NAVR BOTEC

Man meets machine

Vayu Flight Control Stack

Technical Report and System Documentation

Prepared by:
Ashutosh Vishwakarma

2026

v0.0.1

Contents

Abstract

This report presents an end-to-end flight control stack developed on top of the hardware-agnostic infrastructure, NavHAL and VAIOS. In an era of rapid technological advancement, the demand for autonomous systems has increased significantly. Autonomous systems—capable of operating independently, making decisions without human intervention—represent a crucial direction for the future.

We aim to contribute to this evolving domain by focusing on the foundational aspects of flight control systems. We believe that progress in autonomy cannot be decoupled from the underlying low-level mechanisms of flight control stacks. To meet emerging requirements, it is essential to design and adapt these low-level systems with precision and efficiency.

Existing open-source architectures often introduce unnecessary complexity due to their broad scope and general-purpose design. In this project, Vayu, we propose a streamlined and efficient base layer tailored for high-performance autonomous aerial systems. Our approach emphasizes modularity, efficiency, and adaptability, providing a robust foundation for future developments in autonomous flight.

Chapter 1

Introduction

The development of this work is rooted in early exposure to embedded systems and aerial robotics during undergraduate studies at the Indian Institute of Technology (IIT) Jammu. Initial hands-on experience involved assembling and operating multiple drone platforms. While these systems enabled rapid prototyping, they largely relied on integrating existing components rather than building or understanding the system from first principles. This distinction highlighted a key limitation: the inability to fully control or adapt the underlying software stack.

During the later stages of the second year, an attempt was made to implement encrypted telemetry over low-frequency RF communication. Standard protocols such as MAVLink do not natively support secure communication, which required introducing additional hardware and software layers. The implemented solution involved routing telemetry data through UART to an ESP32 for AES-based encryption, transmitting over LoRa, and performing decryption on the receiving end before forwarding data to a ground station.

Although functional, this approach introduced significant complexity, latency, and development overhead. A feature that ideally should be integrated within the system required multiple intermediate components, increasing cost and points of failure. This experience highlighted a broader issue: existing drone software stacks are not designed for deep customization, lightweight integration of advanced features, or complete system control.

This realization led to a key insight. If implementing a relatively contained feature such as secure telemetry required substantial effort, the challenges faced by small teams or early-stage drone companies would be significantly greater. In practice, many such teams depend on large, externally developed platforms such as PX4 and ArduPilot, often modifying them to suit specific requirements. While these platforms provide flexibility, they introduce dependency on foreign-developed systems, increase system complexity, and require sustained engineering effort for customization and maintenance.

In addition, the lack of a widely adopted, publicly accessible indigenous drone flight stack in India presents both a challenge and an opportunity. Existing solutions are either highly abstracted or tightly coupled to specific hardware and software ecosystems, limiting their adaptability for emerging applications in robotics and autonomous systems. This dependency becomes particularly critical in domains such as defense, where reliance on external technology stacks is undesirable.

Another key limitation of current systems is hardware lock-in. Developing new hard-

ware alongside a compatible firmware stack is a complex and resource-intensive process, especially for smaller teams. This barrier can slow innovation and restrict design flexibility.

Motivated by these challenges, the Vayu project was initiated in June 2025 with the goal of designing a complete flight control stack from first principles. The objective is to achieve vertical integration across all layers of the system, from hardware abstraction to high-level control logic, while maintaining modularity and performance.

A key design philosophy of this work is to minimize dependency on specific microcontroller families, external libraries, and monolithic frameworks. By maintaining control over all layers of the stack, the system aims to provide improved flexibility, portability, and long-term scalability.

The Vayu flight control stack is designed not only as a drone control system but also as a foundational platform for broader robotics and autonomous systems. By providing a structured and maintainable base layer, it enables developers and manufacturers to focus on application-driven innovation without being constrained by low-level system complexities.

We envision a future driven by autonomous systems capable of human-independent decision-making. Achieving this vision requires control over the foundational layers of the technology stack. This work represents an effort to move beyond assembly-based development toward building complete systems with full ownership of architecture and behavior.

The remainder of this report is structured as follows. Chapter 2 discusses existing systems and their limitations. Chapter 3 presents the proposed system architecture. Chapter 4 describes the hardware platform, followed by Chapters 5 and 6, which detail the design of NavHAL and VAIOS. Finally, Chapter 7 presents the Vayu flight control stack.

Chapter 2

Preliminaries

This chapter provides an overview of the fundamental concepts and system components required to understand the subsequent technical details of the proposed system.

The reader is assumed to have a basic familiarity with the domain, as it is not feasible to cover all topics in depth within this chapter. Instead, we present concise discussions of key concepts to establish the necessary background and ensure clarity in later sections.

This chapter is organized into nine sections, which can be broadly categorized into five functional layers:

1. Sections 1–3: Foundation Layer
2. Sections 4–5: Perception Layer
3. Sections 6–7: Control and Actuation
4. Section 8: Communication Layer
5. Section 9: Safety and Failsafe Mechanisms

2.1 Embedded System

An embedded system is a specialized computing system designed to perform a specific set of functions within a larger system. Unlike general-purpose computing systems, embedded systems are tailored for dedicated tasks and are often tightly coupled with the hardware they control. These systems are commonly found in domains such as robotics, automotive systems, consumer electronics, and aerospace applications.

Embedded systems are characterized by several key properties. First, they are typically *resource-constrained*, operating with limited processing power, memory, and energy availability. This necessitates efficient software design and careful resource management. Second, they often operate under *real-time constraints*, where tasks must be completed within strict timing deadlines to ensure correct system behavior. In control systems such as flight controllers, this includes maintaining consistent control loop frequencies and minimizing timing jitter.

Another important characteristic is *deterministic execution*. Embedded systems are expected to behave predictably under defined conditions, with minimal variation in execution timing. This is particularly critical in safety-sensitive applications, where unpredictable behavior can lead to system instability or failure.

Embedded systems are also *hardware-dependent*, meaning that the software is closely tied to the underlying microcontroller architecture and peripheral interfaces. This tight coupling can make portability across different hardware platforms challenging without appropriate abstraction layers.

Finally, embedded systems are often designed for *continuous operation* and must be robust against faults. They are expected to operate reliably over extended periods, often in constrained or harsh environments, with minimal human intervention.

These characteristics collectively influence the design and implementation of embedded software systems, particularly in applications requiring high reliability and real-time performance.

2.1.1 Typical Embedded System Architecture

A typical embedded system is organized around a microcontroller unit (MCU) that integrates processing, memory, and peripheral interfaces to interact with the physical environment. The architecture is designed to efficiently acquire data from sensors, process it using application-specific logic, and generate outputs to drive actuators, all under strict resource and timing constraints.

At its core, the microcontroller executes program instructions stored in non-volatile memory (Flash), while runtime data is maintained in volatile memory (SRAM). The MCU provides a variety of on-chip peripherals that enable interaction with external components. These include communication interfaces such as I2C, SPI, and UART, general-purpose input/output (GPIO), timers, analog-to-digital converters (ADC), and pulse-width modulation (PWM) units.

Sensors interface with the system through these peripherals, providing measurements of physical quantities such as acceleration, angular velocity, pressure, and position. The embedded software processes this data to estimate system state and compute appropriate control actions. These outputs are then applied to actuators, such as motors and servos, typically through PWM signals or other hardware-driven interfaces.

To support efficient and responsive operation, embedded systems rely on mechanisms such as interrupts and direct memory access (DMA). Interrupts allow the processor to respond to asynchronous events with low latency, while DMA enables high-throughput data transfers without continuous CPU involvement. These mechanisms are essential for maintaining deterministic behavior and meeting real-time requirements.

While this architecture provides a flexible and powerful interface to hardware, it also introduces significant hardware dependency. Software developed for a specific microcontroller is often tightly coupled to its peripheral set, register layout, and communication interfaces. This tight coupling makes portability across different hardware platforms challenging and increases development complexity.

To address these challenges, modern embedded systems adopt structured abstraction layers that decouple application logic from hardware-specific details. This approach enables portability, improves maintainability, and simplifies system design. The concept and role of such abstraction are discussed in the section 2.2 on Hardware Abstraction Layers (HAL).

Figure 2.1 illustrates the internal architecture and peripheral interconnections within a representative microcontroller unit. It is important to note that this diagram is a high-level representation and is specific to a particular class of MCUs (e.g., STM32). The exact

architecture may vary across different microcontroller families.

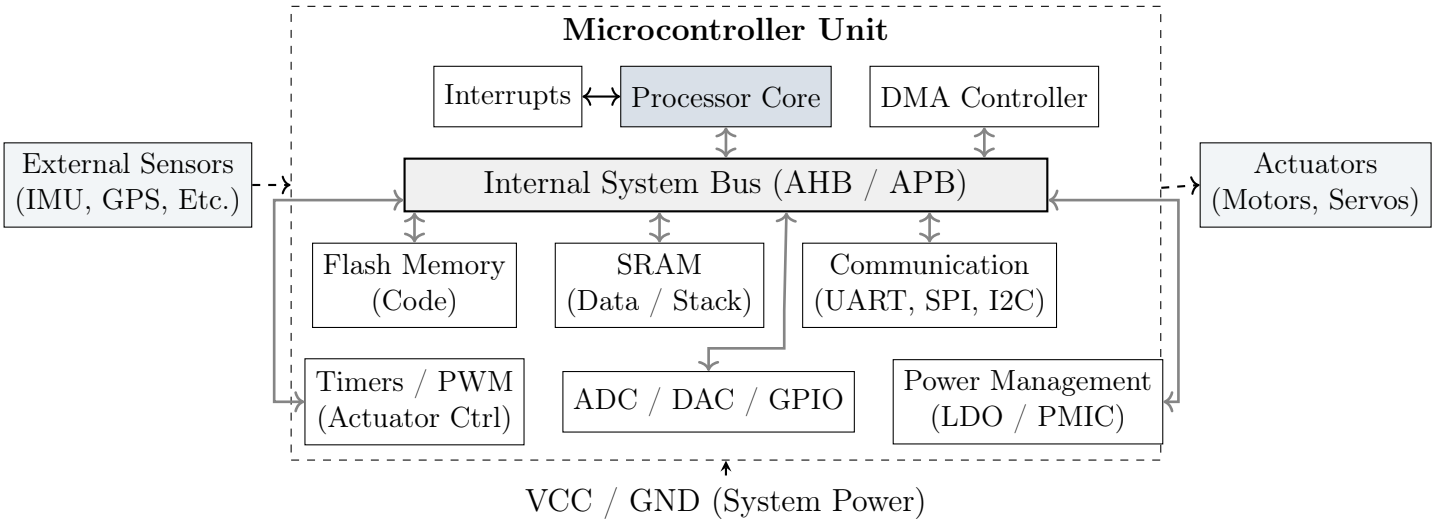

Figure 2.1: Internal architecture and peripheral interconnections of a representative microcontroller unit

2.1.2 Execution Model

The execution model of an embedded system defines how software tasks are scheduled and executed over time to interact with hardware and meet system requirements. Unlike general-purpose computing systems, embedded systems must operate under strict timing constraints, where predictable and deterministic execution is essential for correct system behavior.

At the simplest level, embedded systems may follow a *bare-metal* execution model, where the application runs in a continuous loop, often referred to as the superloop. In this model, tasks such as sensor reading, state estimation, control computation, and actuator updates are executed sequentially. While this approach is simple and efficient, it becomes difficult to manage as system complexity increases, especially when multiple tasks with different timing requirements must be handled.

To improve responsiveness, embedded systems rely on *interrupt-driven execution*. Interrupts allow the processor to temporarily suspend the current task and execute a specific handler in response to asynchronous events such as sensor data availability, communication requests, or timer triggers. This mechanism enables low-latency response to critical events and is fundamental to real-time embedded systems.

A key requirement in many embedded applications, particularly in control systems, is *deterministic timing*. Tasks must be executed at well-defined intervals, such as fixed-frequency control loops (e.g., hundreds of Hz to several kHz). Variations in execution timing, known as jitter, can degrade system performance and, in extreme cases, lead to instability.

As system complexity grows, managing multiple concurrent activities using only loops and interrupts becomes increasingly challenging. This leads to the adoption of structured scheduling mechanisms, where tasks are organized based on priority and timing requirements. These systems may implement cooperative or preemptive scheduling to ensure that critical tasks are executed within their deadlines.

Real-time operating systems (RTOS) provide a formal framework for such execution models. They enable task scheduling, synchronization, and resource management in a controlled and predictable manner. RTOS-based systems allow developers to define multiple tasks with specific priorities, ensuring that time-critical operations such as control loops and sensor processing are executed reliably.

The choice of execution model has a direct impact on system performance, responsiveness, and maintainability. In high-performance embedded applications such as flight control systems, a well-designed execution model is essential to ensure stable control, efficient resource utilization, and scalability as system complexity increases. The role of RTOS in enabling such structured execution is discussed in the section 2.3.

2.1.3 I/O and Peripheral Interfaces

Input/Output (I/O) and peripheral interfaces form the primary means by which an embedded system interacts with the external world. These interfaces enable the acquisition of sensor data and the transmission of control signals to actuators, making them fundamental to the operation of any embedded control system.

Microcontrollers integrate a variety of on-chip peripherals to support communication and signal processing. Common digital communication interfaces include Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI), and Universal Asynchronous Receiver-Transmitter (UART). These interfaces are widely used to connect sensors such as inertial measurement units (IMUs), barometers, and GPS modules, as well as external devices for communication and debugging.

In addition to digital interfaces, microcontrollers provide analog peripherals such as Analog-to-Digital Converters (ADC) and Digital-to-Analog Converters (DAC). ADCs are used to convert analog sensor signals into digital values that can be processed by the system, while DACs (less commonly used in control systems) enable the generation of analog output signals.

General-purpose input/output (GPIO) pins provide flexible digital interfacing capabilities, allowing the system to read binary inputs or control external devices. Timers and pulse-width modulation (PWM) units are particularly important in control applications, where they are used to generate precise timing signals and drive actuators such as motors and servos.

Efficient use of these peripherals often involves hardware features such as interrupts and Direct Memory Access (DMA), which enable low-latency and high-throughput data transfer without excessive CPU intervention. This is especially important in systems that require high-frequency data acquisition and real-time response.

Despite their versatility, peripheral interfaces are inherently hardware-specific. Differences in register layouts, configuration procedures, and peripheral capabilities across microcontroller families introduce significant complexity in software development. As a result, directly interacting with peripherals can lead to tightly coupled and non-portable code.

2.1.4 Limitations and Design Challenges

Despite their efficiency and suitability for dedicated tasks, embedded systems present several inherent limitations and design challenges. These constraints significantly influence

system architecture, software design, and overall reliability.

One of the primary limitations of embedded systems is their *resource-constrained nature*. Microcontrollers typically have limited processing capability, memory, and storage compared to general-purpose computing systems. This requires careful optimization of both software and hardware usage to ensure efficient operation without exceeding available resources.

Another critical challenge is meeting *real-time requirements*. Many embedded applications, particularly control systems, depend on precise timing guarantees. Tasks such as sensor acquisition, state estimation, and control computation must be executed within strict deadlines. Any delay or variation in execution timing (jitter) can degrade system performance and may lead to instability in closed-loop systems.

Hardware dependency further complicates embedded system development. Software is often tightly coupled to specific microcontroller architectures, peripheral configurations, and register-level implementations. This lack of portability makes it difficult to reuse code across different hardware platforms and increases development effort when adapting systems to new devices.

Concurrency and synchronization also pose significant challenges. Embedded systems frequently handle multiple tasks such as communication, sensing, and control simultaneously. Managing these tasks efficiently without introducing race conditions, deadlocks, or priority inversion requires careful system design, particularly in real-time environments.

Another limitation is the difficulty of *debugging and testing*. Embedded systems operate in constrained and often hardware-dependent environments, making traditional debugging techniques less effective. Observability is limited, and reproducing faults—especially those related to timing or concurrency—can be challenging.

Finally, embedded systems must ensure *robustness and reliability*, often operating continuously in real-world conditions with minimal supervision. Fault tolerance, safe failure modes, and recovery mechanisms are essential, particularly in safety-critical applications such as autonomous systems.

These challenges highlight the need for structured system design approaches that improve modularity, portability, and maintainability. Abstraction mechanisms, standardized interfaces, and well-defined execution models play a crucial role in addressing these issues. In particular, hardware abstraction layers provide a means to isolate application logic from hardware-specific complexity, enabling scalable and adaptable system development.

2.2 Hardware Abstraction Layer

A Hardware Abstraction Layer (HAL) is a software layer that provides a uniform interface between application logic and the underlying hardware. It abstracts the details of microcontroller peripherals, communication interfaces, and hardware-specific configurations, enabling higher-level software components to operate independently of the target platform.

In embedded systems, direct interaction with hardware typically involves low-level operations such as register manipulation, peripheral configuration, and device-specific initialization. While this approach can yield high performance, it results in tightly coupled code that is difficult to maintain, reuse, and port across different hardware platforms. HAL addresses this challenge by encapsulating hardware-specific implementations behind well-defined interfaces.

A typical HAL exposes functionalities such as communication (e.g., I2C, SPI, UART), timing (timers, delays), digital and analog I/O (GPIO, ADC), and actuator control (PWM). These interfaces allow application-level modules to interact with hardware without requiring knowledge of the underlying microcontroller architecture. As a result, the same application logic can be reused across different platforms by modifying only the HAL implementation.

The role of HAL becomes increasingly important in complex systems where multiple peripherals and subsystems must operate in coordination. By providing a structured interface to hardware resources, HAL improves modularity and simplifies system integration. It also enables parallel development, where application logic and hardware-specific implementations can be developed and tested independently.

However, traditional HAL designs often introduce trade-offs. Many implementations are tightly coupled to a specific microcontroller family or vendor ecosystem, limiting portability across heterogeneous platforms. Additionally, overly generic abstractions may incur performance overhead or restrict access to advanced hardware features. In some cases, HAL implementations expose inconsistent interfaces, leading to fragmentation and increased integration complexity.

These limitations highlight the need for a carefully designed abstraction layer that balances portability, performance, and flexibility. An effective HAL should provide consistent and minimal interfaces, while still allowing access to hardware capabilities when required. It should also support scalability across different classes of microcontrollers and system configurations.

In the context of flight control systems, the HAL plays a critical role in isolating control logic, estimation algorithms, and communication modules from hardware-specific details. This separation is essential for enabling portability across different flight controller boards and for simplifying system evolution as hardware platforms change as shown in Figure 2.2.

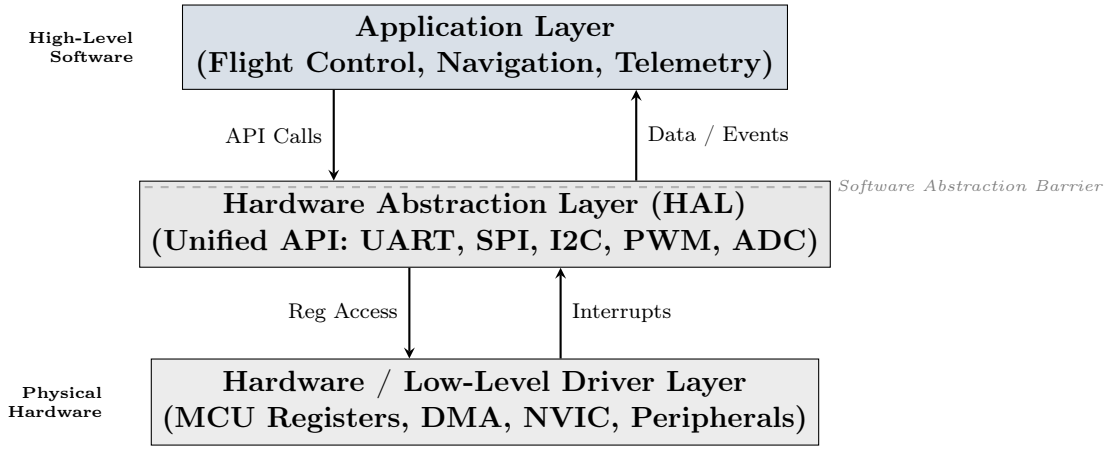

Figure 2.2: Layered architecture illustrating the role of HAL in decoupling application logic from hardware-specific implementations

The design and implementation of such a structured abstraction layer form a key component of the proposed system. In subsequent chapters, we introduce our approach to hardware abstraction, which aims to provide a lightweight, flexible, and hardware-agnostic interface tailored for high-performance embedded control systems.

2.3 Real-Time Operating System

A Real-Time Operating System (RTOS) is a software layer that manages the execution of tasks in an embedded system while ensuring that timing constraints are met. Unlike general-purpose operating systems, which prioritize throughput and user experience, an RTOS is designed to provide predictable and deterministic behavior, making it suitable for time-critical applications such as flight control systems.

In embedded systems, multiple tasks must often be executed concurrently, including sensor data acquisition, state estimation, control computation, and communication. An RTOS provides a structured framework to manage these activities through mechanisms such as task scheduling, synchronization, and inter-task communication. Each task can be assigned a priority and execution constraints, allowing the system to ensure that critical operations are performed within their required deadlines.

One of the core components of an RTOS is the *scheduler*, which determines the order and timing of task execution. Scheduling policies may be cooperative or preemptive. In cooperative scheduling, tasks voluntarily yield control, whereas in preemptive scheduling, higher-priority tasks can interrupt lower-priority ones to ensure timely execution. Preemptive scheduling is commonly used in real-time systems where strict timing guarantees are required.

An RTOS provides synchronization primitives such as mutexes, semaphores, and event flags to coordinate access to shared resources and enable communication between tasks. These mechanisms are essential to prevent issues such as race conditions, deadlocks, and priority inversion in concurrent systems.

Another important feature of RTOS is its support for precise timing operations. Timers and tick-based scheduling allow tasks to be executed periodically at fixed intervals, which is critical for control systems that rely on consistent loop frequencies. For example, flight control systems often require control loops to run at frequencies ranging from hundreds of Hertz to several kilohertz, with minimal timing jitter.

Despite these advantages, RTOS-based systems introduce their own challenges. Improper task design, priority misconfiguration, or excessive context switching can lead to performance degradation and unpredictability. Additionally, traditional RTOS implementations may impose overhead and complexity that are not always necessary for simpler systems.

In the context of embedded control systems, the RTOS serves as the execution backbone, coordinating multiple subsystems while ensuring deterministic behavior. However, achieving the right balance between flexibility, performance, and simplicity requires careful design of the execution model and task structure.

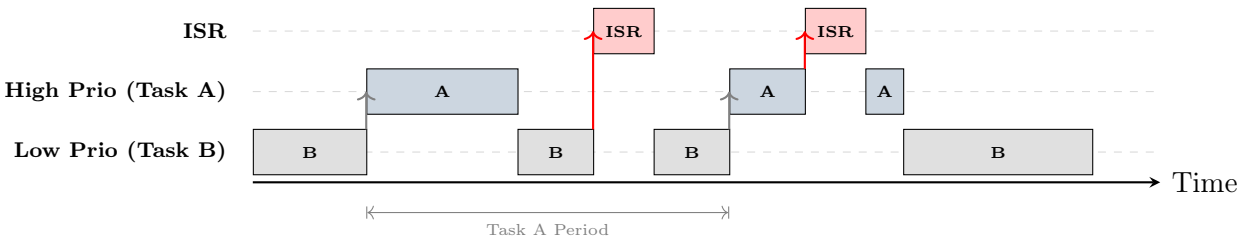

Figure 2.3: RTOS scheduling timeline illustrating task preemption and periodic execution driven by priority and hardware interrupts

The role of RTOS in structuring system execution and enabling predictable behavior forms a critical component of the overall system architecture. In subsequent chapters, we present our approach to system-level execution management, which builds upon these principles to provide a lightweight and scalable solution tailored for high-performance embedded applications.

2.4 Coordinate System

In embedded control systems, particularly in aerial robotics, the representation of position, orientation, and motion is fundamentally dependent on the choice of coordinate systems. A clear definition of coordinate frames and their transformations is essential for accurate state estimation and control.

2.4.1 Reference Frames

A *reference frame* defines a coordinate system with respect to which the position and orientation of an object are measured. In flight control systems, two primary reference frames are commonly used:

- **Inertial Frame (World Frame)**: A fixed reference frame, typically aligned with the Earth. In many applications, the North-East-Down (NED) convention is used, where:
  - X-axis points toward the North
  - Y-axis points toward the East
  - Z-axis points downward
- **Body Frame**: A moving reference frame attached to the vehicle. Its axes are defined relative to the drone:
  - X-axis points forward
  - Y-axis points to the right
  - Z-axis points downward

The body frame moves and rotates with the vehicle, while the inertial frame remains fixed. Sensor measurements are typically obtained in the body frame, whereas navigation and control objectives are often defined in the inertial frame.

2.4.2 Coordinate Transformations

To relate quantities between different frames, coordinate transformations are required. These transformations are typically represented using rotation matrices or equivalent representations. A rotation matrix $R \in SO(3)$ describes the orientation of the body frame with respect to the inertial frame and allows vectors to be transformed between frames.

If $\mathbf{v}_b$ represents a vector in the body frame, and $\mathbf{v}_i$ represents the same vector in the inertial frame, then:

$$\mathbf{v}_i = R\,\mathbf{v}_b$$

These transformations are essential for interpreting sensor data and applying control laws in a consistent reference frame.

2.4.3 Attitude Representation

The orientation of a rigid body in three-dimensional space, referred to as its *attitude*, can be represented using several methods. The most common representations include Euler angles, rotation matrices, and quaternions.

- **Euler Angles**: Orientation is described using three sequential rotations, typically roll, pitch, and yaw. While intuitive, Euler angles suffer from singularities (gimbal lock) and are not well-suited for continuous rotational dynamics.
- **Rotation Matrices**: A 3×3 orthonormal matrix representing orientation. Rotation matrices are free from singularities but require more computational resources and must maintain orthogonality.
- **Quaternions**: A compact and numerically stable representation of orientation using four parameters. Quaternions avoid singularities and are widely used in real-time systems for attitude estimation and control.

In practical flight control systems, quaternions are commonly used for internal computations due to their stability and efficiency, while Euler angles may be used for interpretation and user interaction.

2.4.4 Relevance to Control Systems

Accurate representation of coordinate frames and attitude is critical for transforming sensor measurements into meaningful state estimates. For example, inertial measurements obtained from an IMU in the body frame must be transformed into the inertial frame for tasks such as stabilization and navigation.

Similarly, control algorithms often compute desired orientations or angular rates in a specific frame, requiring consistent transformation between reference frames. Errors in these transformations can lead to incorrect state estimation and unstable control behavior.

Thus, a well-defined coordinate system framework forms the foundation for sensor fusion, state estimation, and control in embedded flight systems. Figure 2.4 illustrates the relationship between the fixed inertial frame and the drone's body frame, along with the convention for Euler angle rotations.

