جامعة سطيف 1
UNIVERSITY OF SETIF 1

Development of Modular Robotic Platforms

for Development, Research and Education

Presented by: Assil M. Ferahta
For the degree of **Masters** in **Embedded Systems Electronics**

University: Setif 1 Ferhat Abbas
Thesis Supervisor: Dr. Younes Terchi
Date: 2, 2025

Acknowledgments

Abstract

The development of modular robotic platforms has a set of problems with multiple possible solutions, therefore, defining a good concept to abide by is a good idea in order to realize such platforms and assure their flexibility, adaptability and uniqueness. In this thesis I will present my philosophies regarding the development of modular, upgradable and interchangeable components to form certain families of robotic platforms, as well as using them together to test different related concepts and emergent behavior between them.

Contents

General Introduction

Robotics has emerged as a pivotal technology in fields ranging from industrial automation and research to education and personal innovation. Despite rapid technological advances, a gap persists in providing accessible, versatile, and modular platforms that can cater to a diverse audience, ranging from researchers and students to hobbyists and ordinary users with minimal technical background. This thesis addresses this gap by introducing a set of modular robotic platforms; Bulky, Thegill, and ţtomba, Dronegaze, alongside controllers like iLite and software tools for development and tinkering and interfacing which are designed with adaptability and ease of use in mind.

At the heart of this work is the idea of modularity as a means to bridge diverse domains from rigorous academic research and hands-on educational experimentation to hobbyist tinkering and everyday problem solving. Bulky is designed as a precision rover that integrates a flexible sensor-ready framework, exemplifying how a platform can evolve by accommodating custom modules. Thegill challenges conventional boundaries by operating seamlessly across terrestrial and aquatic environments, pushing the envelope on how modular components can harmonize mechanical adaptability with sophisticated control strategies. Meanwhile, ţtomba distills robotics to its essentials, replacing rigid circuit boards with a breadboard setup to foster an environment of learning, rapid prototyping, and creative exploration.

Together, these platforms are not merely products with predefined applications; each platform embodies a distinct set of features and design philosophies while adhering to a common vision: to democratize robotics by offering modular and upgradable solutions that can be customized, extended, and integrated into various applications. They are conceptual testbeds that encourage users to tap on robotics via open design and abstraction. This thesis systematically explores the theoretical foundations of modular design, the practical integration of mechanical, software and electronic systems, and the experimental validation of these concepts. By doing so, it aims to demonstrate that embracing modularity and abstraction can transform robotics into a universally accessible and continually evolving field, empowering researchers, students, enthusiasts, and even everyday users to craft adaptive solutions in parallel of what is being developed in the industrial world.

This thesis documents the philosophy, problematic, design, implementation, and evaluation of these platforms, demonstrating how a modular approach can address a

variety of challenges in robotics. It presents the theoretical foundations of modular design, details the system architecture, and evaluates the platforms through experimental testing. Ultimately, the work illustrates how such adaptable solutions can serve as universal products, empowering users to explore innovative applications in research, education, and beyond.

0.1 Introduction to Modularity in Robotic Systems

Modularity is a design philosophy that emphasizes constructing complex systems from smaller, self-contained componentscommonly referred to as modules[1]. In the context of robotic platforms, this approach involves decomposing the system into discrete parts, each encapsulating specific functionalities such as sensing, actuation, communication, or processing, thereby simplifying design and troubleshooting.

At its core, modularity offers several significant benefits:

- **Reusability**[2]: Successfully developed and tested modules can be reused across different configurations or projects.
- **Flexibility**[3]: Standardized interfaces[4] and communication protocols enable modules to be interchanged or reconfigured, a feature particularly valuable in rapid prototyping and iterative design.
- **Scalability**[5]: New modules can be integrated without necessitating a complete redesign, making it straightforward to expand functionality or incorporate emerging technologies.
- **Ease of Maintenance**[6]: When a single module fails or becomes outdated, it can be replaced or upgraded without affecting the entire system.

In educational environments, modular robotic platforms serve as excellent teaching tools. They allow students to visualize and understand the intricate relationships

[1]Modules are discrete units of functionality that can be independently developed, tested, replaced, or upgraded.

[2]Reusability refers to the ability to use developed modules in various configurations or projects, reducing overall development time and costs.

[3]Flexibility is the capacity to interchange or reconfigure modules to adapt to new requirements or experimental setups.

[4]Interfaces are predefined methods for communication between modules, ensuring compatibility and ease of integration.

[5]Scalability describes the ease of integrating additional modules into an existing system as project requirements evolve.

[6]Ease of Maintenance refers to the capacity to replace or upgrade individual modules independently, thereby minimizing disruptions to the overall system.

between individual system components and the overall architecture. By experimenting with different module combinations, learners gain practical insights into system integration, problem-solving, and the inherent trade-offs in design decisions.

From a development perspective, modularity supports a more agile workflow. It enables parallel development, where different teams or individuals work simultaneously on separate modules. Upon integration, the overall system benefits from collective innovations, ensuring a robust and versatile platform. This design strategy also opens avenues for further innovation, as new modules can be developed to enhance system capabilities without overhauling the existing architecture.

