UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

DESIGN OF A ROBUST MICROCONTROLLER-TO-MICROCONTROLLER REDUNDANCY PROTOCOL FOR SPACE ENVIRONMENTS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By JOEL NKANGI

Norman, Oklahoma

2026

DESIGN OF A ROBUST MICROCONTROLLER-TO-MICROCONTROLLER REDUNDANCY PROTOCOL FOR SPACE ENVIRONMENTS

A THESIS APPROVED FOR THE

SCHOOL OF AEROSPACE AND MECHANICAL ENGINEERING

BY THE COMMITTEE CONSISTING OF

Dr. Diogo Merguizo Sanchez, Chair

Dr. Wilson Merchan-Merchan

Dr. Hamidreza Shabgard

Abstract

Reliability in space-based electronic systems is often compromised by single points of failure such as intermediate microcontrollers. This work develops a software-based redundancy mechanism enabling direct Microcontroller-to-Microcontroller (Arduino-based) communication for improved system fault tolerance for low-cost small satellites. Traditional architectures rely on intermediary controllers to coordinate communication and synchronization between Arduino nodes. However, under harsh environmental conditions such as radiation exposure, extreme temperatures, and vibration, these microcontrollers are equally susceptible to damage, leading to a complete communication breakdown and system failure. To address this challenge, this study eliminates the dependency on intermediary devices by establishing a direct communication protocol between Arduinos that can autonomously detect and respond to faults.

The proposed mechanism employs a heartbeat-based communication model, where each Arduino periodically transmits and monitors “alive” signals from its peers. If a heartbeat is missed beyond a predefined threshold, the system identifies a potential node failure and automatically activates a standby Arduino to assume the operational responsibilities of the failed unit. This approach is implemented entirely in software, thereby reducing hardware complexity and minimizing additional points of failure. The communication framework is designed to support multiple channels such as serial, I²C, or SPI, depending on the application environment, ensuring adaptability and scalability for various mission-critical systems.

Experimental results demonstrate that interconnected Arduinos can successfully establish mutual heartbeat detection, enabling reliable failure identification within seconds of a node going offline. The system automatically transfers control to the standby Arduino, maintaining continuity of operations without manual intervention. Overall, this research contributes to the development of resilient embedded networks suitable for mission-critical and space applications by enhancing fault tolerance, reducing hardware dependencies, and providing a foundation for scalable, self-healing Arduino networks.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Diogo Sanchez, for his unwavering support and guidance throughout the development of this thesis. His expertise, patience, and consistent encouragement have been instrumental in shaping this research from its initial concept to its final completion.

Dr. Sanchez's readiness to provide clarity whenever challenges arose, his detailed explanations of complex concepts, and his thoughtful feedback at every stage of the process significantly strengthened both the technical depth and overall quality of this work. His commitment to information sharing and constructive discussion ensured that each research decision was well-informed and grounded in sound engineering principles.

Beyond academic supervision, his encouragement and confidence in my abilities provided the motivation needed to persevere through the more demanding phases of this project. The structured guidance and insightful direction he offered not only contributed to the successful completion of this thesis but also enhanced my understanding of research methodology and embedded systems design.

I am deeply grateful for his mentorship and support throughout this journey.

Contents

List of Figures

List of Abbreviations

CRC	Cyclic Redundancy Check
CS	Chip Select
DIP	Dual In-line Package
I^2C	Inter-Integrated Circuit
MISO	Master In Slave Out
MOSI	Master Out Slave In
PICO	Peripheral In Controller Out
POCI	Peripheral Out Controller In
PJON	Padded Jittering Operative Network
RTOS	Real-Time Operating System
SCK	Serial Clock
SCL	Serial Clock Line
SDA	Serial Data Line
SEU	Single Event Upset
SPI	Serial Peripheral Interface
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver-Transmitter

1.0 Introduction and Overview

Modern embedded systems are increasingly deployed in environments where reliability is not optional but fundamental (Solouki et al., 2024). From space missions and defense systems to remote sensing platforms and autonomous robotics, electronic subsystems are expected to operate continuously without physical maintenance or manual intervention. In such contexts, a single failure can compromise an entire mission.

As embedded architectures grow more distributed, microcontrollers such as Arduino class devices are often used as modular control nodes responsible for sensing, decision-making, and communication. Their affordability, accessibility, and flexibility make them attractive for experimental aerospace platforms, CubeSats, and research driven missions. However, while these microcontrollers provide functional capability, their reliability under fault conditions remains a critical concern.

Traditional approaches to improving system robustness have relied heavily on hardware duplication, supervisory controllers, or centralized coordination units. While effective in some domains, such strategies increase system complexity, power consumption, and mass, all of which are tightly constrained in space oriented and harsh-environment applications. Furthermore, centralized architecture introduces structural vulnerabilities, that is, when coordination depends on a single intermediary controller, that device becomes a critical point of failure. Recent research has therefore explored software-based fault tolerance mechanisms as a more efficient alternative (Afonso et al., 2008).

This thesis addresses the challenge of improving fault tolerance in distributed microcontroller systems without introducing additional hardware overhead. Specifically, it investigates how direct Microcontroller-to-Microcontroller communication can be used to implement software-based redundancy. By enabling nodes to monitor one another and autonomously reassign operational roles, the system can maintain functionality even when individual components fail.

The work presented here proposes and implements a lightweight, peer-to-peer redundancy mechanism based on heartbeat monitoring and dynamic role switching. The goal is to demonstrate that resilience traditionally achieved through hardware duplication can, in certain constrained environments, be approximated through carefully designed software coordination. In doing so, this research contributes toward more autonomous, self-healing embedded networks suitable for cost-sensitive and space-inspired applications.

1.1 Background and Motivation:

In many Arduino-based embedded architectures, communication between multiple nodes is commonly managed through an intermediary microcontroller or centralized coordinator. This design simplifies synchronization and task allocation, as a single device governs message routing, timing, and system state awareness. Such an approach is practical in laboratory and terrestrial applications where maintenance and hardware replacement are feasible.

However, this architecture introduces a structural weakness. The intermediary controller becomes a single point of failure. If that device malfunctions due to radiation effects, electrical overstress, software corruption, or environmental stressors, communication between all connected nodes may cease. In mission critical systems particularly those operating in space, such a failure can terminate the entire operation.

Space environments are especially inclement with electronic devices. Radiation-induced single-event upsets (SEUs), defined as unintended changes in logic states caused by high-energy particle strikes, along with transient faults and hardware degradation, further amplify this vulnerability, extreme temperature cycling, vacuum conditions, and electromagnetic interference can disrupt or permanently damage electronic components (Baumann, 2005). Unlike ground-based systems, recovery through physical intervention is impossible. As a result, spacecraft and satellites increasingly demand autonomous fault detection and recovery mechanisms.

Large scale space systems traditionally address this challenge using hardware redundancy, such as Triple Modular Redundancy (TMR) or duplicated onboard computers. While effective, these methods significantly increase mass, volume, and power requirements(Balasubramanian & Prasad, 2016). For small satellites and CubeSats, where every gram and watt must be justified, such solutions are often impractical.

This creates a design tension: how can system resilience be improved without introducing additional hardware complexity?

Software-based redundancy offers a promising alternative. Instead of duplicating hardware with centralized voting logic, nodes can monitor one another directly through periodic status exchanges, commonly referred to as heartbeat messages. When a node fails to respond within a predefined interval, another node can autonomously assume its responsibilities, an approach supported in distributed embedded system research (Afonso et al., 2008).

Despite extensive research on high-performance distributed spacecraft computing (Fayyaz, 2015), there remains limited exploration of lightweight, peer-to-peer redundancy models implemented solely through direct microcontroller communication. Particularly within Arduino-class systems, most implementations still rely on centralized coordinators.

The motivation for this research therefore arises from a clear architectural gap which is enabling distributed, intermediary-free communication between microcontrollers to eliminate single points of failure while maintaining minimal hardware overhead.

By addressing this gap, the study aims to enhance autonomy, improve fault tolerance, and contribute toward the development of resilient embedded systems suitable for harsh and space like environments.

1.2 Problem Statement:

As discussed in the preceding sections, distributed microcontroller systems frequently adopt centralized communication architectures in which an intermediary controller coordinates message exchange and system state awareness. While such designs simplify implementation and synchronization, they introduce a critical structural weakness, the intermediary node becomes a single point of failure.

In environments where reliability is paramount, particularly in space and other harsh operational domains, failure of a single coordinating microcontroller can disrupt inter-node communication entirely, leading to loss of control, halted data acquisition, or mission termination. Radiation induced upsets, transient faults, and hardware degradation further amplify this vulnerability (Baumann, 2005). When redundancy is implemented solely at the hardware supervisory level, system resilience becomes dependent on the continued operation of the very component intended to ensure coordination.

Additionally, incorporating extra supervisory microcontrollers to mitigate this risk increases hardware complexity, mass, power consumption, and overall system cost. In space constrained and energy limited platforms such as CubeSats and experimental aerospace systems, such overhead is undesirable and often impractical, as widely noted in fault-tolerant embedded system design literature (Solouki et al., 2024). The trade-off between resilience and resource efficiency therefore remains a persistent design challenge.

The core problem addressed in this research is the absence of a lightweight, intermediary free redundancy mechanism suitable for Arduino class embedded systems. Specifically, there is a need for a communication architecture that enables direct Arduino-to-Arduino interaction, allowing nodes to monitor each other's operational status without relying on centralized coordination.

Such a mechanism must satisfy three essential requirements:

1. Autonomous detection of peer failure through reliable status exchange,
2. Transfer of operational responsibility upon failure, and
3. Minimal additional hardware overhead.

Without such a distributed redundancy model, Arduino-based systems remain structurally vulnerable to single node failure, limiting their suitability for autonomous and mission-critical applications.

This thesis therefore seeks to address this architectural gap by designing and implementing a software-based, peer-to-peer redundancy framework that enhances fault tolerance while preserving the simplicity and resource efficiency characteristic of Arduino-class platforms, aligning with the broader shift toward software-based fault tolerance approaches (Afonso et al., 2008).

1.3 Research Objectives:

The primary objective of this research is to design, implement, and evaluate a software-based redundancy framework that enables direct Arduino-to-Arduino communication, thereby improving the reliability and autonomy of distributed embedded systems.

To achieve this overarching goal, the study pursues the following specific objectives:

1. To develop a direct peer-to-peer communication protocol that enables Arduino nodes to exchange operational status information (heartbeat signals) without relying on intermediary microcontrollers or centralized coordinators.
2. To design and implement a deterministic heartbeat monitoring algorithm capable of identifying node failure within a predefined and configurable time threshold.
3. To establish an autonomous failover mechanism in which a standby Arduino dynamically assumes operational control upon detection of a peer node failure, ensuring continuity of system functionality.
4. To investigate the scalability of the proposed architecture, demonstrating that multiple Arduino nodes can participate in a cooperative, self-healing network without introducing additional hardware supervisory elements.

Through these objectives, this research seeks to validate that lightweight, software driven redundancy can enhance fault tolerance while preserving the simplicity and resource efficiency characteristic of Arduino class platforms.

1.4 Research Questions:

To fulfil the stated research objectives, this study seeks to address the following questions:

1. How can multiple Arduino boards establish direct peer-to-peer communication to monitor each other's operational status without relying on an intermediary or centralized controller?
2. What algorithmic strategy can reliably and efficiently detect node failure based on the absence or delay of periodic heartbeat signals?

3. How can operational control be transferred autonomously from a failed Arduino to a standby unit while minimizing system downtime and disruption?
4. What impact does direct inter-node communication have on system reliability and communication latency when compared to conventional microcontroller mediated architectures?

1.5 Technical and Practical Contributions:

This research contributes to the advancement of fault tolerant embedded system design by extending distributed software-based redundancy principles to low-cost, resource constrained microcontroller platforms. While software implemented fault tolerance has been widely explored in high performance and RTOS (Real Time Operating System) based space processors, its systematic application to Arduino class peer-to-peer architectures remains limited.

The primary technical contribution of this work is the development of a lightweight, intermediary free redundancy protocol that enables direct Arduino-to-Arduino communication for autonomous failure detection and dynamic role reassignment. By eliminating centralized supervisory controllers, the proposed architecture removes a critical single point of failure while maintaining minimal hardware overhead.

From an engineering perspective, this study demonstrates that deterministic heartbeat monitoring and failover control can be implemented entirely at the application level, without specialized hardware voting circuits, radiation hardened processors, or complex middleware layers. This provides a practical framework for implementing resilience in environments where mass, power, and cost constraints prohibit traditional hardware heavy redundancy schemes.

