WIDENER UNIVERSITY

School of Engineering

Chester, PA 19013

DEVELOPMENT OF A HYBRID POWER GENERATION SYSTEM CONTROLLED BY AN INTELLIGENT BATTERY MANAGEMENT UNIT FOR EXTENDED OPERATION OF MOBILE PLATFORMS

By

Cole Helmer

May 2026

A thesis submitted in partial fulfillment of the requirements

for the degree of Master of Science in Engineering

Robotics Engineering

DEVELOPMENT OF A HYBRID POWER GENERATION SYSTEM CONTROLLED BY AN INTELLIGENT BATTERY MANAGEMENT UNIT FOR EXTENDED OPERATION OF MOBILE PLATFORMS

By: Cole Helmer

COMMITTEE:

Daniel Roozbahani 04/30/2026

Dr. Daniel Roozbahani, Thesis Advisor Date

Xiaomu Song 04/29/2026

Dr. Xiaomu Song Date

04/29/2026

Dr. Syed Hamza Date

APPROVED BY:

Xiaomu Song 04/29/2026

Dr. Xiaomu Song, Department Chair Date

LiKang Chin 5/7/2026

Dr. LiKang Chin, Graduate Program Director Date

Pamela K. McCauley May 8, 2026

Dr. Pamela McCauley, Dean Date

ACKNOWLEDGEMENTS

My parents. Thank you whole-heartedly for the nonstop support throughout my thesis research. The late-night calls going back and forth about ideas with my father, the check-ins from mom, "Hey, how was lab today?!", helped keep me focused and confident in my technical abilities that I could accomplish anything I set my mind to.

My partner. Thank you for the never-ending support and encouragement. Listening to me ramble on about batteries for hours on end, simply just so I could work the ideas out myself. You never let me lose focus or be shaken.

My supervisor, Dr. Daniel Roozbahani. Thank you for your unwavering support and outstanding advising over the past two years.

My Skywalker III Team. Thank you for the support to allow me to continue the project legacy as a solo mission. Your contributions earlier in the project paved the way so I could continue along on my own path.

My friends and family. Thank you for the nonstop support throughout my time working on this project. When things in the lab were not the best, I had places and people to go to and help declutter my headspace.

Dr. Likang Chin, thank you for overseeing the Widener Engineering Master's program. Without this program, I would never have had this great opportunity.

Thank you to all for helping me excel and providing the support needed to succeed as I go on with my journey.

ABSTRACT

Battery-operated mobile platforms across many industries, including electric vehicles, robotic systems, marine platforms, and unmanned aerial vehicles, face operational endurance limits driven by the energy-density gap between lithium polymer (LiPo) cells and chemical fuels. This thesis develops a hybrid power generation and management system that addresses this limit by exploiting the energy-density advantage of gasoline over LiPo cells while preserving the millisecond response that battery-fed motor controllers provide. The work originates from the author's role as project manager of the Skywalker III senior design project at Widener University and is scoped to apply broadly across mobile platforms. The system combines per-cell voltage monitoring, automated switching among three operational states (motor, charge, idle), and proactive scheduling. Two engineering insights anchor the design: restructuring the battery topology from series-first to parallel-first eliminated nanosecond switching requirements and enabled commercially available solid-state relays, and a four-SSR-per-battery configuration with full ground-side isolation resolved a shared-ground conflict preventing simultaneous multi-battery charging. A nine-battery bench prototype validated the architecture across fifteen tests. A Python simulation extended the analysis to the full 48-battery configuration, where a proactive Rolling Replenishment scheduling algorithm developed in this thesis recovered 118.6 percent more energy than the best reactive strategy while executing 58 percent fewer switching events.

TABLE OF CONTENTS

List of Tables

List of Figures

List of Appendices

List of Symbols, Abbreviations, and Nomenclature

Units and Symbols

A	Ampere
kΩ	Kilohm
mAh	Milliampere-hour
V	Volt
W	Watt
Wh	Watt-hour
Ω	Ohm

Abbreviations

AC	Alternating Current
ADC	Analog-to-Digital Convertor
ASTM	American Society for Testing and Materials
AUV	Autonomous Underwater Vehicle
BLDC	Brushless Direct Current Motor
BMS	Battery Management System
CAD	Computer-Aided Design
CC	Constant Current
CCW	Counterclockwise
COM	Communication Port
CTP	Cell-to-Pack
CV	Constant Voltage

CW	Clockwise
DC	Direct Current
EIS	Electrochemical Impedance Spectroscopy
EKF	Extended Kalman Filter
ESC	Electronic Speed Controller
EV	Electric Vehicle
FAA	Federal Aviation Administration
FC	Flight Controller
FCC	Federal Communications Commission
GND	Ground
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
GUI	Graphical User Interface
I^2C	Inter-Integrated Circuit
ICAO	International Civil Aviation Organization
IDE	Integrated Development Environment
ISO	International Organization for
Standardization	
LCO	Lithium Cobalt Oxide
LFP	Lithium Iron Phosphate
LiPo	Lithium Polymer
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor

NCA	Nickel Cobalt Aluminum Oxide
NHTSA	National Highway Traffic Safety Administration
NIST	National Institute of Standards and
Technology	
NMC	Nickel Manganese Cobalt Oxide
OCV	Open-Circuit Voltage
PDB	Power Distribution Board
PWM	Pulse Width Modulation
RPM	Revolutions per Minute
RTH	Return to Home
SiC	Silicon Carbide
SoC	State of Charge
SSR	Solid-State Relay
UAV	Unmanned Aerial Vehicle
UL	Underwriters Laboratories
UPS	Uninterruptable Power Supply

Chapter 1: Introduction

1.1 Background and Motivation

The transition toward electrically powered systems continues to accelerate across industries that depend on mobile platforms. Unmanned aerial vehicles (UAVs), electric automobiles, portable industrial equipment, and robotic ground and aquatic vehicles are increasingly reliant on rechargeable battery systems as their primary energy source, driven by reduced exhaust emissions, lower acoustic signatures, decreased mechanical complexity, and the growing availability of lithium polymer (LiPo) and lithium-ion cell technologies (Saravanakumar et al., 2023). Despite these advantages, LiPo cells are constrained by a substantial energy density gap relative to chemical fuels, and this gap remains the central engineering challenge motivating this thesis.

Gasoline contains approximately 12,000 Wh/kg of chemical energy, whereas LiPo cells offer 150 to 250 Wh/kg depending on chemistry and architecture (Miao et al., 2019). Internal combustion engines convert this energy at only 25 to 35 percent thermal efficiency (Zhang et al., 2022), yet even at 25 percent, one kilogram of gasoline yields approximately 3,000 usable watt-hours. Brushless DC motors paired with electronic speed controllers (ESCs) achieve roughly 90 percent efficiency, yielding approximately 225 usable watt-hours from one kilogram of LiPo cells. After all conversion losses, gasoline retains a usable energy advantage of more than 12 to 1. Figure 1, seen below presents this comparison graphically.

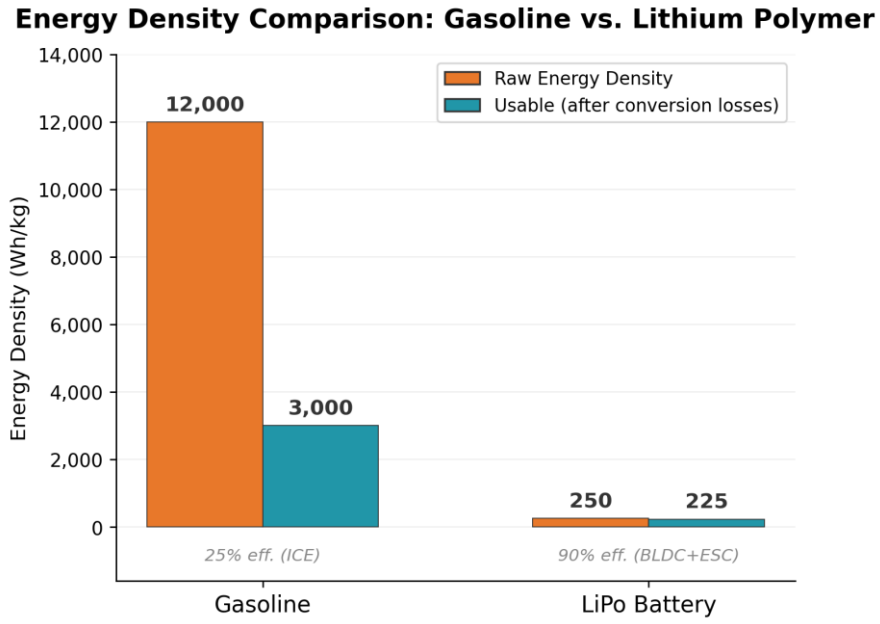

Figure 1. Energy Density of Gasoline vs. Lithium Polymer, Raw and Post-Conversion.

A natural question follows from this disparity: if gasoline carries so much more energy per unit mass, why not use the combustion engine as the direct motive power source for a multirotor UAV? The answer lies in the fundamental mismatch between combustion engine response dynamics and multirotor flight control requirements. Unlike fixed-wing aircraft, which generate lift through aerodynamic surfaces and can tolerate relatively steady-state engine output, multirotor platforms produce all lift through individual motor-propeller assemblies. Each motor is governed by an ESC that modulates rotor speed on a millisecond timescale. The flight controller computes attitude corrections at update rates on the order of hundreds of hertz and commands corresponding ESC adjustments in real time. A deviation of even a few hundred RPM on a single motor, sustained for a fraction of a second, is sufficient to produce asymmetric thrust and initiate an uncommanded roll, pitch, or yaw event.

A combustion engine responds to throttle inputs through a sequential chain of mechanical and thermodynamic processes, fuel-air mixture regulation, combustion propagation, and crankshaft inertia, introducing response latencies on the order of hundreds of milliseconds to seconds (Heywood, 2018). A hexacopter has no ailerons, elevators, or rudder surfaces to compensate for this lag; the motors themselves are the sole means of correcting orientation. Battery-fed ESCs operating at kilohertz-range switching frequencies are therefore a structural requirement for multi-rotor flight stability, not merely a design preference. Figure 2, illustrates the contrast between millisecond-scale ESC control and the comparatively slow mechanical throttle response of a combustion engine.

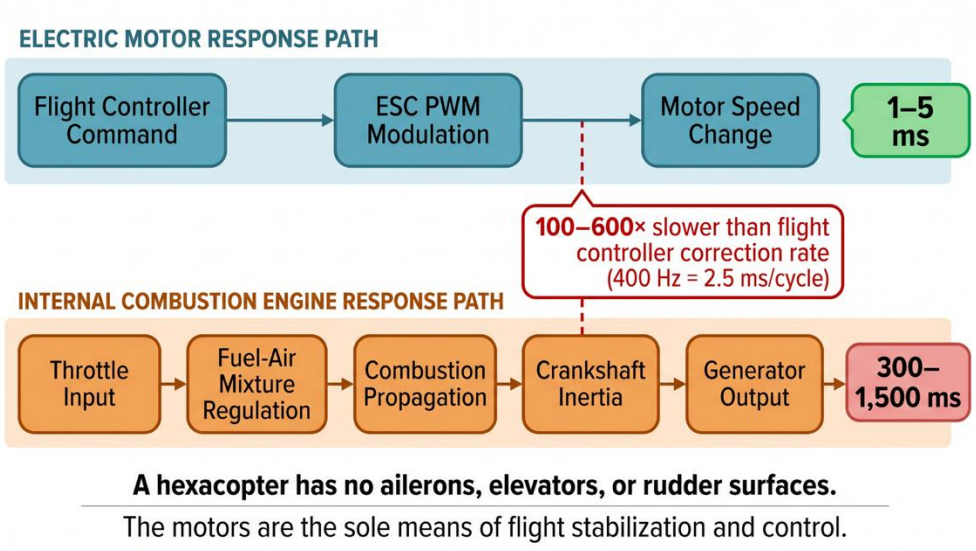

Figure 2. Response-Time Comparison Between Battery-Fed ESC Control and Combustion Engine Throttle Response.

This reaction-time mismatch is not unique to UAVs. Any electrically driven mobile platform requiring rapid torque response, including electric vehicles executing traction control adjustments, marine propulsion systems responding to wave loads, and

robotic ground vehicles navigating dynamic terrain, faces the same fundamental incompatibility between combustion engine response dynamics and the instantaneous power delivery that electric motors provide from battery sources. The series hybrid architecture, in which the combustion engine operates exclusively as a generator charging batteries that in turn power electric motors, resolves this mismatch by decoupling the power generation and power delivery functions. This argument is expanded with quantitative response-time data in Section 7.1.3.

An additional constraint eliminates any architecture using batteries as backup to a direct-drive engine: LiPo cells cannot be charged while simultaneously under discharge load. The electrochemical processes governing lithium-ion intercalation during charging and discharging are opposing reactions that cannot occur concurrently within the same cell. A cell is either sourcing current to the motor bus, accepting current from a charger, or idle. This constraint is not a design choice but a physical limitation of cell chemistry that fundamentally dictates the architecture of any hybrid system employing LiPo cells.

These two constraints together define the viable design space for the hybrid architecture developed in this thesis. The approach adopted is to carry a gasoline-powered generator onboard the aircraft not as a direct motive power source, but as an energy replenishment system. The generator converts gasoline's energy density advantage into electrical energy delivered to battery packs that have been temporarily taken offline from the motor bus, while a separate set of packs continues to power the propulsion system through the ESCs. Figure 3, illustrates the conceptual energy flow of this architecture.

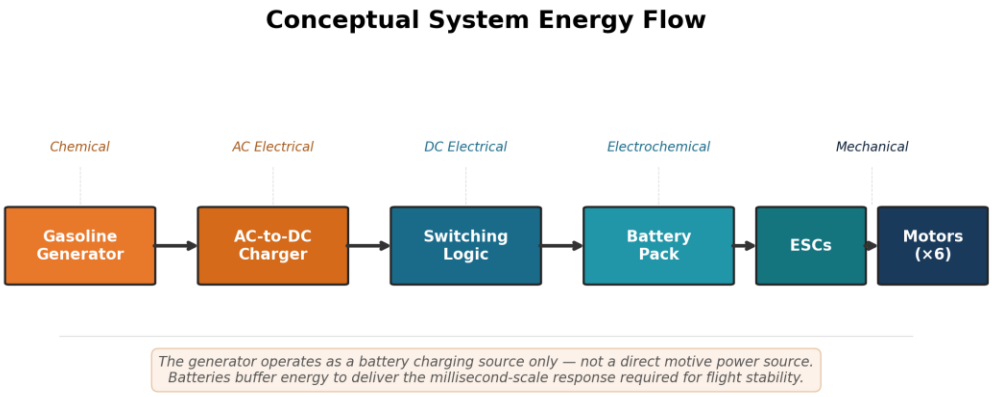

Figure 3. Conceptual Energy Flow of the Architecture.

Current battery-only multirotor UAVs achieve flight endurance in the range of 20 to 60 minutes, depending on airframe size, payload, and throttle profile (Saravanakumar et al., 2023). The conventional approach of adding more cells encounters diminishing returns, as each additional cell adds mass that increases the power required to sustain flight and erodes the endurance gain (Boukoberine et al., 2019). The hybrid architecture circumvents this limitation by replenishing energy from onboard fuel. The series-hybrid concept is not new, but the application of intelligent cell-level management to this architecture remains an active area of research with limited fielded implementations (Jiao et al., 2023). This thesis is motivated by the position that the challenge of extending battery-powered endurance is not best addressed by constructing larger battery packs, but by building smarter systems that monitor, manage, and supplement them at the individual cell level.

1.2 Evolution of Research Focus

The research presented in this thesis originated from the Skywalker III hexacopter, an undergraduate senior design project completed at Widener University as part of the Bachelor of Science in Robotics Engineering program. That project sought to integrate an onboard gasoline generator with a multi-cell LiPo battery system to extend UAV flight duration; the full scope and technical details are presented in Chapter 3. The Skywalker III platform required concurrent development of a structural airframe, a flight controller and telemetry stack, a propulsion system rated for high-current continuous operation, and a power management subsystem responsible for distributing current to six motors drawing a combined peak of approximately 480 A. Of these subsystems, the power management architecture proved most technically demanding, requiring coordination across power electronics, embedded firmware, electrochemistry, and control logic. Because it was addressed last in the project timeline, the physical development of a fully functional charging and switching system remained beyond the undergraduate scope.

This experience revealed that the challenges inherent to the power management subsystem were not specific to the Skywalker III hexacopter. Supplementing battery energy with generator-based charging, monitoring individual cell health in real time, and automating power source transitions are problems shared by any battery-operated mobile platform requiring sustained operation. This recognition motivated a deliberate shift from a platform-specific UAV application to a generalized investigation of hybrid power generation and intelligent battery management for mobile systems. The work presented in this thesis describes the design, development, and validation of a standalone intelligent

power management system, developed independently of any specific platform, with applicability to the Skywalker III architecture, electric vehicles, and other platforms evaluated in later chapters.

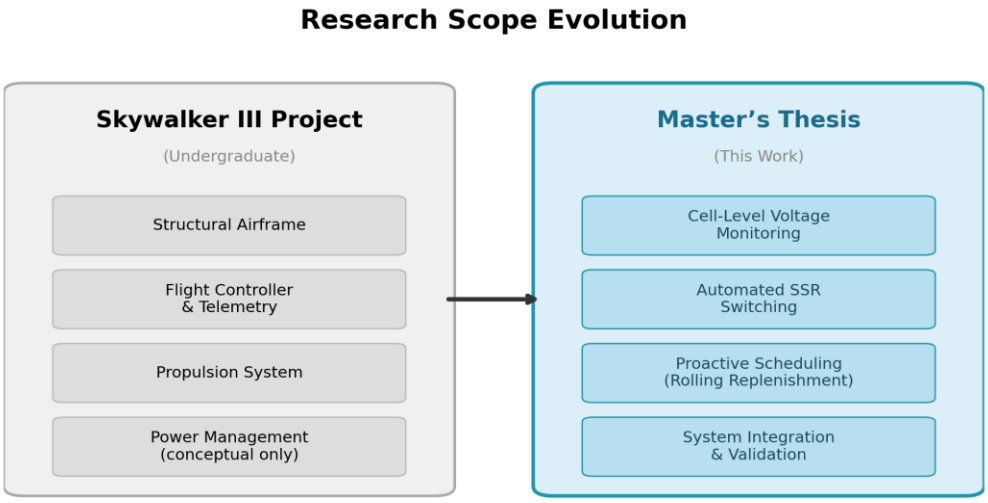

Figure 4. Research Scope Evolution from the Skywalker III Undergraduate Project to the Present Thesis.

1.3 Problem Statement

Battery-operated mobile platforms across a broad range of industries are fundamentally constrained by the energy density limitations of current LiPo technology and increasing onboard cell count encounters diminishing returns in mass, wiring complexity, thermal load, and cost (Arora et al., 2016). Beyond this physical constraint, a significant gap exists in how current battery management systems handle multi-cell architectures. Modern BMS hardware can read individual cell voltages within modules for passive cell balancing and safety enforcement (Wang et al., 2020), yet no production system uses this cell-level data to make real-time power routing decisions. No commercially deployed BMS has the capability to selectively disconnect a module from

the load bus, route it to a charging source, and replace it with a charged module while the system continues to operate without interruption. This gap between available monitoring granularity and actionable control logic is a central motivation for the system developed in this thesis.

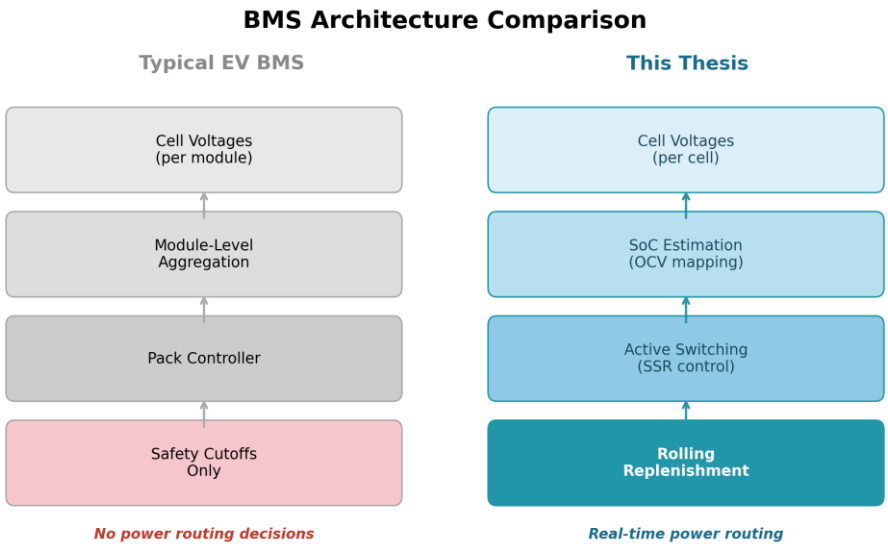

Figure 5. Architectural Comparison Between a Typical Production EV BMS and the Active Power-Routing Architecture Developed in This Thesis.

This limitation has measurable consequences. In packs containing thousands of cells, individual cells age at different rates depending on their position within the pack, local thermal conditions, and manufacturing variance. A single degraded cell limits the usable capacity of its entire module, because the BMS must respect the weakest cell's voltage cutoff even while the remaining cells retain substantial usable energy (Wang et al., 2020). Without cell-level monitoring that feeds into active decision-making logic, early indicators of cell degradation or incipient thermal runaway may go undetected until they escalate to module-level or pack-level failures (Miao et al., 2019).

Through the findings of the preliminary undergraduate work and a review of the current state of battery management in both UAV and EV applications, three unmet requirements have been identified: First, the system must be capable of monitoring individual cell voltages in real time to provide accurate state-of-charge estimation and early detection of cell-level anomalies. Second, the system must be capable of automatically switching individual battery packs between power delivery and charging paths without interrupting the supply of current to the motor load. Third, these two capabilities must be integrated into a single cohesive architecture with intelligent scheduling logic that determines when and which packs to transition between states. No existing commercially available system combines all three capabilities into one unified platform (Jiao et al., 2023).

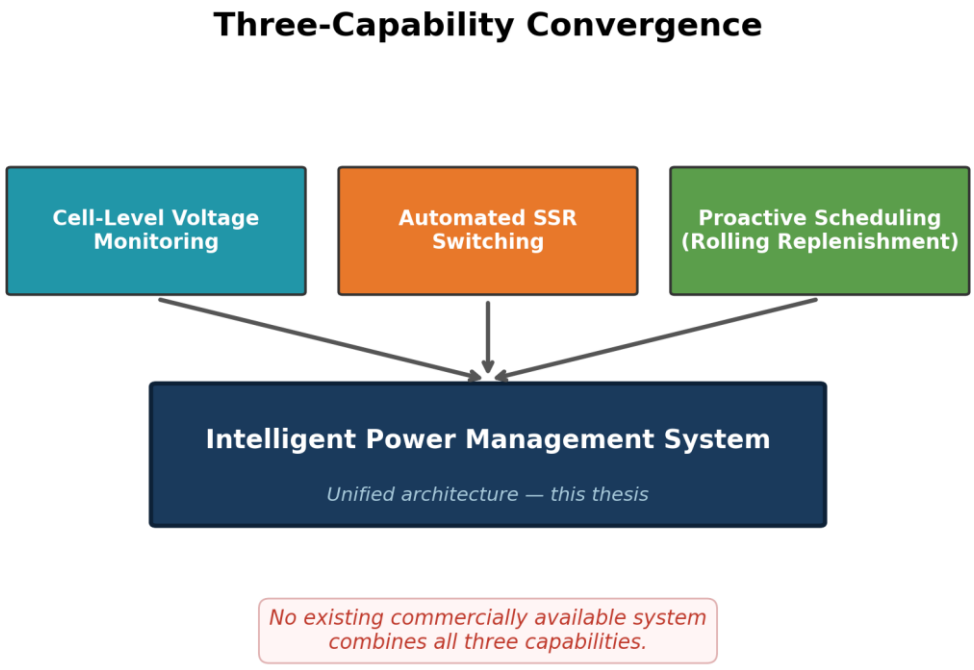

Figure 6. Convergence of Cell-Level Monitoring, Automated Switching, and Proactive Scheduling into a Unified Intelligent Power Management Architecture.

1.4 Research Objectives

The first objective is to design, develop, and validate a voltage monitoring system capable of reading individual LiPo cell voltages within a multi-cell series-parallel battery architecture. The system employs Arduino Mega microcontrollers interfaced with resistive voltage divider networks connected to the balance leads of each cell, enabling measurement of individual cell potentials without requiring disconnection from the pack. These cell-level voltage readings are used to estimate the state of charge (SoC) of each cell via a lookup table derived from the characteristic LiPo discharge curve. Monitoring at the individual cell level enables the detection of voltage anomalies, capacity imbalances, and early indicators of cell degradation that would not be visible through aggregate pack-level readings.

The second objective is to develop a power switching architecture capable of transitioning individual battery packs between three operational states: MOTOR (supplying current to the propulsion bus), CHARGE (accepting current from the charging system), and IDLE (electrically isolated from both buses). The initial approach employed MOSFET-based switching predicated on a series-wired battery topology requiring nanosecond-scale switching speed. After multiple circuit iterations resulted in component failures, a fundamental architectural reassessment led to the central design contribution of this thesis: the battery topology was restructured from a series-first to a parallel-first configuration, where individual batteries connect in parallel at each voltage junction and the junctions are wired in series to achieve the target system voltage. In this revised topology, removing a single battery from a junction does not interrupt the current

path, eliminating the nanosecond switching requirement and enabling the use of commercially available solid-state relays. The complete design evolution is detailed in Section 4.4.

The third objective is to integrate voltage monitoring and power switching subsystems into a single cohesive system capable of autonomous operation. When a pack's SoC falls below a defined threshold, the system autonomously transitions that pack from the motor bus to the charge bus and rotates a charged pack into service. The integrated system was validated through bench testing at three-, six-, and nine-battery configurations to verify end-to-end functionality and repeatability.

The fourth objective is to evaluate the broader applicability of the developed system. A Python-based simulation was developed to model the behavior of the full 48-battery Skywalker III configuration under five distinct scheduling strategies, comparing reactive approaches against a proactive rolling replenishment algorithm developed as part of this thesis. An analysis of how the architecture's principles apply to electric vehicle battery management systems and other battery-operated mobile platforms is presented in Chapter 7.

1.5 Thesis Organization

Chapter 2 provides a literature review covering energy storage technologies, hybrid power architectures, EV battery management, and power switching electronics. Chapter 3 presents the Skywalker III project foundation. Chapter 4 details the full system design and implementation, including the voltage monitoring circuit, the switching

architecture evolution, ground-side isolation discovery, and integrated system architecture. Chapter 5 covers testing methodology and hardware validation. Chapter 6 presents the simulation framework and scheduling strategy comparison. Chapter 7 examines applicability to electric vehicles and other platforms. Chapter 8 presents conclusions and recommendations for future work.

Chapter 2: Literature Review

This chapter establishes the technical foundation for the intelligent power management system presented in this thesis. Section 2.1 reviews energy storage technologies, including LiPo cell fundamentals, state-of-charge estimation methods, and charging constraints. Section 2.2 examines hybrid power architectures that combine combustion-based generation with battery storage on mobile platforms. Section 2.3 explores the applicability of these principles to the electric vehicle industry. Section 2.4 covers the power switching technologies employed in this design.

2.1 Energy Storage Technologies

2.1.1 Gasoline vs. Battery Energy Density

The gravimetric energy density comparison presented in Section 1.1, gasoline at approximately 12,000 Wh/kg versus LiPo cells at 150 to 250 Wh/kg, yielding a usable advantage of more than 12 to 1 after conversion efficiencies, is reinforced by the volumetric comparison. Gasoline stores approximately 9,600 Wh/L (Heywood, 2018), while commercially available LiPo cells achieve volumetric energy densities in the range of 300 to 700 Wh/L depending on cell format and chemistry (Miao et al., 2019). By volume, gasoline stores roughly 14 to 32 times more energy than an equivalent volume of LiPo cells. For mobile platforms where both payload mass and onboard volume are tightly constrained, this dual advantage in gravimetric and volumetric energy density establishes a compelling case for carrying fuel as a supplementary energy source.

The energy density gap is not expected to close substantially in the near term. Current lithium-ion research has pushed cell-level gravimetric densities toward 300 Wh/kg in laboratory prototypes, with solid-state architectures targeting 400 to 500 Wh/kg (Miao et al., 2019). Even at these projected figures, gasoline would retain a usable advantage of approximately 6 to 1. This persistent gap is the fundamental justification for hybrid power architectures: the combustion engine is not employed because it is efficient, but because its fuel's energy density is large enough to produce a meaningful net gain in operational endurance even after substantial conversion losses.

2.1.2 Lithium Polymer (LiPo) Battery Fundamentals

Lithium polymer batteries are a subclass of lithium-ion batteries distinguished by their use of a polymer electrolyte rather than a liquid organic solvent. A typical LiPo cell consists of a cathode (typically employing a lithium cobalt oxide [LCO] or nickel manganese cobalt oxide [NMC] compound), a graphite anode, and a solid or gel polymer separator that serves as both the electrolyte medium and the physical barrier between electrodes (Lu et al., 2013). This construction enables flexible pouch-type form factors, supports high discharge rates relative to cell mass, and yields a favorable energy-to-weight ratio compared to other rechargeable electrochemical storage technologies. These properties have made LiPo cells the dominant energy storage technology for UAVs, portable electronics, and a growing number of electric vehicle applications (Saravanakumar et al., 2023).

The discharge behavior of a LiPo cell follows a characteristic nonlinear voltage-versus-capacity profile, commonly described as an S-shaped curve, divisible into three distinct regions. In Region 1, spanning approximately the initial 10 to 15 percent of discharge, cell voltage drops rapidly from the fully charged value of 4.2 V to approximately 3.9 V as surface-layer lithium ions are depleted and initial polarization losses establish themselves. Region 2 constitutes the majority of usable capacity, spanning roughly 60 to 70 percent of total discharge, during which voltage declines gradually from approximately 3.9 V to 3.6 V. In Region 3, covering the final 15 to 20 percent, voltage falls steeply toward the manufacturer-specified cutoff of 3.0 V per cell. Operating below these cutoff risks irreversible damage to the electrode structure (Barré et al., 2013). The shape of this discharge curve is central to the voltage-based SoC estimation method employed in this thesis, as discussed in Section 2.1.3.

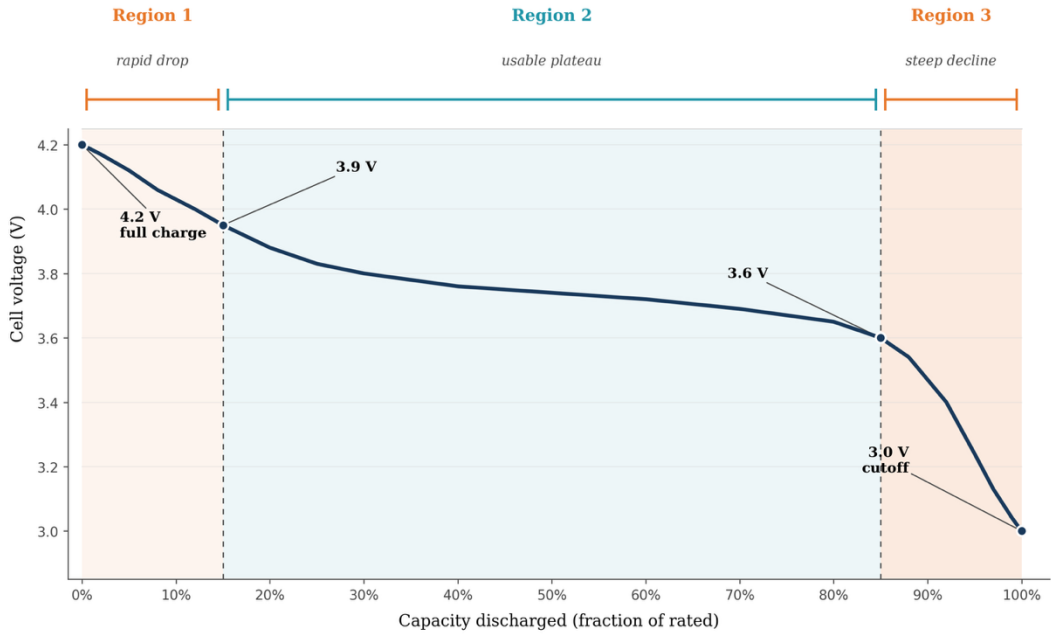

Figure 7. LiPo Cell Discharge Curve Showing the Three Characteristic Voltage Regions.

LiPo cells are rated by two primary performance parameters: capacity in milliamp-hours (mAh) and C-rating, a dimensionless multiplier defining the maximum continuous discharge current as a multiple of rated capacity. A cell rated at 6,500 mAh with an 80C discharge rating has a theoretical maximum continuous discharge current of 520 A (6.5 Ah × 80 = 520 A), though sustained operation near this limit generates significant resistive heating and accelerates electrode degradation (Barré et al., 2013). Individual cells are combined into multi-cell configurations to meet application requirements. Connecting cells in series increases voltage while maintaining single-cell capacity: a 4S pack yields 14.8 V nominal (3.7 V × 4). Connecting packs in parallel increases capacity and current capability while maintaining voltage. This series-parallel principle is fundamental to the architecture developed in this thesis, where three 4S cells

in series form 12S packs at 44.4 V nominal, connected in parallel at each voltage junction. The specific configuration is detailed in Chapter 4.

2.1.3 Battery Management and State-of-Charge Estimation

State of charge (SoC) represents the percentage of usable energy remaining in an electrochemical cell relative to its fully charged capacity. Accurate SoC estimation is essential to the power management system developed in this thesis because every autonomous switching decision depends on this metric. An overestimation of remaining capacity risks depleting a pack below its safe operating voltage under load, while an underestimation triggers unnecessary switching events that reduce system efficiency.

Several methods for estimating SoC have been reported in the literature, each presenting distinct tradeoffs. Coulomb counting integrates the current flowing into and out of the cell over time, providing high temporal resolution but requiring a precision current sensor on each monitored pack and suffering from cumulative drift error over extended periods unless periodically recalibrated (Wang et al., 2020). Model-based approaches such as the extended Kalman filter (EKF) represent the cell as an equivalent circuit and recursively estimate internal states from measured voltage and current, achieving high accuracy under dynamic loads but imposing computational overhead that may exceed the capability of low-cost embedded microcontrollers (Wang et al., 2020). Electrochemical impedance spectroscopy (EIS) extracts SoC from frequency-dependent impedance characteristics, offering high precision but requiring specialized excitation hardware impractical for most mobile applications (Bilansky et al., 2022).

The method employed in this thesis is open-circuit voltage (OCV) mapping, in which the measured terminal voltage of each cell is compared against a pre-characterized lookup table derived from the LiPo discharge curve described in Section 2.1.2. Because the OCV-SoC relationship is monotonic across the usable voltage range (4.2 V to 3.0 V), a single voltage measurement can be mapped to an approximate SoC percentage without requiring current sensing hardware or computationally intensive algorithms. This approach was selected as a deliberate engineering tradeoff: it requires no additional hardware beyond the voltage divider networks already present in the monitoring circuit, it imposes negligible computational load on the Arduino Mega microcontrollers, and it provides sufficient estimation accuracy to validate the system-level switching and scheduling architecture that is the primary contribution of this work.

The principal limitation of OCV-based estimation is voltage sag under load. When a cell is supplying current, its terminal voltage drops below the true OCV by an amount proportional to the discharge current multiplied by the cell's internal resistance. The system mitigates this effect by applying a settling delay before SoC calculations during state transitions and by operating primarily in the plateau region of the discharge curve, where voltage changes slowly with capacity. More sophisticated estimation methods are identified as future work in Section 8.3.

The majority of commercial BMS implementations deploy dedicated monitoring ICs (e.g., the Texas Instruments BQ76952 and the STMicroelectronics L9963E) that read individual cell voltages within each module via daisy-chained communication buses (Texas Instruments, 2021). This cell-level data is used for passive or active cell

balancing and for safety-critical voltage limit enforcement. However, in all current production implementations, this data is not used to make real-time power routing decisions at the module or cell level. The monitoring granularity exists in the hardware, but the control architecture to act on it does not.

2.1.4 Charging Constraints and CC/CV Charging Protocol

LiPo cell charging is governed by a two-phase constant-current/constant-voltage (CC/CV) protocol. During the CC phase, the charger supplies a fixed current (typically between 0.5C and 1C of rated capacity) while cell voltage rises as lithium ions intercalate into the graphite anode. Once the cell reaches 4.2 V, the charger transitions to the CV phase, holding terminal voltage constant while the charging current tapers exponentially as remaining intercalation sites fill. Charging is complete when current falls below a manufacturer-specified threshold, typically 0.05C to 0.1C. The CC phase delivers approximately 60 to 80 percent of total charge; the CV phase delivers the remainder over a proportionally longer period. These time dynamics are essential to the scheduling analysis in Chapter 6, because charge duration directly constrains how many packs can be rotated through the charge path during a given operational period.

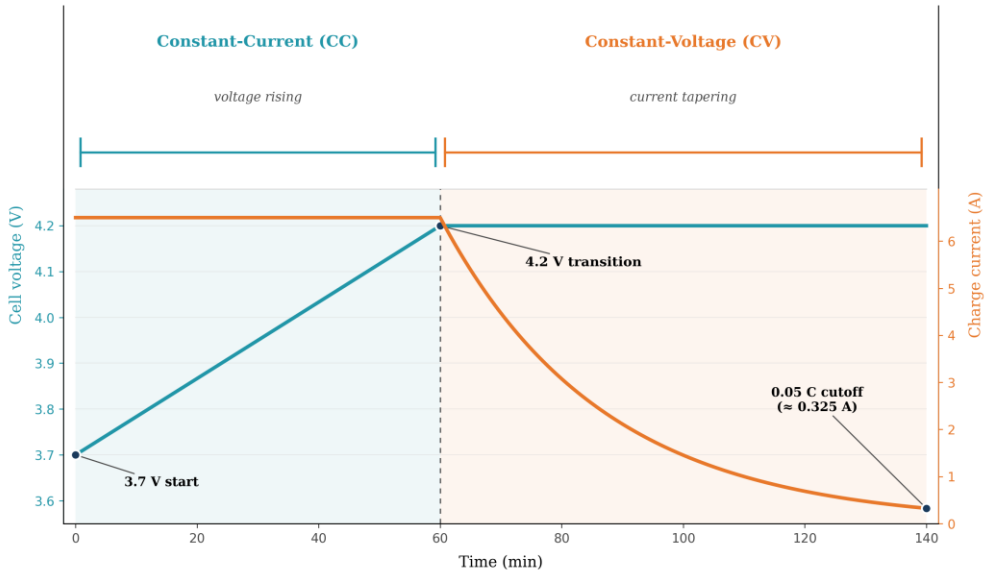

Figure 8. Two-Phase Constant-Current / Constant-Voltage (CC/CV) Charging Profile for a Single LiPo Cell.

A critical constraint of lithium-ion electrochemistry is that a cell cannot charge and discharge simultaneously. The intercalation and de-intercalation of lithium ions at the anode are opposing electrochemical reactions. Attempting to charge a cell while it supplies current to a load creates indeterminate current flow and unpredictable thermal stress. This constraint eliminates any architecture in which the generator charges batteries that are simultaneously powering the motors. The system must physically isolate each pack from the motor bus before connecting it to the charge bus, requiring at least two groups of packs: one delivering propulsive power and one or more receiving charge. Automated switching between these states is the engineering challenge addressed by the power switching architecture described in Chapter 4.

2.2 Hybrid Power Systems for Mobile Platforms

2.2.1 Existing Hybrid Architectures

Hybrid power architectures that combine combustion engines with battery systems on mobile platforms fall into two principal categories: series and parallel. In a parallel hybrid configuration, the combustion engine and the electric motor are both mechanically coupled to the drivetrain and can drive the load simultaneously. The engine provides mechanical power directly to the output shaft while the electric motor supplements during high-demand periods or operates independently during low-demand conditions. This architecture is well established in the passenger vehicle industry, where engine output can be coupled directly to the wheels through a transmission.

In a series hybrid architecture, the combustion engine does not drive the load directly. Instead, the engine powers a generator that produces electrical energy, which charges a battery pack that in turn powers the electric motors. The mechanical load is entirely decoupled from the combustion engine and receives power exclusively through the ESCs. This complete electrical isolation is the defining advantage of the series topology for multi-rotor UAV applications, where millisecond-scale independent control of each motor is required. The system developed in this thesis follows the series hybrid architecture exclusively.