Ultimately, adopting a modular approach in robotic platforms aligns perfectly with the goals of advancing research, fostering educational experiences, and enabling rapid prototyping. By leveraging the principles of modularity, developers can create adaptable and cost-effective platforms that meet the dynamic challenges of modern robotics.

0.2 Notion of platforms

Platforms can be considered a workspace that provides mechanical, electronic, software as well as energetic and power support for a project be it experimental or practical, but at the heart of this concept is "Abstraction", you don't need to understand how semiconductor or hardware works to work with an arduino for an instance, all you need is to be familiar with algorithms and programming; high level programming or even human language in our times of AI, no assembly or machine code anymore, but if you need to or want to go that low, it's still possible for you to do so, the arduino platform allows for all therefore:

Definition. A *platform* is a deliberately stable core together with explicit contracts that enable independent parties to create, combine, and evolve extensions and applications over time without modifying the core.

0.2.1 Platform vs. Product, Toolkit, Framework, Ecosystem

0.2.2 Modular Robotic Platforms (MRP)

An MRP exposes mechanical, electrical, and software contracts for safe composition of actuators, sensors, power, and behaviors.
"ASCE" platforms; what this thesis is about, follows an analogy: "if modularity is a universe, then our platforms allow freedom in both **space** and **time**"

Space-Time Modularity

Space = discipline (mechanical, electronics, software, logic/UX). Time = user maturity (consumer, learner, researcher, professional).

0.2.3 The ASCE Platform Family

0.2.4 Minimum Viable Platform (MVPf)

0.2.5 Axioms and Criteria

0.2.6 Platform Layers for Embedded Robotics

0.2.7 Contract Template

0.2.8 Why ASCE is Different

1 Defining Platforms

1.1 Platform: Working Definition

A *platform* is a deliberately stable core together with explicit contracts that enable independent parties to create, combine, and evolve extensions and applications over time without modifying the core.

Formally, we define a platform as the 5-tuple

$$\mathcal{P} = \langle \mathcal{C}, ; \Gamma, ; \mathcal{E}, ; \mathcal{T}, ; \mathcal{G} \rangle, \tag{1.1}$$

where $\mathcal{C}$ is the core, Γ are the contracts (interfaces and guarantees), $\mathcal{E}$ are extensions, $\mathcal{T}$ are the tools, and $\mathcal{G}$ is governance.

These notions are commonly used in software design and computer science but we can apply the same philosophies behind them in our field of interest; robotics.

1.2 Origin of context

The notion of a *stable core with explicit contracts* emerges from three converging streams. First, modular design and information hiding established that software should be decomposed so that each module encapsulates design decisions behind stable interfaces, allowing implementations to vary without ripple effects [**parnas1972**]. Second, *Design by Contract* made those interface expectations explicit as preconditions, postconditions, and invariants, turning interfaces into formal *contracts* between components [**meyer1992**].

As software systems turned into industry platforms, researchers described architectures with a *core* and a *periphery of complements*, where the cores stability and the clarity of its interfaces enable independent evolution of complements. This core-complement view connects technical modularity to economic ecosystems and governance, giving a unified lens for platform design [**baldwin2009platforms**].

In practice, the software-architecture community operationalized the idea through patterns such as the Hexagonal (Ports & Adapters) Architecture: keep application logic independent of technology details, expose *ports* as stable contracts, and attach

adapters for specific UIs, devices, or protocols. This preserves the core while enabling rapid replacement or addition of peripherals [**cockburn2005hexagonal**].

Management and IS scholarship then integrated architecture with strategy and governance: how a platform owner maintains core stability, versioning, and compatibility while orchestrating third-party innovation. This literature articulates how interface design, API policies, and deprecation rules shape ecosystem health, providing the governance scaffolding that complements the technical contracts [**gawer2002platformleadership**, **tiwana2014platformecosystems**].

Taken together, these threads justify the dissertations framing: a platform is a deliberately stable core with explicit, versioned contracts that empower independent parties to create and evolve extensions over time, without modifying that core or building on top of it. This "core" may be rigid and not as flexible outside intended use cases, but we can add that missing dimension by making it "Modular"; the core is merely a solid kernel that allows for a huge variety of stock or third party modules (or adapters in the sense of hexagonal architecture) to plug in, letting the users shape it as they please (Spacetime liberty!).

1.2.1 Platform vs. Product, Toolkit, Framework, Ecosystem

The magic behind ***ASCE*** platforms is that aside from being platforms, they may also behave as products, toolkits frameworks, and an ecosystem (Since modularity and hexagonal architecture philosophies allow for cross platform and emergent behavior).

Table 1.1: Platform vs. related constructs

Construct	**Core intent**	**Who extends it?**	**Interface stability**	**Typical deliverables**
Product	Solve a specific use case	Vendor	n/a	Device/app with fixed features
Toolkit	Provide parts to assemble	End user	Low–Medium	Components, examples
Framework	Invert control; fill hooks	Developer	Medium–High	Base code + callbacks
Ecosystem	Community of complements	Many parties	Mixed	Marketplaces, standards
Platform	**Stable core + contracts for many products**	**First&third parties**	**High (versioned)**	**Core, interfaces, SDKs, docs, governance**

1.3 Modular Robotic Platforms (MRP)

A *Modular Robotic Platform* (MRP) exposes mechanical, electrical, and software contracts for safe composition of actuators, sensors, power, and behaviors so that a wide range of robots can be built, upgraded, and repurposed without modifying the core.