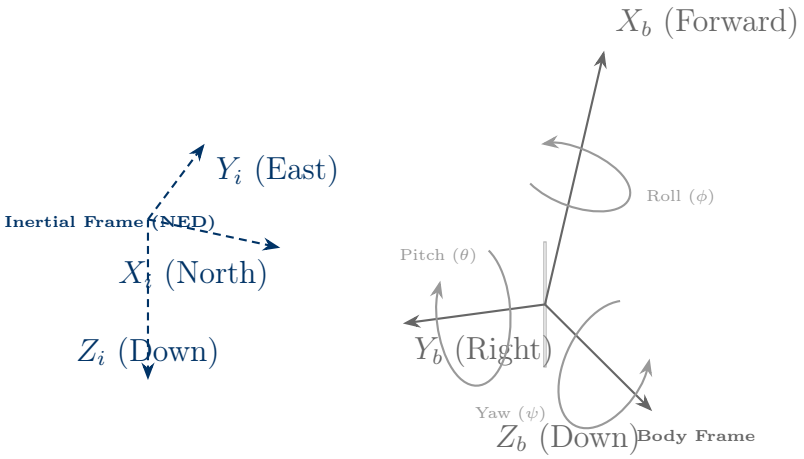

Figure 2.4: Refined 3D coordinate systems showing the Inertial (NED) and Body frames side-by-side.

2.5 Sensor

Sensors form the primary interface between an embedded system and the physical environment. In aerial robotics and embedded control systems, sensors provide measurements of physical quantities such as acceleration, angular velocity, magnetic field, pressure, and position. These measurements are essential for estimating the state of the system and enabling closed-loop control.

2.5.1 Types of Sensors

Flight control systems typically rely on a combination of sensors, each providing complementary information:

- **Inertial Measurement Unit (IMU)**: An IMU consists of accelerometers and gyroscopes. Accelerometers measure linear acceleration, including the effect of gravity, while gyroscopes measure angular velocity. IMUs provide high-frequency data and are fundamental for attitude estimation and stabilization.
- **Magnetometer**: Measures the Earth's magnetic field and is commonly used for heading (yaw) estimation. It helps correct long-term drift in orientation estimates derived from gyroscopes.
- **Barometer**: Measures atmospheric pressure and is used to estimate altitude. It provides relatively low-frequency but stable altitude information.
- **Global Positioning System (GPS)**: Provides global position and velocity estimates. GPS measurements are typically low-frequency and subject to noise and delay but are essential for navigation.

Each sensor has its own characteristics in terms of accuracy, noise, update rate, and reliability. As a result, no single sensor can provide a complete and accurate estimate of the system state.

2.5.2 Sensor Characteristics

Sensor measurements are inherently imperfect and are affected by various sources of error:

- **Noise**: Random variations in sensor readings, often modeled as stochastic processes. Noise limits the precision of measurements and must be filtered.
- **Bias**: A constant or slowly varying offset in sensor output. For example, gyroscope bias can lead to drift in orientation estimation over time.
- **Scale Factor Errors**: Deviations in the proportionality between the measured signal and the actual physical quantity.
- **Drift**: Accumulated error over time, particularly significant in integration-based measurements such as those from gyroscopes.

Understanding these characteristics is essential for designing reliable estimation and control algorithms.

2.5.3 Sensor Fusion

Due to the limitations of individual sensors, modern embedded systems employ *sensor fusion* techniques to combine measurements from multiple sources and obtain a more accurate and robust estimate of the system state.

In flight control systems, sensor fusion typically combines high-frequency IMU data with lower-frequency but more stable measurements such as those from magnetometers, barometers, or GPS. For example, gyroscope measurements provide short-term orientation changes, while accelerometer and magnetometer data help correct long-term drift.

Sensor fusion algorithms aim to balance responsiveness and stability by leveraging the strengths of different sensors. These algorithms may range from simple complementary filters to more advanced techniques such as Kalman filters and nonlinear observers. The general architecture of such a fusion system is illustrated in Figure 2.5.

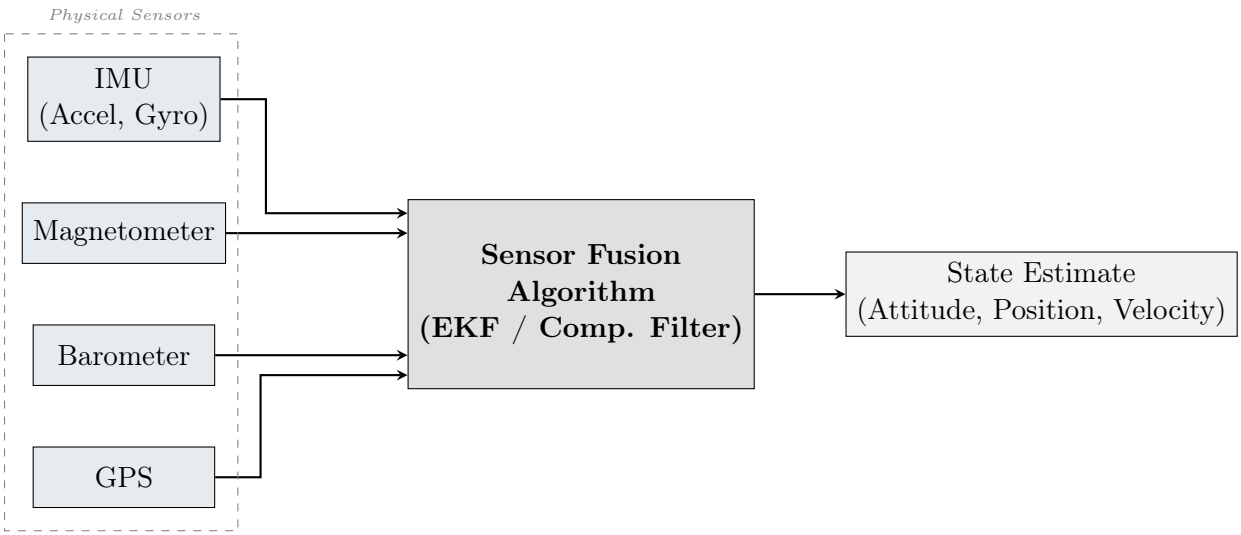

Figure 2.5: Architecture of a sensor fusion system combining disparate data sources into a unified state estimate

2.6 Control

Control systems are responsible for regulating the behavior of a system to achieve desired performance objectives. In embedded flight systems, control algorithms use estimated system states—such as orientation, angular velocity, and position—to compute actuator commands that stabilize and guide the vehicle.

2.6.1 Feedback Control

Most embedded control systems operate using *feedback control*, where the current state of the system is continuously measured and compared against a desired reference. The difference between the desired state and the measured state, known as the *error*, is used to compute corrective actions.

Feedback control enables the system to compensate for disturbances, model uncertainties, and sensor noise, making it essential for stable operation in dynamic environments.

2.6.2 Proportional-Integral-Derivative (PID) Control

One of the most widely used control strategies in embedded systems is the Proportional-Integral-Derivative (PID) controller. A PID controller computes the control input based on three components:

- **Proportional (P)**: Produces an output proportional to the current error. It provides immediate corrective action but may lead to steady-state error.
- **Integral (I)**: Accumulates the error over time to eliminate steady-state error. However, excessive integral action can lead to overshoot and instability.
- **Derivative (D)**: Responds to the rate of change of the error, providing damping and improving system stability. It is sensitive to noise and often requires filtering.

The control input $u(t)$ is given by:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt}$$

where $e(t)$ is the error and K_p, K_i, K_d are the controller gains.

2.6.3 Cascaded Control Architecture

In flight control systems, a single control loop is often insufficient to achieve stable and responsive behavior. Instead, a *cascaded control architecture* is used, where multiple control loops operate at different levels.

A typical structure consists of:

- **Outer Loop (Angle Control)**: Regulates the desired orientation (roll, pitch, yaw) by computing target angular rates.
- **Inner Loop (Rate Control)**: Regulates angular velocity using gyroscope measurements and generates actuator commands.

The inner loop operates at a higher frequency and provides fast response, while the outer loop operates at a lower frequency and ensures stability with respect to the desired orientation.

2.6.4 Actuator Command Generation

The outputs of the control system must be translated into actuator commands. In multi-rotor systems, this involves mapping control inputs (e.g., roll, pitch, yaw, and thrust) to individual motor outputs through a process known as *motor mixing*.

This mapping ensures that the combined effect of all actuators produces the desired forces and torques on the vehicle.

2.6.5 Practical Considerations

In real-world embedded systems, several practical factors influence control performance:

- **Sampling Rate**: Control loops must run at sufficiently high frequencies to ensure responsiveness and stability.
- **Noise and Filtering**: Sensor noise affects derivative terms and requires filtering techniques to maintain stability.
- **Saturation**: Actuator limits can lead to saturation, which may cause integral windup and degraded performance.
- **Tuning**: Proper tuning of controller gains is essential for achieving desired performance and stability.

Designing an effective control system requires balancing responsiveness, stability, and robustness under varying operating conditions. These principles form the foundation for the control strategies implemented in embedded flight systems.

2.7 Communication

Communication systems enable data exchange between different components of an embedded system as well as between the system and external entities such as ground control stations, companion computers, and other devices. In flight control systems, communication plays a crucial role in telemetry, command reception, configuration, and debugging.

2.7.1 Communication Interfaces

Embedded systems support a variety of communication interfaces, each suited for different use cases:

- **UART (Universal Asynchronous Receiver-Transmitter)**: A simple and widely used serial communication interface. UART is commonly used for telemetry, debugging, and communication with external modules such as GPS or radio transceivers.

- **SPI (Serial Peripheral Interface)**: A high-speed, synchronous communication protocol typically used for short-distance communication with sensors and peripherals such as IMUs.
- **I2C (Inter-Integrated Circuit)**: A multi-device communication protocol that allows multiple peripherals to share the same bus. It is commonly used for low-speed sensor communication.
- **CAN (Controller Area Network)**: A robust communication protocol designed for reliability in noisy environments. It is widely used in automotive and increasingly in drone systems for inter-module communication.

Each interface presents trade-offs in terms of speed, complexity, reliability, and wiring requirements, and the choice of interface depends on system constraints and application needs.

2.7.2 Communication Protocols

Beyond physical interfaces, communication protocols define how data is structured, transmitted, and interpreted. In flight control systems, protocols are used to standardize communication between components.

Protocols typically define:

- Message formats and encoding
- Packet structure and framing
- Error detection (e.g., checksums or CRC)
- Synchronization and sequencing

Well-defined protocols ensure interoperability between different system components and enable reliable communication even in the presence of noise or packet loss.

2.7.3 Telemetry and Command Systems

Communication in embedded flight systems is broadly divided into two categories:

- **Telemetry**: Transmission of system data such as sensor readings, state estimates, and system status to external systems. This is essential for monitoring and debugging.
- **Command and Control**: Reception of external inputs such as pilot commands, mission instructions, or configuration parameters. These inputs directly influence system behavior and must be handled with low latency and high reliability.

These communication pathways must be carefully designed to ensure timely and consistent data exchange, especially in real-time systems.

2.7.4 Design Considerations

Designing communication systems for embedded applications involves several practical considerations:

- **Latency**: Communication delays can affect system responsiveness, particularly in control loops.
- **Reliability**: Error detection and recovery mechanisms are essential for maintaining data integrity.
- **Bandwidth**: Limited communication bandwidth requires efficient encoding and prioritization of data.
- **Scalability**: Systems should support the addition of new devices and interfaces without significant redesign.

Effective communication design ensures that data flows reliably across the system, enabling coordination between sensing, control, and external interaction layers.

2.8 Safety

Safety is a critical aspect of embedded control systems, particularly in applications such as aerial robotics where system failures can lead to physical damage or hazardous situations. A robust system must be capable of detecting abnormal conditions and responding in a controlled manner to prevent escalation.

2.8.1 Failure Modes

Embedded flight systems can encounter various types of failures, including:

- **Sensor Failures**: Faulty or inconsistent sensor readings due to hardware malfunction, noise, or environmental interference.
- **Actuator Failures**: Loss of control over motors or servos, which can directly affect system stability.
- **Communication Loss**: Interruption of communication with remote controllers or ground stations.
- **Software Faults**: Unexpected behavior due to bugs, race conditions, or incorrect state estimation.

Identifying these failure modes is the first step toward designing effective safety mechanisms.

2.8.2 Fault Detection

To ensure safe operation, the system must continuously monitor its internal state and sensor inputs. Fault detection mechanisms may include:

- Threshold-based checks on sensor values and system states
- Consistency checks between multiple sensors (e.g., comparing accelerometer and gyroscope estimates)
- Monitoring of communication timeouts
- Detection of actuator saturation or abnormal control outputs

Early detection of anomalies allows the system to take corrective action before instability occurs.

2.8.3 Failsafe Mechanisms

Failsafe mechanisms define how the system responds when a fault is detected. These responses are designed to minimize risk and maintain system safety.

Common failsafe strategies include:

- **Disarm**: Immediately disable actuators to prevent unintended motion.
- **Hover / Stabilize**: Maintain a stable state using available sensor data when partial failures occur.
- **Return-to-Home (RTH)**: Navigate the system back to a predefined safe location in case of communication loss.
- **Controlled Shutdown**: Gradually reduce system activity to avoid abrupt transitions.

The choice of failsafe strategy depends on system capabilities and the nature of the detected fault.

2.8.4 Design Considerations

Designing safety mechanisms involves balancing responsiveness and reliability. Key considerations include:

- **Latency of Detection**: Faults must be detected quickly to prevent escalation.
- **False Positives**: Overly sensitive detection may trigger unnecessary failsafe actions.
- **System Redundancy**: Multiple sensors or fallback mechanisms can improve reliability.
- **Graceful Degradation**: The system should maintain partial functionality when possible, rather than failing abruptly.

A well-designed safety framework ensures that the system can handle unexpected conditions in a predictable and controlled manner, forming an essential component of reliable embedded systems.

Chapter 3

System Architecture

This chapter presents the overall system architecture of the Vayu flight control stack. Building upon the foundational concepts introduced in the previous chapter, we now describe how these elements are integrated into a cohesive and structured system.

The architecture is designed with a focus on modularity, hardware independence, and deterministic execution. It organizes the system into well-defined layers, each responsible for a specific aspect of functionality, ranging from low-level hardware interaction to high-level control logic.

In this chapter, we first provide an overview of the system in Section 3.1, followed by a detailed description of its layered design in Section 3.2, core components in Section 3.3, data flow in Section 3.4, and execution model in Section 3.5. We also discuss the key design principles that guided the development of the system in Section 3.6 and compare our approach with existing solutions in Section 3.7. This structured view establishes a clear understanding of how the Vayu stack operates as an integrated whole.

3.1 Overview

The Vayu flight control stack is designed as a vertically integrated system that spans multiple layers of abstraction, from low-level hardware interaction to high-level control logic. The primary goal of the architecture is to provide a modular, hardware-agnostic, and efficient framework for developing embedded flight systems, while maintaining strict control over execution behavior and system performance.

At a high level, the system is organized into distinct layers, each responsible for a specific aspect of functionality. The lowest layer consists of the physical hardware, including the microcontroller, sensors, and actuators. Above this lies the Hardware Abstraction Layer (NavHAL), which provides a uniform interface to hardware peripherals, enabling portability across different platforms. The execution layer (VAIOS) manages task scheduling, timing, and system coordination, ensuring deterministic and predictable behavior. On top of this foundation resides the Vayu control stack, which implements state estimation, control algorithms, and actuator command generation.

The system follows a clear data flow pipeline, where sensor measurements are acquired from hardware, processed through estimation algorithms, and used by control modules to compute actuator outputs. This structured flow ensures that each stage of processing is well-defined and isolated, improving both maintainability and system clarity.

A key characteristic of the architecture is the separation of concerns between hardware interaction, execution management, and control logic. This separation allows each layer to evolve independently while maintaining a consistent interface with adjacent layers. As a result, the system can be adapted to new hardware platforms or extended with additional functionality without requiring significant changes to core components.

Furthermore, the architecture is designed with real-time constraints in mind. High-frequency control loops, low-latency sensor processing, and reliable communication are all supported through a combination of efficient abstraction and structured execution.

This overview establishes the fundamental structure of the Vayu system. The following sections provide a detailed breakdown of each layer, the interactions between components, and the design decisions that enable a scalable and high-performance flight control stack.

3.2 Layered Architecture

The Vayu system is structured as a layered architecture, where each layer encapsulates a specific set of responsibilities and interacts with adjacent layers through well-defined interfaces. This design promotes modularity, portability, and maintainability, while enabling precise control over system behavior.

At a high level, the architecture is composed of four primary layers:

- **Hardware Layer**: This layer consists of the physical components of the system, including the microcontroller, sensors (e.g., IMU, GPS, barometer), and actuators (e.g., motors, servos). It provides the raw interface to the physical world.
- **Hardware Abstraction Layer (NavHAL)**: NavHAL provides a uniform interface to hardware peripherals such as communication interfaces, timers, GPIO, and sensor drivers. It abstracts hardware-specific details, enabling the upper layers to remain independent of the underlying platform.
- **Execution Layer (VAIOS)**: VAIOS is responsible for managing system execution, including task scheduling, timing, and coordination between different subsystems. It ensures deterministic behavior by organizing tasks according to their timing and priority requirements.
- **Control Layer (Vayu)**: The top layer implements the core flight control logic, including sensor fusion, state estimation, control algorithms, and actuator command generation. It operates on abstracted interfaces provided by the lower layers and remains independent of hardware-specific details.

These layers interact in a hierarchical manner, where each layer depends only on the services provided by the layer immediately below it. This separation of concerns allows changes in one layer—such as adapting to new hardware or modifying execution strategies—without affecting the functionality of other layers.

The layered architecture also enables clear data and control flow across the system. Sensor data is acquired at the hardware level, abstracted by NavHAL, processed within the control layer, and translated into actuator commands that propagate back down to the hardware.

Figure 3.1 illustrates the high-level layered structure of the Vayu system.

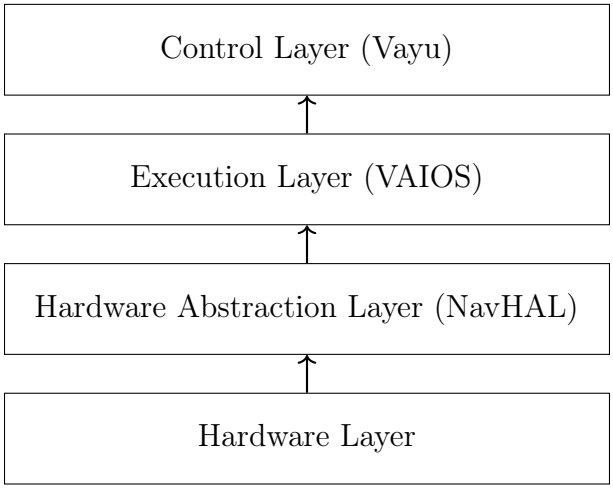

Figure 3.1: Layered architecture of the Vayu flight control system

This layered design provides a clear separation between hardware interaction, execution management, and control logic, forming a scalable and flexible foundation for embedded flight systems.

3.3 System Components

The stack is composed of a set of functional components that collectively process sensor data, estimate system state, and generate actuator commands. These components are organized to reflect the logical flow of information through the system while maintaining a clear separation of responsibilities.

At a high level, the system consists of the following core modules:

- **Sensor Interface (BMX160)**: This module initializes and manages the 6-axis IMU (accelerometer and gyroscope) along with the magnetometer. It acquires raw sensor measurements at high frequency (> 1.5 kHz) and provides time-consistent data to the estimation module. Future iterations of the system will incorporate redundant IMU configurations to improve reliability and fault tolerance.
- **RC Command Interface (iBus)**: This module decodes input signals from the radio receiver (e.g., FlySky iBus protocol) and generates pilot-defined setpoints for roll, pitch, yaw, and thrust. These inputs act as external references for the control system. Support for additional protocols such as PPM and SBUS is planned to improve compatibility with a wider range of receivers.
- **State Estimation (Sensor Fusion)**: The estimation module processes raw sensor data to compute the vehicle's attitude. Lightweight nonlinear observers, such as Mahony or complementary filters, are employed to fuse gyroscope and accelerometer measurements, ensuring robustness against noise and drift. The architecture is designed to accommodate more advanced estimation techniques, such as Extended Kalman Filters, in future iterations.
- **Cascaded PID Control**: The control system is implemented as a cascaded architecture consisting of an outer angle loop and an inner rate loop. The outer loop

regulates orientation, while the inner loop regulates angular velocity, enabling fast response and stable control.

- **Actuation and Motor Output**: This module translates control outputs into actuator-specific commands. It applies a mixing strategy to map roll, pitch, yaw, and thrust commands to individual motor PWM signals, supporting configurations such as X4 multi-rotor systems.

These components operate as a coordinated pipeline within the VAIOS execution environment. Time-critical modules such as sensor acquisition, estimation, and control are executed deterministically, while auxiliary functionalities such as telemetry and monitoring operate asynchronously.

Figure 3.2 illustrates the logical organization and interaction of these components within the Vayu control stack.

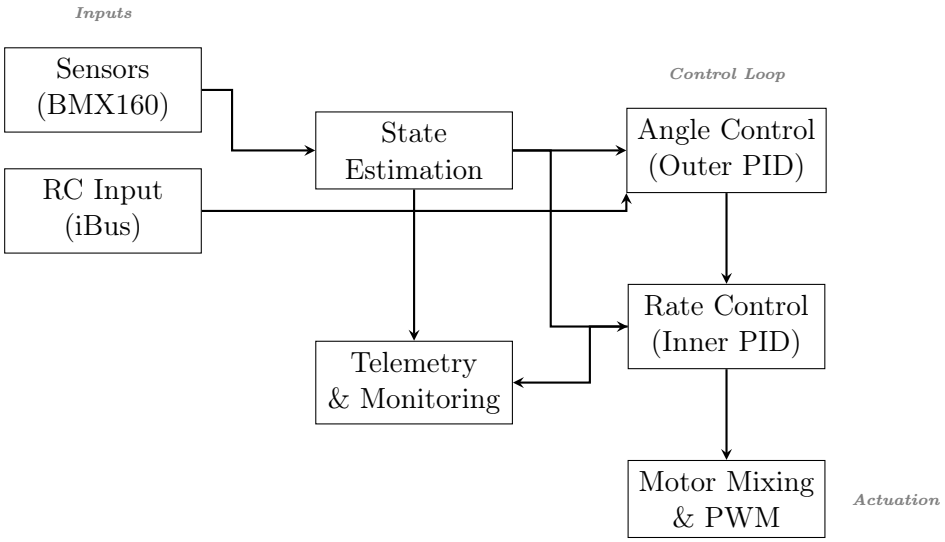

Figure 3.2: Logical organization and data flow between core components of the Vayu control stack.

This component-based organization enables a clear separation between sensing, estimation, control, and actuation, forming a scalable and maintainable foundation for embedded flight control systems.

3.4 Data Flow

The stack implements a deterministic, high-frequency data flow pipeline designed to minimize latency and ensure consistent control performance. Data moves through the system in a structured sequence of sensing, estimation, command processing, and actuation.

3.4.1 Data Processing Pipeline

The core data flow follows a strictly defined path as illustrated in Figure 3.3:

1. **Sensor Acquisition**: Raw measurements are read from the BMX160 IMU and Magnetometer. These values are buffered and low-pass filtered to remove high-frequency noise.

2. **State Estimation**: The filtered measurements are processed by the sensor fusion module (e.g., Mahony filter) to compute the vehicle's current orientation (Roll, Pitch, Yaw).

3. **Command Input Processing**: Concurrently, RC commands are received and decoded from the iBus receiver. These inputs define the desired setpoints for the vehicle's attitude and thrust.

4. **Cascaded PID Control**: The estimated state and reference setpoints are fed into the cascaded control loops. The difference (error) is processed through the outer angle loop and then the inner rate loop to generate torque commands.

5. **Actuator Mapping**: The torque commands are passed to the motor mixer, which calculates the required speed for each individual motor based on the airframe geometry.

6. **Hardware Output**: Final commands are converted into PWM signals and sent to the Electronic Speed Controllers (ESCs).

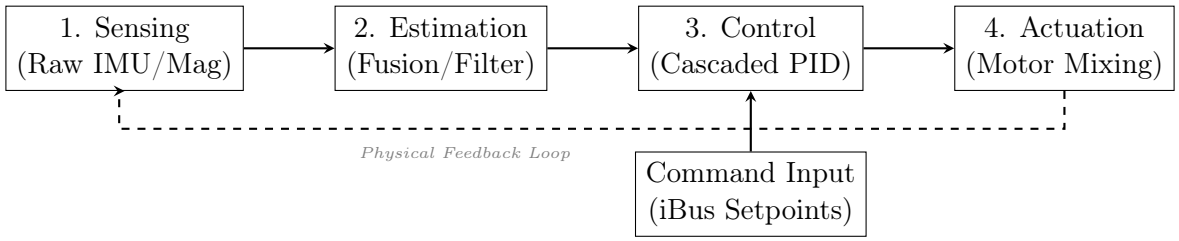

Figure 3.3: Sequential data flow pipeline of the Vayu system, from sensor acquisition to physical actuation.

3.4.2 Synchronization and Buffering

To maintain data integrity across different execution frequencies, Vayu utilizes structured buffers and shared variables managed by VAIOS. For instance, the IMU task updates its buffer at 1 kHz, while the telemetry task may sample this data at 100 Hz. Synchronization holds ensure that the control loops always operate on the most recent and coherent snapshot of the system state, preventing race conditions or data tearing.

The data flow begins with sensor acquisition at the hardware level. Sensors such as the IMU, magnetometer, barometer, and GPS generate measurements of physical quantities, which are accessed through the Hardware Abstraction Layer (NavHAL). These measurements are time-stamped and made available to the higher-level modules in a consistent format.

The acquired sensor data is then processed by the state estimation module. This stage combines measurements from multiple sensors to compute an estimate of the system state, including orientation, angular velocity, and position. High-frequency inertial

measurements provide short-term dynamics, while lower-frequency sensors contribute to long-term stability and drift correction.

The estimated state is used by the control module to compute control actions. Based on the desired reference inputs (e.g., target orientation or angular rates), the control system evaluates the error and generates appropriate control signals using cascaded control loops. The inner loops operate on angular velocity, while outer loops regulate orientation.

These control signals are then passed to the motor mixing module, which converts abstract control inputs (roll, pitch, yaw, and thrust) into actuator-specific commands. The resulting outputs are transmitted through NavHAL to the hardware layer, where they are applied to motors or other actuators.

In parallel, the communication module exchanges data with external systems. Telemetry data, including sensor readings and system states, is transmitted outward, while command inputs and configuration updates are received and integrated into the control pipeline.

This flow can be summarized as:

```
Sensors → Estimation → Control → Mixing → Actuation
```

Figure 3.4 illustrates the end-to-end data flow within the Vayu system.

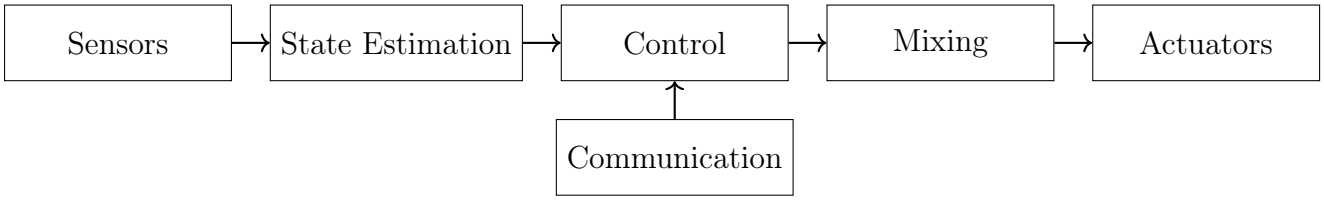

Figure 3.4: Data flow pipeline from sensor acquisition to actuator output

This structured data flow ensures a clear separation between different processing stages, enabling modular design and simplifying debugging and system analysis. Each stage operates on well-defined inputs and outputs, allowing independent development and optimization of system components.