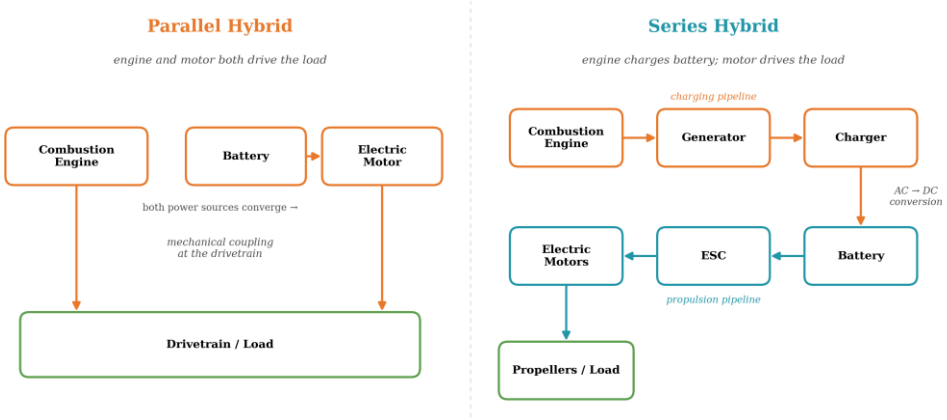

Figure 9. Parallel and Series Hybrid Power Architecture Topologies.

Commercial hybrid multirotor platforms have demonstrated that the series approach can extend flight endurance from the 20- to 60-minute range typical of battery-only systems to operational times measured in hours (Boukoberine et al., 2019). The contribution of this research is not the combustion-to-battery hybrid concept, which has been demonstrated commercially, but the intelligent autonomy layer: cell-level voltage monitoring, automated SoC-based switching with conflict prevention logic, and a proactive scheduling algorithm that anticipates when packs should be rotated rather than reacting after depletion.

2.2.2 Generator-Based Charging Systems

In the series hybrid architecture, the generator bridges chemical and electrochemical energy domains. A gasoline engine spins a rotor within a stator assembly to produce alternating current, which is rectified and regulated for CC/CV charging as described in Section 2.1.4. The power available for charging is limited by the generator's rated continuous output and the efficiency of the conversion chain. For the

Skywalker III flight configuration, the intended source is a WEN GN5602X portable gasoline generator rated at 4,500 W continuous output (5,600 W peak). The generator's AC output feeds a dedicated AC-to-DC charger. The charge current delivered to each battery on the charge bus is determined by:

$$I_{charge} = \frac{P_{charge}}{V \times N_{charging}}$$

where I_{charge} is the charging current per battery (A), $P_{charger}$ is the total available charge power (W), V is the charge bus voltage (V), and $N_{charging}$ is the number of batteries simultaneously on the charge bus. This relationship illustrates a fundamental tradeoff: routing more batteries to charge simultaneously increases total energy throughput but reduces the per-pack charge rate. The scheduling algorithm must balance this tradeoff to maximize energy recovery during a given operational period.

For bench testing, the generator-based charging system was substituted with a Keysight E36233A dual-channel programmable DC power supply providing regulated DC directly to the batteries at 16.8 V with a deliberately limited charge current of approximately 2 A per battery. An EV Peak CQ3 multi-channel LiPo charger was also evaluated during early development but proved incompatible with the custom switching architecture due to its built-in safety interlocks, as documented in Section 4.4.4. The transition to the Keysight supply provided sufficient power to validate the monitoring circuit, SSR switching logic, and scheduling algorithm at the nine-battery prototype scale without charger-imposed constraints. Detailed hardware configurations for both the bench and simulated flight systems are presented in Chapters 4 and 5.

Table 1. Bench vs. Flight Configuration Comparison.

Parameter	Bench (Prototype)	Flight (Full System)
Power Source	Keysight E36233A	WEN GN5602X Generator
Output Type	Programmable DC	AC (Rectified to DC)
Bus Voltage	44.4 V (nominal)	44.4 V (nominal)
Charge Power	400 W	4,000 W
Effective Charge Power	392 W (98% eff.)	3,400 W (85% eff.)
Max Charge Current (16.8 V)	23.8 A	~202 A
Battery Count	9	48
SSR Count	36	192
Batteries per Junction	3	16
Max Charging per Junction	2	15

2.3 Application for Electric Vehicles

2.3.1 EV Battery Architecture and Current Limitations

The electric vehicle industry represents the domain in which the BMS principles developed in this thesis become commercially relevant at the largest scale. The engineering challenges are fundamentally analogous to those addressed in the UAV context: monitoring large numbers of electrochemical cells, estimating SoC under dynamic loads, managing cell-level degradation, and ensuring safe operation across the full charge-discharge envelope. The difference is one of magnitude. A Tesla Model S battery pack contains approximately 7,104 individual 18650-format cells arranged in 16

modules; the Model 3 utilizes approximately 4,416 larger-format 2170 cells (Lambert, 2017). General Motors' Ultium platform employs large-format pouch cells in a modular architecture designed for cross-platform scalability (Arora et al., 2016), while BYD's Blade Battery uses elongated LFP prismatic cells in a cell-to-pack configuration (Arora et al., 2016).

Table 2. Electric Vehicle Battery Architecture Comparison.

Manufacturer	Cell Format	Chemistry	Cell Count	Architecture	Monitoring
Tesla Model S	18650 Cyl.	NCA	~7,104	16 modules, series-parallel	Module-level BMS
Tesla Model 3	2170 Cyl.	NCA/NMC	~4,416	4 modules, series-parallel	Module-level BMS
GM Ultium	Large Pouch	NMC	Varies	Modular cross-platform	Module-level BMS
BYD Blade	LFP Prismatic	LFP	Varies	Cell-to-Pack (CTP)	Pack-level BMS

As established in Section 2.1.3, production BMS hardware reads individual cell voltages but uses this data exclusively for balancing and safety, not for power routing. Individual cells within a module age at different rates depending on their physical position in the pack, local thermal gradients, and manufacturing variance in electrode thickness and internal impedance (Barré et al., 2013). Because the BMS enforces voltage limits at the module level, the weakest cell in any module dictates the usable capacity of the entire module. Over the lifetime of the pack, this effect compounds as cells degrade

at different rates, progressively reducing effective capacity below the theoretical maximum.

The most common mitigation is passive cell balancing, which bleeds excess charge from higher-voltage cells through resistors and dissipates it as heat during charging. Passive balancing is effective for voltage equalization and overcharge prevention but provides no benefit during discharge, does not address capacity imbalances from differential aging, and wastes energy as resistive heat. Active balancing techniques transfer charge between cells using inductors, capacitors, or transformer-coupled circuits and offer improved energy efficiency but add hardware complexity and cost. Active balancing remains uncommon in volume production EV systems (Brandl et al., 2012).

2.3.2 The Case for Module-Level Monitoring and Switching in EVs

If an EV BMS could act on cell-level data for power routing decisions rather than using it solely for balancing and safety, the system could isolate weak cells before they limit module performance, utilize the full electrochemical capacity of the pack, and extend pack lifetime by reducing stress on degraded cells while maintaining system-level output (Ci et al., 2016).

Monitoring and switching all 7,104 individual cells in a Tesla Model S with the hardware granularity of the bench prototype is not practically feasible. The hardware overhead of thousands of individual voltage divider networks and corresponding solid-state relays would be prohibitive in mass, wiring complexity, and cost. However, the

architecture developed in this thesis is not limited to per-cell application. The same monitoring, switching, and scheduling logic can be applied at the module level, where each module containing tens to hundreds of cells is treated as a single switchable unit, analogous to how each 4S battery pack is treated in the bench prototype. A 16-module Tesla pack would require 64 SSRs (four per module for full positive and negative isolation on both motor and charge paths), comparable in scope to the validated nine-battery prototype. The monitoring hardware already exists in production vehicles; what does not exist is the control architecture to selectively disconnect a module, route it to a charging source, and rotate it back into service based on real-time SoC data.

The series hybrid supplementation concept also finds automotive precedent in the BMW i3 Range Extender and the Chevrolet Volt, both now discontinued, which paired battery-powered drivetrains with small combustion engines operating exclusively as generators (Saravanakumar et al., 2023). These implementations validated the series hybrid concept for passenger vehicles and demonstrated that combustion-supplemented battery systems can substantially extend operational range. The full analysis of applicability to the EV industry is presented in Chapter 7.

2.4 Power Electronics for Battery Systems

2.4.1 MOSFET-Based Switching and UPS Concepts

A metal-oxide-semiconductor field-effect transistor (MOSFET) is a voltage-controlled semiconductor switch that permits current flow between its drain and source terminals when the gate-to-source voltage exceeds a threshold value. The on-state

resistance ($R_{DS}(on)$) of the conductive channel determines the power dissipated as heat during operation according to:

$$P = I^2 \times R_{DS}(on)$$

where P is the dissipated power (W), I is the drain current (A), and $R_{DS}(on)$ is the on-state resistance (Ω). For the IRFB4110PbF power MOSFET considered in the initial switching design, $R_{DS}(on)$ is rated at 3.7 mΩ (International Rectifier, n.d.-b). At the per-motor peak current of approximately 80 A, this device dissipates 23.7 W. In a system requiring multiple MOSFETs switching simultaneously across a high-current bus, the aggregate thermal management challenge requires dedicated heatsinking to prevent junction temperatures from exceeding safe operating limits.

MOSFETs require carefully designed gate drive circuitry to ensure predictable switching behavior. High-side MOSFETs, which switch the positive rail of a power bus, require bootstrap or charge pump circuits to generate gate voltages above the source potential. Gate driver ICs such as the IR2110 are commonly used for this purpose (International Rectifier, n.d.-a). However, the gate drive circuit introduces additional failure points: if the gate is not held in a defined state during power-up, the MOSFET may conduct before the controller issues commands, resulting in uncontrolled current flow. Pulldown resistors, gate-source clamping, and sequenced power-up logic are all necessary preventive measures. The complexity and failure sensitivity of this supporting circuitry contributed to the design challenges documented in Section 4.4.2.

The initial switching architecture drew conceptual inspiration from uninterruptible power supply (UPS) systems used in data center and critical infrastructure applications. MOSFET-based UPS topologies achieve switching times on the order of nanoseconds, far faster than mechanical relays or contactors (Sahin, 2022). This zero-downtime transfer concept was the functional goal of the initial design: transitioning a battery from the motor bus to the charge bus so rapidly that the ESCs detect no discontinuity. Practical implementation at the required current levels encountered fundamental challenges that led to the architectural revision described in Section 4.4, which eliminated the need for nanosecond switching entirely.

2.4.2 Solid-State Relay (SSR) Technology

A solid-state relay (SSR) is a semiconductor-based switching device that performs the same function as an electromechanical relay without moving parts. The absence of mechanical contacts eliminates contact bounce, arc formation, and the mechanical wear that limits the operational life of traditional relays, making SSRs suitable for applications requiring frequent switching over extended periods. The internal architecture of a typical DC SSR consists of three functional stages: an input stage that receives a low-voltage control signal, an optical isolation stage providing galvanic separation between the control and load circuits using an LED-photodetector pair, and an output stage containing one or more power MOSFETs with integrated gate drive circuitry and protection networks. Because the manufacturer integrates all supporting circuitry within the SSR package, the end user is not required to design the bootstrap circuits, charge pumps, pulldown networks, or gate resistor configurations required by discrete MOSFETs.

The SSR selected for this thesis is the Fotek SSR-40DD, a DC-to-DC solid-state relay rated for a control input of 3 to 32 V DC and a load output of 5 to 60 V DC at a maximum continuous current of 40 A (Fotek, n.d.).

This device can be driven directly from the digital output pins of an Arduino Mega at 5 V logic levels without intermediate driver circuitry. The SSR-40DD provides switching times on the order of milliseconds rather than nanoseconds; as established by the architectural revision in Section 4.4, nanosecond speed is not required in the parallel-first topology, because removing one battery from a parallel junction does not interrupt the current path.

SSRs carry limitations that must be addressed in the system design. The on-state voltage drop across the internal MOSFETs generates heat proportional to load current, requiring thermal derating at sustained high currents and adequate heatsinking for continuous operation near the rated limit. A common failure mode is a short-circuit condition in which the internal MOSFET welds shut, leaving the SSR permanently conducting. The system firmware addresses this through diagnostic logic that compares commanded relay states against measured bus voltages, a capability enabled by the voltage monitoring subsystem. The complete SSR-based switching architecture, including the four-SSR-per-battery configuration that emerged from the ground-side isolation discovery, is detailed in Section 4.4.

Chapter 3: Skywalker Project Foundation

3.1 Skywalker Overview

Skywalker III was an undergraduate senior design project conducted at the Widener University School of Engineering during the 2024–2025 academic year. The project was carried out by an undergraduate engineering team, with the author serving as project manager under the advisement of Dr. Daniel Roozbahani. The primary objective was to design and construct a large-scale hexacopter UAV with an approximately 10-ft motor-to-motor span, incorporating an onboard combustion-powered generator to extend flight duration beyond the limits of battery-only operation.

Figure 10. Skywalker III Hexacopter

The airframe consisted of a three-level aluminum T-slot extrusion structure. The top level housed the flight controller, electronic speed controllers, and six motor assemblies mounted at the ends of radial arms. The middle level carried the lithium polymer battery packs and the high-current power distribution wiring connecting them to the motor bus. The bottom level supported the gasoline generator within a dedicated sub-chassis and a custom landing gear system incorporating shock absorbers to attenuate impact loads during touchdown. This three-tier arrangement separated high-vibration components from sensitive electronics while positioning the heaviest components near the geometric center of the airframe.

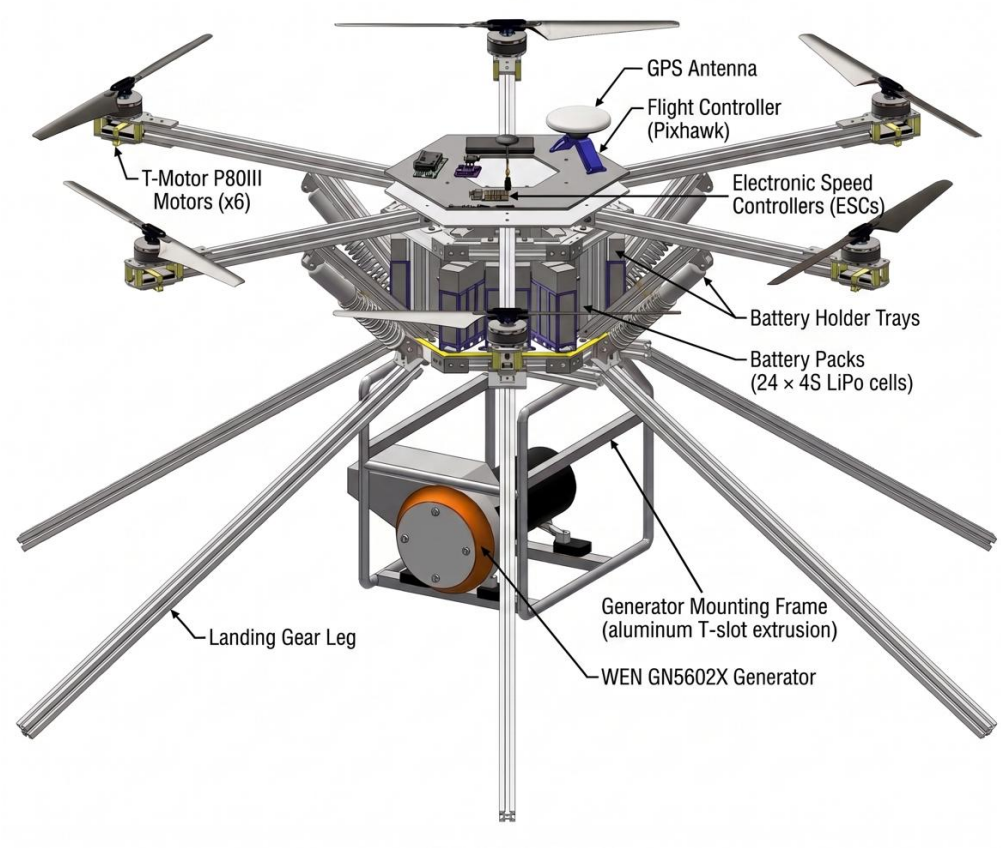

Figure 11. Skywalker III Component Layout and Major Subsystems.

Propulsion was provided by six T-Motor P80III KV120 brushless DC (BLDC) motors, each paired with a 30-inch T-Motor MF3016 propeller and driven by a Flycolor Flydragon V4 80 A ESC. Flight control was managed by a Pixhawk 2.4.8 running ArduPilot firmware, with telemetry and manual override provided through a TBS Tango II transmitter communicating via the TBS Crossfire long-range protocol. Position data was supplied by an ArduSimple SimpleRTK2B GPS module.

While the Skywalker III project encompassed the full scope of airframe design, propulsion integration, and flight system development, the remainder of this chapter focuses specifically on the power system architecture, as it is the subsystem most directly relevant to the research presented in this thesis. Complete details of the structural, flight control, and propulsion subsystems are documented in the Skywalker III senior design report.

3.2 Power System Architecture

3.2.1 Battery Configuration

The six BLDC motors and supporting avionics imposed a combined peak current demand of approximately 480 A at a nominal bus voltage of 44.4 V. No commercially available battery pack met both the voltage and current requirements at the capacity necessary for meaningful flight endurance. A custom series-parallel battery architecture was therefore developed using individual lithium polymer cells as the fundamental building block.

The design specified 48 Youme Power 4S 6500 mAh 80C lithium polymer cells as the primary energy storage, of which 24 were physically installed. Each 4S cell provides a nominal voltage of 14.8 V (3.7 V/cell × 4 cells) and a rated capacity of 6,500 mAh. Three of these 4S cells were wired in series to form a single 12S module, tripling the voltage to 44.4 V nominal (3.7 V/cell × 12 cells) while maintaining a per-module capacity of 6,500 mAh. The 12S voltage matched the rated input of the ESCs and the operating range of the BLDC motors (T-Motor, 2024).

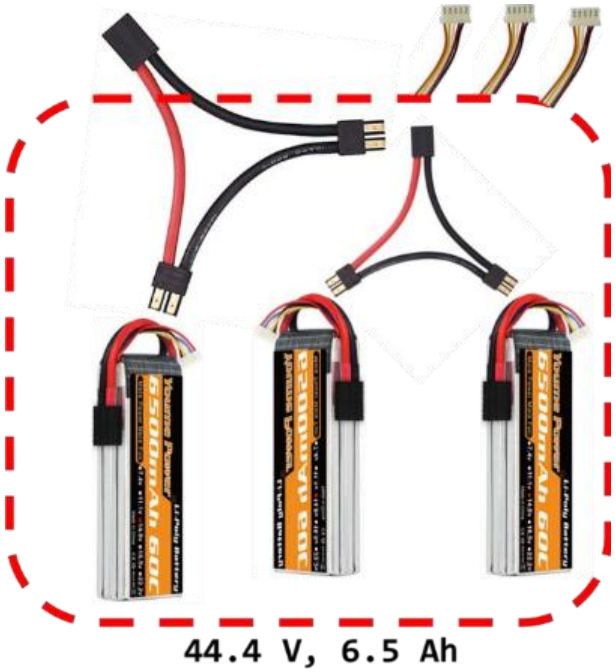

Figure 12. Three 4S LiPo Cells Joined in Series to Form a 12S Pack.

Eight of these 12S modules were then connected in parallel to scale total current capacity, preserving the 44.4 V bus voltage while summing the capacity of all connected modules. Table 3 summarizes the complete battery configuration.

Table 3. Skywalker III Battery Configuration Summary.

Parameter	As-Built	Design Target
Cell Model	Youme Power 4S 6500 mAh 80C LiPo	Youme Power 4S 6500 mAh 80C LiPo
Nominal Cell Voltage	14.8 V (3.7 V/cell × 4)	14.8 V (3.7 V/cell × 4)
Cell Capacity	6,500 mAh (6.5 Ah)	6,500 mAh (6.5 Ah)
Batteries per Module (Series)	3	3
Module Configuration	12S (44.4 V nominal)	12S (44.4 V nominal)
Modules in Parallel	8	16
Total Battery Count	24	48
Total Bank Capacity	52 Ah (8 × 6.5 Ah)	104 Ah (16 × 6.5 Ah)
Total Stored Energy	~2,300 Wh	~4,600 Wh
Nominal Bus Voltage	44.4 V	44.4 V
Peak System Current	~480 A	~480 A

At the system's peak current demand, this configuration provided approximately 10 to 15 minutes of battery-only flight time before the cells would reach their minimum safe operating voltage. This limited endurance was the fundamental constraint motivating the hybrid architecture: extending operational duration required supplementing battery energy with an onboard generation source rather than simply adding more cells, as each additional cell contributes mass that increases the power required to sustain flight.

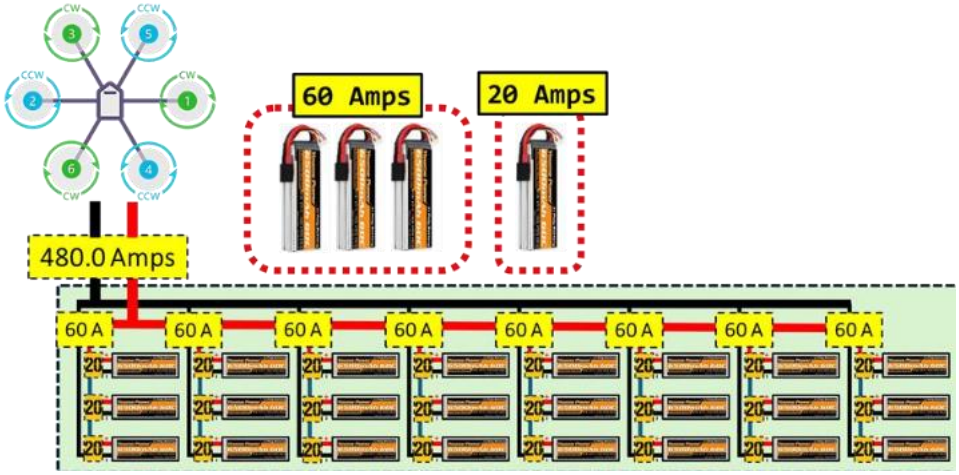

Figure 13. Skywalker III Current Demand Distribution Across the 24-Cell Battery Bank.

The operational concept divided the modules into two groups. At design capacity (16 modules), each group would contain eight modules; the as-built configuration of eight modules used groups of four. At any given time, one group would supply current to the motor bus while the second group received charge from the onboard generator through an AC-to-DC charger. Each group, consisting of eight parallel modules at 44.4 V nominal, carried a combined capacity of 52 Ah and approximately 2,300 Wh. An intelligent management layer would monitor the SoC of each group and automatically swap their roles when the active group is depleted to a predetermined threshold. The recently charged group would then assume the motor load, and the depleted group would transition to the charger. This dual-group rotation was intended to enable flight duration limited only by the generator's fuel supply.

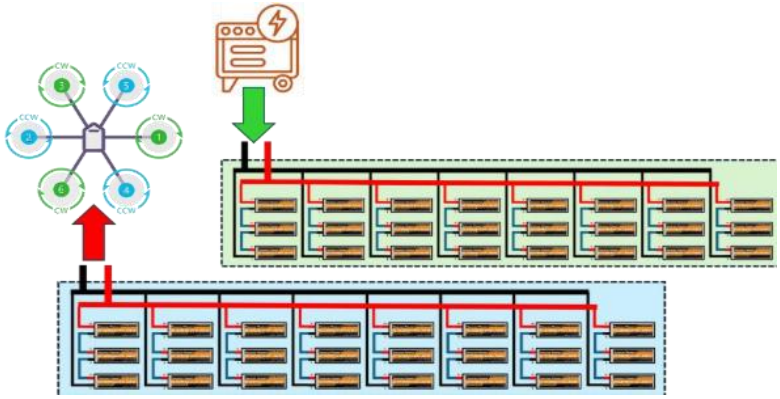

Figure 14. Skywalker III Two-Group Switching Concept.

However, the monitoring and switching intelligence required to execute this concept remained at a theoretical level during the undergraduate project. The project scope prioritized airframe construction, propulsion integration, and flight testing, and the battery management layer was identified as a separate research effort requiring dedicated development.

3.2.2 Generator Integration

The supplemental power source selected for the Skywalker III platform was a WEN GN5602X portable gasoline generator. Table 4 summarizes the generator specifications relevant to the power system design.

Table 4. WEN GN5602X Generator Specifications.

Parameter	Value
Model	WEN GN5602X
Engine Type	224 cc 4-Stroke OHV Single Cylinder
Continuous Rated Power	4,500 W

Peak Rated Power	5,600 W
Fuel Tank Capacity	16.5 L (Gasoline)
Dimensions	579 × 599 × 584 mm
Fuel Energy Content	34,200 kJ/L (Gasoline)

The generator's role in the Skywalker III architecture was explicitly that of a battery charging source, not a direct motive power source, for the reaction-time reasons established in Section 1.1. The generator was physically mounted to the bottom level of the airframe chassis, suspended beneath the battery and power distribution level. A custom landing gear system incorporating shock absorbers was installed beneath the generator sub-chassis to absorb touchdown forces.

The energy available from the generator's fuel supply was estimated to establish the theoretical endurance of the hybrid system. At a fuel energy content of 34,200 kJ/L, the 16.5 L tank contains approximately 156,750 Wh of chemical energy. Assuming a thermal efficiency of 25 percent for the four-stroke engine, the usable electrical output over the full tank is approximately 39,200 Wh. This represents roughly 8.5 times the total energy stored in the fully charged battery bank (2,300 Wh as-built; 4,600 Wh at design capacity). Even after further losses through the AC-to-DC charger, the fuel-based energy reserve substantially exceeds the battery-only capacity, reinforcing the fundamental rationale for the hybrid architecture.

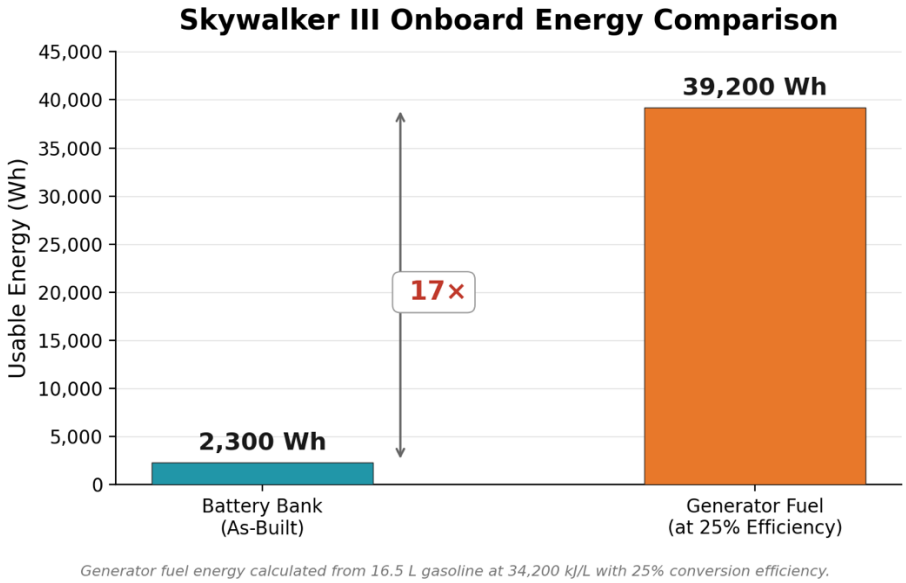

Figure 15. Onboard Usable Energy Comparison Between the Battery Bank and the Generator Fuel Reserve.

Although the generator was physically installed on the airframe during construction, full electrical integration with the battery system was never completed. The charging path from generator output to battery input, the switching mechanism to route batteries between buses, and the management intelligence to coordinate these operations all remained at a conceptual stage by the end of the undergraduate project.

3.2.3 Switching Concept

The power management concept developed during the Skywalker III project was based on a two-group switching model. While one group of eight 12S modules powered the motors, the generator would charge the second group. When the SoC of the active group fell to a theoretical threshold of 30 percent, the system would swap group roles. This cycle was intended to repeat continuously for as long as the generator had fuel.

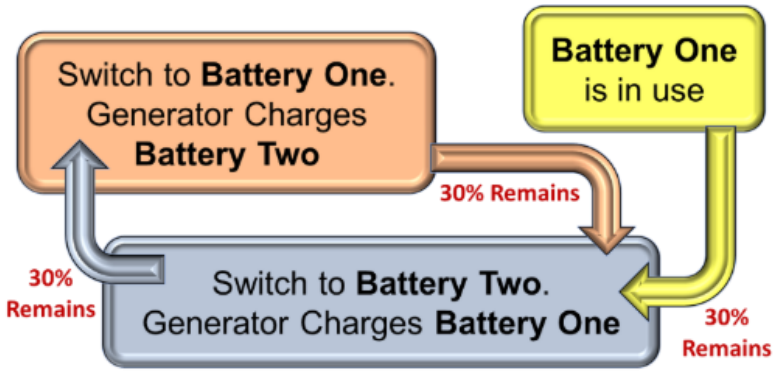

Figure 16. Theoretical Two-Group State Rotation Logic at the 30% SoC Threshold.

The 30 percent threshold was selected as a conservative operating floor. As described in Section 2.1.2, LiPo cells exhibit a steep voltage decline below approximately 20 percent SoC, and sustained discharge into this region accelerates electrode degradation. Setting the switching threshold at 30 percent maintained a safety margin above the critical voltage region while allowing sufficient discharge depth to provide the charging system adequate time to replenish the offline group.

Three critical questions remained unresolved at the conclusion of the undergraduate project, each of which carried directly into the research scope of this thesis.

The first concerned monitoring. The system had no mechanism for determining the SoC of either battery group: no voltage sensing circuits, no analog-to-digital conversion hardware, and no firmware estimating remaining capacity. Without this capability, the system could not detect when the active group had reached the 30 percent threshold.

The second concerned the switching mechanism. The active battery group supplies approximately 480 A to the motor bus at peak demand. Any interruption in this

current path, even for a fraction of a second, would cause the ESCs to lose input power and the motors to decelerate. On a hexacopter with no aerodynamic control surfaces, a momentary loss of motor power produces immediate and unrecoverable loss of attitude control. The switching mechanism therefore needed to transfer the motor load with zero perceptible interruption, a requirement demanding either extremely fast semiconductor switching or an architecture in which the transition could occur without breaking the current path.

The third concerned charge-rate sustainability. Whether the charger's output rate was sufficient to replenish the offline group before the active group drained to the 30 percent threshold depended on the actual power consumption profile during flight. No analysis or simulation had been performed during the undergraduate project to evaluate this question. It is addressed quantitatively through simulation in Chapter 6.

3.3 Lessons Learned and Research Motivation

The Skywalker III project provided direct experience with every subsystem required to build and operate a large-scale multirotor UAV. The structural airframe, flight controller, and propulsion system each had established design methodologies and prior implementations from which to draw guidance. The power management subsystem had none of these advantages. Distributing 480 A of current from 48 LiPo cells in a series-parallel topology, monitoring individual cell health, and switching groups between motor and charge buses represented a convergence of power electronics, embedded

systems, electrochemistry, and control logic for which no commercially available solution existed.

The project also exposed a fundamental limitation in the two-group model. Treating each group of eight modules as a monolithic unit provided no visibility into the condition of individual modules or cells within a group. If one module degraded faster than the others, the entire group's performance would be constrained by that weakest module with no way to identify, isolate, or redistribute its load. Achieving meaningful battery management required monitoring at the individual cell level and switching at the individual module level.

Beyond the Skywalker III platform, a broader observation emerged: the three core challenges, monitoring cell-level voltages in a multi-cell architecture, estimating SoC, and switching individual power sources between load and charge paths without interrupting operation, are not unique to hexacopter UAVs. Electric vehicles, ground-based robotic systems, and marine vehicles face the same challenges, differing only in cell count and current magnitude.

This recognition motivated a deliberate separation of the intelligent power management problem from the hexacopter platform. A reduced-scale bench prototype was designed and built independently, enabling rapid iteration on the monitoring, switching, and scheduling subsystems. The resulting system was validated at three-, six-, and nine-battery configurations, with the full 48-battery Skywalker III configuration evaluated through simulation. The design, development, and validation of this standalone intelligent power management system are the subject of the remaining chapters.

Chapter 4: Intelligent Power Management System Design

4.1 System Overview and Architecture

The intelligent power management system developed in this thesis comprises three core subsystems operating as a single cohesive unit. The first is a voltage monitoring circuit that reads individual cell voltages from every battery in the system and converts those readings into real-time state-of-charge estimations. The second is a power switching network built on solid-state relays capable of connecting or disconnecting any individual battery from the motor bus or the charge bus on command. The third is a software layer that integrates the monitoring and switching subsystems: a Python-based dashboard that aggregates incoming data, displays the full system state in real time, and provides diagnostic visibility for system validation during testing.

The full-scale Skywalker III power system, as described in Chapter 3, was designed around 48 batteries organized into 16 parallel modules of three series-connected cells each. The bench prototype developed for this thesis operates at a reduced scale of nine batteries due to scope and resource constraints inherent to a single-researcher graduate effort. This nine-battery configuration represents the minimum scale at which the complete series-parallel junction topology can be demonstrated with meaningful scheduling flexibility. No element of the monitoring, switching, or control architecture depends on the nine-battery count; every subsystem was designed to scale to the full 48-battery configuration or beyond.

The physical architecture employs a parallel-first, series-second battery topology. Nine Youme Power 4S 6500 mAh 80C lithium polymer batteries are organized into three groups of three, with each group residing at a junction point. Within each junction, the three batteries are wired in parallel. The three junctions are then connected in series, producing a total system voltage equal to the sum of the three junction voltages. At nominal conditions, each junction contributes approximately 14.8 V, yielding a system voltage of 44.4 V. At full charge (4.2 V per cell), this rises to 50.4 V. The rationale for selecting this parallel-first topology over the series-first approach used in the original Skywalker III design is documented in Section 4.4.

Every battery in the system exists in one of three operational states at any given time:

MOTOR: the battery is actively connected to the motor bus and contributing current to the load.

CHARGE: the battery has been disconnected from the motor bus and connected to the charge bus, where it receives current from the charging source.

IDLE: the battery is electrically isolated from both buses.

The SSR network controls these state transitions by connecting or disconnecting both the positive and negative leads of each battery, as described in Section 4.4.4.

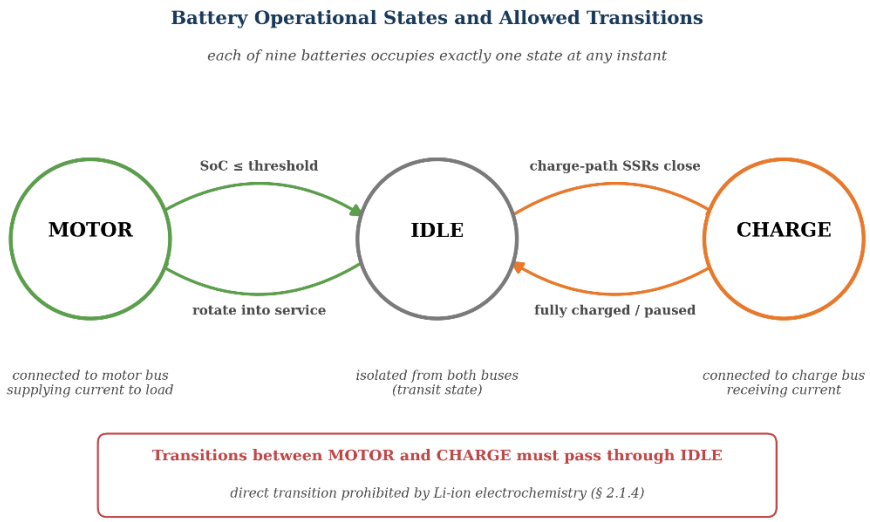

Figure 17. Three-State Battery Model with Allowed Transitions.

The single most critical constraint governing all system operations is the junction rule: each of the three junctions must always maintain at least one battery in the MOTOR state. If any junction drops to zero active batteries, the series chain is broken, and the entire system loses power instantaneously with no possibility of recovery. This constraint is enforced at every layer of the architecture, from the scheduling algorithm to the SSR control firmware, and cannot be overridden. An emergency shutdown command is also implemented, capable of setting all 36 SSRs to the off state simultaneously, removing all batteries from both buses for rapid de-energization in the event of a fault condition.

The monitoring subsystem employs three Arduino Mega 2560 microcontrollers, one per junction, each reading the four individual cell voltages from its three assigned batteries through resistive voltage divider networks. These readings are transmitted over

USB serial to a monitoring laptop, where the Python dashboard renders a live display of all 36 cells, all nine batteries, and all three junctions on a single interface. The SSR switching network is controlled by a fourth Arduino Mega that receives serial commands and actuates the appropriate relays.

4.2 Design Requirements and Constraints

The design requirements were established by the functional capabilities the system was expected to deliver and by the electrical characteristics of the hardware it was designed to manage.

The system was required to monitor individual cell voltages across all batteries in real time at the individual cell level, consistent with the thesis argument that cell-level visibility detects degradation, imbalance, and incipient failure conditions that aggregate monitoring misses. From these cell voltage readings, the system was required to produce reliable SoC estimations suitable for driving automated switching decisions.

The switching architecture was required to transition any individual battery between the motor bus, the charge bus, or an idle state without interrupting power delivery to the remaining batteries. During a state transition, the motors continue to draw current from the remaining active batteries at each junction, and the transition of one battery occurs without affecting the rest of the series chain. The switching hardware was required to provide full electrical isolation of both positive and negative leads when disconnecting a battery from either bus. This dual-rail isolation requirement emerged

during development when positive-rail-only switching proved insufficient for multi-battery charging, as documented in Section 4.4.4.

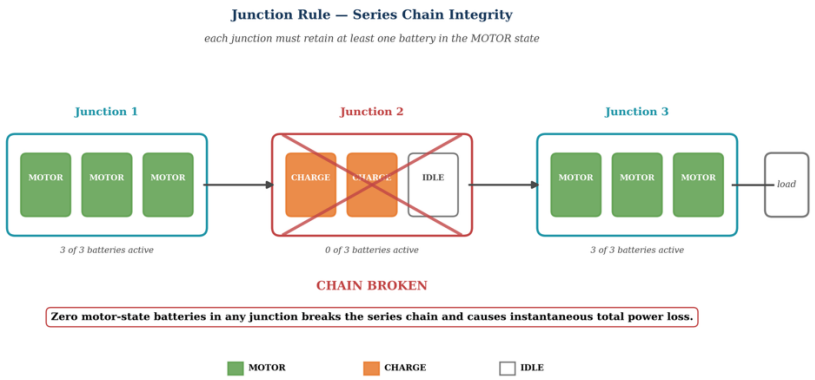

Figure 18. Junction Rule Enforcing Series-Chain Integrity Across All Three Junctions.

On the hardware side, the electrical requirements were established by the Skywalker III propulsion system. Six T-Motor P80III KV120 BLDC motors driven by Flycolor Flydragon V4 80 A ESCs impose a combined peak current demand of approximately 480 A at 44.4 V (T-Motor, 2024). Any component in the power path was required to be rated for both the system voltage and the per-junction current levels demanded by the motors underload.

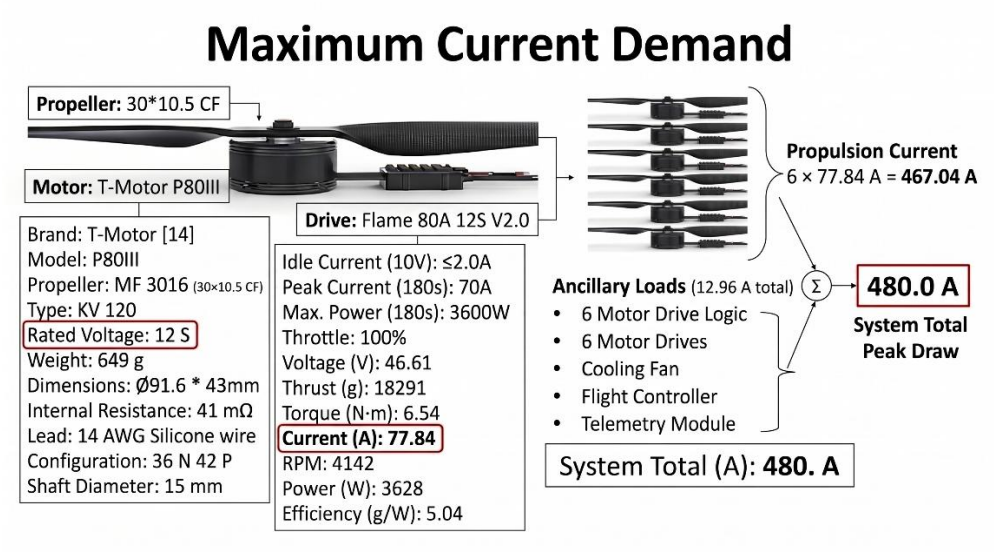

Figure 19. Derivation of the 480 A Peak System Current Demand.

The monitoring hardware was required to accommodate voltages up to 16.8 V per battery (4.2 V per cell at full charge) while interfacing with the Arduino Mega's 10-bit ADC, which accepts a maximum input of 5.0 V. At a 5.0 V reference voltage and 10-bit resolution (1,024 discrete steps), the ADC provides approximately 4.88 mV per step. After voltage division, each ADC step corresponds to approximately 21 mV at the battery tap, establishing the measurement resolution of the monitoring system.