1.3.1 Space-Time Modularity

Space is *discipline* (mechanical, electronics, software, logic/UX). Time is *user maturity* (consumer, learner, researcher, professional). A high-quality MRP provides variation points across space and growth paths across time.

Table 1.2: ASCE Space–Time matrix of modularity

Time ↓ / Space →	**Mechanical**	**Electronics**	**Software**	**Logic & UX**
Consumer	Prebuilt mounts; safe geometry	Pre-wired modules	App presets	One-click modes, arming rules
Learner	Parametric parts	Labeled harnesses; rails	Config files; examples	Guided labs, wizards
Researcher/Dev	Custom frames; CoG guidance	Swappable drivers; EMI budget	Ports/adapters; SDK	Tests, CI, telemetry
Professional	Optimized rigs; stiffness specs	Power tiers; fusing; connectors	RT guarantees; schedulers	Toolchains, HIL, certification hooks

1.4 Minimum Viable Platform (MVPf)

A *Minimum Viable Platform* requires: (1) a stable, versioned contract with at least one extension point per relevant layer; (2) two distinct realizations exercising the contracts; (3) tooling to build/test/document extensions; and (4) governance for compatibility and deprecation.

1.5 ASCE Contracts (Selected)

This section provides concise, testable contract specifications for three key ports. Each contract includes functional semantics, timing, error model, and compliance tests. Versions use semantic versioning (e.g., `imu.v2`).

Table 1.3: MVPf checklist (evidence at submission time)

Criterion	Evidence artifact
Stable contract(s)	Interface spec (versioned), timing/error semantics, unit tests
Two realizations	e.g., Drone + Line-follower reusing core; alternate IMU/ESC drivers
Tooling	Templates, CI, documentation, simulator/HIL scripts
Governance	SemVer policy, compatibility matrix, deprecation schedule

1.5.1 IMU Port (`imu.v2`)

Identifier & Version `imu.v2`.

Functional Semantics Provide linear acceleration (a_x, a_y, a_z) in $\mathrm{m/s^2}$ and angular velocity $(\omega\_x, \omega\_y, \omega\_z)$ in rad/s; right-handed body frame; timestamps in microseconds since boot.

Sampling Required supported rates: 100, 200, 400 Hz; default 200 Hz. Report actual rate via `sampleRateHz()`.

Timing Contract End-to-end latency (sensor read $\rightarrow$ consumer) < 2 ms (typ.), jitter < 500 μs (95th percentile) at 200 Hz.

Calibration Bias/scale persisted via Store port; soft-iron/hard-iron optional; expose `setCalibration()` and `getCalibration()`.

Error Model Health states: `OK`, `DEGRADED` (auto-retrying), `FAILED` (requires reinit); provide error codes and backoff policy.

Power/EMI 3.3V logic; I^2C or SPI; max I^2C clock 400 kHz; bus hang recovery via clock stretching up to 9 pulses.

Security/Integrity Optional CRC in SPI; sequence counters for drop detection in DMA pipelines.

Compliance Tests (i) rate conformance; (ii) unit/axis sign tests; (iii) jitter histogram under CPU load; (iv) power brownout recovery; (v) hot-swap detection (if bus supports it).

1.5.2 ESC Port (`esc.v1`)

Identifier & Version `esc.v1`.

Functional Semantics Command normalized throttle $u \in [0, 1]$; monotonic mapping to PWM or DSHOT. Provide `arm()`, `disarm()`, `write(u)`, `stop()`.

Timing Contract Update at 400 Hz (min). Command-to-output latency < 1 ms typ.; report `lastWriteTime()`.

Safety Disarmed default; loss-of-signal watchdog < 50 ms; min/max clamps; ramp-rate limits configurable.

Error Model `OK`, `TIMEOUT`, `BUS_ERROR`, `OVERCURRENT` (if telemetried). Failsafe maps to `stop()`.

Electrical 3.3V PWM or digital protocol (e.g., DSHOT600); shared ground; specify rise/fall constraints for LEDC or timer units.

Compliance Tests (i) monotonicity sweep; (ii) timing scope traces; (iii) watchdog trip; (iv) arming sequence; (v) ramp-rate bound check.

1.5.3 Radio Port (`radio.v1`)

Identifier & Version `radio.v1`.

Functional Semantics Bidirectional datagrams with topics (e.g., `rc/*`, `telemetry/*`). Provide `publish(topic, payload)` and `subscribe(topic, handler)`.

Timing Contract Control path latency < 20 ms one-way (95th percentile). Provide `rssi()` and link `quality()`.

Error Model Retries with exponential backoff; duplicate filtering via sequence numbers. Health: `OK`, `DEGRADED`, `DISCONNECTED`.

Security Optional pairing with nonce; packet authentication (HMAC) optional but recommended for professional tier.

Compliance Tests (i) range vs. latency curves; (ii) packet loss under interference; (iii) reconnect times; (iv) topic throughput fairness.

1.6 Proof by Substitution: Swapping IMUs in Drongaz

This section demonstrates that ASCE's ports & adapters architecture allows replacing one IMU with another without changing the core (controllers, estimator, mixer, or telemetry schema).

