SuperBIT launch, which had proven reliable and compatible with most of the required I/O. Initially,

we preferred to continue with a newer Advantech model, but discovered that their recent offerings were not as rugged or temperature-resistant as needed.

Some additional candidates were ruled out due to insufficient performance. Since the Advantech ARK-1250L (equipped with an Intel Core i5) had previously been determined to be too slow for our application, we knew that any computer with less recent processors or less RAM would likely underperform. Given that most of these computers cost several thousand dollars, our budget limited us to selecting only one for initial latency testing.

From the remaining candidates, we selected the computer with the most processing power, memory, and I/O ports while still maintaining a temperature rating down to -40°C (matching the proven ARK-1250L). We opted for maximum capabilities to establish our performance ceiling, reasoning that it would be better to know our lowest possible latency. If we could exceed our requirements, this would benefit future balloon designs. While our capstone team's testing budget didn't permit purchasing multiple computers, the cost of a high-performance computer remains insignificant compared to the telescope lens, making overspecification less concerning for multiple balloon launches.

We ultimately selected the OnLogic Karbon 801 Low-Profile High-Performance Rugged Computer, configured with:

- 24-core 5.2GHz Intel i9 processor
- 32GB RAM
- Operating temperature range: -40°C to 70°C

Front I/O	4× USB 3.2 Gen 2 ports 2× COM RS-232/422/485 ports 1× GPIO Terminal block (DIO, CAN, Ext. Switch) 2× 3FF Micro-SIM 1× 3.5mm audio 1× Power button 1× External fan connector
Rear I/O	2× or 6× 2.5 GbE LAN (2× PoE optional) 2× USB 3.2 Gen 2 ports 2× DisplayPort 5-Pin Terminal Block Power Input (12-48 VDC)

Table 5: OnLogic Karbon 801 I/O Specifications

6.2 Microcontroller Selection

Alongside the main computer, we needed to select appropriate microcontrollers to interface with various sensors and actuators throughout the system, including cameras, motor drivers, and encoders. Our stakeholders provided specific requirements regarding the type, quantity, and latency of I/O peripherals needed.

For this initial stage of the project, our goal was to identify and test a range of microcontrollers to determine which would best meet our requirements. Since we had flexibility in choosing communication buses and network protocols, we decided to test microcontrollers supporting various options to compare latencies and select the optimal configuration for each application within the system.

A critical consideration in our microcontroller selection was performance improvement over the previous design. Without faster microcontrollers, we would be unable to reduce latency in this area, making it difficult to justify deviating from the previous design. This led us to focus on high-performance microcontrollers, which our research narrowed to two main options:

- **STM32 microcontrollers**: Offering up to 480MHz clock speeds with excellent Rust support libraries
- **Teensy microcontrollers**: Providing higher speeds (up to 600MHz) but with limited Rust support

The STM32 family stands out as one of the highest-performing non-industrial COTS microcontrollers available, with thousands of options and extensive support for Rust programming. Since Rust support was emphasized as a key requirement by our stakeholders, this gave STM32 a significant advantage despite its slightly lower maximum speed compared to Teensy.

GigE / USB3 Connectivity Only a small subset of STM32 microcontrollers offer Gigabit Ethernet (GigE) capabilities. After investigating the available options, we found only four models with GigE support in stock, and only one made practical sense for our project. The other three were considerably more expensive due to additional features like hardware AI acceleration that weren't necessary for our application. We therefore converged on the **STM32MP157D-DK1**, the only available STM32 microcontroller offering GigE connectivity without unnecessary features.

Our research revealed that no standard microcontrollers, including the entire STM32 family, offer USB3.0 support. However, this limitation didn't impact our project since we only needed either GigE or USB3.0 for communicating with the Star Camera, not both simultaneously.

DIO/Analog and PWM Support Another requirement was for a microcontroller with numerous digital I/O pins and 12-bit precision analog inputs that could also generate PWM signals. These features are common in microcontrollers, with dozens of suitable options in the STM32 family alone. The NUCLEO STM32 development boards are among the most widely used and supported options, with abundant tutorials and example code repositories.

Considering availability and delivery time from suppliers like Digikey, we selected the **NUCLEO-H723ZG** board. This choice was influenced by its widespread adoption, comprehensive documentation, and immediate availability.

CAN and SSI Communication Controller Area Network (CAN) is a robust networking protocol, while Synchronous Serial Interface (SSI) is a serial communication protocol. We needed to identify boards with hardware support for these protocols, as shown in the comparison table below:

Board	CAN	SSI	UART	Price
NUCLEO-F756ZG	1	42	10.5	24.08
Nucleo-H753ZI	8	150	15	28.15
STM32F746G Discovery	1	42	10.5	
NUCLEO-G474RE	8	32	10.5	15.88
Nucleo-L476RG	8	10	3	14.85
Nucleo-F303RE	8	24	3	16.75
Nucleo-F401RE	8	42	10.5	20.96
B-L475E-IOT01A	1	10	3	
STM32F3DISCOVERY	1	24	3	25.27
Waveshare STM32H743I	8	150	15	
Waveshare CoreH743I	8	150	15	

Table 6: Microcontroller Board Comparison for CAN/SSI Communication

Fortunately, the NUCLEO-H723ZG board previously selected for DIO and PWM applications also offers excellent CAN and SSI capabilities, making it suitable for testing all our required communication protocols.

Teensy Boards To ensure comprehensive evaluation of all options, we also acquired several Teensy 4.1 boards. These represent the only commercially available microcontrollers faster than the STM32 family we were testing. While our preference was for the STM32 platform due to its superior Rust support (making it easier to program for both our team and future system integrators), testing the Teensy boards provided a performance benchmark and alternative option if the STM32's speed proved insufficient.

6.3 Motor Control Interfaces

Motor controllers form a critical interface between the motor actuation layer and the microprocessors in our system. Each motor includes a Hall Effect sensor for position feedback, requiring the motor controller to decode this position data and communicate it to the microprocessor.

Several models support this functionality, and we selected the following candidates for evaluation:

- Nucleo Shield for STM32
- AEK-MOT-2DC70S1 Tri-Motor Controller
- BLDC-Shield for STM32

These were chosen based on several criteria:

- Cost under $100 per unit (budget constraint set by the PI)
- Support for multiple motors (minimum of two required)
- Compatibility with our selected microcontrollers
- Potential to provide sufficient latency margin for our overall 1ms latency requirement

These motor controllers will be evaluated for latency, amongst the other metrics listed in the requirements model. The controller should provide sufficient latency margin to meet the overall 1ms latency requirement.

6.4 Networking and Bus Protocols

Our distributed architecture requires efficient communication between components. The selection of networking and bus protocols was guided by our microcontroller capabilities and system latency requirements. The primary protocols under consideration include:

- **Gigabit Ethernet (GigE)**: Selected for high-bandwidth communication with the Star Camera, this protocol offers the throughput needed for image data transfer but requires specific hardware support limited to certain microcontrollers.
- **Controller Area Network (CAN)**: Known for its robustness in automotive and industrial applications, CAN provides deterministic communication between control nodes with built-in error detection and prioritization features. Our selected NUCLEO-H723ZG supports up to 8 CAN interfaces.
- **Synchronous Serial Interface (SSI)**: This reliable protocol for sensor data acquisition allows for precise timing control and is well-supported on our selected microcontroller with up to 150 SSI-capable pins.
- **UART**: For simpler point-to-point connections where the higher complexity of other protocols isn't needed, UART provides a straightforward communication option also supported by our microcontrollers.

The versatility of our selected NUCLEO-H723ZG microcontroller in supporting all these protocols allows us to perform comprehensive testing to determine the optimal configuration for different subsystems within our distributed control architecture. This testing will focus on achieving the balance of reliability, determinism, and low latency required for the GigaBIT system's demanding performance specifications.

7. Prototype Tests

To ensure the selected hardware can meet the sub-millisecond latency requirements of the GigaBIT control system, we conducted a focused prototyping campaign to measure the timing performance of each critical subsystem—compute, sensing, actuation, and network communication—under controlled bench conditions. All subsystems were synchronized to a common GPS-disciplined pulse-per-second (PPS) timebase, and cycle-accurate latency measurements were captured using hardware counters and packet timestamps. Each test was repeated across more than 1,000 trials to characterize mean, median, and worst-case (99.9-percentile) behavior, providing statistically robust validation against system-level timing budgets.

The prototyping tests were organized into four main blocks: microcontroller computational load testing, motor control and interrupt latency evaluation, sensor filter processing assessment, and Gigabit Ethernet communication timing. Each block was designed to isolate and stress a key portion of the control loop under realistic conditions, with acceptance thresholds set to ensure sufficient margin for environmental deratings in flight. Together, these results validate the feasibility of the distributed control architecture and quantify timing slack available for integration.

7.1 Microcontroller Computational Performance Tests

7.1.1 Test Setup and Methodology

In our distributed control architecture, a microcontroller will be used to execute the required attitude determination algorithms that process sensor data and generate motor control commands. For this system to meet our sub-millisecond latency requirements, it is essential to select a microcontroller with sufficient computational performance to handle these time-critical operations.

Rather than implementing the entire control algorithm on each candidate microcontroller, which would be time-consuming and unnecessary, we worked with the Balloon Astronomy Group to characterize the computational workload through assembly code analysis for their existing algorithms. The core quaternion operations were broken down into their fundamental arithmetic operations as shown in Table 7.