In the context of small satellite and CubeSat systems, the proposed approach offers a scalable method for improving onboard electronic survivability. By enabling distributed nodes to monitor and recover autonomously, the architecture supports extended mission continuity and improved tolerance to radiation-induced or transient faults.

Beyond aerospace applications, the findings of this research could be applicable to autonomous robotics, industrial automation, and remote sensing platforms, where maintenance access is limited and operational continuity is essential.

1.6 Thesis Structure:

This thesis is organized into five chapters, each building progressively toward the design and validation of a software-based Arduino-to-Arduino redundancy framework.

Chapter 2 presents a comprehensive review of related literature. It examines established fault tolerance strategies in embedded systems, including hardware-based redundancy, software-

implemented fault tolerance, heartbeat monitoring algorithms, and distributed failover mechanisms. The chapter also analyzes existing Arduino communication protocols and identifies gaps in current peer-to-peer redundancy research, particularly within resource-constrained microcontroller networks.

Chapter 3 introduces the proposed system architecture and design methodology. It details the communication model, heartbeat protocol, failure detection logic, and dynamic role-switching mechanism. Implementation considerations, including timing thresholds, communication interface selection, and scalability design, are discussed in depth.

Chapter 4 describes the experimental set-up and presents the results of practical validation. It evaluates direct Arduino-to-Arduino communication performance, heartbeat detection reliability, failover latency, and overall system behavior under simulated fault conditions. The findings are analyzed in relation to the research questions and objectives established in Chapter 1.

Chapter 5 concludes the thesis by summarizing the key findings and contributions of the study. It reflects on the effectiveness of the proposed redundancy mechanism, discusses current limitations, and outlines recommendations for future development and extended research, particularly in the context of space-oriented embedded systems.

2.0 Review of Related Literature – Introduction

The increasing reliance on embedded systems in mission-critical environments has intensified the need for reliable, fault-tolerant architecture that can sustain operations despite unpredictable failures. From aerospace and defense to industrial automation and transportation, embedded controllers are now expected to function continuously under harsh environmental and operational conditions. In space applications where radiation, thermal extremes, and communication delays are inherent challenges, system resilience is not merely desirable but essential.

Traditionally, fault tolerance in embedded systems has been achieved through hardware redundancy techniques such as Triple Modular Redundancy (TMR), standby duplication, and replicated communication buses. While these approaches offer strong fault masking capabilities, they impose significant overhead in terms of mass, power consumption, physical space, and cost. Such overhead is increasingly difficult to justify in emerging platforms like CubeSats and other small satellite systems, where resource efficiency is a primary design constraint. Consequently, there has been a noticeable shift in research toward software-based and distributed redundancy mechanisms that provide resilience without extensive hardware duplication.

Parallel to these developments, Arduino-class microcontrollers have gained widespread adoption due to their simplicity, affordability, and accessibility. Although not originally designed for mission-critical use, their flexibility and ease of integration have made them attractive for experimental aerospace, educational satellite missions, and distributed embedded prototypes. However, Arduino platforms lack radiation hardening, hardware voting circuits, and advanced supervisory infrastructure, raising important questions about how reliable redundancy can be achieved using such minimal hardware.

This literature review examines the evolution of fault tolerance in embedded systems, the transition from hardware-heavy to software-defined redundancy, and the emerging role of distributed peer-to-peer monitoring mechanisms. By synthesizing research in hardware redundancy, heartbeat-based fault detection, distributed embedded networks, and space-grade software resilience, this review identifies a critical gap: the limited exploration of lightweight, autonomous Arduino-to-Arduino redundancy protocols tailored for harsh environments. The subsequent sections build the theoretical and practical foundation upon which the proposed redundancy framework is developed.

2.1 Overview of Fault Tolerance in Embedded Systems

Fault tolerance is a foundational design principle for embedded systems that operate in safety critical and mission critical environments such as aerospace, defense, industrial automation, transportation, and nuclear systems. In these domains, embedded controllers are often required to operate continuously under harsh environmental conditions (radiation, temperature extremes, vibration, or electromagnetic interference), where both transient and permanent faults are likely to

occur. Consequently, fault tolerance mechanisms are introduced to ensure that system functionality and timing constraints are preserved despite component failures or unexpected disturbances.
Research consistently emphasizes that the dependability of embedded systems defined in terms of reliability, availability, safety, and maintainability cannot be achieved without explicit fault tolerance strategies integrated at the architectural level (Solouki et al., 2024). In real-time embedded systems, fault tolerance is particularly challenging because fault detection, isolation, and recovery must occur within strict timing deadlines (Afonso et al., 2008).

2.1.1 Definition and Importance

Fault tolerance refers to the ability of a system to continue delivering correct or acceptable service in the presence of faults affecting its hardware, software, or communication infrastructure. A fault-tolerant embedded system is therefore designed not only to detect errors, but also to mask, isolate, or recover from them without interrupting system operation (Powell, 2001). The importance of fault tolerance is magnified in mission critical applications, where a single failure can propagate into catastrophic system level consequences, including loss of human life or severe financial damage. Studies show that hardware faults such as single event upsets (SEUs), aging related degradation, and manufacturing defects remain a dominant source of failures in embedded platforms deployed in aerospace and industrial environments (Cui et al., 2019).

2.1.2 Hardware-Based Fault Tolerance Techniques

Traditional embedded systems have predominantly relied on hardware redundancy to achieve fault tolerance. The most widely adopted techniques include:

- Triple Modular Redundancy (TMR)
- Cold, warm, or hot standby configurations, where backup components are maintained in different states of readiness. Cold standby remains powered off until needed, warm standby is partially active, and hot standby runs in parallel and can take over immediately upon failure.
- Duplication of processing units, communication buses, and sensors which involves replicating critical system components so that if one fails, redundant units can continue operation, thereby improving system reliability and fault tolerance.

Among these, TMR is one of the most established and effective approaches. In a TMR system, three identical hardware modules execute the same task in parallel, and a majority voter determines the correct output. As long as at least two modules operate correctly, the system can mask a single fault without service interruption (Khatri, 2020). Empirical studies demonstrate that TMR significantly improves system reliability and is widely used in avionics, spacecraft, and nuclear control systems (Balasubramanian & Prasad, 2016). However, the voter (decision-making logic or component that compares the outputs of redundant modules and selects the correct result based on majority agreement) itself becomes a critical component, and its failure can compromise the entire redundancy scheme, prompting research into fault tolerant voter designs (Hadifur Rahman, 2017).

2.1.3 Limitations of Hardware Redundancy

Despite their effectiveness, hardware-based fault tolerance techniques impose substantial area, power, and cost overheads. TMR can require nearly three times the hardware resources of a non-redundant system, along with additional logic for voting and synchronization (Rogenmoser et al., 2025). These overheads make classical hardware redundancy impractical for resource-constrained embedded platforms, such as microcontroller-based systems such as Arduino-class devices, where limited memory, processing capability, and power budgets are dominant design constraints. Recent studies highlight that while TMR remains suitable for high end FPGA (Field-Programmable Gate Array) and multicore processors, lightweight embedded systems increasingly require alternative or hybrid fault tolerance approaches that trade some redundancy for efficiency (Shukla & Ray, 2022). In summary, hardware-based fault tolerance, particularly TMR, provides strong reliability guarantees for mission-critical embedded systems, but its high power, mass, and cost overhead limit its applicability in low-resource platforms, motivating the exploration of software-based and hybrid fault tolerance techniques.

2.1.4 Shift Toward Software Redundancy

As embedded systems have become smaller, more distributed, and more resource-constrained, there has been a gradual but clear shift away from purely hardware-based fault tolerance toward software-implemented redundancy mechanisms. While hardware redundancy techniques such as Triple Modular Redundancy (TMR) remain effective, their cost in terms of power, mass, and complexity makes them impractical for small microcontroller platforms and space-constrained systems. Consequently, recent research increasingly emphasizes algorithmic fault detection, isolation, and recovery, implemented at the software, operating system, or communication protocol level (Solouki et al., 2024).

One of the earliest demonstrations of this shift was presented by (Afonso et al., 2008), who introduced an aspect-oriented fault tolerance framework for real-time embedded systems. Their approach integrated redundancy and fault-handling logic into the operating system and middleware layers, allowing applications to tolerate both hardware and software faults without duplicating physical components. Crucially, their work showed that software-managed redundancy could achieve dependability comparable to hardware redundancy while imposing only minimal performance and memory overhead. This result was significant because it challenged the long-standing assumption that high reliability necessarily requires hardware duplication.

Building on this idea, (Barbosa, 2008) proposed a layered fault tolerance model for distributed embedded systems, where faults are detected and handled at multiple abstraction levels: hardware, node, and system. Rather than relying on a central controller, this model distributes responsibility across nodes, enabling localized fault detection and coordinated recovery. This distributed philosophy is particularly relevant to embedded networks deployed in harsh environments, where centralized controllers represent dangerous single points of failure.

More recent surveys further confirm this trend. (Solouki et al., 2024) highlight that software faults and transient hardware faults now dominate failure modes in modern embedded systems, especially as device feature sizes shrink and susceptibility to radiation-induced errors increases. Their review emphasizes that software-based fault tolerances such as watchdogs, checkpointing, task replication, and distributed monitoring offer a scalable and cost-effective alternative to hardware duplication, particularly for real-time and space-constrained systems.

A key advantage of software redundancy is that it enables distributed fault tolerance, where each node participates in monitoring system health rather than relying on a single supervisory device. This is especially important in space environments, where radiation, extreme temperature, and vibration can disable any single component without warning. Studies on node-level fault tolerance demonstrate that allowing each embedded node to independently detect failures and initiate recovery significantly improves overall system resilience. (Fayyaz, 2015) designed a fault-tolerant distributed computing model for onboard spacecraft that eliminates reliance on centralized controllers by implementing node-local recovery protocols, showing enhanced reliability in radiation-rich environments.

Despite these advances, most existing software-based fault tolerance frameworks still assume the presence of relatively powerful processors, operating systems, or middleware layers to manage redundancy. Many implementations depend on centralized coordinators, real-time operating systems, or complex communication stacks to orchestrate fault detection and recovery (Afonso et al., 2008). While effective, these assumptions limit applicability to low-cost microcontroller platforms, such as Arduino class devices, which lack sophisticated operating system support and operate under strict memory and power constraints, a defining challenge in CubeSat architectures constrained by size, weight, and power limitations.

This limitation creates a clear research gap. Although the literature strongly supports the viability of software-based redundancy, very little work addresses lightweight, direct microcontroller-to-microcontroller redundancy mechanisms implemented entirely in application-level code. In particular, there is a lack of studies demonstrating how small, resource-constrained controllers can autonomously monitor one another and recover from failures without intermediary hardware, centralized supervisors, or operating systems.

This research positions itself precisely within that gap. By leveraging heartbeat-based communication between multiple Arduinos, fault detection and failover can be achieved through simple, deterministic software logic. Each Arduino acts as both a functional node and a monitoring agent, periodically broadcasting its operational status and observing the health of its peers. When a heartbeat is missed beyond a defined threshold, the system identifies a failure and automatically transfers control to a standby node. This approach aligns with established principles of distributed software redundancy yet adapts them to an extremely light-weight embedded context suitable for space applications.

In doing so, the proposed Arduino-to-Arduino redundancy protocol demonstrates that software-only fault tolerance is not only feasible but practical even on minimal microcontroller platforms. It represents a natural evolution of the broader shift toward software redundancy, extending these ideas into a domain that has so far received limited attention in literature.

2.1.5 Recent Studies on Hybrid Redundancy Models

While the shift toward software-based fault tolerance has gained significant momentum, recent research indicates that hybrid redundancy models which combine limited hardware duplication with intelligent software monitoring remain an active and evolving area of study. These hybrid approaches aim to strike a balance between the strong fault-masking capabilities of hardware redundancy and the flexibility and efficiency of software-driven recovery mechanisms. For embedded systems operating under real-time and safety-critical constraints, such balance is often necessary to maintain deterministic behavior without incurring excessive hardware overhead.

(Mahmood & Khairullah, 2025) proposed a Triple Modular Redundancy (TMR)-based architecture for embedded cyber-physical systems, achieving high reliability with a reported hardware failure rate of 0.03 failures per hour. Their design employed strategic voter and monitoring circuits to reduce the performance penalties typically associated with full hardware replication. Although this work remains firmly rooted in hardware redundancy, it provides valuable insight into deterministic decision logic and monitoring strategies that are essential in safety-critical systems. From the perspective of microcontroller-based platforms, the study highlights how monitoring and decision-making components, not just redundant hardware, play a central role in overall system dependability. These lessons are particularly relevant when such monitoring logic is migrated from hardware into software, as is the case in lightweight Arduino-based systems.