3.5 Execution Model

The stack utilizes a multi-tasking execution model managed by the **VAIOS** (Vayu Input/Output System) real-time kernel. This model ensures that high-frequency control loops, asynchronous communication, and background maintenance tasks coexist on a single microcontroller while satisfying strict real-time constraints.

3.5.1 Task Scheduling and Priorities

The system organizes functionality into discrete tasks, each assigned a priority level and an execution frequency. VAIOS employs a priority-based preemptive scheduler, ensuring that critical tasks can interrupt lower-priority operations to meet timing guarantees.

The task hierarchy is structured into three primary tiers:

- **Tier 1: High-Priority Control (Priority 2)**: The `bmx160_read` task resides here. Since state estimation and control accuracy depend on precise sensor timing, sensor acquisition is given the highest precedence to minimize jitter.
- **Tier 2: Real-Time Processing (Priority 1)**: This tier contains the *Control Loops* (Angle and Rate PID) and the *Motor Output* task. These tasks run at fixed intervals (e.g., 400 Hz to 1 kHz) and consume the data provided by Tier 1.
- **Tier 3: Asynchronous Utilities (Priority 0)**: Lower-priority tasks such as `comm_processor` (telemetry), `heartbeat`, and `logger`. These tasks utilize the remaining CPU cycles to handle non-critical data exchange and system monitoring.

3.5.2 Execution Timing Diagram

Figure 3.5 illustrates the interleaved execution of high and low priority tasks over a single control cycle.

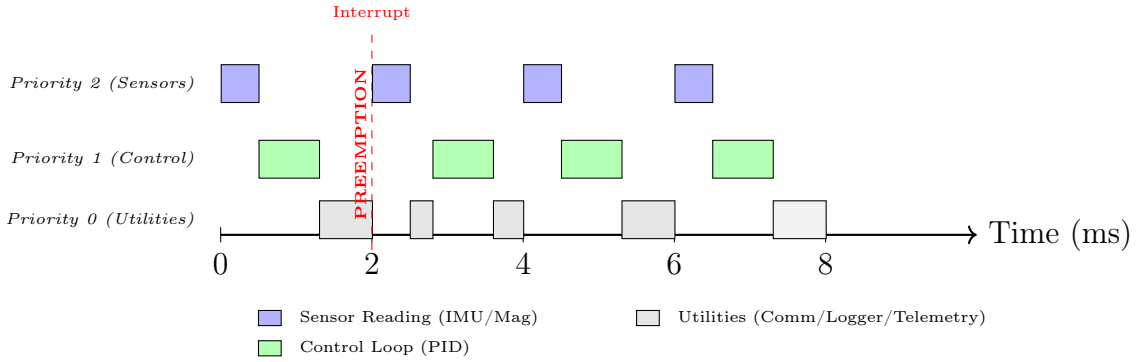

Figure 3.5: Preemptive scheduling model of VAIOS, showing how sensor tasks interrupt lower-priority execution to ensure deterministic timing.

3.5.3 Deterministic Guarantees

By utilizing a hardware-backed DWT (Data Watchpoint and Trace) timer and the VAIOS scheduler, the system achieves sub-microsecond jitter for task wakeup. This determinism is critical for the stability of the inner PID loop, where even a few milliseconds of variance in the sampling interval (dt) can lead to oscillations or loss of control during aggressive maneuvers.

3.6 Design Principles

The stack is designed around a set of core architectural principles that guide its structure, execution, and reliability. These principles are not only conceptual but are directly reflected in the current implementation, ensuring that the system remains stable, maintainable, and scalable.

3.6.1 Architectural Pillars

The system architecture is structured around four primary pillars:

- **Decoupled Processing (Modularity)**: Functional logic is partitioned into independent modules (e.g., `sensor`, `control`, `comm`, `logger`), enabling clear separation of concerns. This modular structure supports isolated testing, simplified debugging, and seamless replacement or extension of individual components without affecting the overall system.
- **Deterministic Task Execution**: The **VAIOS** execution layer enforces well-defined scheduling of time-critical tasks. By maintaining consistent execution intervals (dt), the system ensures predictable behavior of control loops, which is essential for maintaining stability in real-time control systems.
- **Hardware Abstraction**: The **NavHAL** layer provides a uniform interface to low-level peripherals such as I2C, SPI, and UART. This abstraction isolates hardware-specific implementations from higher-level logic, enabling portability across different microcontroller platforms.
- **State-Aware Fail-Safety**: The system maintains a global system state (`sys_state_t`) to track operational readiness and detect abnormal conditions. Safety checks across modules—such as RC signal validation and IMU health monitoring—enable automatic transitions to predefined failsafe states in the event of faults or signal loss.

These principles collectively ensure that the system remains modular, predictable, and robust under varying operating conditions. By embedding these design choices into the architecture, the Vayu stack achieves a balance between performance, flexibility, and reliability.

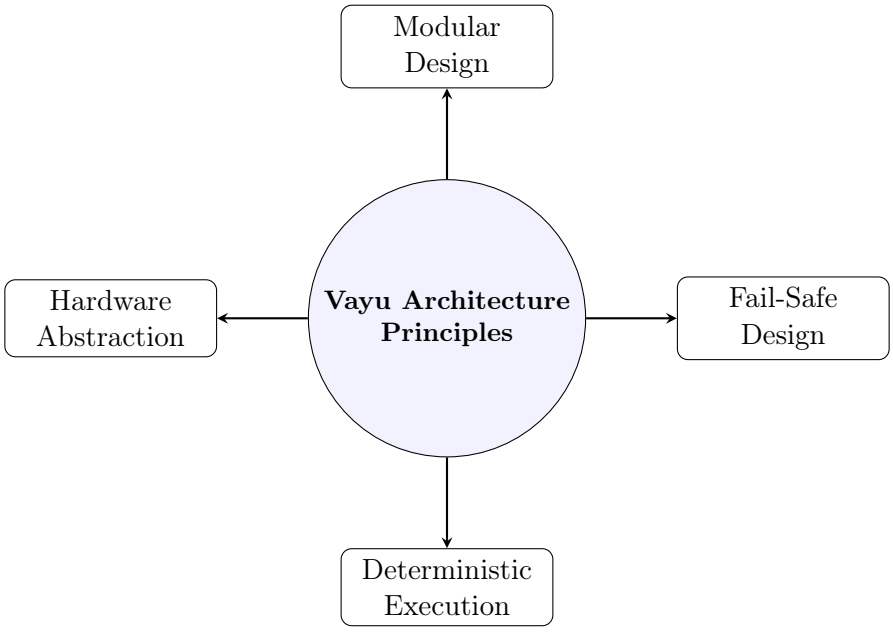

Figure 3.6: Core architectural principles of the Vayu flight control system.

3.7 Comparison

Vayu is designed with a focus on modularity, hardware independence, and deterministic execution. While several open-source flight control frameworks such as PX4 and ArduPilot provide comprehensive and feature-rich solutions, their architectural design and target use cases differ from the objectives of the Vayu system.

Established platforms like PX4 and ArduPilot are designed to support a wide range of hardware configurations and application scenarios. As a result, they often adopt large, monolithic codebases with extensive feature sets. While this provides flexibility, it can introduce additional complexity, increase resource requirements, and make system-level modifications more challenging.

In contrast, the Vayu architecture emphasizes a lightweight and structured design. By explicitly separating hardware abstraction (NavHAL), execution management (VAIOS), and control logic (Vayu), the system achieves a clear separation of concerns. This enables easier customization, improved maintainability, and tighter control over execution behavior.

Another key distinction lies in execution control. Traditional systems rely on general-purpose RTOS frameworks with complex scheduling mechanisms, whereas Vayu adopts a purpose-driven execution model tailored for embedded control systems. This approach simplifies task management while ensuring deterministic timing for critical control loops.

From a hardware perspective, many existing systems are closely tied to specific microcontroller families or vendor-provided libraries. Vayu addresses this limitation through a dedicated hardware abstraction layer, allowing the same control logic to be deployed across different platforms with minimal changes.

Table 3.1 summarizes the key differences between the approaches.

Aspect	**PX4 / ArduPilot**	**Vayu**
Architecture	Feature-rich, monolithic	Layered, modular
Hardware Support	Broad, vendor-linked	Hardware-agnostic (NavHAL)
Execution Model	General-purpose RTOS	Structured, deterministic (VAIOS)
Complexity	High	Controlled and minimal
Customization	Moderate (complex)	High (modular design)

Table 3.1: Comparison of Vayu architecture with existing flight control stacks

It is important to note that the objective of Vayu is not to replace existing systems, but to explore a design space that prioritizes simplicity, control over execution, and hardware independence. This makes it particularly suitable for research, rapid prototyping, and systems where fine-grained control over architecture is required.

3.8 Summary

This chapter presented the overall architecture of the Vayu flight control stack, detailing its layered structure, core components, data flow, execution model, and guiding design principles. The system is organized as a modular and hierarchical framework, with clear separation between hardware abstraction (NavHAL), execution management (VAIOS), and control logic (Vayu).

The layered design, combined with a well-defined data flow pipeline, enables efficient transformation of sensor measurements into actuator commands while maintaining system clarity and modularity. The execution model ensures deterministic behavior for time-critical tasks, which is essential for stable and reliable control.

Through the integration of these elements, the Vayu architecture achieves a balance between flexibility, performance, and maintainability. The comparison with existing systems highlights the design choices that distinguish Vayu, particularly in terms of modularity, hardware independence, and execution control.

Chapter 4

Hardware

This chapter describes the physical hardware platform used to implement the Vayu flight control stack. The hardware design serves as the foundation upon which the software architecture, introduced in the previous chapter, is realized.

The system integrates a microcontroller, sensor suite, communication interfaces, and actuator control mechanisms into a cohesive embedded platform. The hardware is selected and configured to support deterministic execution, low-latency data acquisition, and modular abstraction as required by the Vayu architecture.

In this chapter, we will discuss the following:

- System Overview
- Microcontroller Unit (MCU)
- Sensors
- Communication Interfaces
- Actuation and Motor Control
- Power Management
- PCB Design and Fabrication
- Hardware-Software Co-design
- Design Considerations

4.1 System Overview

The hardware platform for the Vayu flight control stack is designed as an integrated embedded system that supports real-time sensing, computation, and actuation. It brings together a microcontroller unit, sensor suite, communication interfaces, actuator outputs, and power management circuitry into a cohesive and extensible design.

At the core of the system lies a microcontroller-based processing unit, which interfaces with multiple peripherals to acquire sensor data, execute control algorithms, and generate actuator signals. The sensor subsystem includes an inertial measurement unit

(IMU) for high-frequency motion sensing, along with support for additional sensors such as barometers and GPS modules for enhanced state estimation.

The system provides multiple communication interfaces to enable interaction with external components. These include serial communication for radio control input and telemetry, as well as expansion interfaces such as I2C and SPI for connecting additional peripherals. Dedicated connectors are provided to facilitate modular integration of sensors and external devices.

Actuation is achieved through timer-driven PWM outputs, which are routed to electronic speed controllers (ESCs) for motor control. The design supports multi-rotor configurations and ensures precise timing required for stable flight control.

The power subsystem is designed to efficiently regulate input voltage and provide stable supply rails for digital and analog components. A switching regulator is used to generate a 5V rail from the input supply, followed by a linear regulator to produce a clean 3.3V supply for sensitive components such as the microcontroller and sensors.

Figure 4.1 illustrates the high-level organization of the hardware system, highlighting the interaction between major subsystems.

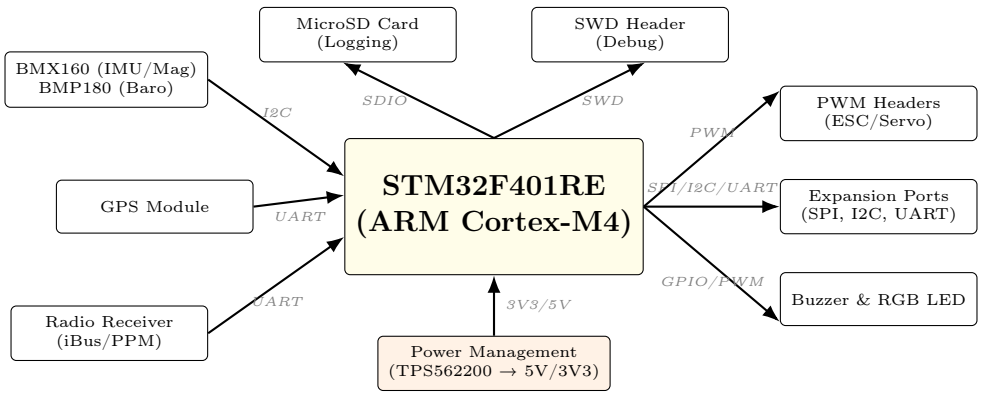

Figure 4.1: Detailed hardware architecture based on the Vayu v0.0.3 schematics, illustrating the integration of the STM32F401RE MCU, sensor suite, and power regulation stages.

This hardware architecture is designed to support the layered software structure introduced in Chapter 3. The microcontroller and peripherals are abstracted through NavHAL, while the deterministic execution requirements are enabled by the underlying hardware capabilities such as timers, interrupts, and communication interfaces. This alignment between hardware and software ensures efficient and predictable system operation.

4.2 Microcontroller Unit (MCU)

The Vayu flight control system is built around the STM32F401RE microcontroller, based on the ARM Cortex-M4 architecture. This microcontroller provides a balance between computational capability, peripheral availability, and real-time performance, making it well-suited for embedded flight control applications.

The Cortex-M4 core operates at high clock frequencies and includes a hardware floating - point unit (FPU), which enables efficient execution of mathematical operations required

for sensor fusion and control algorithms. This is particularly beneficial for real-time attitude estimation and PID-based control loops.

The microcontroller integrates multiple on-chip memory resources, including Flash for program storage and SRAM for runtime data. These memory resources are sufficient to support the modular architecture of the Vayu stack, including sensor processing, control logic, communication, and logging functionalities.

A key aspect of the MCU selection is the availability of rich peripheral interfaces, which are essential for interfacing with sensors, communication modules, and actuators:

- **Timers**: Advanced timer peripherals are used to generate high-resolution PWM signals for motor control. These timers also support precise timing required for deterministic execution of control loops.
- **Communication Interfaces**: The MCU provides multiple UART, SPI, and I2C interfaces. UART is used for radio control input (iBus) and telemetry, while SPI and I2C are used for sensor interfacing and peripheral expansion.
- **Interrupt System**: The nested vectored interrupt controller (NVIC) enables low-latency interrupt handling, which is critical for time-sensitive operations such as sensor sampling and task scheduling.
- **Debug Interface**: A Serial Wire Debug (SWD) interface is provided for programming and debugging, allowing efficient development and testing of the system.

The MCU clocking system is configured using both high-speed and low-speed oscillators. An external high-speed crystal is used for accurate system timing, while a low-speed crystal supports auxiliary timing functions. This configuration ensures stable and precise timing behavior required for real-time control.

From a system architecture perspective, the microcontroller serves as the execution backbone of the Vayu stack. The hardware peripherals are abstracted through the NavHAL layer, enabling portability across different platforms. At the same time, the availability of timers, interrupts, and communication interfaces allows the VAIOS execution layer to implement deterministic scheduling and low-latency task execution.

While the STM32F401RE provides a balanced platform for the current implementation, future iterations of the system are planned to transition to higher-performance microcontrollers such as the STM32H7 series. This transition is motivated by the need for increased computational capability, enhanced peripheral bandwidth, and support for more advanced algorithms such as extended Kalman filtering and multi-sensor fusion.

Overall, the chosen microcontroller platform aligns with the current system requirements while maintaining a clear upgrade path for future scalability.

4.3 Sensors

The Vayu flight control system employs a multi-sensor architecture to obtain the measurements required for state estimation and control. The design combines high-frequency inertial sensing with lower-frequency environmental and positional measurements, enabling both rapid response and long-term stability in estimation.

4.3.1 Inertial Measurement Unit (IMU)

The primary sensing element in the system is the BMX160, which integrates a 3-axis accelerometer, 3-axis gyroscope, and an onboard magnetometer. The IMU provides high-frequency motion data required for real-time attitude estimation and stabilization.

The accelerometer supports configurable measurement ranges (e.g., $\pm 2g$ to $\pm 16g$), while the gyroscope provides angular rate measurements typically in the range of $\pm 125°/s$ to $\pm 2000°/s$. These configurable ranges allow the sensor to be tuned based on the expected motion dynamics of the system.

The sensor is interfaced with the microcontroller over I2C or SPI, with SPI preferred for higher throughput and lower latency. In the current implementation, the IMU is sampled at rates exceeding 1.5 kHz, enabling accurate capture of fast rotational dynamics and minimizing delay in the control loop.

The BMX160 also includes an internal digital filtering stage and FIFO buffering, which can be used to reduce noise and manage high-rate data streams efficiently. These features help maintain consistent sampling and reduce processor overhead.

In operation, gyroscope measurements provide short-term angular rate information with high temporal resolution, while accelerometer measurements serve as a gravity reference for long-term correction. The magnetometer provides heading information, although its accuracy can be affected by local magnetic disturbances.

In the current implementation, an Adafruit BMX160-based 9-axis IMU module is used, interfaced over I2C. The inertial sensors operate at an output data rate (ODR) of approximately 1600 Hz, while the magnetometer operates at a lower rate of approximately 100 Hz. This configuration enables high-frequency attitude estimation while maintaining lower-rate heading updates.

Future iterations of the system will transition to higher-performance IMUs, including devices from the STMicroelectronics ISM family and newer Bosch sensor families. These upgrades will utilize SPI interfaces for improved bandwidth and reduced latency, along with the use of a dedicated external magnetometer to improve heading accuracy and reduce susceptibility to onboard magnetic interference.

These complementary sensing modalities form the basis for the sensor fusion algorithms used in the system, enabling robust and drift-compensated attitude estimation.

4.3.2 Barometric Sensor

A barometric pressure sensor is used to estimate altitude by measuring atmospheric pressure and converting it into height information. Compared to inertial measurements, barometric data is available at lower update rates but provides a relatively stable estimate of vertical position.

In the current implementation, a BMP180 sensor is used, interfaced over I2C. The sensor provides pressure measurements at moderate rates, which are sufficient for altitude estimation and slow vertical dynamics.

While not directly used in high-frequency control loops, barometric measurements contribute to higher-level state estimation and are particularly useful for altitude stabilization and navigation.

Future iterations of the system will transition to more advanced sensors such as the BMP384. These newer sensors offer improved resolution, lower noise, and higher sampling rates, enabling more accurate and responsive altitude estimation.

The integration of barometric data with inertial measurements allows the system to compensate for drift in vertical estimation while maintaining smooth and stable altitude control.

4.3.3 Global Positioning System (GPS)

The system includes support for a GPS module to provide global position and velocity information. In the current implementation, a u-blox M9N module is used, interfaced with the microcontroller over UART.

A dedicated UART port is exposed on the hardware platform to facilitate direct connection of the GPS module. This design allows flexibility in module selection and simplifies integration without requiring modifications to the core system.

The M9N provides multi-constellation GNSS support (e.g., GPS, GLONASS, Galileo), improving positioning accuracy and reliability in diverse operating environments. Typical update rates range from 5–10 Hz, with support for higher rates depending on configuration.

GPS measurements are inherently lower in frequency and are subject to noise and latency; however, they provide absolute position and velocity references that are essential for navigation and position control in outdoor environments.

While GPS data is not suitable for high-frequency control loops, it plays a critical role in higher-level state estimation by correcting long-term drift in position estimates derived from inertial sensors.

The integration of GPS with inertial measurements enables the system to achieve both short-term responsiveness and long-term accuracy, forming the basis for robust navigation capabilities.

4.3.4 Sensor Integration and Design Considerations

The sensor subsystem operates across multiple frequency domains. High-rate IMU data drives real-time control and short-term state estimation, while lower-rate sensors such as barometers and GPS provide long-term correction and global reference.

Ensuring consistent sampling, low-latency data acquisition, and signal integrity is critical for reliable system performance. In particular, the quality of IMU measurements is strongly influenced by power stability and electrical noise, necessitating careful hardware design and filtering.

The architecture is designed to support future enhancements in sensing capabilities. Planned iterations include the use of dual 6-axis IMUs to provide redundancy and improve robustness against sensor failure. Additionally, a dedicated external magnetometer is considered to improve heading estimation by reducing interference from onboard electronics.

For low-altitude operation, additional sensors such as Time-of-Flight (ToF) range sensors are planned to provide accurate height estimation in the near-ground region, where barometric measurements are less reliable. This is particularly useful for achieving smooth landing and takeoff behavior.

Optical flow sensors may also be integrated to estimate planar velocity, enabling improved position stability in GPS-denied or low-altitude environments where GPS performance is degraded.

These extensions are aligned with the modular design philosophy of the Vayu system, allowing new sensing modalities to be incorporated without requiring fundamental changes to the overall architecture.

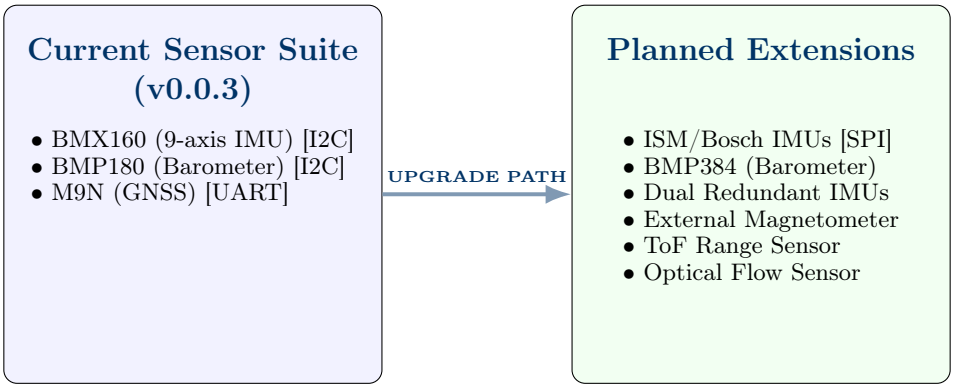

Figure 4.2: Overview of the current sensor implementation and the planned roadmap for enhanced sensing capabilities.

4.4 Communication Interfaces

The Vayu hardware platform incorporates multiple communication interfaces to support interaction with sensors, external modules, and ground control systems. These interfaces are selected to balance data rate, reliability, and system complexity, while enabling modular expansion.

4.4.1 UART Interfaces

Universal Asynchronous Receiver-Transmitter (UART) interfaces are used for communication with external modules requiring serial data exchange. In the current system, UART is primarily used for:

- **Radio Control Input**: The radio receiver communicates using the iBus protocol over UART, providing pilot commands for roll, pitch, yaw, and thrust.
- **GPS Module**: GPS data is received over UART, delivering position and velocity information at lower update rates.
- **Telemetry and Debugging**: UART interfaces are also used for transmitting system data to external devices for monitoring, logging, and debugging.

In the current hardware implementation, two UART ports are exposed. One is dedicated to radio control input, while the other is used for wired telemetry communication with the ground control system (Navigator). This separation ensures reliable handling of control inputs and system monitoring without resource contention.

Future iterations of the system aim to offload telemetry communication to a low-frequency wireless link (e.g., 868 MHz), enabling long-range communication with the ground control system. This transition will free a UART interface for dedicated GPS usage or additional peripherals.

Additionally, USB-based communication (USB CDC) is planned to provide a reliable wired serial interface for debugging, configuration, and data logging, reducing dependency on UART-based connections during development.

UART is chosen for these applications due to its simplicity, low overhead, and wide compatibility with external devices.

4.4.2 I2C Interface

The Inter-Integrated Circuit (I2C) interface is used for communication with low- to moderate-speed peripherals. In the Vayu system, I2C is primarily used to interface with sensors such as the IMU and barometric pressure sensor.

The shared bus structure of I2C allows multiple devices to be connected using a minimal number of wires, making it well-suited for compact embedded designs. However, its relatively lower data rate compared to SPI is taken into account during system design to ensure that timing requirements for sensor acquisition are met.

In the current hardware implementation, a single I2C bus (I2C1) is exposed across four external ports, enabling connection of multiple peripherals. This configuration supports flexibility in sensor placement and simplifies hardware integration.

Future iterations of the system will expand this design to include multiple I2C buses (e.g., I2C1 and I2C2), each exposed across multiple ports. This separation allows distribution of peripherals across independent buses, reducing bus contention and improving reliability and scalability as additional sensors are integrated.

The I2C interface thus provides a balance between simplicity and expandability, aligning with the modular design philosophy of the Vayu platform.

4.4.3 SPI Interface

The Serial Peripheral Interface (SPI) provides high-speed, full-duplex communication and is used for peripherals that require low-latency data transfer and high bandwidth. In the Vayu system, SPI is intended for high-performance sensor interfacing and communication modules.

In the current hardware revision, the SPI interface is exposed but not actively utilized by the primary sensing pipeline. External storage, such as the microSD card, is interfaced using the dedicated SDIO peripheral of the microcontroller instead of SPI, enabling higher data throughput and more efficient data logging.

An 868 MHz RF communication module is planned to operate over the SPI bus, enabling long-range telemetry with the ground control system. In future iterations, the SPI architecture will be further structured into multiple buses. The primary SPI bus (SPI1) will be reserved for internal high-speed peripherals such as next-generation IMUs, ensuring low-latency and high-bandwidth data acquisition.

A secondary SPI bus (SPI2) will be exposed externally through multiple ports, each with dedicated chip-select (CS) lines. This configuration enables simultaneous integration of multiple SPI-based peripherals, such as RF modules and external sensors, without bus contention.

The separation of SPI buses into internal and external domains ensures efficient resource allocation, improved signal integrity, and scalability for future system enhancements.

4.4.4 SDIO Interface

The Secure Digital Input Output (SDIO) interface is used in the Vayu system to connect an onboard microSD card for high-speed data logging. Unlike SPI-based storage, SDIO provides significantly higher data throughput and supports multi-bit data transfer, making it well-suited for recording high-frequency flight data.

The SD card is interfaced directly with the microcontroller's dedicated SDIO peripheral, allowing efficient data transfer with minimal CPU overhead. This enables continuous logging of sensor data, control states, and system diagnostics without interfering with time-critical control tasks.

The use of SDIO ensures that large volumes of data can be recorded reliably, which is essential for post-flight analysis, debugging, and system identification. It also supports future extensions such as onboard data buffering and black-box logging.

By offloading high-bandwidth storage operations to the SDIO interface, the system preserves other communication buses (such as SPI and UART) for real-time sensing and control applications.

4.4.5 USB CDC Interface

The Vayu system includes support for USB Communication Device Class (CDC), enabling the microcontroller to present itself as a virtual serial port when connected to a host system.

This interface is primarily used for debugging, configuration, and data monitoring during development. It provides a reliable and high-speed communication channel compared to traditional UART-based connections.

In addition to development use, USB CDC enables direct interfacing with an onboard computer when present. This allows exchange of data between the flight controller and higher-level processing units, such as vision-based navigation systems or companion computers.

The USB interface supports bidirectional communication, making it suitable for tasks such as parameter tuning, real-time telemetry, firmware updates, and integration with external processing pipelines.

By incorporating USB CDC, the system enhances both development efficiency and extensibility, supporting advanced use cases beyond basic flight control.

4.4.6 Debug and Programming Interface

A Serial Wire Debug (SWD) interface is provided for programming and debugging the microcontroller. This interface enables firmware development, real-time debugging, and system diagnostics during both development and testing phases.