1.6.1 Port Interface (stable)

C++ Interface (header):

```
// Port: Imu (imu.v2)
struct ImuSample {
uint64\_t t\_us; // timestamp (us since boot)
float ax, ay, az;   // m/s^2
float gx, gy, gz;   // rad/s
};

enum class Health { OK, DEGRADED, FAILED };

class Imu {
public:
virtual \~Imu() {}
virtual bool begin() = 0;
virtual bool read(ImuSample& out) = 0; // non-blocking; returns false if no
    new sample
virtual float sampleRateHz() const = 0;
virtual Health health() const = 0;
};
```

1.6.2 Two Adapters (variable)

Adapter A: `Mpu6050Imu` implements `Imu` over I^2C.

Adapter B: `Icm20948Imu` implements `Imu` over SPI.

```
class Mpu6050Imu : public Imu { /\* ... *\/ };
class Icm20948Imu : public Imu { /* ... \*\/ };
```

1.6.3 Dependency Injection into the Core

The estimator depends only on the `Imu` port.

```
class Estimator {
Imu& imu;
public:
explicit Estimator(Imu& ref) : imu(ref) {}
void tick() {
ImuSample s; if (imu.read(s)) { /\* predict-update using s \*/ }
}
};
```

1.6.4 Build-Time Swap (no core edits)

```
// main.cpp (composition root)
\#ifdef USE\_ICM20948
static Icm20948Imu imu;
\#else
static Mpu6050Imu imu;
\#endif
static Estimator estimator(imu);
```

1.6.5 Run-Time Swap via Factory (still no core edits)

```
// config.json (deployed via Store port)
// { "imu": { "driver": "icm20948", "rate\_hz": 400 } }

std::unique\_ptr<Imu> makeImu(const Config& cfg) {
if (cfg.imu.driver == "icm20948") return std::make\_unique<Icm20948Imu>(cfg)
    ;
if (cfg.imu.driver == "mpu6050")  return std::make\_unique<Mpu6050Imu>(cfg);
return nullptr; // TODO: handle error
}
```

1.6.6 Invariants Preserved

- Telemetry schema (units, axes, timestamps) unchanged (consumer code unaffected).
- Controllers/mixer unaffected; only `Imu` adapter swapped.
- RTOS task graph unchanged; timing contract verified against `imu.v2` tests.
- No edits to core files (e.g., `FlightCore`, `Controllers`, `Mixer`, `Telemetry`).

1.7 Contract Template (for New Ports)

Use the following template when specifying additional ports (e.g., `clock.v1`, `telemetry.v1`, `store.v1`).

1. **Identifier & Version**: `<name>.v<major>`.
2. **Functional Semantics**: Inputs/outputs, coordinate frames, units, lifecycle.

3. **Timing Contract**: Rates, latency/jitter bounds, concurrency model, buffering.
4. **Error Model**: Health states, error categories, retry/backoff, degradation behavior.
5. **Safety**: Failsafe states, watchdogs, arming/disarming, clamping.
6. **Electrical/Mechanical (if applicable)**: Voltage/current, connectors, mounting patterns.
7. **Security/Integrity**: Auth, sequence numbers, CRCs, tamper evidence.
8. **Compliance Tests**: Automated checks, HIL procedures, acceptance thresholds.
9. **Deprecation Policy**: Compatibility window and migration notes.

2 Foundational Concepts in Robotics, Kinematics, and Interdisciplinary Integration

This chapter isn't finalized, it's me trying to imagine an initial structure to be edited over time, I am not sure if I should provide superficial insights on robotics to put the reader of this thesis on the same page with me as I document my work more in following chapters, or if I should dive deeper into each part of each chapter which will result in a very rich yet a long thesis which may not even be read in the end

This chapter establishes a rigorous theoretical framework for the work presented in this thesis. It delves into advanced kinematic models, surveys the multidisciplinary nature of robotics, and outlines the research methodology grounded in abstraction and modularity. The discussion is supported by key academic references and mathematical formulations, setting the stage for the detailed design and experimental work in later chapters.

2.1 Overview of Robotics

Robotics is an interdisciplinary field that has evolved from simple automation to the development of complex systems with autonomous decision-making capabilities. This evolution is driven by:

- **Sensor and Actuator Advances:** Modern robots leverage cutting-edge sensors and actuators that provide high-resolution environmental data and precise control [**Craig2005**, **Siciliano2009**].
- **Computational Power and Algorithms:** Enhanced processing capabilities and advanced algorithms, including machine learning and adaptive control, allow robots to perform real-time analysis and complex tasks.
- **Modularity and Open-Source Development:** The trend toward modular design enables scalable, reconfigurable platforms that can be tailored for research, education, and even consumer applications. This paradigm shift facilitates collaborative development and rapid innovation.

Academic studies often address challenges such as system integration, sensor fusion, and the development of robust control algorithms. Foundational works by Siciliano *et al.* [**Siciliano2009**] and Craig [**Craig2005**] have significantly influenced modern robotics by providing theoretical and practical insights that continue to shape the field.

2.2 Fundamentals of Kinematics in Robotics

Kinematics is central to understanding and controlling the motion of robotic systems. This section presents detailed models for both forward and inverse kinematics.