Complementing this approach, (Csaba, 2025) explored a warm redundancy strategy in a twin power converter control system. Rather than relying on cold standby mechanisms that require full system restarts, the proposed design enabled seamless switching between controllers during transient fault conditions. This work underscores the importance of state-aware redundancy, where backup controllers remain partially active and synchronized, allowing rapid takeover without disrupting real-time operation. For distributed microcontroller systems, this concept translates naturally into software-controlled role switching**,** where standby nodes continuously monitor system health and maintain readiness to assume control. Such behavior closely aligns with heartbeat-based redundancy models, where failover decisions are made dynamically based on observed system state rather than fixed hardware roles.

(Grün, 2025) examined embedded fault handling within defense systems, focusing on communication-level diagnostics and software-based validation techniques, including Cyclic Redundancy Check (CRC) for serial data integrity and fault injection testing. This work emphasizes that, in harsh environments, communication faults are as critical as processing faults,

and that timely detection at the protocol level is essential for maintaining system reliability. These findings have direct implications for Arduino-based distributed systems, where serial, I²C, or SPI communication links form the backbone of inter-node coordination. Incorporating lightweight diagnostics and continuous monitoring into communication protocols strengthens fault awareness without increasing hardware complexity.

Taken together, these recent studies demonstrate that effective fault tolerance increasingly relies on intelligent monitoring, rapid role transition, and communication-aware diagnostics, rather than brute-force hardware duplication alone. However, despite their contributions, most hybrid models still assume specialized hardware support, tightly coupled controllers, or domain-specific architectures. There remains limited exploration of how these principles can be distilled into purely software-driven redundancy mechanisms for low-cost, resource-constrained microcontrollers operating without centralized coordination.

This gap is particularly evident in Arduino-class systems, where hardware replication is often impractical and external supervisory controllers introduce additional points of failure. The Arduino-to-Arduino redundancy protocol proposed in this thesis draws inspiration from hybrid models but deliberately shifts the balance further toward software. By combining heartbeat-based monitoring, distributed decision-making, and autonomous role switching, the system achieves many of the benefits of hybrid redundancy while avoiding the complexity and fragility associated with additional hardware components.

In this sense, the proposed approach can be viewed as a lightweight hybridization at the architectural level rather than the hardware level. Minimal hardware assumptions are paired with robust software diagnostics and monitoring. This design philosophy aligns well with the constraints of space environments, where simplicity, autonomy, and resilience are paramount.

2.1.6 Relevance to Arduino-based Systems

Microcontroller platforms such as Arduino operate under strict resource constraints, including limited memory, modest processing capability, and the absence of sophisticated operating system support. While these platforms are attractive due to their simplicity, low cost, and ease of integration, these same characteristics make the direct application of traditional hardware-based fault tolerance techniques impractical. Approaches such as Triple Modular Redundancy or dedicated supervisory hardware introduce prohibitive overheads in terms of power consumption, physical space, and system complexity, which are incompatible with Arduino-class devices.

As a result, fault tolerance in microcontroller-based systems has increasingly shifted toward software-centric solutions. Prior studies have demonstrated that software level mechanisms, such as watchdog timers, execution monitoring, and application layer fault detection can significantly enhance system robustness without requiring additional hardware resources (Györök & Beszédes,

2017). These techniques are particularly well suited to small, embedded controllers, where careful software design often provides the only feasible path to improved dependability.

In distributed microcontroller systems, decentralized fault detection plays a critical role. Rather than relying on a centralized controller, each node can independently monitor system health and participate in recovery decisions. Application-level fault tolerance frameworks for embedded systems have shown that such distributed approaches are both effective and lightweight, provided that monitoring and recovery logic is kept simple and deterministic (Afonso et al., 2008). This design philosophy aligns naturally with Arduino-based networks, where simplicity and autonomy are essential.

One of the most practical mechanisms for enabling distributed fault awareness in resource-constrained systems is heartbeat-based status signaling. In this approach, nodes periodically exchange short messages to indicate continued correct operation, and the absence of a heartbeat within a defined time window is interpreted as a failure. Heartbeat monitoring has been widely used in embedded and real-time systems due to its low computational overhead and predictable behavior (Bucciero et al., 2011). Importantly, similar concepts have already been demonstrated on Arduino-based platforms in non-space domains, confirming the feasibility of heartbeat detection between low-cost microcontrollers (Bowoto et al., 2019).

Beyond fault detection, dynamic role reassignment and failover are essential for maintaining system functionality once a fault has been identified. Software-controlled role switching, where standby nodes assume control responsibilities upon detecting failure, enables continuity of operation without requiring specialized hardware support. Such mechanisms mirror warm-standby concepts explored in hybrid redundancy models but can be implemented entirely in software when supported by reliable internode communication and state awareness (Afonso et al., 2008).

The relevance of these techniques is particularly pronounced in space environments, where embedded systems are exposed to radiation-induced transient faults, unpredictable delays, and communication disturbances. Recent space-oriented research emphasizes that hardware only fault tolerance is increasingly insufficient for modern embedded platforms, and that software-based monitoring and recovery mechanisms are essential for achieving resilience under such conditions (Fuchs et al., 2017).

This thesis builds on these findings by adopting a software-only, distributed redundancy approach tailored specifically to Arduino-class microcontrollers. By combining heartbeat-based monitoring, decentralized fault detection, and dynamic role reassignment, the proposed Arduino-to-Arduino redundancy protocol demonstrates how effective fault tolerance can be achieved with minimal hardware overhead. While related concepts have been explored in higher-end embedded systems and non-space applications, their systematic application to low-cost, peer-to-peer microcontroller

networks for space environments remains largely unexplored, forming the central motivation for this research.

2.2 Redundancy Mechanisms in Embedded Networks

Redundancy in embedded networks plays a critical role in ensuring continued system operation in the presence of faults. In safety-critical domains such as aviation, industrial automation, and defense systems, redundancy has traditionally been achieved through hardware-centric approaches including hot standby, cold standby, and duplex or triplex configurations. These strategies rely on replicated components and dedicated supervisory logic to detect failures and coordinate recovery. While effective, such approaches were originally designed for systems with relatively abundant computational and hardware resources.

Applying these same principles to lightweight microcontroller networks, particularly those built around Arduino platforms, presents a distinct set of challenges. Arduino-class devices operate with limited memory, processing power, and communication bandwidth, and typically lack native support for advanced fault-management infrastructure. As a result, redundancy mechanisms must be simplified and carefully adapted to avoid overwhelming the platform while still providing meaningful fault tolerance.

2.2.1 Hot vs Cold Standby in Microcontroller Systems

Within embedded networks, redundancy is commonly categorized into hot standby and cold standby configurations. In a hot standby system, the redundant node remains active and synchronized with the primary node, allowing it to assume control almost immediately after a failure. In contrast, cold standby systems keep the backup node inactive until a fault is detected, reducing power consumption at the cost of increased recovery latency.

Several studies have demonstrated that hot standby redundancy is feasible even on Arduino platforms provided that synchronization logic is kept lightweight. (Sharpe et al., n.d.). implemented a hot standby architecture in an Arduino-controlled irrigation system, where a secondary Arduino continuously mirrored sensor inputs and actuator states from the primary controller. Their system achieved recovery times below 500 ms, illustrating that real-time failover is achievable on modest 8-bit microcontrollers through continuous inter-node communication.

In contrast, (Bowoto et al., 2019). explored a cold-style redundancy approach using two Arduinos in an IoT fault-tolerant system, where a secondary node only assumed control after heartbeat failure was detected. While this approach reduced energy consumption, the authors reported noticeable delays during role transitions, highlighting the inherent trade-off between power efficiency and responsiveness in Arduino-based redundancy schemes.

Together, these studies suggest that while both hot and cold standby models are technically viable on Arduino platforms, hot standby approaches are better suited to time-critical applications, whereas cold standby may be acceptable in systems where brief interruptions can be tolerated.

2.2.2 Trade-offs: Recovery Time, Communication Load, and Complexity

Introducing redundancy into Arduino networks inevitably increases system complexity. Hot standby systems require continuous synchronization between nodes, often implemented using UART, I²C, or SPI communication. This persistent data exchange increases bus utilization and introduces additional points of failure at the communication layer. Furthermore, Arduino platforms provide limited support for arbitration and collision avoidance, making robust multi-node coordination challenging.

More advanced redundancy schemes, such as *triplex configurations with majority voting*, are largely impractical on Arduino platforms due to limited I/O availability, constrained memory, and the absence of dedicated voting hardware. However, simplified duplex redundancy has been explored. (Sharma & Yadav, 2018) proposed a lightweight software-based coordination mechanism using watchdog timers and parallel heartbeat loops, demonstrating that basic fault masking can be achieved without full hardware voting. Their work reinforces the idea that software coordination can partially substitute for hardware redundancy in resource-constrained systems.

2.2.3 Approximate Fault Tolerance in Arduino Networks

An emerging concept relevant to Arduino-based redundancy is Adaptive Fault Tolerance (AFT). Rather than enforcing strict correctness or timing guarantees under all conditions, AFT allows fault-tolerance mechanisms to adapt dynamically, accepting temporary inaccuracies, delayed responses, or degraded performance in order to preserve system operation and reduce resource overhead. This approach has been demonstrated in resilient embedded and distributed systems, where fault-tolerance strategies evolve at runtime in response to resource availability and fault conditions (Lauer et al., 2016). These principles translate naturally to Arduino networks, where intermittent communication loss or delayed role switching may be acceptable as long as the system's essential functionality is preserved.

2.2.4 Peer-to-Peer Redundancy in Arduino Networks

Most Arduino-based networks are traditionally designed using master-slave architectures, where a single controller coordinates all subordinate nodes. While simple, this structure introduces a critical single point of failure. Recent research has begun to explore peer-to-peer redundancy models, where nodes mutually monitor one another and dynamically assume control when failures occur. (Bowoto et al., 2019) demonstrated a basic mutual monitoring model in which two Arduinos exchanged heartbeat signals and dynamically declared a master role when a failure was detected. While effective as a proof of concept, the system relied on external communication infrastructure and did not address strict timing guarantees or autonomous recovery under harsh environmental conditions. More broadly, research on distributed embedded systems emphasizes that peer-to-peer

fault detection improves resilience by eliminating centralized decision points, particularly in environments prone to intermittent faults and communication disturbances (Rennels et al., 1997). However, these principles have largely been explored using more capable microcontrollers or hybrid hardware-software architectures, rather than minimalist Arduino platforms.

2.2.5 Identified Research Gap

Despite growing interest in Arduino-based fault tolerance, existing studies remain limited in scope. Most implementations either:

- rely on external infrastructure,
- assume relaxed timing constraints,
- or focus on non-critical application domains such as IoT monitoring.

There remains a clear gap in the literature for fully autonomous, Arduino-to-Arduino redundancy protocols that:

- operate without centralized coordinators,
- provide deterministic heartbeat-based failure detection,
- enable rapid role reassignment,
- and are designed for harsh environments such as space systems in mind.

This thesis addresses that gap by proposing a direct, peer-to-peer Arduino redundancy protocol, where each node acts simultaneously as controller and observer. By combining lightweight heartbeat communication with distributed decision-making, the proposed approach builds on existing research while extending it into a domain that has so far received limited focused investigation.

2.3 Heartbeat Monitoring and Failure Detection Algorithms

Reliable failure detection is a cornerstone of fault-tolerant embedded networks. In distributed microcontroller systems, particularly those composed of low-cost devices such as Arduinos, detecting node failures must be achieved with minimal computational overhead, predictable timing behavior, and limited communication bandwidth. Heartbeat-based monitoring has emerged as one of the most widely adopted techniques for this purpose, due to its simplicity and suitability for real-time embedded environments.

Heartbeat monitoring relies on the periodic exchange of short status messages between nodes to indicate continued correct operation. When a heartbeat is not received within a predefined time window, the monitoring node infers that a failure has occurred. This method has been extensively studied in distributed embedded systems literature and is often preferred over more complex diagnostic approaches in resource-constrained platforms (Mishra & Khilar, 2012).