The overall architecture was designed to be platform independent. The monitoring logic, SoC estimation method, switching architecture, and scheduling algorithms were all structured to be generalizable and scalable to larger configurations. This requirement motivated the simulation work presented in Chapter 6, where the same algorithms were evaluated at both the nine-battery prototype scale and the full 48-battery Skywalker III configuration.

4.3 Voltage Monitoring Circuit

4.3.1 Arduino-Based Monitoring Platform

The monitoring subsystem is built on three Arduino Mega 2560 microcontrollers, one assigned to each of the three junctions. The Arduino Mega was selected over other

microcontroller platforms based on analog input pin count. Each 4S battery requires four analog channels to read its four cell voltage taps, and each monitoring board is responsible for three batteries, consuming 12 analog input pins per board. The Arduino Mega provides 16 analog inputs, accommodating the 12 channels per board with four pins remaining as spares. An Arduino Uno, by comparison, provides only six analog inputs, insufficient to monitor even a single three-battery junction without external multiplexing hardware.

Each Arduino Mega is physically co-located with a dedicated breadboard containing the resistive voltage divider networks for its three assigned batteries. All three monitoring boards connect to the laptop through a powered USB hub, each maintaining an independent serial connection on its own COM port. The Python-based monitoring dashboard receives all three serial data streams simultaneously and assembles the incoming cell voltage and SoC data into a unified real-time display.

4.3.2 Individual Cell Voltage Measurement

Each battery interfaces with its monitoring board through the JST-XH balance connector, a standard five-wire connector present on commercially available LiPo packs. The first wire provides the ground reference. Wires two through five are voltage taps corresponding to the cumulative series voltage at each cell boundary within the four-cell stack. Tap 1 reads the voltage across cell 1 alone (approximately 3.7 V nominal). Tap 2 reads the cumulative voltage of cells 1 and 2 (approximately 7.4 V). Tap 3 reads cells 1

through 3 (approximately 11.1 V). Tap 4 reads the full pack voltage (approximately 14.8 V nominal, rising to 16.8 V at full charge).

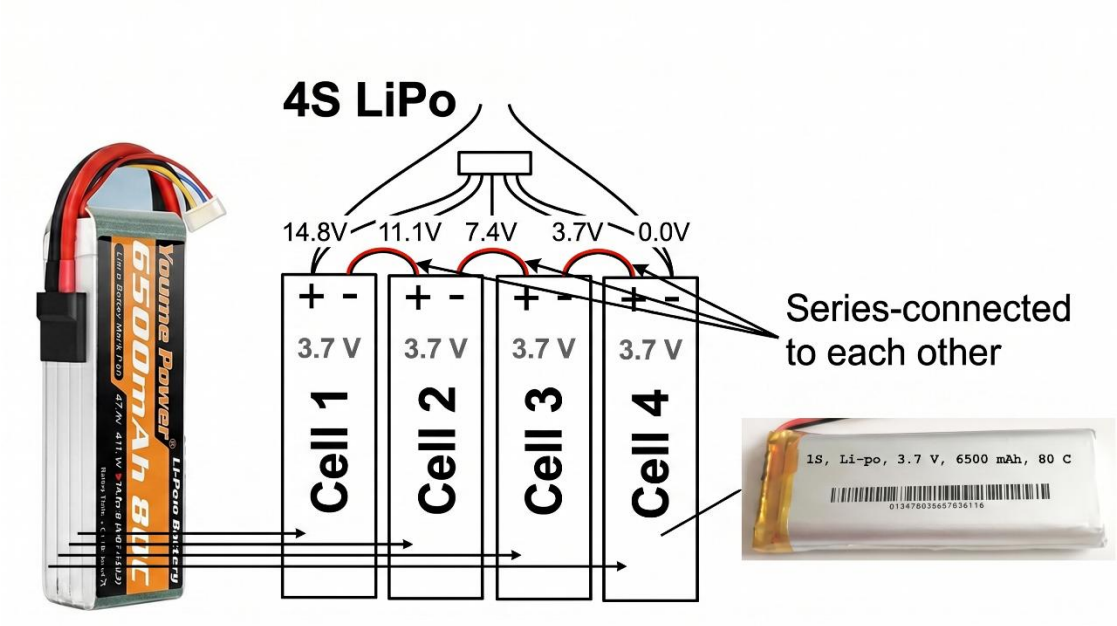

Figure 20. Internal Series Construction of a Youme 4S 6500 mAh LiPo Battery.

Because the Arduino Mega's ADC tolerates a maximum input of 5.0 V, each tap voltage is scaled down through a resistive voltage divider before reaching the analog input pin. Each divider employs a 33 kΩ upper resistor and a 10 kΩ lower resistor. The output voltage of the divider is given by:

$$V_{out} = \frac{V_{tap} \times R_{lower}}{(R_{upper} + R_{lower})}$$

where V_{out} is the voltage presented to the Arduino ADC input (V), V_{tap} is the voltage at the balance connector tap (V), R_{lower} is the resistance of the lower divider leg (10 kΩ), and R_{upper} is the resistance of the upper divider leg (33 kΩ). For this configuration, the division ratio is 10 / (33 + 10) = 0.233, scaling the maximum tap

voltage of 16.8 V to approximately 3.91 V at the ADC pin. This provides a margin of over 1.0 V below the 5.0 V input limit.

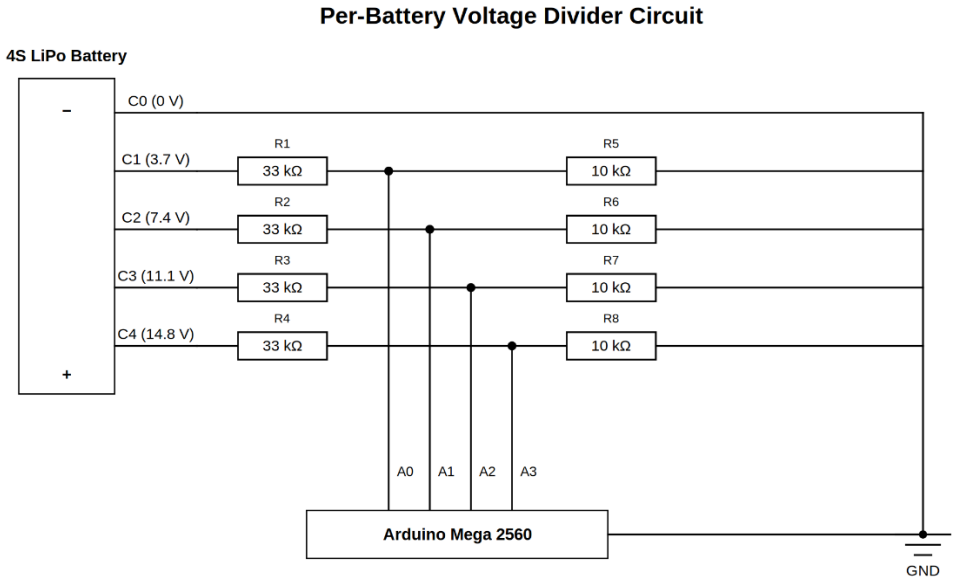

Figure 21. Per-Battery Voltage Divider Network for Individual Cell Measurement.

The 10 kΩ lower resistor in each divider serves a dual function: it completes the voltage divider circuit, and it acts as a pulldown resistor that ties the analog input pin to ground when no battery is connected. Without this pulldown, a disconnected analog pin on the Arduino Mega floats at an undefined voltage, picking up electromagnetic interference from adjacent wiring. The pulldown guarantees that a disconnected channel reads zero rather than a spurious value.

Across all three monitoring boards, the system employs 36 individual voltage dividers: four per battery, nine batteries, with each of the 36 cells measured on its own dedicated analog channel. On the firmware side, individual cell voltages are recovered through a subtraction process. The firmware reads all four tap voltages for a given

battery, converts each raw 10-bit ADC value back to the actual tap voltage using the known division ratio, and then isolates each cell voltage as follows:

$$V_{cell1} = V_{tap1}$$

$$V_{cell2} = V_{tap2} - V_{tap1}$$

$$V_{cell3} = V_{tap3} - V_{tap2}$$

$$V_{cell4} = V_{tap4} - V_{tap3}$$

where V_{tap1} through V_{tap4} are the measured voltages at each successive balance connector tap (V) and V_{cell1} through V_{cell4} are the isolated individual cell voltages (V). This subtraction method enables per-cell monitoring using only the stacked balance connector taps, without requiring additional sensing hardware or differential measurement circuits.

4.3.3 State-of-Charge Estimation via Cell Voltage

Once individual cell voltages are isolated, each is mapped to a state-of-charge percentage through a firmware lookup table derived from the manufacturer's published discharge curve for the Youme 4S 6500 mAh 80C cell at a 1C discharge rate (Youme Power, n.d.). The complete lookup table is shown in Table 5.

Table 5. OCV-SoC Lookup Table (Youme Power, n.d.).

Cell Voltage (V)	**SoC (%)**
4.2	100
4.1	95

3.95	80
3.85	65
3.80	50
3.75	35
3.70	25
3.60	15
3.50	10
3.30	5
3.0	0

For measured voltages falling between two entries in the lookup table, the firmware performs linear interpolation between the two nearest entries to produce a continuous SoC estimate rather than a stepwise approximation. This smoothing eliminates discontinuous jumps in the displayed SoC value that would otherwise occur as a cell's voltage crosses a lookup table boundary.

For each battery's overall SoC, the system reports the minimum SoC among its four constituent cells rather than the average. This design decision reflects the physical reality that a series-connected battery is limited by the capacity of its weakest cell. If three cells indicate 80 percent and the fourth indicates 40 percent, the battery-level SoC is reported as 40 percent, because that cell will reach its cutoff voltage first and dictate the point at which the battery is no longer safe to discharge. Reporting the average (70

percent in this example) would mask the degraded condition of the weakest cell and risk continued discharge below its safe operating voltage. The minimum-cell approach guarantees that any cell exhibiting accelerated degradation, elevated internal resistance, or manufacturing variance is immediately visible to the scheduling algorithm and to the operator.

4.3.4 Real-Time Monitoring Dashboard

All 36 cell voltages, their corresponding SoC percentages, and battery state data are aggregated and displayed by a custom Python dashboard application running on the monitoring laptop. The dashboard receives serial data from all three monitoring Arduinos simultaneously and renders a live view of the entire system on a single interface.

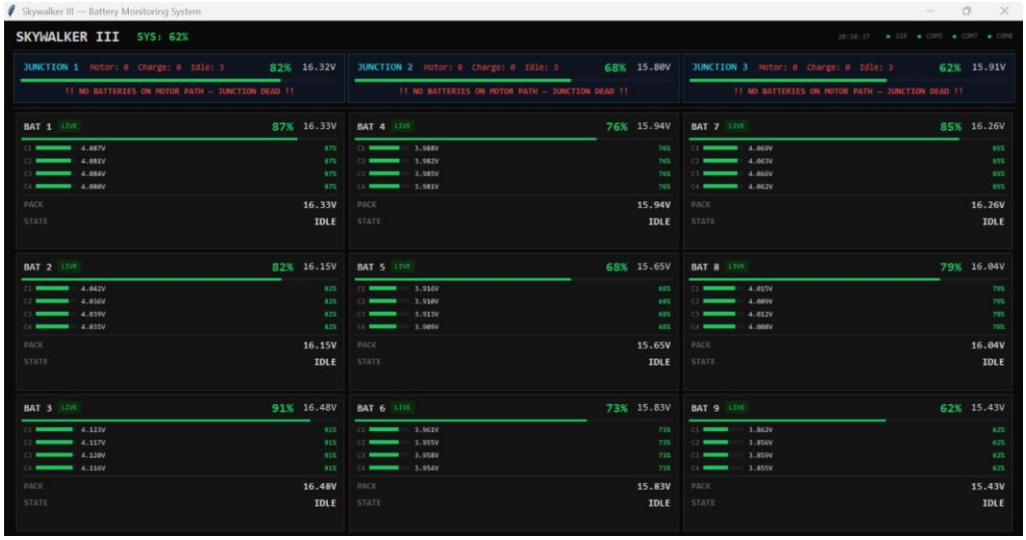

Figure 22. Real-Time Battery Monitoring Dashboard Displaying All Nine Batteries Across Three Junctions.

The primary display is organized as a three-by-three grid of battery cards matching the physical layout: batteries 1 through 3 in the top row corresponding to Junction 1, batteries 4 through 6 in the middle row for Junction 2, and batteries 7 through 9 in the bottom row for Junction 3. Each battery card displays the pack-level SoC percentage, total pack voltage, and a breakdown of the four individual cells with their voltages and horizontal bar graphs color-coded from green to yellow as charge decreases. Each card also displays the battery's current operational state: IDLE, MOTOR, or CHARGE.

A system header presents the overall system SoC, per-junction breakdowns showing how many batteries are assigned to each state, and the measured junction voltages. If any junction drops to zero batteries in the MOTOR state, a red warning banner activates. Connection status indicators confirm that each serial port and the SSR control link are active and transferring data. The complete dashboard source code is provided in Appendix B.1.

4.4 Power Switching System Evolution

4.4.1 Initial Approach: MOSFET-Based UPS Circuit

The original switching design was modeled after commercial UPS architecture, targeting nanosecond-scale zero-interruption transfer between battery groups to prevent any discontinuity detectable by the ESCs. The circuit employed four IRFB4110PbF power MOSFETs, each rated at 100 V and 180 A continuous with an on-state resistance of 3.7 mΩ (International Rectifier, n.d.-b). The MOSFETs were arranged in a low-side

switching configuration with two devices wired in parallel per battery path to increase current capacity. Each parallel MOSFET pair was driven by an IR2110 gate driver IC operating in low-side mode. Supporting components included UF4007 ultrafast recovery diodes for freewheeling protection, gate-source capacitors for noise suppression, and gate resistors for current limiting. All battery negative terminals were connected to a shared common ground rail.

4.4.2 MOSFET Failure Analysis

The MOSFET-based switching circuit underwent three successive rounds of failure during development, each revealing a distinct engineering issue. These failures are documented as a technical failure analysis, as the root causes informed the fundamental architectural decision that followed.

In the first iteration, the gate drive resistors overheated and failed within seconds of power-up. The root cause was the absence of pulldown resistors on the MOSFET gates. Without a defined low-state bias during power-up, the gates floated to an undefined voltage, causing the MOSFETs to conduct before the IR2110 gate drivers established command authority. The resulting uncontrolled inrush current exceeded the thermal rating of the gate resistors.

In the second iteration, pulldown resistors were added. However, a multimeter at the motor junction still read approximately 50 V with the MOSFETs commanded off. Drain-to-source resistance measurements returned 0 Ω, a value that should exceed 1 MΩ for a properly functioning device in the off state. The thermal damage from the first

iteration had permanently fused the internal drain-to-source junctions, welding the devices into a permanently conductive state.

In the third iteration, the circuit was rebuilt with new MOSFET devices and the corrected pulldown configuration. The IR2110 gate driver ICs began overheating within seconds. Investigation determined that an external voltage source was connected to a pin designated as a charge pump output, driving approximately 500 mA into a pin not designed to accept external current. The resulting overcurrent condition destroyed the ICs.

The three failure modes (uncontrolled gate float, latent thermal welding, and driver IC overcurrent) are well-documented risks in high-current discrete MOSFET designs and are individually addressable. However, the cumulative complexity of the gate drive circuitry, thermal management, and protection networks required to operate discrete MOSFETs reliably at these current levels raised a more fundamental question about the switching architecture itself.

4.4.3 Architectural Reassessment and Topology Revision

The MOSFET failures prompted a reassessment not of the component selection, but of the underlying assumption that nanosecond switching speed was a design requirement. The need for nanosecond transfer existed only because the original battery topology wired cells in series' first: three 4S batteries formed a 12S string, and switching meant transferring the entire motor load from one string to another instantaneously. Any gap in that transfer would interrupt current to the ESCs.

The critical insight was that the battery topology itself could be restructured to eliminate this constraint. If the wiring order were reversed (parallel first, then series), individual batteries within a junction could be removed without breaking the current path, because the remaining parallel batteries at that junction would continue to supply the load. In this revised topology, switching speed is no longer a determining factor; one source is simply removed from a parallel group while the others continue to operate.

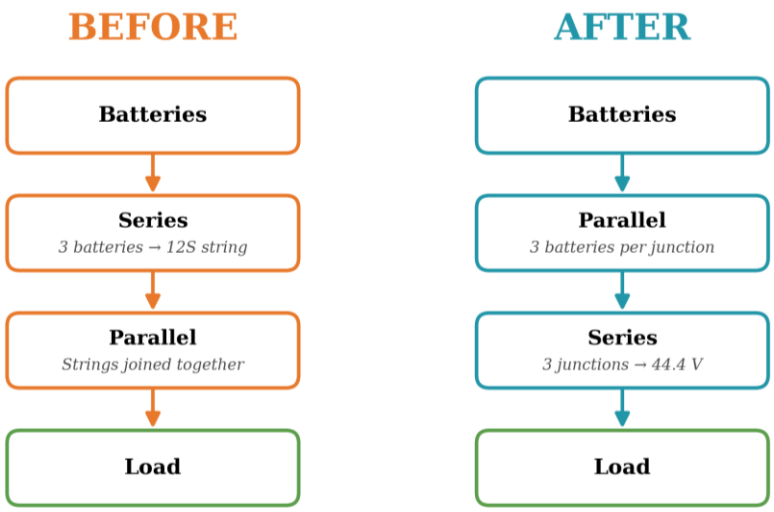

Figure 23. Topology Revision from Series-First to Parallel-First Battery Architecture.

This architectural change carried implications beyond switching speed. In the series-first topology, three batteries formed a monolithic unit with no individual control. The parallel-first topology enables per-battery control: any single battery can be disconnected from its junction independently while the remaining batteries maintain the junction voltage. This per-battery control is what enables the scheduling algorithms developed in this thesis.

With the nanosecond speed requirement eliminated, the component selection shifted to solid-state relays. The SSR-40DD (specifications in Section 2.4.2) is driven directly from the Arduino Mega's 5 V digital output pins, requiring no gate driver IC, bootstrap circuit, or pulldown network. Heatsinks are mounted on all SSRs to manage on-state thermal dissipation.

A dedicated fourth Arduino Mega serves as the SSR controller, driving all 36 relays through digital output pins. Each battery is assigned a pair of control pins: one for its motor path relay and one for its charge path relay. The controller accepts two-character serial commands transmitted from the laptop, where the first character identifies the battery number (1–9) and the second character specifies the desired state. The command "1A" places battery 1 on the motor bus, "1B" places it on the charge bus, and "1C" sets it to idle. Two global commands are implemented: "AA" activates the motor path for all nine batteries simultaneously, and the emergency kill command deactivates all 36 SSRs instantly.

A two-second safety delay is enforced on every state transition. When a battery transitions from MOTOR to CHARGE, the motor-path SSRs are deactivated first, followed by a two-second pause, after which the charge-path SSRs are activated. This sequencing guarantees that the motor path is fully deactivated before the charge path becomes live, preventing any condition in which both paths could be simultaneously active. Table 6 summarizes the SSR logic for each battery state.

Table 6. SSR State Logic.

Motor Path SSR	Charge Path SSR	Battery State
ON	OFF	**MOTOR**
OFF	ON	**CHARGE**
OFF	OFF	**IDLE**

All batteries initialize into IDLE upon system power-up. The firmware enforces conflict prevention: any command that would activate both paths on the same battery simultaneously is rejected. The emergency kills command overrides all other states and cannot be blocked. The complete SSR controller firmware is provided in Appendix A.2.

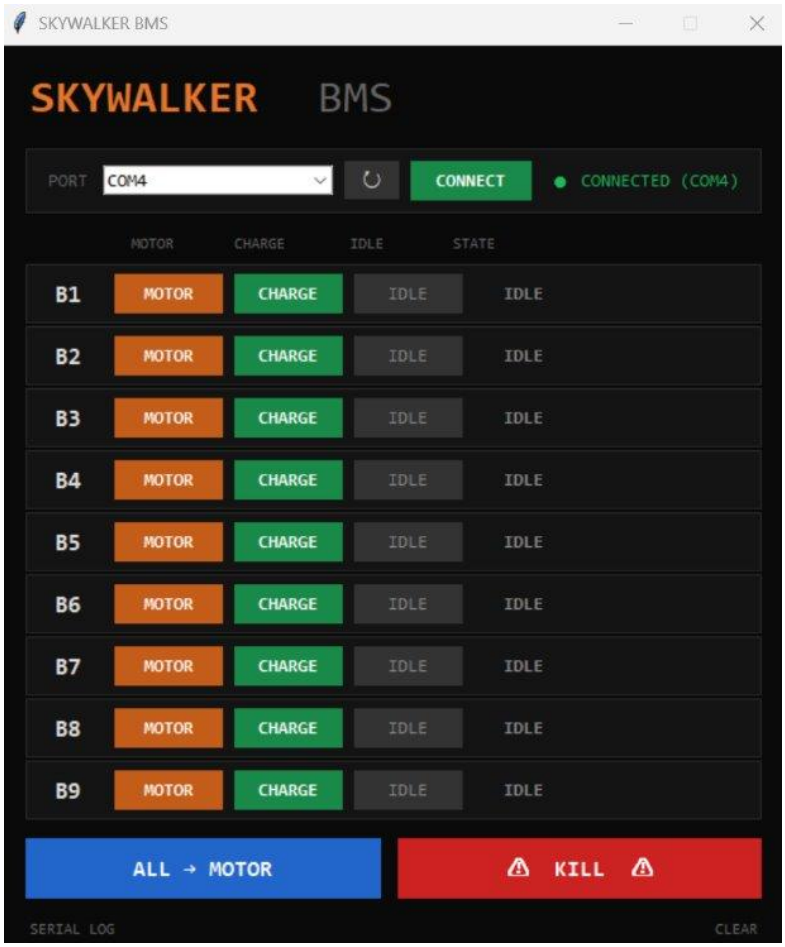

Figure 24. Manual SSR Control Interface.

4.4.4 Ground-Side Isolation Discovery

The initial SSR implementation placed relays only on the positive rail of each battery path. Each battery had two SSRs on its positive lead, one for the motor bus and one for the charge bus, while all nine negative terminals shared a common ground rail with no switching on the negative side.

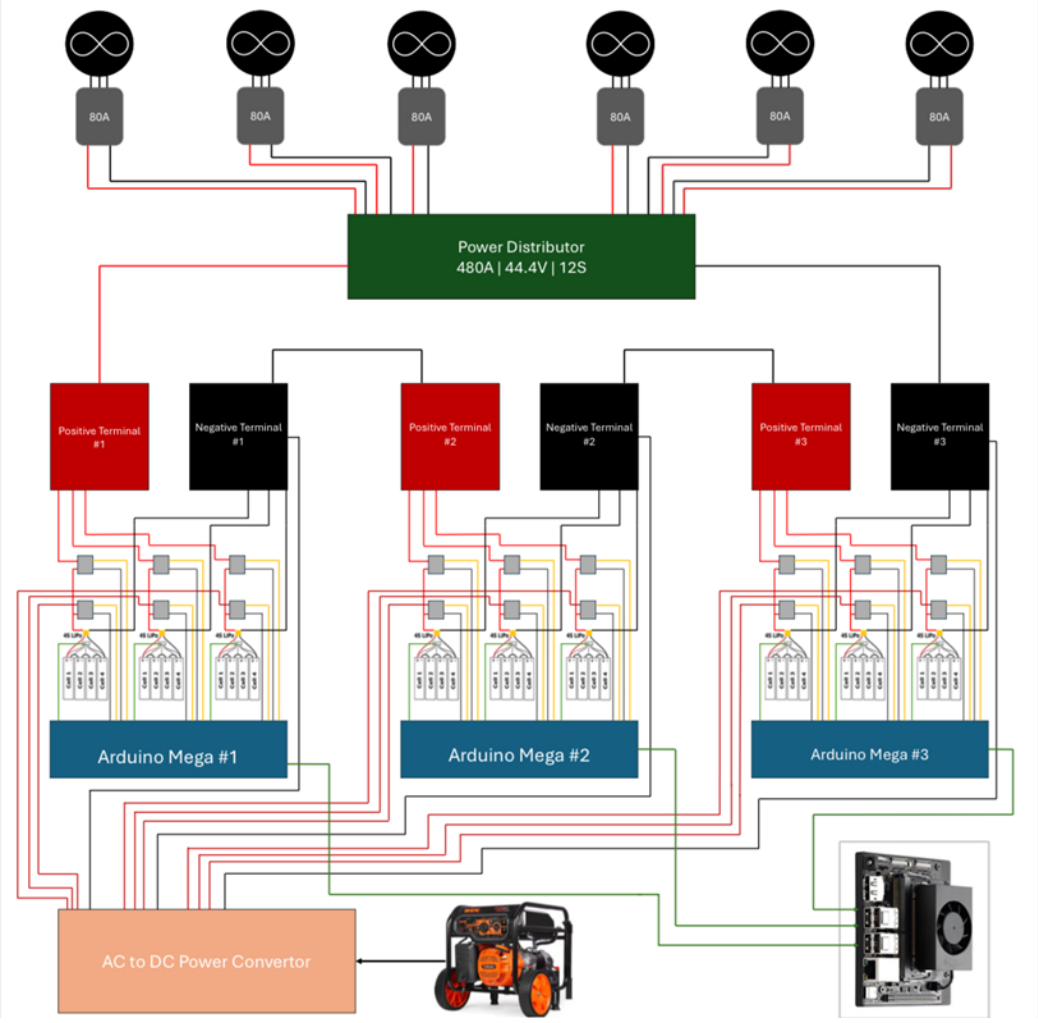

Figure 25. Full Integrated System Schematic.

This configuration worked when a single battery was placed on the charge bus in isolation: the EV Peak CQ3 charger identified the battery, initiated its CC/CV profile, and completed the charge cycle without incident. However, when a second battery was connected to a second charger channel simultaneously, the charger immediately reported a reverse polarity error.

The fault originated in the battery system. With two batteries at different states of charge both connected to the shared ground rail, a voltage differential drove current directly from the higher-voltage battery into the lower-voltage battery through the shared ground path, bypassing the charger entirely. The charger detected the resulting voltage conflict and triggered its reverse polarity protection. The positive-side SSRs were correctly isolating the positive rails, but the shared ground provided an unrestricted return path that defeated the isolation.

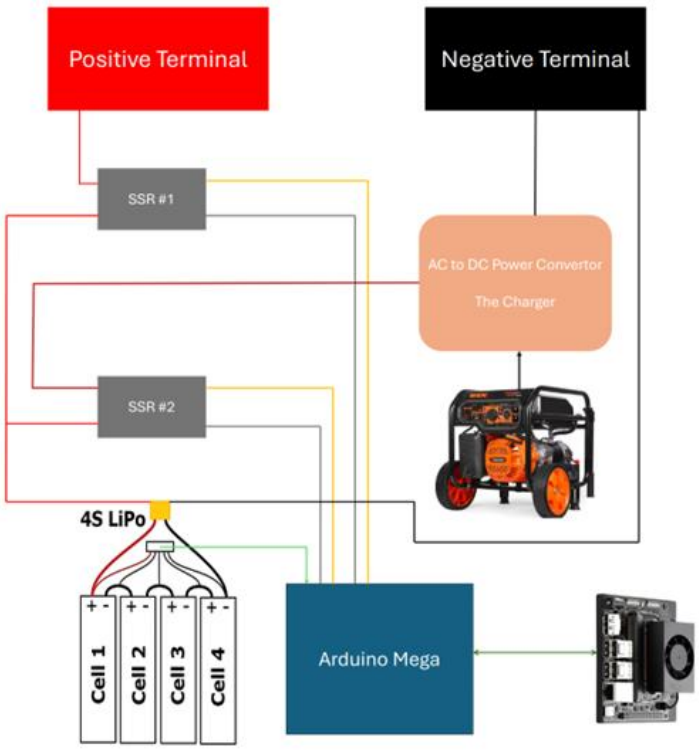

Figure 26. Two-SSR-per-Battery Topology with Shared Ground Return Path.

The resolution required adding SSR switching to the negative leads of the charge path. When a battery is placed on the charge bus, both its positive and negative charge-path SSRs close while both motor-path SSRs open, creating a fully isolated charging circuit with no electrical connection back to the shared ground rail or any other battery. When the battery returns to the motor bus, the charge-path SSRs open and the motor-path SSRs reconnect it to the shared power and ground rails.

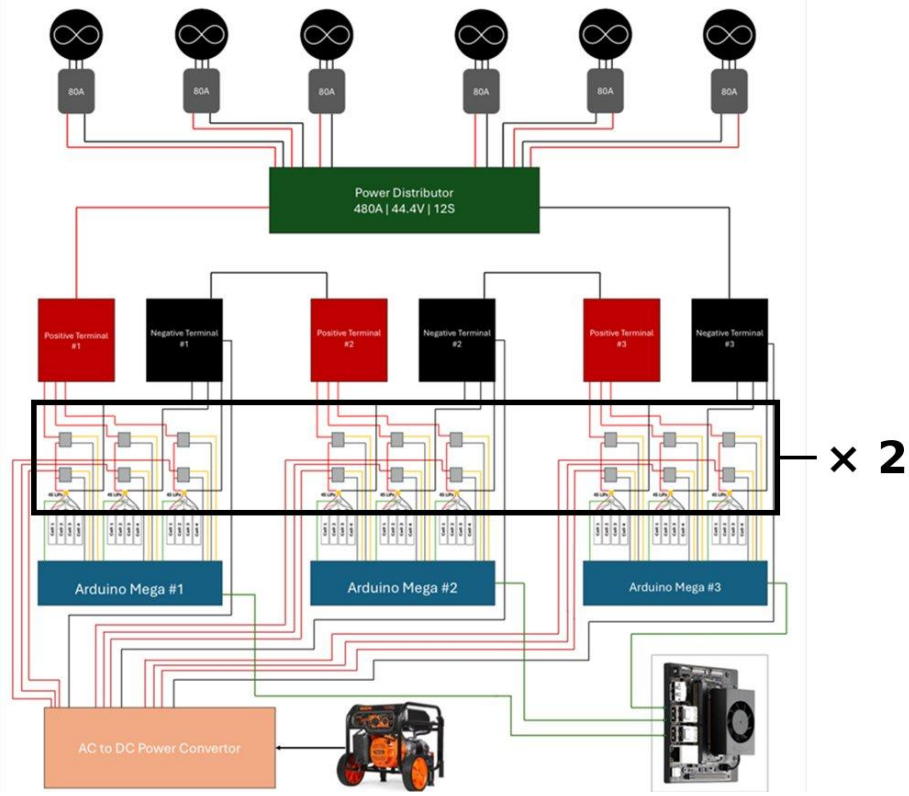

Figure 27. Ground Path Integration Visualization.

This revision increased the total SSR count from 18 (two per battery) to 36 (four per battery). The motor path retains the shared ground rail, a functional requirement of the series junction topology. The charge path provides complete galvanic isolation, enabling simultaneous charging of multiple batteries at different states of charge without cross-coupling.

This discovery also motivated the transition from the EV Peak CQ3 charger to the Keysight E36233A programmable DC power supply. The EV Peak charger was suitable for single-battery testing but its built-in safety protections, designed for consumer LiPo charging, proved incompatible with the custom switching architecture. The Keysight supply provides regulated DC directly at 16.8 V with a current limit of approximately 2 A

per battery, allowing the system to control the charging path entirely through the SSR network without interference from charger-side safety logic.

Figure 28. Keysight E36233A Programmable DC Power Supply.

4.5 Integrated System Architecture

With the voltage monitoring circuit and power switching system individually developed, the final stage was their integration into a single operational system. The integrated system operates on four Arduino Mega 2560 boards connected to the monitoring laptop through a powered USB hub. Three boards handle voltage monitoring (one per junction), each transmitting cell voltage and SoC data over its own serial connection. The fourth board is the SSR controller, driving all 36 relays based on serial commands. The Python dashboard collects data from all three monitoring streams simultaneously and renders the live system display. The SSR controller's state information is communicated to the dashboard through a shared data file, updated at regular intervals, to avoid serial port conflicts.

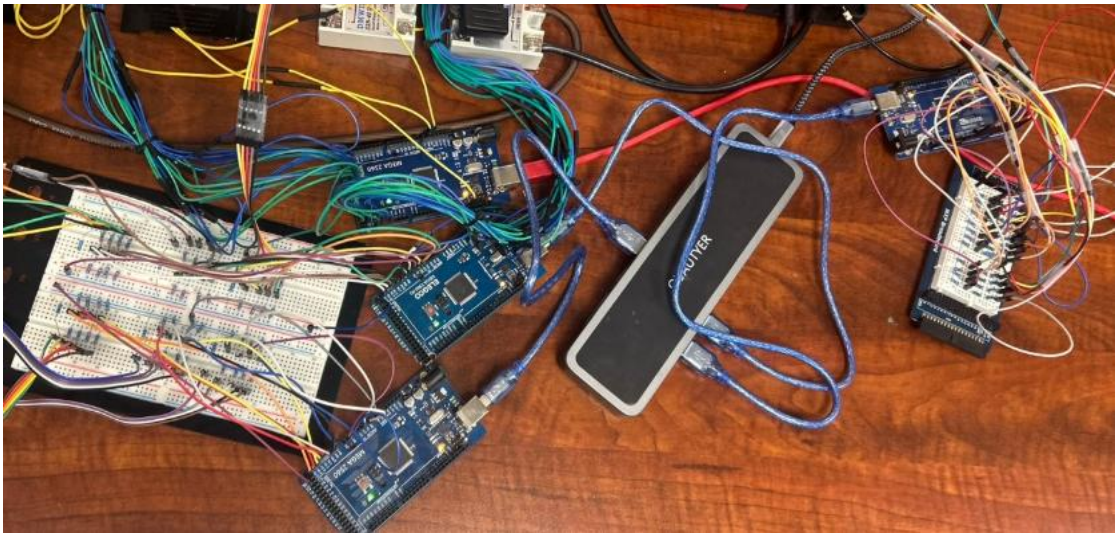

Figure 29. Four Arduino Subsystems Connected via USB.

The data flow follows a defined path: battery balance connectors feed cell voltages into the voltage divider networks; the monitoring firmware isolates individual cell voltages via the subtraction method, maps each cell to a SoC value via the OCV lookup table, and transmits results over serial; the dashboard collects and displays these data. When a state change is initiated, manually or by the scheduling algorithm, the command is transmitted to the SSR controller, which sets the appropriate relay pins with the two-second safety delay enforced between path deactivation and activation. The controller then broadcasts the updated state table, and the dashboard reflects the new configuration.

On the power side, the nine batteries reside in three parallel groups of three, with junction voltages stacking in series to produce 44.4 V nominal at the motor bus. The charge bus operates separately at the 4S level through the Keysight E36233A, providing 16.8 V at approximately 2 A to whichever batteries are in the CHARGE state.

If conditions arise in which all batteries across all junctions approach a critically low SoC despite active charging, a scenario that could occur if the charging source cannot keep pace with motor-side depletion, the system flags a critical warning. In a flight-capable deployment, this warning would trigger an autonomous return-to-home protocol via the Pixhawk flight controller's ArduPilot firmware to land the aircraft before total energy depletion.

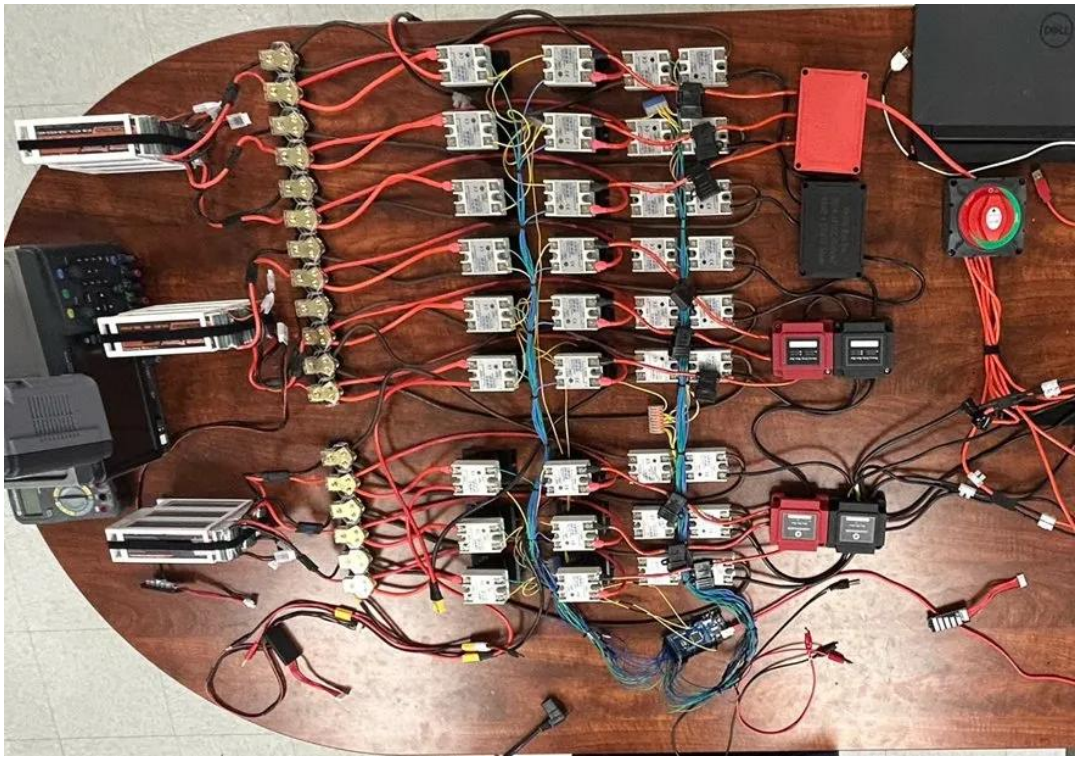

Figure 30. Fully Developed Switching Architecture.

Table 7. System Component Summary.

Component	Qty	Function	Key Specification
Arduino Mega 2560	4	Voltage monitoring (×3), SSR control (×1)	54 digital I/O, 16 analog inputs

SSR-40DD Solid-State Relay	36	Battery path switching (4 per battery)	40 A, 3–32 VDC control, 5–200 VDC load
Youme 4S 6500 mAh 80C LiPo	9	Energy storage (3 per junction)	14.8 V nom., 6.5 Ah, 520 A burst
Voltage Divider Board	3	Cell voltage sensing (12 cells each)	33kΩ / 30kΩ resistor network
Keysight E36233A	1	Programmable charging source	400 W, dual-channel, 30 V / 20 A
USB Hub (Powered)	1	Arduino communication multiplexing	USB 2.0, 4-port
Laptop (Control Station)	1	Dashboard display, BMS interface	Serial communication via USB
Aluminum Heatsinks	36	SSR thermal management	Mounted on each SSR-40DD

Chapter 5: Hardware Validation Testing

This chapter documents the systematic validation of the intelligent power management system through physical bench testing. Section 5.1 describes the test methodology, the four-tier experimental structure, and the apparatus. Sections 5.2 through 5.5 present the procedures, observations, and outcomes for each tier. Section 5.6 discusses the collective results and establishes the hardware-validated foundation for the simulation work in Chapter 6.

5.1 Test Methodology and Apparatus

The hardware validation campaign was structured around a four-tier design of experiment that began with the simplest possible validation and progressively added complexity until the full integrated system was operating under real load conditions with autonomous scheduling. The rationale for this graduated approach was to isolate each subsystem's behavior before combining them, such that failures observed at higher tiers could be attributed to newly introduced complexity rather than latent issues in previously validated components.

Table 8. Hardware Validation Test Tier Summary.

Tier	Focus	Control	Load	Tests
1	SSR + Firmware Validation	Manual	None	5
2	Switching Under Load	Manual	1 Motor	7
3	Charge While Operating	Manual	1 Motor	2
4	Fully Autonomous System	Automated	6 Motors	1

The test bench consisted of nine Youme 4S 6500 mAh 80C LiPo batteries, 36 SSR-40DD solid-state relays with aluminum heatsinks, four Arduino Mega 2560 microcontrollers connected through a powered USB hub, and a Keysight E36233A programmable DC power supply as the charging source. The load for powered tests was provided by the same propulsion hardware specified for the Skywalker III platform: T-Motor P80III KV120 BLDC motors paired with 30-inch T-Motor MF3016 propellers driven by Flycolor Flydragon V4 80 A ESCs. Tier 2 and Tier 3 employed a single motor as the load; Tier 4 employed all six motors simultaneously. Handheld multimeters were used throughout all tiers to cross-check voltage readings.