Table 7: Quaternion Algorithm Computational Requirements

Operation Phase	**Multiplications**	**Additions**	**Transcendental Operations**
Gyroscope data integration	700	500	6
Star camera update	850	600	4
Total	**1,550**	**1,100**	**10**

To evaluate microcontroller performance with this computational load, we developed custom benchmarks for the STM32H723ZG (550 MHz) and the Teensy 4.1 (600 MHz). The benchmarks were designed to model the quaternion operations under three distinct execution patterns, each representing different degrees of optimization:

1. **Sequential execution**: Performing all multiplications first, followed by all additions, and finally all transcendental operations. This represents the most pessimistic case where operations are strictly serialized. When operations are grouped this way, the processor cannot utilize its separate functional units simultaneously, reducing potential throughput.

2. **Interleaved execution**: Mixing different operation types throughout the algorithm. This allows the processor to leverage its superscalar pipeline by executing different operations (additions, multiplications) in parallel on separate functional units, improving computational efficiency.

3. **Parallel execution**: This most realistic scenario introduces independent operations with separate write addresses, allowing for instruction-level parallelism. This most closely approximates the actual quaternion algorithm where not every operation depends on the immediate previous result.

The most realistic test pattern (parallel execution) emulates how quaternion operations would be implemented in practice. As shown in Figure 1, this approach uses multiple accumulators to enable instruction-level parallelism, distributing the workload in a way that allows the processor to perform operations simultaneously.

For all three patterns, we tested both 32-bit (float) and 64-bit (double) precision floating-point operations, as both may be relevant to the pointing precision requirements of the GigaBIT system.

To ensure accurate timing measurements, we utilized the ARM Cortex-M Debug Watch and Trace (DWT) cycle counter, a hardware feature that provides cycle-accurate timing with nanosecond precision. This approach eliminated the overhead and variability associated with software-based timers, giving us highly reliable performance metrics for comparison.

7.1.2 Results and Analysis

The results of our microcontroller benchmark tests are presented in Tables 8, 9, and 10.

Table 8: STM32H723ZG Computation Time (μs)

Execution Pattern	**32-bit Float**	**64-bit Double**
Pattern 1 (Sequential)	300	1,400
Pattern 2 (Interleaved)	200	1,500

Table 9: Teensy 4.1 Computation Time - Patterns 1 & 2 (μs)

Execution Pattern	**32-bit Float**	**64-bit Double**
Pattern 1 (Sequential)	14.82	27.37
Pattern 2 (Interleaved)	20.07	32.54

Analysis of these results reveals several key insights:

```
// Multiple independent accumulators enable instruction-level
   parallelism
double accumulator1 = 1.0, accumulator2 = 1.1, accumulator3 =
   1.2;

// Distribute multiplications across accumulators
for (int i = 0; i < totalMults/3; ++i) {
    accumulator1 *= MULTIPLY_CONSTANT;
    accumulator2 *= MULTIPLY_CONSTANT;
    accumulator3 *= MULTIPLY_CONSTANT;
}

// Distribute additions across accumulators
for (int i = 0; i < totalAdds/3; ++i) {
    accumulator1 += ADD_CONSTANT;
    accumulator2 += ADD_CONSTANT;
    accumulator3 += ADD_CONSTANT;
}

// Distribute transcendental operations across accumulators
for (int i = 0; i < totalTranscendentals/3; ++i) {
    accumulator1 = sin(accumulator1);
    accumulator2 = sin(accumulator2);
    accumulator3 = sin(accumulator3);
}

// Combine results (stored in volatile to prevent optimization)
result = accumulator1 + accumulator2 + accumulator3;
```

Listing 1: Parallel execution benchmark with multiple independent accumulators

Table 10: Teensy 4.1 Computation Time - Pattern 3 (μs)

Execution Pattern	**32-bit Float**	**64-bit Double**
Pattern 3 (Parallel) - Sequential	6.41	20.80
Pattern 3 (Parallel) - Interleaved	5.67	19.13

```
// Enable ARM Cortex-M cycle counter
ARM_DEMCR |= ARM_DEMCR_TRCENA;
ARM_DWT_CTRL |= ARM_DWT_CTRL_CYCCNTENA;

// Function to read cycle counter register
static inline uint32_t getCycles() {
    return ARM_DWT_CYCCNT;
}

// Benchmark timing example
uint32_t startCycles = getCycles();
runQuaternionBenchmark(numMults, numAdds, numTranscendentals);
uint32_t elapsedCycles = getCycles() - startCycles;

// Convert to microseconds (F_CPU = 600MHz on Teensy 4.1)
float elapsedMicros = elapsedCycles / (F_CPU / 1e6);
```

Listing 2: Cycle-accurate timing implementation for Teensy

1. **Precision Impact**: The STM32H723ZG shows a 4.7x slowdown when moving from 32-bit to 64-bit operations, whereas the Teensy 4.1 exhibits only a 1.85x slowdown. This indicates that the Teensy has more efficient double-precision floating-point units.

2. **STM32 Performance**: The STM32H723ZG requires approximately 1.4-1.5 milliseconds to complete the double-precision quaternion operations using patterns 1 and 2. This exceeds our 1-millisecond end-to-end latency budget before even considering sensor readout or motor actuation time, making it unsuitable for our control loop.

3. **Teensy Performance**: The Teensy 4.1 completes the same double-precision workload in 27.37-32.54 microseconds using patterns 1 and 2, and 19.13-20.80 microseconds with pattern 3. This represents a 50-70x performance advantage over the STM32 for our specific workload.

4. **Execution Pattern Effects**: On the Teensy, the more realistic parallel execution pattern improved performance by 41-43% for double-precision calculations compared to the sequential pattern, demonstrating that it is able to take advantage of the processor's pipeline and separate functional units for adds/multiplies.

7.1.3 Microcontroller Selection for Processing Outputs to Motor Controller

Based on our benchmark results, we can make informed decisions about which microcontroller platform is most suitable for the critical quaternion processing tasks in our distributed architecture:

1. **Performance Comparison**: The Teensy 4.1 demonstrates much better performance for our computational workload, completing double-precision calculations approximately 50-70x faster than the STM32H723ZG. With execution times of $\sim$20-33 μs,

the Teensy uses only about 2-3% of our 1 ms latency budget for quaternion operations, providing substantial margin for other control loop components.

2. **Precision Trade-offs**: While double-precision operations are preferred for final pointing accuracy, our results indicate that single-precision operations on the Teensy are approximately 3.4x faster. The STM32 can perform the operations in an acceptable amount of time, potentially making it viable for certain control paths if single-precision calculations would meet the pointing accuracy requirements. However, the STM32's performance is still significantly slower than the Teensy.

3. **Development Experience and Language Support**: Through developing these benchmarks, we gained practical experience with both platforms. The STM32 platform offers excellent Rust support through the stm32-rs ecosystem (12), aligning with our client's programming language preference. In contrast, the Teensy currently has limited Rust support but provides a very good development environment through Arduino tools, making it easy to program in C/C++. Since the client indicated C would be acceptable if Rust isn't feasible, the Teensy's development simplicity is a significant advantage that offsets its more limited language support.

Given these considerations, the **Teensy 4.1** is the better option for the quaternion processing to compute updated control signals to the motors. Its performance allows for significant timing margin that can be used in other parts of the control loop. While the STM32's superior Rust support is valuable, the order-of-magnitude performance difference makes the Teensy the clear choice for time-critical computational tasks, with an acceptable bare-metal C programming environment.

7.2 Ethernet Latency Testing

One of the most promising candidates for controller-to-controller and controller-to-computer communication was the use of Gigabit Ethernet protocol as a means to transmit large amounts of data very rapidly. It was desired to obtain a benchmark for how fast this transmission is so that it could be confirmed whether or not GigE was a viable transmission option.

7.2.1 Test Setup and Methodology

Two STM32 development kit boards with Ethernet capability, specifically model STM32MP157D-DK1, were connected together via a Gigabit Ethernet connection and packets were sent between them to evaluate the speed. Although there exist many different board options with GigE capability, they cost $>$ \$100 each meaning that the budget did not have room to test multiple pairs of boards. However, these boards all have similar Ethernet capability and it is the other accompanying features which vary from board to board, so it was determined useful to benchmark the speed for any of the viable boards, and if the exact model is changed later it should accomplish similar speeds due to the specs of the Ethernet port in particular being standardized and therefore remaining the same across boards.

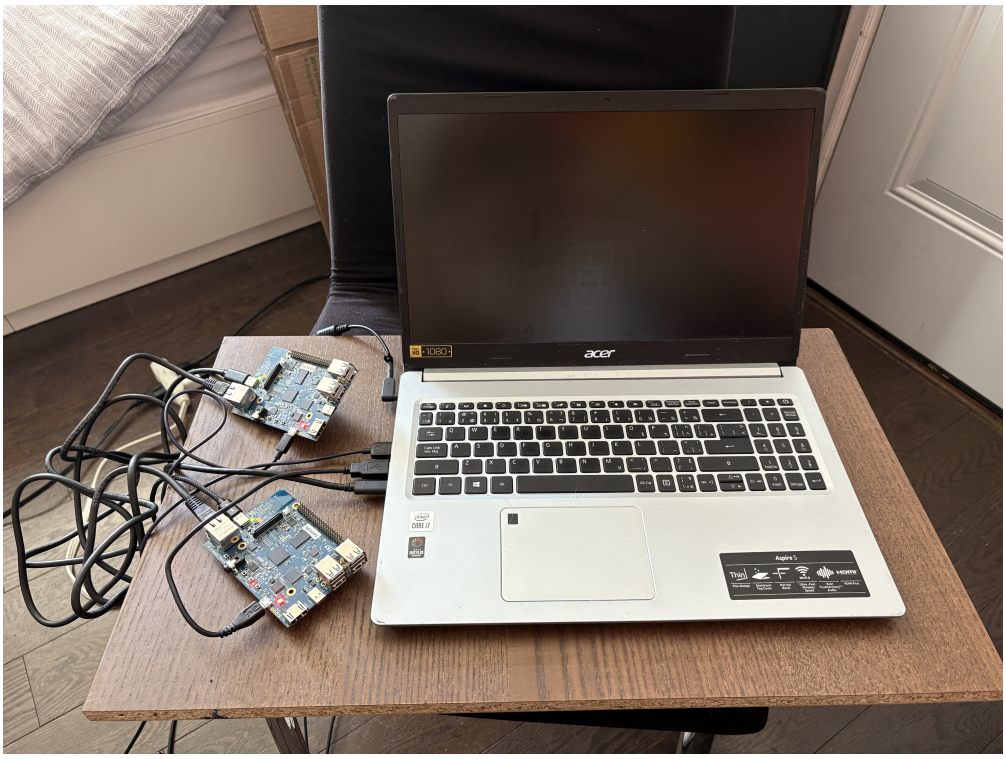

Figure 3: The Ethernet latency testing setup (front view).