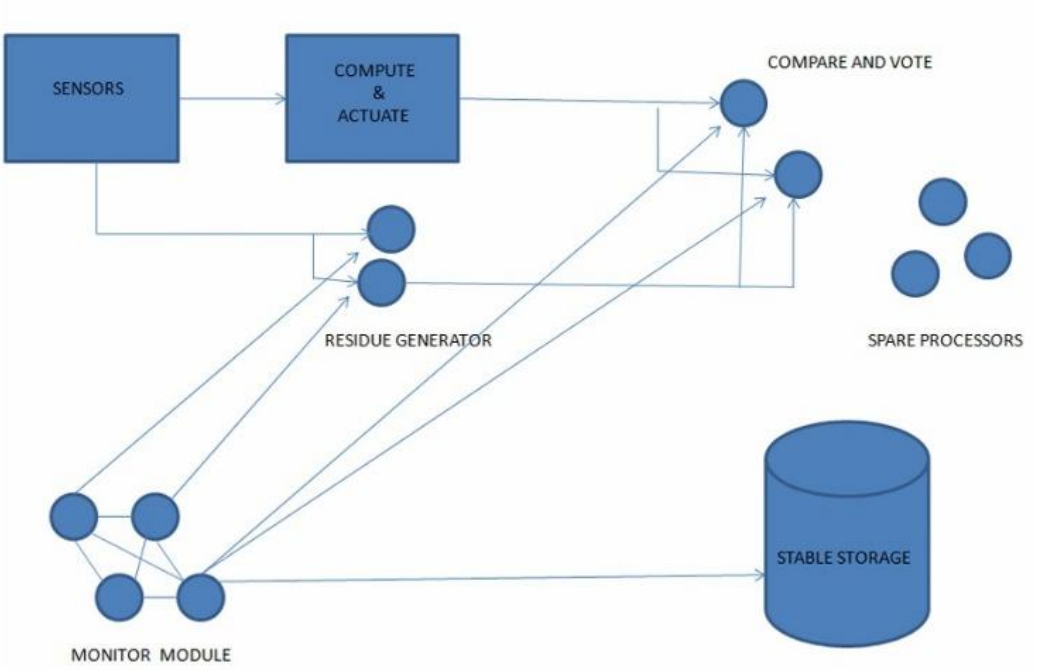

Figure 1. *System model of the fault diagnosis framework proposed by Mishra & Khilar.*

As shown in Figure 1, (Mishra & Khilar, 2012) present a fault diagnosis framework for distributed embedded systems that integrates redundancy, model-based verification, and centralized monitoring. Figure 1 illustrates the system architecture, highlighting the interaction between redundant sensors, compute-and-actuate modules, residue generators, voting units, and a centralized monitoring module. Sensor data are processed both by the actuator logic and by analytical models, and discrepancies between these outputs are evaluated through a voting mechanism to identify faulty actuators.

A notable strength of this framework is its use of analytical redundancy, which enables fault detection without requiring complete hardware replication. By validating actuator outputs against model-derived references, the system can detect unsafe behavior and isolate faulty components before faults propagate. This approach is particularly suitable for safety-critical embedded applications where correctness of control actions is paramount.

Nevertheless, the framework primarily detects faults after functional deviations occur, making it inherently reactive. In addition, its reliance on centralized monitoring and continuous model evaluation introduces scalability and resource concerns when applied to lightweight microcontroller networks. In Arduino-based or similar low-cost embedded systems, processing power, memory, and communication reliability are limited, and assumptions of stable connectivity and sustained analytical computation may not always hold.

These limitations highlight the need for complementary fault detection mechanisms that are better aligned with constrained embedded platforms. Lightweight techniques such as heartbeat monitoring can provide early indications of node-level failures, tolerate transient communication delays, and support graceful degradation. Such characteristics are especially relevant in

microcontroller networks, where maintaining system continuity is often prioritized over strict real-time precision.

2.3.1 Timeout Models and Adaptive Thresholds

A critical design parameter in heartbeat-based systems is the timeout threshold used to declare a node faulty. Fixed timeout values are simple to implement, but they are vulnerable to false positives when communication delays or transient disturbances occur. This problem is particularly relevant in embedded serial communication links, such as UART-based (Universal Asynchronous Receiver-Transmitter, a serial communication protocol) Arduino-to-Arduino connections, where timing jitter and buffering delays are common.

(Mishra & Khilar, 2012) proposed adaptive timeout models that account for message delays and loss, reducing unnecessary failover events while maintaining timely fault detection. Their work demonstrates that even lightweight systems can benefit from dynamically tuned heartbeat intervals, provided the algorithm remains computationally simple. While their framework was validated through simulation rather than physical microcontroller networks, the underlying principles translate well to Arduino-based systems, where conservative timeout selection is necessary to balance responsiveness and stability.

In practical Arduino implementations, most existing systems continue to rely on static heartbeat intervals and fixed timeouts. This simplification reduces code complexity but limits robustness under varying load or environmental conditions, an important consideration for space-oriented embedded systems where delays may not be constant.

2.3.2 Unidirectional vs. Bidirectional Heartbeat Models

Heartbeat mechanisms can be broadly categorized as unidirectional or bidirectional. In unidirectional models, a monitored node periodically transmits heartbeats to a supervisor. This approach is simple, but it introduces a single point of failure at the monitoring node. Bidirectional models, in contrast, involve mutual heartbeat exchanges, allowing nodes to monitor each other's liveness and detect failures in a more decentralized manner.

Research in distributed embedded systems increasingly favors bidirectional heartbeat models due to their improved fault coverage and resilience. This approach is especially attractive for Arduino-to-Arduino communication, where peer-to-peer links are easy to establish using serial interfaces and where centralized supervision may be undesirable or infeasible.

Several Arduino-based fault-tolerant designs adopt implicit bidirectional monitoring through acknowledgment messages or watchdog resets, but these implementations are often tightly coupled to specific applications and lack formalized role-switching logic (Bowoto et al., 2019). As a result, failure detection is achieved, but recovery behavior remains limited or ad-hoc.

2.3.3 Dynamic Role-Switching and Failover Algorithms

Detecting a failure is only the first step in achieving fault tolerance. The system must also decide how and when to reassign control. Dynamic role-switching algorithms enable a standby node to assume the role of a failed controller without external intervention. In embedded microcontroller networks, this process must be deterministic and fast, yet simple enough to be implemented without an operating system or complex middleware.

(Rennels et al., 1997) demonstrated early role-based recovery mechanisms in fault-tolerant microcontroller nodes, where modules dynamically transitioned between master, slave, and offline states based on heartbeat loss and timeout events. While their architecture relied on additional logic and multiple redundancy channels, the concept of state-based role reassignment remains highly relevant for Arduino-class systems. The authors describe a fault-tolerant microcontroller node architecture in which multiple redundant processor modules share common digital and analog I/O connections to external sensors and actuators, as shown in Figure 2. All input signals are broadcast to every module, enabling independent sampling and comparison, while output lines are bidirectional and dynamically configurable. Under normal operation, only a designated Master module is permitted to drive the output lines, whereas Slave modules monitor these outputs as inputs in order to perform consistency checks.

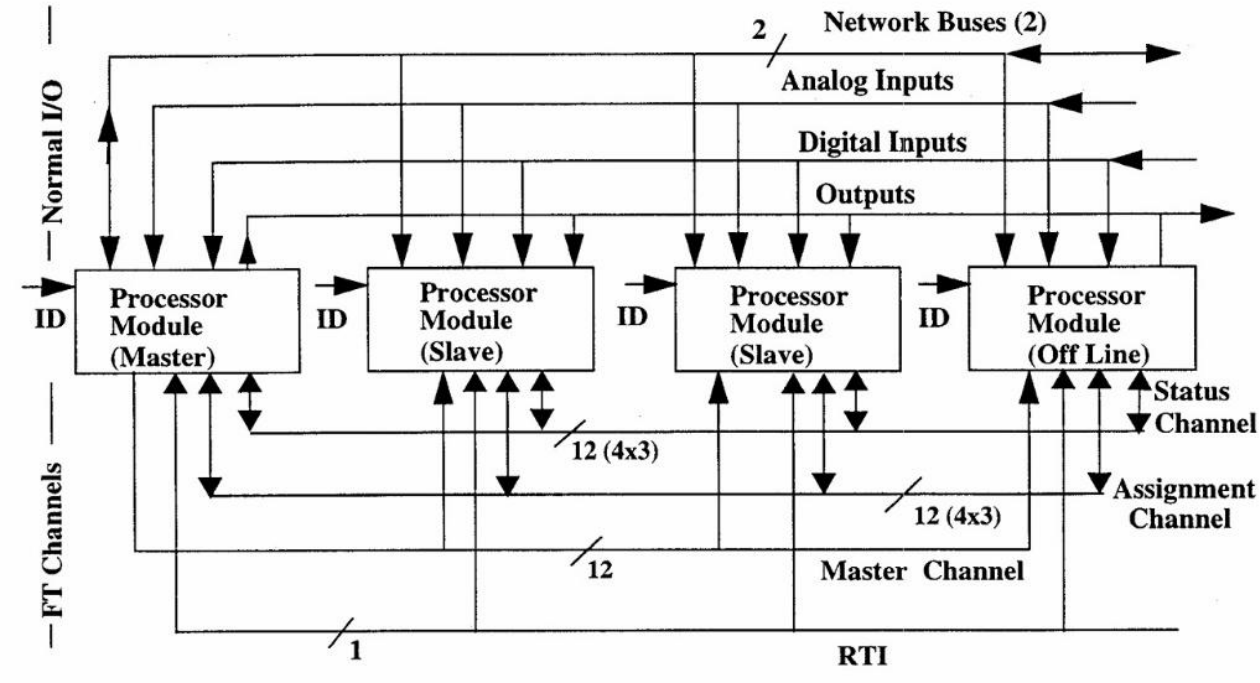

Figure 2. *Redundant microcontroller node architecture illustrating multiple processor modules (master, slave, and offline) interconnected via shared I/O buses and coordination channels for fault detection and role reassignment (adapted from Rennels et al.)*

A key strength of this design is its strict containment of faulty behavior at the hardware level. Output generation is protected by an independent hardware enable circuit within each module, ensuring that a single faulty or "babbling" processor cannot arbitrarily assert outputs. Master authority is granted only after a voting process on a dedicated assignment channel, requiring agreement from multiple modules before output drivers are enabled. This hardware-supported voting mechanism significantly reduces the risk of unsafe actuation and provides strong protection against single-point failures.

However, this robustness is achieved at the cost of architectural complexity and hardware overhead. The reliance on multiple dedicated fault-tolerance channels, additional voting logic, and redundant enable circuitry increases system cost, wiring complexity, and power consumption. Such assumptions are less compatible with Arduino-class microcontroller networks, where simplicity, low pin count, and minimal external logic are often primary design constraints.

Furthermore, the approach presumes reliable and tightly synchronized communication among modules to support real-time comparison and voting. In loosely coupled or resource-constrained microcontroller networks, transient communication delays or partial connectivity loss are common and may not justify immediate master re-election or hardware-level isolation. As a result, while the Rennels et al. architecture provides strong guarantees of correctness and safety, it is less adaptable to lightweight embedded systems that prioritize scalability, tolerance of temporary inconsistencies, and graceful degradation.

These limitations motivate the exploration of lighter-weight coordination and failure detection mechanisms, such as heartbeat-based monitoring, which reduce hardware dependencies while still enabling timely detection of node failures. Such approaches are particularly well suited to Arduino-based distributed systems, where maintaining operational continuity with minimal overhead is often more practical than enforcing strict, hardware-driven voting at every I/O interaction.

2.3.4 Implications for Arduino-to-Arduino Communication

Across the reviewed literature, heartbeat-based failure detection is well established as a reliable and lightweight mechanism for distributed embedded systems. Nevertheless, most existing implementations either target simulated environments, rely on RTOS enabled microcontrollers, or assume additional hardware support. Fully autonomous Arduino-to-Arduino heartbeat protocols**,** operating in a peer-to-peer manner with deterministic role switching and minimal software overhead, remain relatively underexplored.

This gap is particularly significant for environments such as space systems, where external supervision is unavailable, communication delays are variable, and recovery actions must be taken locally and reliably. The lack of standardized, reusable heartbeat-driven redundancy protocols for Arduino-class devices motivates the design explored in this thesis. By focusing on direct serial

communication, bidirectional heartbeat monitoring, and software-only role reassignment, the proposed solution builds upon established research while addressing a clear and practical limitation in current Arduino-based fault-tolerant systems.

2.4 Arduino Communication Capabilities and Limitations

Reliable inter-node communication is the foundation of any redundancy mechanism in embedded systems. In Arduino-based architectures, heartbeat monitoring, dynamic role switching, and fault detection depend entirely on how reliably nodes exchange information. Therefore, understanding the strengths and limitations of Arduino's native communication interfaces is essential when designing a robust Arduino-to-Arduino failover protocol.