4.4.7 Communication Design Considerations

The communication architecture of the Vayu system is designed with a strong emphasis on determinism, resource isolation, and scalability. Given the real-time nature of flight control systems, careful allocation of communication interfaces is essential to ensure that time-critical data paths remain unaffected by auxiliary operations.

A key consideration in the design is the separation of communication roles across different interfaces. Critical inputs such as radio control signals are isolated on dedicated UART channels, ensuring low-latency and reliable reception. Similarly, sensor communication over I2C is structured to meet timing requirements while maintaining simplicity in hardware design.

High-bandwidth and non-time-critical data flows, such as logging and debugging, are offloaded to dedicated interfaces like SDIO and USB CDC. The use of SDIO for onboard storage enables efficient high-speed data logging without occupying general-purpose communication buses. Likewise, USB CDC provides a robust and high-throughput channel for development, debugging, and interaction with external computing systems.

The architecture also anticipates future communication requirements. Planned integration of long-range telemetry using low-frequency RF (e.g., 868 MHz) allows offloading of telemetry traffic from UART interfaces, freeing resources for additional peripherals such as GPS. Similarly, the separation of SPI buses into internal and external domains enables high-speed peripherals to operate without contention while maintaining expandability.

Scalability and modularity are further enhanced through the exposure of multiple communication ports, allowing flexible integration of sensors, communication modules, and companion systems. This modular approach ensures that new functionalities can be incorporated without significant redesign of the core system.

Overall, the communication design reflects a balance between real-time performance, efficient resource utilization, and future extensibility, forming a robust foundation for both current operation and system evolution.

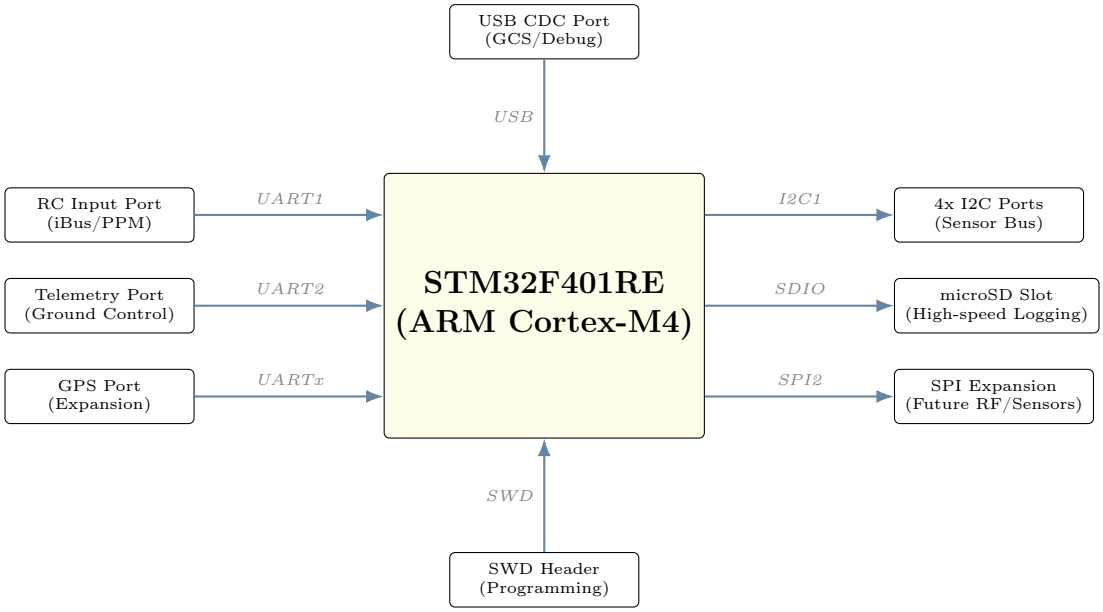

Figure 4.3: Mapping of communication interfaces and physical ports on the Vayu flight controller.

4.5 Actuation and Motor Control

The actuation subsystem of the Vayu flight control system is responsible for translating control outputs into physical forces and torques that drive the vehicle. This is achieved

through electronic speed controllers (ESCs) and motors, which are controlled via high-resolution PWM signals generated by the microcontroller.

4.5.1 PWM-Based Actuation

The microcontroller generates pulse-width modulation (PWM) signals using its internal timer peripherals. These signals are routed to ESCs, which regulate the speed of the motors based on the duty cycle of the PWM input.

In the current implementation, conventional PWM (CPWM) is used with an update frequency of approximately 400 Hz, which is compatible with standard ESCs. This frequency provides a balance between control responsiveness and signal stability.

Timer 1, an advanced-control timer, is used for driving the primary four motors. Its enhanced capabilities, such as higher resolution and synchronized channel outputs, make it well-suited for precise multi-channel motor control.

Figure 4.4 demonstrates the forward control path from the processing unit to the physical actuators.

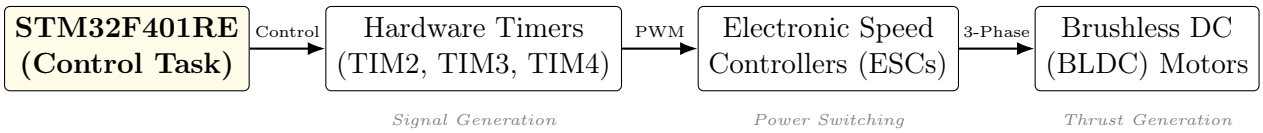

Figure 4.4: Actuation signal flow illustrating the transition from software control logic to physical motor thrust via hardware timers and speed controllers.

The use of hardware timers ensures precise timing and high-resolution signal generation, which is critical for stable and responsive motor control. Consistent PWM update rates are maintained to ensure smooth motor response and avoid control oscillations.

4.5.2 Motor Configuration and Mixing

In multi-rotor systems, control inputs such as roll, pitch, yaw, and thrust must be mapped to individual motor commands. This process, known as motor mixing, ensures that the combined effect of all motors produces the desired forces and torques.

The Vayu system supports standard multi-rotor configurations such as X4. A mixing matrix is used to distribute control signals to individual motors, taking into account their relative positions and rotation directions.

4.5.3 Timing and Control Requirements

Actuation is tightly coupled with the control loop and must satisfy strict timing requirements. The PWM signals are updated at a fixed rate synchronized with the control loop to ensure deterministic behavior.

Low-latency updates are essential to maintain stability, especially in the inner rate control loop, which operates at high frequencies. The use of dedicated hardware timers allows concurrent generation of multiple PWM signals without burdening the CPU.

The synchronization between control loop execution and PWM update ensures that actuator commands are applied consistently, minimizing phase delay in the control system and improving closed-loop stability.

4.5.4 Design Considerations

The actuation system is designed to ensure reliability, timing precision, and scalability. Care is taken to maintain signal integrity between the microcontroller and ESCs, and to minimize timing jitter, which can adversely affect control stability.

The use of an advanced timer (TIM1) for primary motor outputs ensures precise and synchronized signal generation, which is critical for balanced thrust and stable flight.

The modular design allows for extension to different configurations, including additional motors or alternative actuator types such as servos.

Future enhancements include support for digital ESC communication protocols such as DShot. Unlike traditional PWM, DShot provides a digital interface with higher update rates, improved noise immunity, and built-in error detection, making it suitable for high-performance flight control applications.

4.6 Power Management

The power management subsystem of the Vayu flight control platform is responsible for providing stable and efficient voltage regulation to all system components. Given the sensitivity of sensors and the real-time requirements of control algorithms, maintaining clean and reliable power rails is critical for system performance.

4.6.1 Power Architecture

The system is powered from an external supply, typically a battery source, which is regulated through a two-stage power conversion architecture.

Figure 4.5 illustrates the two-stage regulation approach used to provide stable power rails to the system.

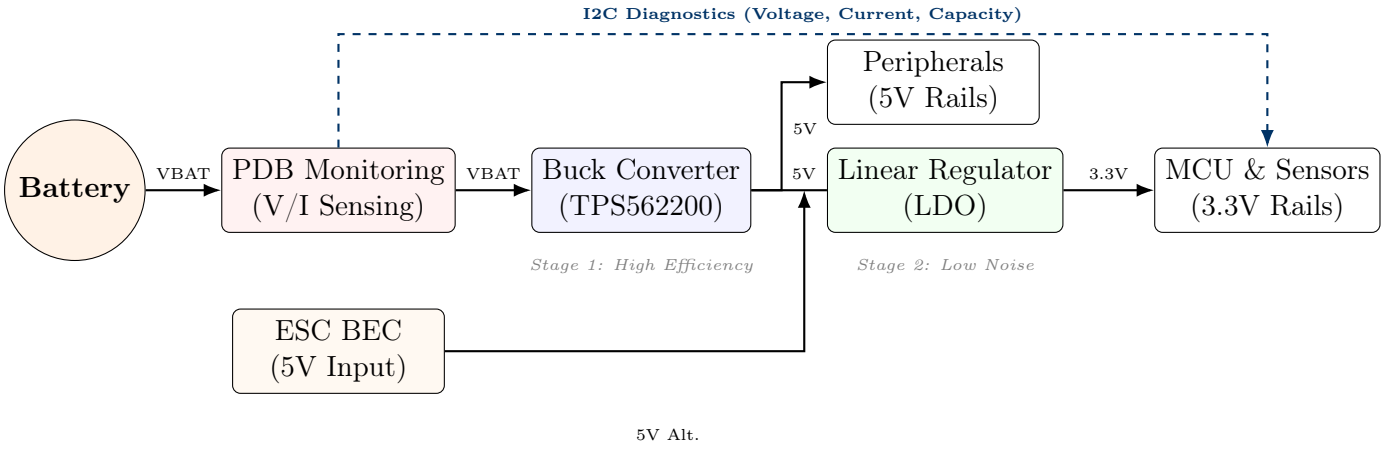

Figure 4.5: Two-stage power regulation architecture with integrated battery monitoring via the PDB, providing real-time diagnostics to the MCU.

In the first stage, a switching regulator (buck converter) is used to step down the input voltage to a stable 5V rail. This stage is implemented using a high-efficiency regulator, enabling the system to handle varying input voltages while minimizing power loss. Additionally, the system can be powered via a 5V input from an external Battery Elimination

Circuit (BEC) integrated within the Electronic Speed Controllers (ESCs). This provides a redundant or alternative power source for the 5V rail, particularly useful in secondary system testing or when the main battery-fed regulator is not active.

In the second stage, a linear regulator (LDO) is used to derive a clean 3.3V rail from the 5V supply. This 3.3V rail powers sensitive components such as the microcontroller and sensor modules.

4.6.2 Noise and Signal Integrity

Switching regulators, while efficient, introduce high-frequency noise into the power supply. To mitigate this, the two-stage regulation approach isolates noise-sensitive components by using the LDO to provide a cleaner supply.

Decoupling capacitors are placed close to power pins of the microcontroller and sensors to filter high-frequency noise and ensure stable voltage levels during transient load conditions.

This is particularly important for the IMU, where power supply noise can directly affect measurement accuracy and degrade estimation performance.

4.6.3 Power Distribution

The regulated power rails are distributed across the system to support different subsystems, including the microcontroller, sensors, communication interfaces, and peripheral modules.

Careful routing and grounding practices are followed to minimize voltage drops and electromagnetic interference. Analog and digital components are supplied through stable rails to ensure consistent operation under dynamic load conditions.

4.6.4 Battery Monitoring and Power Diagnostics

In addition to voltage regulation, the Vayu system incorporates battery monitoring capabilities to track power consumption and ensure safe operation during flight.

The system is designed to measure both battery voltage and current consumption, enabling real-time estimation of power usage. This information is critical for implementing features such as low-battery warnings, failsafe triggers, and flight time estimation.

A dedicated power distribution board (PDB) integrates a monitoring integrated circuit that provides voltage sensing, current measurement, and coulomb counting functionality. Coulomb counting enables accurate estimation of remaining battery capacity by integrating current consumption over time, offering a more reliable metric than voltage-based estimation alone.

The PDB communicates with the microcontroller via the I2C interface, allowing periodic acquisition of power-related data within the system. This integration ensures that battery diagnostics can be incorporated into higher-level system monitoring and decision-making processes.

Battery monitoring data can be used by the control system to enforce safety constraints, such as initiating controlled landing during low-power conditions or limiting system load under high current draw.

Future extensions may include integration of advanced battery management features such as state-of-charge (SoC) estimation, temperature monitoring, and predictive power modeling.

Overall, the inclusion of battery monitoring enhances system reliability, safety, and operational awareness, making it a critical component of the overall flight control platform.

4.6.5 Design Considerations

The power system is designed to balance efficiency, stability, and simplicity. The use of a switching regulator ensures efficient energy conversion, while the LDO provides clean power for critical components.

Overall, the power management subsystem plays a crucial role in ensuring the stability and accuracy of the Vayu flight control system by providing reliable and low-noise power to all components.

4.7 PCB Design and Fabrication

The physical implementation of the Vayu flight control system is realized through a custom-designed printed circuit board (PCB). The design emphasizes compactness, signal integrity, and reliable power distribution, ensuring compatibility with the real-time and noise-sensitive requirements of flight control systems.

4.7.1 PCB Design

The PCB was designed using KiCad, an open-source electronics design automation (EDA) suite. Figure 4.6 shows the 3D representation of the board design, highlighting the integration of the STM32F4 microcontroller, sensor interfaces, communication ports, and power management circuitry.

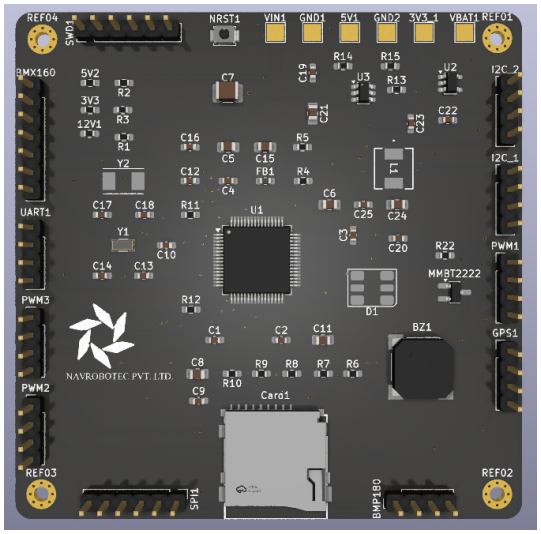

Figure 4.6: 3D visualization of the Vayu PCB design in KiCad.

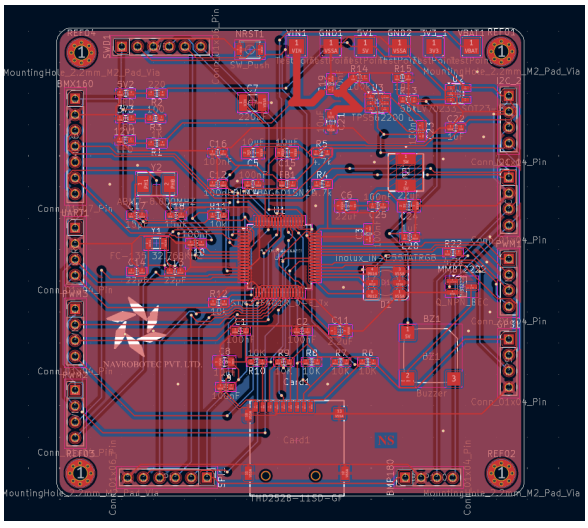

Figure 4.7: Detailed trace layout and routing of the Vayu flight controller PCB.

The PCB employs a two-layer design. Critical high-speed and noise-sensitive signals, such as IMU communication lines and clock traces, are routed with minimal length and proper grounding to reduce electromagnetic interference (EMI).

Power traces are designed with sufficient width to handle current requirements, particularly for the 5V and 3.3V rails. Dedicated ground planes are used to ensure stable reference potentials and improve signal integrity across the board.

Component placement is carefully optimized to minimize noise coupling. Sensitive components such as the IMU are placed away from switching regulators and high-current paths, reducing measurement disturbances. Decoupling capacitors are positioned close to power pins of the microcontroller and sensors to suppress high-frequency noise.

Connector placement is arranged to support modularity and ease of integration, with clearly exposed interfaces for UART, I2C, SPI, and power connections.

4.7.2 Fabrication and Assembly

Following the design phase, the PCB was fabricated and assembled. The final board integrates all system components, including the microcontroller, sensor interfaces, communication ports, and power management circuitry, into a compact and lightweight form factor suitable for drone applications.

Figure 4.8 shows the assembled hardware.

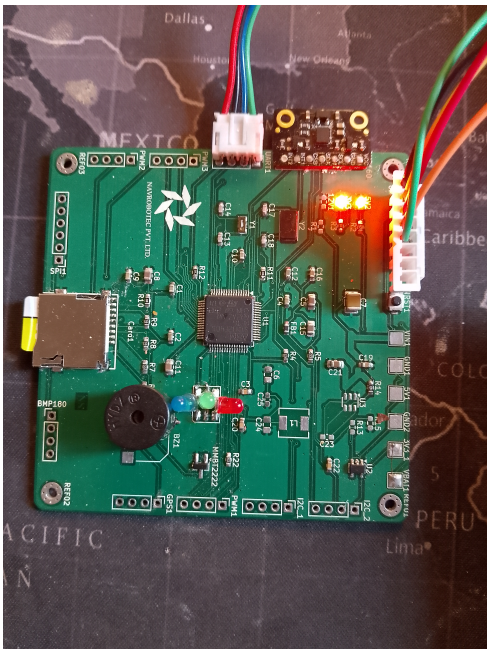

Figure 4.8: The fabricated and assembled Vayu flight control board.

The assembled board was tested for electrical continuity, power stability, and interface functionality. Successful integration of all subsystems validates the hardware design and provides a stable platform for the Vayu flight control stack.

The PCB design reflects a balance between compact layout, electrical robustness, and modular expandability, ensuring that the hardware platform can support both current implementation and future system enhancements.

4.8 Hardware-Software Co-design

The Vayu flight control system is developed using a hardware–software co-design approach, where the hardware platform and software architecture are designed in tandem to achieve deterministic performance, modularity, and scalability.

Figure 4.9 illustrates the layered architecture of the Vayu platform, highlighting the abstraction between the physical hardware and high-level application logic.

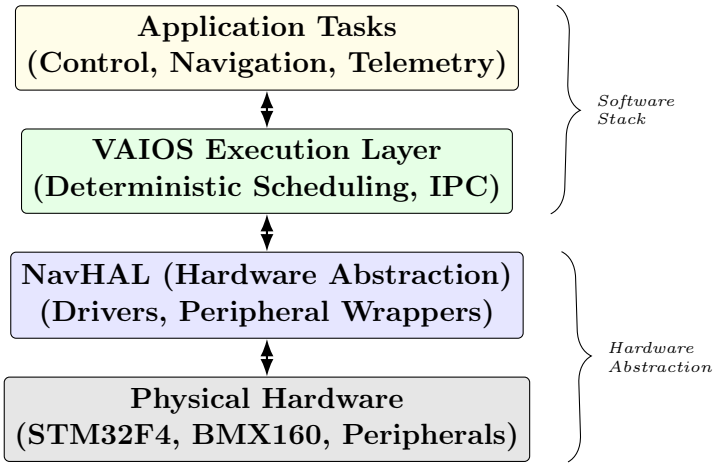

Figure 4.9: Layered system architecture demonstrating the hierarchy from physical hardware to application-level flight control tasks through abstraction layers.

Rather than treating hardware and software as independent layers, the system is structured such that hardware capabilities directly inform software design decisions, and software requirements guide hardware selection and configuration.

4.8.1 Mapping of Hardware Resources to Software Modules

Each hardware component in the system is accessed through a corresponding software module, ensuring a clear and structured interface between physical devices and application logic.

Hardware Component	Interface	Software Module
BMX160 IMU	I2C	Sensor Task
Barometer (BMP180)	I2C	Sensor Module
GPS Module	UART	Communication Module
RC Receiver (iBus)	UART	RC Input Module
PWM Outputs (ESC)	Timer	Motor Control Task
SD Card	SDIO	Logging Module

Table 4.1: Mapping between hardware components and software modules

This structured mapping ensures that all hardware interactions are mediated through well-defined software interfaces, enabling modular development and easier system maintenance.

4.8.2 Role of NavHAL

The Hardware Abstraction Layer (NavHAL) acts as the bridge between hardware and higher-level software components. It provides standardized interfaces for peripherals such as communication buses, timers, and GPIO.

By abstracting hardware-specific details, NavHAL allows the upper layers—including estimation, control, and communication modules—to operate independently of the underlying microcontroller. This enables portability across different hardware platforms without requiring significant changes to application logic.

4.8.3 Execution Support through Hardware Features

The hardware platform is specifically selected to support the deterministic execution model implemented by VAIOS. Key hardware features that enable this include:

- **Timers**: Used for precise scheduling of control loops and generation of PWM signals.
- **Interrupts**: Enable low-latency handling of sensor data and communication events.
- **Multiple Communication Interfaces**: Allow concurrent data exchange without resource contention.

These features ensure that time-critical tasks such as sensor acquisition and control computation can be executed reliably at fixed intervals.

4.8.4 Design Alignment and Scalability

The co-design approach ensures that the system remains scalable and adaptable to future enhancements. For example, the addition of new sensors or communication protocols can be accommodated by extending the corresponding modules without modifying the overall architecture.

Similarly, hardware upgrades—such as transitioning to higher-performance microcontrollers—can be achieved with minimal impact on higher-level software due to the abstraction provided by NavHAL.

This tight integration between hardware and software enables the Vayu system to achieve predictable performance, efficient resource utilization, and long-term maintainability.

4.9 Design Considerations

The hardware design of the Vayu flight control system is guided by several key considerations that ensure reliable operation, efficient performance, and seamless integration with the software architecture.

4.9.1 Real-Time Constraints

Flight control systems operate under strict real-time requirements. The hardware is selected to support deterministic execution through the availability of high-resolution timers, low-latency interrupt handling, and efficient communication interfaces. These features enable consistent sampling of sensor data and timely execution of control loops, which are essential for system stability.

Additionally, the allocation of dedicated interfaces for time-critical operations—such as UART for radio input and timers for actuation—ensures minimal latency and predictable system behavior.

4.9.2 Signal Integrity and Noise Management

Maintaining signal integrity is critical, particularly for sensitive components such as the IMU. The use of a two-stage power regulation system (buck converter followed by LDO), along with proper decoupling and grounding practices, helps reduce electrical noise. This ensures that sensor measurements remain accurate and are not adversely affected by power supply fluctuations or electromagnetic interference.

Special attention is given to PCB layout, component placement, and power routing to minimize noise coupling between high-current switching elements and precision sensing circuits.

4.9.3 Modularity and Expandability

The hardware platform is designed with modularity in mind, exposing communication interfaces such as UART, SPI, and I2C through dedicated connectors. This allows additional sensors, peripherals, and communication modules to be integrated without redesigning the core system.

Future revisions will extend this modularity further through additional interfaces such as USB-based communication and expanded bus availability, enabling integration with companion systems and advanced modules.

4.9.4 Scalability and Upgrade Path

The system is designed to accommodate future enhancements in both hardware and software. The use of a hardware abstraction layer (NavHAL) enables portability across different microcontrollers, while the modular hardware design supports the addition of redundant sensors, advanced communication protocols, and higher-performance processing units.

Planned upgrades include higher-performance MCUs, SPI-based high-speed sensors, and digital ESC communication protocols such as DShot.

4.9.5 Debugging and Development Support

Development and testing are facilitated through dedicated debugging interfaces such as SWD and UART-based telemetry. These interfaces allow real-time monitoring of system behavior, aiding in debugging, performance tuning, and validation of control algorithms.

Future iterations will incorporate USB CDC for higher-bandwidth debugging and interaction with external computing systems.

4.9.6 Hardware–Software Alignment

A key consideration throughout the design is the alignment between hardware capabilities and software requirements. The selection of peripherals, communication interfaces, and timing resources is closely matched to the needs of NavHAL and VAIOS, ensuring efficient abstraction and deterministic execution.

4.9.7 System Reliability and Safety

The hardware design is structured to support reliable system operation under dynamic conditions. Stable power regulation, proper grounding, and isolation of critical interfaces ensure consistent behavior across varying load scenarios.

Future updates will incorporate battery monitoring capabilities, including voltage and current sensing, enabling implementation of power-aware safety mechanisms such as low-battery failsafe and controlled landing strategies.

Overall, these design considerations ensure that the hardware platform not only supports current system requirements but also provides a robust, scalable, and reliable foundation for future development and deployment of the Vayu flight control stack.

Chapter 5

NavHAL

NavHAL is a core component of the Vayu flight control stack, responsible for providing a deterministic and structured interface between the underlying hardware platform and higher-level software layers.

Unlike conventional hardware abstraction layers that act as thin wrappers over vendor-provided drivers, NavHAL is designed as a compile-time configured system abstraction. It enforces consistency, portability, and predictable execution by defining a unified contract for all hardware interactions, while avoiding unnecessary runtime overhead.

Traditional embedded stacks often rely on layered abstractions consisting of vendor HALs, middleware, and optional RTOS components. While these approaches simplify development, they introduce hidden execution costs, increase system complexity, and make timing behavior difficult to reason about—an undesirable property in real-time flight control systems.

NavHAL addresses these limitations by adopting a minimal and deterministic design philosophy. It provides direct yet controlled access to hardware through memory-mapped register abstractions, lightweight APIs, and modular subsystems. All implementations are selected at compile time, ensuring that only the required components are included in the final binary, thereby reducing both code size and runtime overhead.

A key design principle of NavHAL is execution-layer independence. The abstraction layer operates natively in bare-metal environments while remaining compatible with higher-level execution frameworks such as VAIOS. This allows system designers to choose between fully deterministic bare-metal execution and structured multitasking without modifying the underlying hardware interface.

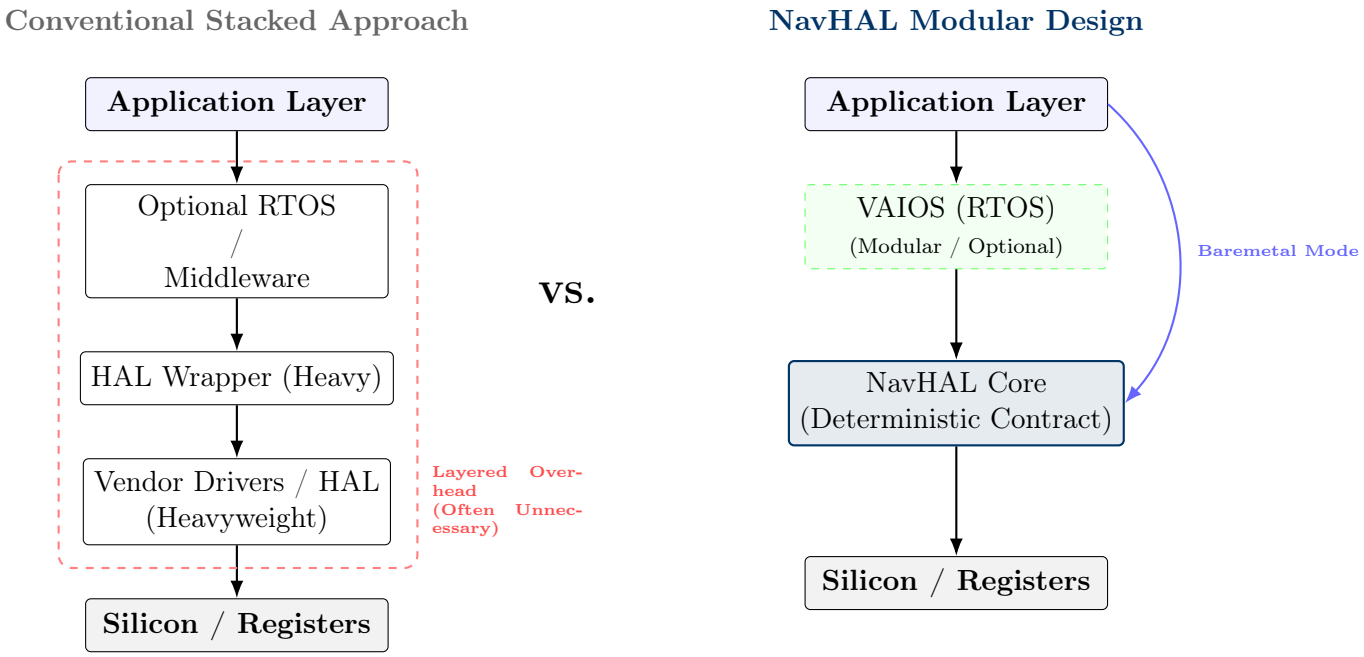

Figure 5.1: Comparison between conventional layered HAL architectures and the NavHAL design. NavHAL eliminates unnecessary abstraction layers and supports both bare-metal execution and optional RTOS integration without compromising determinism.