To perform the Ethernet test, a cross-compiled C program employing UDP sockets was transferred from the laptop to each of the 2 STM32 boards, with 1 acting as the receiver and 1 acting as the sender. An Ethernet connection was set up between the two boards using innate Linux commands such as *ifconfig*. The Linux could be operated directly from the laptop via a serial PuTTY connection communicatioin over Micro-USB. This connection was also used to launch the C program and observe the results.

7.2.2 Avenues Explored

The final decision to cross-compile C code on an external CPU and then transfer a compiled version over Ethernet to to the STM32 micro-controller was only decided after attempting some of the various other potential options.

When the STM32 was received from shipment it contained a pre-loaded Linux distribution which is the Starter Package[1]. Using this distribution would be ideal because it does not require flashing a new Linux onto an SD card, but the starter package is missing some important Linux applications such as opkg, gcc, and netcat. Specifically, missing Netcat meant that UDP packets could not be sent over Ethernet directly using Linux commands. Moreover, no GCC meant that C code code not be compiled directly on the board, and no opkg meant that the two aforementioned applications could not simply be installed through the internet.

One of the first workarounds attempted was to use Python to compile a basic UDP test

[1] https://wiki.st.com/stm32mpu/wiki/STM32MP13_Discovery_kits_-_Starter_Package

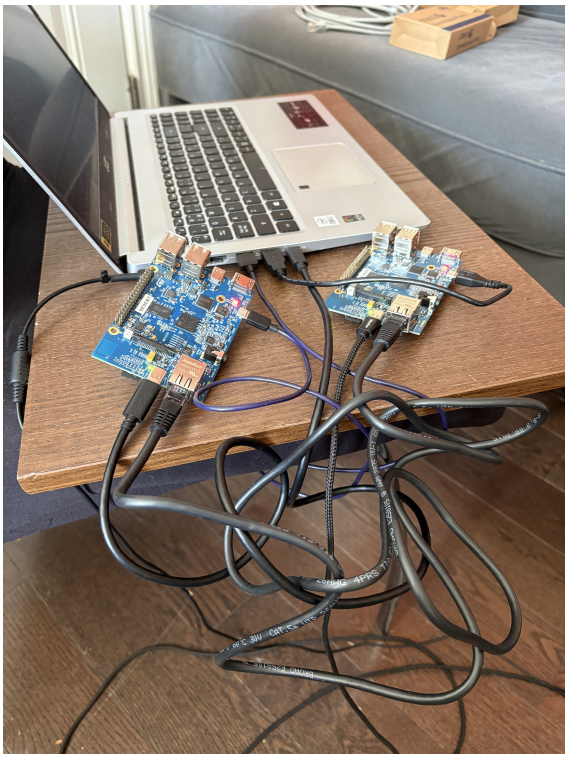

Figure 4: The Ethernet latency testing setup (side view).

locally on the STM32. The exact code can be found here. However, the issue with this method is that Python has a high overhead which makes it slow compared to lower-level programming languages such as C or Rust. Table 12 shows that the average round-trip time is 1764us, which uses almost the entire allowed millisecond each way and would not leave almost any time for calculations or sensor reading if the desired 1 millisecond total one-way latency is to be achieved. As such, the numbers clearly showed that Python was not a viable solution for receiving and sending Ethernet data within GigaBIT. The 7 tests run are not enough to be able to directly compare this data to the C performance (Table 13), but since the best-case Python timing exceeds the worst-case C timing (with task priority), it was deemed that more tests would not be able to produce a better result and this solution was ruled out.

The initial desire was to attempt to obtain the gcc C compiler onto the STM32 so that this lower-level language could be used instead. The hope was that less overhead would result in significantly faster packet transmission times. One promising method of doing this was to compile a more powerful STM32 Linux version in the Distribution Package[2]. Although no specific documentation could be found outlining the exact specifications of this Linux system, the fact that it is intended to be the most powerful one available appeared promising enough to give it an attempt. After many attempts and ¿9 hours of compilation time this Linux image was finally compiled only to discover that it did not contain any more useful applications than the original Starter Package (none of the applications outlined in Table 11 were affected). It appears that the Distribution Package provides a way to edit

[2] https://wiki.st.com/stm32mpu/wiki/STM32MPU_Distribution_Package

Application	Status
`ip`	✓
`ifconfig`	✓
`ping`	✓
`nc`	✓
`ethtool`	✓
`wget`	✓
`curl`	✓
`ssh`	✓
`netcat`	✗
`opkg`	✗
`gcc`	✗

Table 11: Presence of key networking and utility applications in the STM32 root filesystem.

	Mean (us)	Min (us)	Median (us)	Max (us)
Python Round Trip	1763.78	1637	1805	1859

Table 12: Round-trip packet latencies measured from 7 Python tests.

the Linux Kernel before compiling it but this is outside of our area of expertise and so it was deemed that this was not a reasonable solution which could be accomplished within the available time to complete the Capstone project. The full notes on what was attempted are visible in Appendix A.

The final attempted approach which was attempted and ended up succeeding involved the use of a Software Development Kit (SDK) to cross-compile C programs on an x86 computer but targeted towards the STM32's ARM architecture so that the compiled file could then be transferred and run on the micro-controller. This approach was not originally preferred because the need to transfer files before running them made the development cycle more complex as the time between making changes to the code and being able to run it and test it increased significantly. However this still remains a valid approach because once GigaBIT is in the air it is running compiled code and no changes can be made regardless. One advantage of such a cross-compiler approach is that the full power of modern IDE's and code editors, potentially even including AI-assisted programming, can be used to write the code since it is now done on a computer. There likely exists a way to use this SDK for Rust compilation but time did not permit for testing of anything except straightforward C programming, which did not require any changes to the existing SDK. The SDK used is contained within STM32's own Developer Package[3].

Once cross-compiled there were two possibilities for transferring the compiled binaries to the micro-controller. The first involved reading the SD card from the laptop, mounting the Linux file system, and copying it to the right location in the SD card. This method was not chosen because an SD card reader was not available at the time. The alternative method

[3] https://wiki.st.com/stm32mpu/wiki/STM32MPU_Developer_Package

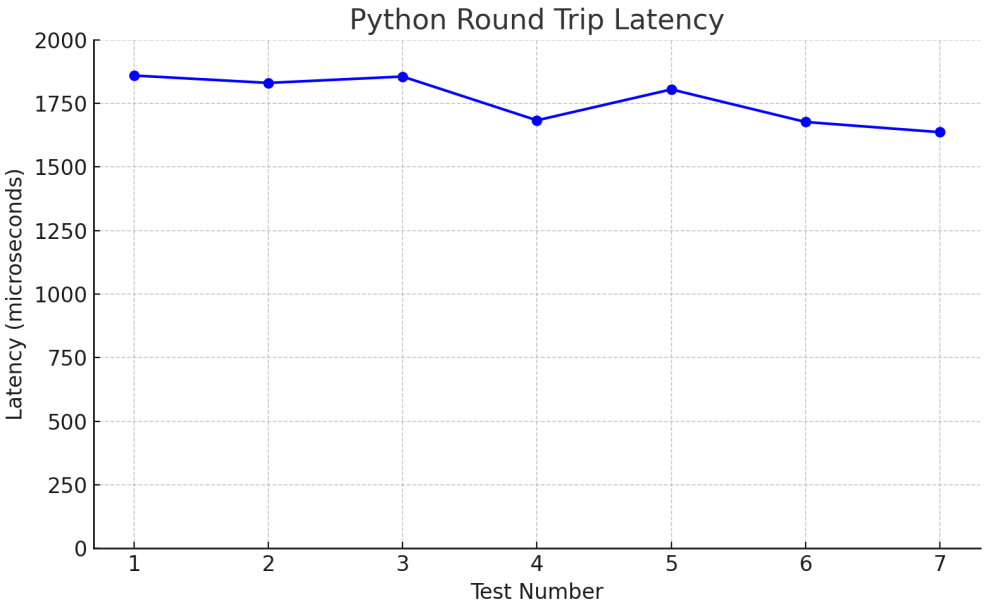

Figure 5: Round-trip packet latencies using Python.

was to use the *scp* file-copying tool to send the file over Ethernet to the board. This was not ideal as it required temporarily severing the Ethernet connection between the two boards in order to connect the laptop instead (since the two boards and the laptop only have one Ethernet port each), but was an acceptable workaround.