2.4.1 Serial vs. Parallel Communication

Before examining specific interfaces, it is useful to distinguish serial and parallel communication. Parallel communication transmits multiple bits simultaneously across multiple wires, increasing theoretical throughput but requiring more pins, tighter timing synchronization, and more complex routing. As distances increase, signal skewing and crosstalk become significant issues. As illustrated in Figure 4, parallel communication relies on multiple dedicated data lines transmitting bits concurrently, along with a shared clock signal to maintain synchronization between the transmitter and receiver.

Serial communication, by contrast, transmits data one bit at a time over fewer wires, reducing hardware complexity and improving noise tolerance. As shown in Figure 3, data is sent sequentially over a single channel and synchronized using a clock signal, making it more suitable for resource-constrained microcontroller systems (New Haven Display International, 2025).

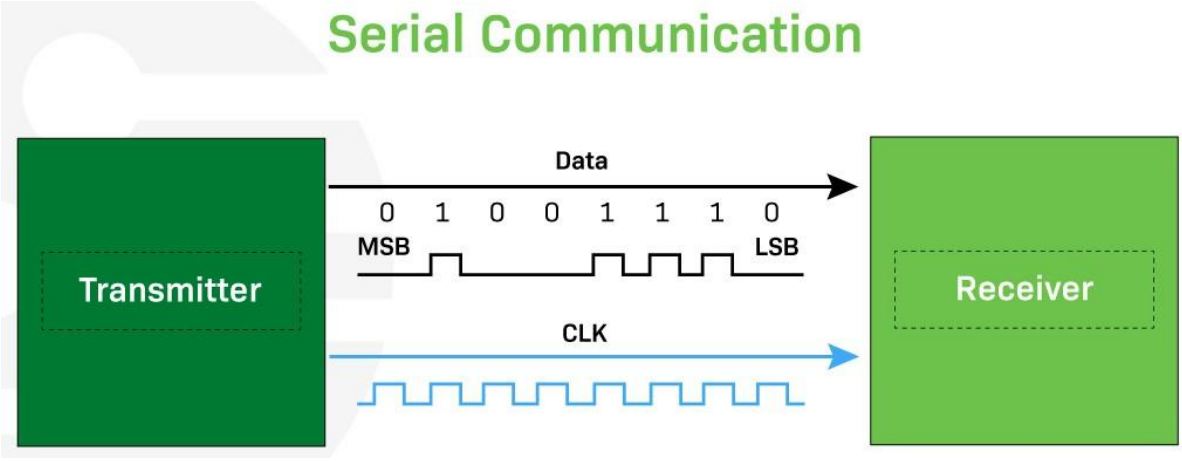

Figure 3. *A transmitter sends one bit at a time to the receiver for every pulse clock (CLK). The receiver interprets the bits as the binary number 01001110 (adapted from New Haven Display International, 2025)*

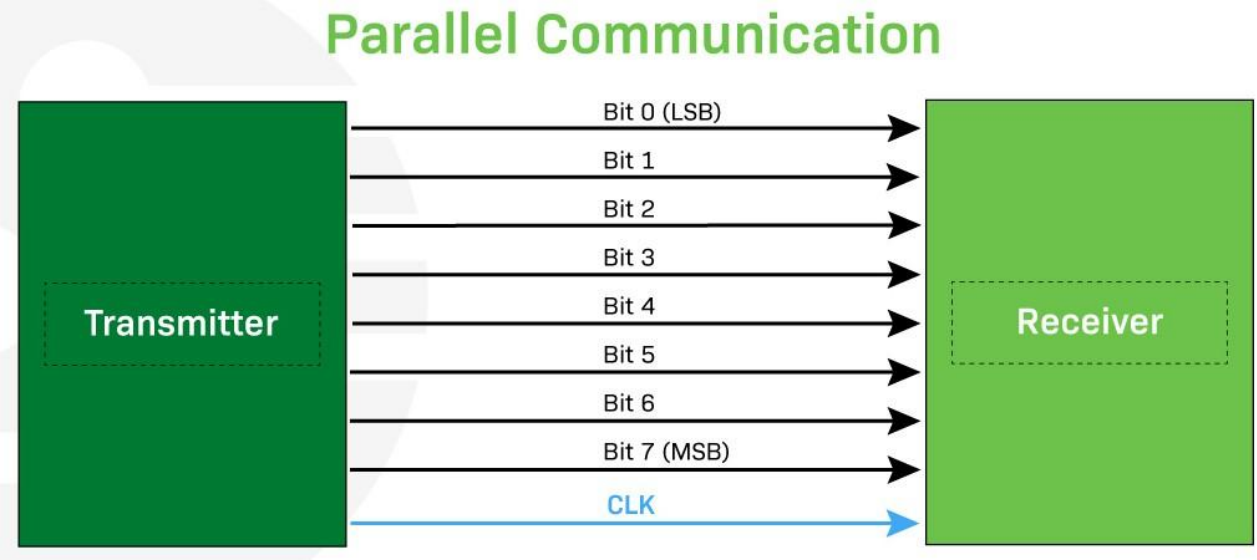

Figure 4. *Parallel communication simultaneously sends multiple bits to the receiver for every clock pulse (CLK). The receiver interprets the data as the binary number 01001110 (adapted from New Haven Display International, 2025)*

Because Arduino platforms are I/O constrained, serial communication dominates in embedded redundancy designs.

2.4.2 Universal Asynchronous Receiver-Transmitter (UART)

UART is the most straightforward and widely used serial interface in Arduino systems. It is asynchronous, meaning no shared clock line is required. Instead, both devices agree on a baud rate, and each data frame includes start and stop bits for synchronization (New Haven Display International, 2025). It is a block of circuitry responsible for implementing serial communication. Essentially, the UART acts as an intermediary between parallel and serial interfaces. At one end of the UART is a bus of eight-or-so data lines (plus some control pins), on the other is the two serial wires - RX and TX (Jim Blom, n.d.).

As illustrated in Figure 5, the UART interface bridges parallel and serial communication domains by converting multi-bit parallel data from the microcontroller's internal data bus into a serialized bit stream transmitted over a single TX line, while incoming serial data on the RX line is reconstructed into parallel form. The figure highlights how control signals and clock-independent operation enable asynchronous communication, making UART well-suited for simple, low-overhead data exchange in resource-constrained embedded systems

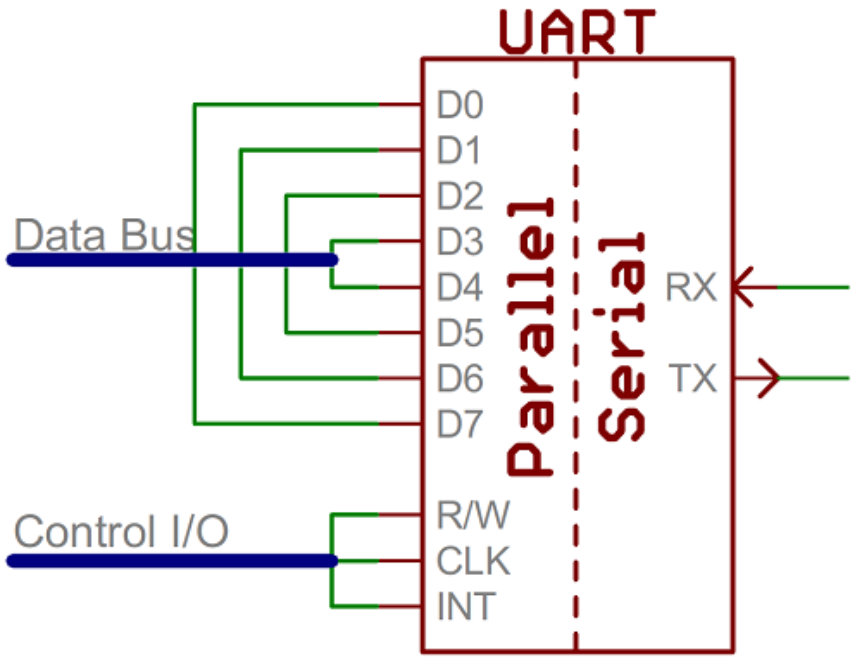

Figure 5. *Simplified UART interface showing parallel-to-serial data conversion between a microcontroller data bus and serial TX/RX lines for asynchronous communication (adapted from (Jim Blom, n.d.)).*

UART typically requires only TX to transmit, RX to receive, and a common ground. This minimal wiring and native hardware support make UART ideal for heartbeat-based redundancy, were small, periodic status messages must be exchanged predictably. Peer-reviewed work supports this approach. (Bowoto et al., 2019) demonstrated heartbeat-based failure detection between Arduino nodes using UART links, showing that simple serial communication is sufficient for small, distributed redundancy systems.

However, UART has clear limitations, these include:

- It is inherently point-to-point, making multi-node scaling complex.
- It lacks strong built-in error detection beyond optional parity.
- It is vulnerable to noise over longer distances without additional shielding or validation.

For safety-critical contexts, higher-layer integrity checks are therefore essential.

2.4.3 Serial Peripheral Interface (SPI)

Serial Peripheral Interface (SPI) is an interface bus commonly used to send data between microcontrollers and small peripherals such as shift registers, sensors, and SD cards. It uses separate clock and data lines, along with a select line to choose the device you wish to talk to.

A common serial port, the kind with TX and RX lines, is called "asynchronous" (not synchronous) because there is no control over when data is sent or any guarantee that both sides are running at precisely the same rate. Since computers normally rely on everything being synchronized to a

single “clock” (the main crystal attached to a computer that drives everything), this can be a problem when two systems with slightly different clocks try to communicate with each other.

To work around this problem, asynchronous serial connections add extra start and stop bits to each byte help the receiver sync up to data as it arrives. Both sides must also agree on the transmission speed (such as 9600 bits per second) in advance (Mike Grusin, n.d.).

As illustrated in Figure 6, asynchronous serial communication transmits data in framed packets consisting of start bits, data bits, and stop bits without a shared clock signal. The receiver relies on agreed timing (baud rate) to sample incoming bits, which introduces potential synchronization errors if timing deviates. This framing mechanism simplifies wiring but makes communication more susceptible to jitter and timing mismatches compared to synchronous protocols.

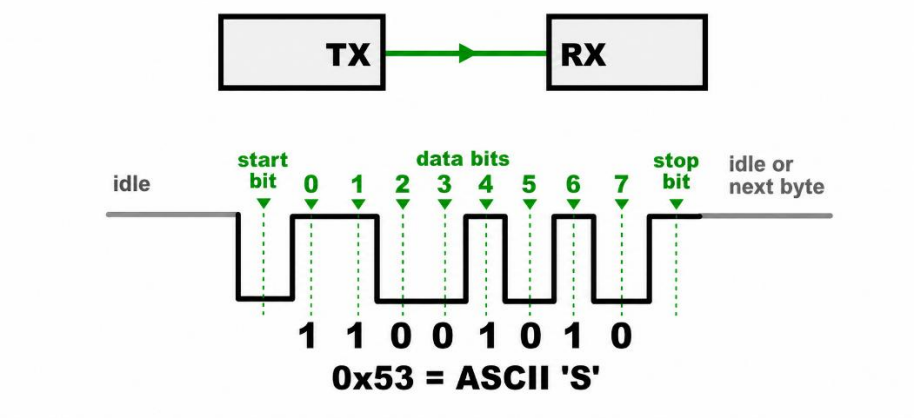

Figure 6. *Asynchronous serial data transmission showing start bit, data bits, and stop bit framing without a shared clock signal (Mike Grusin, n.d.)*

SPI works in a slightly different manner. It's a "synchronous" data bus, which means that it uses separate lines for data and a "clock" that keeps both sides in perfect sync. The clock is an oscillating signal that tells the receiver exactly when to sample the bits on the data line.

In SPI, only one side generates the clock signal (usually called CLK or SCK for Serial Clock). The side that generates the clock is called the "controller", and the other side is called the "peripheral". There is always only one controller (which is almost always the microcontroller), but there can be multiple peripherals.

When data is sent from the controller to a peripheral, it's sent on a data line called PICO, (Peripheral In / Controller Out). If the peripheral needs to send a response back to the controller, the controller will continue to generate a prearranged number of clock cycles, and the peripheral will put the data onto a third data line called POCI, (Peripheral Out / Controller In).

As shown in Figure 7, SPI communication uses a shared clock (SCK) to synchronize data transfer between the controller and peripheral devices. Data is transmitted simultaneously in both directions via separate lines controller-to-peripheral (PICO) and peripheral-to-controller (POCI). Each clock pulse shifts one bit, enabling deterministic and high-speed full-duplex communication.

