AIAA Regional Student Conferences
, Multiple Locations
2025 Regional Student Conferences

10.2514/6.2025-99648

Real Time Operating Systems in Active Controlled Rockets

Kanav Chugh *, Samuel Peterson † and Felipe Martins ‡
Georgia Institute of Technology, Atlanta, Georgia, 30332

Modern rocketry requires sophisticated algorithms to handle complex control and monitoring tasks in real-time environments. Although a simple loop suffices for non-critical applications, thrust vector control (TVC) systems require simultaneous management of TVC, data acquisition, and telemetry. In a single loop, these would be handled sequentially, with the system crashing if one process fails. To mitigate this risk, a real-time operating system (RTOS) provides multithreading, scheduling, and deterministic timing. This paper presents the development and integration of an RTOS on an L2 TVC rocket and details the overall system architecture, kernel, I/O, and its application on an avionics system. The operating system features six layers: a CMSIS layer for registers, a HAL for peripherals, a device layer managing sensors, a kernel layer for scheduling and memory management, a file system layer for data storage, and an application layer for tasking and algorithms. The kernel includes a preemptive scheduler that manages context switching, processing, and memory. I/O is handled through a device driver framework that supports interrupts and DMA-based transfers. In its integration on a high-powered rocket, the avionics system maintained consistent 3 millisecond latency on telemetry and sub-microsecond latency in context-switching. Flight testing validated the architecture's reliability with telemetry having 3.2% packet loss and data-logging on a flash chip. A key innovation in this application is the implementation of a Linear Quadratic Regulator (LQR) within a timer interrupt. Through this operating system, this system provides a foundation for multithreaded rocket software that can be applied in subscale and larger-scale rocketry.

I. Nomenclature

API	=	Application Program Interface
CMSIS	=	Common Microcontroller Software Interface Standard
CPU	=	Central Processing Unit
DMA	=	Direct Memory Access
HAL	=	Hardware Abstraction Layer
I/O	=	Input/Output
ISR	=	Interrupt Service Routine
LQR	=	Linear Quadratic Regulator
MCU	=	Microcontroller Unit
NVIC	=	Nested Vector Interrupt Controller
OS	=	Operating System
QSPI	=	Quad Serial Peripheral Interface
RAM	=	Random Access Memory
RTOS	=	Real-Time Operating System
SPI	=	Serial Peripheral Interface
TVC	=	Thrust-Vector Control
UART	=	Universal Asynchronous Receiver Transmitter

*Undergraduate Student, Van Leer Department of Electrical and Computer Engineering, Georgia Institute of Technology, 777 Atlantic Dr NW, Atlanta, GA 30332, AIAA Student Member 1810487
†Undergraduate Student, College of Computing, Georgia Institute of Technology, 777 Atlantic Dr NW, Atlanta, GA 30332, AIAA Student Member 1810845
‡Undergraduate Student, Van Leer Department of Electrical and Computer Engineering, Georgia Institute of Technology, 777 Atlantic Dr NW, Atlanta, GA 30332, AIAA Student Member 1810876

II. Introduction

Rocketry has transformed fundamentally in software and control system architecture. Since the 1960s, NASA has evolved its programming techniques inside of its avionics systems, from bare-metal programming to large-scale operating systems. Modern day rocketry implements thrust-vector control (TVC) for actuation, adding additional software complexity, processing, and hardware. With TVC, sophisticated solutions must satisfy critical requirements. First, TVC systems must simultaneously manage multiple critical processes: the TVC actuation itself, continuous data acquisition from various sensors, state estimation, sensor fusion, and real-time telemetry. In traditional single-loop architectures, these processes would be handled sequentially, creating potential points of failure where one process malfunction could lead to catastrophic system failure. To address these challenges, a real-time operating system (RTOS) provides these essential capabilities: multitasking for parallel process execution, scheduling for process management, memory management, and deterministic timing for critical operations.

While an RTOS is ubiquitously implemented into large-scale rockets such as SpaceX's Starship, they remain notably absent in the subscale rocketry community. Most hobbyist and amateur TVC rockets continue to rely on simple, sequential programming approaches, despite their inherent limitations. This gap between professional and subscale implementations creates an opportunity to bring enterprise-grade control system architectures to smaller rockets. This architectural divide is not merely a matter of scale. The absence of RTOS in subscale rockets is due to several factors: perceived complexity of implementation, limited documentation in avionics applications, a historical focus on simplicity in amateur rocketry software, and hardware limitations of microcontrollers. However, as subscale rockets become more sophisticated, the limitations of sequential programming become increasingly apparent. Single-threaded approaches force critical control tasks to compete with memory and data-critical operations, introducing potential failure modes and limiting system reliability.