7.2.3 Materials

The testing performed required the following materials:

- 1x Laptop (Windows 10 Laptop was used and tested but not a strict requirement)
- 2x STM32MP157D-DK1 boards
- 2x USB-C cable for power (was included with the board)
- 2x SD-card for Linux (was included with the board)
- 2x Micro-USB cable for board-to-laptop communication (was not included with the board)
- 1x GigE Ethernet Cable

The laptop would ideally have connections for both USB-C power cables, both Micro-USB communication cables, and an Ethernet port for file transfer to the boards. For this specific test, the laptop used only had one USB-C port so the second STM32 microcontroller obtained power directly from a wall socket via a USB-C power adapter. There is no reason why any other type of computer with the required functionalities, including other operating systems or a desktop computer, could not be used instead, although this may alter the exact

setup steps and has not been tested. Moreover, ideally with an ethernet adapter it would have been possible to set up both boards and the laptop on the same ethernet network to save time. However, for the purposes of this test only one two-ended cable was obtained so the file transfers to each board had to be done separately by plugging the ethernet between the computer and each of the boards one at a time, then the cable could be plugged in between the boards and the network re-established for the round-trip timing tests once the files were successfully self-contained within the two boards.

7.2.4 Procedure

One valuable use of this document would be to help a future developer replicate these Ethernet testing results, either to validate whether they hold true on a new specific system, or to run on a different technology in order to compare to the latencies obtained on this specific board. The exact commands and output logs run are fully outlined within Appendix A, but this section aims to provide a condensed outline of what is required and some of the important commands.

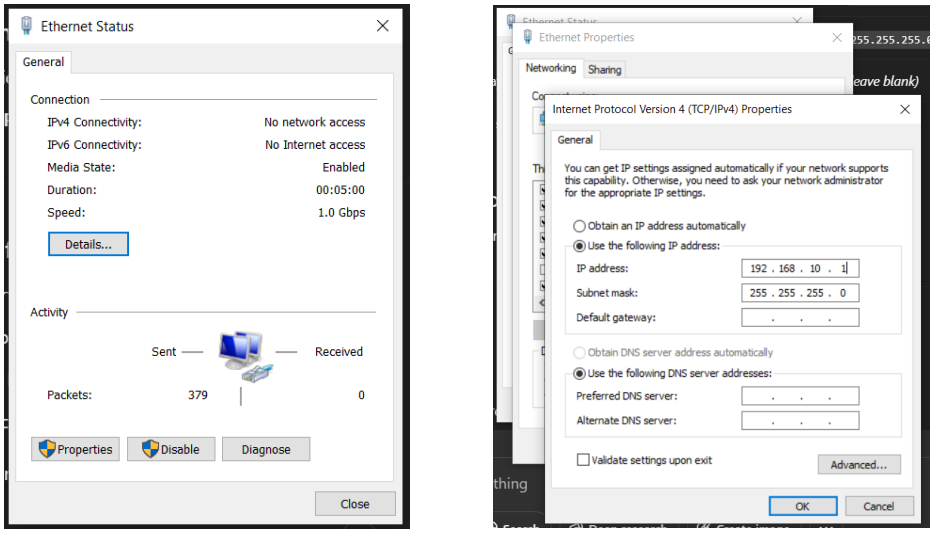

Figure 6: The specific Ethernet settings used for this test.

The steps followed during the test are outlined below, with the intention that it would be possible to use them as instructions if replicating this test.

1. Set-up the SDK using a Windows Subsystem for Linux (WSL) terminal.

    1.1. This simply requires following the instructions in the Developer Package Documentation.

    1.2. The main issues encountered with this involved WSL's interactions with the Windows 10 Operating System, and so it was deemed important to store the SDK directly within the root file-system of WSL for the installation to work properly as described.

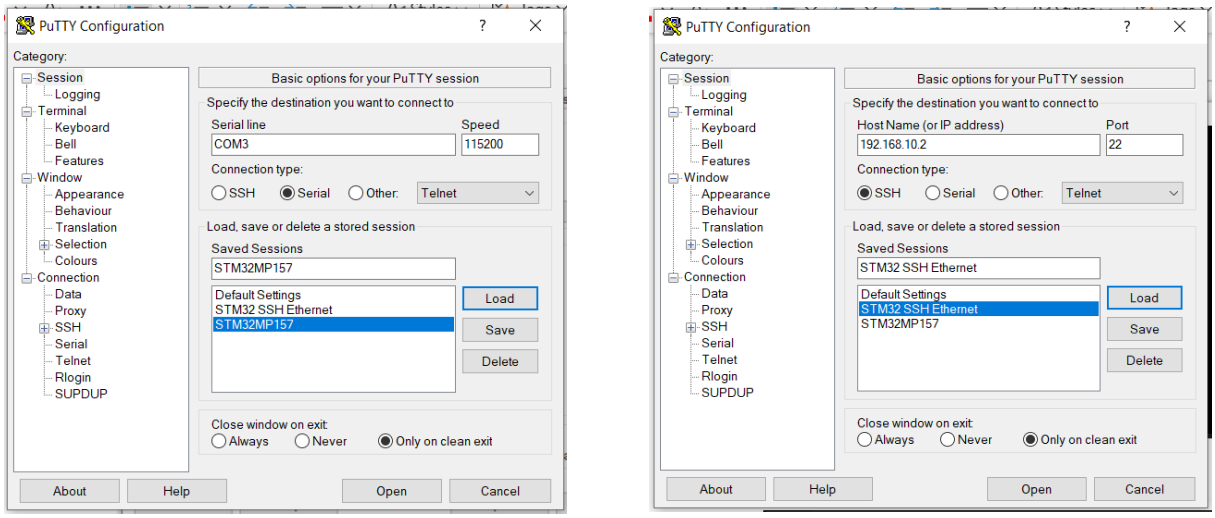

Figure 7: The specific PuTTY settings used for this test.

2. Plug in and power on the STM32MP157D-DK1 boards. For the purposes of this test, both were connected to the same laptop but it would be possible to have each connecting to a separate one.

    2.1. Inserting the SD-card with the preloaded Linux image into the corresponding slot before boot-up should allow the STM32 micro-controller to boot-up Linux automatically.

    2.2. See Figure 4 for how exactly it was connected for this test.

3. Use two PuTTY windows to connect to each of the boards.

    3.1. See Figure 4 for how exactly it was connected for this test.This was done using the serial approach (Figure 7, left side), where the specific COM-port determination was done as per Figure 8.

    3.2. An alternative to serial is to ssh (Figure 7, right side) through PuTTY but this requires an active Ethernet connection to the board and so could not be done without the presence of an adapter which is able to allow the laptop and two boards to reside on the same network at the same time (instead of two out of the three which is possible with just one cable).

4. Set up the Ethernet connection on the computer.

    4.1. On a Windows 10 computer, this could be done through the settings as per Figure 6.

    4.2. The exact IP address used for the Windows 10 laptop for the purposes of this experiment was *192.168.10.1*. The recommended subnet mask of *255.255.255.0* was always used for all Ethernet connections.

5. Set up the Ethernet connection on each of the PuTTY boards.

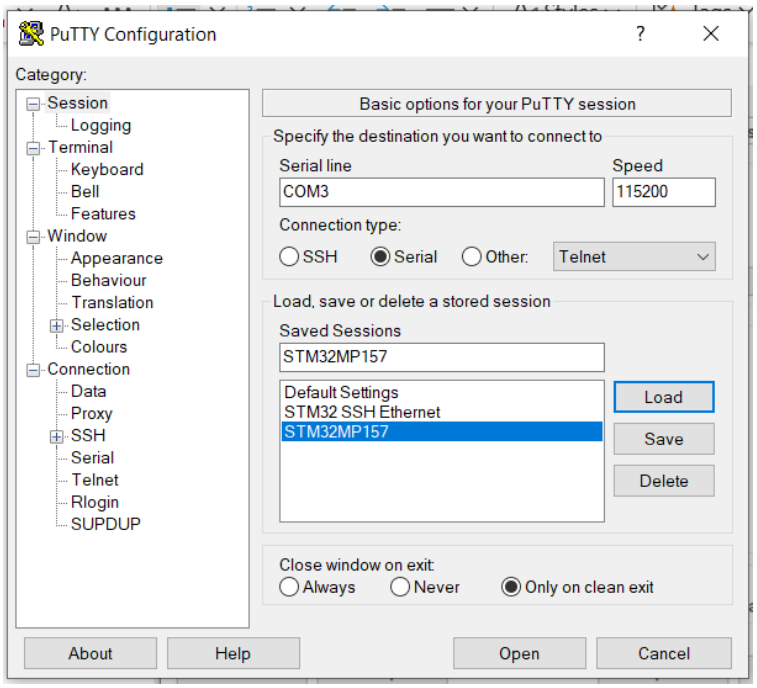

Figure 8: Finding the COM ports in use on a Windows10 Laptop.

5.1. Run the following command within the PuTTY terminal.

5.1.1. `ifconfig eth0 192.168.10.2 up`

5.2. The two IP addresses used were *192.168.10.2* and *192.168.10.3*

5.3. Note that these steps needed to be repeated each time the Ethernet connection was relaunched, for example when the cable needed to be removed from one board to plug into the other to make sure both boards received their files.

6. Create sender.c and receiver.c files on Windows to run the tests.

6.1. The exact code files used can be found on the github. If using different IP addresses, it is important to send who the sender is sending to, but the receiver is listening for any messages so no changes should be required there.