This chapter presents the motivation, design goals, architecture, and implementation details of NavHAL. It further discusses its integration with the VAIOS execution layer and compares it with existing abstraction approaches.

5.1 Motivation

Embedded flight control systems require precise timing, efficient execution, and direct interaction with hardware peripherals. The choice of hardware abstraction layer (HAL) plays a crucial role in achieving these requirements.

High-level frameworks such as the Arduino ecosystem prioritize ease of development. They provide simple and intuitive APIs that enable rapid prototyping. However, this abstraction hides low-level details, limiting control over hardware behavior and introducing execution overhead that is difficult to quantify. This lack of visibility and control makes such frameworks unsuitable for time-critical control systems.

In contrast, vendor-provided HALs expose detailed access to peripherals and enable efficient use of hardware resources. While this allows high-performance implementations, these frameworks are often complex, tightly coupled to specific microcontroller families, and difficult to scale or port across platforms.

These two approaches represent opposing design philosophies—one optimized for simplicity and the other for control and performance—leaving a gap in between that is not adequately addressed by existing solutions.

Feature	Arduino Framework	Vendor HAL	NavHAL
Ease of Use	High	Low	High
Performance	Low	High	High
Hardware Control	Limited	Extensive	Controlled
Portability	Moderate	Low	High
Determinism	Low	Moderate	High
Abstraction Overhead	High	Moderate	Low
Learning Curve	Low	High	Moderate

Table 5.1: Comparison of abstraction approaches highlighting differences in usability, control, and performance.

NavHAL is designed to address this gap by providing a structured interface that maintains low-level control while simplifying interaction with hardware. It achieves this through compile-time configuration, consistent APIs, and direct mapping to underlying peripherals.

The design enables predictable execution, reduced overhead, and improved portability without introducing additional runtime complexity. At the same time, it remains flexible enough to operate independently in bare-metal systems or alongside higher-level execution frameworks such as VAIOS.

5.2 Design Goals

NavHAL is designed to address the limitations of existing abstraction approaches by combining simplicity, performance, and control within a single unified framework. The design is guided by the following principles:

- **Simple and Consistent API:** Provide a clean and intuitive interface for peripheral interaction, enabling rapid development without exposing unnecessary hardware complexity.
- **Deterministic Behavior:** Ensure that hardware operations have predictable and bounded execution characteristics, suitable for real-time control systems.
- **Low Runtime Overhead:** Minimize abstraction cost by reducing indirection and leveraging compile-time configuration, ensuring performance close to direct register access.
- **Compile-Time Configuration:** Select implementations for core, vendor, and board at build time, eliminating runtime polymorphism and reducing binary size.
- **Portability Across Hardware:** Isolate platform-specific details while maintaining a uniform interface, enabling reuse of application code across different microcontrollers and boards.
- **Explicit Resource Control:** Provide direct and transparent access to hardware resources without implicit allocation or hidden state management.

- **Execution Independence:** Allow seamless operation in both bare-metal environments and with higher-level execution frameworks such as VAIOS.
- **Modular and Extensible Design:** Enable addition of new peripherals, architectures, or boards without affecting existing implementations.

These goals collectively define a design that prioritizes predictability, efficiency, and developer usability, forming the foundation for the NavHAL architecture.

5.3 Architecture of NavHAL

NavHAL is structured as a statically configured abstraction layer that separates hardware interaction across three orthogonal domains: processor architecture, vendor-specific implementation, and board-level configuration. This separation enables fine-grained hardware control while maintaining portability and scalability.

At a high level, NavHAL consists of a unified interface layer built on top of an architecture-specific core, which is specialized through three components: *core*, *vendor*, and *board*. These components are selected at build time, ensuring that only the required implementation is included in the final binary.

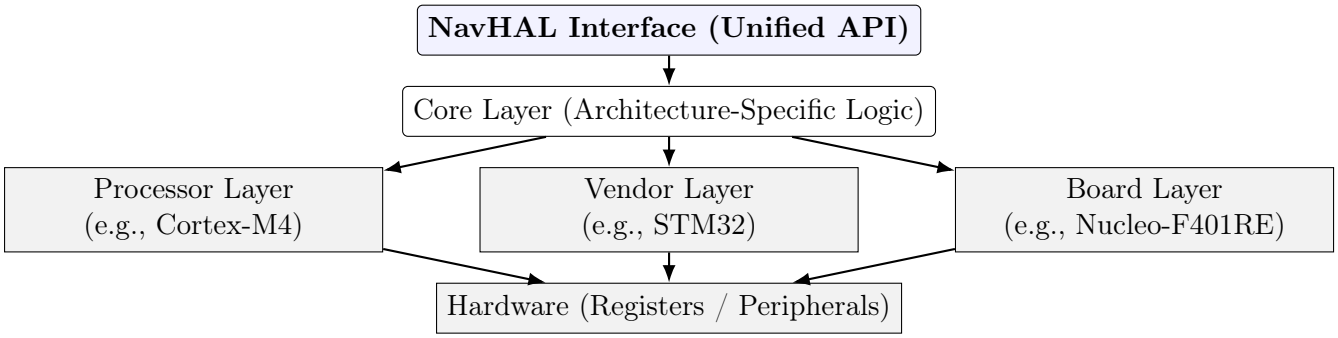

Figure 5.2: NavHAL architecture illustrating separation across processor, vendor, and board layers.

The **Interface Layer** provides a consistent and hardware-agnostic API for peripherals such as GPIO, UART, I2C, SPI, and timers. It defines the contract used by upper layers without exposing implementation details.

The **Core Layer** contains architecture-specific logic and implements the functional behavior of peripherals. It translates API calls into hardware operations using definitions provided by the vendor and board layers.

The architecture is further decomposed into three domains:

- **Processor Layer:** Implements architecture-level features such as startup routines, interrupt handling, and low-level execution primitives (e.g., Cortex-M4 support).
- **Vendor Layer:** Defines microcontroller-specific details including register mappings, memory layouts, and peripheral structures. It enables direct access to hardware registers.

- **Board Layer:** Provides board-level configuration such as pin mappings and default peripheral assignments, allowing application code to remain independent of physical layouts.

This separation allows independent evolution of each domain. Supporting a new board requires only board-level changes, while adapting to a different microcontroller primarily affects the vendor layer.

The abstraction is resolved at compile time through build configuration parameters (processor, vendor, and board). This eliminates unused components and reduces binary size.

To maintain efficiency, the design minimizes runtime indirection. Public APIs remain lightweight, while internal operations rely on inline functions and macros. As a result, most interactions map closely to direct register access.

NavHAL operates as a stateless access layer and does not enforce resource ownership or concurrency control. These responsibilities are delegated to the execution layer (e.g., VAIOS) when present, allowing NavHAL to remain lightweight and flexible.

Overall, the architecture combines structured abstraction with direct hardware access, enabling portability without sacrificing performance.

5.4 Interface Design (APIs)

The interface design of NavHAL is centered around simplicity, consistency, and explicit control over hardware behavior. The API is intentionally designed to provide a development experience similar to high-level frameworks, while preserving the determinism and performance required in real-time embedded systems.

Design Principles

The following principles guide the API design:

- **Simplicity:** APIs are designed to be minimal and intuitive, reducing the amount of boilerplate required for common operations.
- **Consistency:** All peripherals follow a uniform naming and usage pattern, enabling developers to interact with different modules using a predictable interface.
- **Explicit Configuration:** Hardware configuration is always performed explicitly by the user, avoiding hidden initialization or implicit behavior.
- **Low Overhead:** APIs are implemented as lightweight functions, with internal operations resolved using inline functions and macros to minimize runtime cost.
- **Hardware Abstraction:** Peripheral identifiers and configurations abstract underlying hardware details, allowing code to remain portable across platforms.

API Structure

NavHAL follows a consistent naming convention across all modules:

`hal_<peripheral>_<operation>`

Examples include:

- `hal_gpio_setmode()`
- `hal_gpio_digitalwrite()`
- `hal_uart_init()`
- `hal_i2c_read()`

This structure ensures that all peripherals expose a uniform and predictable interface.

Example: GPIO Interface

A representative example of the API design is shown below:

```
#define CORTEX_M4
#include "navhal.h"

int main(void)
{
    systick_init(1000); // 1 ms tick

    hal_gpio_setmode(GPIO_PA05, GPIO_OUTPUT, GPIO_PUPD_NONE);

    while (1)
    {
        hal_gpio_digitalwrite(GPIO_PA05, GPIO_HIGH);
        delay_ms(100);
        hal_gpio_digitalwrite(GPIO_PA05, GPIO_LOW);
        delay_ms(100);
    }
}
```

Listing 5.1: Example LED blink using NavHAL

This example highlights key characteristics of the interface:

- **Readable and Minimal:** The code closely resembles high-level frameworks, reducing the learning curve.
- **Explicit Timing Control:** System timing is initialized explicitly using `systick_init()`, ensuring predictable behavior.
- **Abstract Pin Representation:** Identifiers such as `GPIO_PA05` abstract the underlying hardware mapping, enabling portability.
- **Deterministic Execution:** No hidden scheduling or background processes are involved; all operations are directly controlled by the application.

Compile-Time Resolution

NavHAL APIs are designed such that most hardware-specific details are resolved at compile time. Peripheral identifiers are represented as compile-time constants, which are translated into register-level operations through inline functions and macros within the core and vendor layers.

This approach ensures that API usage does not introduce significant runtime overhead, and that generated code remains close to hand-written low-level implementations.

Design Trade-offs

To maintain simplicity and performance, NavHAL avoids introducing complex configuration objects or runtime polymorphism. While this reduces flexibility in dynamic scenarios, it ensures predictable behavior and minimal overhead, which are critical in embedded flight control systems.

Additionally, NavHAL does not manage resource ownership or concurrency. These responsibilities are delegated to the execution layer when required, allowing the API to remain lightweight and execution-model agnostic.

5.5 Peripheral Abstractions

NavHAL provides a uniform abstraction model across all hardware peripherals, enabling consistent interaction patterns while maintaining low-level control and minimal runtime overhead. Each peripheral module follows a common design structure that separates interface, implementation, and hardware mapping.

Design Pattern

All peripherals in NavHAL follow a consistent abstraction pattern:

```
Identifier → Macro Resolution → Register Access
```

Peripheral identifiers (e.g., GPIO pins, UART instances) are represented as compile-time constants. These identifiers encode sufficient information to determine the associated hardware resource.

At compile time, these identifiers are resolved through macros and inline functions into direct register-level operations. This eliminates the need for runtime lookup tables or dynamic dispatch mechanisms.

Reference Implementation: GPIO

The GPIO module serves as a representative example of this abstraction model.

GPIO pins are defined as encoded identifiers, where the port and pin number are embedded within a single value. Helper macros extract this information:

- `GPIO_GET_PORT(pin)` → resolves to the memory-mapped GPIO register block
- `GPIO_GET_PIN(pin)` → extracts the pin index within the port

This enables direct register access as shown below:

```
static inline void hal_gpio_digitalwrite(hal_gpio_pin pin,
                                         hal_gpio_state state) {
  if (state)
    GPIO_GET_PORT(pin)->BSRR = (1U << GPIO_GET_PIN(pin));
  else
    GPIO_GET_PORT(pin)->BSRR = (1U << (GPIO_GET_PIN(pin) + 16));
}
```

The resulting compiled code closely matches hand-written register-level implementations, ensuring minimal overhead.

More complex configuration operations, such as setting pin modes or alternate functions, are implemented as standard functions in the core layer, combining readability with efficiency.

Consistency Across Peripherals

This abstraction pattern is consistently applied across all supported peripherals, including UART, I2C, SPI, timers, and others. Each module:

- Uses compile-time identifiers to represent hardware instances
- Resolves hardware access through macros and inline functions
- Provides a uniform API naming convention
- Avoids runtime indirection and dynamic memory usage

This consistency reduces cognitive load and allows developers to interact with different peripherals using a predictable interface.

Generic Interface Selection

For certain peripherals, such as UART, NavHAL supports multiple variants of an operation (e.g., writing a single byte, buffer, or structured data). To maintain a clean and unified API, NavHAL leverages C11 `_Generic` macros to select the appropriate implementation at compile time.

This enables usage such as:

```
hal_uart_write(uart, data);
```

where the underlying function is resolved based on the type of `data`. This approach avoids function overloading limitations in C while preserving type safety and eliminating runtime branching.

Performance Characteristics

The peripheral abstraction model is designed to introduce minimal runtime overhead. Key characteristics include:

- Compile-time resolution of peripheral identifiers

- Direct mapping to memory-mapped registers
- Use of inline functions for performance-critical operations
- Absence of dynamic dispatch or lookup tables

As a result, the generated machine code closely resembles manually written low-level implementations, making NavHAL suitable for performance-critical embedded applications.

Scalability

The abstraction model scales naturally to additional peripherals and hardware platforms. New peripherals can be integrated by following the same pattern, while new architectures or microcontrollers can be supported by extending the core and vendor layers without modifying the interface.

5.6 Resource Management

NavHAL adopts a minimal and explicit approach to resource management, prioritizing determinism and low overhead over dynamic flexibility. Unlike traditional abstraction layers that manage hardware resources through runtime allocation or centralized control, NavHAL delegates resource ownership and coordination to the application or execution layer.

Static Resource Binding

All hardware resources in NavHAL are bound at compile time. Peripheral instances such as GPIO pins, UART ports, I2C buses, and timers are selected through configuration parameters and encoded identifiers.

This approach ensures that:

- No runtime allocation or initialization of resources is required
- Peripheral mappings are fixed and known at build time
- Binary size is minimized by excluding unused components

As a result, resource access remains predictable and free from dynamic overhead.

No Implicit Ownership Model

NavHAL does not enforce ownership or locking mechanisms for hardware resources. Multiple modules can access the same peripheral without restriction at the abstraction layer.

This design avoids:

- Mutexes or synchronization primitives within the HAL
- Hidden state or resource tracking

- Runtime arbitration logic

Instead, it provides direct access to hardware, leaving coordination responsibility to higher layers.

Execution-Layer Responsibility

In systems where concurrency is present (e.g., when using VAIOS), resource management is handled externally. The execution layer is responsible for:

- Preventing conflicting access to shared peripherals
- Scheduling tasks that interact with hardware
- Ensuring safe interaction across multiple execution contexts

This separation keeps NavHAL lightweight while allowing flexible system design.

Deterministic Access Model

By avoiding dynamic allocation and runtime arbitration, NavHAL ensures that all hardware interactions have predictable execution characteristics. Each operation directly translates to register-level access, with no hidden delays or blocking behavior introduced by the abstraction layer.

Design Trade-offs

The chosen approach trades centralized resource management for simplicity and performance. While this requires careful coordination at the application or execution layer, it provides:

- Lower latency and reduced overhead
- Greater transparency in hardware interaction
- Full control over peripheral usage patterns

This model aligns with the requirements of real-time systems, where predictability and efficiency are often more critical than dynamic flexibility.

5.7 Timing and Determinism

Deterministic execution is a fundamental requirement in flight control systems, where control loops operate at high frequencies and depend on predictable timing behavior. NavHAL is designed to ensure that hardware interactions exhibit bounded and analyzable execution characteristics.

Deterministic Execution Model

NavHAL achieves determinism by eliminating sources of runtime variability commonly found in traditional abstraction layers. Specifically:

- No dynamic memory allocation
- No runtime peripheral discovery or lookup
- No internal scheduling or blocking abstractions
- No hidden background processes

All hardware interactions are resolved through direct register access, ensuring that the execution cost of each operation is fixed and predictable.

Compile-Time Resolution

Peripheral selection and configuration are resolved at compile time. This removes the need for runtime decision-making, ensuring that:

- Code paths are static and analyzable
- Instruction count remains consistent across executions
- Branching overhead is minimized

As a result, the timing behavior of API calls can be estimated directly from the generated machine code.

Direct Register Access

NavHAL maps all peripheral operations directly to memory-mapped registers using inline functions and macros. For example, GPIO operations reduce to a small number of register writes without intermediate layers.

This approach ensures:

- Constant-time execution for most operations
- Minimal instruction overhead
- Absence of function call chains or indirect dispatch

Interrupt Handling Model

Interrupt handling in NavHAL follows a lightweight and predictable structure. The abstraction layer provides:

- Direct enable/disable control over NVIC interrupts
- Static callback registration through function pointers
- A centralized handler that dispatches to user-defined callbacks

Since interrupt vectors are resolved statically and callback invocation is a direct function call, interrupt latency remains bounded and consistent.

Cycle-Level Measurement Support

NavHAL provides access to the Cortex-M Data Watchpoint and Trace (DWT) unit, enabling cycle-accurate timing measurement. This allows developers to:

- Measure execution time of critical sections
- Validate timing constraints of control loops
- Profile performance without external instrumentation

Such capabilities are essential for verifying real-time behavior in embedded systems.

Execution-Layer Separation

NavHAL does not introduce scheduling or concurrency mechanisms that could affect timing behavior. When used in conjunction with an execution layer such as VAIOS, task scheduling and synchronization are handled externally.

This separation ensures that:

- Hardware access latency remains independent of scheduling policies
- Timing characteristics of peripheral operations are preserved

Determinism in Practice

Due to the combination of compile-time configuration, direct register access, and absence of runtime abstraction overhead, most NavHAL operations execute in a small and fixed number of instructions. This makes the system suitable for high-frequency control loops where timing jitter must be minimized.

5.8 Portability Model

NavHAL achieves portability through structured separation of hardware concerns and compile-time specialization. Instead of relying on runtime abstraction layers, the system follows a clear adaptation flow where only the required components are implemented or replaced based on the target platform.

Porting Flow Overview

Portability in NavHAL follows a layered adaptation path:

```
Application → Interface → Core → Vendor → Board → Hardware
```

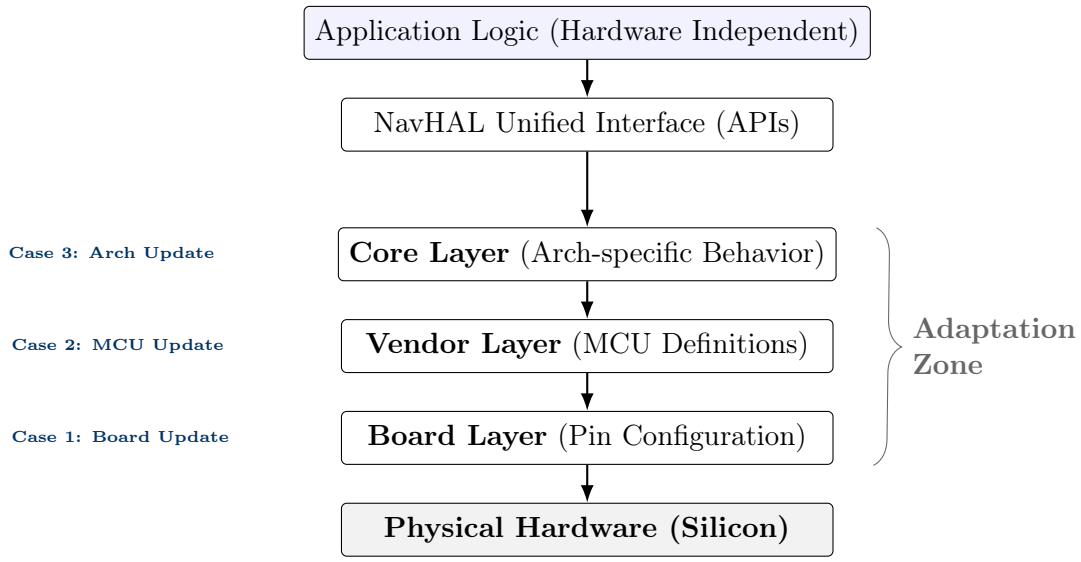

Figure 5.3: Portability model of NavHAL illustrating the vertical adaptation stack. Only the lower layers are modified when porting between boards, MCUs, or architectures.

The application and interface layers remain unchanged. Porting is performed by adapting the lower layers depending on what changes in the hardware platform.

Case 1: Porting to a New Board (Same MCU)

When the microcontroller remains the same but the physical board changes, only the **board layer** needs to be updated.

Flow:

- Define board-specific pin mappings (e.g., mapping logical pins to actual GPIOs)
- Configure default peripherals (LEDs, communication ports, etc.)
- Update linker script if memory layout differs

Impact:

- No changes required in core or vendor layers
- No changes required in application code

This is the simplest portability case and enables reuse of the entire software stack across different hardware layouts.

Case 2: Porting to a New Microcontroller (Same Architecture)

When moving to a different microcontroller within the same architecture (e.g., STM32F401 → STM32F411), the **vendor layer** is adapted.

Flow:

- Define register base addresses and peripheral mappings
- Update memory layout and peripheral availability

- Provide vendor-specific definitions required by the core layer

The core layer continues to use these definitions to implement peripheral behavior.
Impact:

- Core logic remains unchanged
- Board layer may require minor updates
- Application code remains unchanged

This separation allows reuse of peripheral logic while adapting only low-level hardware details.

Case 3: Porting to a New Architecture

When targeting a completely different processor architecture (e.g., moving from Cortex-M4 to another architecture), a new **core layer** must be implemented.
Flow:

- Implement architecture-specific startup code and interrupt handling
- Provide core implementations for all common HAL APIs
- Integrate vendor-specific register definitions for the new platform

Impact:

- Vendor and board layers are built on top of the new core
- Interface remains unchanged
- Application code remains unchanged

Although this is the most involved step, it is performed once per architecture and reused across multiple platforms.

Compile-Time Binding

All portability decisions are resolved at compile time through build configuration parameters such as:

- Target processor (core)
- Vendor family
- Board configuration

The build system includes only the selected implementations, ensuring:

- No runtime branching or hardware detection
- Reduced binary size
- Direct mapping to hardware

Key Observation

In all cases, the upper layers of the system remain unchanged. Portability is achieved by replacing only the minimal required layer:

Change Type	Layer Modified
New Board	Board Layer
New MCU (same arch)	Vendor Layer
New Architecture	Core Layer

5.9 Integration with VAIOS

NavHAL is designed to operate independently of any execution framework and does not require a real-time operating system for its functionality. As a result, its integration with VAIOS is minimal and well-defined, ensuring a clear separation between hardware access and execution control.

Unlike tightly coupled HAL-RTOS designs, NavHAL does not embed scheduling logic, synchronization mechanisms, or task management within its implementation. All peripheral operations exposed by NavHAL remain identical whether the system is running in bare-metal mode or under VAIOS.

Interaction Model

When VAIOS is used, NavHAL functions are invoked within tasks managed by the scheduler, in the same manner as they would be called in a bare-metal loop. This ensures that:

- No changes are required in NavHAL APIs when transitioning between execution models
- Peripheral access semantics remain consistent
- Timing behavior is preserved and controlled externally by the scheduler

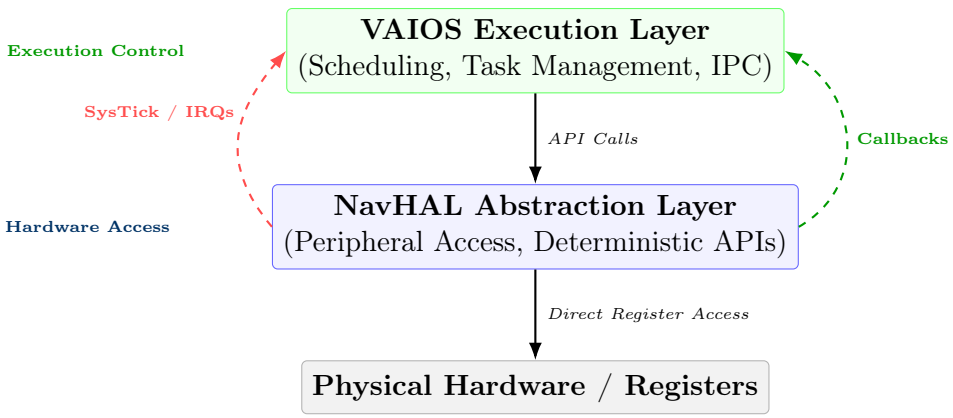

Figure 5.4: Interaction model between VAIOS and NavHAL illustrating functional boundaries and integration points.

Defined Integration Points

The interaction between NavHAL and VAIOS is limited to a small number of well-defined mechanisms:

- **System Tick (SysTick):** The SysTick interrupt is used by VAIOS as the primary timing source for scheduling. NavHAL provides the low-level configuration, while VAIOS utilizes it for task management.
- **PendSV Interrupt:** PendSV is used by VAIOS for context switching. NavHAL exposes interrupt control and handler linkage, allowing VAIOS to implement its scheduling mechanism without modifying HAL internals.
- **Interrupt Callback Mechanism:** NavHAL provides a generic interrupt attachment interface (`hal_interrupt_attach_callback`), which VAIOS uses to bind scheduler-related handlers and peripheral-driven events.
- **Communication Interfaces (UART/USB):** Peripheral drivers such as UART (and future USB CDC) are used by VAIOS for logging, telemetry, and debugging. NavHAL handles the hardware interaction, while VAIOS defines how and when data is processed.

Concurrency Model

NavHAL itself does not implement any concurrency control mechanisms such as mutexes, locks, or resource arbitration. This design choice ensures minimal overhead and deterministic behavior.

When VAIOS is present:

- Task scheduling and concurrency management are handled entirely by VAIOS
- Prevention of concurrent peripheral access is the responsibility of the execution layer
- NavHAL remains a stateless access layer with no knowledge of task context

Design Implication

This separation results in a clean architectural boundary:

```
NavHAL → Hardware Access        VAIOS → Execution Control
```

NavHAL focuses solely on efficient and deterministic interaction with hardware, while VAIOS manages when and how these interactions occur.

5.10 Performance Evaluation

The performance of NavHAL was evaluated using a set of micro-benchmarks on an STM32F401RE microcontroller. All experiments were conducted on March 20, 2026. Measurements were obtained using the Cortex-M4 Data Watchpoint and Trace (DWT) cycle counter, enabling cycle-accurate timing independent of clock frequency. This ensures that the reported results reflect intrinsic software overhead rather than variations in processor frequency.