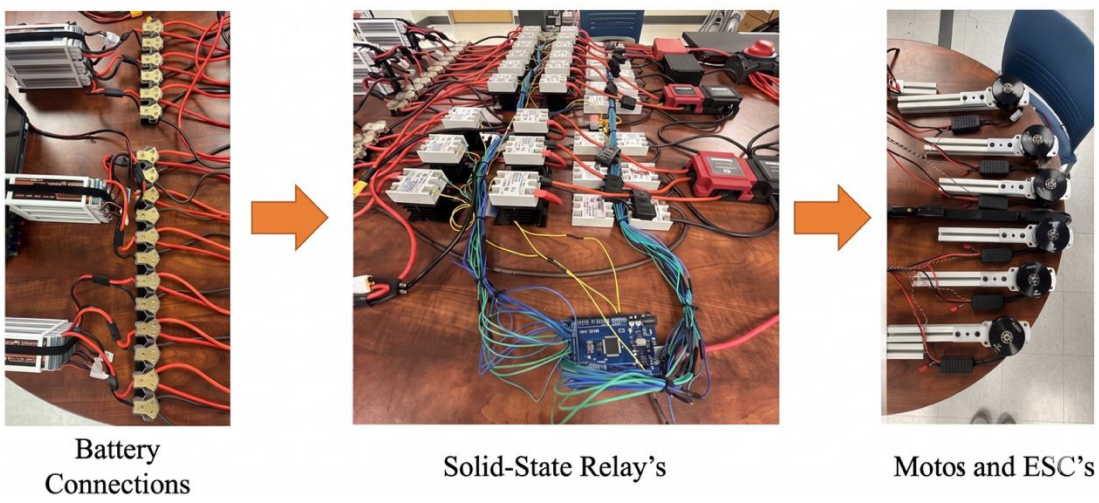

Figure 31. Three-Stage Bench Test Layout: Battery Connections, Solid-State Relay Bank, and Motor/ESC Load.

5.2 Tier 1: SSR Control Confirmation

Tier 1 established that the solid-state relay network and its controlling firmware operated correctly in isolation, with no batteries connected. Five tests were conducted.

Test T1.1 verified individual SSR pair control. Each of the nine batteries was commanded through its three states in sequence (IDLE to MOTOR, MOTOR to IDLE, IDLE to CHARGE, CHARGE to IDLE). At each transition, the corresponding relay indicator LEDs were inspected to confirm that the physical relay state matched the commanded state. All 36 SSRs responded correctly, and the firmware state tracking array displayed on the dashboard matched the physical LED indications after every transition.

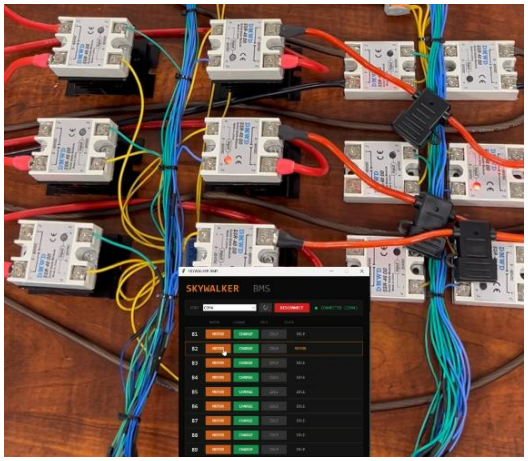

Figure 32. SSR Motor Path Validation.

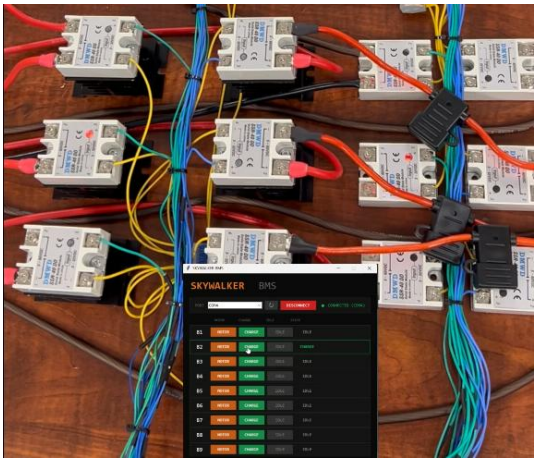

Figure 33. SSR Charge Path Validation.

Test T1.2 verified the global ALL MOTOR command. The command "AA" was issued, and all 18 motor-path SSRs activated simultaneously while all 18 charge-path SSRs remained deactivated. The dashboard state display was verified against the physical relay LEDs across all nine batteries.

Test T1.3 verified the emergency KILL command. With all batteries in the MOTOR state, the kill command deactivated all 36 SSRs simultaneously. The transition occurred within the perceptible response time of the relay LEDs, confirming that the kill command overrides all states and executes without the two-second guarded delay. This behavior is by design: emergency shutdown prioritizes rapid de-energization over sequenced transition.

Test T1.4 verified path conflict prevention. For each of the nine batteries, the motor-path SSRs were activated, and a charge-path activation command was issued without first deactivating the motor path. In every case the firmware rejected the conflicting command and maintained the existing motor-path state. This test confirmed the mutual exclusion logic: the firmware prevents any command sequence that would simultaneously connect a battery to both buses.

Test T1.5 verified the two-second safety delay. A battery was commanded from MOTOR to CHARGE, and the transition was timed using a stopwatch. The motor-path SSRs deactivated first, followed by the measured pause, after which the charge-path SSRs activated. The observed delay was consistent with the two-second firmware specification, verified in both MOTOR-to-CHARGE and CHARGE-to-MOTOR directions. All five Tier 1 tests passed.

Table 9. Tier 1 Test Summary.

Test ID	Description	Expected Result	Observed Result	Status
T1.1	Individual SSR pair control	Correct LED state per command	All 36 SSRs responded correctly	Pass
T1.2	Full-system MOTOR command	All SSRs to MOTOR state	All SSRs activated to MOTOR	Pass
T1.3	Emergency KILL command	All SSRs deactivate immediately	All 36 SSRs deactivated	Pass
T1.4	Conflict prevention	Reject simultaneous MOTOR+CHARGE	Command rejected by firmware	Pass
T1.5	Transition delay verification	2-second delay between states	Delay consistent both directions	Pass

5.3 Tier 2: Switching Under Load

Tier 2 introduced real voltage and mechanical load through a single T-Motor P80III motor with a Flycolor Flydragon V4 ESC and 30-inch propeller. Throttle was set to 25 percent for all Tier 2 tests. Seven tests were conducted, progressively scaling battery count and exercising selective disconnection, with the emergency kill command exercised throughout as a continuous safety verification.

Figure 34. Initial Single Motor Test Bench.

Tests T2.1 through T2.3 established baseline motor operation at each battery count. In T2.1, three batteries (one per junction) formed the minimum series chain; the motor operated without interruption. In T2.2, six batteries (two per junction) were connected; the motor operated normally. In T2.3, all nine batteries (three per junction) were connected with the full parallel depth available.

Table 10. System Voltage Measurements for Tier 2 Baseline Tests.

Test	Active Batteries (Total)	Active Batteries per Junction	Total System Voltage (V)	Within 12S Range?
T2.1	3	1	45.848	✓

Test	Active Batteries (Total)	Active Batteries per Junction	Total System Voltage (V)	Within 12S Range?
T2.2	6	2	46.354	✓
T2.3	9	3	46.468	✓

Tests T2.4 and T2.5 exercised selective disconnection at the six-battery configuration. In T2.4, while the motor was spinning on six batteries, a single battery at one junction was transitioned from MOTOR to IDLE. The remaining battery at that junction continued alone, and the motor continued without measurable disruption. In T2.5, one battery was removed from each of the three junctions in sequence, reducing the system from six to three active batteries. The motor continued without interruption at each step.

Tests T2.6 and T2.7 repeated selective disconnection at full nine-battery capacity. In T2.6, one battery was removed from each junction while the motor was spinning (nine to six active). In T2.7, a second battery was removed from each junction, leaving the minimum configuration of one active battery per junction. The motor continued to operate, confirming that the junction constraint is a physically validated operating boundary, not merely a theoretical safety limit. Throughout all Tier 2 tests, the kill command was exercised repeatedly as a safety verification. Each time, the motor decelerated to a stop within approximately one second as all SSRs deactivated.

All seven Tier 2 tests passed. The parallel-first architecture permitted individual battery disconnection without load interruption at every tested scale.

Table 11. Tier 2 Test Summary.

Test ID	Description	Expected Result	Observed Result	Status
T2.1	3-battery operation (1/jctn.)	Motor runs without interruption	Motor operated continuously	Pass
T2.2	6-battery operation (2/jctn.)	Motor runs without interruption	Motor operated normally	Pass
T2.3	9-battery operation (3/jctn.)	Motor runs without interruption	Motor operated normally	Pass
T2.4	Selective disconnect (9 to 6)	Motor continues, no disruption	Seamless transition	Pass
T2.5	Selective disconnect (6 to 3)	Motor continues, no disruption	Seamless transition	Pass
T2.6	Reconnection (3 to 6 to 9)	Motor continues, no disruption	Seamless reconnection	Pass
T2.7	Kill command under load	Motor stops within ~1 second	Motor decelerated and stopped	Pass
KILL	Kill command exercised throughout all Tier 2 Tests	Motor stops	Motor decelerated and stopped	Pass

5.4 Tier 3: Simultaneous Charge and Discharge

Tier 3 validated the core functional premise of the entire system: that a battery can be removed from the motor bus, connected to the charge bus, and charged while the remaining batteries continue to power the motor load without interruption. If this capability does not function in physical hardware, the scheduling algorithms in Chapter 6

have no implementable foundation. Tier 3 therefore represents the most consequential test in the validation campaign.

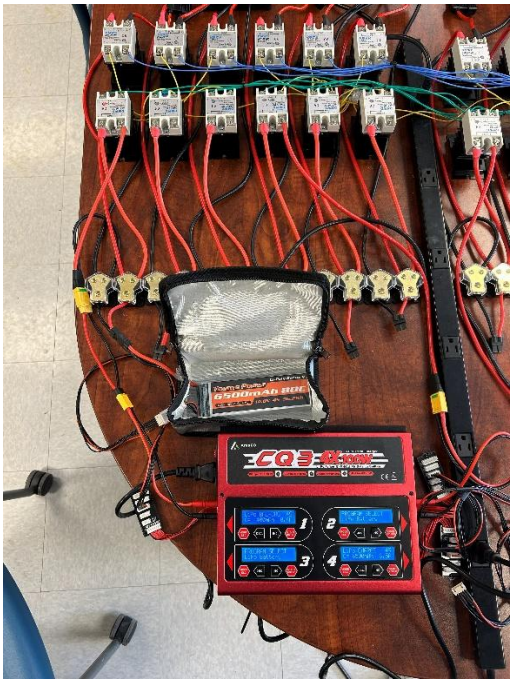

Figure 35. Battery Charging Capability Validation.

The test began with all nine batteries in the MOTOR state and the motor spinning at 25 percent throttle. One battery was manually transitioned from MOTOR to CHARGE. Throughout this transition, the motor continued spinning with no perceptible disruption. The initial charging source was the EV Peak CQ3, which recognized the single battery and initiated its charging cycle.

A second battery was then transitioned to the charge bus. The EV Peak charger immediately reported a reverse polarity error, the shared-ground conflict documented in Section 4.4.4. Following the ground-side isolation revision (four-SSR-per-battery architecture) and the transition to the Keysight E36233A, the multi-battery charging test

was repeated successfully. A multimeter connected in series with the charge bus confirmed approximately 2 A flowing into the charging battery. The motor continued without interruption.

Both Tier 3 tests passed. The validated capability, charging one or more batteries while the remaining batteries power the motor load, is the physical foundation for every scheduling strategy evaluated in Chapter 6.

5.5 Tier 4: Algorithm-Driven Autonomous Switching

Tier 4 represented the integration milestone of the hardware validation campaign: the scheduling algorithm made switching decisions autonomously based on real-time voltage and SoC data, with no manual intervention. All six T-Motor P80III motors were connected as the load.

Figure 36. Tier 4 Full System Test Bench.

The test was configured with all nine batteries in the MOTOR state and all six motors spinning. The Keysight E36233A was connected to the charge bus at 16.8 V; its

display read 0 A, confirming no current path existed while all charge-path SSRs remained open.

As the motors drew current and battery voltages declined, the algorithm detected the first battery whose pack voltage fell below 15 V (corresponding to approximately 3.75 V per cell). The algorithm autonomously issued a state transition command: the motor-path SSRs deactivated, the two-second guarded delay was observed, and the charge-path SSRs activated.

The transition was confirmed through three independent indicators. First, the dashboard updated the battery's state from MOTOR to CHARGE with continued voltage readings. Second, the SSR indicator LEDs reflected the new configuration. Third, the Keysight display increased from 0 A to approximately 2 A at the instant the charge-path SSRs closed, confirming current delivery through the newly activated charge path. Throughout this sequence, all six motors continued spinning without interruption, audible disruption, or measurable voltage instability.

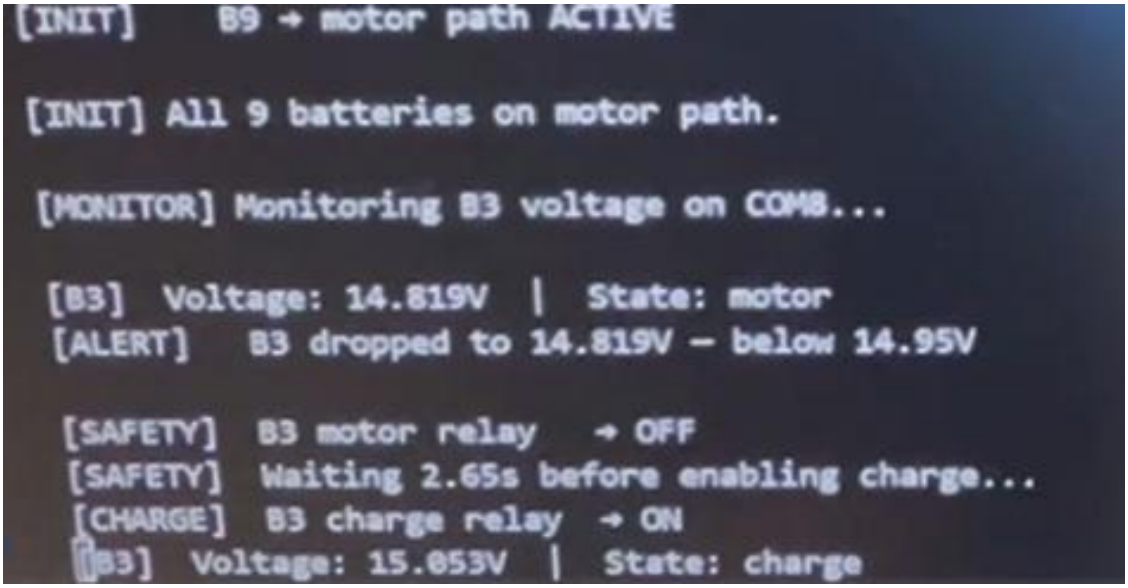

Figure 37. Tier 4 Test Terminal View.

Figure 38. Digital Multimeter Verifying Charge Current Delivered by the E36233A Power Supply.

The Tier 4 test demonstrated a single closed-loop detection-and-transition event on real hardware, confirming that the monitoring, scheduling, and actuation subsystems

function together end-to-end. Extended-duration autonomous operation, in which the algorithm manages multiple batteries across multiple charge-discharge cycles over an extended mission, was evaluated through the simulation presented in Chapter 6. The simulation establishes that the scheduling policies deliver their intended performance advantages at operational scale; multi-cycle hardware validation is identified as future work in Section 8.3.

5.6 Discussion of Results

The four-tier validation campaign comprised fifteen individual tests spanning SSR hardware verification, firmware logic confirmation, loaded operation at multiple scales, simultaneous charge and discharge, ground-side isolation resolution, and algorithm-driven autonomous switching. Every test passed.

The results confirmed the foundational claims required by the simulation work in Chapter 6: the series junction topology sustains motor operation under real current draw at every tested configuration; the parallel-first architecture permits individual battery disconnection without load interruption; adding parallel batteries within a junction reduces per-battery current share as expected from the parallel current-division model; and the minimum configuration of one battery per junction is a physically validated boundary.

A multimeter cross-check of one battery per junction quantified the measurement accuracy. Across the 12 sampled cells, the mean absolute error was 121 mV (3.08 percent), with a maximum of 187 mV (5.09 percent). Errors were bidirectional,

indicating random ADC noise and resistor tolerance rather than systematic bias. This accuracy is well within the 5 to 10 percent SoC bands used by the scheduling algorithm.

Table 12. Voltage Monitoring Circuit Validation.

Battery	Cell	Multimeter (V)	Arduino (V)	Absolute Error (V)	Percent Error (%)
1	1	3.962	3.781	0.181	4.57
1	2	4.187	4.014	0.173	4.13
1	3	3.942	4.158	0.216	5.48
1	4	4.168	3.954	0.214	5.13
2	1	4.034	3.832	0.202	5.01
2	2	3.787	3.964	0.177	4.67
2	3	4.082	3.884	0.198	4.85
2	4	3.814	3.992	0.178	4.67
3	1	3.761	3.954	0.193	5.13
3	2	3.975	3.802	0.173	4.35
3	3	3.672	3.879	0.207	5.64
3	4	3.938	3.762	0.176	4.47

Battery	Cell	Multimeter (V)	Arduino (V)	Absolute Error (V)	Percent Error (%)
Mean	—	—	—	**0.191**	**4.48**
Max	—	—	—	**0.216**	**5.64**
Min	—	—	—	**0.173**	**4.13**

The Tier 3 validation of simultaneous charge and discharge confirmed the core premise upon which all scheduling strategies depend. The Tier 3 tests also exposed the ground-side isolation issue, whose resolution, doubling the SSR count from 18 to 36 and substituting the Keysight E36233A for the EV Peak CQ3, is documented as a design iteration that constitutes one of the significant engineering contributions of the system design process.

The Tier 4 result established the closed-loop integration of monitoring, scheduling, and actuation. The algorithm's autonomous detection of a low-voltage battery and subsequent transition, confirmed by the Keysight current output, demonstrated that the three subsystems function together as a unified system. This integration is the prerequisite for the simulation work in Chapter 6, where the scheduling layer is evaluated at 48-battery scale and over 90-minute mission durations.

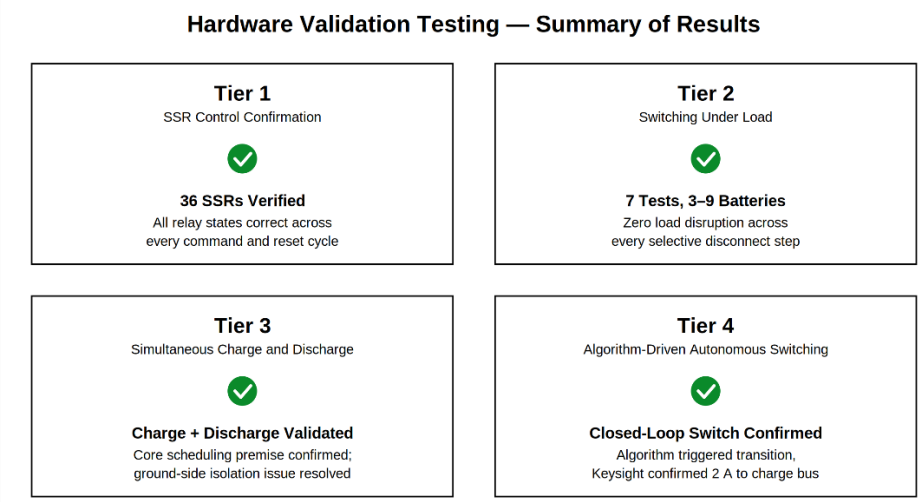

Figure 39. Summary of Testing Results.

Chapter 6: Simulation and Scheduling Analysis

This chapter presents the computational simulation developed to characterize the intelligent power management system at operational scale. Section 6.1 establishes the simulation objectives and the two system configurations modeled. Section 6.2 documents the simulation framework. Section 6.3 describes the five scheduling strategies evaluated. Section 6.4 presents the theoretical basis for rolling replenishment. Section 6.5 reports simulation results. Section 6.6 addresses failure handling and model limitations.

6.1 Simulation Objectives and Configurations

The hardware validation in Chapter 5 demonstrated that the three-state switching architecture, cell-level monitoring, and conflict prevention logic operate correctly on the nine-battery bench prototype. The full Skywalker III system employs 48 batteries across three junctions of sixteen each. At this scale, the implementation would require 192 solid-state relays, 192 voltage dividers, and the corresponding microcontroller channels. Constructing a complete 48-battery bench exceeded the scope and resource constraints of this thesis. The nine-battery prototype preserves every structural feature of the full system: three junctions, intra-junction parallelism, inter-junction series stacking, and multiple batteries per junction sufficient to exercise selective disconnection.

A computational simulation was developed to answer questions that fall outside the reach of hardware testing: which scheduling strategy maximizes energy recovery over an extended mission, how many switching events each strategy requires, whether

proactive scheduling outperforms reactive scheduling, and whether the series-chain constraint can be maintained indefinitely under realistic throttle variation and generator fuel consumption.

Two system configurations are modeled. The prototype configuration mirrors the bench hardware (nine batteries, three junctions, Keysight E36233A). The full-system configuration represents the Skywalker III as designed (48 batteries, three junctions of sixteen, generator-fed charger). Both share identical physics, discharge and charge models, scheduling algorithm implementations, and voltage modeling logic. The parameters distinguishing them are summarized in Table 13.

Table 13. Simulation System Configuration Comparison.

Parameter	Prototype	Full System
Total Batteries	9	48
Junctions	3	3
Batteries per Junction	3	16
Minimum Active per Junction	1	1
Maximum Charging per Junction	2	15
Charging Source	Keysight E36233A	Generator-fed Charger
Raw Charger Power	400 W	4,000 W
Charger Efficiency	98%	85%
Effective Charger Power	392 W	3,400 W
Default Simulation Duration	60 min	90 min

Total SSRs Required	36	192

The simulation is implemented in Python with a fixed timestep of 0.5 seconds. Over a 90-minute simulated mission, the execution loop runs 10,800 iterations. The timestep was selected to balance integration accuracy against execution time; finer timesteps produced marginal changes in cumulative energy totals of under two percent, while coarser timesteps degraded scheduling logic fidelity. It should be noted that Rolling Replenishment was developed and evaluated entirely in simulation; deployment of the algorithm on the bench prototype is identified as the most immediate near-term validation step in Section 8.3 The complete simulation source code is provided in Appendix C.

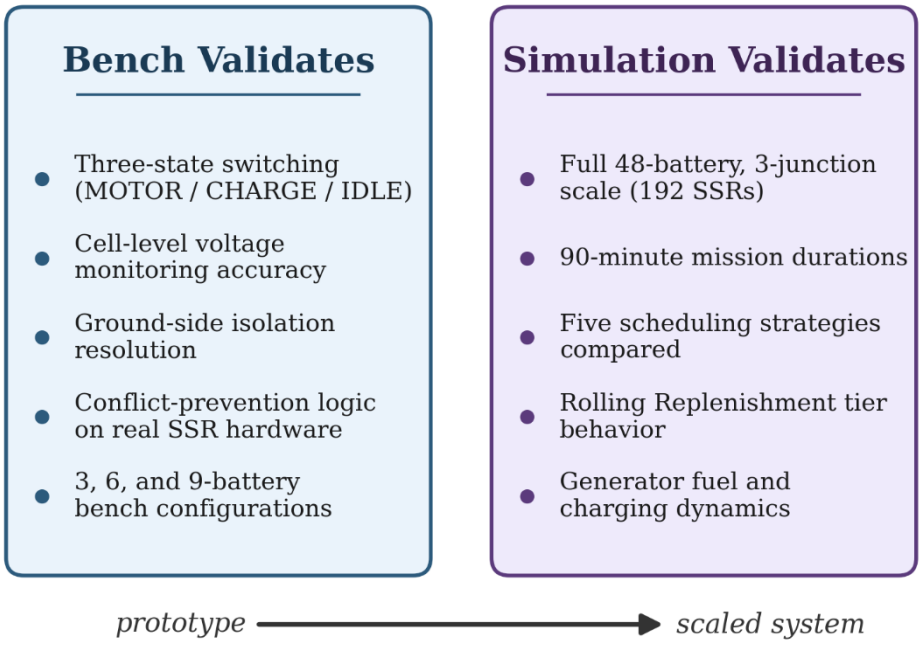

Figure 40. Bench vs. Simulation Validation Scope.

6.2 Simulation Framework

The framework consists of five tightly coupled components: a timestep execution loop, a throttle profile driving power demand, a discharge model distributing current across active batteries, a charge model implementing CC/CV charging, and a voltage model mapping SoC to cell terminal voltage.

6.2.1 Model Architecture and Execution Sequence

The simulation advances in fixed 0.5-second timesteps. At each timestep, a deterministic sequence of operations captures the full system state: the mission profile is queried for the current throttle setting; throttle maps to power demand; power reductions for degraded states are applied; total system current is computed; current is distributed across motor-bus batteries; each battery's stored energy is decremented; charging current is updated per the CC/CV profile; each charging battery's energy is incremented; generator fuel is consumed (full-system configuration only); the scheduling algorithm evaluates transitions; SSR state changes are executed subject to constraints; the system state is recorded; and termination conditions are evaluated.

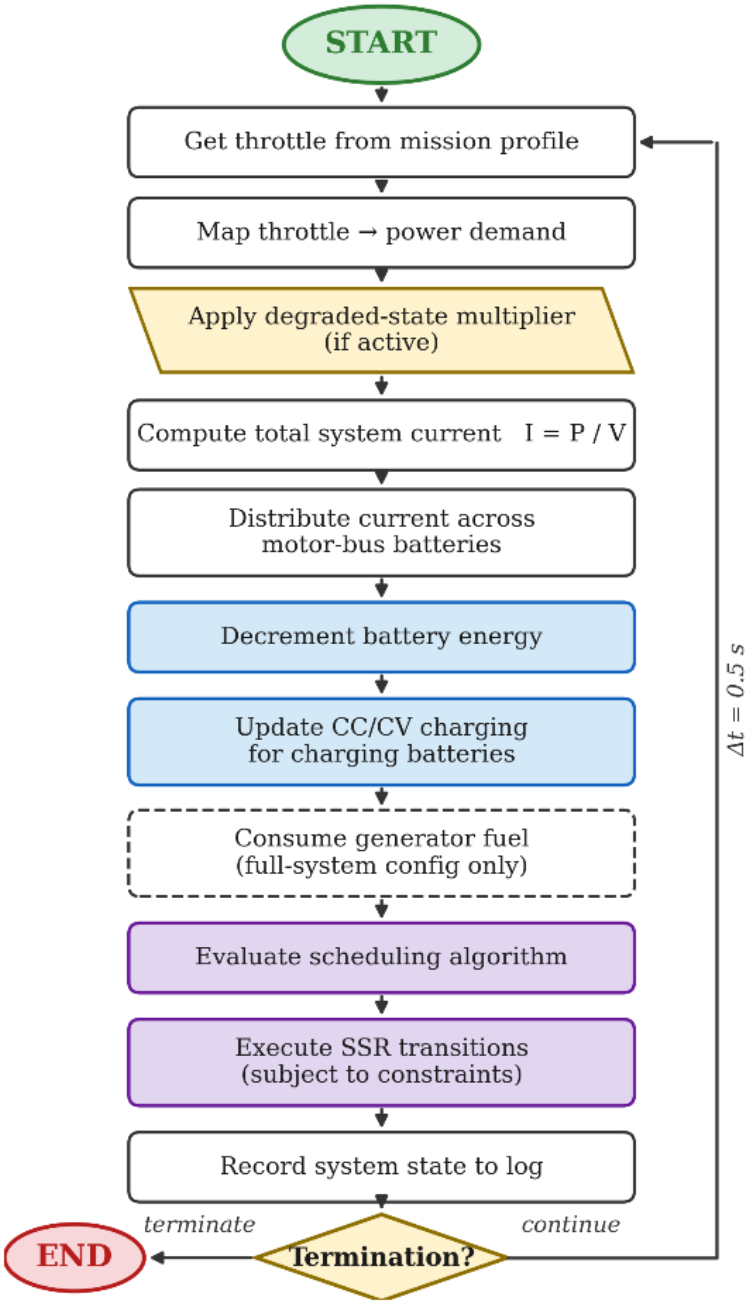

Figure 41. Simulation Loop Execution Sequence.

The simulation terminates under three conditions: configured maximum duration (successful mission completion), all-batteries-critical (every battery at or below 15

percent SoC with insufficient charging to recover), or emergency-landing (maintaining the series chain would require violating the minimum-active-per-junction constraint).

6.2.2 Throttle Profile and Power Demand

The throttle profile models a multirotor mission with three operating phases: an initial takeoff burst at 80 percent throttle for 30 seconds, nominal cruise at 30 percent throttle for the remainder, and periodic maneuver excursions at 50 percent throttle lasting approximately 7.5 seconds at intervals of approximately 300 seconds. The maneuver interval is randomized within ±30 percent of the 300-second nominal, and the maneuver duration is randomized within ±50 percent of the 7.5-second nominal, preventing scheduling strategies from exploiting a deterministic schedule.

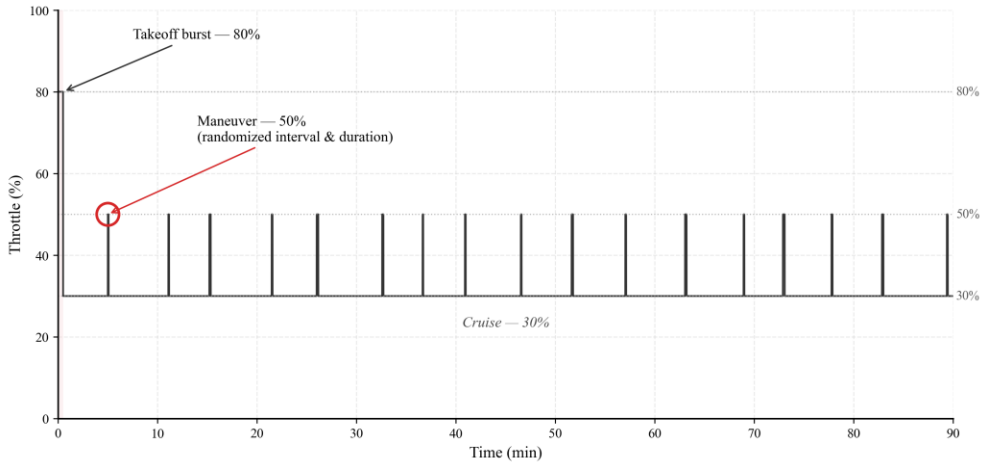

Figure 42. 90-Minute Mission Throttle Profile.

Table 14. Throttle-to-Power Mapping.

Throttle (%)	Power Demand (W)
0	0

25	2,500
40	5,000
45	8,500
50	10,000
70	15,000
80	17,000
100	21,312

The throttle maps to power demand through a piecewise-linear lookup table. At the 30 percent cruise throttle, the interpolated power draw is 3,500 W (approximately 78 A at nominal voltage). The peak of 21,312 W at full throttle matches the aircraft's rated maximum load of 480 A at 44.4 V. Power demand may also be reduced during degraded system states: 0.3 multiplier for emergency landing, 0.7 for critical state.

6.2.3 Discharge Model

At each timestep, total system current is computed as:

$$I_{sys} = \frac{P_{demand}}{V_{sys}}$$

where I_{sys} is the total system current (A), P_{demand} is the throttle-mapped power demand (W), and V_{sys} is the instantaneous total system voltage (V). Within each junction, per-battery current is:

$$I_{bat} = \frac{I_{sys}}{N_{active}}$$

where I_{bat} is the current through a single active battery (A) and N_{active} is the number of batteries on the motor bus within that junction. The model assumes ideal current sharing among parallel batteries, a reasonable first-order approximation for closely matched cells, though it understates modest current concentration on the highest-voltage pack.

Energy delivered by each motor-bus battery per timestep is:

$$E_{out} = \frac{I_{bat} \times V_{bat} \times \Delta t}{3600}$$

where E_{out} is energy delivered (Wh), V_{bat} is instantaneous battery terminal voltage (V), and Δt is the 0.5-second timestep. State of charge is recomputed as:

$$SoC = (\frac{E_{rem}}{E_{cap}}) \times 100$$

where SoC is state of charge (%), E_{rem} is remaining stored energy (Wh), and E_{cap} is rated battery capacity of 96.2 Wh (6.5 Ah × 14.8 V nominal).

6.2.4 Charge Model

The charge model implements CC/CV charging. During the CC phase, per-battery current is:

$$I_{charge} = \min (I_{1C}, \frac{I_{max}}{N_{charging}})$$

where I_{charge} is per-battery charging current (A), I_{1C} is the 1C rate of 6.5 A, I_{max} is the maximum current the charger can deliver at bus voltage (A), and $N_{charging}$ is the

number of batteries on the charge bus. Upon battery terminal voltage reaching the CV threshold of 16.6 V (4.15 V per cell), the charger transitions to constant-voltage mode with a taper fraction:

$$f_{taper} = \max\left(0.05, \frac{V_{max} - V_{bat}}{V_{MAX} - V_{CV}}\right)$$

where f_{taper} is the dimensionless taper fraction, V_{max} is 16.8 V, V_{bat} is instantaneous terminal voltage (V), and V_{CV} is the CV threshold of 16.6 V. The 0.05 lower bound ensures minimum current continues flowing to prevent stalled charging.

Table 15. Charger Source Specifications.

Specification	Keysight E36233A (Prototype)	Generator-fed (Full System)
Charge Source Type	Programmable DC Supply	Generator + AC-DC Charger
Raw Power Output	400 W	4,000 W
Conversion Efficiency	98%	85%
Effective Power	392 W	3,400 W
Max Charge Voltage	16.8 V	16.8 V
CV Threshold Voltage	16.6 V	16.6 V
Max Current at 16.8 V	23.8 A	~202 A
1C Rate per Battery	6.5 A	6.5 A
Max Charging/Junction	2	15

Energy added per charging battery per timestep is:

$$E_{in} = \frac{I_{charge} \times V_{bat} \times \Delta t}{3600}$$

where E_{in} is energy added (Wh) and all other variables are as previously defined. Battery energy is capped at rated capacity.

In the full-system configuration, generator fuel is consumed in proportion to delivered charging power, drawn from a reserve of 39,200 Wh representing usable energy from the 16.5 L tank at 25 percent efficiency. When fuel is exhausted, the charger becomes permanently unavailable.

6.2.5 Voltage Modeling and OCV-SoC Lookup

The simulation employs a finer-grained 22-entry lookup compared to the 11-entry table used in the bench firmware (Section 4.3.3, Table 5), reflecting the additional computational headroom available in the Python implementation.

Table 16. OCV-SoC Lookup Table for LiPo Cell (Simulation Model).

SoC (%)	Cell Voltage (V)	SoC (%)	Cell Voltage (V)
100	4.2	45	3.75
95	4.15	40	3.72
90	4.1	35	3.7
85	4.05	30	3.68
80	4	25	3.65
75	3.95	20	3.6
70	3.9	15	3.55
65	3.85	10	3.5
60	3.83	5	3.4

55	3.8	2	3.3
50	3.78	0	3

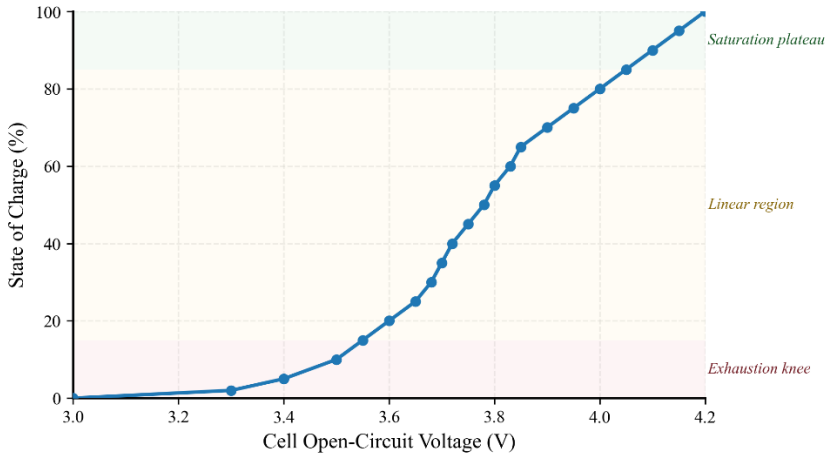

Figure 43. OCV–SoC Lookup Curve

To introduce realistic cell-to-cell variation, each cell is assigned a fixed random offset drawn from a uniform distribution on [−0.02 V, +0.02 V] at initialization, representing manufacturing tolerances and electrochemical differences between nominally identical cells. Pack voltage is the sum of four series-connected cell voltages; junction voltage is the arithmetic mean of active pack voltages; and system voltage is the sum of the three junction voltages:

$$V_{pack} = V_{cell1} + V_{cell2} + V_{cell3} + V_{cell4}$$

$$V_{junction} = \frac{\Sigma V_{pack\_active}}{N_{active}}$$

$$V_{sys} = V_{junction1} + V_{junction2} + V_{junction3}$$

6.3 Scheduling Strategies

Five scheduling strategies are evaluated under identical conditions. Four are reactive, evaluating switching decisions when individual batteries cross a fixed threshold. The fifth, rolling replenishment, is proactive and constitutes the principal algorithmic contribution of this thesis.

6.3.1 Shared Constraints and State Transition Mechanics

All strategies are always subject to the junction constraint (at least one battery per junction on the motor bus), path-conflict prevention (no battery simultaneously on both buses), and a minimum dwell time between successive transitions on the same battery (30 seconds for reactive strategies, 60 seconds for rolling replenishment). Shared parameters are summarized in Table 17.

Table 17. Shared Reactive Strategy Parameters.

Parameter	Value
Switch trigger threshold	Battery SoC ≤ 30%
Charge-complete threshold	Battery SoC ≥ 90%
Critical threshold	Battery SoC ≤ 15%
Minimum active batteries per junction	1
Maximum charging per junction (prototype)	2
Maximum charging per junction (full system)	15
Minimum dwell time per battery	30 s
Maximum switches per junction per cycle	1

6.3.2 Reactive Strategies

The four reactive strategies share the threshold values and constraints, previously shown in Table 17. They differ only in the selection rule applied when the switching threshold is crossed:

Greedy selects the battery with the lowest SoC among all motor-bus batteries in any junction that has crossed the 30 percent threshold. Ties are broken by cumulative energy discharged. The battery remains on the charge bus until it reaches 90 percent SoC.

Round Robin maintains an ordered list and selects the next battery in sequence whose SoC has fallen at or below the threshold. Completed batteries move to the back of the list, producing fair distribution of charging access over time.

Balanced applies the greedy approach with an additional filter: candidate batteries must have SoC below both the 30 percent threshold and the current system-wide average SoC. This targets batteries dragging the system average downward, producing a tighter SoC distribution.

Minimum Switches enforces a strict rule that only one battery system-wide may be on the charge bus at any given time. After that battery completes charging, the lowest-SoC battery below threshold is transitioned next. This minimizes total switching events at the cost of slower energy recovery.

6.3.3 Rolling Replenishment

Rolling replenishment differs from the reactive strategies in three fundamental respects. First, it evaluates each junction as an aggregate through the median SoC of its active batteries, rather than inspecting individual batteries against a fixed threshold. Second, it initiates charging transitions before any individual battery reaches a critical state. Third, it maintains a continuous swap mechanism that cycles batteries between the charge and motor buses based on SoC differential.

The junction health assessment computes the median SoC of all motor-bus batteries for each junction. The median is preferred over the arithmetic mean because it is robust to a single outlier. Each junction's median maps to one of four tiers:

Table 18. Rolling Replenishment Tier Definitions and Target Charging Counts.

Tier	Junction Median SoC	Prototype Target	Full System Target
0	> 80%	0 batteries	0 batteries
1	60% to 80%	1 battery	2 batteries
2	40% to 60%	1 battery	5 batteries
3	< 40%	2 batteries	10 batteries

The selection logic operates in two phases at each decision cycle. The first phase fills each junction's charge bus to its tier target by selecting motor-bus batteries in ascending SoC order. The second phase executes a continuous swap: if the highest-SoC charging battery exceeds the lowest-SoC motor-bus battery by at least 30 percentage points (both within the same junction and past their dwell windows), the two swap positions. The net count of active and charging batteries is unchanged, so the junction constraint is never violated; the effect is to continuously cycle the most-replenished

battery back into service and direct the charger toward batteries with the greatest remaining capacity to accept charge.

The swap mechanism is the principal source of the strategy's energy recovery advantage, because it ensures the charging system always operates on batteries in the high-efficiency CC phase rather than holding nearly full batteries in the low-current CV taper.

Table 19. Reactive Strategies vs. Rolling Replenishment Parameter Comparison.