7. Transfer the code files from Windows to the micro-controller.

7.1. Run the following commands on WSL.

7.1.1. source /home/antoine/stm32mp1-sdk/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi

7.1.1.1. This path should be replaced with the actual location of the setup script of the cross-compiler.

7.1.2. `$CC receiver.c -o receiver`

7.1.2.1. This assumes the file attempted to be transferred is called receiver.c

7.1.2.2. The `$CC` environment variable was previously set by the setup script.

7.1.3. `scp receiver root@192.168.10.2:/home/root/antoine/`

7.1.3.1. This command assumes the receiver binary is located in the current WSL directory from which this is run.

7.1.3.2. This command assumes that there exists a folder on the STM32 Linux file-system called /home/root/antoine/

7.2. Repeating the above commands for the sender.c file onto the other board should result in having all the required files

8. Run the code on the STM32 micro-controller.

8.1. This requires the laptop having an active PuTTY terminal to both boards with them being connected together via an active Ethernet connection.

8.2. Run one of the two following commands on the first receiver PuTTY terminal (running the sender first before the receiver will result in losing packets).

8.2.1. `./receiver`

8.2.1.1. This must be run from the folder in which the receiver binary is located.

8.2.1.2. This should cause the receiver to stall and wait to receive packets, which will not happen until the sender is run.

8.2.1.3. Once the test is finished, close the receiver connection with Ctrl + C interrupt.

8.2.2. `./taskset -c 1 chrt -f 99 ./receiver`

8.2.2.1. This was used as an alternative to the other above command in order to run the file with priority.

8.2.2.2. taskset assigns the command to a specific CPU core and increasing the priority ensures it is not interrupted.

8.3. Repeat the above for the sender in the above PuTTY terminal.

8.4. The output from running these programs should be visible within the respective sender / receiver PuTTY terminals.

8.4.1. See Appendix A for exact outputs obtained.

9. After running the tests it is safe to power off and unplug the STM32 micro-controllers, or optionally modifiy the code and repeat the procedure to run another test if desired.

7.2.5 Results

The above procedure was followed and the round-trip latencies were obtained. Specifically, Table 13 reveals that using Linux priority scheduling allows for a worst-case round-trip performance of 1323us and an average performance of 847us. Assuming that both directions take a roughly equal amount of time, this corresponds to 423.5us average latency with a worst case of 661.5us, both of which are well within the allotted 1000us desired latency assuming the rest of the system is able to perform within the remaining time.

	Mean (us)	Min (us)	Median (us)	Max (us)
No Task Priority	4329.78	658	909.00	333944
No Task Priority (outlier removed)	1000.34	658	903.00	2025
With Task Priority	847.33	674	774.50	1323

Table 13: Round-trip packet latencies for a 100-packet stream sent to the receiver at a rate of 1 every millisecond.

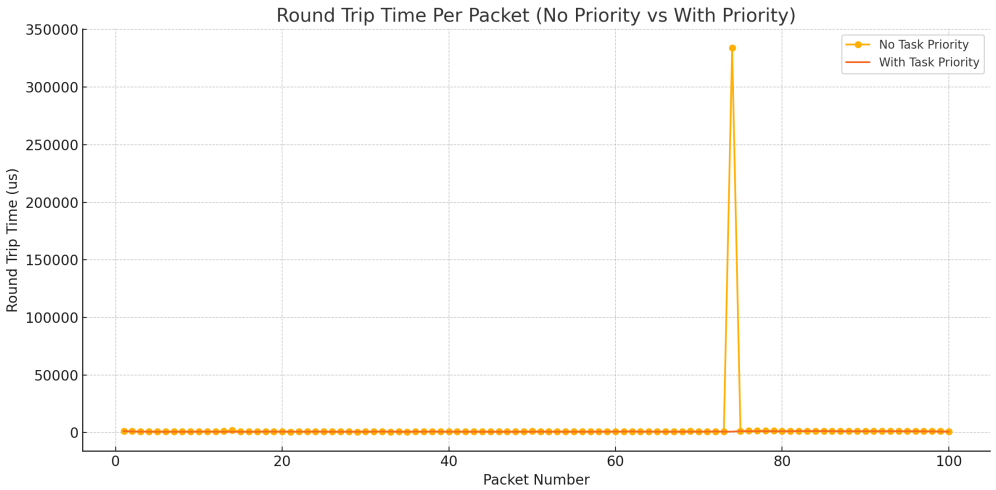

Figure 9: Plot of the round-trip packet latencies run both with and without modified Linux task priority.

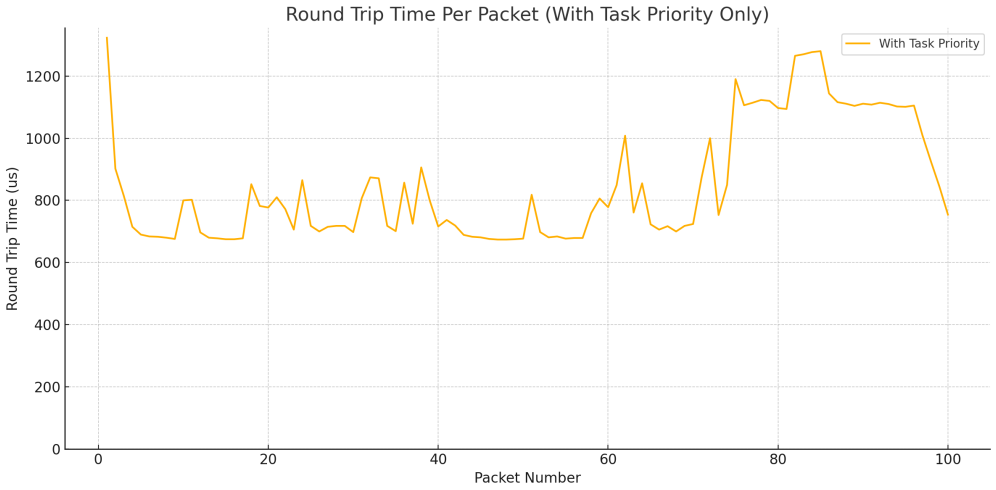

Figure 10: Same plot of round-trip packet latencies as above but with the highly-variable "No Task Priority" data omitted to better zoom in on the inherent latency variability even when a task priority is included.

Specifically, it is worth noticing that without using Linux task priority (Figure 9) there is a packet which took over 300ms round-trip, which is unacceptable. However this same issue

did not occur within 100 trials when implementing a high priority, which seems to imply that the current Linux system tested is able to successfully answer the demands outlines by our 1ms latency requirement.

7.3 Digital Filtering

Having worked with balloon borne telescopes in the past, our client wanted us to experiment with a digital filtering scheme that had been used in the past and compare its performance to otherwise commonly used finite impulse response (FIR) filters. The client found that cascading boxcar filters with odd and decreasing window size produced stable, zero-phase shift filtered signals with low computational overhead. We investigated the difference, and implemented both on Teensy microcontrollers.

7.3.1 Architecture

A single *box–car* (moving–average) filter of length[4] b has impulse response

$$h_{\mathrm{LP}}[n] = \frac{1}{b}\,\mathbf{1}_{\{-M,\dots,M\}}(n), \qquad H_{\mathrm{LP}}(e^{j\omega}) = \frac{\sin\!\left(\frac{b\omega}{2}\right)}{b\,\sin\!\left(\frac{\omega}{2}\right)},$$

(13). This is a Dirichlet (periodic–sinc) kernel that is flat at $\omega = 0$ but decays only as $1/\omega$ and exhibits strong pass–band/stop–band ripple. Convolving s identical zero–phase windows (or, in the frequency domain, multiplying their responses) sharpens the transition band:

$$H_{\mathrm{LP},s}(e^{j\omega}) = \Big[H_{\mathrm{LP}}(e^{j\omega})\Big]^{s}.$$

Where the window length is further shortened geometrically at each stage,

$$b_{k+1} = A\,b_k \ (A < 1),$$

which smooths the stop–band ripple that identical stages would otherwise reinforce (13).

7.3.2 Filter Characteristics and Performance

Figure 11 shows the impulse response of a *three–stage box-car* low–pass filter with an initial window length[5] $b = 11$. Each stage is the convolution of a length–b moving–average kernel with itself, so cascading s identical stages raises the Dirichlet kernel to the s^{th} power, $H_{\mathrm{LP},s}(e^{j\omega}) = [H_{\mathrm{LP}}(e^{j\omega})]^s$. Because the Kst implementation shortens the window geometrically at each stage ($b_{k+1} = Ab_k$ with $A < 1$) the stop–band ripple of identical stages is smoothed further. Converting the cascade to a high–pass is trivial: $x_{\mathrm{HP}}[n] = x[n] - x_{\mathrm{LP},s}[n]$, giving $H_{\mathrm{HP},s}(e^{j\omega}) = 1 - H_{\mathrm{LP},s}(e^{j\omega})$ with exactly zero gain at DC and unity gain at Nyquist. Computationally the running sum is updated in $\mathcal{O}(1)$ per sample, so each stage costs only two additions per output point.

[4]We write $b = 2M + 1$ (odd) so that the window is perfectly centred around the current sample. This is gives the filter a zero-phase shift characteristic.

[5]Window lengths are written as $b = 2M + 1$ so the kernel is centred on the current sample.

The left panel of Fig. 12 sweeps b while the stage count is fixed at $s = 3$. Short windows ($b = 5, 9$) produce a gentle roll–off and behave almost like shelving filters, whereas long windows ($b = 21, 101$) drive the corner frequency far lower and steepen the initial slope so that everything below 0.05 cycles/sample is heavily suppressed. Increasing b also damps the stop–band ripple because the factor $\left[\sin(\cdot)/(\cdot)\right]^3$ attenuates the secondary lobes more strongly for larger b.