2.2.1 Forward Kinematics

Forward kinematics is the process of determining the position and orientation of a robot's end-effector from known joint parameters. The Denavit-Hartenberg (D-H) convention is a standard method used to express the transformation for each joint. For joint i, the transformation matrix is:

$$\mathbf{T}_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The cumulative transformation to the end-effector is given by:

$$\mathbf{T}_{\text{end-effector}} = \prod_{i=1}^{n} \mathbf{T}_i$$

This formulation is vital for simulation and trajectory planning, and it is widely implemented in robotics software such as MATLAB and ROS [**Spong2006**].

2.2.2 Inverse Kinematics

Inverse kinematics seeks the joint parameters that achieve a desired end-effector pose. Unlike forward kinematics, inverse kinematics is generally nonlinear and may yield multiple or no solutions due to the presence of singularities. Two primary approaches include:

- **Analytical Methods:** Closed-form solutions are feasible for simpler kinematic chains but quickly become intractable with increased degrees of freedom.

- **Numerical Methods:** Iterative algorithms such as the Newton-Raphson method or methods based on the Jacobian matrix are employed to converge on a solution.

The Jacobian matrix $\mathbf{J}$ is defined as:

$$\mathbf{J}(\mathbf{q}) = \frac{\partial \mathbf{x}}{\partial \mathbf{q}}$$

where $\mathbf{q}$ represents the joint variables and $\mathbf{x}$ the end-effector position. It relates the joint velocities $\dot{\mathbf{q}}$ to the end-effector velocity $\dot{\mathbf{x}}$:

$$\dot{\mathbf{x}} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}$$

Advanced studies in redundancy resolution and singularity avoidance [**Lynch2017**] further extend the practical application of these models.

2.3 Interdisciplinary Nature of Robotics

Robotic systems integrate diverse fields, each contributing unique methodologies and insights.

2.3.1 Mechanical Engineering

Mechanical engineering underpins the design and fabrication of the physical structure. Key considerations include:

- **Structural and Material Analysis:** Techniques such as finite element analysis (FEA) assess the durability and performance of robot components.
- **Mechanism Design:** Analyzing kinematic chains, joint configurations, and mobility mechanisms is essential for ensuring reliable operation under various conditions.

Classical texts, such as those by Spong *et al.* [**Spong2006**], provide deep insights into these principles.

2.3.2 Electrical and Electronics Engineering

This discipline focuses on:

- **Circuit and Sensor Integration:** Designing custom PCBs, interfacing diverse sensors, and ensuring efficient power management are critical, especially for platforms requiring long battery life.

- **Signal Processing:** Filtering and interpreting sensor data to provide accurate feedback for control algorithms.

These topics are discussed extensively in foundational works like [**Craig2005**].

2.3.3 Computer Science and Control Theory

Software and control algorithms are the brain of robotic systems:

- **Control Strategies:** The implementation of PID controllers, model predictive control (MPC), and adaptive algorithms enables stable and responsive behavior.
- **Software Architectures:** Developing modular, scalable, and real-time software platforms is essential for integrating various subsystems. Open-source frameworks like ROS have revolutionized this integration.

For a comprehensive treatment, refer to [**Siciliano2009**].

2.3.4 Systems Engineering

Systems engineering ensures that all components operate harmoniously:

- **Integration and Testing:** Systematic approaches for validating interoperability and performance are vital.
- **Lifecycle Management:** Structured methodologies help manage the complexity of designing, developing, and deploying advanced robotic systems.

This holistic perspective is crucial for the successful implementation of modular and adaptable platforms.

2.4 Research Approach

The methodology of this thesis is built on the principles of abstraction and modularity. Rather than developing fixed-function systems, the focus is on creating versatile platforms with the following core components:

- **Abstraction:** By decoupling hardware from software, the design allows for independent evolution and easy integration of new technologies.
- **Modularity:** Systems are conceived as collections of interchangeable modules, enabling rapid prototyping, iterative testing, and scalability. This approach is supported by recent research on modular self-reconfigurable robots [**Yim2007**].

- **Theory-Practice Integration:** Theoretical models, such as the kinematic formulations presented earlier, are used to inform design choices and are validated through extensive simulations and experiments.

This integrated research strategy ensures that the platforms developed are not only scientifically sound but also practical and adaptable to various applications. Future work may extend these concepts to explore self-reconfigurable systems and advanced adaptive control methods.

2.5 Conclusion

This chapter has established a superficial robotics foundation for the thesis. By exploring advanced kinematic models, the interdisciplinary framework that supports modern robotics, and a methodologically rigorous research approach, the groundwork has been laid for the subsequent chapters. The next sections will build on these concepts by detailing the design, implementation, and experimental validation of the modular robotic platforms, thereby contributing to the evolution of accessible and adaptable robotics.

3 Battery Power, Characteristics, and Longevity in Robotics

3.1 Introduction

In robotic systems, batteries are not merely an energy sourcethey are a critical component that directly influences performance, autonomy, and reliability. Robotics applications, from mobile platforms to embedded sensor nodes, require power systems that are both energy-dense and robust against degradation. This chapter provides a comprehensive overview of battery power fundamentals, key battery parameters (such as capacity, C-rate, and internal resistance), energy conversion and loss mechanisms, charging methodologies, and advanced battery management techniques. Special emphasis is placed on how these concepts underpin robotic applications. Recent research highlights, including works by Roscher *et al.* [**Roscher2011**], Rauf *et al.* [**Rauf2022**], and Rahman and Alharbi [**Rahman2024**], are discussed throughout to present state-of-the-art insights.