5.10.1 GPIO Toggle Performance

A GPIO pin was toggled for 100,000 iterations across multiple abstraction layers.

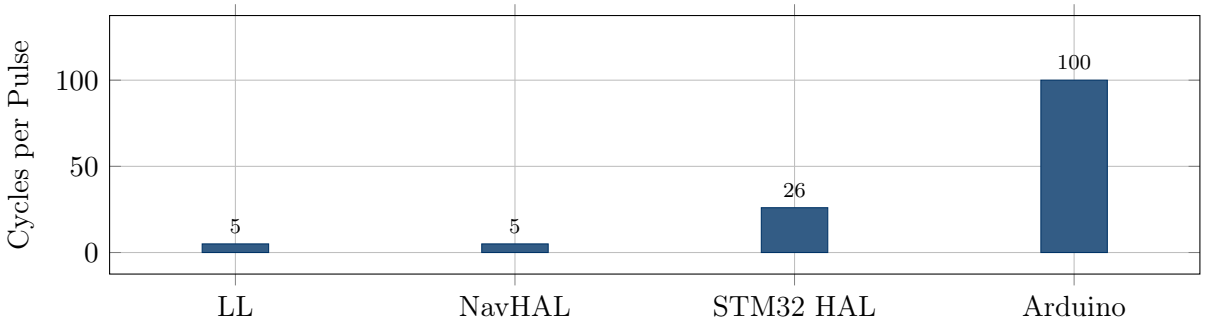

Figure 5.5: GPIO toggle performance across abstraction layers.

The GPIO toggle benchmark, shown in Fig. 5.5, captures the fundamental cost of interacting with hardware, where a single pulse consists of one SET and one RESET operation. Direct register access achieves a cost of 5 cycles per pulse, which represents the hardware-imposed lower bound for this operation and serves as a baseline for comparison. NavHAL achieves the same execution cost of 5 cycles, indicating that its abstraction introduces no additional overhead. This equivalence suggests that all abstraction layers in NavHAL are resolved at compile time, allowing the generated code to directly map to hardware operations without intermediate penalties. In contrast, the STM32 HAL implementation requires 26 cycles per pulse, which is approximately five times higher than the baseline. This additional cost can be attributed to the layered design of the HAL, including function call overhead, parameter validation, and generalized handling of peripheral configurations. Arduino exhibits the highest cost at 100 cycles per pulse, corresponding to nearly twenty times the baseline. This significant overhead arises from its highly abstracted and generic programming model, which relies on runtime handling and does not provide direct mapping to hardware registers.

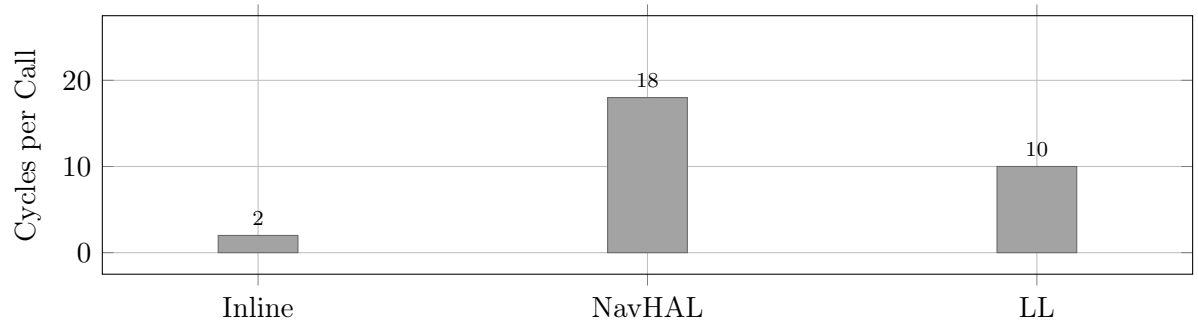

Figure 5.6: Function call overhead comparison.

Inline operations incur minimal cost, with execution requiring only 2 cycles, shown in Fig. 5.6, which effectively represents the lower bound for function execution when the compiler eliminates call overhead through inlining. In contrast, non-inlined function calls introduce additional latency due to stack operations, parameter passing, and control transfer. The LL implementation incurs a cost of 10 cycles per call, reflecting the overhead of a standard function invocation with minimal abstraction. NavHAL, when

not inlined, requires 18 cycles per call, indicating a higher overhead compared to LL due to its abstraction layers. However, this overhead remains modest and is primarily attributable to controlled function wrapping and interface design rather than excessive layering. Importantly, NavHAL mitigates this cost through the use of aggressive inlining in performance-critical paths, ensuring that frequently executed operations approach the lower bound observed for inline execution. These results highlight that while function calls inherently introduce overhead, careful design choices such as limiting call depth and leveraging compile-time inlining enable NavHAL to maintain efficient execution without sacrificing abstraction.

5.10.2 Determinism and Jitter

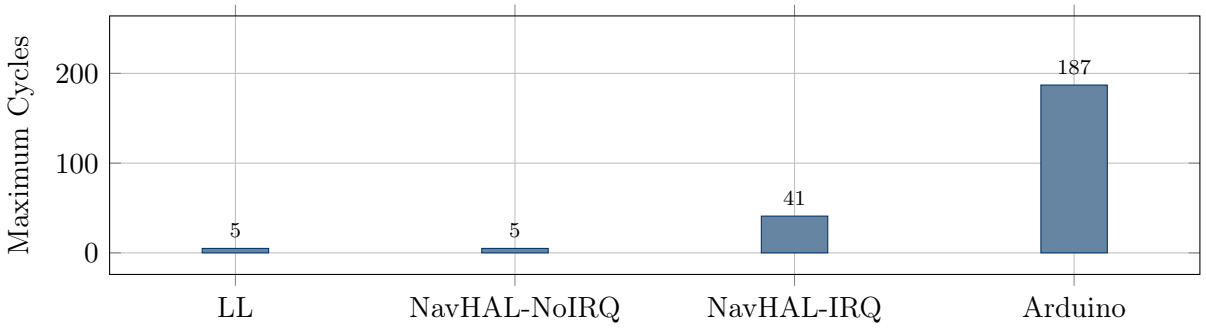

Figure 5.7: Worst-case execution latency (jitter).

Deterministic execution is critical for real-time systems, where consistent timing guarantees are required for reliable operation. As shown in Fig. 5.7, both direct register access (LL) and NavHAL without interrupts exhibit constant execution time at 5 cycles, indicating fully deterministic behavior in the absence of external interference. When interrupts are enabled, NavHAL demonstrates bounded latency, with execution time increasing up to 41 cycles. This variation represents interrupt-induced jitter and remains within a predictable and controlled range. In contrast, Arduino exhibits significantly higher variability, with worst-case latency reaching 187 cycles, reflecting both interrupt effects and additional overhead from its abstraction model. The results indicate that the jitter observed in NavHAL arises solely from interrupt preemption rather than intrinsic software overhead, thereby preserving deterministic behavior under controlled conditions while maintaining predictable bounds in interrupt-driven scenarios.

5.10.3 Interrupt Overhead

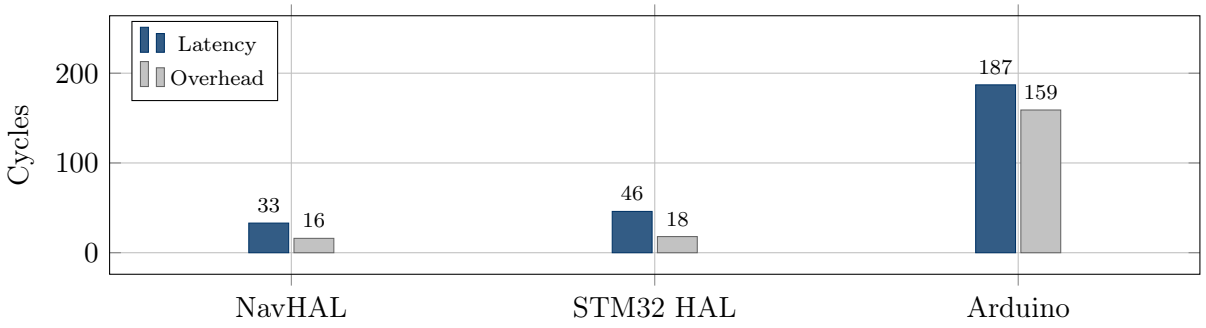

Figure 5.8: Interrupt latency and dispatch overhead.

Interrupt latency directly affects responsiveness in real-time control systems, as it determines how quickly the system can react to external events. As shown in Fig. 5.8, NavHAL exhibits a latency of 33 cycles with an additional dispatch overhead of approximately 16 cycles, resulting in a low overall interrupt handling cost. STM32 HAL shows slightly higher latency at 46 cycles and a comparable overhead of 18 cycles, reflecting the impact of its layered abstraction and generalized interrupt handling mechanisms. In contrast, Arduino demonstrates significantly higher latency at 187 cycles and a substantial overhead of 159 cycles, indicating considerable delay introduced by its abstraction model and runtime handling. These results highlight that NavHAL maintains efficient interrupt handling comparable to low-level implementations while minimizing additional overhead. Such low and predictable interrupt costs are essential for maintaining responsiveness and stability in time-critical applications, whereas the significantly higher overhead observed in Arduino can adversely impact control loop performance and system determinism.

5.10.4 Design Principles

The performance characteristics of NavHAL are achieved through the following design principles:

- **Compile-time specialization:** Hardware configuration is resolved at compile time, eliminating runtime branching.
- **Direct register access:** Peripheral interaction is performed through memory-mapped structures without intermediate abstraction layers.
- **Inline critical paths:** Frequently executed operations are implemented as inline functions.
- **Minimal call depth:** API functions are designed to avoid deep call hierarchies.
- **No dynamic dispatch:** The absence of virtual functions or runtime lookup ensures predictable execution.

5.11 Comparison with Existing HALs

This section compares NavHAL with commonly used hardware abstraction layers, including STM32 HAL, Arduino, and low-level (LL) register access. The comparison is based on key factors relevant to embedded system development, including performance, abstraction overhead, determinism, flexibility, and ease of use.

Unlike low-level register programming, which provides maximum performance at the cost of portability and maintainability, higher-level frameworks such as STM32 HAL and Arduino improve developer productivity but introduce additional runtime overhead and reduced control over hardware behavior. As demonstrated in Section 5.10, these trade-offs can significantly impact execution efficiency and timing predictability in real-time applications.

NavHAL is designed to bridge this gap by providing a high-level, portable abstraction while retaining performance characteristics equivalent to direct register access. This is achieved through compile-time specialization, minimal call depth, and the elimination of runtime dispatch mechanisms.

5.11.1 Performance and Overhead

NavHAL achieves performance comparable to LL implementations while significantly outperforming STM32 HAL and Arduino in all evaluated benchmarks. Unlike conventional HALs, which introduce layered abstractions and runtime checks, NavHAL resolves hardware interactions at compile time, eliminating unnecessary overhead.

5.11.2 Determinism and Real-Time Behavior

Deterministic execution is critical in time-sensitive applications such as robotics and flight control systems. NavHAL exhibits consistent execution timing in non-interrupt conditions and bounded latency under interrupt-driven scenarios. In contrast, higher-level frameworks such as Arduino introduce significant variability due to internal abstraction layers and runtime handling.

5.11.3 Abstraction vs Control

STM32 HAL and Arduino prioritize ease of use and portability but often abstract away hardware details, limiting fine-grained control. LL access provides full control but at the cost of code complexity and reduced portability. NavHAL maintains a balance by exposing a clean abstraction while preserving direct access to hardware capabilities.

5.11.4 Scalability and Maintainability

NavHAL is designed to scale across different microcontrollers and platforms through compile-time configuration. This approach avoids code duplication and reduces maintenance overhead, unlike traditional HALs that often rely on extensive conditional logic and device-specific implementations.

5.12 Summary

This chapter presented NavHAL, a hardware abstraction layer designed to provide high-level usability while preserving the performance characteristics of direct register access. Through a series of micro-benchmarks, NavHAL was shown to achieve zero-cost abstraction, deterministic execution, and low overhead, making it suitable for real-time embedded applications where both efficiency and predictability are critical. The results demonstrate that NavHAL eliminates the traditional trade-off between abstraction and performance observed in existing frameworks such as STM32 HAL and Arduino.

At present, NavHAL supports Cortex-M4-based microcontrollers, specifically the STM -32F401RE platform, which served as the evaluation target in this work. Ongoing development is focused on rapidly extending support across a broader range of STM32 microcontrollers as part of the Vayu project, with the goal of enabling scalable deployment across diverse embedded systems.

In addition, future work includes extending support to AVR-based platforms to improve accessibility for beginners and educational use cases. By combining performance, portability, and ease of use, NavHAL aims to serve as a unified abstraction layer for both advanced real-time systems and entry-level embedded development.

Chapter 6

VAIOS

VAIOS is the runtime layer of the Vayu system, designed to provide deterministic execution, efficient task management, and essential system services for real-time embedded applications. While NavHAL abstracts the underlying hardware and Vayu defines the higher-level flight control logic, VAIOS acts as the core system layer that coordinates execution across tasks, manages resources, and ensures predictable system behavior.

The design of VAIOS is centered around the requirements of time-critical systems such as drone flight controllers, where latency, determinism, and reliability are more important than general-purpose flexibility. Unlike conventional operating systems, VAIOS avoids complex and heavyweight abstractions, instead adopting a minimal and tightly controlled architecture tailored for embedded environments. It provides a preemptive scheduler, inter-task communication mechanisms, dynamic memory management, and system-level services such as file system access and logging.

A key focus of VAIOS is maintaining low overhead while enabling modularity. Features such as priority-based scheduling with round-robin execution, efficient context switching using hardware-supported mechanisms, and a configurable system design allow it to scale across different applications without compromising performance. Additionally, support for simulation through semihosting enables development and testing in environments where hardware may not be available.

This chapter presents the design and implementation of VAIOS, covering its architecture, scheduling model, task management, memory system, communication primitives, and integration with the Vayu flight control stack. Together, these components form a cohesive runtime environment that bridges low-level hardware interaction and high-level application logic.

6.1 Overview

VAIOS provides a structured runtime environment for managing execution and system services in embedded applications. It is organized around a set of core subsystems that together define how tasks are scheduled, how resources are managed, and how different parts of the system interact.

At the kernel level, VAIOS implements a preemptive scheduler with multiple priority levels and time-sliced execution within each level. Tasks are maintained in priority-specific ready queues, enabling efficient selection of the next task to execute. Context switching

is performed using hardware-supported mechanisms, ensuring low overhead during task transitions.

The system supports a well-defined task lifecycle, including ready, running, blocked, delayed, and terminated states. Separate stacks are allocated for each task, allowing independent execution and simplifying context management. Delayed and blocked tasks are maintained in dedicated lists, with periodic updates ensuring timely wake-up and rescheduling.

VAIOS also provides a set of system services required for building complete applications. These include inter-task communication primitives such as semaphores and mutexes, dynamic memory allocation with safety checks, and a virtual file system interface for interacting with storage devices. These components are designed to operate cohesively within the scheduler framework.

Configuration is handled at compile time through a centralized set of macros, allowing system parameters such as time slice duration, memory limits, and feature flags to be tailored to specific use cases. This approach keeps the runtime lean while enabling flexibility across different deployments.

6.2 Design Goals

The design of VAIOS is guided by the requirements of real-time embedded systems, where predictable execution, low overhead, and reliability are essential. The primary objective is to provide a runtime environment that supports structured application development without introducing unnecessary complexity or performance penalties.

A key goal is deterministic behavior. The system is designed to ensure that task scheduling and execution remain predictable under varying workloads. This is achieved through a priority-based preemptive scheduler, bounded interrupt handling, and controlled use of synchronization primitives. By avoiding hidden or implicit behaviors, VAIOS allows developers to reason about system timing more effectively.

Efficiency and minimal overhead are also central to the design. Core operations such as context switching, task scheduling, and inter-task communication are implemented to minimize runtime cost. Hardware-assisted mechanisms are leveraged wherever possible to reduce software overhead, ensuring that the system remains suitable for time-critical control applications.

Simplicity is another important consideration. The system avoids overly complex abstractions and focuses on a small, well-defined set of features that are sufficient for most embedded use cases. This reduces the learning curve and makes the system easier to debug and maintain, while still providing the necessary functionality for building complete applications.

Configurability is achieved through compile-time parameters, allowing the system to be tailored to different requirements without modifying core components. This includes control over scheduling behavior, memory limits, and feature enablement. Such flexibility enables VAIOS to adapt to both resource-constrained environments and more feature-rich deployments.

Finally, the system is designed with modularity in mind. Different subsystems, including scheduling, memory management, communication, and storage, are organized in a way that allows them to evolve independently. This supports incremental development and makes it easier to extend the system as new requirements emerge.

6.3 System Architecture

VAIOS is organized into a modular architecture that separates core system functionality from hardware-specific implementations. This separation allows the system to remain portable across different microcontroller platforms while maintaining a consistent kernel interface.

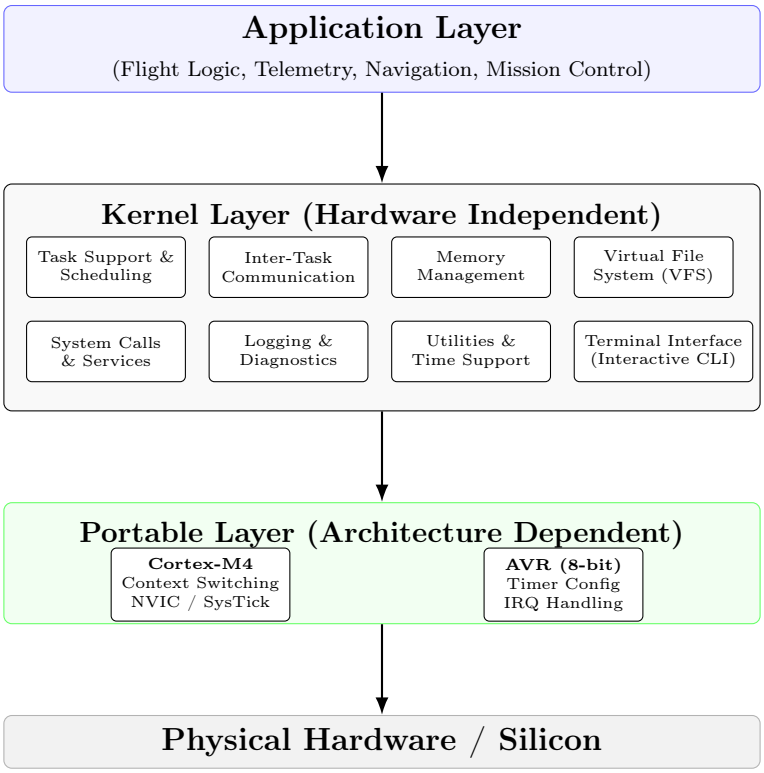

Figure 6.1: Layered architecture of VAIOS illustrating modular separation between the kernel services and hardware-specific implementations.

At a high level, the system is divided into two primary layers: the kernel layer and the portable layer.

The **kernel layer** contains the core components responsible for task management, scheduling, inter-task communication, memory management, and system services. This layer is hardware-independent and forms the main logic of the operating system. It includes modules for task scheduling and lifecycle management, synchronization primitives such as semaphores and mutexes, dynamic memory allocation, and higher-level services such as the virtual file system and terminal interface. Additional components provide logging, utility functions, and system call handling, enabling structured interaction between different parts of the system.

The **portable layer** encapsulates all hardware-dependent functionality. This includes context switching, interrupt configuration, and low-level processor-specific operations. Separate implementations are provided for different architectures, such as Cortex-M4 and AVR, allowing the same kernel to be reused across platforms. The portable layer exposes a well-defined interface to the kernel, ensuring that all hardware interactions are localized and can be adapted without modifying core system logic.

Within this structure, execution flow is coordinated between the two layers. The kernel determines scheduling decisions and system behavior, while the portable layer performs the necessary low-level operations such as saving and restoring task context during a switch. This division ensures that performance-critical operations are implemented close to the hardware, while higher-level logic remains clean and maintainable.

System services are integrated into the kernel as modular components rather than tightly coupled subsystems. For example, memory management, inter-task communication, and file system access are implemented as independent modules that interact through well-defined interfaces. This modularity simplifies development and allows individual components to be extended or replaced without affecting the overall system.

Overall, the architecture of VAIOS emphasizes separation of concerns, portability, and modular design. By isolating hardware-specific functionality and organizing system services into clear modules, the system achieves both flexibility and efficiency across different deployment scenarios.

6.4 Task Model

VAIOS follows a task-based execution model, where each unit of execution is represented as an independent task managed by the kernel. Each task is described by a Task Control Block (TCB), which stores all information required for scheduling, context switching, and lifecycle management.

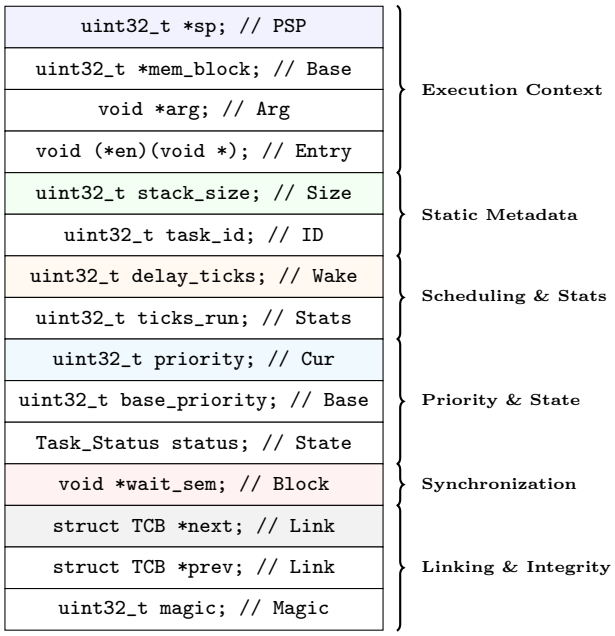

Figure 6.2: Memory layout of the VAIOS Task Control Block (TCB) showing field groupings by functional role.

The TCB , shown in Fig. 6.2, maintains the execution context of a task, including the current stack pointer, stack memory base, entry function, and argument. It also stores scheduling-related metadata such as priority, delay information, and runtime statistics. Additional fields are used for tracking synchronization state, linking tasks within scheduling lists, and ensuring system integrity through sanity checks. This structure enables

efficient context switching and flexible task management while keeping all relevant task information localized.

Each task is allocated a separate stack at creation time using the system memory allocator. The stack is initialized to simulate an exception return frame, allowing the task to start execution through the same mechanism used during context switching. This design ensures a uniform execution model and simplifies the scheduler implementation. Stack isolation also prevents interference between tasks and allows independent execution.

The lifecycle of a task is defined through a set of states: *READY*, *RUNNING*, *BLOCKED*, *DELAYED*, and *TERMINATED*. Tasks in the READY state are maintained in priority-specific queues and are eligible for execution. The RUNNING state corresponds to the currently executing task. Tasks enter the BLOCKED state when waiting for synchronization primitives such as semaphores or mutexes, while the DELAYED state is used for time-based suspension using system ticks. Once execution is complete or explicitly terminated, tasks transition to the TERMINATED state and are later cleaned up by the system.

Task creation involves allocating memory for both the TCB and its stack, initializing the execution context, and inserting the task into the appropriate ready queue. The system maintains separate lists for ready, blocked, and delayed tasks, allowing efficient state transitions and scheduling decisions. Tasks can be dynamically delayed, blocked, or unblocked through kernel APIs, enabling flexible control over execution flow.

An idle task is always present in the system and is scheduled when no other tasks are ready to execute. In addition to occupying the processor during idle periods, it performs background operations such as reclaiming memory from terminated tasks, ensuring that resource cleanup does not interfere with active task execution.

6.5 Scheduler Design

VAIOS employs a preemptive, priority-based scheduling strategy with round-robin execution among tasks of equal priority. The scheduler is designed to provide deterministic task selection with minimal overhead, making it suitable for real-time embedded systems. The flow is shown in Fig. 6.3.

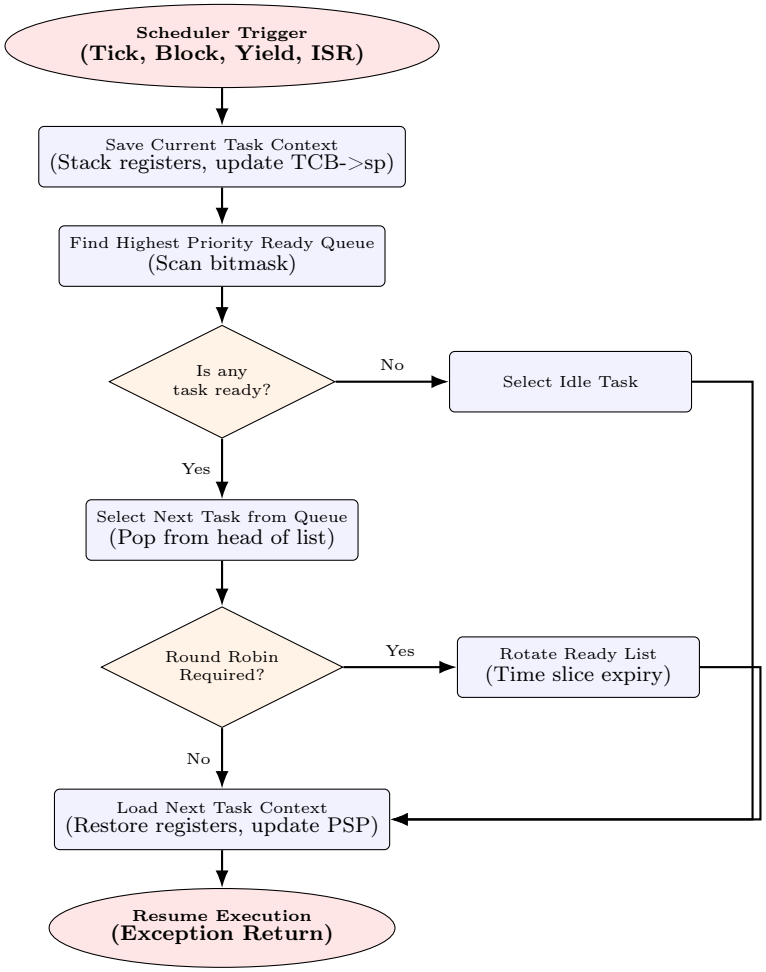

Figure 6.3: Operational flowchart of the VAIOS preemptive scheduler illustrating context management and priority-based selection logic.

Tasks are organized into multiple ready queues, one for each priority level. The system supports a fixed range of priorities, where higher numerical values correspond to higher scheduling priority. Each ready queue is implemented as a linked list, allowing tasks of the same priority to be scheduled in a round-robin manner. When a running task exhausts its time slice or yields execution, it is reinserted at the end of its corresponding ready queue, ensuring fair execution among tasks with equal priority. The structure is shown in Fig. 6.4.