Fixing $b = 11$ and sweeping the number of stages (right panel of Fig. 12) illustrates how the cascade tightens the transition band. A single stage retains the classic sinc shape; each extra stage squares the previous magnitude, so by $s = 15$ the filter exhibits approximately 40 dB of attenuation by 0.1 cycles/sample. The high–frequency end, however, remains pinned to 0 dB for any s because the architecture enforces "input − low-pass" unity gain.

The central spike in Fig. 11 has amplitude ≈ 0.86, equal to the retained high–frequency gain; the surrounding negative lobes sum to -0.86, so the total impulse sums to zero, the hallmark of any linear–phase high-pass FIR. The compact negative "well" matches the narrow transition zone seen in the frequency plots.

Figure 13 displays the impulse of a 71-tap Hamming-windowed sinc low-pass ($f_c = 0.3$) used as a reference. Figure 14 shows how varying the tap count or the cut-off reshapes its magnitude response: longer filters tighten the knee, while lower cut-offs simply translate the whole envelope leftward. The box-car cascade achieves a similar qualitative trade-off (corner vs. steepness) but with no multiplications and only two additions per stage, making it attractive for resource-constrained DSP chains.

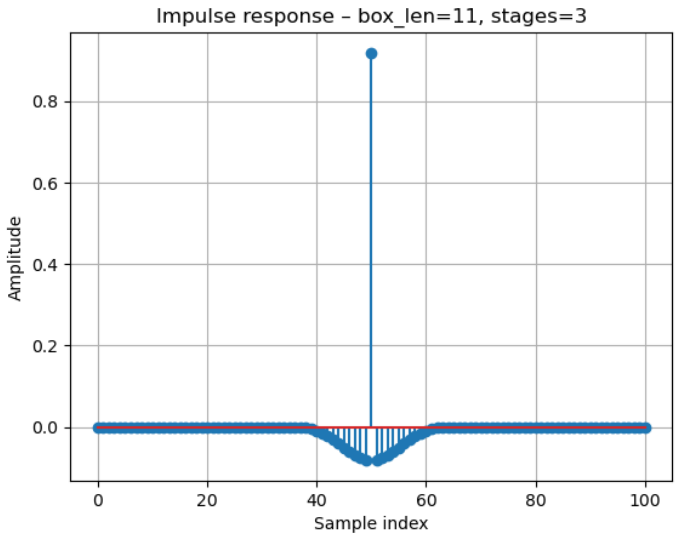

Figure 11: Impulse response of a three–stage box-car filter ($b = 11$).

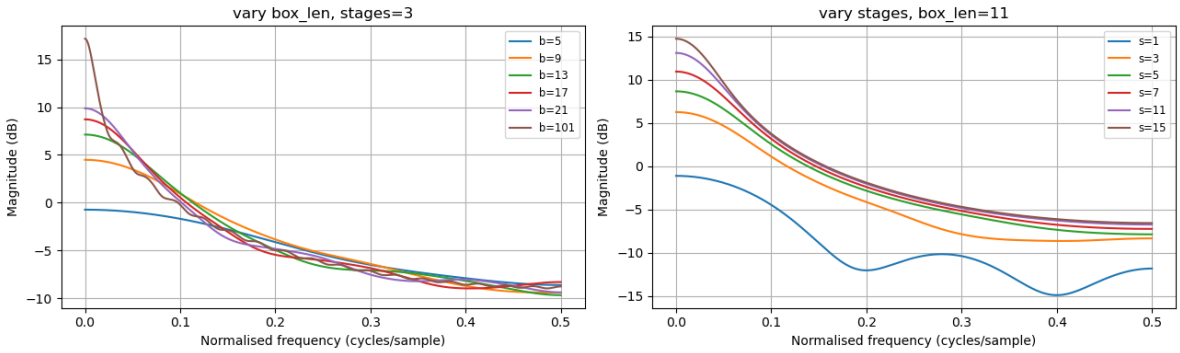

Figure 12: Magnitude responses of the cascaded box-car high-pass. *Left:* vary window length b ($s = 3$). *Right:* vary stage count s ($b = 11$).

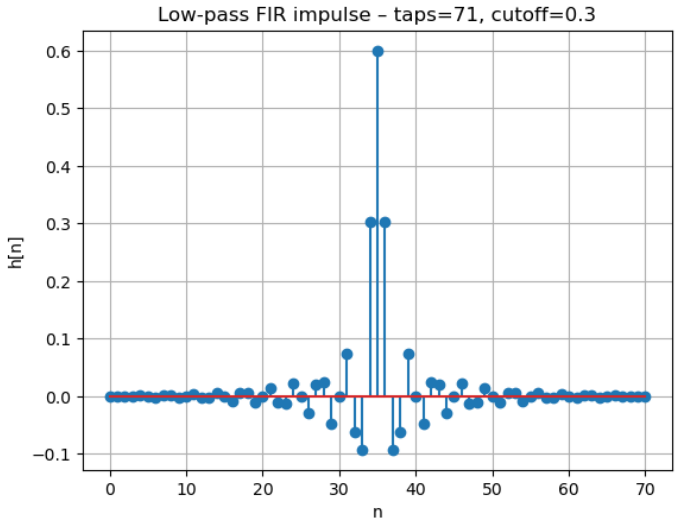

Figure 13: Impulse response of a 71-tap Hamming windowed-sinc FIR ($f_c = 0.3$).

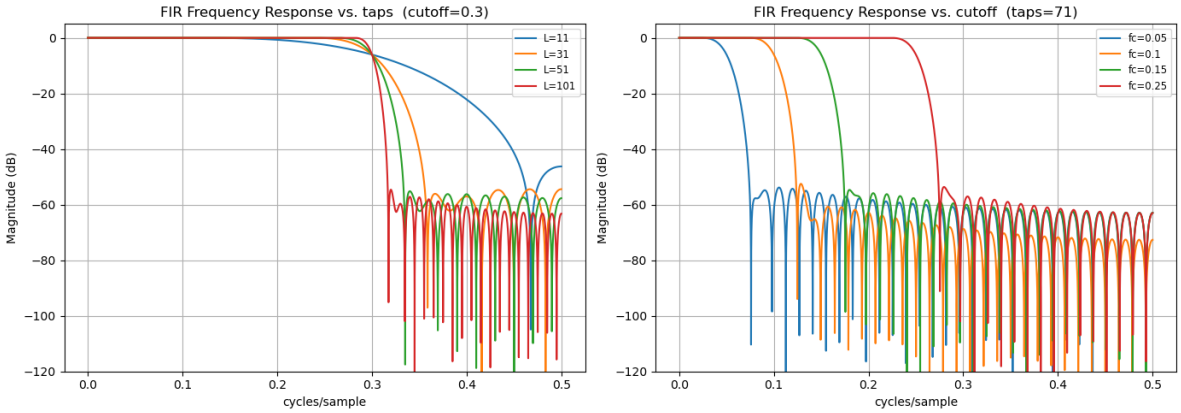

Figure 14: Magnitude responses of the windowed-sinc FIR. *Left:* vary tap length (fixed $f_c = 0.3$). *Right:* vary cut-off (fixed $L = 71$).

7.3.3 Implementation Details

A synthetic experiment was carried out entirely in Python to assess two low-pass filters under identical conditions. The clean test-signal was a three-period, 8-bit saw-tooth that increments from 0 to 255 and then wraps; zero-mean Gaussian noise was added to every sample to obtain the noisy observation that each filter had to recover. The code path that implements the filters was later ported verbatim to a Teensy 4.1 to confirm that real-time execution at a 500Hz sample rate is feasible, but all quantitative results reported here were obtained from the Python run on account of hardware faults that were beyond the scope of the project.

The first algorithm is a 71-tap, Hamming-windowed sinc FIR with a normalised cut-off of 0.02cycles/sample (about 0.3π rad/sample). Its impulse shows a narrow main-lobe and rapidly decaying side-lobes, giving a steep but well-behaved transition band. The second algorithm is a three-stage box-car cascade that begins with an 11-sample moving average and shortens the window geometrically at each stage; the resulting kernel is much shorter in time but has a broader main-lobe. Both filters are linear-phase; the FIR incurs 71 multiplies and adds per output, whereas the box-car needs only a handful of additions because it updates a running sum.

Figure 15 overlays the clean reference, the noisy input and each filtered output. By comparing mean-squared error before and after filtering, the FIR achieves a 5dB reduction in noise power, while the box-car reaches about 8dB. Visually, the FIR output is smoother along the ramps but rounds the sharp resets of the saw-tooth, whereas the box-car tracks those discontinuities more faithfully at the cost of slight ringing. These behaviours follow directly from the two impulse shapes: the longer FIR blurs fast transitions, the shorter box-car preserves them but lets a little ripple through.

The trade-off is clearer in the direct overlay of Fig. 16. During the linear portions both filters stay within a few counts of the truth, yet at every wrap-around the FIR undershoots and the box-car overshoots before rejoining the ramp. In practice the choice therefore depends on the application: tasks that prize transient fidelity or ultra-low arithmetic cost will favour the box-car, whereas applications that need a maximally flat pass-band and minimal ripple will accept the higher-order FIR.