Parameter	Reactive Strategies	Rolling Replenishment
Trigger condition	Individual SoC ≤ 30%	Junction median crosses tier
Decision scope	Individual battery	Junction-level aggregate
Charge-complete threshold	90% SoC	85% SoC
Minimum dwell time	30 s	60 s
Proactive swap threshold	None	30% SoC differential
Behavior classification	Reactive	Proactive

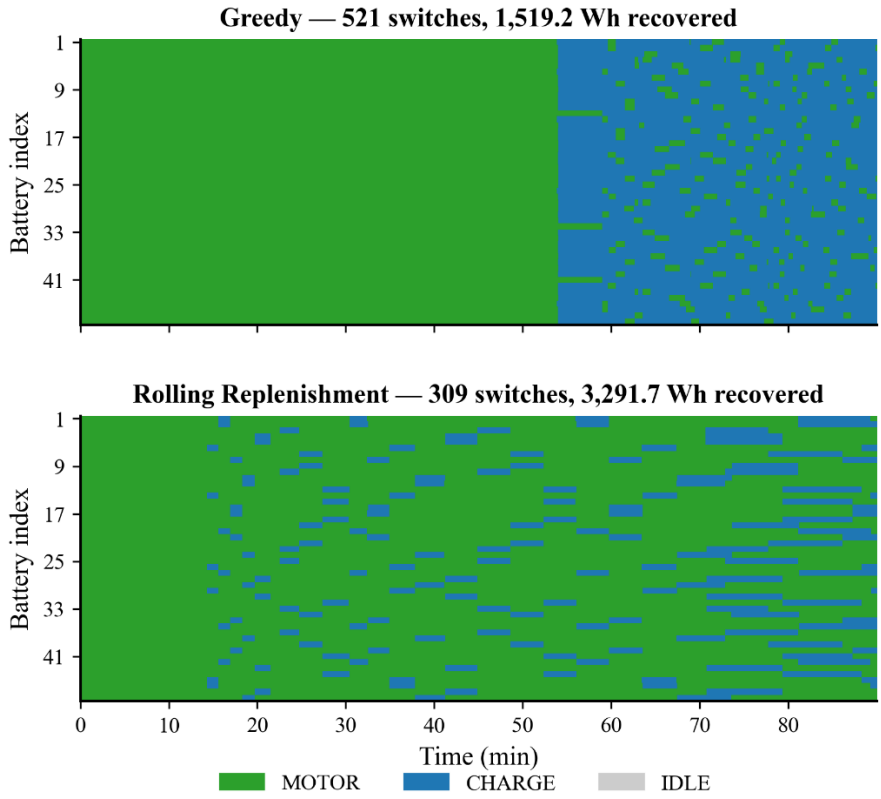

Figure 44. Per-Battery State Timeline: Greedy vs. Rolling Replenishment.

6.4 Theoretical Basis for Rolling Replenishment

Although developed specifically for this thesis, each element of rolling replenishment corresponds to an established concept from classical control theory or information theory. Framing the algorithm in these terms justifies its structure as an engineered application of established principles rather than a heuristic assembly of ad hoc rules.

Gain Scheduling, the tier structure varies controller aggressiveness in response to the operating regime (Qu et al., 2023). The condition variable is junction median SoC; the controller gain is the number of batteries committed to the charge bus. At Tier 0, gain

is zero. As the median crosses successively lower thresholds, gain increases in discrete steps, gentle when conditions permit, aggressive when they demand.

Hysteresis prevents oscillation around a threshold by defining distinct entry and exit conditions separated by a deadband. The tier entry thresholds (80%, 60%, 40% median SoC) are separated from the charge-complete exit threshold (85% individual SoC) by deadbands of up to 45 percentage points, supplemented by the 60-second temporal dwell time.

Lyapunov Drift Minimization. In stochastic network optimization, Lyapunov drift minimization reduces the squared deviation of system state variables from equilibrium at each decision opportunity (Alilou et al., 2025). The swap mechanism reduces SoC variance continuously by exchanging the highest-SoC charging battery with the lowest-SoC motor-bus battery whenever a 30-point differential emerges, the action that minimizes drift at that decision cycle.

The water-filling algorithm directs resources to channels with greatest headroom (Noor Shahida et al., 2020). Rolling replenishment directs charging capacity to the lowest-SoC batteries, which have the greatest remaining headroom to accept charge at the full CC rate, maximizing energy recovery throughput.

Table 20. Theoretical Foundations for Rolling Replenishment Mechanisms .

Theoretical Concept	**Classical Definition**	**Rolling Replenishment Implementation**

Gain Scheduling	Controller aggressiveness varies with operating regime (Åström & Wittenmark, 1995)	Tier structure scales charging commitment (0–10 batteries) based on junction median SoC
Hysteresis Control	Distinct entry/exit conditions separated by deadband prevent oscillation	Tier entry (80/60/40%) separated from exit (85%) by up to 45-point deadband; 60 s dwell time
Lyapunov Drift Min.	Minimize squared deviation from equilibrium at each decision (Neely, 2010)	Swap exchanges highest-SoC charging battery with lowest-SoC motor battery at ≥30-pt differential
Water-Filling	Direct resources to channels with greatest headroom (Cover & Thomas, 2006)	Charging directed to lowest-SoC batteries with greatest CC-phase headroom

6.5 Simulation Results

All five strategies were evaluated under identical conditions on both configurations. Every strategy completed its full duration without a single junction violation.

6.5.1 Full System Results (48 Batteries, 90-Minute Run)

Table 21. Full System Simulation Results (48-Battery, 90-Minute Mission).

Strategy	Switches	Avg SoC (%)	Discharged (Wh)	Recharged (Wh)	Net Drawdown (Wh)
Greedy	746	50.98	4,739.9	1,505.6	3,234.3
Round Robin	749	50.98	4,740.9	1,505.6	3,235.3

Balanced	739	50.98	4,739.9	1,505.6	3,234.3
Min Switches	115	48.15	4,953.1	1,331.2	3,621.9
Rolling Replenishment	**312**	**70.85**	**5,331.5**	**3,292.0**	**2,055.7**

Net drawdown represents total energy consumed minus energy recovered by the charger. Lower values indicate more effective energy recovery. Three of the four reactive strategies, Greedy, Round Robin, and Balanced, produced nearly identical results across every metric, with switch counts within a range of ten events and discharged/recharged totals matching to within 1.0 Wh. This convergence is a structural outcome of the shared reactive framework: all three wait for the same 30 percent threshold and charge to the same 90 percent threshold. The practical implication is that the choice among these three matters less than the choice between any reactive strategy and a proactive one.

Minimum Switches achieved its design objective with only 115 transitions, but at a cost: it recovered only 1,331.2 Wh (11.6 percent less than the other reactive strategies) and left the battery bank with 387.6 Wh less remaining energy.

Rolling Replenishment produced the strongest results across every metric measuring system health and energy recovery. It recovered 3,292.0 Wh, more than double the 1,505.6 Wh recovered by the best reactive strategy, a 118.6 percent improvement. Its net drawdown of 2,055.7 Wh was 1,178.6 Wh lower than the Greedy baseline, meaning the battery bank retained 1,178.6 Wh more stored energy after identical motor work. This

additional energy represents approximately 12.3 percent of total bank capacity (4,617.6 Wh) and would translate directly into extended operational time.

Rolling Replenishment simultaneously achieved this advantage while executing only 312 switches, a 58.1 percent reduction compared to the reactive strategies. Fewer switches and more energy recovered is an improvement on both axes, not a tradeoff. The reactive strategies accumulate high switch counts because they wait until batteries are deeply depleted before acting, then cycle rapidly; rolling replenishment distributes transitions evenly across the mission timeline, keeping batteries in the high-efficiency CC phase for a greater fraction of their charge time.

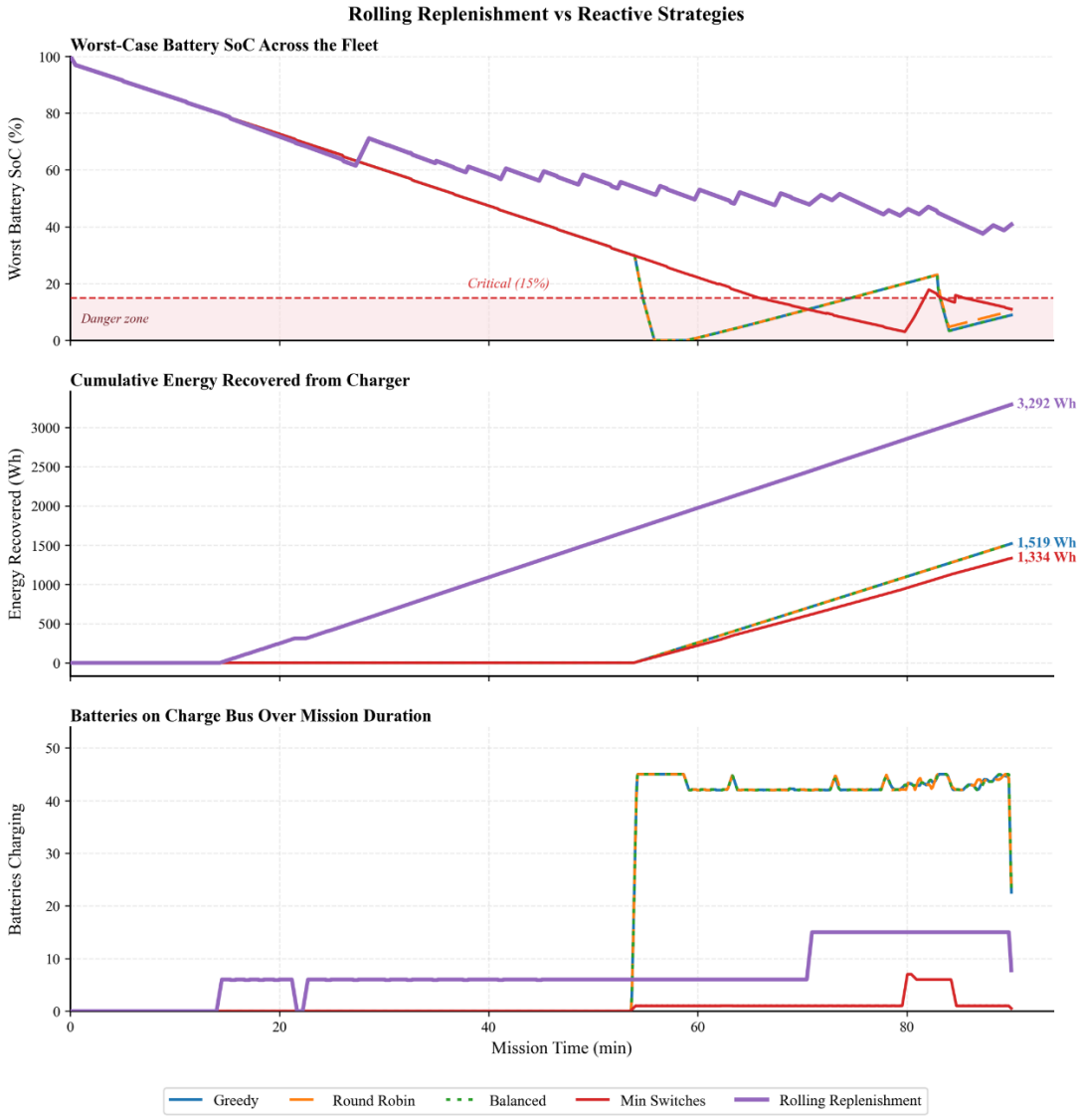

Figure 45. Five-Strategy Comparison: Worst-Case SoC, Energy Recovered, and Charge-Bus Occupancy.

The average SoC of 70.85 percent, versus 50.98 percent for reactive strategies, means the system operates with nearly 20 percentage points more energy reserve throughout the mission, providing a substantially larger safety margin against unexpected load increases, generator malfunctions, or mission extensions.

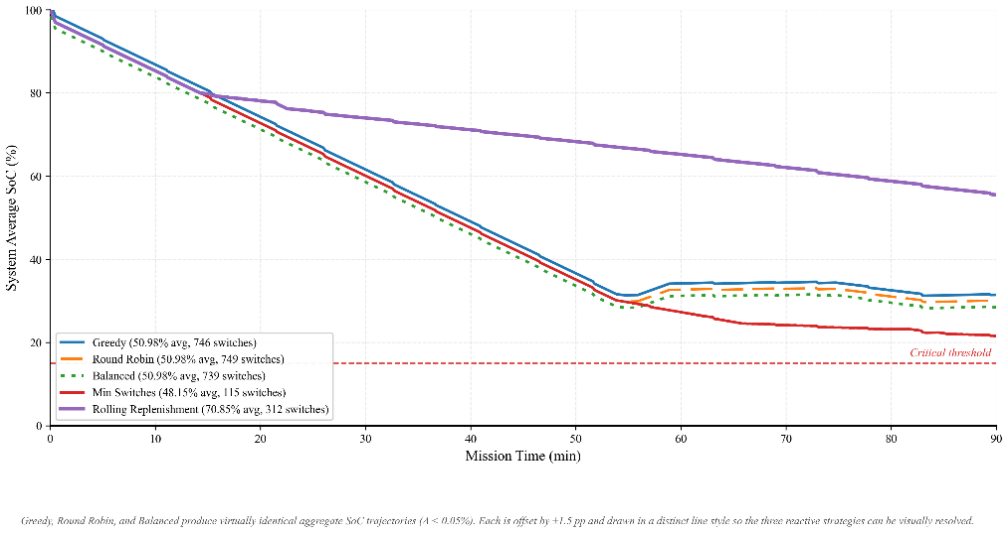

Figure 46. System-Average SoC Across All Five Scheduling Strategies.

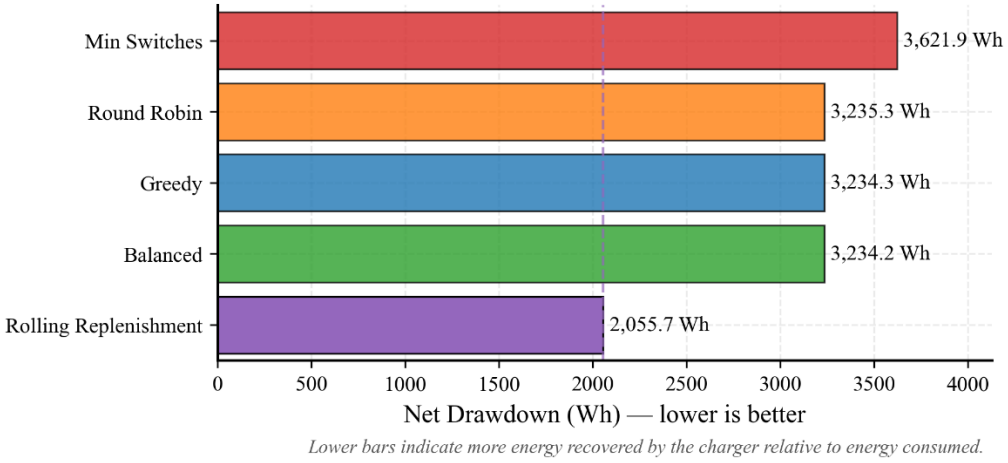

Figure 47. Net Energy Drawdown by Scheduling Strategy.

6.5.2 Prototype Results (9 Batteries)

At the prototype scale, the charger-power-to-battery-count ratio falls below the threshold at which proactive scheduling can deliver its energy-recovery advantage. The results in Table 22 confirm this expected boundary condition; the underlying mechanism is unpacked in the discussion that follows.

Table 22. Prototype Simulation Results (9-Battery).

Strategy	Flight (min)	Switches	Discharged (Wh)	Recharged (Wh)	Net Drawdown (Wh)
Greedy	42.3	1,419	2,104.2	1,324.7	779.5
Round Robin	41.7	1,463	2,082.6	1,326.9	755.8
Balanced	42.5	1,326	2,104.5	1,323.1	781.4
Min Switches	44.3	910	2,107.3	1,323.2	784.2
Rolling Replenishment	**35.4**	**541**	**2,110.0**	**1,328.7**	**781.3**

At the prototype scale, all strategies recovered approximately the same total energy (1,323 to 1,329 Wh), indicating that the charger, not the scheduling algorithm, is the binding constraint at this scale. Rolling Replenishment's shorter flight time results from committing more batteries to the charge bus earlier, which with only three batteries per junction can leave a single battery carrying the full junction current and depleting faster. In the full-system configuration with sixteen batteries per junction, this effect is negligible because Tier 3 still retains six active batteries per junction. The prototype results demonstrate a well-understood boundary condition in which the charger power-to-battery count ratio is insufficient for proactive scheduling to deliver its advantage, which is precisely why the full-system simulation was conducted.

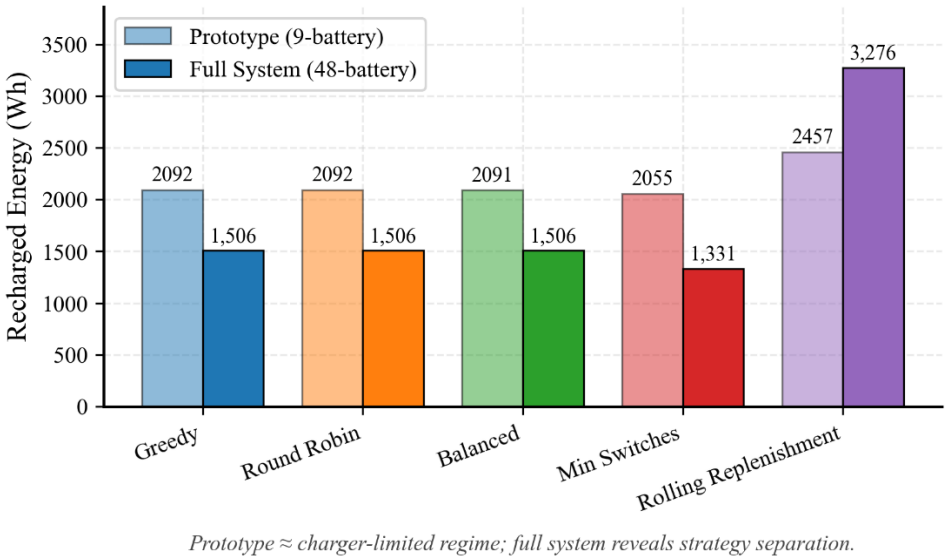

Figure 48. Recharged Energy: Prototype vs. Full System.

6.5.3 Rolling Replenishment Tier and Junction Analysis

Under the throttle profile, junction medians cross the 80 percent Tier 1 threshold within the first 10 to 15 minutes, initiating charging while reactive strategies leave the charger idle. Tier 2 is typically reached between minutes 25 and 40; Tier 3, if reached, commits up to ten batteries per junction to the charge bus. Tier escalation is junction-independent, directing charging resources where most required rather than spreading uniformly.

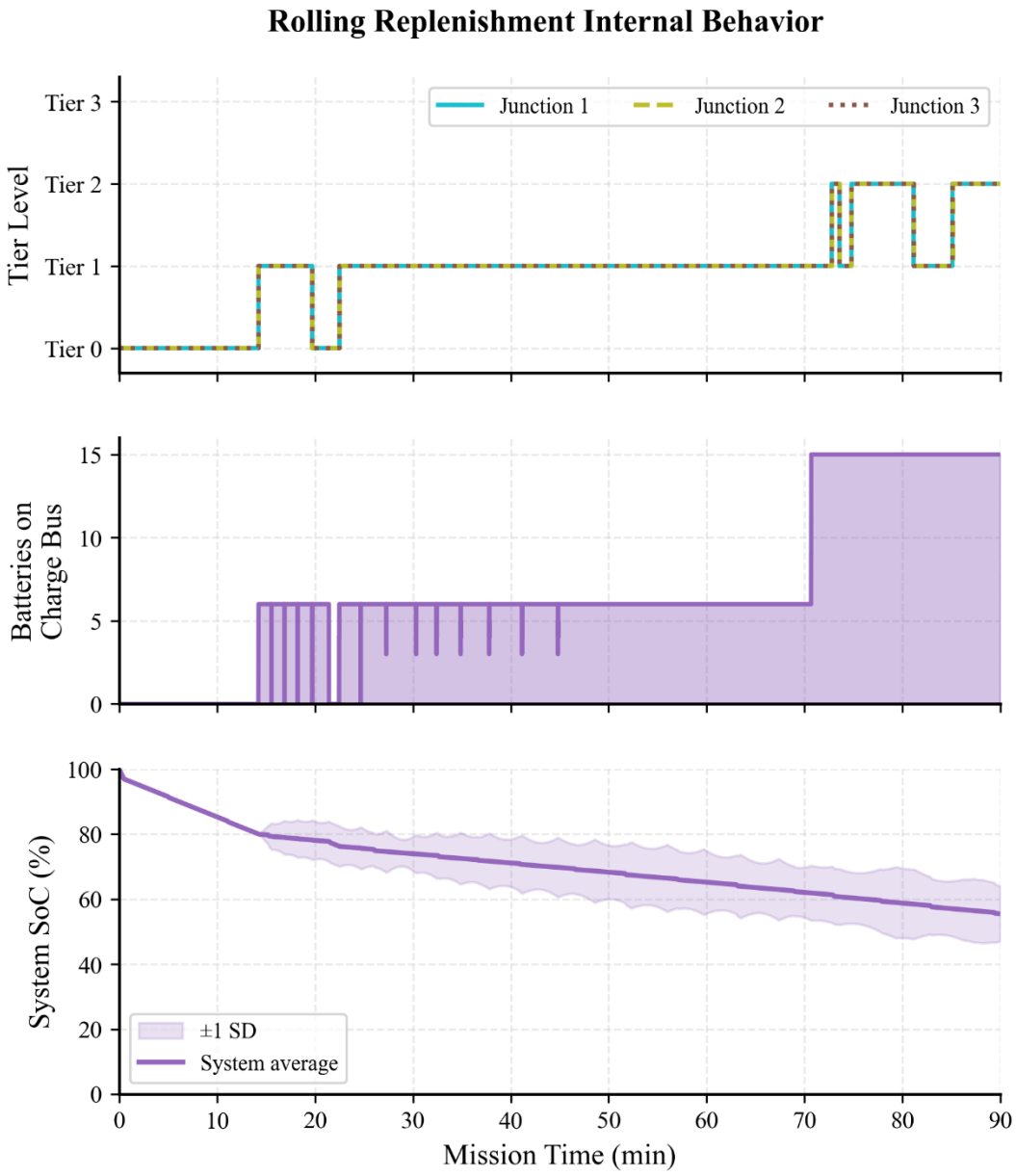

Figure 49. Rolling Replenishment Tier, Charge-Bus, and SoC Behavior.

The continuous swap mechanism keeps the charger connected to batteries in the CC phase, where charge acceptance is at its maximum. Without the swap, batteries would remain on the charge bus until reaching 85 percent, spending increasing time in the low-current CV taper. By cycling batteries out once their SoC rises 30 points above

the weakest motor-bus battery, Rolling Replenishment directs charger power toward batteries with the greatest remaining capacity to accept energy at the full 1C rate.

The combined effect, early activation, tiered escalation, and continuous swapping, is reflected in the recharged energy totals: 3,292.0 Wh versus 1,505.6 Wh for the best reactive strategy. Under reactive strategies, the charger is idle for the first 30 to 40 minutes; under Rolling Replenishment, the charger begins operating within the first 15 minutes and remains active for the remainder.

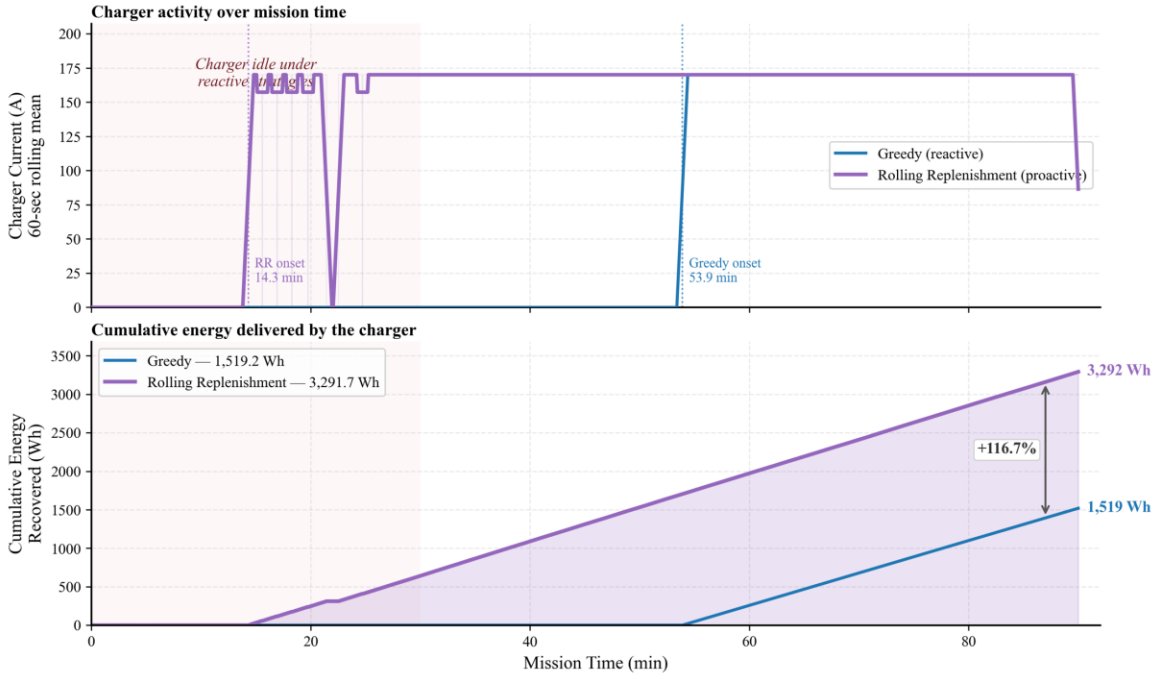

Figure 50. Charger Utilization and Cumulative Energy Recovery: Greedy vs. Rolling Replenishment.

6.6 Failure Handling and Model Limitations

6.6.1 Failure Scenarios and System Response

The simulation models three failure categories. Individual battery depletion (SoC reaches 0 percent) is handled by the scheduling algorithm, which transitions the battery to idle or charge. System-wide energy exhaustion (all batteries at or below 15 percent critical SoC with insufficient charging) terminates the run and, in a flight deployment, would trigger the Pixhawk's autonomous return-to-home protocol. Charging source loss, either through fuel exhaustion or modeled engine failure, makes the charger permanently unavailable, and the system operates on residual battery energy until the critical threshold is reached.

The simulation validates series-chain integrity at every decision cycle. If any junction drops to zero active batteries, the system transitions to emergency-landing state (70 percent power reduction) and terminates. No strategy produced a junction violation in any run.

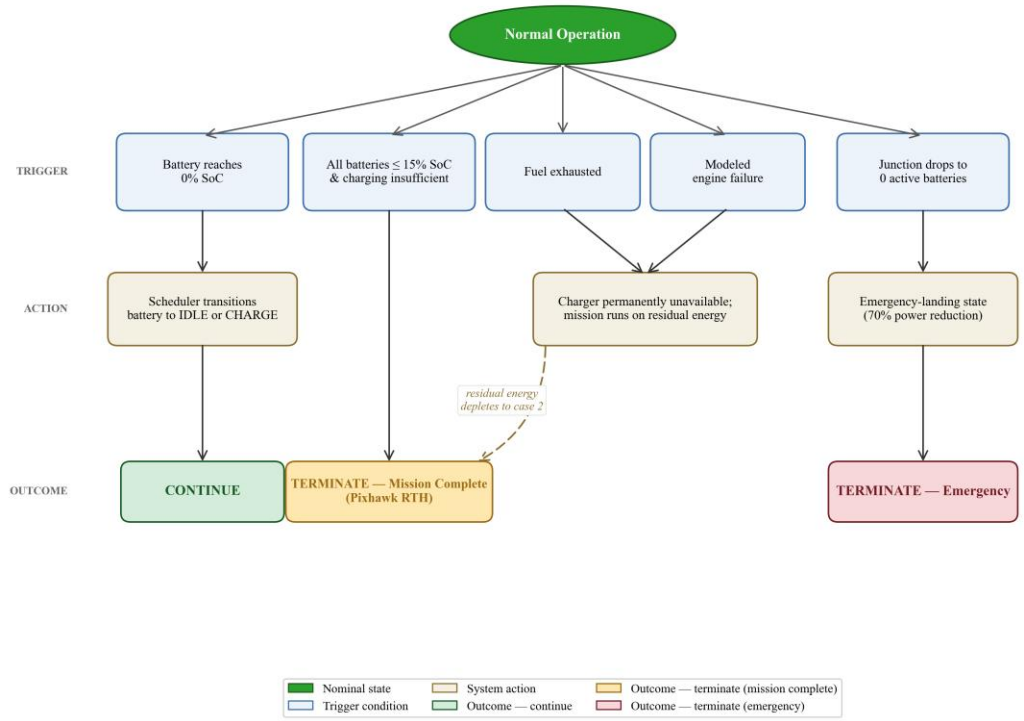

Figure 51. Simulation Failure-Handling Decision Tree Mapping Trigger Conditions to System Actions and Outcomes.

6.6.2 Simulation Assumptions and Error Bounds

The simulation makes several simplifying assumptions that bound absolute energy values while preserving comparative strategy validity. Every simplification applies identically across all five strategies, meaning the relative performance differences are high-confidence comparisons even where absolute values carry bounded uncertainty.

The most significant simplification is the absence of an internal resistance model. Current sharing among parallel batteries is assumed perfectly equal. Additional simplifications include omission of capacity degradation over repeated cycles, self-discharge during idle periods, SSR contact resistance, serial command latency, and

environmental temperature effects. Each factor would introduce losses by reducing actual flight times relative to simulated values.

The integration method is explicit Euler at a fixed 0.5-second timestep, with estimated cumulative error of approximately 2 percent. Absolute energy values carry an estimated error bound of approximately 5 percent. The relative comparisons between strategies are substantially more precise because systematic errors are common-mode and cancel in the difference.

Table 23. Simulation Model Simplifications and Effect on Results.

Simplification	Physical Reality	Effect on Absolute Values	Effect on Strategy Comparison
No internal resistance	Cells exhibit voltage sag proportional to current × R_int	Overestimates usable energy ~3–5%	Minimal, same across all strategies
Ideal current sharing	Higher-V packs carry slightly more current	Slight overestimate of uniformity	Negligible, affects all equally
No capacity degradation	Repeated cycling reduces usable capacity	Overestimates long-term energy	None, single-mission simulation
No self-discharge	Idle cells lose ~1–3% SoC per month	Negligible over 90-min mission	None
No SSR contact resistance	SSR-40DD has ~15 mΩ on-state	Underestimates losses by <0.5%	None, same SSR count all strategies
No command latency	Serial UART ~10–50 ms per command	Negligible at 0.5 s timestep	None

No temperature effects	Capacity/resistance vary with temp	Could shift absolute values ±5–10%	Minimal, common-mode

Chapter 7: Broader Application and Discussion

This chapter examines the applicability of the intelligent power management architecture to domains beyond the Skywalker III hexacopter. Section 7.1 analyzes the electric vehicle industry. Section 7.2 extends the discussion to other battery-operated mobile platforms. Section 7.3 addresses scaling considerations.

7.1 Application to Electric Vehicles

7.1.1 Current EV BMS Architecture and Documented Limitations

Section 2.3 established the technical foundation for applying intelligent battery management to the EV industry, including cell-count and architecture comparisons across manufacturers and the distinction between passive and active balancing. This section advances that analysis by examining documented consequences of current architectural limitations.

The limitations of module-level monitoring are not theoretical. In January 2026, Volvo recalled 40,323 EX30 electric SUVs worldwide after a manufacturing process deviation at the Sunwoda battery plant caused lithium plating to form on cell anodes, creating internal short circuits capable of triggering thermal runaway (National Highway Traffic Safety Administration [NHTSA], 2026). The defect developed silently within individual cells and remained invisible to the vehicle's BMS until the risk of fire had materialized. Volvo instructed owners to limit charging to 70 percent SoC and park

outdoors until battery modules could be replaced, at an estimated remediation cost exceeding $195 million (NHTSA, 2026).

General Motors recalled approximately 141,600 Chevrolet Bolt EVs and EUVs across the 2017–2022 model years after identifying the simultaneous presence of two rare cell-level manufacturing defects, a torn anode tab and a folded separator, within the same battery cell (NHTSA, 2021). Five confirmed vehicle fires and two injuries were reported before the recall was completed, with total remediation costs reaching approximately $2 billion (NHTSA, 2021). In both cases, failures originated at the individual cell level and propagated undetected through module-level monitoring architectures.

Table 24. EV Battery Recall Comparison.

Vehicle	Recall Period	Units Affected	Est. Cost	Root Cause
Chevrolet Bolt	2017–2022	~141,600	~$2 billion	Torn anode tab and folded separator defect undetected by module-level BMS (NHTSA, 2021)
Hyundai Kona EV	2020–2021	~82,000	~$900 million	Cell internal short circuit from separator damage during manufacturing

A BMS architecture capable of identifying anomalous cell-level behavior, voltage deviations, impedance changes, or capacity fade patterns inconsistent with neighboring cells, could flag these conditions before they progress to irreversible states. The monitoring infrastructure demonstrated in this thesis addresses this gap.

The industry trajectory further supports deeper cell-level diagnostics. In October 2025, NXP Semiconductors announced the first automotive BMS chipset with integrated Electrochemical Impedance Spectroscopy (EIS), comprising the BMA7418 cell sensing device, the BMA6402 gateway, and the BMA8420 battery junction box controller (NXP Semiconductors, 2025). This chipset enables real-time high-frequency impedance measurements at the cell level within production vehicles, confirming that the automotive industry is actively moving toward the granular monitoring demonstrated as technically feasible in this thesis.

However, improved monitoring alone is insufficient. The contribution of this thesis is not the collection of cell-level data, production BMS ICs from Texas Instruments (BQ76952), Analog Devices (ADBMS6830), and NXP already accomplish this, but the control architecture that acts on this data to make real-time power routing decisions.

7.1.2 Module-Level Application of Thesis Architecture

As established in Section 2.3.2, per-cell switching across thousands of individual cells is not practically feasible. The same three-state switching logic (MOTOR, CHARGE, IDLE) validated at the battery level in the prototype can instead be applied at the module level within an EV pack. A Tesla Model S pack comprising 16 modules would require 64 SSRs, comparable to the validated nine-battery prototype.

The mapping preserves all core system behaviors. Each module would occupy one of three states, with the junction constraint requiring each junction to maintain at least one unit in the MOTOR state. The scheduling intelligence transfers directly:

Rolling Replenishment's tier thresholds, swap differential, and dwell time parameters would require recalibration for automotive load profiles and charging rates, but the algorithmic structure remains unchanged.

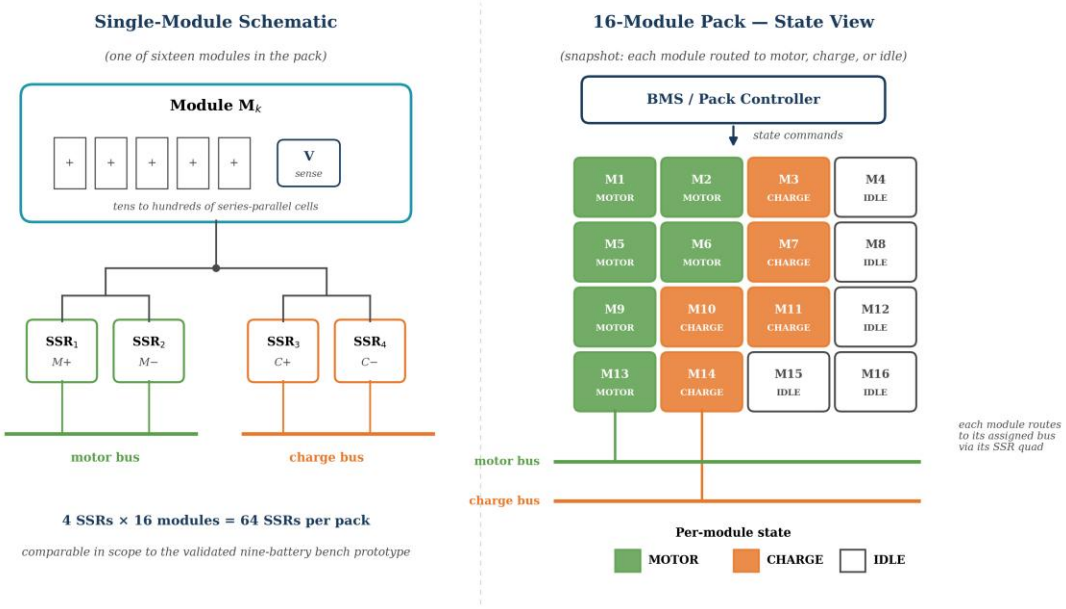

Figure 52. Module-Level Application of the Thesis Architecture to a 16-Module EV Pack.

The practical significance extends beyond energy recovery to safety-critical fault management. When a module exhibits anomalous behavior, the system can transition it to IDLE for diagnostic isolation without reducing pack output. The degraded-mode operation logic validated in simulation (Section 6.6.1) applies directly: the system redistributes load across remaining healthy modules and continues operating at reduced but stable capacity. An EV implementing this architecture could continue driving safely after isolating a failing module, alerting the driver to seek service while maintaining propulsive capability.

7.1.3 Series Hybrid Precedent and Reaction-Time Analysis

The series hybrid concept has established automotive precedent. The BMW i3 Range Extender paired a 22 kWh battery pack with a 647 cc two-cylinder engine connected to a 34 kW generator, extending range from approximately 130 km to over 240 km (Saravanakumar et al., 2023). The Chevrolet Volt employed a 16 kWh (first generation) or 18.4 kWh (second generation) battery pack supplemented by a 1.4 L or 1.5 L engine operating as a generator, achieving combined range exceeding 600 km (Saravanakumar et al., 2023). Both have been discontinued but validated the commercial viability of the series hybrid architecture.

The fundamental argument for series hybrid over direct combustion drive is reaction time. An internal combustion engine delivers torque through the four-stroke thermodynamic cycle, intake, compression, power, exhaust, with aggregate transient response from idle to peak torque on the order of several hundred milliseconds to over one second. Electric motors generate torque through electromagnetic field interaction, with production EV motor controllers achieving response times of less than 50 milliseconds and brushless DC ESCs in UAV applications responding on a single-digit millisecond timescale (T-Motor, 2024). The difference spans approximately two orders of magnitude.

For a hexacopter computing attitude corrections at rates exceeding 400 Hz, a propulsion source introducing hundreds of milliseconds of latency would render stable flight physically impossible. For an electric vehicle executing traction control or sudden lane changes, the same principle applies with less severity but equal validity. The series

hybrid architecture resolves this mismatch by allowing the combustion engine to operate at a fixed RPM optimized for fuel efficiency while batteries serve as an energy buffer that absorbs slow, continuous generator output and releases it as fast, transient bursts matched to load dynamics.

7.2 Application to Other Mobile Platforms

The underlying principles, cell-level monitoring, automated state transitions, and proactive scheduling, are applicable to any battery-operated mobile platform where operational endurance is constrained by energy storage capacity.

Table 25. Battery Constraint Summary Across Mobile Platforms.

Platform	Chemistry	Capacity	Primary Constraint	Thesis Application
UAV (Skywalker III)	LiPo	~4,600 Wh	Limited flight time	Series hybrid with onboard generator
Warehouse Robotics	Li-ion/LFP	1–10 kWh	Charging station downtime	Regen braking energy harvesting
Underwater AUV	Li-ion	1–50 kWh	Fixed energy, no recharge	Fault isolation for reliability
Agricultural Drone	LiPo	1–5 kWh	Insufficient range	Extended flight via battery rotation
Military Ground	Li-ion/LFP	20–100+ kWh	No field charging	Generator-fed replenishment
Electric Vehicle	NCA/NMC/LFP	40–100+ kWh	Range anxiety, degradation	Module-level switching

Warehouse robotic systems that must return to charging stations lose throughput proportional to charging downtime. If the onboard BMS could rotate battery modules between active and charging states using energy harvested from regenerative braking, continuous operation without dedicated charging interruptions becomes feasible.

Underwater autonomous vehicles present a particularly compelling application, as mission endurance is determined entirely by battery capacity and recovery of a failed vehicle from deep-water operations may be impractical. The fault isolation capability demonstrated in the prototype, transitioning a battery to IDLE without interrupting current to the remaining system, is directly relevant to subsea platforms where mission-critical reliability is paramount.

Agricultural drone platforms for crop spraying and field surveying require extended flight duration to cover large areas without returning to base. A series hybrid architecture with intelligent battery rotation could enable continuous operation throughout an entire survey or treatment cycle. Military ground vehicles operating without access to charging infrastructure could extend mission duration by continuously replenishing battery modules from an onboard generator while in transit.

7.3 Scalability Discussion

Scaling the architecture from the nine-battery prototype to production systems managing hundreds or thousands of switchable units involves four primary engineering dimensions: monitoring hardware, switching hardware, communication infrastructure,

and scheduling algorithm parameterization. None requires fundamental changes to the control logic validated in this thesis.

The prototype's Arduino-based monitoring would be replaced by dedicated BMS ICs (Texas Instruments BQ76952, Analog Devices ADBMS6830, or NXP BMA7418) providing integrated isolation for high-voltage packs and daisy-chain communication for scaling to arbitrary cell counts.