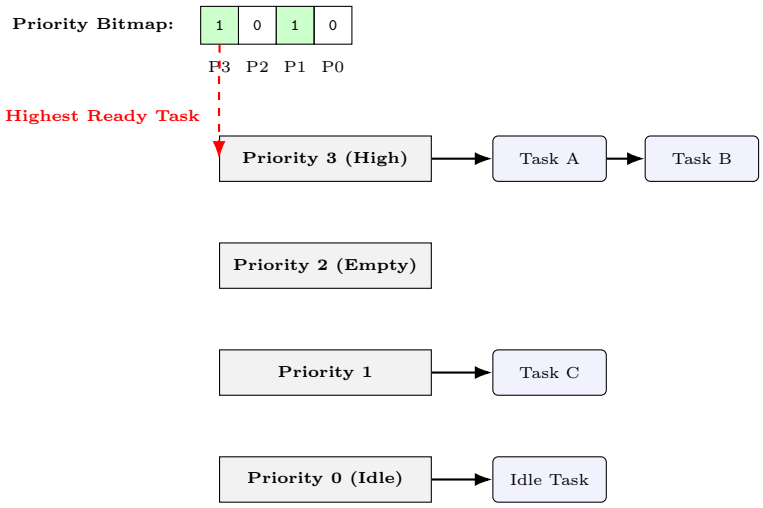

Figure 6.4: Priority-based ready queue structure in VAIOS. The scheduler uses a bitmap to identify the highest active queue, ensuring deterministic task selection complexity.

To enable efficient selection of the next task, VAIOS maintains a bitmap that tracks which priority levels have ready tasks. This allows the scheduler to identify the highest-priority ready task without scanning all queues. The highest active priority is determined by inspecting this bitmap, after which the corresponding ready queue is accessed to select the next task. This approach significantly reduces scheduling overhead and ensures predictable behavior.

The scheduling process is triggered by system events such as time slice expiration, task blocking, or explicit yield requests. When a scheduling event occurs, the current task is evaluated and, if still runnable, returned to the appropriate ready queue. The scheduler then selects the next task based on priority and queue ordering, updates its state to RUNNING, and performs a context switch.

Time slicing is configurable through compile-time parameters and applies to tasks within the same priority level. This allows developers to balance fairness and responsiveness based on application requirements. Higher-priority tasks always preempt lower-priority tasks, ensuring that critical operations receive immediate processor time.

The scheduler also integrates with delayed and blocked task management. Tasks in the DELAYED state are periodically checked against the system tick and moved back to the ready queues when their delay expires. Similarly, tasks waiting on synchronization primitives are transitioned from the BLOCKED state to READY when the required condition is satisfied.

6.6 Timing and Tick Management

VAIOS uses a periodic system tick to drive time-dependent behavior such as task delays, time slicing, and runtime accounting. The system tick is generated using a hardware timer (typically SysTick on Cortex-M), with the period configured at compile time. Each tick represents a fixed time interval and serves as the base unit for scheduling decisions.

A global tick counter is incremented on every tick interrupt. This counter is used to track delays and determine when suspended tasks should resume execution. Tasks that request a delay specify a duration in ticks, which is converted into an absolute wake-up

time. These tasks are moved to a delayed list and remain inactive until their deadline is reached. During scheduling, the system periodically checks this list and transitions expired tasks back to the ready state.

Time slicing is implemented to ensure fair execution among tasks with the same priority. Each task is allowed to execute for a configurable number of ticks before a scheduling decision is triggered. When the time slice expires, a context switch is requested, allowing other tasks of equal priority to execute. This mechanism is controlled through compile-time configuration, enabling adjustment based on application requirements.

The tick system also supports basic runtime statistics. Each task maintains a counter of the total ticks it has executed, allowing coarse-grained monitoring of CPU usage. This information can be used for debugging, profiling, or system analysis.

To minimize overhead, the tick handler performs only essential operations such as incrementing the global tick count and initiating scheduling when required. More complex operations, including task selection and context switching, are deferred to lower-priority handlers. This separation ensures that interrupt latency remains low while maintaining accurate timing behavior.

6.7 Inter-Task Communication

VAIOS provides a set of inter-task communication (IPC) primitives that enable coordination and data exchange between tasks. These mechanisms are designed to support synchronization, resource sharing, and event-driven execution while maintaining predictable behavior in a preemptive environment.

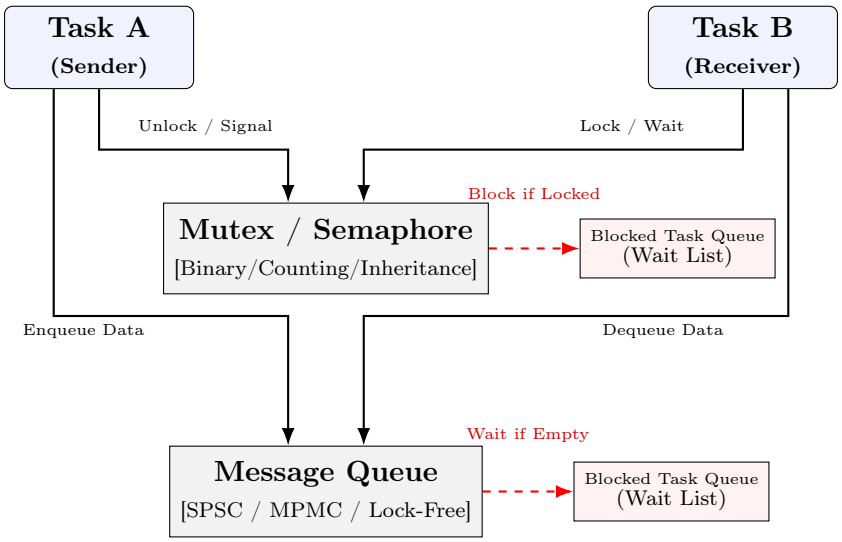

Figure 6.5: Interaction model for VAIOS inter-task communication. Tasks coordinate through kernel primitives, with support for blocking operations and automated wait-list management.

The core IPC primitives include binary semaphores, counting semaphores, mutexes, and recursive mutexes. Semaphores are used for signaling and resource counting, allowing tasks to block until a resource becomes available or an event occurs. Mutexes provide mutual exclusion for shared resources, ensuring that only one task can access a critical

section at a time. Recursive mutexes extend this behavior by allowing the owning task to acquire the same lock multiple times, maintaining an internal recursion count.

Each synchronization primitive maintains an internal wait queue of tasks that are blocked while attempting to acquire the resource. When a resource becomes available, one of the waiting tasks is unblocked and moved back to the ready state. Blocking operations support optional timeouts, allowing tasks to resume execution if the wait condition is not satisfied within a specified number of ticks.

VAIOS also provides ISR-safe variants of certain IPC operations, allowing interrupt handlers to signal tasks without directly performing complex scheduling operations. These functions defer scheduling decisions to the appropriate context, ensuring that interrupt latency remains low while still enabling responsive task wake-up.

In addition to traditional synchronization primitives, VAIOS includes lock-free queue implementations for efficient data transfer between tasks. Single-producer single-consumer (SPSC) and multi-producer multi-consumer (MPMC) queues are provided to support different communication patterns. These structures enable high-throughput data exchange without requiring locks, making them suitable for performance-critical paths.

All IPC mechanisms are tightly integrated with the scheduler. Tasks that block on synchronization primitives are placed into the appropriate wait queues and are automatically transitioned back to the ready state when conditions are met or timeouts expire. This integration ensures that communication and synchronization are handled efficiently and consistently within the system.

6.8 Memory Management

VAIOS provides a lightweight dynamic memory management system to support task creation and system services. The allocator is designed with simplicity, safety, and low overhead in mind, making it suitable for embedded environments where memory resources are limited and predictable behavior is important.

The heap is initialized as a contiguous memory region and managed as a sequence of variable-sized blocks. Each allocated region is preceded by a metadata header that stores information such as block size, allocation status, and a sanity check value. This structure allows the allocator to track memory usage and detect errors such as invalid frees or heap corruption.

Memory allocation follows a linear first-fit strategy. The allocator scans the heap to find a suitable free block or an uninitialized region that can satisfy the requested size. Allocated blocks are aligned to ensure proper access on the target architecture. When a free block is larger than required, it is split into two blocks, allowing efficient utilization of memory.

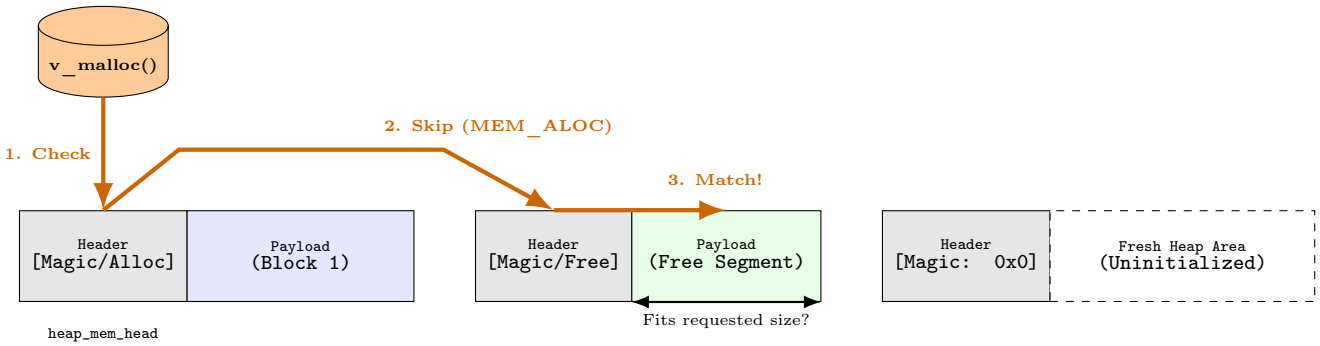

Figure 6.6: Visual representation of the VAIOS first-fit memory allocation search. The allocator traverses the block list, inspecting headers to find the first suitable free or fresh memory region.

Deallocation marks a block as free and attempts to merge it with adjacent free blocks to reduce fragmentation. Both forward and backward coalescing are performed when possible, helping to maintain larger contiguous memory regions over time. This approach balances simplicity with basic fragmentation control without introducing complex data structures.

To ensure correctness in a concurrent environment, all allocation and deallocation operations are protected using critical sections. This guarantees thread safety while keeping the implementation straightforward. Additional safety mechanisms include sanity checks using magic values, detection of double frees, and validation of pointer boundaries within the heap.

VAIOS also incorporates runtime monitoring features such as allocation counters and total allocated size tracking. A configurable heap watermark mechanism is used to detect low-memory conditions and trigger a system panic if predefined thresholds are exceeded. This provides an additional layer of protection in safety-critical applications.

Memory allocation is primarily used for task stacks and control structures, while the overall system behavior remains largely predictable due to controlled usage patterns. The design avoids overly complex allocation schemes, prioritizing reliability and transparency over constant-time allocation guarantees.

6.9 Integration with Vayu

VAIOS serves as the execution backbone for the Vayu system, providing the runtime environment in which all application logic is organized and executed. Within Vayu, system functionality is decomposed into a set of tasks, each responsible for a specific operation such as sensor handling, control computation, communication, or system services.

At system startup, VAIOS is initialized with the required hardware configuration through NavHAL, after which essential tasks are created and scheduled. These tasks typically include core system functions that persist throughout execution, such as control loops, sensor data acquisition, and communication interfaces. Additional tasks may be created dynamically during runtime for specific operations such as calibration, diagnostics, or boot-time checks, and may terminate once their purpose is fulfilled.

Each task operates independently within the scheduling framework, allowing concurrent execution of multiple system components. For example, storage-related opera-

tions can be handled by dedicated tasks that interact with the virtual file system, while computation-heavy tasks such as control algorithms run at higher priorities to ensure timely execution. The use of task-level abstraction enables clear separation of responsibilities and simplifies system design.

The integration also demonstrates how system services provided by VAIOS are utilized in practice. Tasks can perform file operations through the VFS layer, interact with hardware via NavHAL, and use IPC mechanisms for synchronization and data exchange. In the provided example, a dedicated task performs storage benchmarking by writing and reading data through the VFS interface while periodically yielding to maintain system responsiveness. This illustrates how long-running operations can coexist with other system activities without blocking execution.

The lifecycle of tasks in Vayu varies depending on their role. Persistent tasks, such as control and communication modules, remain active throughout system operation, while transient tasks, such as calibration routines or initialization checks, execute once and terminate. Resource cleanup for completed tasks is handled by the system, ensuring efficient memory utilization without disrupting active components.

Overall, the integration of VAIOS with Vayu enables a modular and scalable system design, where functionality is expressed in terms of interacting tasks. This approach provides flexibility in extending system capabilities while maintaining predictable execution behavior required for real-time applications.

6.10 Performance Analysis

The performance of VAIOS was evaluated using a comprehensive benchmark suite executed on 5 March 2026. The benchmarks were designed to assess the behavior of key subsystems including task scheduling, inter-task communication, memory management, floating-point computation, and DMA-based data transfer. All tests were conducted on an STM32F401RE (Cortex-M4) platform with a 1 ms system tick.

The benchmark suite covered a total of 14 test cases spanning multiple subsystems, all of which completed successfully without errors, timeouts, or failures. The results demonstrate stable operation under both isolated and concurrent workloads.

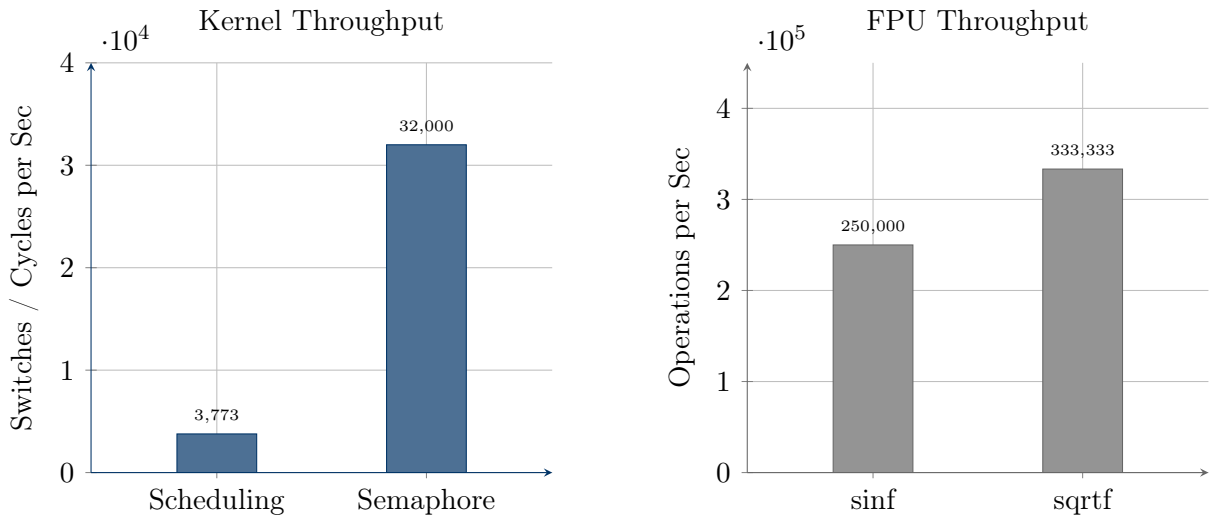

Figure 6.7: Performance benchmarks for VAIOS subsystems on STM32F401RE. The results demonstrate efficient kernel execution and high computational throughput using the hardware FPU.

6.10.1 Experimental Platform

The benchmarks were conducted on a Nucleo-F401RE development board. The configuration details are summarized in Table 6.1.

Field	Value
Board	STM32F401RE (Nucleo-64)
CPU	Cortex-M4 @ 84 MHz
FPU	Enabled (hard-float ABI)
DMA	Enabled (UART logging)
SysTick	1 ms period
Build	Release (-O2)

Table 6.1: Experimental platform configuration for VAIOS performance characterization.

6.10.2 Benchmark Summary

A comprehensive benchmark suite was executed on 5 March 2026 to evaluate the system's performance across five critical sub-domains. The tests were performed on an STM32F401RE platform running at 84 MHz.

Benchmark	Status	Duration	Throughput
FPU: `sinf` throughput	PASS	4 ms	250,000 ops/s
FPU: `sqrtf` throughput	PASS	3 ms	333,333 ops/s
FPU: multi-task context-save	PASS	4 ms	250,000 ops/s
DMA: Memory-to-Memory (Looped)	PASS	20 ms	∼2,000 KB/s
DMA: Concurrent Streams	PASS	21 ms	∼1,904 KB/s
TASK: Context switch rate	PASS	212 ms	3,773 sw/s
TASK: Priority preemption	PASS	65 ms	Verified
TASK: Delay accuracy	PASS	101 ms	1% Error
IPC: Semaphore ping-pong	PASS	31 ms	32,258 trips/s
IPC: Mutex shared counter	PASS	69 ms	Verified
MEM: Alloc/free throughput	PASS	97 ms	6,185 ops/s
MEM: Fragmentation resilience	PASS	—	Coalesced
STRESS: All subsystems concurrent	PASS	8,004 ms	Verified

Table 6.2: VAIOS Benchmark Results Summary. All 14 tests passed, demonstrating robust subsystem integration and performance.

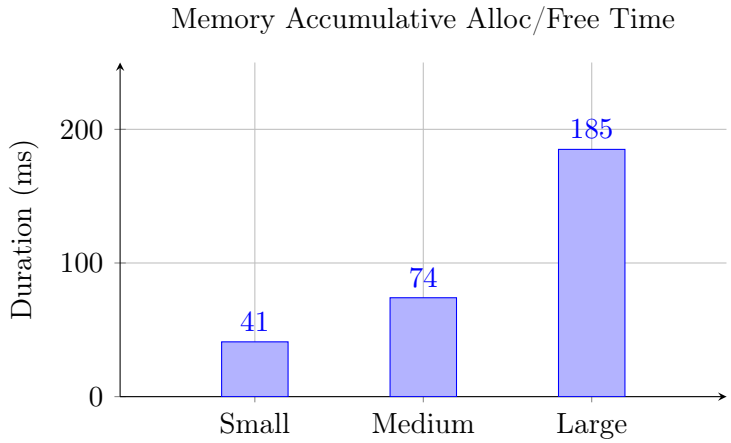

Figure 6.8: Aggregate memory allocation latency for varying workload sizes (Small: 1000 × 32B, Medium: 500 × 128B, Large: 100 × 1024B). The linear search time is proportional to the total heap traversal depth.

Task Scheduling: The scheduler achieved a context switch rate of approximately 3,773 switches per second under release build conditions at 84 MHz. This translates to approximately 22,000 clock cycles per context switch, including PendSV overhead and scheduler logic. Delay accuracy was measured with a 1% error margin (101 ms measured for a 100 ms requested delay), which is the theoretical limit for a 1 ms system tick.

Computational Performance: The hardware FPU demonstrates high efficiency, with `sqrtf` outperforming `sinf` by approximately 33%, consistent with the hardware-accelerated square root instruction set. Multi-tasking tests verified the FPU context preservation logic, with lazy stacking ensuring that only active FPU-using tasks incur the register save/restore overhead.

Subsystem Reliability: A 3,000 ms high-contention stress test successfully executed 200 DMA transfers, 6,902 semaphore trips, and numerous memory operations concurrently. Zero allocation failures and zero data corruption incidents were observed, con-

firming kernel stability under high-load autonomous flight scenarios. Significant reliability fixes were implemented during benchmarking, including a bounded spin-wait in the logging subsystem and optimized DMA flag clearing sequences to prevent spurious interrupts.

6.11 Limitations and Future Work

While VAIOS provides a robust and efficient runtime environment for real-time embedded systems, there are several limitations in the current implementation that present opportunities for future improvement.

One of the primary limitations lies in the use of dynamic memory allocation with a linear first-fit strategy. Although block splitting and coalescing are implemented, allocation time is not constant and may increase with heap fragmentation. For highly constrained or safety-critical systems, a fully static or region-based allocation strategy may provide more predictable behavior. Future work includes exploring alternative allocation schemes such as fixed-size block pools or hybrid memory models to improve determinism.

The scheduling system, while efficient, currently supports a fixed number of priority levels and does not include advanced features such as deadline-based scheduling or priority inheritance across all synchronization primitives. Although basic priority management is supported, extending this to more comprehensive real-time scheduling policies could improve responsiveness in complex workloads.

Inter-task communication is implemented using mutexes, semaphores, and lock-free queues; however, higher-level abstractions such as message queues with structured payloads or event groups are not yet fully developed. Expanding the IPC subsystem to include these features would simplify application-level design and improve expressiveness.

The virtual file system currently relies on a global locking mechanism to ensure thread safety. While this simplifies implementation, it limits concurrent file access and may become a bottleneck in I/O-intensive applications. Future improvements could include finer-grained locking or asynchronous I/O mechanisms to enhance performance.

Although VAIOS supports execution in simulated environments through semihosting, this approach is primarily intended for development and debugging and does not accurately reflect real-time performance characteristics. Enhancing simulation capabilities, including better timing emulation and hardware abstraction layers, would improve testing fidelity.

Platform support is currently focused on Cortex-M4 microcontrollers, with ongoing work to extend compatibility to additional STM32 devices and AVR-based platforms. Broadening hardware support will improve portability and accessibility across a wider range of applications.

Finally, while the current system demonstrates stability under benchmark conditions, long-term reliability features such as fault recovery, watchdog integration, and advanced diagnostics can be further developed. These enhancements will be important for deploying VAIOS in production-grade and safety-critical systems.

Overall, these limitations highlight areas for continued development, with a focus on improving determinism, scalability, and system robustness while preserving the simplicity and efficiency of the current design.

Chapter 7

Vayu

Vayu represents the flight control layer of the system, built on top of NavHAL and VAIOS to provide a complete and functional drone control stack. While NavHAL abstracts the underlying hardware and VAIOS manages execution and system services, Vayu defines the application-level logic that governs sensing, decision-making, and actuation.

The design of Vayu focuses on structuring the flight control problem into well-defined subsystems, each responsible for a specific aspect of system behavior. These subsystems operate as coordinated tasks within a real-time environment, enabling concurrent execution of sensing, estimation, control, and communication processes. This approach allows the system to maintain responsiveness and predictability while handling multiple interacting components.

At its core, Vayu processes data from onboard sensors, estimates the system state, and applies control algorithms to generate actuator commands. These commands are then translated into motor outputs, completing the control loop required for stable flight. Additional functionality such as communication, logging, and system monitoring is integrated into the same framework, allowing the system to operate as a cohesive unit.

This chapter presents the structure and implementation of Vayu, describing how its subsystems interact, how execution is organized, and how real-time constraints are handled within the overall system. The goal is to provide a clear understanding of how the flight control stack is constructed and how it operates in practice.

7.1 Overview

Vayu organizes the flight control system as a set of interacting subsystems that collectively implement the sensing-to-actuation pipeline required for stable flight. Each subsystem is responsible for a specific function, such as sensor acquisition, state estimation, control computation, or output generation, and communicates with others through well-defined interfaces.

Sensor data is acquired from onboard devices, including the BMX160 inertial measurement unit and the BMP180 barometric sensor, using hardware interfaces provided by NavHAL. High-frequency IMU data is continuously updated using a DMA-driven acquisition loop, while lower-rate sensors are accessed periodically. This data is then processed by estimation modules to compute orientation and other state variables required for control.

Control logic is structured into multiple loops operating at different frequencies. A high-frequency rate control loop handles rapid stabilization using gyroscope data, while

a lower-frequency attitude control loop computes desired rates based on orientation estimates. These control outputs are translated into motor commands and applied through hardware timers, enabling precise actuator control.

Execution is managed through VAIOS, where each subsystem operates as an independent task. Time-critical tasks are assigned higher priority and use periodic delays to maintain consistent execution rates. Less critical operations, such as communication, telemetry, and logging, are handled by lower-priority tasks, ensuring that core control functionality remains unaffected under load.

Communication with external systems is supported through UART-based interfaces. Remote control input is received using the iBus protocol, while telemetry and debugging data can be transmitted over serial channels. Additionally, the system supports data logging through the VFS layer, enabling persistent storage of flight data for analysis.

7.2 System Architecture

The architecture of Vayu is organized as a layered and data-driven pipeline, where information flows from sensors to control algorithms and finally to actuator outputs. Each stage of the system is implemented as a set of modular components that interact through well-defined interfaces, enabling clear separation of responsibilities and ease of extension.

At a high level, the system can be viewed as a sequence of processing stages: sensor acquisition, state estimation, control computation, and actuation. Supporting subsystems such as communication and logging operate alongside this pipeline without interfering with time-critical operations.

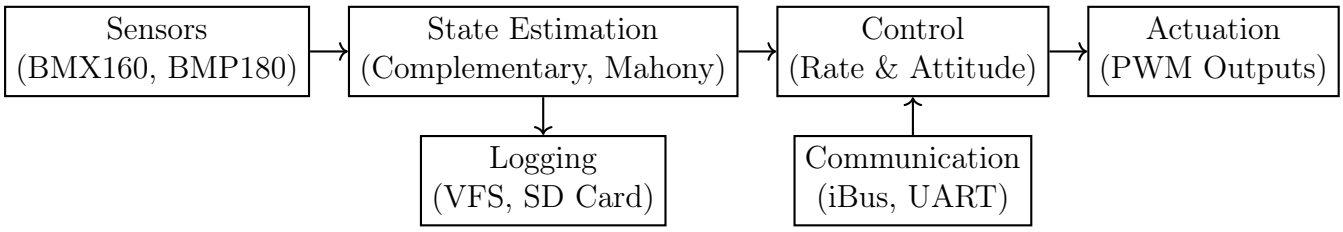

Figure 7.1: High-level architecture of the Vayu flight control system showing the flow from sensing to actuation along with supporting subsystems.

Sensor acquisition forms the first stage of the pipeline. High-frequency inertial data from the BMX160 is continuously updated using a DMA-driven loop, ensuring minimal latency and consistent sampling. Additional sensor inputs, such as barometric pressure from the BMP180, are integrated at lower frequencies to provide complementary information.

The estimation stage processes raw sensor data to compute the system state, including orientation. Multiple estimation algorithms are supported, such as complementary filtering and Mahony filtering, allowing trade-offs between computational cost and accuracy. This stage provides the necessary inputs for downstream control logic.

Control is implemented as a hierarchy of loops operating at different frequencies. A high-frequency rate control loop ensures rapid stabilization by directly regulating angular velocities, while a lower-frequency attitude control loop computes desired rates based

on orientation. This separation enables both responsiveness and stability in the control system.

The actuation stage converts control outputs into hardware signals that drive the motors. In the current implementation, pulse-width modulation (PWM) signals are generated using hardware timers, providing precise and deterministic control of motor outputs.

Supporting subsystems operate alongside the main control pipeline. Communication modules handle remote control input using the iBus protocol and provide telemetry through UART interfaces. Logging and storage are managed through the VFS layer, enabling persistent recording of system data without interfering with real-time execution.

Overall, the architecture of Vayu emphasizes modularity and clear data flow, allowing individual subsystems to be developed, tested, and extended independently while maintaining a cohesive and efficient flight control system.

7.3 Execution Model

The execution model of Vayu is based on a task-oriented design built on top of VAIOS, where each subsystem operates as an independent task scheduled according to priority and timing requirements. This approach enables concurrent execution of sensing, estimation, control, and auxiliary functions while maintaining deterministic behavior required for real-time control.

Each functional component of the system is implemented as a separate task. High-frequency and time-critical operations, such as control loops and sensor acquisition, are assigned higher priorities to ensure minimal latency and consistent execution. Lower-priority tasks handle communication, telemetry, logging, and background operations, allowing the system to remain responsive without impacting control performance.

Periodic execution is achieved using delay-based scheduling. Tasks use fixed-duration delays to maintain consistent execution rates, effectively creating software-timed loops. Due to the preemptive priority-based scheduler in VAIOS, tasks with higher priority resume execution immediately upon wake-up, minimizing jitter and ensuring that critical loops meet their timing constraints.