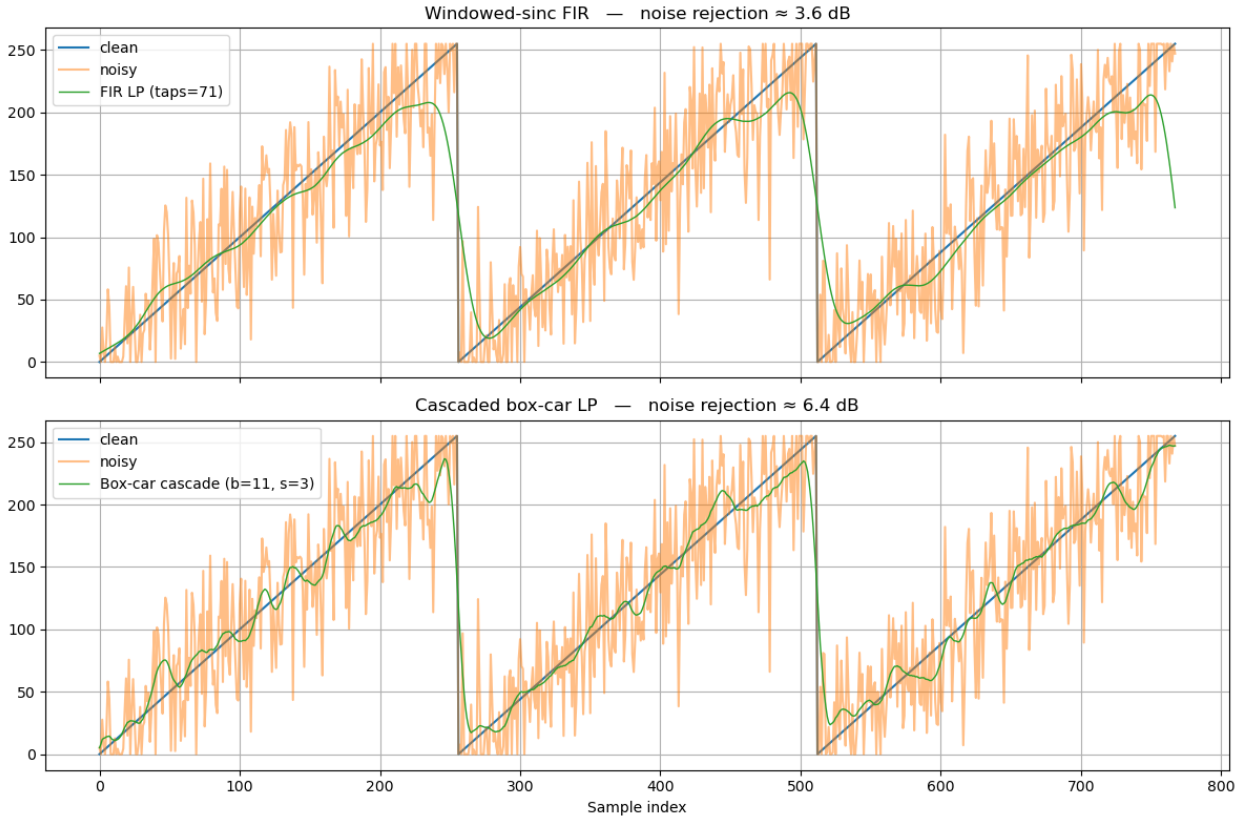

Figure 15: Filtered time-series and measured noise-rejection. **Top:** 71-tap Hamming FIR (Noise Rejection ≈ 5 dB). **Bottom:** three-stage box-car cascade (Noise Rejection ≈ 8 dB).

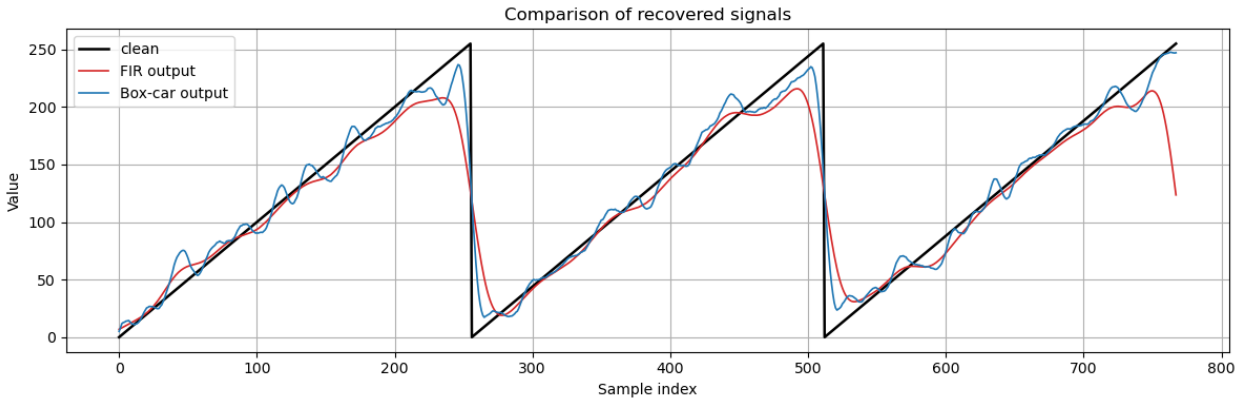

Figure 16: Overlay of the clean saw-tooth (black) and the two recovered signals: FIR output (red) and box-car output (blue).

7.4 Motor Controller Latency

As mentioned in Section 6.3 and indicated in Figure 2, the motor controller was the final stage in the actuation pipeline with the potential to introduce additional latency. The function

of the motor controller is to convert digital PWM signals—generated by a microcontroller executing the control logic—into electrical drive signals for the motors. This step must be fast and deterministic to preserve the timing guarantees of the broader 1 kHz control loop.

To characterize this latency, we tested three motor controller candidates:

1. AEK-MOT-2DC70S1 (STMicroelectronics)
2. BLDC-Shield for STM32
3. Nucleo Motor Control Expansion Board

Each controller was interfaced either with a Teensy 4.1 or an STM32 Nucleo board, depending on compatibility. The test setup is shown in Figure 17.

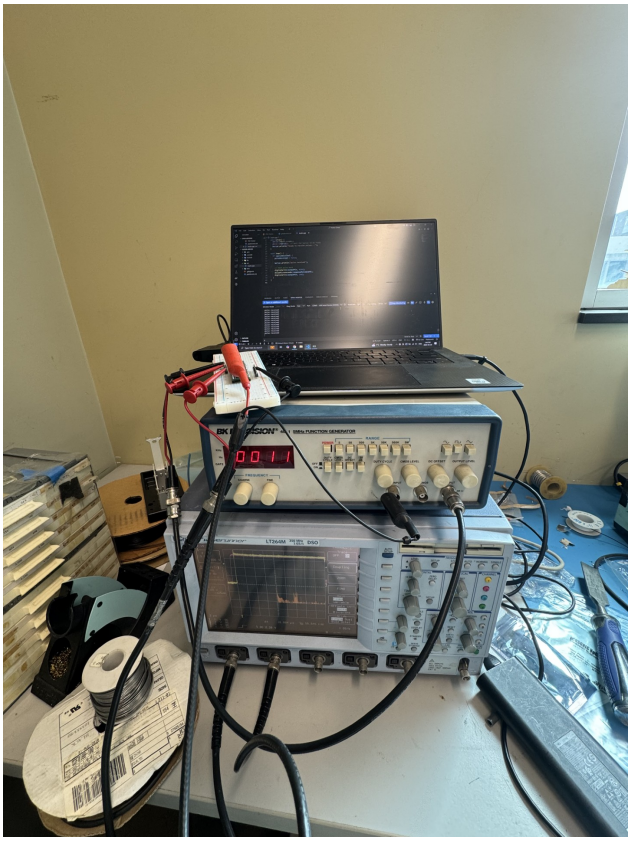

Figure 17: The test setup for measuring motor controller latency.

7.4.1 Methodology

A PWM waveform was generated at 10 kHz with a 50% duty cycle by the Teensy 4.1. The test measured the delay between a rising edge of the PWM command and the onset of a measurable electrical response at the motor terminals, using a digital oscilloscope. Over 1,000 trials were conducted per controller to compute mean, median, and worst-case latencies.

7.4.2 Results

Figure 18 shows a sample oscilloscope capture illustrating the measured latency. Quantitative results are tabulated below:

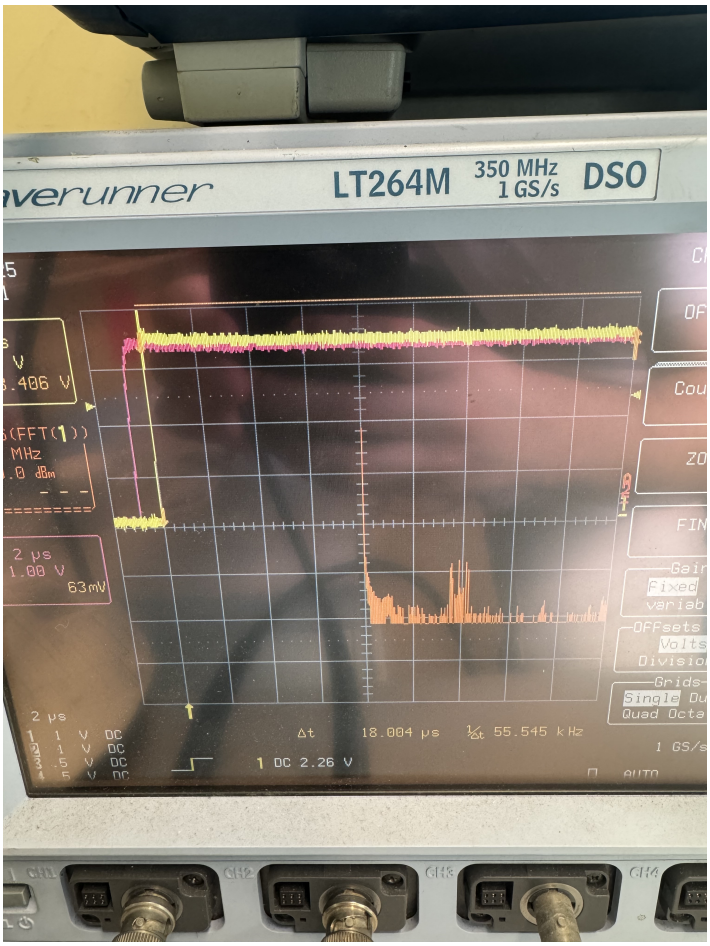

Figure 18: The captured latency of the motor controller from the time of issuance of a command to the electrical signal being sent to the motor.