The challenge lies in adapting the RTOS concepts to the constraints of subscale rocketry. Unlike commercial launch vehicles with powerful flight computers, subscale rockets must operate with small processing speeds, memory, storage, and power. Traditional RTOS implementations often assume resources that are simply not available in microcontroller-based avionics systems, such as memory management units and multicore CPUs. This paper presents a comprehensive RTOS specifically designed for subscale rocket applications on microcontrollers, bridging the gap between professional and amateur rocketry practices. The system architecture addresses these challenges through a bare metal approach, incorporating hardware abstraction, kernel-level scheduling, and device drivers while remaining lightweight for embedded systems.

III. Background

To fundamentally understand how this bare-metal RTOS operates, it is essential to know the concepts of cores, threads, and tasks. A core serves as a unit of a processor inside a larger-scale unit such as a CPU or an MCU, containing its own registers, cache, and data. A thread represents the smallest sequence of instructions that can operate independently, managing elements such as a program counter, link register, status register, or stack pointer. Tasks function as higher-level operating system units that typically encompass multiple threads running in parallel. Most microcontrollers in avionics systems employ single-core systems, which prevent true multithreading. Instead, the scheduler, which functions as the unit that manages tasks and threads, rapidly alternates between tasks to create the illusion of a parallel system. It actively saves the current task's state including register values and stack pointer and loads the new task's state.

IV. Basic Requirements

The following are basic requirements for a real-time operating system:

Multitasking and preemptability: To support multiple tasks in real-time, an RTOS must be preemptable. The scheduler should be able to preempt any task in the system and give the resource to the task that needs it most. An RTOS should also handle multiple levels of interrupts to handle multiple priority levels.

Dynamic deadline identification: In order to achieve preemptability, an RTOS should be able to dynamically identify the task with the earliest deadline. To handle deadlines, deadline information may be converted to priority levels that are used for resource allocation.

Predictable synchronization: For multiple threads to communicate among themselves in a timely fashion, predictable inter-task communication and synchronization mechanisms are required. Semantic integrity and timeliness constitute predictability.

Sufficient Priority Levels: When using prioritized task scheduling, the RTOS must have a sufficient number of priority levels, for effective implementation. If the number of priority levels is too little, tasks may not be properly distinguished, leading to unintended delays. This, in turn, leads to priority inversion which occurs when a higher priority task must wait on a lower priority task to release a resource and in turn the lower priority task is waiting upon a medium priority task.

Predefined latencies: The timing of system calls must be defined using the following specifications [1]:

- Task switching latency or the time to save the context of a currently executing task and switch to another.
- Interrupt latency or the time elapsed between the execution of the last instruction of the interrupted task and the first instruction of the interrupt handler.
- Interrupt dispatch latency or the time to switch from the last instruction in the interrupt handler to the next task scheduled to run.

V. System Architecture

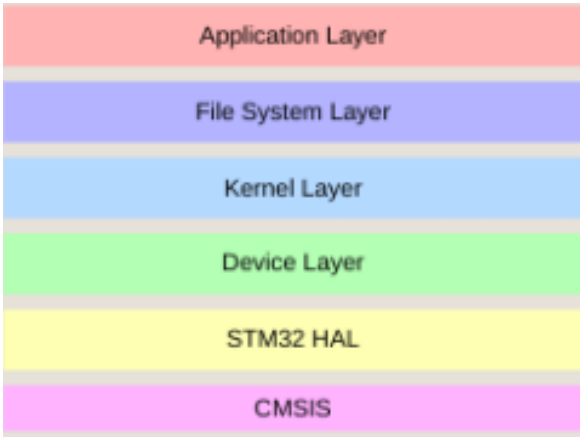

Fig. 1 General System Architecture

A. Operating System Overview

The operating system architecture implements a layered approach for abstraction and service management for avionics systems [2]. Each layer provides specific services to the layers above while utilizing functionality from other layers, creating a hierarchical structure that ensures modularity and reliability. The lowest layer is CMSIS which provides direct hardware access, CPU register information, standardized register definitions, and core processor functionality. This standardization ensures portability across different ARM-based microcontrollers and provides low-level services such as interrupt support.