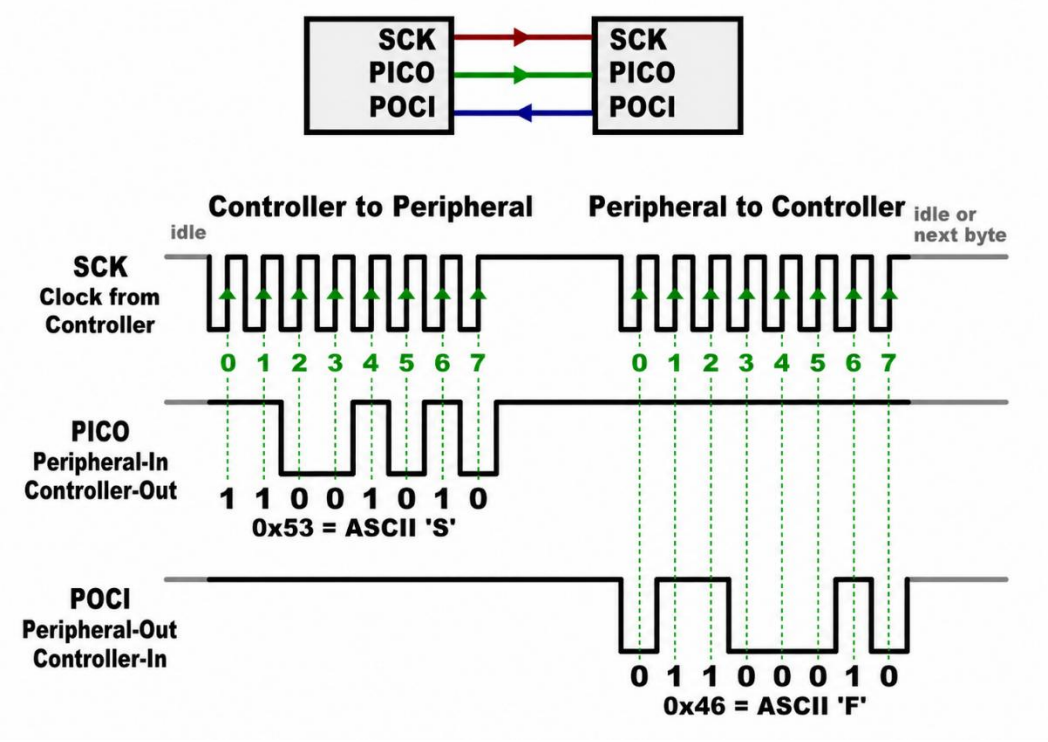

Figure 7. *SPI data transfer illustrating synchronized full-duplex communication using clock (SCK), controller-to-peripheral (MOSI/PICO), and peripheral-to-controller (MISO/POCI) lines (Mike Grusin, n.d.)*

Because the controller always generates the clock signal, it must know in advance when a peripheral needs to return data and how much data will be returned. This is very different than asynchronous serial, where random amounts of data can be sent in either direction at any time.

One additional called the CS, (Chip Select), tells the peripheral that it should wake up and receive or send data and is also used when multiple peripherals are present to select the one to talk to.

In Figure 8, the chip select (CS) line enables the controller to activate a specific peripheral while keeping others inactive on the shared bus. This mechanism allows multiple devices to share the same clock and data lines without interference, ensuring controlled and collision-free communication in multi-device SPI configurations.

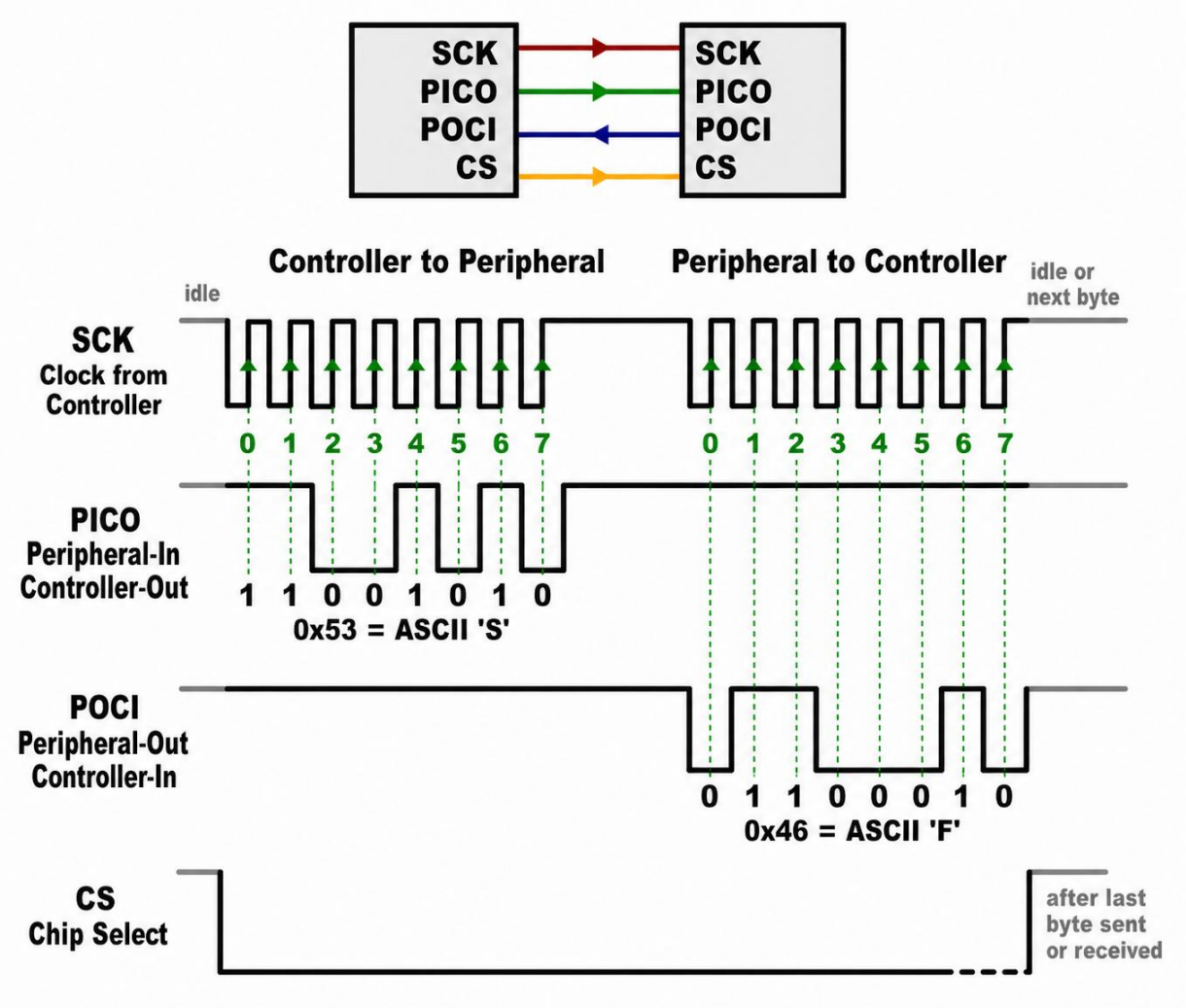

Figure 8. *SPI communication with chip select (CS) line demonstrating selective activation of peripherals on a shared bus (Mike Grusin, n.d.)*

2.4.3.1 Multiple Peripherals

There are two ways of connecting multiple peripherals to an SPI bus:

1. **Dedicated Chip-Select Line Configuration:**
   In general, each peripheral will need a separate CS line. To talk to a particular peripheral, that peripheral's CS line will be made low and the CS lines of other peripherals kept high. (two peripherals should not be activated at the same time, or they may both try to talk on the same POCI line resulting in garbled data). However, lots of peripherals will require lots of CS lines.

   As illustrated in Figure 9, each peripheral is connected to the controller through shared clock (SCK) and data lines (PICO and POCI), while individual chip-select (CS) lines are used to enable only one device at a time, preventing bus contention and ensuring controlled communication.

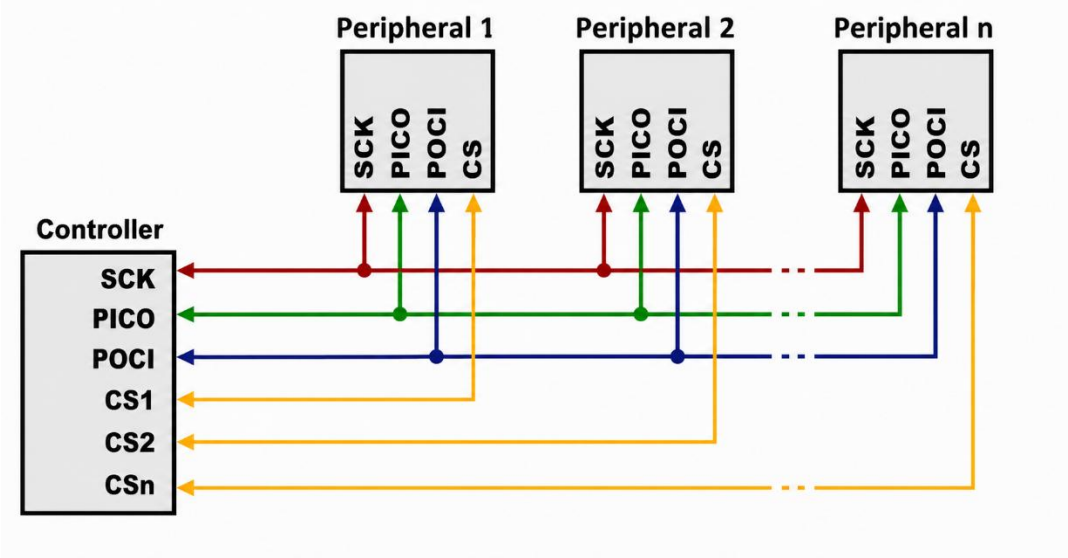

Figure 9. *SPI multiple peripheral configurations using dedicated chip-select (CS) lines, where each peripheral is individually enabled while sharing common clock and data lines (Mike Grusin, n.d.)*

2. **Shared Chip-Select SPI Configuration:**
   On the other hand, some parts prefer to be daisy-chained together, with the POCI (output) of one going to the PICO (input) of the next. In this case, a single CS line goes to all the peripherals. Once all the data is sent, the CS line is raised, which causes all the chips to be activated simultaneously. This is often used for daisy-chained shift registers and addressable LED drivers.
   As shown in Figure 10, peripherals are connected in a daisy-chain arrangement where data propagates sequentially from one device to the next via shared data lines, and a single chip-select signal controls all devices simultaneously, requiring the transmission of aggregated data to reach a specific peripheral.

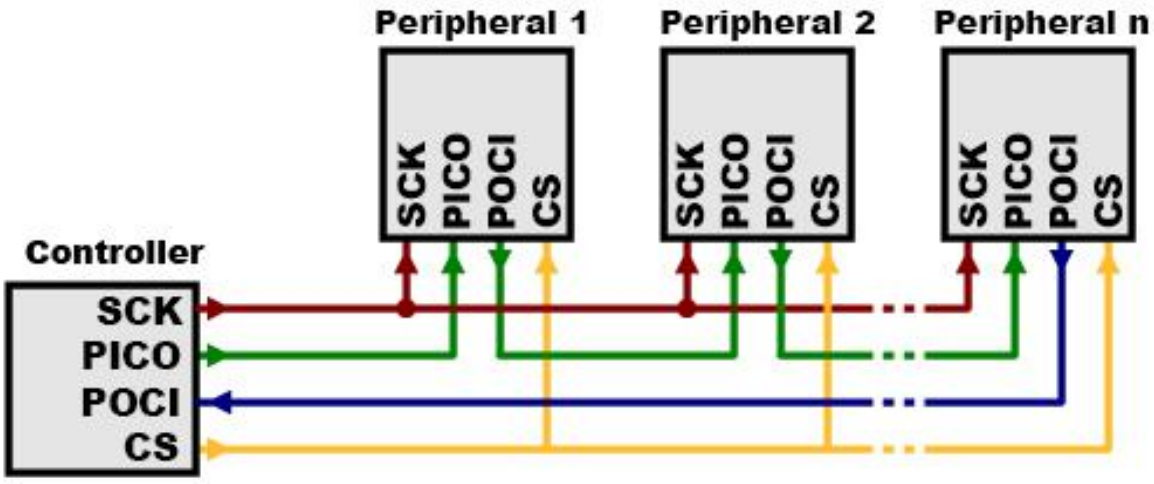

Figure 10. *Daisy-chained SPI configuration with a shared chip-select (CS) line, where data shifts sequentially through multiple peripherals on a common bus (Mike Grusin, n.d.)*

Note that, for this layout, data overflows from one peripheral to the next, so to send data to any one peripheral, enough data needs to be transmitted to reach all of them. Also, the first piece of data transmitted will end up in the last peripheral. This type of layout is typically used in output-only situations, such as driving LEDs where there's no need to receive any data back.