3.2 Fundamentals of Battery Power

Batteries convert stored chemical energy into electrical energy, powering robotic operations through controlled electrochemical reactions. The instantaneous power output P of a battery is given by:

$$P = V \times I, \tag{3.1}$$

where V is the terminal voltage and I is the current. Over time, the total work W provided by the battery is:

$$W = \int_{t_0}^{t_f} V(t) \cdot I(t)\, dt. \tag{3.2}$$

These relationships are fundamental when designing power budgets for robotic missions, where continuous operation under variable load conditions is required.

3.3 Key Battery Characteristics and Terminology

A thorough understanding of battery performance involves multiple key parameters:

3.3.1 Capacity and Energy Storage

Battery capacity, measured in ampere-hours (Ah), represents the total electric charge a battery can store. The energy content E (in watt-hours, Wh) is given by:

$$E = V \times Q, \tag{3.3}$$

where Q denotes the capacity. In robotics, high capacity is crucial for extended missions, yet must be balanced against weight and size constraints.

3.3.2 C-Rate and Operational Dynamics

The C-rate is defined as the charge or discharge current normalized by the battery's nominal capacity. For example, a 1C rate implies that the battery discharges its full capacity in one hour. If a 2 Ah battery is discharged at 2 A, the rate is 1C; at 4 A, it is 2C. Higher C-rates can lead to rapid energy delivery, which is desirable for high-power maneuvers in robotics. However, they also induce thermal stress and accelerate degradation [**Rahman2024**].

3.3.3 Internal Resistance and Voltage Drop

The batterys internal resistance R_{int} is a key factor that affects its performance. Under load, the voltage drop across R_{int} can be expressed as:

$$V_{\text{drop}} = I \times R_{\text{int}}. \tag{3.4}$$

Consequently, the power loss due to internal resistance is:

$$P_{\text{loss}} = I^2 \times R_{\text{int}}. \tag{3.5}$$

These losses generate heat and reduce the effective energy delivered, making thermal management a priority in robotic power systems.

3.4 Energy Conversion and Loss Mechanisms

Understanding energy conversion in batteries is essential for efficient system design. While the chemical-to-electrical conversion is highly effective, several loss mechanisms

reduce the net energy output:

- **Resistive Losses:** As described by Equation (4), losses due to internal resistance diminish available power.
- **Thermal Losses:** Elevated operating temperatures increase resistive losses and can lead to accelerated degradation.
- **Electrochemical Inefficiencies:** Side reactions and incomplete charge transfer reduce the effective capacity.

In robotics, minimizing these losses directly translates to longer operational time and higher reliability.

3.5 Battery Charging Techniques

Efficient charging is critical for maintaining battery health and longevity in robotic systems.

3.5.1 The CCCV Charging Protocol

Most modern batteries, particularly lithium-ion types, are charged using the Constant Current-Constant Voltage (CCCV) method. Initially, a constant current is applied until the battery voltage reaches a set threshold. Thereafter, the charger maintains a constant voltage, and the current gradually decreases. This protocol minimizes overcharging and thermal stress, thereby reducing degradation over multiple cycles.

3.5.2 Advanced Charging Strategies

Beyond CCCV, advanced techniques such as pulse charging and adaptive charging algorithms are under investigation. These strategies aim to optimize charge acceptance and reduce aging by dynamically adjusting the charging profile based on battery temperature, state-of-charge (SOC), and internal resistance feedback [**Rauf2022**].

3.5.3 Battery Management Systems (BMS)

A sophisticated BMS is crucial for robotics, where the battery must be monitored and controlled in real time. Modern BMS architectures integrate sensor arrays to measure voltage, current, temperature, and impedance. By using data-driven methods and machine learning, the BMS can predict state-of-health (SOH) and remaining useful life (RUL), and adjust charging parameters to extend battery life [**Rahman2024**].

3.6 Battery Degradation Mechanisms

Battery degradation is an inevitable phenomenon that impacts the performance and safety of robotic systems. Key degradation factors include:

3.6.1 Cycling Degradation

I may include some graphics and schematics on the mechanism of battery degradation here ?

Frequent charge-discharge cycles lead to capacity fade. Repeated cycling results in the gradual breakdown of electrode materials and the formation of passivation layers, which increases internal resistance and reduces energy storage capacity [**Roscher2011**].

3.6.2 Thermal Effects

High operating temperatures accelerate degradation through enhanced chemical reactions. Excessive heat can lead to electrolyte breakdown and structural damage within the cell. Robotic systems operating in harsh environments must implement active or passive thermal management strategies to mitigate these effects.

3.6.3 Depth of Discharge and Calendar Aging

Deep discharges stress the battery and reduce its overall life span. Additionally, even without cycling, batteries undergo calendar aging due to slow chemical reactions over time. Both factors must be carefully managed in robotics to ensure consistent performance.