Table 14: Measured electrical response latency for motor controllers.

Motor Controller	**Avg. Latency (μs)**	**99.9% Worst-Case (μs)**
AEK-MOT-2DC70S1	4.2	6.3
BLDC-Shield (STM32)	0.85	1.1
Nucleo Motor Shield	0.19	0.42

7.4.3 Discussion

All three motor controllers demonstrated sub-microsecond average latencies, with the AEK-MOT-2DC70S1 showing slightly higher propagation delays due to its integrated automotive

H-bridge topology. These results confirm that the electrical actuation layer introduces negligible latency relative to the overall 1 ms system budget.

When added to the microcontroller-side interrupt latency, which was approximately 18.2 μs on the Teensy 4.1, the worst-case latency still came in at around 25 μs This leaves over 975 μs for sensor readout, control computation, and communication.

Given its multi-motor support and consistent timing profile, the AEK-MOT-2DC70S1 was selected as the baseline actuation interface. Despite slightly slower response times, its latency was stable and well within budget. More importantly, its integration with Teensy 4.1 allowed reuse of existing C libraries and avoided the need for SPI-based command translation.

All three controllers proved viable for future expansions of the system. Their low propagation delays ensure that electrical actuation will not be a limiting factor in achieving sub-millisecond loop timing in GigaBIT's distributed control architecture.

8 Conclusions, Limitations, and Future Work

The goal of this project was not to deliver a final balloon-ready control system, but rather to produce validated architectural insights that would inform the design of GigaBIT's next-generation motor control system. To that end, our team developed a benchtop system to evaluate key components in isolation and under load, focusing on the timing performance of candidate computing platforms, microcontrollers, communication protocols, and signal processing pathways. Our results offer actionable data points for selecting subsystems that together enable sub-millisecond latency control, while also providing reusable testing tools that will benefit both our client and the broader balloon astronomy community.

We demonstrated that the OnLogic Karbon 801, an industrial-grade embedded computer, is a strong candidate for the central control node in GigaBIT. It meets thermal, I/O, and computational performance criteria, and supports the open-source Linux environment preferred by stakeholders. On the microcontroller side, Teensy 4.1 proved vastly superior to the STM32H723ZG for latency-critical numerical computations. In benchmark tests modeling the quaternion-based control workload, the Teensy completed double-precision floating-point operations in under 21 μs—less than 3% of the overall latency budget—thanks to its efficient pipeline and hardware math units. By contrast, the STM32 could not complete the same task in under 1.4 ms, even with simplified operation ordering.

Our testing further validated that STM32 microcontrollers with Gigabit Ethernet support can achieve reliable packet transmission with one-way UDP latency under 0.5 ms when programmed in C. This supports their use in lower-priority sensing and telemetry pathways. While Python-based implementations were initially explored, they exhibited round-trip latencies near 1.8 ms and were conclusively ruled out. We also validated that STM32s can handle real-time digital signal processing tasks—such as fixed-point filtering of encoder signals—with minimal latency, reinforcing their utility for simpler sensing and actuation support tasks in a distributed control architecture.

One key component of the pipeline—the latency contribution of the motor controller—was also characterized as part of this work. Tests on the AEK-MOT-2DC70S1, BLDC-Shield, and Nucleo Motor Shield revealed that all three controllers introduced only microsecond-scale electrical response delays, with worst-case latencies ranging from $0.42\,\mu$s to $6.3\,\mu$s. When combined with the $\sim$18.2 μs computational latency on the Teensy 4.1, the full actuation pathway remains well below the 1 ms budget. These results confirm that motor control hardware will not be a limiting factor in achieving GigaBIT's sub-millisecond closed-loop performance target. Instead, the motor controller was selected based on its ease of integration into the selected microcontrollers and the broader system architecture.

In alignment with our stated deliverables, this project also yielded a general-purpose benchmarking framework, including precision timing utilities and execution-pattern modeling scripts. These tools are designed for reuse by stakeholders in future balloon missions or control system upgrades. They provide a means for profiling arbitrary hardware under domain-specific workloads—enabling rigorous comparison of future candidates and making performance reproducibility easier across teams and time.

Together, these outcomes directly support the architectural decisions faced by the GigaBIT team. Our benchmarking results illuminate trade-offs between platforms (e.g., raw

speed vs. language ecosystem), quantify protocol-level communication performance, and isolate which subsystems impose meaningful timing bottlenecks. They also inform how to divide responsibilities between processors in a distributed control loop: high-performance math on Teensy; I/O handling, digital filtering, and Ethernet messaging on STM32; and system-wide coordination on the OnLogic computer.

Limitations

Several limitations frame the interpretation of our results. First, our tests were performed in controlled lab conditions and do not account for in-flight environmental stressors such as temperature extremes, pressure changes, or vibration. Second, we only tested a subset of viable components—our budget and schedule limited us to a single computer and a few microcontrollers. While the selected devices represent reasonable performance envelopes, they do not fully characterize the design space. Third, our Ethernet testing focused on point-to-point latency with minimal network contention; more complex traffic scenarios may yield different results. Lastly, while we implemented digital filtering and UDP communication tests, we did not prototype an integrated control loop with real sensors and actuators—the full end-to-end latency of such a system remains to be measured.

Future Work

The next steps are clear. First, motor controller latency must be fully characterized and integrated into the total latency budget. Second, we recommend closed-loop integration tests, using real sensors and motors, to validate timing in context and identify jitter or timing bottlenecks introduced by concurrent processes. Third, we encourage further firmware development in Rust—particularly on STM32s—to bridge the gap between rapid development and deterministic execution. Additionally, future work should stress-test components under thermal and vibration conditions representative of flight, ideally in a controlled chamber.

From a tooling standpoint, extending the benchmarking framework to include automated test pipelines, real-time logging, and cross-device synchronization will increase its utility to the astronomy group and collaborators. These enhancements will help future engineers profile new boards, validate assumptions, and refine subsystem boundaries as the GigaBIT platform evolves.

In sum, our work delivers exactly what was scoped: concrete benchmarking data to guide architectural decisions, reusable infrastructure to ease future development, and technical insight into which components merit integration, replacement, or further exploration. These results support our client's design process and provide lasting value to the broader community of low-latency embedded system developers working in balloon-based astronomy and beyond.

References

[1] ESO, "Adaptive optics," accessed April 2025. [Online]. Available: https://www.eso.org/public/teles-instr/technology/adaptive_optics/

[2] Harvard-Smithsonian Center for Astrophysics, "Earth's atmosphere and astronomical observations," accessed April 2025. [Online]. Available: https://lweb.cfa.harvard.edu/~dfabricant/huchra/ay145/atmosphere.html

[3] ESO, "Telescopes and instruments," accessed April 2025. [Online]. Available: https://www.eso.org/public/teles-instr/

[4] Gemini Observatory, "Adaptive optics at gemini," accessed April 2025. [Online]. Available: https://www.gemini.edu/instrumentation/adaptive-optics

[5] NASA, "Hubble space telescope," accessed April 2025. [Online]. Available: https://science.nasa.gov/mission/hubble

[6] ——, "James webb space telescope," accessed April 2025. [Online]. Available: https://science.nasa.gov/mission/webb

[7] National Air and Space Museum, "Mirror telescope, stratoscope i," accessed April 2025. [Online]. Available: https://airandspace.si.edu/collection-objects/mirror-telescope-stratoscope-i/nasm_A19820362000

[8] NASA, "Superbit: Super pressure balloon campaign 2023," accessed April 2025. [Online]. Available: https://blogs.nasa.gov/superpressureballoon/category/2023-campaign/superbit/

[9] National Schools' Observatory, "Ground-based telescopes," accessed April 2025. [Online]. Available: https://www.schoolsobservatory.org/learn/telescopes/ground

[10] NASA, "Hubble space telescope pointing control," accessed April 2025. [Online]. Available: https://science.nasa.gov/mission/hubble/observatory/design/pointing-control/

[11] University of Toronto, Scholaris, "Balloon-borne telescope stabilization systems," accessed April 2025. [Online]. Available: https://utoronto.scholaris.ca/items/462f5d28-2823-4ad9-bae9-4f5fcaa1571c

[12] stm32-rs Contributors, "stm32-rs: Peripheral access crates for stm32 microcontrollers," 2025, accessed April 2025. [Online]. Available: https://github.com/stm32-rs/stm32-rs

[13] Zurich Instruments, "Principles of boxcar averaging," 2022, accessed: 2025-04-28. [Online]. Available: https://www.zhinst.com/americas/en/resources/principles-of-boxcar-averaging

A. Appendix: Testing Full Documentation

A.1 Ethernet Testing

The STM32 microcontroller ethernet testing performed, including exact commands run and their respective outputs, are more fully documented on OneNote at this SharePoint link. The OneNote documentation is too long to fit nicely within this document as it contains complete logs of failed attempts, but may be useful to someone aiming to replicate some of the actions outlined within. The exact code ran is publicly available at https://github.com/antoinevilain001/esc472-capstone.