Building upon CMSIS, the HAL provides hardware-specific abstraction for the microcontroller. This layer manages hardware initialization, protocol initialization, clock configuration, and basic peripheral drivers. It abstracts the complexities of hardware timers, protocol implementations, DMA operations, and low-level interrupt handling, providing a consistent interface for higher layers while maintaining direct control over hardware resources. The HAL provides hardware independence, allowing different microcontrollers to be used by only requiring changes to the HAL implementation while maintaining the same interface for higher layers. While this RTOS architecture supports various hardware abstraction layers, this implementation specifically utilizes the STM32 HAL.

The Device Layer sits above the HAL, implementing device drivers and managing hardware interfaces. This layer manages device-specific interrupt service routines, power state status, and DMA operations. This abstraction allows sensors such as IMUs, GPSs, barometers, and radio modules to be managed effectively.

The kernel provides core operating system services, forming the central management unit for the avionics system. It serves as an intermediary between hardware resources and user applications, managing the scheduler, tasking, and memory. The kernel implements fundamental services such as task scheduling, context switching, memory management, and software interrupt handling. Unlike traditional operating systems, a bare-metal kernel does not have dynamic

memory allocation as microcontrollers do not have a dedicated memory management unit.

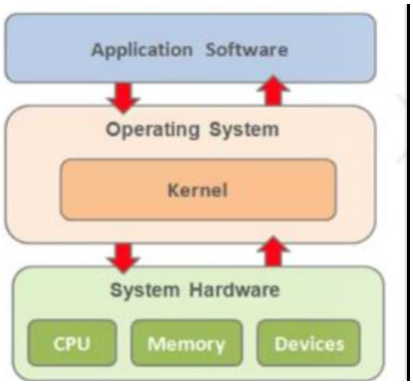

Fig. 2 Kernel Layer Diagram

Above the kernel, the File System layer manages data storage operations, implementing file systems for non-volatile storage. This layer handles buffer management, storage device interfaces, and data transfers. It provides essential services for flight data logging, configuration storage, and system state management.

At the highest level, the Application Layer contains user tasks, state estimation, and control algorithms. This layer implements flight control logic, sensor fusion, telemetry handling, and system monitoring. As seen in Fig. 2, application code utilizes all underlying services while remaining isolated from hardware complexities. This layered architecture provides several key advantages for rocket avionics: clear separation of concerns reduces potential failure modes, modular design allows for component updates without system-wide changes, and hardware abstraction simplifies portability across different microcontroller platforms.

B. CMSIS

CMSIS forms the foundation of the operating system, providing standardized access to an ARM processor, registers, and peripherals. CMSIS maintains direct control over the hardware. For avionics applications, CMSIS provides core processor access, standardized peripheral access, and digital signal processing capabilities.

The CMSIS-Core component provides standardized access to the processor core and peripherals, particularly important for interrupt handling and system timing. While it can be considered as hardware abstraction rather than true low level control, for the scope of this operating system, it is the lowest level for consideration. It enables direct configuration of NVIC, System Clock, ARM Registers, and core access functions, which are crucial for managing hardware interfaces on an ARM Cortex microcontroller. The standardization of register access ensures consistent behavior across different ARM-based flight computers.

CMSIS-Driver provides standardized peripheral driver interfaces. This abstraction is particularly valuable for avionics systems, which must interface with multiple sensors through various protocols. The driver interfaces standardize access to communication peripherals such as SPI and UART with lower-level register manipulation. It includes additional protocols for device-to-device communication, wireless communication, storage, and external memory. These standardized interfaces ensure that sensor drivers remain portable across different ARM-based microcontrollers.

CMSIS-DSP provides common digital signal processing functions such as matrix operations, filtering, transforms, interpolation, quaternion functions, and complex math functions. These optimizations enable the development of state estimation algorithms and controllers without dynamic allocation nor larger memory overhead.