3.6.4 Mathematical Modeling of Degradation

Quantitative models for battery degradation are crucial for predicting SOH and RUL. A common expression for SOH is:

$$\text{SOH} = \frac{Q_{\text{actual}}}{Q_{\text{nominal}}} \times 100\%, \tag{3.6}$$

where Q_{actual} is the current capacity and Q_{nominal} is the initial capacity. More sophisticated models incorporate changes in internal resistance and other parameters, enabling predictive maintenance and adaptive energy management in robotics [**Roscher2011**].

3.7 State-of-Health (SOH) and Remaining Useful Life (RUL) Estimation

SOH and RUL are vital metrics for battery performance monitoring:

- **State-of-Health (SOH):** Represents the current capacity relative to the nominal capacity. It is a direct measure of degradation and informs about the battery's ability to deliver power.
- **Remaining Useful Life (RUL):** Estimates the number of cycles or the time remaining before the battery degrades to a level that necessitates replacement.

Advanced techniques for SOH and RUL estimation leverage both physics-based models and data-driven approaches, including Kalman filters and machine learning algorithms. These methodologies have been successfully applied to optimize battery performance in robotics, where real-time estimation is essential [**Rauf2022**, **Rahman2024**].

3.8 Implications for Robotics

In robotics, battery performance directly affects operational efficiency, endurance, and reliability. Key considerations include:

- **Energy Density vs. Weight:** High energy density is critical for extending mission duration, yet must be balanced against weight constraints in mobile robotic platforms.
- **Thermal Management:** Robust cooling strategies are essential to mitigate heat-induced degradation, especially in high-performance or outdoor robotics.
- **Modular Battery Architectures:** For robotics, designing batteries that are easily replaceable or upgradable enables flexibility and extended system lifetimes.
- **Adaptive Power Management:** Incorporating real-time SOH and RUL estimations allows robotic systems to adapt their energy usage and charging cycles dynamically, thereby maximizing operational efficiency.

3.9 Case Studies and Applications in Robotics

Several robotic systems have successfully integrated advanced battery management to enhance performance:

- **Mobile Robotics:** Autonomous rovers and drones require batteries that can handle high C-rates for rapid acceleration and deceleration. The integration of BMS with predictive algorithms ensures that these platforms can operate reliably in dynamic environments.
- **Embedded Systems:** In sensor networks and IoT devices, battery longevity is paramount. Techniques such as adaptive charging and low-power management extend the operational life of these systems.
- **Educational and Research Platforms:** Open-source robotics platforms benefit from modular battery architectures, allowing students and researchers to experiment with different power management strategies and observe the effects of degradation in real time.

These examples underscore the importance of battery optimization in achieving the desired performance and reliability in diverse robotic applications.

3.10 Future Research Directions and Challenges

Despite significant advancements, several challenges remain:

- **Improving Degradation Models:** There is a need for more accurate and robust models that can account for the complex interplay of factors affecting battery degradation in real-world robotics applications.
- **Integration of AI in BMS:** Further integration of machine learning and artificial intelligence could enable more precise real-time monitoring and adaptive control, leading to better SOH and RUL predictions.
- **Sustainable Battery Technologies:** Research into new battery chemistries and materials that offer higher energy density and longer life spans will be crucial for the future of robotics.
- **Miniaturization and Modularity:** Developing modular battery packs that can be easily integrated into various robotic platforms without compromising performance remains an ongoing engineering challenge.

Addressing these challenges will be key to developing the next generation of robotic systems with enhanced autonomy and reliability.

3.11 Conclusion

This chapter has provided an in-depth examination of battery power fundamentals, key characteristics, and longevity considerations within the context of robotics. We have discussed the electrical principles that govern battery operation, explored critical parameters such as capacity, C-rate, and internal resistance, and detailed the mechanisms of energy loss and degradation. Advanced charging techniques, sophisticated battery management systems, and modern degradation models have been reviewed, emphasizing their applications in robotics. By integrating theoretical insights with state-of-the-art research, this chapter lays the foundation for designing robust, efficient, and long-lasting power systems essential for next-generation robotic platforms.

4 Discovery and pairing system:

We've already established that our notion of "generalized platforms" implies cross-platform functionlaity, we need us an "on-platform" system that can identify other pair-able platforms nearby/from the cloud, their types, fetch their characteristics and parameters and give the user the ability to exploit such features. How can we go on about such a system?

We could initially develop our own physical link, protocol and system, or we can use something cheap and robust and already well documented such as ESP-NOW, since most of our robots are ESP32 based or incorporate ESP32 for connectivity, this makes it the perfect choice. ESP-NOW should give us a ready to use physical link and protocol, and all is left for us to develop is the pairing system and identification system, which is mostly software after we make the boards on BULKY and THEGILL self identify the plugged in peripherals, and Microtomba and others being breadboard based and manually configured, we can implement such system on every platform but with different interfaces, where we have a UI we can control behaviours from the robot itself such the case of BULKY, or we can use a controller for such behaviours too such is the case with ILITE, we can make all intercommunicate to create synergy or emergeant behaviour or a hive system, let's call it ASCE OPENswarm system, this would raise certain concerns like safety of operation, how do we detect unsafe scenarios and raise an exception to the user ? how do we make certain features apply to only a certain combination of platforms ? so on so on.

5 Software Backbone:

Embedded systems of this sort use software; ubiquitous information right ? But apparently, "software" is a universe that allows for countless philosophies and approaches.