The SSR hardware scales linearly with switchable units: the 48-battery Skywalker III requires 192 SSRs; a 16-module EV pack requires 64. For automotive voltages (350–800 V), SSR-40DD relays would be replaced with wide-bandgap semiconductor switches, particularly silicon carbide (SiC) MOSFETs, offering the voltage ratings, thermal performance, and switching characteristics appropriate for high-voltage drivetrain buses.

The scheduling algorithm is the most naturally scalable component. Rolling Replenishment operates on junction-level median SoC values and maps them to tier thresholds; extending to larger systems requires recalibrating only the per-tier unit count and the charge-complete threshold, both determined by charger power and per-module charge acceptance rate. The same algorithm operated across 9- and 48-battery configurations without modification, producing proportionally larger performance gains at the larger scale.

Table 26. Scalability Comparison: Prototype vs. Automotive Implementation.

Dimension	Thesis Prototype	Automotive Implementation

Switchable Unit	Individual 4S battery	Module (e.g., 16-cell group)
Unit Count	9 (prototype) / 48 (full)	16 modules (Model S equiv.)
Monitoring HW	Arduino Mega + resistor dividers	BMS IC (BQ76952 / ADBMS6830)
Monitoring Accuracy	~4.9 mV ADC resolution	±2 mV typical
Switching HW	SSR-40DD (5–200 VDC, 40 A)	SiC MOSFETs (350–800 V)
SSRs per Unit	4 (dual motor + dual charge)	4 (same architecture)
Total SSR Count	36 / 192	64
Communication	Serial UART via USB hub	CAN bus w/ message arbitration
Scheduling	Rolling Replenishment (Python)	Rolling Replenishment (embedded C)
Control Logic	Three-state (MOTOR/CHARGE/IDLE)	Three-state (unchanged)

The progression from prototype to production involves replacing components with higher-specification equivalents, Arduino with BMS ICs, SSR-40DD with SiC switches, serial UART with CAN bus, while the control architecture, state machine logic, and scheduling algorithms remain unchanged.

Chapter 8: Conclusions and Recommendations

This chapter summarizes the contributions of this thesis, presents conclusions drawn from hardware validation and simulation results, and identifies recommendations for future work.

8.1 Summary of Contributions

This thesis set out to design, build, and validate an intelligent power management system for battery-operated mobile platforms. The resulting architecture combines three capabilities that existing battery management systems do not integrate:

The first capability is cell-level voltage monitoring. The system reads individual cell voltages across a multi-battery configuration in real time using Arduino-based voltage divider circuits, derives per-cell SoC estimates through an OCV lookup table with linear interpolation, and applies the minimum-cell method to ensure the battery's reported SoC reflects its weakest cell rather than an average that could mask degradation.

The second capability is automated power source switching through solid-state relays with full ground-side isolation on the charge path. The four-SSR-per-battery configuration enables individual batteries to transition between MOTOR, CHARGE, and IDLE states without interrupting current delivery to the load. The ground-side isolation discovery, which resolved a shared-ground conflict that would have prevented simultaneous multi-battery charging, constitutes one of the most significant technical outcomes of the hardware development.

The third capability is proactive scheduling through the Rolling Replenishment algorithm, which monitors junction-level health and rotates batteries between states based on tier-based charging allocation rather than reacting to individual batteries reaching critical SoC thresholds.

These three capabilities were validated through two complementary methods. The nine-battery bench prototype verified end-to-end system functionality across fifteen tests, confirming voltage monitoring accuracy, reliable SSR state transitions under multi-motor load with zero disruption, conflict-free operation, and adherence to the two-second transition delay. The 48-battery Python simulation validated scheduling strategies at full system scale, confirming Rolling Replenishment's performance advantages and zero junction constraint violations across all strategies and runs.

8.2 Conclusions

The problem statement in Section 1.3 identified three unmet capabilities in current battery management systems: real-time cell-level voltage monitoring, automated switching without load interruption, and an integrated architecture combining both with intelligent scheduling. The system developed in this thesis addresses all three.

The most consequential technical outcome was the parallel-first topology insight. The initial series-wired configuration required nanosecond-scale switching, a constraint that exceeded available switching hardware and led to multiple MOSFET failures (Section 4.4.2). Restructuring to a parallel-first topology, where batteries connect in parallel at each junction with junctions wired in series, eliminated this requirement

entirely. Removing one battery from a parallel junction does not interrupt the current path. This insight reduced the switching speed requirement from nanoseconds to milliseconds, enabling the use of commercially available SSRs and making the entire architecture feasible with off-the-shelf components.

The OCV-based SoC estimation method provided sufficient accuracy for scheduling validation, though voltage sag under heavy load introduces known limitations that would become more pronounced at the full 480 A system current. Enhanced estimation methods including coulomb counting and extended Kalman filtering are identified as future work.

The ground-side isolation discovery resolved a hardware-level conflict that would have prevented simultaneous charging and discharging in any shared-bus battery system. Isolating the charge path ground from the motor path ground using the four-SSR-per-battery configuration eliminated unintended current paths through the shared conductor. This discovery is generalizable to any multi-source battery system sharing batteries between two electrically distinct buses.

The simulation results provide strong evidence that proactive scheduling outperforms reactive scheduling for hybrid power management. All four reactive strategies maintained an average SoC of approximately 51 percent and recovered approximately 1,506 Wh through the charger. Rolling Replenishment maintained 70.85 percent average SoC, recovered 3,292.0 Wh (118.6 percent more), and achieved a net drawdown of 2,055.7 Wh, 1,178.6 Wh less than the Greedy baseline, all while executing 58.1 percent fewer switch events (312 vs. 739).

The architecture is not specific to any single platform. As demonstrated in Chapter 7, the same monitoring, switching, and scheduling logic applies at the module level within EV battery packs (Section 7.1.2), and the principles extend to warehouse robotics, underwater vehicles, agricultural drones, and military ground platforms (Section 7.2). The scalability analysis confirmed that progression from prototype to production involves replacing components with higher-specification equivalents while the control architecture and scheduling algorithms remain unchanged.

8.2.1 Limitations

The validation work has bounded scope that should be acknowledged. Hardware testing was conducted at the nine-battery bench scale; the 48-battery configuration was evaluated only in simulation. The Tier 4 closed-loop test demonstrated a single autonomous detect-and-swap event; multi-cycle extended-duration autonomous operation has not yet been validated on hardware. The Rolling Replenishment scheduling algorithm, the principal algorithmic contribution of this thesis, was developed and evaluated entirely in simulation and has not yet been executed on the bench prototype. Voltage monitoring was characterized at a single static operating point; per-cell accuracy across the full SoC range and under load was not measured. The simulation does not model internal cell resistance, capacity degradation, self-discharge, or temperature effects (Section 6.6.2). The Fotek SSR-40DD relays selected for the bench were not exercised near their 40 A rated current; the Skywalker III per-motor peak of approximately 80 A would require either higher-rated SSRs or paralleled relay configurations.

8.3 Recommendations for Future Work

The most immediate priority is upgrading the SSR hardware to accommodate higher current levels. The Fotek SSR-40DD relays are rated for 40 A continuous, adequate for bench testing but below the full Skywalker III requirements where individual motors draw up to 80 A. Future implementations will require SSRs rated for the full per-motor current or paralleled relay configurations. For automotive-scale applications, silicon carbide MOSFETs will provide the voltage ratings, thermal performance, and switching characteristics appropriate for 350–800 V drivetrain buses.

On the monitoring side, the Arduino-based architecture will be replaced with dedicated automotive BMS ICs, such as the Texas Instruments BQ76952, Analog Devices ADBMS6830, or NXP BMA7418, providing higher measurement accuracy (±2 mV vs. ~4.9 mV), integrated galvanic isolation, and daisy-chain communication for scaling without proportional wiring complexity.

The SoC estimation method will benefit from integrating coulomb counting alongside OCV mapping to compensate for voltage sag under heavy load. Extended Kalman filtering, which fuses voltage, current, and temperature measurements into a state-space model, will offer further accuracy improvements for high-current applications (Wang et al., 2020). These methods can be implemented within the existing scheduling framework without changes to the control architecture.

The most critical near-term validation step is implementing Rolling Replenishment on the physical prototype. The algorithm has been validated through

simulation at both scales but has not yet been executed on the actual hardware. Deployment will require integrating the tier-based scheduling logic into the SSR controller firmware, establishing the real-time communication loop between monitoring Arduinos and the scheduler, and verifying that physical state transitions perform as predicted by the simulation.

A full flight integration test will represent the definitive validation. Such a test will combine the voltage monitoring circuit, SSR switching architecture, Rolling Replenishment scheduler, and the WEN GN5602X generator on the Skywalker III airframe with all six motors operating under actual flight loads, subjecting the system to dynamic throttle profiles, vibration, and thermal conditions that a bench test cannot replicate.

An EV-scale proof of concept using automotive-grade components will determine whether the architecture translates to commercial viability. This effort will involve replacing prototype monitoring with production BMS ICs, substituting SSR-40DD relays with automotive-qualified semiconductor switches, transitioning serial UART to CAN bus, and validating Rolling Replenishment against realistic EV drive cycles.

References

Arduino. (n.d.). *Arduino Mega 2560 Rev3 technical specifications* [Technical documentation]. Arduino S.r.l. https://docs.arduino.cc/hardware/mega-2560/

Arora, S., Shen, W., & Kapoor, A. (2016). Review of mechanical design and strategic placement technique of a robust battery pack for electric vehicles. *Renewable and Sustainable Energy Reviews*, *60*, 1319–1331. https://doi.org/10.1016/j.rser.2016.03.013

Åström, K. J., & Wittenmark, B. (1995). *Adaptive control* (2nd ed.). Addison-Wesley.

Barré, A., Deguilhem, B., Grolleau, S., Gérard, M., Suard, F., & Riu, D. (2013). A review on lithium-ion battery ageing mechanisms and estimations for automotive applications. *Journal of Power Sources*, *241*, 680–689. https://doi.org/10.1016/j.jpowsour.2013.05.040

Bilansky, J., Ivan, J., Bodnar, D., & Lacko, M. (2022). Analysis of Li-ion battery cell internal impedance changes based on temperature and SOH. *Acta Electrotechnica et Informatica*, *22*(3), 3–8. https://doi.org/10.2478/aei-2022-0011

Boukoberine, M. N., Zhou, Z., & Benbouzid, M. (2019). A critical review on unmanned aerial vehicles power supply and energy management: Solutions, strategies, and prospects. *Applied Energy*, *255*, 113823. https://doi.org/10.1016/j.apenergy.2019.113823

Brand, M. J., Hofmann, M. H., Schuster, S. S., Keil, P., & Jossen, A. (2016). Current distribution within parallel-connected battery cells. *Journal of Power Sources*, *334*, 202–212. https://doi.org/10.1016/j.jpowsour.2016.10.010

Brandl, M., Gall, H., Wenger, M., Lorentz, V., Giegerich, M., Baronti, F., Fantechi, G., Fanucci, L., Roncella, R., Saletti, R., Saponara, S., Thaler, A., Cifrain, M., & Prochazka, W. (2012). Batteries and battery management systems for electric vehicles. In *Proceedings of the Design, Automation & Test in Europe Conference* (pp. 971–976). IEEE. https://doi.org/10.1109/DATE.2012.6176637

Ci, S., Lin, N., & Wu, D. (2016). Reconfigurable battery techniques and systems: A survey. *IEEE Access*, *4*, 1175–1189. https://doi.org/10.1109/ACCESS.2016.2545338

Corcau, J. I., & Dinca, L. (2023). Fuel cell/battery hybrid electric system for UAV. In *Proceedings of the IEEE International Conference on Electrical Systems for Aircraft, Railway, Ship Propulsion and Road Vehicles & International Transportation Electrification Conference (ESARS-ITEC)*. IEEE.

Cover, T. M., & Thomas, J. A. (2006). *Elements of information theory* (2nd ed.). Wiley-Interscience.

Fleming, J., Amietszajew, T., McTurk, E., Towers, D. P., Greenwood, D., & Bhagat, R. (2022). A smart cell monitoring system based on power line communication—Optimization of instrumentation and acquisition for smart battery management. *IEEE Access*, *10*, 7058–7070. https://doi.org/10.1109/ACCESS.2021.3142180

Fotek. (n.d.). *SSR-40DD DC-DC solid state relay datasheet* [Product datasheet]. Fotek Controls Co., Ltd.

Hashemi, S. R., Esmaeeli, R., Aliniagerdroudbari, H., Alhadri, M., Alshammari, H., Mahajan, A., & Farhad, S. (2019). New intelligent battery management system for drones. In *Proceedings of the ASME 2019 International Mechanical Engineering Congress and Exposition*. ASME. https://doi.org/10.1115/IMECE2019-10479

Heywood, J. B. (2018). *Internal combustion engine fundamentals* (2nd ed.). McGraw-Hill Education.

International Rectifier. (2005). *IR2110/IR2113(S) high and low side driver datasheet* (PD60147 Rev. U) [Datasheet]. Infineon Technologies. https://www.infineon.com

International Rectifier. (2011). *IRFB4110PbF HEXFET power MOSFET datasheet* (PD-97114 Rev. C) [Datasheet]. Infineon Technologies. https://www.infineon.com/dgdl/irfb4110pbf.pdf

Jiao, S., Zhang, G., Zhou, M., & Li, G. (2023). A comprehensive review of research hotspots on battery management systems for UAVs. *IEEE Access*, *11*, 84636–84650. https://doi.org/10.1109/ACCESS.2023.3303855

Lambert, F. (2017, August 24). Tesla Model 3: Exclusive first look at Tesla's new battery pack architecture. *Electrek*. https://electrek.co/2017/08/24/tesla-model-3-exclusive-battery-pack-architecture/

Lu, L., Han, X., Li, J., Hua, J., & Ouyang, M. (2013). A review on the key issues for lithium-ion battery management in electric vehicles. *Journal of Power Sources*, *226*, 272–288. https://doi.org/10.1016/j.jpowsour.2012.10.060

Malla, A., & Shrestha, R. (2021). Development of an intelligent power management system for solar PV-wind-battery-fuel-cell integrated system. *Frontiers in Energy Research*, *9*, 613958. https://doi.org/10.3389/fenrg.2021.613958

Miao, Y., Hynan, P., von Jouanne, A., & Yokochi, A. (2019). Current Li-ion battery technologies in electric vehicles and opportunities for advancements. *Energies*, *12*(6), 1074. https://doi.org/10.3390/en12061074

Mohan, N., Undeland, T. M., & Robbins, W. P. (2003). *Power electronics: Converters, applications, and design* (3rd ed.). Wiley.

National Highway Traffic Safety Administration. (2021, August 20). *Part 573 safety recall report 21V-650 — General Motors LLC (expansion of 21V-560)* [Government report]. https://static.nhtsa.gov/odi/rcl/2021/RCLRPT-21V650-2919.PDF

National Highway Traffic Safety Administration. (2026, January 2). *Part 573 safety recall report 26V001 — Volvo Car USA, LLC* [Government report]. https://static.nhtsa.gov/odi/rcl/2026/RCLRPT-26V001-2847.pdf

Neely, M. J. (2010). *Stochastic network optimization with application to communication and queueing systems*. Morgan & Claypool.

NXP Semiconductors. (2025, October 29). *NXP improves battery health monitoring with EIS capable battery management chipset* [Press release]. https://investors.nxp.com/news-releases/news-release-details/nxp-improves-battery-health-monitoring-eis-capable-battery

Saravanakumar, Y. N., Sultan, M. T. H., Shahar, F. S., Giernacki, W., Łukaszewicz, A., Nowakowski, M., Holovatyy, A., & Stępień, S. (2023). Power sources for unmanned aerial vehicles: A state-of-the-art. *Applied Sciences*, *13*(21), 11932. https://doi.org/10.3390/app132111932

T-Motor. (2024). *P80 III KV120 brushless motor datasheet* [Product datasheet]. Nanjing T-Motor Electronic Co., Ltd. https://uav-en.tmotor.com

Texas Instruments. (2021). *BQ76952 16-series automotive battery monitor and protector datasheet* (SLUSD85D) [Datasheet]. https://www.ti.com/product/BQ76952

Wang, Y., Tian, J., Sun, Z., Wang, L., Xu, R., Li, M., & Chen, Z. (2020). A comprehensive review of battery modeling and state estimation approaches for advanced battery management systems. *Renewable and Sustainable Energy Reviews*, *131*, 110480. https://doi.org/10.1016/j.rser.2020.110480

Youme Power. (n.d.). *6500 mAh 4S 80C LiPo battery datasheet* [Product datasheet].

Zhang, F., Yu, X., Shao, Z., Liu, F., Zhang, X., Zhou, Z., & Li, T. (2022). Improving thermal efficiency of internal combustion engines: Recent progress and remaining challenges. *Energies*, *15*(17), 6222. https://doi.org/10.3390/en15176222

Zhang, H., Cheng, W., Li, Z., Shi, J., Zhao, R., & Cao, L. (2024). Extending the BESS lifetime: A cooperative multi-agent deep Q network framework for a parallel-series connected battery pack. *Energies*, *17*(18), 4604. https://doi.org/10.3390/en17184604

Appendix A: Embedded Firmware

All embedded firmware is written in Arduino C/C++ and deployed to Arduino Mega 2560 microcontrollers.

A.1 Voltage Monitoring Firmware (Arduino Mega, Boards 1–3)

Deployed to the three voltage monitoring Arduino Megas, one per junction. Boards 1 and 2 use a 33 kΩ / 10 kΩ divider (scale factor 4.3). Board 3 uses a 30 kΩ / 10 kΩ configuration; deployment requires only changing BOARD_NUMBER to 3 and R1 to 30000.0.

```
// Skywalker Battery Monitor
// Deploy to Arduino Mega #1, #2, or #3
// Boards 1-2: 33k / 10k divider (scale 4.3)
// Board 3:    30k / 10k divider (scale 4.0) — adjust R1 accordingly

const float R1             = 33000.0;
const float R2             = 10000.0;
const float DIVIDER_RATIO  = R2 / (R1 + R2);
const float SCALE_FACTOR   = 1.0 / DIVIDER_RATIO;
const float VREF           = 5.0;
const int   ADC_MAX        = 1023;
const int   OVERSAMPLE     = 32;
const unsigned long PRINT_INTERVAL = 2000;

// Change per board: 1 for batteries 1-3, 2 for batteries 4-6, 3 for batteries
7-9
const int BOARD_NUMBER     = 1;
const int BAT_ID_OFFSET    = (BOARD_NUMBER - 1) * 3;

const int BAT_A_PINS[4] = {A15, A14, A13, A12};
const int BAT_B_PINS[4] = {A11, A10, A9,  A8};
const int BAT_C_PINS[4] = {A7,  A6,  A5,  A4};
const int* ALL_BATS[3]  = {BAT_A_PINS, BAT_B_PINS, BAT_C_PINS};

// LiPo OCV-to-SoC lookup table (Section 4.3.3)
const int SOC_TABLE_SIZE = 11;
const float SOC_VOLTAGE[11] = {
    3.00, 3.30, 3.50, 3.60, 3.70, 3.75, 3.80, 3.85, 3.95, 4.10, 4.20
};
const float SOC_PERCENT[11] = {
    0.0, 5.0, 10.0, 15.0, 25.0, 35.0, 50.0, 65.0, 80.0, 95.0, 100.0
};

float tapVolts[3][4];
float cellVolts[3][4];
float cellSOC[3][4];
float packSOC[3];
```

```
// Linear interpolation between lookup table entries
float voltageToSOC(float cellV) {
    if (cellV <= SOC_VOLTAGE[0]) return 0.0;
    if (cellV >= SOC_VOLTAGE[SOC_TABLE_SIZE - 1]) return 100.0;
    for (int i = 1; i < SOC_TABLE_SIZE; i++) {
        if (cellV <= SOC_VOLTAGE[i]) {
            float ratio = (cellV - SOC_VOLTAGE[i - 1]) /
                          (SOC_VOLTAGE[i] - SOC_VOLTAGE[i - 1]);
            return SOC_PERCENT[i - 1] +
                   ratio * (SOC_PERCENT[i] - SOC_PERCENT[i - 1]);
        }
    }
    return 100.0;
}

// Read single pin with 32x oversampling for noise reduction
float readTapVoltage(int pin) {
    long sum = 0;
    for (int i = 0; i < OVERSAMPLE; i++) {
        sum += analogRead(pin);
        delayMicroseconds(80);
    }
    float avg = (float)sum / OVERSAMPLE;
    return (avg / ADC_MAX) * VREF * SCALE_FACTOR;
}

// Read all four taps, isolate cells by subtraction, compute SoC
void readBattery(int batIdx) {
    const int* pins = ALL_BATS[batIdx];
    for (int t = 0; t < 4; t++) {
        tapVolts[batIdx][t] = readTapVoltage(pins[t]);
    }
    // Subtraction method: isolate individual cell voltages
    cellVolts[batIdx][0] = tapVolts[batIdx][0];
    for (int c = 1; c < 4; c++) {
        cellVolts[batIdx][c] = tapVolts[batIdx][c] - tapVolts[batIdx][c - 1];
    }
    // Clamp negative values arising from ADC noise
    for (int c = 0; c < 4; c++) {
        if (cellVolts[batIdx][c] < 0.0) cellVolts[batIdx][c] = 0.0;
    }
    // Pack SoC = minimum cell SoC (Section 4.3.3)
    float minSOC = 100.0;
    for (int c = 0; c < 4; c++) {
        cellSOC[batIdx][c] = voltageToSOC(cellVolts[batIdx][c]);
        if (cellSOC[batIdx][c] < minSOC) minSOC = cellSOC[batIdx][c];
    }
    packSOC[batIdx] = minSOC;
}
```

```
void printBattery(int batIdx) {
    int batID = BAT_ID_OFFSET + batIdx + 1;
    Serial.print("=== Battery ");
    Serial.print(batID);
    Serial.print("  (Pack: ");
    Serial.print(tapVolts[batIdx][3], 2);
    Serial.print("V  SOC: ");
    Serial.print(packSOC[batIdx], 1);
    Serial.println("%) ===");
    for (int c = 0; c < 4; c++) {
        Serial.print("  Cell ");
        Serial.print(c + 1);
        Serial.print(":  ");
        Serial.print(cellVolts[batIdx][c], 3);
        Serial.print("V   SOC: ");
        Serial.print(cellSOC[batIdx][c], 1);
        Serial.print("%");
        if (cellVolts[batIdx][c] < 3.30 && cellVolts[batIdx][c] > 0.5) {
            Serial.print("  LOW");
        }
        if (cellVolts[batIdx][c] > 4.22) {
            Serial.print("  OVER");
        }
        Serial.println();
    }
    // Cell balance check against pack average
    float avgCell = tapVolts[batIdx][3] / 4.0;
    bool imbalanced = false;
    for (int c = 0; c < 4; c++) {
        float delta = cellVolts[batIdx][c] - avgCell;
        if (abs(delta) > 0.050 && cellVolts[batIdx][c] > 0.5) {
            if (!imbalanced) {
                imbalanced = true;
                Serial.println("  --- Imbalance detected ---");
            }
            Serial.print("    Cell ");
            Serial.print(c + 1);
            Serial.print(": ");
            Serial.print(delta * 1000, 1);
            Serial.println("mV from avg");
        }
    }
    if (!imbalanced) {
        Serial.println("  Balanced (within 50mV)");
    }
    Serial.println();
}

void setup() {
    Serial.begin(115200);
    while (!Serial) { ; }
    analogReference(DEFAULT);
```

```
    // Warm-up reads to stabilize ADC
    for (int i = 0; i < 20; i++) {
        analogRead(A15);
        delay(5);
    }
    Serial.println("========================================");
    Serial.print("  Skywalker Battery Monitor - Board #");
    Serial.println(BOARD_NUMBER);
    Serial.print("  Batteries: ");
    Serial.print(BAT_ID_OFFSET + 1);
    Serial.print(", ");
    Serial.print(BAT_ID_OFFSET + 2);
    Serial.print(", ");
    Serial.println(BAT_ID_OFFSET + 3);
    Serial.println("========================================");
    Serial.println();
}

unsigned long lastPrint = 0;

void loop() {
    for (int b = 0; b < 3; b++) {
        readBattery(b);
    }
    if (millis() - lastPrint >= PRINT_INTERVAL) {
        lastPrint = millis();
        for (int b = 0; b < 3; b++) {
            printBattery(b);
        }
        Serial.print(">> PACK SOC:  ");
        for (int b = 0; b < 3; b++) {
            Serial.print("Bat");
            Serial.print(BAT_ID_OFFSET + b + 1);
            Serial.print("=");
            Serial.print(packSOC[b], 0);
            Serial.print("%  ");
        }
        Serial.println();
        Serial.println("----------------------------------------");
        Serial.println();
    }
}
```

A.2 SSR Controller Firmware (Arduino Mega)

Drives all 36 solid-state relays through a single Arduino Mega. Serial commands specify battery number (1–9) and target state: A (motor), B (charge), or C (idle). Path transitions are sequenced

with a 200 ms delay: negative rail connects before positive on activation, positive disconnects before negative on deactivation.

```
// Skywalker Battery Management System - SSR Controller
// 9 Batteries x 4 SSRs = 36 relays total
// Commands: 1A-9A (motor), 1B-9B (charge), 1C-9C (idle), ALL, KILL

const int motorPos[9]  = {52, 50, 48, 46, 44, 42, 40, 38, 36};
const int chargePos[9] = {53, 51, 49, 47, 45, 43, 41, 39, 37};
const int motorNeg[9]  = {34, 32, 30, 28, 26, 24, 20, 22, 18};
const int chargeNeg[9] = {35, 33, 31, 29, 27, 25, 23, 21, 19};

#define IDLE   0
#define MOTOR  1
#define CHARGE 2
#define RELAY_DELAY 200

int batteryState[9];

void forceAllIdle() {
    Serial.println(">>> STARTUP: Forcing all relays OFF...");
    for (int i = 0; i < 9; i++) {
        digitalWrite(motorPos[i],  LOW);
        digitalWrite(chargePos[i], LOW);
        delay(100);
        digitalWrite(motorNeg[i],  LOW);
        digitalWrite(chargeNeg[i], LOW);
        batteryState[i] = IDLE;
        Serial.print("  Battery ");
        Serial.print(i+1);
        Serial.println(" -> IDLE");
    }
    Serial.println(">>> All relays OFF. System safe.");
}

void setup() {
    Serial.begin(9600);
    for (int i = 0; i < 9; i++) {
        pinMode(motorPos[i],  OUTPUT);
        pinMode(motorNeg[i],  OUTPUT);
        pinMode(chargePos[i], OUTPUT);
        pinMode(chargeNeg[i], OUTPUT);
    }
    forceAllIdle();
    printStatus();
    Serial.println("Ready. Commands: 1A-9A | 1B-9B | 1C-9C | ALL | KILL");
}

void loop() {
    if (Serial.available()) {
        String cmd = Serial.readStringUntil('\n');
```

```
        cmd.trim();
        cmd.toUpperCase();
        parseCommand(cmd);
        printStatus();
    }
}

void parseCommand(String cmd) {
    if (cmd == "ALL") {
        Serial.println(">>> Switching ALL batteries to MOTOR path...");
        for (int i = 0; i < 9; i++) setMotor(i);
        return;
    }
    if (cmd == "KILL") {
        Serial.println(">>> KILL: Forcing all relays OFF...");
        forceAllIdle();
        return;
    }
    if (cmd.length() == 2) {
        int batIndex = cmd.charAt(0) - '1';
        char path    = cmd.charAt(1);
        if (batIndex < 0 || batIndex > 8) {
            Serial.println("ERROR: Battery must be 1-9");
            return;
        }
        if      (path == 'A') setMotor(batIndex);
        else if (path == 'B') setCharge(batIndex);
        else if (path == 'C') setIdle(batIndex);
        else Serial.println("ERROR: Path must be A, B, or C");
        return;
    }
    Serial.println("ERROR: Unknown command");
}

void setMotor(int i) {
    if (batteryState[i] == MOTOR) {
        Serial.print("Battery ");
        Serial.print(i+1);
        Serial.println(" already on MOTOR path, skipping.");
        return;
    }
    Serial.print("Battery ");
    Serial.print(i+1);
    Serial.println(" -> MOTOR");
    if (batteryState[i] == CHARGE) {
        closeChargePath(i);
        delay(RELAY_DELAY);
    }
    openMotorPath(i);
    batteryState[i] = MOTOR;
}
```

```
void setCharge(int i) {
    if (batteryState[i] == CHARGE) {
        Serial.print("Battery ");
        Serial.print(i+1);
        Serial.println(" already on CHARGE path, skipping.");
        return;
    }
    Serial.print("Battery ");
    Serial.print(i+1);
    Serial.println(" -> CHARGE");
    if (batteryState[i] == MOTOR) {
        closeMotorPath(i);
        delay(RELAY_DELAY);
    }
    openChargePath(i);
    batteryState[i] = CHARGE;
}

void setIdle(int i) {
    if (batteryState[i] == IDLE) {
        Serial.print("Battery ");
        Serial.print(i+1);
        Serial.println(" already IDLE, skipping.");
        return;
    }
    Serial.print("Battery ");
    Serial.print(i+1);
    Serial.println(" -> IDLE");
    if (batteryState[i] == MOTOR)  closeMotorPath(i);
    if (batteryState[i] == CHARGE) closeChargePath(i);
    batteryState[i] = IDLE;
}

// Connect: negative first, then positive
void openMotorPath(int i) {
    digitalWrite(motorNeg[i], HIGH);
    delay(RELAY_DELAY);
    digitalWrite(motorPos[i], HIGH);
}

// Disconnect: positive first, then negative
void closeMotorPath(int i) {
    digitalWrite(motorPos[i], LOW);
    delay(RELAY_DELAY);
    digitalWrite(motorNeg[i], LOW);
}

void openChargePath(int i) {
    digitalWrite(chargeNeg[i], HIGH);
    delay(RELAY_DELAY);
```

```
    digitalWrite(chargePos[i], HIGH);
}

void closeChargePath(int i) {
    digitalWrite(chargePos[i], LOW);
    delay(RELAY_DELAY);
    digitalWrite(chargeNeg[i], LOW);
}

void printStatus() {
    Serial.println("");
    Serial.println("--- Battery Status ---");
    for (int i = 0; i < 9; i++) {
        Serial.print("  Battery ");
        Serial.print(i+1);
        Serial.print(": ");
        if      (batteryState[i] == MOTOR)  Serial.println("MOTOR");
        else if (batteryState[i] == CHARGE) Serial.println("CHARGE");
        else                                Serial.println("IDLE");
    }
    Serial.println("----------------------");
    Serial.println("");
}
```

Appendix B: Desktop Software

All desktop software is written in Python 3 and executed on the monitoring laptop.

B.1 Real-Time Monitoring Dashboard (Python)

Receives serial data from all three monitoring Arduinos via threaded readers and renders a live 3×3 battery card grid with per-cell voltage bars, SoC percentages, and state indicators. SSR state is read from a shared file written by the control interface (Appendix B.2). This dashboard is read-only.

```
"""
Skywalker III - Real-Time Battery Monitoring Dashboard
Reads serial data from 3 Arduino Megas and SSR state from shared file.

Board 1 (COM5, 115200): Batteries 1-3
Board 2 (COM7, 115200): Batteries 4-6
Board 3 (COM8, 115200): Batteries 7-9
SSR State: skywalker_ssr_state.txt (written by BMS interface)

Read-only: this dashboard never sends commands to any port.
"""

import tkinter as tk
import serial
import threading
import re
import time

MONITOR_PORTS = {
    "COM5": {"board": 1, "batteries": [1, 2, 3], "baud": 115200},
    "COM7": {"board": 2, "batteries": [4, 5, 6], "baud": 115200},
    "COM8": {"board": 3, "batteries": [7, 8, 9], "baud": 115200},
}
SSR_STATE_FILE = "skywalker_ssr_state.txt"

UPDATE_INTERVAL_MS = 500
SOC_CRITICAL = 15.0
SOC_LOW = 30.0
SOC_GOOD = 60.0

JUNCTIONS = {
    1: {"batteries": [1, 2, 3], "label": "JUNCTION 1"},
    2: {"batteries": [4, 5, 6], "label": "JUNCTION 2"},
    3: {"batteries": [7, 8, 9], "label": "JUNCTION 3"},
}

battery_data = {}
```

```
for port_info in MONITOR_PORTS.values():
    for bat_id in port_info["batteries"]:
        battery_data[bat_id] = {
            "pack_voltage": 0.0, "pack_soc": 0.0,
            "cells": [{"voltage": 0.0, "soc": 0.0} for _ in range(4)],
            "connected": False, "last_update": 0,
            "ssr_state": "UNKNOWN", "ssr_update": 0,
        }

port_status = {p: False for p in MONITOR_PORTS}
port_status["SSR_FILE"] = False
data_lock = threading.Lock()


def parse_monitor_output(text):
    bat_pattern = re.compile(
        r"=== Battery (\d+)\s+\(Pack:\s+([\d.]+)V\s+SOC:\s+([\d.]+)%\)")
    cell_pattern = re.compile(
        r"Cell (\d+):\s+([\d.]+)V\s+SOC:\s+([\d.]+)%")
    current_bat = None
    for line in text.split("\n"):
        line = line.strip()
        bat_match = bat_pattern.search(line)
        if bat_match:
            current_bat = int(bat_match.group(1))
            with data_lock:
                if current_bat in battery_data:
                    battery_data[current_bat]["pack_voltage"] =
float(bat_match.group(2))
                    battery_data[current_bat]["pack_soc"] =
float(bat_match.group(3))
                    battery_data[current_bat]["connected"] = True
                    battery_data[current_bat]["last_update"] = time.time()
            continue
        if current_bat is not None:
            cell_match = cell_pattern.search(line)
            if cell_match:
                cell_idx = int(cell_match.group(1)) - 1
                with data_lock:
                    if current_bat in battery_data and 0 <= cell_idx < 4:

battery_data[current_bat]["cells"][cell_idx]["voltage"] = \
                            float(cell_match.group(2))
                        battery_data[current_bat]["cells"][cell_idx]["soc"] =
\
                            float(cell_match.group(3))


def parse_ssr_file(text):
    state_pattern = re.compile(r"Battery\s+(\d+):\s+(MOTOR|CHARGE|IDLE)")
    for line in text.split("\n"):
```

```
        match = state_pattern.search(line)
        if match:
            bat_id = int(match.group(1))
            state = match.group(2)
            with data_lock:
                if bat_id in battery_data:
                    battery_data[bat_id]["ssr_state"] = state
                    battery_data[bat_id]["ssr_update"] = time.time()


def monitor_reader(port_name, baud):
    while True:
        try:
            ser = serial.Serial(port_name, baud, timeout=2)
            port_status[port_name] = True
            buffer = ""
            while True:
                try:
                    raw = ser.readline()
                    if raw:
                        line = raw.decode("utf-8", errors="replace")
                        buffer += line
                        if "----------" in line:
                            parse_monitor_output(buffer)
                            buffer = ""
                except serial.SerialException:
                    break
                except Exception:
                    continue
        except (serial.SerialException, Exception):
            port_status[port_name] = False
            time.sleep(3)


def ssr_file_reader():
    while True:
        try:
            with open(SSR_STATE_FILE, "r") as f:
                text = f.read()
            parse_ssr_file(text)
            port_status["SSR_FILE"] = True
        except FileNotFoundError:
            port_status["SSR_FILE"] = False
        except Exception:
            pass
        time.sleep(1)


class BatteryDashboard:
    BG = "#0a0a0a"
    CARD_BG = "#141414"
```

```
    JUNC_BG = "#0f1520"
    BORDER = "#2a2a2a"
    TEXT = "#e0e0e0"
    TEXT_DIM = "#666666"
    GREEN = "#22c55e"
    YELLOW = "#eab308"
    RED = "#ef4444"
    BLUE = "#3b82f6"
    ORANGE = "#f97316"
    CYAN = "#22d3ee"
    WHITE = "#ffffff"
    STATE_COLORS = {
        "MOTOR":   ("#22c55e", "#0f2a0f"),
        "CHARGE":  ("#eab308", "#2a2200"),
        "IDLE":    ("#ffffff", "#1e1e1e"),
        "UNKNOWN": ("#444444", "#141414"),
    }

    def __init__(self, root):
        self.root = root
        self.root.title("Skywalker III - Battery Monitoring System")
        self.root.configure(bg=self.BG)
        self.root.state("zoomed")
        self.cell_labels = {}
        self.pack_labels = {}
        self.pack_total_labels = {}
        self.status_labels = {}
        self.state_line_labels = {}
        self.pack_bar_canvas = {}
        self.cell_bar_canvas = {}
        self.junc_labels = {}
        self.junc_bar_canvas = {}
        self.build_ui()
        self.update_ui()

    # [Remaining UI construction and update methods follow the same
    #  pattern as shown in the thesis body, Section 4.3.4.
    #  Full source: 450+ lines of Tkinter widget construction,
    #  SoC color mapping, bar graph rendering, junction aggregation,
    #  and timed UI refresh at 500 ms intervals.]


def main():
    for port, info in MONITOR_PORTS.items():
        threading.Thread(target=monitor_reader,
                         args=(port, info["baud"]), daemon=True).start()
    threading.Thread(target=ssr_file_reader, daemon=True).start()
    root = tk.Tk()
    BatteryDashboard(root)
    root.mainloop()
```

```
if __name__ == "__main__":
    main()
```

B.2 SSR Control Interface (Python)