C. HAL

The Hardware Abstraction Layer (HAL) builds upon CMSIS-Core to provide high-level, hardware-specific interfaces for the STM32 peripherals. While CMSIS-Core provides the foundational definitions for memory layout, interrupt numbers, and register mappings, the HAL implements the actual driver code that controls these peripherals. The HAL offers two complementary levels: the high-level HAL drivers and the low-level (LL) drivers. The HAL drivers provide feature-oriented APIs that abstract hardware complexity while ensuring portability across the STM32 family.

For avionics applications, these drivers manage critical peripherals such as SPI for sensor communication, UART for telemetry, and timers for control system timing. The LL drivers operate closer to the hardware level, providing register-level access for performance-critical operations where minimal overhead is required, such as in interrupt handlers or high-speed data acquisition.

In this avionics implementation, the HAL interfaces with CMSIS-Core's definitions to manage hardware resources (see Fig. 4). For example, when configuring SPI for an IMU, the HAL uses CMSIS-Core's peripheral base addresses and register definitions while providing the actual implementation of SPI initialization, data transfer, and interrupt handling through the STM32 HAL. This layered approach allows the HAL to maintain hardware independence in the upper layers while efficiently managing hardware-specific details.

The HAL's abstraction is particularly valuable for avionics systems in several ways. First, it provides standardized error handling and peripheral state management. Furthermore, it implements efficient protocol implementations such as UART, SPI, and QSPI. Finally, it manages clock configurations and power modes, allowing for optimal performance and power management. For example, on an STM32H7, a developer can configure its power mode to either use a switch-mode power supply (SMPS) or a low-drop off (LDO) regulator, providing variable power sequencing

Code configuration is completed through STM32CubeMX. Through CubeMX, the HAL provides a configuration system that generates initialization code for all peripherals. This approach ensures proper hardware setup while maintaining the flexibility to optimize critical sections using LL drivers when necessary. The HAL's structure supports the layered architecture of the avionics system, providing a robust foundation for device drivers while abstracting hardware complexities from higher layers.

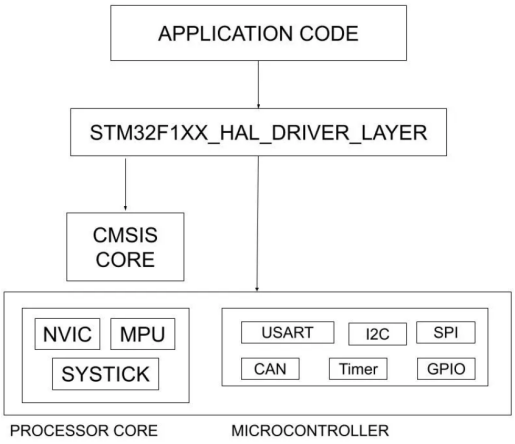

Fig. 3 HAL Diagram

D. Devices and I/O

The Device Layer manages hardware peripherals and their interfaces using a device driver network. Unlike traditional operating systems that use separate device trees from the OS image, this implementation incorporates device configurations directly into the system image through a centralized device.c file. This approach, while less flexible than dynamic loading, reduces boot-time overhead.

Device drivers in this layer build upon the HAL's peripheral interfaces to implement higher-level device management. Each driver manages device-specific state machines, error handling, and data processing. For avionics applications, this includes drivers for Inertial Measurement Units (IMU), barometers, magnetometers, and Global Position System (GPS) modules, as well as actuators for TVC. The drivers handle protocol-specific implementations while managing device-specific ISRs and DMA operations.

I/O operations are optimized through a combination of interrupt and DMA-based transfers. For sensor data acquisition, DMA channels are configured to transfer data directly from peripherals to memory, minimizing CPU overhead. Interrupt service routines (ISRs) are structured hierarchically: hardware interrupts are caught by HAL handlers, which then trigger device-specific callbacks in the appropriate driver. This architecture ensures efficient handling of multiple concurrent I/O operations while maintaining system responsivity.

The device layer provides several abstractions for hardware management. First, it implements device initialization and configuration sequences, ensuring proper hardware setup during system startup. Second, it manages device power states and error recovery. Third, it provides buffering and queuing mechanisms for device data, essential for managing sensor data streams and telemetry. These abstractions isolate hardware-specific details from higher layers.

E. Kernel

The kernel is the core component within the OS and bridges the hardware components and software algorithms. Each executing program is a task under control of the operating system. If an operating system can execute multiple tasks, it is said to be multitasking. The use of a multitasking operating system simplifies the design of what would otherwise be a complex software application. The kernel layer implements tasking, scheduling, and context switching.