SPI offers several practical advantages, particularly in embedded systems. It is significantly faster than asynchronous serial communication because it uses a shared clock to synchronize data transfer, allowing high-speed, full-duplex communication. The receive hardware can be implemented using a simple shift register, making it efficient and relatively easy to integrate at the hardware level. Additionally, SPI supports multiple peripherals on the same bus, enabling a single controller to communicate with several devices. However, these benefits come with trade-offs. SPI requires more signal lines than protocols like I²C or UART, increasing wiring complexity. Communication parameters must be clearly defined in advance, as SPI does not naturally support flexible or variable-length data exchanges. The protocol is also strictly controller driven, meaning peripherals cannot communicate directly with each other. Furthermore, most implementations require separate chip-select (CS) lines for each peripheral, which can become problematic in systems with many devices due to limited I/O pins, an important consideration in Arduino-based redundancy designs.

2.4.4 I²C (Inter-Integrated Circuit)

The Inter-Integrated Circuit (I^2C) Protocol is a protocol intended to allow multiple peripheral digital integrated circuits (chips) to communicate with one or more controller chips. Like the Serial Peripheral Interface (SPI), it is only intended for short distance communications within a single device. Like Asynchronous Serial Interfaces (UARTs), it only requires two signal wires to exchange information.

I²C communication illustrated in figure 11 uses two shared lines serial data (SDA) and serial clock (SCL) to connect multiple controllers and peripherals on the same bus. All devices are connected in parallel to these lines, enabling addressed communication where each peripheral responds only when its unique address is called. This shared-bus architecture reduces wiring complexity while supporting multi-device and multi-controller configurations.

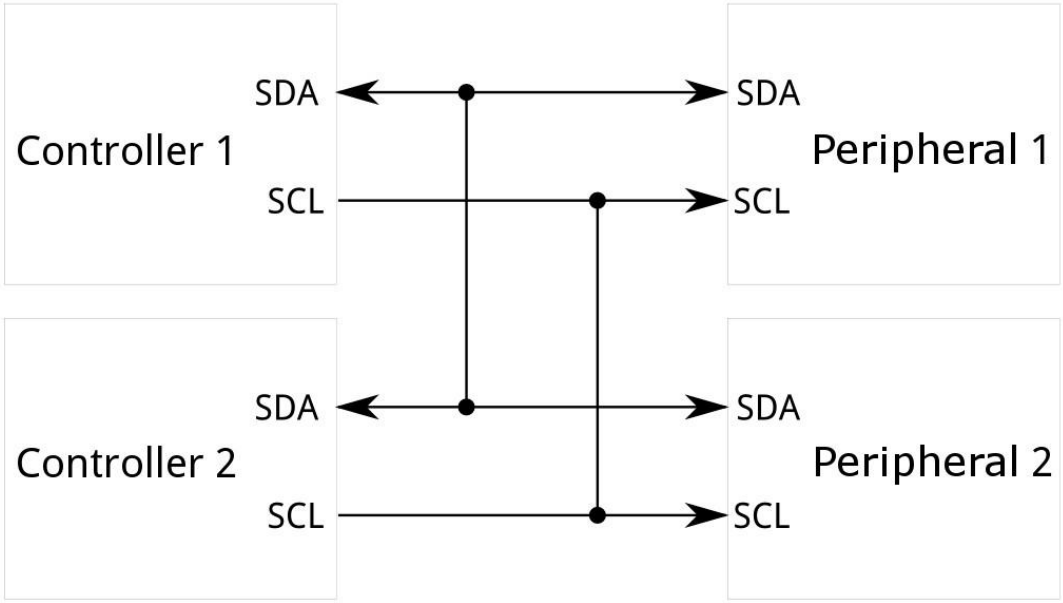

Figure 11. *I²C bus architecture showing multiple controllers and peripherals connected via shared SDA (data) and SCL (clock) lines enabling addressed multi-device communication (SparkFun Electronics, n.d.)*

Unlike asynchronous serial communication, I²C uses a shared clock (SCL) to synchronize data transfer, eliminating the need for precise baud rate matching between devices and improving reliability in multi-device communication. The two devices must also have clocks that are close to the same rate and will remain so excessive differences between clock rates on either end will cause garbled data. Asynchronous serial ports require hardware overhead, the UART at either end is relatively complex and difficult to accurately implement in software if necessary.

Another core fault in asynchronous serial ports is that they are inherently suited to communications between two, and only two, devices. While it is possible to connect multiple devices to a single serial port, bus contention (where two devices attempt to drive the same line at the same time) is always an issue and must be dealt with carefully to prevent damage to the devices in question, usually through external hardware. Data rates are also an issue. While there is no theoretical limit to asynchronous serial communications, most UART devices only support a certain set of fixed baud rates, and the highest of these is usually around 230400 bits per second.

For SPI interfaces, the most obvious drawback is the number of pins required. Connecting a single controller to a single peripheral with an SPI bus requires four lines, each additional peripheral

device requires one additional chip select I/O pin on the controller. The rapid proliferation of pin connections makes it undesirable in situations where lots of devices must be connected to one controller. Also, the large number of connections for each device can make routing signals more difficult in tight PCB layout situations.

SPI only allows one controller on the bus, but it does support an arbitrary number of peripherals (subject only to the drive capability of the devices connected to the bus and the number of chips select pins available).

I^2C requires a mere two wires, like asynchronous serial, but those two wires can support up to 1008 peripheral devices. Also, unlike SPI, I^2C can support a multi-controller system, allowing more than one controller to communicate with all peripheral devices on the bus (although the controller devices can't talk to each other over the bus and must take turns using the bus lines).

Data rates fall between asynchronous serial and SPI, most I^2C devices can communicate at 100kHz or 400kHz. There is some overhead with I^2C, for every 8 bits of data to be sent, one extra bit of meta data must be transmitted. The hardware required to implement I^2C is more complex than SPI, but less than asynchronous serial. It can be trivially implemented in software.

2.4.4.1 I^2C at the Hardware Level

Each I^2C bus consists of two signals, SDA and SCL. SDA (Serial Data) is the data signal and SCL (Serial Clock) is the clock signal. The clock signal is always generated by the current bus controller. Some peripheral devices may force the clock low at times to delay the controller sending more data (or to require more time to prepare data before the controller attempts to clock it out). A scenario called "clock stretching"(SparkFun Electronics, n.d.).

Unlike UART or SPI connections, the I^2C bus drivers are open drain, meaning that they can pull the corresponding signal line low, but cannot drive it high. Thus, there can be no bus contention where one device is trying to drive the line high while another tries to pull it low, eliminating the potential for damage to the drivers or excessive power dissipation in the system. Each signal line has a pull-up resistor on it, to restore the signal to high when no device is asserting it low.

As illustrated in Figure 12, both the controller and peripheral use open-drain drivers on the SDA and SCL lines, allowing devices to pull the lines low while external pull-up resistors restore them to a high state when idle. This shared-line configuration enables safe multi-device communication and prevents direct contention between devices driving opposing logic levels.

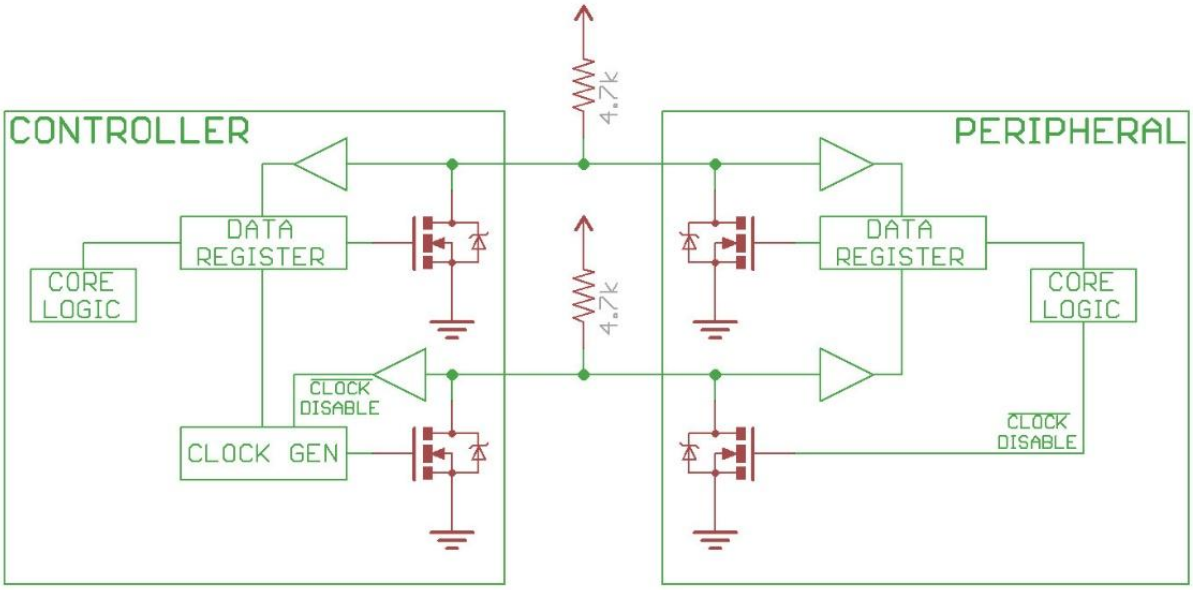

Figure 12. *I²C hardware-level schematic showing open-drain SDA and SCL lines with pull-up resistors enabling shared-bus communication between controller and peripheral devices (SparkFun Electronics, n.d.)*

In practice, however, I²C introduces challenges for redundancy systems. Arbitration, clock stretching, and shared-bus contention can lead to non-deterministic delays, which complicate heartbeat-based failure detection. Studies on I²C-based multi-controller systems note that if a faulty node holds the SDA line low, the entire bus may become unresponsive, effectively freezing communication across all connected devices (Leens, 2009). Such behavior presents a critical limitation for redundancy architectures, where a single failing node must not compromise the entire communication backbone.

2.5 Recent Research in Arduino-to-Arduino Networking

In recent years, Arduino platforms have evolved from simple standalone controllers into components of distributed embedded networks. Researchers have explored Arduino-to-Arduino communication in contexts such as mesh networking, swarm robotics, environmental monitoring, and hybrid microcontroller systems. While these efforts demonstrate feasibility, they also reveal clear limitations in autonomy, fault recovery, and deterministic failover gaps directly relevant to this thesis.

2.5.1 Arduino Networking Libraries and Toolkits

Several open-source libraries have emerged to simplify Arduino networking. Among the most widely referenced is Padded Jittering Operative Network, (PJON). This toolkit abstracts low-level serial communication and provides packet handling, addressing, and optional reliability mechanisms.

PJON (Padded Jittering Operative Network), developed by (Giovanni Blu Mitolo, 2022), is a modular communication stack designed specifically for microcontroller-based systems such as Arduino. It supports multi-device networking across UART, SPI, and software-defined buses while maintaining low memory overhead, making it suitable for resource-constrained platforms. PJON incorporates packet framing, addressing, acknowledgment mechanisms, and CRC-based error detection, enabling more structured peer-to-peer communication than native Arduino serial libraries. However, while PJON facilitates multi-master communication and improves transport-layer reliability, it does not natively implement distributed failover logic or autonomous redundancy control. Role arbitration and fault recovery must therefore be implemented at the application layer.

2.5.2 Arduino in Mesh Networking and Swarm Robotics

Arduino-based mesh and swarm networks have been widely explored in distributed sensing and cooperative robotics. Swarm robotic platforms built on low-cost microcontrollers demonstrate decentralized communication models using peer-to-peer signaling and short-range wireless modules, enabling scalable coordination without centralized control (Brambilla et al., 2013). These systems emphasize collective behavior, local decision-making, and robustness to communication loss. However, most swarm implementations assume that individual nodes remain operational and focus primarily on coordination rather than controller-level redundancy. While packet loss and intermittent connectivity are tolerated, deterministic failover mechanisms where one controller autonomously assumes the functional role of a failed peer are rarely implemented. As a result, swarm networking research provides strong evidence for decentralized communication feasibility, but leaves a clear gap in structured, heartbeat-driven redundancy control for Arduino-class microcontrollers operating in mission-critical environments.

2.5.3 Arduino Hybrid Architectures for Wireless Monitoring

A recent example of hybrid microcontroller networking is presented by (Dhrubo & Qayum, 2025). The study proposes a dual-node architecture using STM32F103C8T6 microcontrollers communicating via HC-05 Bluetooth modules in a master–slave configuration.