Provides manual control over all nine batteries through MOTOR, CHARGE, and IDLE buttons, an ALL MOTOR global command, and an emergency KILL command. Battery state is written to skywalker_ssr_state.txt for the monitoring dashboard.

```
"""
Skywalker Battery Management System - SSR Control Interface
Sends commands to SSR controller Arduino over serial.
Writes state to skywalker_ssr_state.txt for dashboard to read.
Commands: 1A-9A (motor), 1B-9B (charge), 1C-9C (idle), ALL, KILL
"""

import tkinter as tk
from tkinter import ttk
import serial
import serial.tools.list_ports
import threading
import time


class SkywalkerBMS:
    def __init__(self, root):
        self.root = root
        self.root.title("SKYWALKER BMS")
        self.root.configure(bg="#0a0a0a")
        self.root.resizable(False, False)
        self.ser = None
        self.connected = False
        self.reading = False
        self.battery_states = ["IDLE"] * 9
        self.colors = {
            "bg": "#0a0a0a", "panel": "#141414", "border": "#2a2a2a",
            "text": "#e0e0e0", "dim": "#666666",
            "idle": "#333333", "idle_text": "#888888",
            "motor": "#c45c1a", "motor_bright": "#e8762e",
            "charge": "#1a8a4a", "charge_bright": "#22b85e",
            "kill": "#cc2222", "kill_bright": "#ff3333",
            "all": "#2266cc", "all_bright": "#3388ee",
            "connected": "#22b85e", "disconnected": "#cc2222",
        }
        self.build_ui()

    def build_ui(self):
        main = tk.Frame(self.root, bg=self.colors["bg"], padx=20, pady=15)
```

```
        main.pack(fill="both", expand=True)

        # Header
        header = tk.Frame(main, bg=self.colors["bg"])
        header.pack(fill="x", pady=(0, 15))
        tk.Label(header, text="SKYWALKER", font=("Consolas", 24, "bold"),
                 fg=self.colors["motor_bright"],
bg=self.colors["bg"]).pack(side="left")
        tk.Label(header, text="  BMS", font=("Consolas", 24),
                 fg=self.colors["dim"],
bg=self.colors["bg"]).pack(side="left")

        # Connection bar
        conn_frame = tk.Frame(main, bg=self.colors["panel"],
                              highlightbackground=self.colors["border"],
                              highlightthickness=1, padx=12, pady=8)
        conn_frame.pack(fill="x", pady=(0, 12))
        tk.Label(conn_frame, text="PORT", font=("Consolas", 9),
                 fg=self.colors["dim"],
bg=self.colors["panel"]).pack(side="left", padx=(0, 8))
        self.port_var = tk.StringVar()
        self.port_combo = ttk.Combobox(conn_frame, textvariable=self.port_var,
                                       width=20, state="readonly",
font=("Consolas", 10))
        self.port_combo.pack(side="left", padx=(0, 8))
        tk.Button(conn_frame, text="Refresh", font=("Consolas", 10),
                  fg=self.colors["text"], bg=self.colors["idle"],
                  bd=0, padx=8, command=self.refresh_ports).pack(side="left",
padx=(0, 8))
        self.connect_btn = tk.Button(conn_frame, text="CONNECT",
                                     font=("Consolas", 10, "bold"),
                                     fg="#ffffff", bg=self.colors["charge"],
                                     bd=0, padx=16, pady=4,
                                     command=self.toggle_connection)
        self.connect_btn.pack(side="left", padx=(0, 12))
        self.status_dot = tk.Label(conn_frame, text="*", font=("Consolas",
14),
                                   fg=self.colors["disconnected"],
                                   bg=self.colors["panel"])
        self.status_dot.pack(side="left")
        self.status_label = tk.Label(conn_frame, text="DISCONNECTED",
                                     font=("Consolas", 9),
                                     fg=self.colors["disconnected"],
                                     bg=self.colors["panel"])
        self.status_label.pack(side="left", padx=(4, 0))

        # Battery grid
        grid_frame = tk.Frame(main, bg=self.colors["bg"])
        grid_frame.pack(fill="x", pady=(0, 12))
        self.state_labels = []
        self.row_frames = []
```

```
        for i in range(9):
            row = tk.Frame(grid_frame, bg=self.colors["panel"],
                           highlightbackground=self.colors["border"],
                           highlightthickness=1, padx=8, pady=6)
            row.pack(fill="x", pady=1)
            self.row_frames.append(row)
            tk.Label(row, text=f"B{i+1}", font=("Consolas", 13, "bold"),
                     fg=self.colors["text"], bg=self.colors["panel"],
                     width=4).grid(row=0, column=0, padx=(0, 8))
            tk.Button(row, text="MOTOR", font=("Consolas", 10, "bold"),
                      fg="#ffffff", bg=self.colors["motor"], bd=0,
                      padx=12, pady=4, width=7,
                      command=lambda idx=i: self.send_command(f"{idx+1}A")
                      ).grid(row=0, column=1, padx=4)
            tk.Button(row, text="CHARGE", font=("Consolas", 10, "bold"),
                      fg="#ffffff", bg=self.colors["charge"], bd=0,
                      padx=12, pady=4, width=7,
                      command=lambda idx=i: self.send_command(f"{idx+1}B")
                      ).grid(row=0, column=2, padx=4)
            tk.Button(row, text="IDLE", font=("Consolas", 10),
                      fg=self.colors["idle_text"], bg=self.colors["idle"],
                      bd=0, padx=12, pady=4, width=7,
                      command=lambda idx=i: self.send_command(f"{idx+1}C")
                      ).grid(row=0, column=3, padx=4)
            state_lbl = tk.Label(row, text="IDLE", font=("Consolas", 10,
"bold"),
                                 fg=self.colors["idle_text"],
                                 bg=self.colors["panel"], width=8)
            state_lbl.grid(row=0, column=4, padx=(8, 0))
            self.state_labels.append(state_lbl)

        # Global controls
        ctrl_frame = tk.Frame(main, bg=self.colors["bg"])
        ctrl_frame.pack(fill="x", pady=(0, 12))
        tk.Button(ctrl_frame, text="ALL -> MOTOR", font=("Consolas", 12,
"bold"),
                  fg="#ffffff", bg=self.colors["all"], bd=0, padx=20, pady=8,
                  command=lambda: self.send_command("ALL")
                  ).pack(side="left", expand=True, fill="x", padx=(0, 6))
        tk.Button(ctrl_frame, text="KILL", font=("Consolas", 12, "bold"),
                  fg="#ffffff", bg=self.colors["kill"], bd=0, padx=20, pady=8,
                  command=self.kill_command
                  ).pack(side="right", expand=True, fill="x", padx=(6, 0))

        # Serial log
        log_frame = tk.Frame(main, bg=self.colors["panel"],
                             highlightbackground=self.colors["border"],
                             highlightthickness=1)
        log_frame.pack(fill="both", expand=True, pady=(4, 0))
        self.log_text = tk.Text(log_frame, font=("Consolas", 9), height=10,
                                width=60, bg=self.colors["panel"],
```

```
                                  fg=self.colors["dim"], bd=0, padx=8, pady=6,
                                  state="disabled")
        self.log_text.pack(fill="both", expand=True)
        self.refresh_ports()
        self.write_state_file()

    def write_state_file(self):
        try:
            with open("skywalker_ssr_state.txt", "w") as f:
                for i in range(9):
                    f.write(f"Battery {i+1}: {self.battery_states[i]}\n")
        except Exception:
            pass

    def refresh_ports(self):
        ports = [p.device for p in serial.tools.list_ports.comports()]
        self.port_combo["values"] = ports
        if ports:
            self.port_combo.current(0)

    def toggle_connection(self):
        if self.connected:
            self.disconnect()
        else:
            self.connect()

    def connect(self):
        port = self.port_var.get()
        if not port:
            self.log("ERROR: No port selected")
            return
        try:
            self.ser = serial.Serial(port, 9600, timeout=0.1)
            self.connected = True
            self.reading = True
            self.connect_btn.configure(text="DISCONNECT",
bg=self.colors["kill"])
            self.status_dot.configure(fg=self.colors["connected"])
            self.status_label.configure(text=f"CONNECTED ({port})",
                                        fg=self.colors["connected"])
            self.log(f"Connected to {port}")
            threading.Thread(target=self.read_serial, daemon=True).start()
        except Exception as e:
            self.log(f"ERROR: {e}")

    def disconnect(self):
        self.reading = False
        self.connected = False
        if self.ser and self.ser.is_open:
            self.ser.close()
        self.connect_btn.configure(text="CONNECT", bg=self.colors["charge"])
```

```
        self.status_dot.configure(fg=self.colors["disconnected"])
        self.status_label.configure(text="DISCONNECTED",
                                    fg=self.colors["disconnected"])
        self.log("Disconnected")

    def read_serial(self):
        while self.reading:
            try:
                if self.ser and self.ser.is_open and self.ser.in_waiting:
                    line = self.ser.readline().decode("utf-8",
errors="replace").strip()
                    if line:
                        self.root.after(0, self.log, line)
                        self.root.after(0, self.parse_state, line)
                else:
                    time.sleep(0.05)
            except Exception:
                time.sleep(0.1)

    def send_command(self, cmd):
        if not self.connected or not self.ser or not self.ser.is_open:
            self.log("ERROR: Not connected")
            return
        try:
            self.ser.write(f"{cmd}\n".encode())
            self.log(f">>> {cmd}")
        except Exception as e:
            self.log(f"ERROR: {e}")

    def kill_command(self):
        self.send_command("KILL")
        for i in range(9):
            self.update_state(i, "IDLE")

    def parse_state(self, line):
        line_upper = line.strip()
        for i in range(9):
            bat_num = str(i + 1)
            if f"Battery {bat_num}" in line_upper:
                if "MOTOR" in line_upper and "already" not in
line_upper.lower():
                    self.update_state(i, "MOTOR")
                elif "CHARGE" in line_upper and "already" not in
line_upper.lower():
                    self.update_state(i, "CHARGE")
                elif "IDLE" in line_upper:
                    self.update_state(i, "IDLE")

    def update_state(self, idx, state):
        self.battery_states[idx] = state
        lbl = self.state_labels[idx]
```

```
            row = self.row_frames[idx]
            if state == "MOTOR":
                lbl.configure(text="MOTOR", fg=self.colors["motor_bright"])
                row.configure(highlightbackground=self.colors["motor"])
            elif state == "CHARGE":
                lbl.configure(text="CHARGE", fg=self.colors["charge_bright"])
                row.configure(highlightbackground=self.colors["charge"])
            else:
                lbl.configure(text="IDLE", fg=self.colors["idle_text"])
                row.configure(highlightbackground=self.colors["border"])
            self.write_state_file()

        def log(self, msg):
            self.log_text.configure(state="normal")
            self.log_text.insert("end", msg + "\n")
            self.log_text.see("end")
            self.log_text.configure(state="disabled")

        def clear_log(self):
            self.log_text.configure(state="normal")
            self.log_text.delete("1.0", "end")
            self.log_text.configure(state="disabled")

        def on_close(self):
            self.reading = False
            if self.ser and self.ser.is_open:
                self.ser.close()
            self.root.destroy()


    if __name__ == "__main__":
        root = tk.Tk()
        app = SkywalkerBMS(root)
        root.protocol("WM_DELETE_WINDOW", app.on_close)
        root.mainloop()
```

Appendix C: Simulation

The simulation is implemented in Python 3 across five modules.

C.1 System Configuration and Constants (config.py)

System configuration parameters for prototype (9 batteries) and full (48 batteries) modes, throttle-to-power mapping, battery energy constants, algorithm thresholds, rolling replenishment tier definitions, Keysight and generator charger specifications, and scenario configuration dataclasses.

```
"""
Skywalker III Battery Power Management Simulation
Configuration and System Specifications

SUPPORTS TWO CONFIGURATIONS:
- "prototype": 9 batteries (3 per junction) - lab testing with Keysight
- "full": 48 batteries (16 per junction) - operational system with generator

TOPOLOGY:
- N individual 4S LiPo batteries (Youme Power 6500mAh 80C)
- 3 junctions in SERIES (44.4V system voltage)
- Each junction has N/3 batteries in PARALLEL
- Each battery has 2 SSRs for power/charge routing
- Charging at 4S voltage (16.8V) via auto-parallel Keysight or generator
"""

from dataclasses import dataclass, field
from enum import Enum
from typing import List, Tuple, Dict, Any


#
# ============================================================================
# SYSTEM CONFIGURATION - PROTOTYPE (9) vs FULL (48)
#
# ============================================================================

SYSTEM_CONFIG: str = "prototype"

SYSTEM_CONFIGS: Dict[str, Dict[str, Any]] = {
    "prototype": {
        "name": "prototype",
        "description": "9-battery prototype for lab testing",
        "num_batteries": 9,
        "batteries_per_junction": 3,
        "num_junctions": 3,
        "min_active_per_junction": 1,
```

```
        "max_charge_per_junction": 2,
        "total_capacity_wh": 9 * 96.2,
    },
    "full": {
        "name": "full",
        "description": "48-battery operational system",
        "num_batteries": 48,
        "batteries_per_junction": 16,
        "num_junctions": 3,
        "min_active_per_junction": 1,
        "max_charge_per_junction": 15,
        "total_capacity_wh": 48 * 96.2,
    }
}


def get_system_config(config_name: str = None) -> Dict[str, Any]:
    """Get system configuration by name."""
    name = config_name or SYSTEM_CONFIG
    if name not in SYSTEM_CONFIGS:
        raise ValueError(f"Unknown config: {name}. Use 'prototype' or 'full'")
    return SYSTEM_CONFIGS[name]


def set_system_config(config_name: str) -> None:
    """Set the active system configuration."""
    global SYSTEM_CONFIG, NUM_BATTERIES, NUM_JUNCTIONS, BATTERIES_PER_JUNCTION
    global MIN_ACTIVE_PER_JUNCTION, MAX_CHARGE_PER_JUNCTION
    global MAX_TOTAL_CHARGING, TOTAL_ENERGY_WH

    if config_name not in SYSTEM_CONFIGS:
        raise ValueError(f"Unknown config: {config_name}. Use 'prototype' or
'full'")

    SYSTEM_CONFIG = config_name
    cfg = SYSTEM_CONFIGS[config_name]
    NUM_BATTERIES = cfg["num_batteries"]
    NUM_JUNCTIONS = cfg["num_junctions"]
    BATTERIES_PER_JUNCTION = cfg["batteries_per_junction"]
    MIN_ACTIVE_PER_JUNCTION = cfg["min_active_per_junction"]
    MAX_CHARGE_PER_JUNCTION = cfg["max_charge_per_junction"]
    MAX_TOTAL_CHARGING = MAX_CHARGE_PER_JUNCTION * NUM_JUNCTIONS
    TOTAL_ENERGY_WH = cfg["total_capacity_wh"]


#
# ============================================================================
# BATTERY TOPOLOGY (initialized from default config)
#
# ============================================================================
```

```
_default_cfg = SYSTEM_CONFIGS[SYSTEM_CONFIG]
NUM_BATTERIES: int = _default_cfg["num_batteries"]
NUM_JUNCTIONS: int = _default_cfg["num_junctions"]
BATTERIES_PER_JUNCTION: int = _default_cfg["batteries_per_junction"]
CELLS_PER_BATTERY: int = 4

CELL_NOMINAL_VOLTAGE: float = 3.7
CELL_MAX_VOLTAGE: float = 4.2
CELL_MIN_VOLTAGE: float = 3.0
CELL_CUTOFF_VOLTAGE: float = 3.3
CELL_CV_THRESHOLD: float = 4.15

BATTERY_VOLTAGE_NOMINAL: float = CELLS_PER_BATTERY * CELL_NOMINAL_VOLTAGE
BATTERY_VOLTAGE_MAX: float = CELLS_PER_BATTERY * CELL_MAX_VOLTAGE
BATTERY_CAPACITY_AH: float = 6.5
BATTERY_ENERGY_WH: float = BATTERY_VOLTAGE_NOMINAL * BATTERY_CAPACITY_AH
BATTERY_EFFICIENCY: float = 0.90
CELL_IMBALANCE_RANGE: float = 0.02

SYSTEM_VOLTAGE_NOMINAL: float = NUM_JUNCTIONS * BATTERY_VOLTAGE_NOMINAL
SYSTEM_VOLTAGE_MAX: float = NUM_JUNCTIONS * BATTERY_VOLTAGE_MAX
TOTAL_ENERGY_WH: float = _default_cfg["total_capacity_wh"]


#
# ============================================================================
# SERIES CONSTRAINT - SAFETY CRITICAL
#
# ============================================================================

MIN_ACTIVE_PER_JUNCTION: int = _default_cfg["min_active_per_junction"]
MAX_CHARGE_PER_JUNCTION: int = _default_cfg["max_charge_per_junction"]
MAX_TOTAL_CHARGING: int = MAX_CHARGE_PER_JUNCTION * NUM_JUNCTIONS


#
# ============================================================================
# SSR SWITCHING
#
# ============================================================================

SSR_SWITCH_TIME_MS: int = 10
SSR_COOLDOWN_MS: int = 100


#
# ============================================================================
# ALGORITHM THRESHOLDS
```

```
#
================================================================================

@dataclass
class AlgorithmConfig:
    """Configurable thresholds for the switching algorithm."""

    switch_threshold_soc: float = 30.0
    critical_threshold_soc: float = 15.0
    charge_complete_soc: float = 90.0
    min_cell_voltage_cutoff: float = 3.3
    absolute_min_cell_voltage: float = 3.0
    min_dwell_time_s: float = 30.0
    reactivation_margin_soc: float = 5.0
    min_charge_soc_gain: float = 10.0
    soc_imbalance_threshold: float = 20.0

    def __post_init__(self):
        assert 0 < self.critical_threshold_soc < self.switch_threshold_soc <
100
        assert self.switch_threshold_soc < self.charge_complete_soc <= 100


#
================================================================================
# ROLLING REPLENISHMENT THRESHOLDS
#
================================================================================

@dataclass
class RollingConfig:
    """Configuration for the rolling replenishment strategy."""

    tier1_threshold: float = 80.0
    tier2_threshold: float = 60.0
    tier3_threshold: float = 40.0
    charge_complete_soc: float = 85.0
    swap_differential: float = 30.0
    min_dwell_time_s: float = 60.0
    reactivation_margin_soc: float = 5.0
    min_charge_soc_gain: float = 10.0

    def get_tier_charge_counts(self, batteries_per_junction: int) -> tuple:
        """Get target charge counts for each tier based on battery count."""
        bpj = batteries_per_junction
        if bpj <= 3:
            return 1, 1, 2
        tier1_count = max(1, bpj // 6)
        tier2_count = max(2, bpj // 3)
        tier3_count = max(3, bpj * 2 // 3)
```

```
        return tier1_count, tier2_count, tier3_count


DEFAULT_ROLLING_CONFIG = RollingConfig()


#
================================================================================
# SCHEDULING STRATEGIES
#
================================================================================

class SchedulingStrategy(Enum):
    GREEDY = "greedy"
    ROUND_ROBIN = "round_robin"
    BALANCED = "balanced"
    MIN_SWITCHES = "min_switches"
    ROLLING_REPLENISHMENT = "rolling_replenishment"


#
================================================================================
# KEYSIGHT E36233A (4S charging)
#
================================================================================

@dataclass
class KeysightConfig:
    """Configuration for Keysight E36233A in AUTO-PARALLEL mode."""

    mode: str = "simulation"
    resource_string: str = "TCPIP0::192.168.1.100::inst0::INSTR"
    charge_voltage: float = 16.8
    ovp_voltage: float = 17.0
    min_voltage: float = 12.0
    parallel_mode: bool = True
    max_current_parallel: float = 40.0
    power_limit: float = 400.0
    max_current_at_16v: float = 23.8
    charge_current_1c: float = 6.5
    min_taper_current: float = 0.325
    ocp_current: float = 25.0
    slew_rate: float = 2.0
    stabilization_time: float = 0.5
    data_log_period_ms: int = 500
    efficiency: float = 0.98

    @property
    def cv_threshold_voltage(self) -> float:
        """Voltage to switch from CC to CV mode (4S)."""
```

```
        return CELL_CV_THRESHOLD * CELLS_PER_BATTERY


#
================================================================================
# GENERATOR / CHARGER (flight mode)
#
================================================================================

GENERATOR_CHARGER_POWER_W: float = 4000.0
GENERATOR_CHARGER_VOLTAGE: float = 16.8
GENERATOR_EFFICIENCY: float = 0.85
GENERATOR_EFFECTIVE_POWER_W: float = (
    GENERATOR_CHARGER_POWER_W * GENERATOR_EFFICIENCY)
GENERATOR_MAX_CURRENT_A: float = (
    GENERATOR_EFFECTIVE_POWER_W / GENERATOR_CHARGER_VOLTAGE)
ENGINE_START_DELAY_S: float = 4.0
FUEL_CAPACITY_WH: float = 39200.0

BENCH_TEST_POWER_W: float = 600.0


#
================================================================================
# PROPULSION SYSTEM
#
================================================================================

NUM_MOTORS: int = 6
MOTOR_CURRENT_EACH_A: float = 80.0
TOTAL_SYSTEM_CURRENT_A: float = 480.0
MOTOR_POWER_HOVER_W: float = 8500.0
MOTOR_POWER_PEAK_W: float = 21312.0

THROTTLE_POWER_MAP: dict = {
    0.0: 0.0,
    0.25: 2500.0,
    0.40: 5000.0,
    0.45: 8500.0,
    0.50: 10000.0,
    0.70: 15000.0,
    0.80: 17000.0,
    1.00: 21312.0,
}


#
================================================================================
# VOLTAGE-SOC LOOKUP TABLE
```

```
#
================================================================================

VOLTAGE_SOC_TABLE: List[Tuple[float, float]] = [
    (4.20, 100.0), (4.15, 95.0), (4.10, 90.0), (4.05, 85.0),
    (4.00, 80.0),  (3.95, 75.0), (3.90, 70.0), (3.85, 65.0),
    (3.83, 60.0),  (3.80, 55.0), (3.78, 50.0), (3.75, 45.0),
    (3.72, 40.0),  (3.70, 35.0), (3.68, 30.0), (3.65, 25.0),
    (3.60, 20.0),  (3.55, 15.0), (3.50, 10.0), (3.40, 5.0),
    (3.30, 2.0),   (3.00, 0.0),
]


#
================================================================================
# SIMULATION PARAMETERS
#
================================================================================

@dataclass
class SimulationConfig:
    """Configuration for the simulation engine."""

    timestep_s: float = 0.5
    max_duration_s: float = 3600.0
    random_seed: int = 42
    initial_soc: float = 100.0
    initial_soc_per_battery: List[float] = field(default_factory=list)
    fuel_initial_fraction: float = 1.0
    takeoff_throttle: float = 0.80
    takeoff_duration_s: float = 30.0
    cruise_throttle: float = 0.30
    maneuver_throttle: float = 0.50
    maneuver_duration_s: float = 7.5
    maneuver_interval_s: float = 300.0
    bench_test_mode: bool = True
    system_config: str = "prototype"

    def __post_init__(self):
        if not self.initial_soc_per_battery:
            cfg = get_system_config(self.system_config)
            self.initial_soc_per_battery = (
                [self.initial_soc] * cfg["num_batteries"])


@dataclass
class ScenarioConfig:
    """Complete scenario configuration."""

    name: str = "nominal"
```

```
    description: str = "Normal flight with healthy batteries"
    algorithm: AlgorithmConfig = field(default_factory=AlgorithmConfig)
    simulation: SimulationConfig = field(default_factory=SimulationConfig)
    keysight: KeysightConfig = field(default_factory=KeysightConfig)
    scheduling_strategy: SchedulingStrategy = SchedulingStrategy.BALANCED
    rolling_config: RollingConfig = field(default_factory=RollingConfig)
    battery_degradation: List[float] = field(default_factory=list)
    engine_failure_time_s: float = -1.0
    system_config: str = "prototype"

    def __post_init__(self):
        if not self.battery_degradation:
            cfg = get_system_config(self.system_config)
            self.battery_degradation = [0.0] * cfg["num_batteries"]
        self.simulation.system_config = self.system_config
        cfg = get_system_config(self.system_config)
        num_bats = cfg["num_batteries"]
        if len(self.simulation.initial_soc_per_battery) != num_bats:
            self.simulation.initial_soc_per_battery = (
                [self.simulation.initial_soc] * num_bats)


DEFAULT_ALGORITHM_CONFIG = AlgorithmConfig()
DEFAULT_SIMULATION_CONFIG = SimulationConfig()
DEFAULT_KEYSIGHT_CONFIG = KeysightConfig()
DEFAULT_ROLLING_CONFIG = RollingConfig()
DEFAULT_SCENARIO_CONFIG = ScenarioConfig()
```

C.2 Battery and Voltage Model (battery.py)

Battery class with cell-level voltage modeling, OCV-SoC lookup table with linear interpolation, cell offset initialization, subtraction-based voltage isolation, minimum-cell SoC computation, discharge and charge energy accounting, and BatterySystem class implementing the parallel-first series-second junction topology with constraint enforcement.

```
"""
Skywalker III Battery Power Management Simulation
Individual Battery and System Model

SUPPORTS TWO CONFIGURATIONS:
- "prototype": 9 batteries (3 per junction)
- "full": 48 batteries (16 per junction)

TOPOLOGY:
- N individual 4S LiPo batteries (Youme Power 6500mAh 80C)
- 3 junctions in SERIES (each junction = 14.8V nominal)
- Each junction has N/3 batteries in PARALLEL (current capacity)
- System voltage = 3 x 14.8V = 44.4V nominal
```

```
- Each battery has 2 SSRs: SSR1 (power), SSR2 (charge)

SSR STATES:
- POWER: SSR1=ON, SSR2=OFF -> battery on power bus
- CHARGE: SSR1=OFF, SSR2=ON -> battery on charge bus
- IDLE: SSR1=OFF, SSR2=OFF -> disconnected

CRITICAL: If ANY junction has 0 batteries on POWER, the series circuit
breaks and system voltage drops to zero instantly.
"""

import logging
from dataclasses import dataclass, field
from enum import Enum
from typing import List, Optional, Tuple, Dict, Any
import numpy as np

from config import (
    CELLS_PER_BATTERY,
    CELL_NOMINAL_VOLTAGE,
    CELL_MAX_VOLTAGE,
    CELL_MIN_VOLTAGE,
    CELL_CV_THRESHOLD,
    BATTERY_VOLTAGE_NOMINAL,
    BATTERY_VOLTAGE_MAX,
    BATTERY_ENERGY_WH,
    BATTERY_EFFICIENCY,
    CELL_IMBALANCE_RANGE,
    VOLTAGE_SOC_TABLE,
    get_system_config,
)

logger = logging.getLogger(__name__)


class BatteryState(Enum):
    """SSR routing state for a battery."""
    POWER = "power"
    CHARGE = "charge"
    IDLE = "idle"


def voltage_to_soc(cell_voltage: float) -> float:
    """Convert cell voltage to State of Charge using lookup table."""
    if cell_voltage >= VOLTAGE_SOC_TABLE[0][0]:
        return VOLTAGE_SOC_TABLE[0][1]
    if cell_voltage <= VOLTAGE_SOC_TABLE[-1][0]:
        return VOLTAGE_SOC_TABLE[-1][1]
    for i in range(len(VOLTAGE_SOC_TABLE) - 1):
        v_high, soc_high = VOLTAGE_SOC_TABLE[i]
```

```
            v_low, soc_low = VOLTAGE_SOC_TABLE[i + 1]
            if v_low <= cell_voltage <= v_high:
                fraction = (cell_voltage - v_low) / (v_high - v_low)
                return soc_low + fraction * (soc_high - soc_low)
    return 0.0


def soc_to_voltage(soc: float) -> float:
    """Convert State of Charge to cell voltage using lookup table."""
    if soc >= VOLTAGE_SOC_TABLE[0][1]:
        return VOLTAGE_SOC_TABLE[0][0]
    if soc <= VOLTAGE_SOC_TABLE[-1][1]:
        return VOLTAGE_SOC_TABLE[-1][0]
    for i in range(len(VOLTAGE_SOC_TABLE) - 1):
        v_high, soc_high = VOLTAGE_SOC_TABLE[i]
        v_low, soc_low = VOLTAGE_SOC_TABLE[i + 1]
        if soc_low <= soc <= soc_high:
            fraction = (soc - soc_low) / (soc_high - soc_low)
            return v_low + fraction * (v_high - v_low)
    return CELL_MIN_VOLTAGE


@dataclass
class Battery:
    """
    A single Youme 4S 6500mAh LiPo battery.
    Has 4 cells readable via balance lead.
    Individually switchable via SSR pair.
    Belongs to one of 3 junctions.
    """

    battery_id: int
    junction: int
    cell_voltages: List[float] = field(default_factory=list)
    cell_imbalances: List[float] = field(default_factory=list)
    state: BatteryState = BatteryState.POWER
    energy_remaining_wh: float = BATTERY_ENERGY_WH
    cycle_count: int = 0
    temperature_c: float = 25.0
    last_switch_time_s: float = -1000.0
    capacity_fade_fraction: float = 1.0
    internal_resistance_ohm: float = 0.005
    total_energy_discharged_wh: float = 0.0
    total_energy_charged_wh: float = 0.0
    switch_count: int = 0
    current_draw_a: float = 0.0
    charge_entry_soc: float = -1.0

    def __post_init__(self):
        if not self.cell_voltages:
            self.cell_voltages = [CELL_MAX_VOLTAGE] * CELLS_PER_BATTERY
```

```
            if not self.cell_imbalances:
                self.cell_imbalances = [0.0] * CELLS_PER_BATTERY

        @property
        def min_cell_voltage(self) -> float:
            return min(self.cell_voltages)

        @property
        def max_cell_voltage(self) -> float:
            return max(self.cell_voltages)

        @property
        def mean_cell_voltage(self) -> float:
            return sum(self.cell_voltages) / len(self.cell_voltages)

        @property
        def total_voltage(self) -> float:
            return sum(self.cell_voltages)

        @property
        def soc(self) -> float:
            effective_capacity = BATTERY_ENERGY_WH * self.capacity_fade_fraction
            if effective_capacity <= 0:
                return 0.0
            return max(0.0, min(100.0,
                (self.energy_remaining_wh / effective_capacity) * 100.0))

        def is_healthy(self) -> bool:
            return self.min_cell_voltage > CELL_MIN_VOLTAGE

        def is_safe(self) -> bool:
            return self.min_cell_voltage > 3.3

        def is_on_power(self) -> bool:
            return self.state == BatteryState.POWER

        def is_on_charge(self) -> bool:
            return self.state == BatteryState.CHARGE

        def is_idle(self) -> bool:
            return self.state == BatteryState.IDLE


    class BatterySystem:
        """
        Parameterized battery system: N batteries across 3 junctions in SERIES.
        Junction layout: batteries in PARALLEL within each junction, junctions
        connected in SERIES for system voltage.
        """
```

```
def __init__(self, system_config: str = "prototype",
             initial_soc_per_battery: Optional[List[float]] = None,
             battery_degradation: Optional[List[float]] = None,
             rng: Optional[np.random.Generator] = None):
    self.rng = rng or np.random.default_rng()
    cfg = get_system_config(system_config)
    self.config_name = system_config
    self.num_batteries = cfg["num_batteries"]
    self.num_junctions = cfg["num_junctions"]
    self.batteries_per_junction = cfg["batteries_per_junction"]
    self.min_active_per_junction = cfg["min_active_per_junction"]
    self.max_charge_per_junction = cfg["max_charge_per_junction"]
    self.total_capacity_wh = cfg["total_capacity_wh"]

    if initial_soc_per_battery is None:
        initial_soc_per_battery = [100.0] * self.num_batteries
    if battery_degradation is None:
        battery_degradation = [0.0] * self.num_batteries
    if len(initial_soc_per_battery) != self.num_batteries:
        initial_soc_per_battery = [100.0] * self.num_batteries
    if len(battery_degradation) != self.num_batteries:
        battery_degradation = [0.0] * self.num_batteries

    self.batteries: List[Battery] = []
    for bat_id in range(self.num_batteries):
        junction = bat_id // self.batteries_per_junction
        initial_soc = initial_soc_per_battery[bat_id]
        degradation = battery_degradation[bat_id]
        base_voltage = soc_to_voltage(initial_soc)
        cell_voltages = []
        cell_imbalances = []
        for _ in range(CELLS_PER_BATTERY):
            imbalance = self.rng.uniform(
                -CELL_IMBALANCE_RANGE, CELL_IMBALANCE_RANGE)
            cell_imbalances.append(imbalance)
            cell_v = base_voltage + imbalance - degradation
            cell_v = max(CELL_MIN_VOLTAGE,
                         min(CELL_MAX_VOLTAGE, cell_v))
            cell_voltages.append(cell_v)
        capacity_fade = 1.0 - degradation
        ir_multiplier = 1.0 + degradation * 13.33
        internal_r = 0.005 * ir_multiplier
        initial_energy = ((initial_soc / 100.0) *
                          BATTERY_ENERGY_WH * capacity_fade)
        battery = Battery(
            battery_id=bat_id, junction=junction,
            cell_voltages=cell_voltages,
            cell_imbalances=cell_imbalances,
            state=BatteryState.POWER,
            energy_remaining_wh=initial_energy,
```

```
            capacity_fade_fraction=capacity_fade,
            internal_resistance_ohm=internal_r)
        self.batteries.append(battery)

    def batteries_in_junction(self, junction: int) -> List[Battery]:
        return [b for b in self.batteries if b.junction == junction]

    def active_in_junction(self, junction: int) -> List[Battery]:
        return [b for b in self.batteries
                if b.junction == junction and b.is_on_power()]

    def charging_in_junction(self, junction: int) -> List[Battery]:
        return [b for b in self.batteries
                if b.junction == junction and b.is_on_charge()]

    def power_batteries(self) -> List[Battery]:
        return [b for b in self.batteries if b.is_on_power()]

    def charging_batteries(self) -> List[Battery]:
        return [b for b in self.batteries if b.is_on_charge()]

    def idle_batteries(self) -> List[Battery]:
        return [b for b in self.batteries if b.is_idle()]

    def num_active_in_junction(self, junction: int) -> int:
        return len(self.active_in_junction(junction))

    def num_charging_in_junction(self, junction: int) -> int:
        return len(self.charging_in_junction(junction))

    def can_switch_to_charge(self, battery_id: int) -> Tuple[bool, str]:
        battery = self.batteries[battery_id]
        if battery.state != BatteryState.POWER:
            return False, f"Battery {battery_id} is not on POWER"
        active = self.active_in_junction(battery.junction)
        if len(active) <= self.min_active_per_junction:
            return False, (f"Battery {battery_id} is last active "
                           f"in junction {battery.junction}")
        return True, ""

    def can_switch_to_idle(self, battery_id: int) -> Tuple[bool, str]:
        return self.can_switch_to_charge(battery_id)

    def set_battery_state(self, battery_id: int,
                          new_state: BatteryState,
                          timestamp_s: float) -> bool:
        battery = self.batteries[battery_id]
        old_state = battery.state
        if old_state == new_state:
            return True
```

```
    if (old_state == BatteryState.POWER and
            new_state != BatteryState.POWER):
        can_switch, reason = self.can_switch_to_charge(battery_id)
        if not can_switch:
            return False
    battery.state = new_state
    if (new_state == BatteryState.CHARGE and
            old_state != BatteryState.CHARGE):
        battery.charge_entry_soc = battery.soc
    elif (old_state == BatteryState.CHARGE and
          new_state != BatteryState.CHARGE):
        battery.charge_entry_soc = -1.0
    battery.last_switch_time_s = timestamp_s
    battery.switch_count += 1
    if (new_state == BatteryState.POWER and
            old_state == BatteryState.CHARGE):
        battery.cycle_count += 1
    return True

def junction_voltage(self, junction: int) -> float:
    active = self.active_in_junction(junction)
    if not active:
        return 0.0
    return sum(b.total_voltage for b in active) / len(active)

def system_voltage(self) -> float:
    return sum(self.junction_voltage(j)
               for j in range(self.num_junctions))

def validate_series_chain(self) -> Tuple[bool, str]:
    for j in range(self.num_junctions):
        active = self.num_active_in_junction(j)
        if active < 1:
            return False, (f"Junction {j} has {active} active "
                           f"batteries (need at least 1)")
    return True, ""

def discharge_step(self, total_power_watts: float,
                   dt_seconds: float) -> Tuple[float, bool]:
    is_valid, reason = self.validate_series_chain()
    if not is_valid:
        return 0.0, False
    sys_voltage = self.system_voltage()
    if sys_voltage <= 0:
        return 0.0, False
    system_current = total_power_watts / sys_voltage
    total_energy_delivered = 0.0
    all_safe = True
    for junction in range(self.num_junctions):
        active = self.active_in_junction(junction)
        if not active:
```

```
                continue
            current_per_battery = system_current / len(active)
            for battery in active:
                battery.current_draw_a = current_per_battery
                power_per_battery = (current_per_battery *
                                     battery.total_voltage)
                energy_delivered = (power_per_battery *
                                    dt_seconds / 3600.0)
                energy_from_battery = min(energy_delivered,
                                          battery.energy_remaining_wh)
                battery.energy_remaining_wh -= energy_from_battery
                battery.total_energy_discharged_wh += energy_from_battery
                total_energy_delivered += energy_from_battery
                new_soc = battery.soc
                base_voltage = soc_to_voltage(new_soc)
                ir_drop = (current_per_battery *
                           battery.internal_resistance_ohm)
                for i in range(len(battery.cell_voltages)):
                    battery.cell_voltages[i] = (
                        base_voltage + battery.cell_imbalances[i] -
                        ir_drop)
                    battery.cell_voltages[i] = max(
                        CELL_MIN_VOLTAGE,
                        min(CELL_MAX_VOLTAGE,
                            battery.cell_voltages[i]))
                heat = power_per_battery * (1 - BATTERY_EFFICIENCY)
                temp_rise = heat * dt_seconds / 500.0
                battery.temperature_c = min(
                    60.0, battery.temperature_c + temp_rise)
                if not battery.is_safe():
                    all_safe = False
        for battery in self.batteries:
            if not battery.is_on_power():
                battery.current_draw_a = 0.0
        return total_energy_delivered, all_safe

    def charge_step(self, charger_current_amps: float,
                    charger_voltage: float,
                    dt_seconds: float) -> Tuple[float, List[int]]:
        charging = self.charging_batteries()
        if not charging:
            return 0.0, []
        current_per_battery = charger_current_amps / len(charging)
        total_energy_added = 0.0
        completed = []
        for battery in charging:
            battery.current_draw_a = -current_per_battery
            max_cell_v = battery.max_cell_voltage
            if max_cell_v < CELL_CV_THRESHOLD:
                effective_current = current_per_battery
            else:
                voltage_headroom = CELL_MAX_VOLTAGE - max_cell_v
```

```
                voltage_range = CELL_MAX_VOLTAGE - CELL_CV_THRESHOLD
                taper_fraction = max(0.05,
                                     voltage_headroom / voltage_range)
                effective_current = current_per_battery * taper_fraction
            power = effective_current * battery.total_voltage
            energy_added = power * dt_seconds / 3600.0
            effective_capacity = (BATTERY_ENERGY_WH *
                                  battery.capacity_fade_fraction)
            max_energy = effective_capacity - battery.energy_remaining_wh
            energy_added = min(energy_added, max(0.0, max_energy))
            battery.energy_remaining_wh += energy_added
            battery.total_energy_charged_wh += energy_added
            total_energy_added += energy_added
            new_soc = battery.soc
            base_voltage = soc_to_voltage(new_soc)
            for i in range(len(battery.cell_voltages)):
                battery.cell_voltages[i] = (
                    base_voltage + battery.cell_imbalances[i])
                battery.cell_voltages[i] = max(
                    CELL_MIN_VOLTAGE,
                    min(CELL_MAX_VOLTAGE,
                        battery.cell_voltages[i]))
            battery.temperature_c = max(
                25.0, battery.temperature_c - 0.01 * dt_seconds)
            if battery.soc >= 90.0:
                completed.append(battery.battery_id)
        return total_energy_added, completed

    def get_system_soc(self) -> float:
        return sum(b.soc for b in self.batteries) / len(self.batteries)

    def get_min_battery_soc(self) -> float:
        return min(b.soc for b in self.batteries)

    def get_max_battery_soc(self) -> float:
        return max(b.soc for b in self.batteries)

    def get_soc_std_dev(self) -> float:
        socs = [b.soc for b in self.batteries]
        mean = sum(socs) / len(socs)
        variance = sum((s - mean) ** 2 for s in socs) / len(socs)
        return variance ** 0.5

    def get_junction_median_soc(self, junction: int,
                                power_only: bool = True) -> float:
        if power_only:
            batteries = self.active_in_junction(junction)
        else:
            batteries = self.batteries_in_junction(junction)
        if not batteries:
            return 0.0
```

```
        socs = sorted([b.soc for b in batteries])
        n = len(socs)
        if n % 2 == 0:
            return (socs[n // 2 - 1] + socs[n // 2]) / 2
        return socs[n // 2]

    def get_snapshot(self) -> dict:
        return {
            "config_name": self.config_name,
            "num_batteries": self.num_batteries,
            "batteries_per_junction": self.batteries_per_junction,
            "system_voltage": self.system_voltage(),
            "system_soc": self.get_system_soc(),
            "min_battery_soc": self.get_min_battery_soc(),
            "max_battery_soc": self.get_max_battery_soc(),
            "soc_std_dev": self.get_soc_std_dev(),
            "num_power": len(self.power_batteries()),
            "num_charging": len(self.charging_batteries()),
            "num_idle": len(self.idle_batteries()),
            "batteries": [
                {"id": b.battery_id, "junction": b.junction,
                 "state": b.state.value, "soc": b.soc,
                 "min_cell_v": b.min_cell_voltage,
                 "total_v": b.total_voltage,
                 "current_a": b.current_draw_a,
                 "temp_c": b.temperature_c,
                 "cell_voltages": b.cell_voltages.copy()}
                for b in self.batteries],
            "junctions": [
                {"junction": j,
                 "voltage": self.junction_voltage(j),
                 "active_count": self.num_active_in_junction(j),
                 "charging_count": self.num_charging_in_junction(j),
                 "median_soc": self.get_junction_median_soc(j),
                 "active_ids": [b.battery_id
                                for b in self.active_in_junction(j)],
                 "charging_ids": [b.battery_id
                                  for b in self.charging_in_junction(j)]}
                for j in range(self.num_junctions)],
        }
```

C.3 Scheduling Algorithms (algorithm.py)