1. Tasking

The kernel implements a task-based concurrency model optimized for avionics applications [4]. Each task represents an independent thread of execution with its own context, stack space, and priority level. The task structure, derived from FreeRTOS [5], is defined by a Task Control Block (TCB), which contains:

- **Task Identifier:** Unique 32-bit identifier for system reference
- **Priority Level:** 8-bit value determining scheduling precedence
- **State:** Current execution state (Running, Ready, Blocked, or Suspended)
- **Event Flags:** Bitmap indicating pending events or conditions
- **Deadline:** Optional temporal constraint for hard real-time tasks

Task creation follows a static allocation model, where all tasks are created during system initialization. This approach, while less flexible than dynamic creation, eliminates any heap fragmentation issues. The static model is particularly advantageous for avionics systems where task requirements are well-defined. Tasks are created through TaskCreate, which implements the following sequence:

- Stack allocation from the system heap
- TCB initialization with task parameters
- Context frame setup on the task's stack
- Registration with the scheduler

The system supports four fundamental task states, each serving a specific purpose in the avionics context:

1) **Running:** The task is currently executing on the processor.
2) **Ready:** The task is eligible for execution but waiting for processor time.
3) **Blocked:** The task is waiting for a temporal or I/O event.
4) **Suspended:** The task has been explicitly suspended by the system.

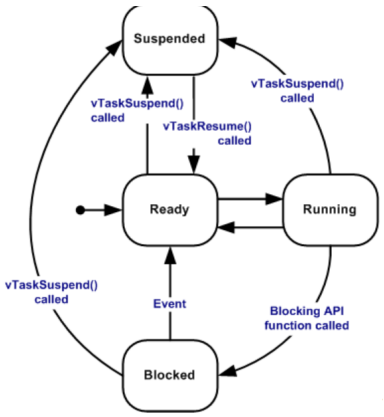

Fig. 4 Task State Machine

This tasking implementation is specifically optimized for avionics applications in several ways:

- Stack allocations are calculated based on worst-case execution paths, preventing stack overflow. For example, an LQR task typically requires 4KB of stack space to accommodate trim point calculations.
- The system implements 256 priority levels, allowing fine-grained control over task execution. Critical tasks such as telemetry typically operate at priorities 200-255.
- The task context is optimized for ARM Cortex-M processors, storing only essential registers to minimize context switching overhead. This optimization achieves sub-microsecond context switch times.

2. Scheduler

The scheduler forms the key component regarding task management. While general-purpose schedulers often prioritize fairness or throughput, avionics applications demand precise timing and predictable behavior. This RTOS implements a priority-based preemptive scheduler with round-robin time slicing, specifically designed to meet the demanding requirements of flight control systems. For aerospace applications, three key scheduler features are essential:

1) **Fixed Priority Preemption**: The scheduler uses 256 priority levels to ensure critical control tasks receive immediate processor attention. This granular priority system allows precise control over task execution, ensuring that high-priority operations like telemetry can preempt lower-priority tasks like data logging. This is crucial for maintaining stable flight control. For example a delayed TVC response could lead to flight instability.
2) **Deterministic Timing**: The scheduler maintains strict timing guarantees through a tick-based system. This enables precise scheduling of periodic tasks, such as sensor sampling and control loop execution. The tick frequency is configurable, allowing adaptation to different flight control requirements while maintaining temporal accuracy. For a microcontroller, the tick must be set at System Tick for the scheduler to manage real-time context switching.
3) **Round-Robin Scheduling**: For tasks of equal priority, the scheduler implements round-robin time slicing. This ensures fair distribution of processor time among tasks with the same priority level, particularly important for managing multiple tasks at different frequencies (e.g. PID and data logging) that operate at the same priority.