For our platforms, we want software to be modular, you can pick software and use elsewhere, this notion already exists, it's libraries, packages, modules...etc But how do we make it as versatile as our systems are aiming to get ? One possible way to do this is by dividing software into layers, for now, 3 layers suffice; -The Logic-Software layer (Functional/OOP); takes care of behaviours and logic -The Software-Middlewear layer (RTOS) -The Hardware layer (HAL);

one way to do this is by using a hexagonal architecture

5.1 Introduction

Designing reliable software for cyber-physical systems (e.g. drones, robots, autonomous vehicles) requires more than just writing code that runs. It requires a principled approach to structuring the software such that logic, hardware interaction, and operating system details are separated. This separation fosters modularity, portability, and long-term maintainability. In this chapter we outline a general philosophy for embedded systems architecture using a layered approach inspired by ports-and-adapters (also known as the hexagonal architecture).

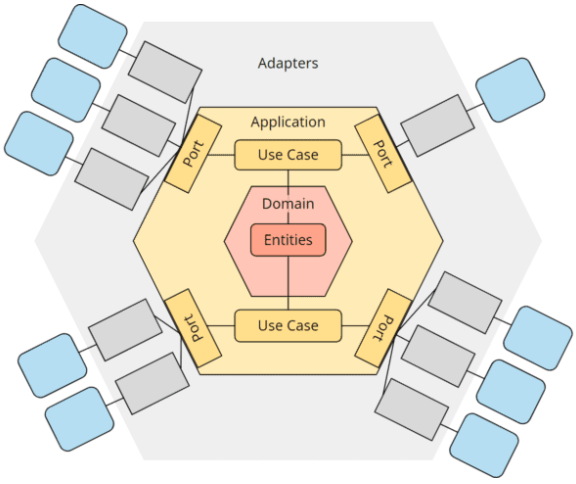

Figure 5.1: Enter Caption

5.2 Motivation

Embedded systems integrate three worlds:

1. **The application logic** the mathematical and algorithmic core, e.g. control laws, estimators, and data fusion.
2. **The hardware interface** the code that speaks to sensors, actuators, communication buses, and storage devices.
3. **The middleware** the operating system or runtime services that schedule tasks, manage concurrency, and orchestrate communication.

Without clear boundaries, these three concerns tend to intertwine. The result is fragile software that is difficult to port to new hardware or to test in isolation. A layered philosophy addresses this by enforcing strict separation of responsibilities.

5.3 Core Logic Layer

At the center lies the *core logic layer*. This layer contains:

- State estimation and sensor fusion algorithms.
- Controllers (e.g. PID, cascaded angle-rate loops).
- Mixers that translate control outputs into actuator commands.
- Safety mechanisms such as failsafe behaviour.
- Telemetry formatting and semantics.

A key design rule is that the core layer must remain **platform independent**. It does not include device drivers, operating system calls, or hardware registers. Instead, it depends only on abstract interfaces that define what the system *requires* (e.g. “read IMU sample”, “write motor outputs”, “send telemetry”).

5.4 Ports and Adapters

Between the core and the physical world sit the *ports and adapters*.

- **Ports** are abstract interfaces, defined in the core, that describe required capabilities such as reading a sensor, commanding an actuator, or storing configuration data.

- **Adapters** are concrete implementations of these ports for a specific platform. For example, one adapter may implement the motor control port using PWM timers on an ARM microcontroller, while another implements the same port using LEDC on an ESP32.

This inversion of dependencies (the core depends only on abstractions, not concrete drivers) allows us to swap hardware, libraries, or protocols without touching the control logic.

5.5 Middleware Layer

The middleware layer encompasses the runtime environment and operating system services:

- Task scheduling and priorities (e.g. FreeRTOS tasks at fixed update rates).
- Communication orchestration (e.g. network discovery, pairing protocols).
- Synchronization primitives for safe access to shared state.

The middleware injects concrete adapters into the core at startup. It is the *composition root*, responsible for wiring together the system: instantiating drivers, providing clock sources, and starting scheduled tasks.

5.6 External Entities

Finally, beyond the software layers are the *external entities*:

- Physical hardware such as sensors, actuators, and microcontrollers.
- Human operators and controllers.
- Ground stations or monitoring clients.

These interact with the system exclusively through the defined adapters and ports. This ensures that the boundary between "inside the software" and "outside in the world" is explicit and testable.

5.7 Benefits of the Layered Philosophy

This separation yields several advantages:

1. **Portability**: Core logic runs unchanged on different microcontrollers or operating systems, as long as suitable adapters exist.

2. **Testability**: The core can be simulated on a desktop computer with mock adapters, enabling unit tests and hardware-in-the-loop simulation.

3. **Maintainability**: Hardware upgrades or protocol changes only require replacing or extending adapters, not rewriting algorithms.

4. **Clarity**: A well-defined architecture provides a conceptual map, reducing coupling and making reasoning about safety and performance easier.

5.8 Conclusion

The general philosophy presented here is applicable to a wide range of embedded and control systems. By clearly distinguishing between the logic, the hardware interface, and the middleware, engineers can create software that is modular, reusable, and robust. This layered approach transforms complex projects from monolithic and brittle implementations into structured systems that can evolve over time without losing reliability.

General conclusion