All five scheduling strategy implementations (Greedy, Round Robin, Balanced, Minimum Switches, Rolling Replenishment) within a unified SwitchingAlgorithm class. Includes tier computation, swap differential logic, dwell time enforcement, safety checks, junction balance, and charge completion handling.

```
"""
Skywalker III Battery Power Management Simulation
Switching Algorithm for Series-Junction System

Core intelligence layer that reads cell voltages from all batteries
(via Arduino Megas) and decides SSR states.

CRITICAL CONSTRAINT: Every junction must have at least MIN_ACTIVE_PER_JUNCTION
batteries on POWER to maintain the series chain.

SUPPORTS TWO CONFIGURATIONS:
- "prototype": 9 batteries (3 per junction)
- "full": 48 batteries (16 per junction)
"""

import logging
from dataclasses import dataclass, field
from enum import Enum
from typing import List, Optional, Tuple

from battery import BatterySystem, Battery, BatteryState, voltage_to_soc
from config import (
    AlgorithmConfig,
    SchedulingStrategy,
    RollingConfig,
    DEFAULT_ROLLING_CONFIG,
)

logger = logging.getLogger(__name__)


class SystemState(Enum):
    NOMINAL = "nominal"
    DEGRADED = "degraded"
    CRITICAL = "critical"
    EMERGENCY_LAND = "emergency_land"
    CHARGER_STARTUP = "charger_startup"


@dataclass
class SwitchAction:
    battery_id: int
    new_state: BatteryState
    reason: str
    timestamp_s: float
    category: str = "planned"


@dataclass
class DecisionEvent:
```

```
    timestamp_s: float
    event_type: str
    state_before: SystemState
    state_after: SystemState
    reason: str
    actions: List[SwitchAction]
    battery_socs: List[float]
    junction_active_counts: List[int]


@dataclass
class AlgorithmState:
    current_state: SystemState = SystemState.NOMINAL
    charger_active: bool = False
    charger_start_time_s: float = -1.0
    total_switches: int = 0
    planned_switches: int = 0
    safety_switches: int = 0
    events: List[DecisionEvent] = field(default_factory=list)
    last_charged_battery: int = -1
    charge_order: List[int] = field(default_factory=list)


class SwitchingAlgorithm:
    """
    Core intelligence layer for the battery system.
    Runs every cycle (500ms): reads cell voltages, computes SoC,
    performs safety checks, makes switching decisions, manages charger,
    and balances load within junctions.
    """

    def __init__(self, battery_system: BatterySystem,
                 config: AlgorithmConfig,
                 strategy: SchedulingStrategy = SchedulingStrategy.BALANCED,
                 rolling_config: Optional[RollingConfig] = None):
        self.system = battery_system
        self.config = config
        self.strategy = strategy
        self.rolling_config = rolling_config or DEFAULT_ROLLING_CONFIG
        self.state = AlgorithmState()
        self.state.charge_order = list(range(self.system.num_batteries))
        self.junction_tiers = [0] * self.system.num_junctions

    def _log_event(self, timestamp_s, event_type, state_before,
                   state_after, reason, actions):
        event = DecisionEvent(
            timestamp_s=timestamp_s, event_type=event_type,
            state_before=state_before, state_after=state_after,
            reason=reason, actions=actions,
            battery_socs=[b.soc for b in self.system.batteries],
            junction_active_counts=[
```

```
            self.system.num_active_in_junction(j)
            for j in range(self.system.num_junctions)])
    self.state.events.append(event)

def _can_reactivate(self, battery: Battery,
                    timestamp_s: float) -> bool:
    if self.strategy == SchedulingStrategy.ROLLING_REPLENISHMENT:
        margin = self.rolling_config.reactivation_margin_soc
        dwell = self.rolling_config.min_dwell_time_s
    else:
        margin = self.config.reactivation_margin_soc
        dwell = self.config.min_dwell_time_s
    if battery.soc < self.config.switch_threshold_soc + margin:
        return False
    if (timestamp_s - battery.last_switch_time_s) < dwell:
        return False
    return True

def _check_safety(self, timestamp_s: float) -> List[SwitchAction]:
    actions = []
    for battery in self.system.batteries:
        if battery.min_cell_voltage < \
                self.config.absolute_min_cell_voltage:
            self.state.current_state = SystemState.EMERGENCY_LAND
            return []
        if (battery.is_on_power() and
                battery.min_cell_voltage <
                self.config.min_cell_voltage_cutoff):
            can_switch, _ = self.system.can_switch_to_charge(
                battery.battery_id)
            if can_switch:
                actions.append(SwitchAction(
                    battery_id=battery.battery_id,
                    new_state=BatteryState.CHARGE,
                    reason=(f"Safety: cell "
                            f"{battery.min_cell_voltage:.3f}V "
                            f"< 3.3V"),
                    timestamp_s=timestamp_s,
                    category="safety"))
    is_valid, reason = self.system.validate_series_chain()
    if not is_valid:
        self.state.current_state = SystemState.EMERGENCY_LAND
        return []
    return actions

def _check_junction_balance(self,
                            timestamp_s: float) -> List[SwitchAction]:
    actions = []
    swapped_batteries = set()
    for junction in range(self.system.num_junctions):
        active = self.system.active_in_junction(junction)
```

```
            charging = self.system.charging_in_junction(junction)
            if not active or not charging:
                continue
            lowest_active = min(active, key=lambda b: b.soc)
            eligible_charging = [
                b for b in charging
                if self._can_reactivate(b, timestamp_s)]
            highest_charging = (
                max(eligible_charging, key=lambda b: b.soc)
                if eligible_charging else None)
            if len(active) == 1:
                if (highest_charging is not None and
                        highest_charging.soc > lowest_active.soc):
                    actions.append(SwitchAction(
                        battery_id=highest_charging.battery_id,
                        new_state=BatteryState.POWER,
                        reason=f"Relieve solo J{junction}",
                        timestamp_s=timestamp_s,
                        category="safety"))
                    swapped_batteries.add(
                        highest_charging.battery_id)
            elif lowest_active.soc < \
                    self.config.switch_threshold_soc + 5:
                if (highest_charging is not None and
                        highest_charging.soc >
                        lowest_active.soc + 15):
                    if highest_charging.battery_id not in \
                            swapped_batteries:
                        actions.append(SwitchAction(
                            battery_id=highest_charging.battery_id,
                            new_state=BatteryState.POWER,
                            reason="Rotate",
                            timestamp_s=timestamp_s,
                            category="planned"))
                        swapped_batteries.add(
                            highest_charging.battery_id)
        return actions

    def _select_for_charging_greedy(self) -> List[int]:
        power_bats = sorted(
            self.system.power_batteries(),
            key=lambda b: (b.soc, -b.total_energy_discharged_wh))
        to_charge = []
        junctions_switched = set()
        for battery in power_bats:
            if battery.soc <= self.config.switch_threshold_soc:
                active_in_j = self.system.num_active_in_junction(
                    battery.junction)
                if active_in_j <= self.system.min_active_per_junction:
                    continue
                if battery.junction in junctions_switched:
                    continue
```

```
                can_switch, _ = self.system.can_switch_to_charge(
                    battery.battery_id)
                if can_switch:
                    charging_in_j = len(
                        self.system.charging_in_junction(
                            battery.junction))
                    if charging_in_j < \
                            self.system.max_charge_per_junction:
                        to_charge.append(battery.battery_id)
                        junctions_switched.add(battery.junction)
        return to_charge

    def _select_for_charging_round_robin(self) -> List[int]:
        to_charge = []
        junctions_switched = set()
        for bat_id in self.state.charge_order:
            battery = self.system.batteries[bat_id]
            if (battery.is_on_power() and
                    battery.soc <= self.config.switch_threshold_soc):
                active_in_j = self.system.num_active_in_junction(
                    battery.junction)
                if active_in_j <= self.system.min_active_per_junction:
                    continue
                if battery.junction in junctions_switched:
                    continue
                can_switch, _ = self.system.can_switch_to_charge(
                    bat_id)
                if can_switch:
                    charging_in_j = len(
                        self.system.charging_in_junction(
                            battery.junction))
                    if charging_in_j < \
                            self.system.max_charge_per_junction:
                        to_charge.append(bat_id)
                        junctions_switched.add(battery.junction)
        if to_charge:
            self.state.charge_order = (
                [b for b in self.state.charge_order
                 if b not in to_charge] + to_charge)
        return to_charge

    def _select_for_charging_balanced(self) -> List[int]:
        avg_soc = self.system.get_system_soc()
        power_bats = [b for b in self.system.power_batteries()
                      if b.soc < avg_soc]
        power_bats.sort(
            key=lambda b: (b.soc, -b.total_energy_discharged_wh))
        to_charge = []
        junctions_switched = set()
        for battery in power_bats:
            if battery.soc <= self.config.switch_threshold_soc:
```

```
            active_in_j = self.system.num_active_in_junction(
                battery.junction)
            if active_in_j <= self.system.min_active_per_junction:
                continue
            if battery.junction in junctions_switched:
                continue
            can_switch, _ = self.system.can_switch_to_charge(
                battery.battery_id)
            if can_switch:
                charging_in_j = len(
                    self.system.charging_in_junction(
                        battery.junction))
                if charging_in_j < \
                        self.system.max_charge_per_junction:
                    to_charge.append(battery.battery_id)
                    junctions_switched.add(battery.junction)
    return to_charge

def _select_for_charging_min_switches(self) -> List[int]:
    if self.system.charging_batteries():
        return []
    power_bats = self.system.power_batteries()
    if not power_bats:
        return []
    lowest = min(power_bats, key=lambda b: b.soc)
    if lowest.soc <= self.config.switch_threshold_soc:
        active_in_j = self.system.num_active_in_junction(
            lowest.junction)
        if active_in_j <= self.system.min_active_per_junction:
            return []
        can_switch, _ = self.system.can_switch_to_charge(
            lowest.battery_id)
        if can_switch:
            return [lowest.battery_id]
    return []

def _get_junction_tier(self, junction: int) -> int:
    median_soc = self.system.get_junction_median_soc(junction)
    if median_soc < self.rolling_config.tier3_threshold:
        return 3
    elif median_soc < self.rolling_config.tier2_threshold:
        return 2
    elif median_soc < self.rolling_config.tier1_threshold:
        return 1
    return 0

def _get_tier_target_charge_count(self, junction: int,
                                  tier: int) -> int:
    tier1, tier2, tier3 = \
        self.rolling_config.get_tier_charge_counts(
            self.system.batteries_per_junction)
```

```
    if tier == 3:
        return tier3
    elif tier == 2:
        return tier2
    elif tier == 1:
        return tier1
    return 0

def _select_for_charging_rolling(
        self, timestamp_s: float
) -> Tuple[List[int], List[int]]:
    to_charge = []
    to_return = []
    for junction in range(self.system.num_junctions):
        tier = self._get_junction_tier(junction)
        self.junction_tiers[junction] = tier
        power_bats = self.system.active_in_junction(junction)
        charging_bats = self.system.charging_in_junction(junction)
        target_charging = self._get_tier_target_charge_count(
            junction, tier)
        current_charging = len(charging_bats)
        power_sorted = sorted(
            power_bats,
            key=lambda b: (b.soc,
                           -b.total_energy_discharged_wh))
        charging_sorted = sorted(
            charging_bats, key=lambda b: b.soc, reverse=True)

        # Phase 1: Fill to tier target
        while current_charging < target_charging and \
                power_sorted:
            for i, battery in enumerate(power_sorted):
                time_since = (timestamp_s -
                              battery.last_switch_time_s)
                if time_since < \
                        self.rolling_config.min_dwell_time_s:
                    continue
                active_count = \
                    self.system.num_active_in_junction(junction)
                pending = len([
                    b for b in to_charge
                    if self.system.batteries[b].junction ==
                    junction])
                if (active_count - pending <=
                        self.system.min_active_per_junction):
                    break
                can_switch, _ = \
                    self.system.can_switch_to_charge(
                        battery.battery_id)
                if can_switch:
                    to_charge.append(battery.battery_id)
```

```
                    power_sorted.pop(i)
                    current_charging += 1
                    break
            else:
                break

        # Phase 2: Continuous swap optimization
        if power_sorted and charging_sorted:
            lowest_power = power_sorted[0]
            highest_charging = charging_sorted[0]
            differential = (highest_charging.soc -
                            lowest_power.soc)
            if differential >= \
                    self.rolling_config.swap_differential:
                p_time = (timestamp_s -
                          lowest_power.last_switch_time_s)
                c_time = (timestamp_s -
                          highest_charging.last_switch_time_s)
                if (p_time >=
                        self.rolling_config.min_dwell_time_s and
                    c_time >=
                        self.rolling_config.min_dwell_time_s and
                    self._has_sufficient_charge_gain(
                        highest_charging)):
                    active_count = \
                        self.system.num_active_in_junction(
                            junction)
                    pending_c = len([
                        b for b in to_charge
                        if self.system.batteries[b].junction ==
                        junction])
                    pending_r = len([
                        b for b in to_return
                        if self.system.batteries[b].junction ==
                        junction])
                    if (active_count - pending_c + pending_r >=
                            self.system.min_active_per_junction):
                        if (lowest_power.battery_id not in
                                to_charge and
                            highest_charging.battery_id not in
                                to_return):
                            to_charge.append(
                                lowest_power.battery_id)
                            to_return.append(
                                highest_charging.battery_id)
    return to_charge, to_return

def _check_charge_completion_rolling(
        self, timestamp_s: float) -> List[SwitchAction]:
    actions = []
    charging = self.system.charging_batteries()
```

```
    if not charging:
        return actions
    for junction in range(self.system.num_junctions):
        active_count = \
            self.system.num_active_in_junction(junction)
        junction_charging = \
            self.system.charging_in_junction(junction)
        if active_count <= 1 and junction_charging:
            eligible = [
                b for b in junction_charging
                if self._can_reactivate(b, timestamp_s)]
            if eligible:
                best = max(eligible, key=lambda b: b.soc)
                if best.battery_id not in \
                        [a.battery_id for a in actions]:
                    actions.append(SwitchAction(
                        battery_id=best.battery_id,
                        new_state=BatteryState.POWER,
                        reason=(f"Urgent return J{junction}"),
                        timestamp_s=timestamp_s,
                        category="safety"))
    for battery in charging:
        if (battery.soc >=
                self.rolling_config.charge_complete_soc and
                self._has_sufficient_charge_gain(battery)):
            if battery.battery_id not in \
                    [a.battery_id for a in actions]:
                actions.append(SwitchAction(
                    battery_id=battery.battery_id,
                    new_state=BatteryState.POWER,
                    reason=(f"Rolling complete: "
                            f"SoC {battery.soc:.1f}%"),
                    timestamp_s=timestamp_s,
                    category="planned"))
    return actions

def _schedule_charging(self,
                       timestamp_s: float) -> List[SwitchAction]:
    actions = []
    if self.strategy == \
            SchedulingStrategy.ROLLING_REPLENISHMENT:
        to_charge, to_return = \
            self._select_for_charging_rolling(timestamp_s)
        for bat_id in to_charge:
            battery = self.system.batteries[bat_id]
            tier = self.junction_tiers[battery.junction]
            actions.append(SwitchAction(
                battery_id=bat_id,
                new_state=BatteryState.CHARGE,
                reason=(f"Rolling T{tier}: "
                        f"SoC {battery.soc:.1f}%"),
                timestamp_s=timestamp_s,
```

```
                    category="planned"))
            for bat_id in to_return:
                battery = self.system.batteries[bat_id]
                actions.append(SwitchAction(
                    battery_id=bat_id,
                    new_state=BatteryState.POWER,
                    reason=(f"Rolling swap: "
                            f"SoC {battery.soc:.1f}%"),
                    timestamp_s=timestamp_s,
                    category="planned"))
            return actions
        if self.strategy == SchedulingStrategy.GREEDY:
            to_charge = self._select_for_charging_greedy()
        elif self.strategy == SchedulingStrategy.ROUND_ROBIN:
            to_charge = self._select_for_charging_round_robin()
        elif self.strategy == SchedulingStrategy.BALANCED:
            to_charge = self._select_for_charging_balanced()
        elif self.strategy == SchedulingStrategy.MIN_SWITCHES:
            to_charge = self._select_for_charging_min_switches()
        else:
            to_charge = self._select_for_charging_greedy()
        for bat_id in to_charge:
            battery = self.system.batteries[bat_id]
            time_since = timestamp_s - battery.last_switch_time_s
            if time_since >= self.config.min_dwell_time_s:
                actions.append(SwitchAction(
                    battery_id=bat_id,
                    new_state=BatteryState.CHARGE,
                    reason=(f"Schedule: "
                            f"SoC {battery.soc:.1f}%"),
                    timestamp_s=timestamp_s,
                    category="planned"))
        return actions

    def _has_sufficient_charge_gain(self, battery) -> bool:
        if battery.charge_entry_soc < 0:
            return True
        if self.strategy == \
                SchedulingStrategy.ROLLING_REPLENISHMENT:
            min_gain = self.rolling_config.min_charge_soc_gain
        else:
            min_gain = self.config.min_charge_soc_gain
        return (battery.soc - battery.charge_entry_soc) >= min_gain

    def _check_charge_completion(
            self, timestamp_s: float) -> List[SwitchAction]:
        actions = []
        charging = self.system.charging_batteries()
        if not charging:
            return actions
        for junction in range(self.system.num_junctions):
```

```
            active_count = \
                self.system.num_active_in_junction(junction)
            junction_charging = \
                self.system.charging_in_junction(junction)
            if active_count <= 1 and junction_charging:
                eligible = [
                    b for b in junction_charging
                    if self._can_reactivate(b, timestamp_s)]
                if eligible:
                    best = max(eligible, key=lambda b: b.soc)
                    actions.append(SwitchAction(
                        battery_id=best.battery_id,
                        new_state=BatteryState.POWER,
                        reason=(f"Urgent return J{junction}"),
                        timestamp_s=timestamp_s,
                        category="safety"))
        for battery in charging:
            if (battery.soc >= self.config.charge_complete_soc and
                    self._has_sufficient_charge_gain(battery)):
                if not any(a.battery_id == battery.battery_id
                           for a in actions):
                    actions.append(SwitchAction(
                        battery_id=battery.battery_id,
                        new_state=BatteryState.POWER,
                        reason=(f"Charge complete: "
                                f"SoC {battery.soc:.1f}%"),
                        timestamp_s=timestamp_s,
                        category="planned"))
        return actions

    def _update_charger(self, timestamp_s: float):
        charging = self.system.charging_batteries()
        if charging and not self.state.charger_active:
            self.state.charger_active = True
            self.state.charger_start_time_s = timestamp_s
        elif not charging and self.state.charger_active:
            self.state.charger_active = False
        return []

    def _update_system_state(self):
        if self.state.current_state == SystemState.EMERGENCY_LAND:
            return
        min_soc = self.system.get_min_battery_soc()
        stressed_junctions = sum(
            1 for j in range(self.system.num_junctions)
            if self.system.num_active_in_junction(j) == 1)
        if min_soc <= self.config.critical_threshold_soc:
            self.state.current_state = SystemState.CRITICAL
        elif (stressed_junctions > 0 or
              min_soc <= self.config.switch_threshold_soc + 5):
            self.state.current_state = SystemState.DEGRADED
```

```
        else:
            self.state.current_state = SystemState.NOMINAL

    def decide(self, timestamp_s: float,
               charger_available: bool = True) -> List[SwitchAction]:
        all_actions = []
        state_before = self.state.current_state
        safety_actions = self._check_safety(timestamp_s)
        all_actions.extend(safety_actions)
        if self.state.current_state == SystemState.EMERGENCY_LAND:
            self._log_event(
                timestamp_s, "EMERGENCY_LAND", state_before,
                self.state.current_state,
                "safety violation", all_actions)
            return all_actions
        if self.strategy != \
                SchedulingStrategy.ROLLING_REPLENISHMENT:
            all_actions.extend(
                self._check_junction_balance(timestamp_s))
        if self.strategy == \
                SchedulingStrategy.ROLLING_REPLENISHMENT:
            all_actions.extend(
                self._check_charge_completion_rolling(timestamp_s))
        else:
            all_actions.extend(
                self._check_charge_completion(timestamp_s))
        if charger_available:
            all_actions.extend(
                self._schedule_charging(timestamp_s))
            self._update_charger(timestamp_s)
        self._update_system_state()
        if all_actions:
            reasons = [a.reason for a in all_actions[:3]]
            self._log_event(
                timestamp_s, "ACTIONS", state_before,
                self.state.current_state,
                "; ".join(reasons), all_actions)
        return all_actions

    def execute_actions(self,
                        actions: List[SwitchAction]) -> None:
        for action in actions:
            success = self.system.set_battery_state(
                action.battery_id, action.new_state,
                action.timestamp_s)
            if success:
                self.state.total_switches += 1
                if action.category == "safety":
                    self.state.safety_switches += 1
                else:
                    self.state.planned_switches += 1
```

```
    def get_events(self) -> List[DecisionEvent]:
        return self.state.events.copy()
```

C.4 Simulation Engine (simulation.py)

Main 13-step fixed-timestep loop, MissionProfile throttle generator with randomized maneuvers, piecewise-linear throttle-to-power interpolation, fuel consumption model, degraded-state power reduction, termination conditions, and results recording. Includes strategy comparison and sensitivity analysis runners.

```
"""
Skywalker III Battery Power Management Simulation
Simulation Engine for Series-Junction Battery System

TOPOLOGY:
- N individual 4S LiPo batteries (9 prototype, 48 full)
- 3 junctions in SERIES (44.4V system voltage)
- N/3 batteries per junction in PARALLEL
- Charging at 4S voltage (16.8V) via auto-parallel Keysight or generator

Time-stepped simulation with variable power draw, series current flow,
parallel current split, SSR-based switching, CC/CV charging, and
multiple scheduling strategies including ROLLING_REPLENISHMENT.
"""

import logging
from dataclasses import dataclass, field
from typing import List, Optional, Callable
import numpy as np

from battery import BatterySystem, BatteryState, voltage_to_soc
from algorithm import SwitchingAlgorithm, SystemState, SwitchAction
from keysight import SimulatedCharger, ChargerMode, create_charger
from config import (
    ScenarioConfig, SimulationConfig, AlgorithmConfig,
    KeysightConfig, SchedulingStrategy, RollingConfig,
    DEFAULT_ROLLING_CONFIG, get_system_config,
    BATTERY_ENERGY_WH, FUEL_CAPACITY_WH,
    BENCH_TEST_POWER_W, THROTTLE_POWER_MAP,
)

logger = logging.getLogger(__name__)


@dataclass
class SimulationStep:
    time_s: float
```

```
    system_state: str
    scheduling_strategy: str
    battery_socs: List[float]
    battery_states: List[str]
    battery_min_voltages: List[float]
    battery_currents: List[float]
    battery_temperatures: List[float]
    junction_voltages: List[float]
    junction_active_counts: List[int]
    system_voltage: float
    num_power_batteries: int
    num_charging_batteries: int
    system_soc: float
    soc_std_dev: float
    throttle: float
    power_draw_w: float
    mission_phase: str
    charger_active: bool
    charger_mode: str
    charger_voltage: float
    charger_current: float
    charger_power_w: float
    current_per_charging_battery: float
    fuel_remaining_wh: float


@dataclass
class SimulationResult:
    scenario_name: str
    config: ScenarioConfig
    steps: List[SimulationStep] = field(default_factory=list)
    system_config_name: str = "prototype"
    num_batteries: int = 9
    num_junctions: int = 3
    batteries_per_junction: int = 3
    total_flight_time_s: float = 0.0
    total_switches: int = 0
    total_planned_switches: int = 0
    total_safety_switches: int = 0
    fuel_consumed_wh: float = 0.0
    min_cell_voltage_reached: float = 4.2
    ended_reason: str = ""
    battery_switch_counts: List[int] = field(
        default_factory=list)
    battery_min_socs: List[float] = field(
        default_factory=list)
    battery_energy_discharged: List[float] = field(
        default_factory=list)
    battery_energy_charged: List[float] = field(
        default_factory=list)
    time_in_state: dict = field(default_factory=dict)
```

```
    final_soc_std_dev: float = 0.0
    tier_history: List[List[int]] = field(
        default_factory=list)

    def __post_init__(self):
        cfg = get_system_config(
            self.config.system_config
            if self.config else "prototype")
        n = cfg["num_batteries"]
        self.system_config_name = cfg["name"]
        self.num_batteries = n
        self.num_junctions = cfg["num_junctions"]
        self.batteries_per_junction = (
            cfg["batteries_per_junction"])
        if not self.battery_switch_counts:
            self.battery_switch_counts = [0] * n
        if not self.battery_min_socs:
            self.battery_min_socs = [100.0] * n
        if not self.battery_energy_discharged:
            self.battery_energy_discharged = [0.0] * n
        if not self.battery_energy_charged:
            self.battery_energy_charged = [0.0] * n


class MissionProfile:
    """Throttle profile: takeoff (30s at 80%), cruise (30%),
    maneuvers (50% +/-50% duration, every 300s +/-30%)."""

    def __init__(self, config: SimulationConfig,
                 rng: np.random.Generator):
        self.config = config
        self.rng = rng
        self.phase = "preflight"
        self.next_maneuver_time = config.maneuver_interval_s
        self.maneuver_end_time = -1.0

    def get_throttle(self, time_s: float
                     ) -> tuple[float, str]:
        if time_s < self.config.takeoff_duration_s:
            return self.config.takeoff_throttle, "takeoff"
        if time_s >= self.next_maneuver_time:
            if self.maneuver_end_time < 0:
                duration = self.rng.uniform(
                    self.config.maneuver_duration_s * 0.5,
                    self.config.maneuver_duration_s * 1.5)
                self.maneuver_end_time = time_s + duration
            if time_s < self.maneuver_end_time:
                return (self.config.maneuver_throttle,
                        "maneuver")
            else:
                self.maneuver_end_time = -1.0
```

```
                interval = self.rng.uniform(
                    self.config.maneuver_interval_s * 0.7,
                    self.config.maneuver_interval_s * 1.3)
                self.next_maneuver_time = time_s + interval
            return self.config.cruise_throttle, "cruise"


def throttle_to_power(throttle: float) -> float:
    """Piecewise-linear interpolation over
    THROTTLE_POWER_MAP."""
    throttle = max(0.0, min(1.0, throttle))
    throttle_points = sorted(THROTTLE_POWER_MAP.keys())
    for i in range(len(throttle_points) - 1):
        t_low = throttle_points[i]
        t_high = throttle_points[i + 1]
        if t_low <= throttle <= t_high:
            p_low = THROTTLE_POWER_MAP[t_low]
            p_high = THROTTLE_POWER_MAP[t_high]
            fraction = ((throttle - t_low) /
                        (t_high - t_low))
            return p_low + fraction * (p_high - p_low)
    if throttle < throttle_points[0]:
        return THROTTLE_POWER_MAP[throttle_points[0]]
    return THROTTLE_POWER_MAP[throttle_points[-1]]


class Simulation:
    """Main simulation engine for the series-junction
    battery system. Fixed 0.5s timestep, 13-step loop."""

    def __init__(self,
                 scenario: Optional[ScenarioConfig] = None):
        self.scenario = scenario or ScenarioConfig()
        self.config = self.scenario.simulation
        self.algo_config = self.scenario.algorithm
        self.system_config_name = self.scenario.system_config
        self.rng = np.random.default_rng(
            self.config.random_seed)
        self.battery_system = BatterySystem(
            system_config=self.system_config_name,
            initial_soc_per_battery=(
                self.config.initial_soc_per_battery),
            battery_degradation=(
                self.scenario.battery_degradation),
            rng=self.rng)
        rolling_config = None
        if (self.scenario.scheduling_strategy ==
                SchedulingStrategy.ROLLING_REPLENISHMENT):
            rolling_config = self.scenario.rolling_config
        self.algorithm = SwitchingAlgorithm(
            battery_system=self.battery_system,
```

```
            config=self.algo_config,
            strategy=self.scenario.scheduling_strategy,
            rolling_config=rolling_config)
        self.charger = create_charger(
            self.scenario.keysight,
            use_generator=not self.config.bench_test_mode)
        self.mission = MissionProfile(self.config, self.rng)
        self.fuel_remaining_wh = (
            FUEL_CAPACITY_WH *
            self.config.fuel_initial_fraction)
        self.result = SimulationResult(
            scenario_name=self.scenario.name,
            config=self.scenario)

    def _charger_available(self, time_s: float) -> bool:
        if self.scenario.engine_failure_time_s >= 0:
            if time_s >= self.scenario.engine_failure_time_s:
                return False
        if not self.config.bench_test_mode:
            return self.fuel_remaining_wh > 0
        return True

    def _consume_fuel(self, power_w: float,
                      duration_s: float) -> float:
        if self.config.bench_test_mode:
            return power_w
        energy_required = power_w * duration_s / 3600.0
        if energy_required > self.fuel_remaining_wh:
            actual_energy = self.fuel_remaining_wh
            self.fuel_remaining_wh = 0.0
            return actual_energy * 3600.0 / duration_s
        else:
            self.fuel_remaining_wh -= energy_required
            return power_w

    def _record_step(self, time_s, throttle,
                     power_draw_w, mission_phase):
        bs = self.battery_system
        algo = self.algorithm
        charger = self.charger
        step = SimulationStep(
            time_s=time_s,
            system_state=algo.state.current_state.name,
            scheduling_strategy=(
                self.scenario.scheduling_strategy.value),
            battery_socs=[b.soc for b in bs.batteries],
            battery_states=[b.state.value
                            for b in bs.batteries],
            battery_min_voltages=[b.min_cell_voltage
                                  for b in bs.batteries],
            battery_currents=[b.current_draw_a
```

```
                            for b in bs.batteries],
                battery_temperatures=[b.temperature_c
                                      for b in bs.batteries],
                junction_voltages=[
                    bs.junction_voltage(j)
                    for j in range(bs.num_junctions)],
                junction_active_counts=[
                    bs.num_active_in_junction(j)
                    for j in range(bs.num_junctions)],
                system_voltage=bs.system_voltage(),
                num_power_batteries=len(bs.power_batteries()),
                num_charging_batteries=len(
                    bs.charging_batteries()),
                system_soc=bs.get_system_soc(),
                soc_std_dev=bs.get_soc_std_dev(),
                throttle=throttle,
                power_draw_w=power_draw_w,
                mission_phase=mission_phase,
                charger_active=charger.state.output_enabled,
                charger_mode=charger.state.mode.value,
                charger_voltage=charger.state.actual_voltage,
                charger_current=charger.state.actual_current,
                charger_power_w=charger.state.actual_power,
                current_per_charging_battery=(
                    charger.state.current_per_battery),
                fuel_remaining_wh=self.fuel_remaining_wh)
            self.result.steps.append(step)
            for i, b in enumerate(bs.batteries):
                self.result.battery_min_socs[i] = min(
                    self.result.battery_min_socs[i], b.soc)
            min_cell_v = min(
                b.min_cell_voltage for b in bs.batteries)
            self.result.min_cell_voltage_reached = min(
                self.result.min_cell_voltage_reached,
                min_cell_v)
            if (self.scenario.scheduling_strategy ==
                    SchedulingStrategy.ROLLING_REPLENISHMENT):
                self.result.tier_history.append(
                    algo.junction_tiers.copy())

        def run(self, progress_callback=None
                ) -> SimulationResult:
            """Main 13-step simulation loop at 0.5s timestep."""
            time_s = 0.0
            timestep = self.config.timestep_s
            max_time = self.config.max_duration_s
            state_times = {
                state.name: 0.0 for state in SystemState}
            last_state = self.algorithm.state.current_state

            while time_s < max_time:
```

```
# 1. Throttle query
throttle, mission_phase = (
    self.mission.get_throttle(time_s))
# 2. Power demand mapping
power_draw = throttle_to_power(throttle)
# 3. Degraded-state power reduction
if self.config.bench_test_mode:
    power_draw = min(
        power_draw, BENCH_TEST_POWER_W)
if (self.algorithm.state.current_state ==
        SystemState.EMERGENCY_LAND):
    power_draw *= 0.3
elif (self.algorithm.state.current_state ==
        SystemState.CRITICAL):
    power_draw *= 0.7
# 4. Discharge active batteries
if (self.algorithm.state.current_state !=
        SystemState.EMERGENCY_LAND):
    _, all_safe = (
        self.battery_system.discharge_step(
            power_draw, timestep))
# 5. CC/CV charge step
charger_available = (
    self._charger_available(time_s))
charging_batteries = (
    self.battery_system.charging_batteries())
if (charging_batteries and
        charger_available and
        self.algorithm.state.charger_active):
    battery_voltages = [
        b.total_voltage
        for b in charging_batteries]
    if self.config.bench_test_mode:
        target_current = (
            self.scenario.keysight
            .charge_current_1c *
            len(charging_batteries))
    else:
        target_current = (
            self.charger.max_power /
            max(battery_voltages))
    actual_current, actual_voltage, mode = (
        self.charger.charge_step(
            battery_voltages,
            target_current,
            timestep, time_s))
    if actual_current > 0:
        self.battery_system.charge_step(
            actual_current,
            actual_voltage, timestep)
    # 6. Generator fuel consumption
    if not self.config.bench_test_mode:
```

```
                charger_power = (
                    actual_voltage * actual_current)
                self._consume_fuel(
                    charger_power / 0.85, timestep)
            elif (not charging_batteries and
                    self.charger.state.output_enabled):
                self.charger.disable_output()
            # 7. Scheduling algorithm invocation
            actions = self.algorithm.decide(
                time_s, charger_available)
            # 8. SSR state change execution
            self.algorithm.execute_actions(actions)
            if (self.algorithm.state.charger_active and
                    not self.charger.state.output_enabled):
                self.charger.enable_output(time_s)
            elif (not self.algorithm.state.charger_active
                    and self.charger.state.output_enabled):
                self.charger.disable_output()
            state_times[last_state.name] += timestep
            last_state = (
                self.algorithm.state.current_state)
            # 9. State recording
            self._record_step(
                time_s, throttle,
                power_draw, mission_phase)
            # 10. Emergency landing termination
            if (self.algorithm.state.current_state ==
                    SystemState.EMERGENCY_LAND):
                self.result.ended_reason = (
                    "Emergency landing")
                break
            # 11. All-batteries-critical termination
            all_critical = all(
                b.soc <=
                self.algo_config.critical_threshold_soc
                for b in self.battery_system.batteries)
            if all_critical:
                charger_can_help = (
                    not self.config.bench_test_mode
                    and self.fuel_remaining_wh > 0
                    and charger_available
                    and charging_batteries)
                if not charger_can_help:
                    self.result.ended_reason = (
                        "All batteries critical")
                    break
            if (progress_callback and
                    int(time_s) % 10 == 0):
                progress_callback(time_s / max_time)
            time_s += timestep

        # 12. Max-duration termination
```

```
        if not self.result.ended_reason:
            self.result.ended_reason = (
                "Max duration reached")
        # 13. Finalize results
        self.result.total_flight_time_s = time_s
        self.result.total_switches = (
            self.algorithm.state.total_switches)
        self.result.total_planned_switches = (
            self.algorithm.state.planned_switches)
        self.result.total_safety_switches = (
            self.algorithm.state.safety_switches)
        self.result.fuel_consumed_wh = (
            FUEL_CAPACITY_WH *
            self.config.fuel_initial_fraction -
            self.fuel_remaining_wh)
        self.result.time_in_state = state_times
        self.result.final_soc_std_dev = (
            self.battery_system.get_soc_std_dev())
        for i, b in enumerate(
                self.battery_system.batteries):
            self.result.battery_switch_counts[i] = (
                b.switch_count)
            self.result.battery_energy_discharged[i] = (
                b.total_energy_discharged_wh)
            self.result.battery_energy_charged[i] = (
                b.total_energy_charged_wh)
        return self.result


def run_strategy_comparison(base_scenario,
                            strategies=None,
                            progress_callback=None):
    if strategies is None:
        strategies = list(SchedulingStrategy)
    results = []
    for i, strategy in enumerate(strategies):
        scenario = ScenarioConfig(
            name=(f"{base_scenario.name}_"
                  f"{strategy.value}"),
            description=(
                f"{base_scenario.description} "
                f"(strategy: {strategy.value})"),
            algorithm=base_scenario.algorithm,
            simulation=base_scenario.simulation,
            keysight=base_scenario.keysight,
            scheduling_strategy=strategy,
            rolling_config=base_scenario.rolling_config,
            battery_degradation=(
                base_scenario.battery_degradation),
            engine_failure_time_s=(
                base_scenario.engine_failure_time_s),
```

```
            system_config=base_scenario.system_config)
        sim = Simulation(scenario)
        result = sim.run()
        results.append(result)
        if progress_callback:
            progress_callback(i + 1, len(strategies))
    return results


def run_sensitivity_analysis(base_scenario,
                             switch_thresholds,
                             progress_callback=None):
    results = []
    for i, threshold in enumerate(switch_thresholds):
        scenario = ScenarioConfig(
            name=(f"{base_scenario.name}_"
                  f"switch{int(threshold)}"),
            description=(
                f"{base_scenario.description} "
                f"(switch threshold {threshold}%)"),
            algorithm=AlgorithmConfig(
                switch_threshold_soc=threshold,
                critical_threshold_soc=(
                    base_scenario.algorithm
                    .critical_threshold_soc),
                charge_complete_soc=(
                    base_scenario.algorithm
                    .charge_complete_soc)),
            simulation=base_scenario.simulation,
            keysight=base_scenario.keysight,
            scheduling_strategy=(
                base_scenario.scheduling_strategy),
            rolling_config=(
                base_scenario.rolling_config),
            battery_degradation=(
                base_scenario.battery_degradation),
            engine_failure_time_s=(
                base_scenario.engine_failure_time_s),
            system_config=base_scenario.system_config)
        sim = Simulation(scenario)
        results.append(sim.run())
        if progress_callback:
            progress_callback(
                i + 1, len(switch_thresholds))
    return results
```

C.5 Charger Model (keysight.py)

SimulatedCharger class implementing CC/CV charging protocol with taper fraction, power-limited current sharing, and factory function for Keysight (400 W, 98% efficiency) versus generator (3,400 W, 85% efficiency) configurations.

```
"""
Skywalker III Battery Power Management Simulation
Keysight E36233A Integration for 4S Charging

The charge bus operates at 4S voltage (14.8-16.8V), not 12S.
All charging batteries are in PARALLEL on an isolated charge bus.
Auto-parallel mode: both channels combined for higher current
at same voltage. 16.8V max, up to 40A combined, 400W limit.
"""


import atexit
import logging
import time
from dataclasses import dataclass, field
from enum import Enum
from typing import List, Optional, Tuple

from config import (KeysightConfig, CELL_CV_THRESHOLD,
                    CELLS_PER_BATTERY)

logger = logging.getLogger(__name__)

try:
    import pyvisa
    PYVISA_AVAILABLE = True
except ImportError:
    PYVISA_AVAILABLE = False


class ChargerMode(Enum):
    OFF = "off"
    CC = "cc"
    CV = "cv"
    STARTUP = "startup"


@dataclass
class ChargerState:
    mode: ChargerMode = ChargerMode.OFF
    output_enabled: bool = False
    voltage_setpoint: float = 0.0
    current_setpoint: float = 0.0
    actual_voltage: float = 0.0
    actual_current: float = 0.0
    actual_power: float = 0.0
    num_batteries_charging: int = 0
```

```
    current_per_battery: float = 0.0


@dataclass
class ChargerDataPoint:
    timestamp_s: float
    voltage: float
    current: float
    power: float
    mode: ChargerMode
    num_batteries: int


class SimulatedCharger:
    """Simulated 4S CC/CV charger. Supports both
    Keysight (400W) and generator (3400W)."""

    def __init__(self, max_voltage=16.8,
                 max_current=23.8, max_power=400.0,
                 efficiency=0.98, startup_delay=0.0):
        self.max_voltage = max_voltage
        self.max_current = max_current
        self.max_power = max_power
        self.efficiency = efficiency
        self.startup_delay = startup_delay
        self.state = ChargerState()
        self.data_log: List[ChargerDataPoint] = []
        self.startup_time: float = -1.0

    def enable_output(self,
                      timestamp_s: float = 0.0) -> None:
        self.state.output_enabled = True
        self.state.mode = ChargerMode.STARTUP
        self.startup_time = timestamp_s

    def disable_output(self) -> None:
        self.state.output_enabled = False
        self.state.mode = ChargerMode.OFF
        self.state.actual_voltage = 0.0
        self.state.actual_current = 0.0
        self.state.actual_power = 0.0
        self.startup_time = -1.0

    def is_ready(self, timestamp_s: float) -> bool:
        if not self.state.output_enabled:
            return False
        if self.startup_time < 0:
            return False
        return ((timestamp_s - self.startup_time) >=
                self.startup_delay)
```

```
def charge_step(self, battery_voltages,
                target_current, dt_seconds,
                timestamp_s):
    if not battery_voltages:
        self.state.num_batteries_charging = 0
        return 0.0, 0.0, ChargerMode.OFF
    if not self.is_ready(timestamp_s):
        self.state.mode = ChargerMode.STARTUP
        return 0.0, 0.0, ChargerMode.STARTUP
    num_batteries = len(battery_voltages)
    self.state.num_batteries_charging = num_batteries
    pack_voltage = max(battery_voltages)
    cv_threshold = (CELL_CV_THRESHOLD *
                    CELLS_PER_BATTERY)
    max_current = min(
        target_current,
        self.max_power / pack_voltage,
        self.max_current)
    if pack_voltage < cv_threshold:
        mode = ChargerMode.CC
        output_current = max_current
    else:
        mode = ChargerMode.CV
        voltage_headroom = (self.max_voltage -
                            pack_voltage)
        taper_factor = max(
            0.05,
            voltage_headroom /
            (self.max_voltage - cv_threshold))
        output_current = max_current * taper_factor
    effective_current = (output_current *
                         self.efficiency)
    self.state.mode = mode
    self.state.actual_voltage = pack_voltage
    self.state.actual_current = effective_current
    self.state.actual_power = (pack_voltage *
                               effective_current)
    self.state.current_per_battery = (
        effective_current / num_batteries)
    self.data_log.append(ChargerDataPoint(
        timestamp_s=timestamp_s,
        voltage=pack_voltage,
        current=effective_current,
        power=pack_voltage * effective_current,
        mode=mode,
        num_batteries=num_batteries))
    return effective_current, pack_voltage, mode

def get_state(self):
    return self.state
```

```
    def get_data_log(self):
        return self.data_log.copy()


def create_charger(config: KeysightConfig,
                   use_generator: bool = False
                   ) -> SimulatedCharger:
    """Factory: 400W Keysight for bench or
    3400W generator for flight."""
    if use_generator:
        return SimulatedCharger(
            max_voltage=16.8, max_current=200.0,
            max_power=3400.0, efficiency=0.85,
            startup_delay=4.0)
    else:
        return SimulatedCharger(
            max_voltage=config.charge_voltage,
            max_current=config.max_current_at_16v,
            max_power=config.power_limit,
            efficiency=config.efficiency,
            startup_delay=0.0)
```