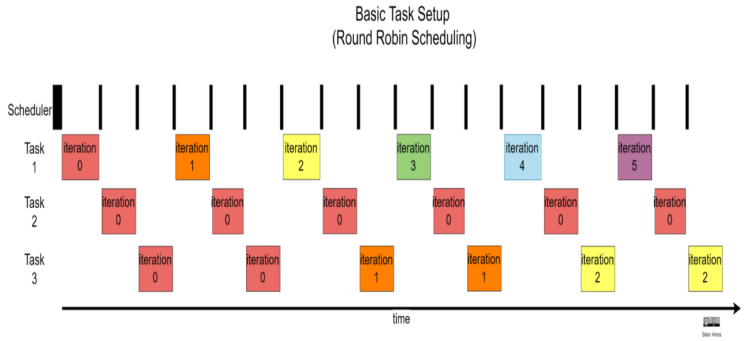

Fig. 5 Round Robin Scheduler Example

The scheduler includes these lists for organizing tasks:
- Ready List: Priority-ordered queue of tasks ready for execution
- Delayed List: Time-ordered list of tasks waiting for temporal events
- Event Lists: Lists of tasks blocked on specific system events
- Pending Ready List: Temporary storage for tasks made ready while scheduler is suspended

3. Context Switching and Synchronization Mechanisms

Context switching is fundamental to maintaining stable flight control while managing multiple tasks. The RTOS implements several types of context switching mechanisms to handle different needs in avionics applications.

Binary Semaphores: Essential for basic task synchronization, particularly between interrupt service routines and tasks. In avionics applications, binary semaphores enable efficient signaling of data availability without the overhead of message queues.

Mutexes: Mutexes implement a priority inheritance mechanism to prevent priority inversion, making them ideal

for protecting shared resources in TVC applications. A mutex acts like a token that is used to guard a resource. When a task wishes to access the resource it must first obtain ('take') the token. When it has finished with the resource it must 'give' the token back - allowing other tasks the opportunity to access the same resource.

Event Groups: Event groups indicate if an event has occurred. They enable tasks to wait for multiple synchronization conditions simultaneously, useful for complex synchronization scenarios in sensor fusion or control systems. These are usually integrated as bit flags on a single variable. It includes multiple events on a single call, support for AND/OR logic in events, and atomic operations for bit masking.

Queues: Queues provide thread-safe data transfer and synchronization between tasks, and between tasks and interrupts. They can be used to send messages between interrupts and tasks, using a FIFO buffer.

These synchronization primitives work together to ensure reliable system operation. For example, during a typical flight, TVC and parachute deployment use direct notifications for minimal-latency synchronization. Sensor data acquisition uses binary semaphores for interrupt signaling. Data logging uses mutexes to protect shared state vectors. System monitoring uses event groups to track multiple health indicators. Telemetry and message buffers use queues to buffer and transmit flight data.

F. File System

The File System layer in the RTOS provides a structured interface for data storage operations on external memory devices, essential for flight data logging and configuration management. Unlike traditional operating system file systems that manage program storage and execution, this implementation focuses specifically on handling flight data through non-volatile storage devices like NOR flash chips and SD cards. This specialization allows for optimized performance in high-speed data logging scenarios while maintaining data integrity throughout flight operations.

For TVC rocket applications, the file system layer serves several critical functions. Primary among these is high-speed flight data logging for post-flight analysis, which requires fast write speeds and guaranteed data integrity. The system also manages storage of configuration parameters and calibration data, providing atomic write operations to prevent corruption during updates.

The file system manages two primary types of storage devices, each serving distinct purposes in the avionics system. Flash chips, accessed through QSPI interfaces, serve as the primary storage medium for high-speed flight data logging. These chips are optimized for sequential write operations and store critical flight parameters and telemetry data. SD cards function as secondary storage, implementing File Allocation Table File System (FATFS) for improved data accessibility.

G. User/Application

The User layer provides the interface between application-level flight software and the underlying RTOS services. This layer is specifically designed to support complex flight control algorithms. The API integrates high-level control algorithms such as an LQR or an Extended Kalman Filter with low-level system services such as sensor acquisition and interrupt handling. The tasking API forms the core of application development, providing functions for task creation, scheduling, and synchronization. Complex algorithms are implemented as individual tasks with specific timing and priority requirements.

The API's architecture supports integration of any high-level user program. The standardized interface allows developers to implement custom algorithms or control schemes without modifying the underlying RTOS. This flexibility enables the system to support various applications beyond TVC, such as data acquisition systems, test stands for static fires, or experimental control algorithms. Developers can utilize the same task management, timing control, and synchronization primitives while implementing their specific applications, making the RTOS adaptable to different subscale rocketry requirements.