The control system is structured as multiple loops operating at different frequencies. The rate control loop runs at a high frequency (approximately 1 kHz) to handle rapid stabilization, while the attitude control loop operates at a lower frequency (approximately 250 Hz). This multi-rate design balances responsiveness with computational efficiency. Sensor acquisition tasks and communication handlers operate at frequencies appropriate to their respective roles.

Sensor data acquisition follows a hybrid model. High-frequency IMU data is driven by a DMA-based continuous acquisition loop, where each completed transfer triggers the next read, ensuring uninterrupted data flow. Other sensors and inputs are handled through periodic task execution. This combination allows efficient use of hardware capabilities while maintaining a consistent software interface.

Inter-task communication is achieved using shared buffers and synchronization primitives provided by VAIOS. Data flows between tasks through structured buffers, enabling decoupling between producers and consumers. This ensures that delays or variations in one subsystem do not directly stall others, improving overall system robustness.

The system initialization sequence creates all required tasks and starts the scheduler, after which execution is entirely managed by the RTOS. Persistent tasks, such as control

and communication modules, run continuously, while transient tasks, such as boot-time checks or calibration routines, execute once and terminate.

7.4 Sensor Interface

The sensor interface in Vayu is responsible for acquiring raw data from onboard sensors and making it available to downstream subsystems such as state estimation and control. This layer abstracts the underlying hardware communication details while ensuring timely and reliable data delivery.

The system currently utilizes a Bosch BMX160 inertial measurement unit and a Bosch BMP180 barometric pressure sensor. Both sensors are interfaced over a shared I2C bus managed through NavHAL. The BMX160 provides accelerometer, gyroscope, and magnetometer data required for orientation estimation, while the BMP180 supplies pressure-based altitude information.

High-frequency IMU data acquisition is implemented using a DMA-driven approach. Sensor reads are initiated over I2C, and upon completion of each DMA transfer, an interrupt is generated to trigger the next read operation. This creates a continuous acquisition loop that minimizes CPU involvement and ensures consistent sampling at high rates. By offloading data transfer to DMA, the system reduces latency and avoids blocking execution of other tasks.

In contrast, lower-frequency sensors such as the BMP180 are accessed through periodic task execution. These sensors do not require continuous high-rate sampling and are read at intervals appropriate to their role in the system. This distinction allows efficient utilization of system resources while maintaining adequate data freshness.

Acquired sensor data is stored in shared buffers, enabling decoupling between acquisition and processing stages. Estimation tasks consume data from these buffers without directly interacting with hardware, allowing sensor drivers to remain independent of higher-level logic. This design improves modularity and simplifies integration of additional sensors in the future.

Initialization of sensors is performed during system startup, where communication interfaces are configured and device-specific parameters are set. Once initialized, sensor tasks operate continuously within the execution framework, providing a steady stream of data for the rest of the system.

7.5 State Estimation

The state estimation module in Vayu is responsible for computing the orientation of the system using data from the inertial measurement unit. It fuses accelerometer, gyroscope, and magnetometer measurements to obtain roll, pitch, and yaw angles required by the control system.

Two sensor fusion approaches are implemented: a complementary filter and the Mahony filter. The choice of algorithm is configurable at compile time, allowing the system to balance computational cost and estimation accuracy depending on the application requirements.

7.5.1 Accelerometer and Magnetometer Estimation

The accelerometer and magnetometer provide absolute orientation references. The accelerometer is used to estimate roll and pitch based on the direction of gravity, while the magnetometer provides heading information.

Roll and pitch are computed as:

$$\phi = \tan^{-1}\left(\frac{-a_y}{a_z}\right), \quad \theta = -\tan^{-1}\left(\frac{a_x}{\sqrt{a_y^2 + a_z^2}}\right)$$

Yaw is computed using tilt-compensated magnetometer measurements:

$$\psi = \tan^{-1}\left(\frac{-m'_y}{m'_x}\right)$$

where the magnetometer readings are compensated using the estimated roll and pitch.

These estimates are stable but noisy and are therefore combined with gyroscope integration.

7.5.2 Complementary Filter

The complementary filter combines the short-term stability of gyroscope integration with the long-term stability of accelerometer and magnetometer measurements.

The orientation is updated as:

$$\theta = \alpha\left(\theta + \omega \cdot dt\right) + (1 - \alpha)\theta_{\text{acc/mag}}$$

where:

- θ is the estimated angle
- ω is the angular velocity from the gyroscope
- dt is the time step
- α is the blending factor (typically close to 1)

This approach provides a computationally efficient solution suitable for real-time systems, with minimal processing overhead.

7.5.3 Mahony Filter

The Mahony filter is a quaternion-based sensor fusion algorithm that provides improved accuracy by incorporating feedback correction. It estimates orientation using gyroscope integration while correcting drift using accelerometer and magnetometer measurements.

The error between measured and estimated directions is computed as:

$$\mathbf{e} = \mathbf{v}_{\text{measured}} \times \mathbf{v}_{\text{estimated}}$$

This error is used to correct the gyroscope measurements:

$$\omega_{\text{corrected}} = \omega + K_p\mathbf{e} + K_i \int \mathbf{e}\, dt$$

The quaternion is then updated as:

$$\dot{q} = \frac{1}{2} q \otimes \omega_{\text{corrected}}$$

where:

- q is the orientation quaternion
- K_p and K_i are proportional and integral gains

The quaternion is normalized after each update to maintain numerical stability and is converted to Euler angles for use by the control system.

7.5.4 Implementation Considerations

The time step dt is computed dynamically using the processor cycle counter, allowing accurate integration even under varying execution timing. Sensor measurements are normalized before use to ensure stability of the estimation algorithms.

The complementary filter provides a lightweight solution suitable for lower computational load, while the Mahony filter offers improved robustness and drift correction at the cost of additional computation. The availability of both approaches allows flexibility in adapting the system to different performance requirements.

7.6 Control System

The control system in Vayu generates actuator commands required to stabilize and control the drone based on estimated state and user inputs. It is implemented as a cascaded architecture consisting of an outer attitude loop and an inner rate loop, operating at different frequencies to balance stability and responsiveness.

7.6.1 Control Architecture

The system follows a hierarchical structure where the attitude controller computes desired angular rates, which are then tracked by the rate controller. This separation enables robust stabilization while allowing intuitive control through orientation commands.

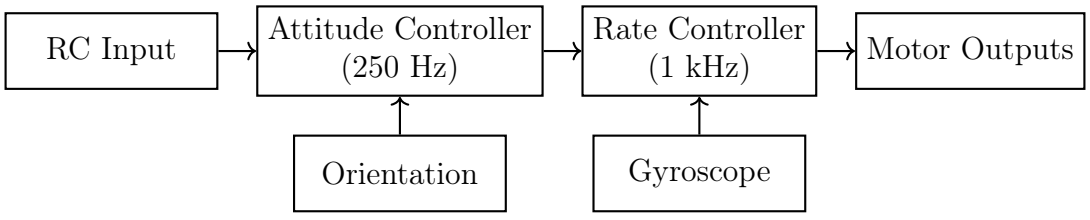

Figure 7.2: Cascaded control structure in Vayu.

The attitude controller converts user commands into target angles and computes desired angular rates using PID control. The rate controller then tracks these target rates using gyroscope feedback and produces control outputs for each axis.

7.6.2 PID Controller Formulation

The controller output is computed as:

$$u = K_p e + K_i \int e\,dt + K_d \frac{d}{dt}(-y) + K_{ff}\dot{r}$$

where $e = r - y$ is the error between setpoint and measurement.

7.6.3 Implementation Details

The implementation incorporates several practical enhancements:

Derivative on Measurement:

$$D = -K_d \frac{y(t) - y(t - dt)}{dt}$$

This avoids derivative kick during rapid setpoint changes.

Low-Pass Filtering:

$$D_f = D_f + \alpha(D - D_f), \quad \alpha = \frac{dt}{dt + RC}$$

Integral Anti-Windup:

$$I = \text{clamp}(I + K_i e dt, -I_{\max}, I_{\max})$$

Integration is limited when actuator saturation occurs to prevent windup.

Feedforward Term: A feedforward component based on setpoint rate improves responsiveness.

Multi-Rate Execution: The rate loop operates at approximately 1 kHz for fast stabilization, while the attitude loop runs at approximately 250 Hz for smoother control.

7.6.4 Data Flow Between Controllers

The attitude controller produces desired angular rates which are passed to the rate controller through a lock-free buffer mechanism, ensuring non-blocking communication between task. The rate controller then computes final control outputs using real-time gyroscope measurements.

7.7 Actuation and Output

The actuation subsystem in Vayu converts control outputs into motor commands that drive the drone. It forms the final stage of the control pipeline, translating abstract control signals into hardware-level outputs.

7.7.1 Motor Mixing

The control outputs from the rate controller correspond to roll, pitch, and yaw corrections. These are combined with throttle input to generate individual motor commands using a mixing algorithm specific to the quadrotor configuration.

For a quadrotor layout, the motor outputs are computed as:

$$M_1 = T - R + P + Y$$
$$M_2 = T - R - P - Y$$
$$M_3 = T + R - P + Y$$
$$M_4 = T + R + P - Y$$

where:

- T is throttle
- R, P, Y are roll, pitch, and yaw control outputs

This mixing ensures that control inputs are properly distributed across all motors to achieve the desired motion.

7.7.2 Output Normalization

To maintain valid actuator commands, the motor outputs are constrained within allowable limits. If any output exceeds bounds, all outputs are scaled proportionally to preserve control direction while ensuring safe operation. Similarly, negative outputs are shifted to maintain non-negative motor commands.

7.7.3 Throttle Handling

At low throttle values, control authority is reduced proportionally to prevent instability during arming and low-power operation. This ensures smooth motor behavior and avoids abrupt responses.

7.7.4 Hardware Output

The final motor commands are applied using pulse-width modulation (PWM) generated by hardware timers (TIM1). This provides precise and deterministic control of motor speeds with minimal CPU overhead.

7.7.5 System Behavior

The actuation pipeline ensures that control outputs are translated into smooth and stable motor responses. The combination of mixing, normalization, and hardware-timed output enables reliable execution of control commands in real time.

Overall, the actuation subsystem completes the control loop by bridging the gap between control computation and physical system response.

7.8 Communication and Telemetry

The communication subsystem in Vayu enables interaction between the flight controller and external systems, including remote control input, command processing, and real-time telemetry transmission. It is designed to handle both low-latency control input and structured data exchange without interfering with time-critical control tasks.

7.8.1 Remote Control Input

User input is received using the iBus protocol over UART with DMA-based reception. A circular DMA buffer is used to continuously capture incoming data, minimizing CPU overhead and ensuring reliable reception at high update rates.

Incoming bytes are parsed incrementally, and valid frames are extracted and stored in shared buffers for use by control tasks. A normalization stage converts raw channel values into standardized control inputs, including deadband handling and scaling.

The system also integrates state management with RC input. Arming and disarming logic is derived from switch positions and throttle levels, enabling safe transitions between system states such as standby, armed, and failsafe. :contentReference[oaicite:0]index=0

7.8.2 Command Processing

In addition to RC input, Vayu supports command-based communication through a packet-based protocol. Incoming packets are decoded and dispatched based on type, allowing execution of system-level commands.

Supported commands include:

- Heartbeat synchronization
- Device identification
- Sensor calibration (e.g., IMU calibration)
- Runtime task management

Certain commands dynamically create or terminate tasks, enabling flexible runtime behavior without requiring system restart.

7.8.3 Telemetry Architecture

Telemetry is transmitted using a structured packet-based protocol over a configurable communication channel. Data is organized into different packet types, allowing selective transmission of system information.

The telemetry system operates on a multi-rate schedule, where different data streams are transmitted at different frequencies:

- High-frequency IMU data (compressed and full)
- Attitude estimates
- RC input data

- Motor outputs
- Control loop diagnostics (PID data)
- System state and status
- Logging data

This design ensures efficient bandwidth utilization while prioritizing critical information.

7.8.4 Adaptive Data Streaming

To reduce bandwidth usage while maintaining data fidelity, the system employs delta compression for high-rate sensor data. Instead of transmitting full values every cycle, differences between successive samples are encoded using reduced precision formats.

Full data frames are transmitted periodically to maintain synchronization, while intermediate updates use compressed representations. This approach balances accuracy and communication efficiency.

7.8.5 Logging Integration

The telemetry subsystem is tightly integrated with the logging system. Log messages are buffered and transmitted as part of the telemetry stream, enabling real-time debugging and monitoring without disrupting control execution.

7.8.6 System Behavior

The communication system is designed to operate asynchronously with respect to control tasks. Input processing, command handling, and telemetry transmission are handled by dedicated tasks running at lower priorities. This ensures that communication overhead does not interfere with time-critical operations such as control loops.

7.9 Data Logging and Storage

The data logging subsystem in Vayu provides persistent storage of system data for debugging, analysis, and future extensions such as flight data recording. It is built on top of the virtual file system (VFS) layer and is designed to ensure reliable operation without impacting real-time performance.

7.9.1 File-Based Logging Architecture

Logging is organized into multiple categories, each mapped to a dedicated file on storage. The current implementation supports separate logging streams for:

- System-level events
- Communication (Navlink) data

- General-purpose logging

Each logging stream operates independently, allowing selective recording of different types of data without interference.

7.9.2 Preallocation and Deterministic Writes

To ensure predictable performance, log files are preallocated to fixed sizes during system initialization. This avoids dynamic file growth and reduces fragmentation, which is critical for maintaining consistent write latency in embedded storage systems.

Before logging begins, each file is expanded to its configured size and initialized, ensuring that all subsequent writes occur within a known memory region.

7.9.3 Circular Logging Mechanism

Logging is implemented using a circular buffer approach at the file level. Each log file maintains a write pointer that advances with each write operation. When the end of the file is reached, the write pointer wraps around to the beginning, overwriting older data.

$$\text{write\_pos} = (\text{write\_pos} + \text{len}) \bmod \text{file\_size}$$

This approach ensures continuous logging without requiring explicit file management or risking storage exhaustion.

7.9.4 Thread-Safe Operation

To support concurrent access from multiple tasks, each logging stream is protected using mutexes. This guarantees safe and consistent writes even when multiple subsystems attempt to log data simultaneously.

Write operations follow a structured sequence:

1. Acquire mutex
2. Seek to current write position
3. Write data to storage
4. Synchronize file to ensure persistence
5. Update write pointer
6. Release mutex

This design ensures data integrity while maintaining simplicity.

7.9.5 Integration with System Components

The logging subsystem is integrated with other parts of Vayu, including communication and telemetry. Log data can be generated by various modules and stored persistently, while selected logs may also be transmitted in real time through the telemetry system.

Currently, the logging system is used to store calibration data and system-level information. These stored values can be retrieved across system restarts, enabling persistent configuration.

7.9.6 Future Extensions

The logging infrastructure is designed to support future expansion. Planned extensions include:

- Full flight data recording (sensor, control, and state data)
- Blackbox-style logging for post-flight analysis
- Configurable logging levels and formats
- Efficient binary encoding for high-frequency data

With the underlying VFS and SD card support already in place, these features can be integrated without significant changes to the system architecture.

7.9.7 System Behavior

The logging subsystem operates independently of time-critical control tasks. By using preallocation, circular buffering, and mutex protection, it ensures that storage operations remain predictable and do not introduce blocking behavior in high-priority tasks.

7.10 System Initialization and Boot Flow

The system initialization and boot flow in Vayu defines the sequence of operations that transition the system from power-on to a fully operational state. This process ensures that hardware, system services, and application tasks are correctly initialized before entering normal operation.

7.10.1 Initialization Sequence

System startup begins with low-level hardware configuration, followed by initialization of system services and application components. The sequence is structured to ensure that all dependencies are satisfied before tasks begin execution.

The initialization flow proceeds as follows:

1. **Clock Configuration:** The system clock is configured using the phase-locked loop (PLL) to achieve the required operating frequency. This ensures consistent timing for all subsystems.
2. **Core System Initialization:** VAIOS is initialized using a configuration structure that defines system behavior, including clock handling and peripheral setup.
3. **Peripheral Initialization:** Hardware interfaces such as I2C and UART are initialized through NavHAL. This includes setting up DMA for sensor acquisition and communication channels.
4. **Subsystem Initialization:** Core subsystems including logging, system state management, sensor interfaces, and communication buffers are initialized.

5. **Task Creation:** System and application tasks are created and registered with the scheduler. These include control tasks, sensor acquisition tasks, communication handlers, and auxiliary tasks.

6. **Timer Setup:** Periodic timer callbacks are configured to support time-based operations and high-frequency timing requirements.

7. **Scheduler Start:** The VAIOS scheduler is started, after which all task execution is managed by the RTOS.

7.10.2 Boot Task and System Checks

A dedicated boot task performs system validation checks during startup. This task is responsible for verifying that critical components are functioning correctly before enabling normal operation.

The boot checks include:

- System clock validation
- Storage subsystem availability
- Basic system readiness verification

The results of these checks are accumulated into a boot status register, which is used to determine the final system state.

If all required checks pass, the system transitions to a standby state, indicating readiness for operation. If any check fails, the system enters a failsafe state to prevent unsafe behavior.

7.10.3 State Transition

The system state progresses through a well-defined sequence:

$$\text{INIT} \rightarrow \text{STANDBY} \rightarrow \text{ARMED}$$

During initialization, the system remains in the *INIT* state until all boot checks are completed. Once validated, it transitions to *STANDBY*, where it awaits user input. The transition to the *ARMED* state is controlled by external input (e.g., RC commands).

7.10.4 Task Activation

After the scheduler starts, all created tasks begin execution based on their assigned priorities. High-priority tasks such as control loops and sensor acquisition become active immediately, while lower-priority tasks handle communication, logging, and background operations.

The boot task terminates itself after completing initialization, ensuring that it does not consume system resources during normal operation.

7.10.5 System Behavior

The structured initialization flow ensures that all components are properly configured before entering real-time execution. By separating initialization, validation, and runtime operation, the system maintains robustness and avoids undefined behavior during startup.

7.11 Task Organization

Vayu organizes system functionality as a collection of independent tasks executed within the VAIOS scheduling framework. Each task is responsible for a specific subsystem, allowing clear separation of concerns and predictable execution behavior.

7.11.1 Task Classification

Tasks in Vayu can be broadly classified into three categories based on their role in the system:

Control Tasks: These tasks implement the core flight control pipeline and are assigned higher priorities to ensure timely execution. They include:

- Attitude controller task
- Rate controller task
- Motor output task

These tasks operate at high frequencies and are critical for maintaining system stability.

Sensor and Data Acquisition Tasks: These tasks are responsible for collecting and preparing data required by the control system. They include:

- IMU acquisition task (DMA-driven)
- RC input processing task

Sensor tasks ensure continuous data availability while minimizing CPU overhead.

Communication and Auxiliary Tasks: These tasks handle non-critical operations such as communication, telemetry, logging, and system services:

- Communication processor task
- Telemetry task
- Logging and storage tasks
- Heartbeat and monitoring tasks

These tasks run at lower priorities and are designed to operate without interfering with control loops.

7.11.2 Task Prioritization

Task priorities are assigned based on timing requirements. Control tasks are given higher priority to ensure minimal latency, while communication and background tasks are assigned lower priorities. Sensor acquisition tasks may use a combination of hardware-driven mechanisms (e.g., DMA) and task scheduling to balance efficiency and responsiveness.

This priority-based organization ensures that time-critical operations are executed deterministically, even under system load.

7.11.3 Inter-Task Communication

Tasks communicate using shared buffers and lock-free data structures. For example, the attitude controller passes target rates to the rate controller through a single-producer single-consumer (SPSC) queue, enabling efficient and non-blocking data transfer.

Similarly, sensor data and telemetry information are exchanged through dedicated buffers, allowing decoupling between producers and consumers. This design prevents delays in one subsystem from propagating to others.

7.11.4 Task Lifecycle

Tasks in Vayu follow different lifecycle patterns depending on their role:

- **Persistent tasks:** Core tasks such as control, sensor acquisition, and communication run continuously throughout system operation.
- **Transient tasks:** Certain tasks, such as boot checks or calibration routines, are created dynamically and terminate after completing their function.

Dynamic task creation allows the system to adapt to runtime requirements without increasing baseline resource usage.

7.11.5 Execution Characteristics

Each task operates as a periodic loop using delay-based scheduling. Combined with the preemptive scheduler, this ensures that higher-priority tasks resume execution immediately upon wake-up, minimizing jitter and maintaining consistent timing.

The use of separate stacks for each task ensures isolation and simplifies context management, while the scheduling framework guarantees fair execution among tasks of the same priority.

7.11.6 System Behavior

The task-based organization enables Vayu to execute multiple subsystems concurrently while maintaining clear boundaries between them. By assigning responsibilities to individual tasks and coordinating them through structured communication, the system achieves both modularity and real-time performance.

7.12 Extensibility

Vayu is designed with extensibility as a core principle, enabling the system to evolve with new hardware, features, and application requirements without requiring major architectural changes. This is achieved through modular design, clear separation of responsibilities, and configurable system components.

7.12.1 Modular Architecture

The system is organized into independent subsystems such as sensing, estimation, control, communication, and logging. Each subsystem interacts with others through well-defined interfaces, allowing components to be modified or replaced without affecting the rest of the system.

For example, sensor drivers are decoupled from estimation algorithms through shared buffers, enabling new sensors to be integrated with minimal changes. Similarly, control algorithms can be extended or replaced while maintaining the same input-output structure.

7.12.2 Configurable Components

Many aspects of the system are configurable at compile time through centralized configuration headers. This includes:

- Selection of estimation algorithms (e.g., complementary or Mahony filter)
- Control gains and limits
- Scheduling parameters such as task priorities and time slices
- Logging and telemetry settings

This approach allows the system to be adapted for different use cases without modifying core logic.

7.12.3 Task-Based Expansion

The use of a task-oriented execution model enables straightforward addition of new functionality. New features can be implemented as independent tasks and integrated into the system without disrupting existing components.

Examples of extensible functionality include:

- Additional sensors (e.g., GPS, vision modules)
- Advanced control algorithms
- Navigation and mission planning modules
- Diagnostics and monitoring tools

Dynamic task creation further allows certain features, such as calibration or testing routines, to be invoked only when needed.

7.12.4 Hardware Abstraction

NavHAL provides a hardware abstraction layer that isolates Vayu from microcontroller-specific details. This allows the system to be ported across different microcontrollers with minimal changes to higher-level logic.

Current support focuses on Cortex-M4-based systems, with ongoing work to extend compatibility to additional microcontrollers, including other STM32 variants and AVR-based platforms.

7.12.5 Storage and Data Expansion

The logging and VFS infrastructure enables persistent storage of system data, which can be extended to support additional use cases such as full flight data recording, configuration storage, and post-flight analysis.

Since the storage layer is abstracted through a virtual file system, new storage backends or formats can be integrated without modifying application logic.

7.12.6 Communication Flexibility

The communication subsystem supports structured packet-based data exchange and can be extended to support additional interfaces such as wireless communication or higher-level protocols. This enables integration with ground stations, mobile applications, or distributed systems.

7.12.7 System Evolution

The overall design of Vayu supports incremental development. New features can be introduced, tested, and integrated without requiring a redesign of the system. This makes the platform suitable for both experimental development and deployment in evolving applications.

7.13 Current Status and Roadmap

7.13.1 Current Status

Vayu has reached a functional stage where the core flight control pipeline is fully implemented and operational. The system supports end-to-end execution from sensor acquisition to motor actuation, enabling closed-loop control suitable for real-time applications.

The current implementation includes:

- Sensor integration using BMX160 (9-axis IMU) and BMP180 (barometer)
- State estimation using complementary and Mahony filters
- Cascaded control system with rate (1 kHz) and attitude (250 Hz) loops
- PWM-based motor actuation using hardware timers
- RC input via iBus protocol with DMA-based reception

- Structured telemetry system with multi-rate data streaming
- Persistent logging using VFS with SD card support
- Preemptive RTOS-based execution using VAIOS

The system is capable of stable real-time operation with modular task-based execution. Core subsystems have been tested in isolation and integrated environments, and the overall architecture has been validated through system-level execution.

7.13.2 Roadmap

The development of Vayu is ongoing, with focus on improving robustness, feature completeness, and system scalability. The roadmap is structured into short-term and long-term goals.

Short-Term Goals:

- Hardware validation through flight testing
- Sensor calibration improvements and automated routines
- Tuning and optimization of control loops
- Enhanced failsafe mechanisms and safety features
- Improved telemetry visualization and ground interface tools

Mid-Term Goals:

- Integration of additional sensors (e.g., GPS, magnetometer refinement)
- Advanced state estimation techniques
- Flight data recording (blackbox logging)
- Support for additional communication interfaces

Long-Term Goals:

- Autonomous flight capabilities and navigation stack
- Multi-vehicle coordination and networking
- Expansion to multiple hardware platforms
- Integration with higher-level mission planning systems

7.13.3 Development Direction

The long-term vision of Vayu is to provide a fully indigenous, modular, and high-performance flight control stack that can be adapted across a wide range of applications. Development will continue to focus on maintaining a balance between performance, simplicity, and extensibility, ensuring that the system remains both practical and scalable.

Chapter 8

Conclusion

This work presented the design and implementation of a complete, indigenous flight control stack consisting of NavHAL, VAIOS, and Vayu. The system was developed with the objective of achieving full control over the embedded stack while maintaining modularity, real-time performance, and scalability.

The motivation for this work originated from practical challenges encountered while working with existing drone platforms. Extending system functionality, such as implementing secure telemetry, required complex external workarounds involving additional hardware and software layers. These experiences highlighted the limitations of current approaches, where developers are often constrained by rigid architectures and forced to build around the system rather than within it.

In response to these challenges, this work adopts a first-principles approach to system design. NavHAL provides a lightweight and deterministic hardware abstraction layer, enabling direct interaction with microcontroller peripherals without unnecessary overhead. VAIOS builds on this foundation to deliver a preemptive real-time operating system with priority-based scheduling, task isolation, and efficient inter-task communication. Together, these layers establish a controlled and predictable execution environment.

Vayu, the flight control layer, integrates sensing, estimation, control, and actuation into a cohesive system. By structuring functionality as coordinated real-time tasks, the system achieves clear separation of concerns while maintaining deterministic behavior. The use of multi-rate control loops, DMA-driven sensor acquisition, and optimized communication and logging mechanisms enables efficient and responsive operation suitable for embedded flight control applications.

A key outcome of this work is the demonstration that high-performance flight control systems can be built from scratch with full ownership of the software stack. By minimizing dependencies on external frameworks and hardware-specific constraints, the system provides greater flexibility, improved transparency, and easier adaptability for new features and applications.

The current implementation establishes a functional and extensible foundation, with core subsystems validated through integrated system execution. However, further work is required to transition the system toward deployment-ready maturity. This includes comprehensive flight testing, enhanced safety and fault-handling mechanisms, and refinement of estimation and control algorithms.

Looking forward, the system is designed to evolve as a broader platform for robotics and autonomous systems. Planned developments include support for additional hardware

platforms, advanced state estimation techniques, full flight data logging, and integration of higher-level navigation and mission planning capabilities.

In summary, this work represents a shift from assembly-based development toward building complete systems with full architectural ownership. It provides a structured and scalable foundation for developing indigenous, flexible, and high-performance drone and robotic systems, enabling future advancements in autonomy and embedded intelligence.