VI. Integration

The RTOS was integrated and flight tested on a high-powered rocket in November 2024. The avionics system was built around an STM32H7 microcontroller featuring an ARM Cortex-M7 processor running at 224 MHz. This is due to its higher CPU frequency, low cost, and uniquity in avionics applications. The microcontroller utilized 1MB of SRAM and 2MB of internal flash memory. The system was organized into four primary tasks:

1) **Receive** Task: Receives data in DMA for ADCs, sensors, state estimation on an extended Kalman filter, and ground station commands

2) **Transmit** Task: Synchronizes all transmit calls to another microcontroller, data logging, and telemetry to the ground station
3) **Controls** Task: Executes the Linear Quadratic Regulator at 50Hz for TVC. The LQR updates occur within an (ISR) for guaranteed time-step precision [6]. The primary task handles the LQR computations.
4) **State Flash** Task: Flashes all sensor, controls, and state estimation data during flight onto an SD card and QSPI flash chip.

After flight, the system calculated the performance metrics using standard operating system benchmarking techniques. It determines memory utilization through custom linker script optimizations that tracked stack and heap allocations during compilation. Power consumption measurements came from 16-bit ADC readings using a dedicated current sensor on the VCC rail. The throughput figures reflect actual performance against the ARM Cortex-M7 specifications, measured in dynamic millions of instructions per second over clock frequency.

The performance parameters presented represent averaged specifications derived from in-flight telemetry, data logging, and benchmarking. This measures performance in CPU cycles rather than absolute time measurements since cycle counts provide a more direct correlation to the ARM Cortex M7 architecture and are not affected by clock frequency variations nor propagation delay from the setup that can occur during power state transitions. RTOSBench conducted post flight analysis on the processor.

Regarding threading-specific performance metrics, the benchmarking focused primarily on context switching efficiency and synchronization primitives, as these represent the most significant determinants of real-time performance.

Table 1 CPU Performance

Metric	Value
Throughput	5.05 DMIPS/MHz
Power Consumption	0.72W
Stack Memory Usage	15kB
Flash Usage	90kB
Scheduling Overhead	14 cycles
Context Switch Cycle	223 cycles
Interrupt Latency	101 cycles
Inter-Task Synchronization	969 cycles
Inter-Task Communication	923 cycles

The flight computer integrates an IMU, barometer, and GPS module, which is the basis of the state estimation algorithms. This prioritizes the IMU with the highest sampling rate to provide high-frequency inertial data essential for quaternion calculations and Kalman filter updates. The barometer operates at a moderate sampling rate with SPI transmission on DMA channels, providing redundant altitude measurements while minimizing CPU overhead. Operating at the lowest rate, the GPS module delivers position updates, with UART communication managed through DMA to efficiently handle the larger packet bursts characteristic of UBX. The state estimation is synchronized on the **Receive** thread to synchronize the state estimates with sensor readings and controller updates.

Table 2 Sensor Performance

Metric	IMU	Barometer	GPS
Type	ADIS16500	MS5607	ZED-F9P-04B
Protocol	SPI	SPI with DMA	UART with DMA
Speed	3MHz	1.5MHz	115200 baud
Sampling Rate	9.6kHz	2kHz	30Hz

For telemetry, the flight computer additionally includes a custom protocol over Zigbee/UART using XBee modules, with specialized packet framing and error handling. This custom protocol maintained reliable communication throughout the one-hour pre-flight operations and during the entire flight sequence, achieving almost no packet loss. The ground station received a continuous stream of 104-byte packets containing critical flight parameters, state estimates, and system health indicators. The **Transmit** and **Receive** threads center their operations around the XBee module.

Table 3 Telemetry Performance

Metric	Value
Data Packets Received	101,432
Packets Lost	3245 (3.2%)
Packet Size	104 bytes
Protocol	Zigbee over UART
Protocol Speed	57600 baud

The flight avionics storage system includes two systems for redundancy. The W25Q64JV NOR flash chip served as the primary flight recorder, storing raw sensor data, state estimates, controller outputs, and system events as binary streams using JFFS2. This approach maximized write performance while minimizing file system overhead. Post-flight, the RTOS transferred and converted this binary data to human-readable text files on an SD Card using FATFS. If this system used an SD card for the primary purpose, the latency would result in as much as 50 ms per packet burst. The majority of this operation is completed on the **State Flash** task.

Table 4 Storage Performance

Metric	NOR Flash	SD Card
Type	NOR	SDSC
File System	JFFS2	FATFS
Data Stored	32MB	128MB
Data Lost	0%	0%
Write Speed	32MB/s	500KB/s
Latency	100 ns	50 ms

VII. Analysis

A. Real-Time Control Performance

The system results validate the effectiveness of this implementation across multiple performance dimensions. The CPU demonstrated efficient resource utilization, with the system achieving 5.05 DMIPS/MHz while maintaining less than 1W of power consumption, more than double the predicted 2.14 DMIPS/MHz from the STM32H7 datasheet. Regarding the thread latency, all threads and metrics include submicrosecond latency on a 224MHz clock. The deterministic timing guarantees provided by the kernel layer enabled stable control despite varying system loads and position-velocity estimates. Maximum actuation deviations were limited to 48 degrees during rocket acceleration.

B. Resource Utilization

Throughout the flight, the system maintained consistent resource utilization with memory utilization remaining stable at 15% of available RAM; the remaining headroom accommodating stack growth during computationally intensive operations. While this RTOS provides larger overhead than sequential programming approaches, this system remained under the available RAM.

C. Telemetry and Data Logging Performance

The telemetry system transmitted 48 parameters with only 3.2% packet loss throughout the flight, significantly outperforming previous high-powered launches that experienced up to 25% packet loss using sequential approaches. The DMA-based data acquisition and transmission, managed by dedicated tasks, maintained consistent bandwidth without impacting control system and state estimation performance. Flight data logging captured data without a single missed data point, providing comprehensive data for post-flight analysis.

D. Comparative Analysis

When compared to traditional sequential programming approaches commonly used in amateur rocketry, this RTOS implementation demonstrated several advantages.

Table 5 RTOS vs. Sequential Implementation

Metric	GNC RTOS	Sequential Approach
Telemetry Latency	3ms	50ms
Packet Loss	3.2%	25%
Simultaneous Tasks	4 prioritized tasks	Sequential operation

Additionally, a benchmark comparison with Zephyr RTOS was conducted to assess relative performance. RTOSBench conducted and returned the specific criteria below.

Table 6 Performance Comparison: RTOS vs. Zephyr

Metric	GNC RTOS	Zephyr RTOS
Context Switch Cycles	223	524
Interrupt Latency Cycles	101	143
Mutex Cycles	1964	969
Message Queue Cycles	1058	923

The lower context switching and interrupt latency in this RTOS make it better suited for real-time flight control, where fast task preemptions are critical. Meanwhile, Zephyr's more efficient mutex and message queue handling makes it better suited for multicore applications. However, in the context of TVC rocketry, the ability to respond to rapid sensor updates and TVC actuation makes this RTOS implementation more appropriate for mission-critical avionics applications.

VIII. Conclusion

This study demonstrates that a well-optimized bare-metal RTOS can significantly improve the performance of avionics systems. The flight results confirm that properly implemented real-time operating systems can provide substantial benefits for subscale rocket applications. The submicrosecond context switching and interrupt latency enabled precise control timing without compromising overall system performance. The RTOS results highlight the improvements in complex software architectures that support TVC actuation, telemetry, state estimation, and data logging. Future work should focus on extending this architecture to support multi-core processing, which could further improve performance by allowing true parallelism between tasks.

Acknowledgments

The author would like to acknowledge Amazon Web Services for the open-source development for FreeRTOS. The content is heavily inspired by the approach taken in FreeRTOS. Additionally, the author would like to thank the Guidance, Navigation, and Control project at the Ramblin' Rocket Club at Georgia Tech for the assistance in the development of the high powered rocket's flight computer.

References

[1] Q. Li and C. Yao, "Real-Time Concepts for Embedded Systems," CRC Press, 2020.

[2] T. Noergaard, "Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers," Elsevier, 2012.

[3] STMicroelectronics, "STM32H7 Series Reference Manual," STMicroelectronics, 2020.

[4] E. A. Lee and S. A. Seshia, "Introduction to Embedded Systems: A Cyber-Physical Systems Approach," MIT Press, 2017.

[5] R. Barry, "FreeRTOS—Reference Manual: API Functions and Configuration Options," Amazon Web Services, 2022.

[6] C. Anderson and M. Lewis, "Modern Avionics Control Systems: Architecture, Design, Implementation," Springer, 2022.