The system consists of two physically separated embedded units: a “Green House” controller node responsible for collecting environmental data (temperature, humidity, soil moisture, raindrop detection, and obstacle distance), and a “Red House” peripheral node responsible for receiving, displaying, and reacting to the transmitted data. Communication occurs through asynchronous USART at 9600 baud, with Bluetooth configured in controller-peripheral mode. Notably, Bluetooth configuration required the use of an Arduino board, as STM32CubeIDE did not directly support setting the module into master mode.

Figure 13 illustrates the system architecture, clearly showing the controller-peripheral communication pathway and peripheral integration. The diagram highlights how sensing, processing, and wireless transmission are consolidated within the STM32 nodes, while OLED displays provide local feedback.

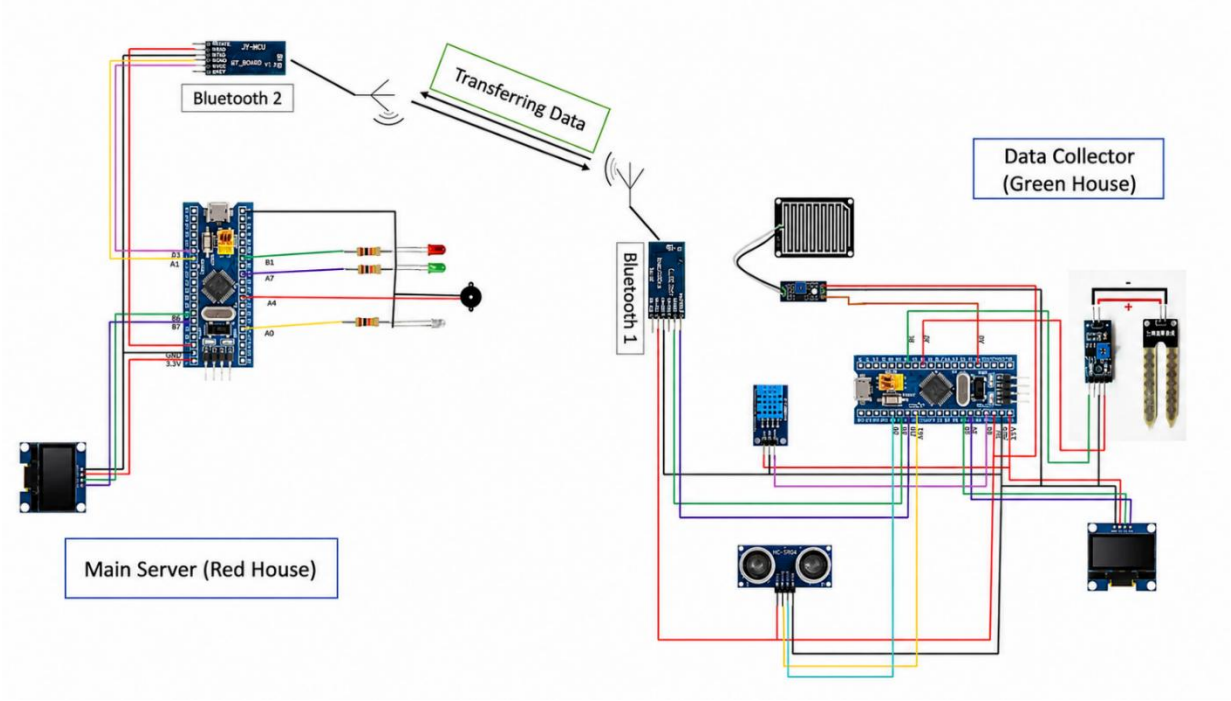

Figure 13. *Architecture of the Proposed Any Area Alert Monitoring System Using the STM32F103C8T6 Microprocessor*

The authors also provide a structured comparison between Arduino (AVR-based) and STM32 (ARM Cortex-M3) architecture Table 1, emphasizing STM32's higher clock speed (72 MHz vs. 16 MHz for Arduino Uno), expanded memory, and enhanced peripheral support. This justifies the use of STM32 in applications requiring improved processing power and real-time responsiveness.

Table 1. *Comparative Analysis of Arduino and STM32 Microcontroller Architectures*

Aspect	**Arduino (ATmega/AVR)**	**STM32 (ARM Cortex M)**
CPU Architecture	8-bit AVR (e.g., ATmega328); some use 32-bit SAM3X (e.g., Arduino Due)	32-bit ARM Cortex-M series (M0–M7), including F0 to H7 families
Clock Speed	16 MHz (Uno); up to 84 MHz (Due)	From 24 MHz (F0) up to 800 MHz
RAM / Flash Memory	Typically, 2KB RAM/32KB Flash (Uno); 96 KB RAM in Due	64 KB–512+ KB Flash; tens to hundreds of KB RAM depending on model
Peripherals and I/O	GPIO, UART, SPI, I2C; limited ADC/DAC	CAN, USB OTG, camera interface, JPEG decoding, cryptography, timers, DMA
Performance	Suitable for simple tasks; limited in complex or multimedia processing	High performance for real-time, multimedia, or complex embedded tasks
Ease of Use / Libraries	Beginner-friendly IDE and rich library ecosystem	STM32CubeIDE with HAL/LL libraries; more complex setup
Coding and Development	Arduino C/C++; easy syntax, limited low level access	Requires good C knowledge; supports RTOS and low-level control
Community and Support	Large hobbyist/educational base; many tutorials	Smaller but active community with strong ST documentation
Use Cases	Ideal for education, DIY, and prototyping	Industrial applications, automation, robotics, and embedded systems
Hardware Cost and Availability	Low cost and widely available	Slightly higher cost. Blue Pill/Nucleo boards are affordable options
Scalability/Future proofing	Limited by 8-bit architecture	Highly scalable with wide range of performance tiers
Code Portability	Portable but constrained by architecture	STM32duino and HAL support flexible and advanced portability

However, despite improved computational capability and wireless synchronization, the architecture remains fundamentally centralized. The system relies on a designated controller node, and communication is limited to one data packet at a time due to Bluetooth constraints. Furthermore, failure of the controller node would prevent further data propagation, as no autonomous role reassignment or redundancy protocol is implemented.

Thus, while hybrid microcontroller networks improve processing power and communication bandwidth, they frequently preserve hierarchical control structures. This partially undermines the

objective of fully autonomous Arduino-to-Arduino redundancy, where each node must independently detect failure and assume operational control without reliance on a fixed master or gateway device.

2.5.4 Identified Gaps

Across the literature, Arduino networking research focuses primarily on:

- Message routing
- Packet reliability
- Multi-node connectivity
- Wireless scalability

However, there remains limited scholarly work addressing:

- Deterministic heartbeat-driven failure detection
- Fully autonomous Arduino-to-Arduino role reassignment
- Latency-bounded failover mechanisms
- Space-constrained or radiation-sensitive environments

Most implementations assume functional nodes and prioritize connectivity over survivability. Few studies design Arduino networks where each node simultaneously acts as controller, observer, and backup without reliance on centralized infrastructure.

2.6 Software-Based Redundancy in Space Systems

Space is an unforgiving operational environment. Electronics deployed beyond Earth's atmosphere are continuously exposed to ionizing radiation, extreme temperature cycling, vacuum conditions, electromagnetic interference, and launch-induced vibration. Unlike terrestrial systems, space platforms cannot rely on physical maintenance or rapid hardware replacement. As a result, reliability must be engineered directly into both hardware and software.

Historically, spacecraft reliability has been achieved primarily through hardware redundancy, especially duplication and majority-voting schemes. However, as missions shift toward CubeSats, nanosatellites, and cost-constrained platforms, mass, power, and volume limitations increasingly restrict the feasibility of hardware-heavy redundancy. This has driven a gradual transition toward software-based fault tolerance, where resilience is achieved through monitoring, reconfiguration, and autonomous recovery rather than physical duplication.

For Arduino-class microcontrollers devices that lack radiation hardening, error correcting memory, and hardware voting circuits this transition is especially relevant.

2.6.1 From Hardware Redundancy to Software Fault Tolerance

Traditional spacecraft commonly employ Triple Modular Redundancy (TMR), where three identical processors execute the same task and a voter selects the majority output. This concept,

formalized by (Lyons & Vanderkulk, 1962), provides strong fault masking but significantly increases system mass and complexity.

As illustrated in Figure 14, Triple Modular Redundancy employs three identical logic circuits that process the same input in parallel, with a majority gate selecting the most common output. This configuration enables the system to tolerate a single fault by masking erroneous outputs from any one module.

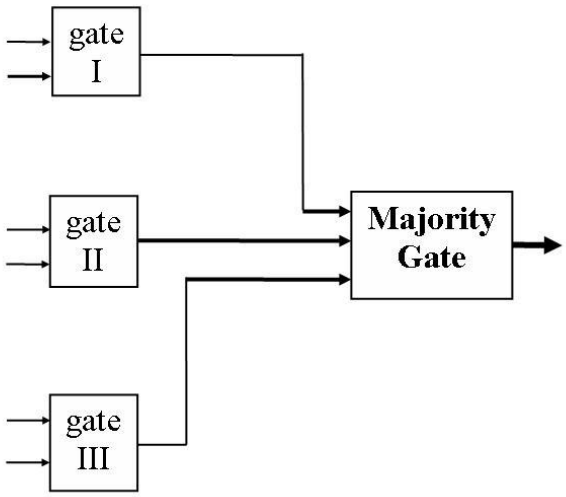

Figure 14. *Triple Modular Redundancy. Three identical logic circuits (logic gates) are used to compute the specified Boolean function. The set of data at the input of the first circuit is identical to the input of the second and third gates. Courtesy Wikipedia.*

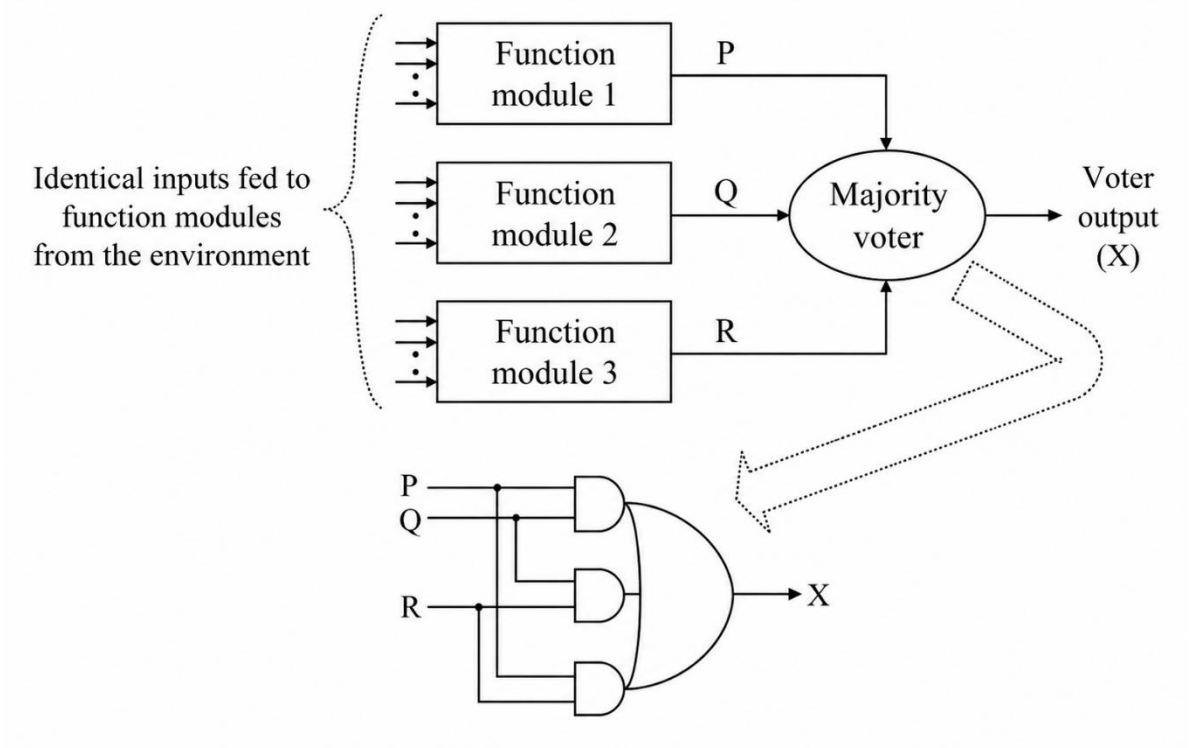

Figure 15. *System-level TMR architecture illustrating three redundant function modules and a majority voter used for fault masking, (Balasubramanian, 2016)*

Figure 15 shows a system-level implementation of TMR consisting of three functionally identical modules receiving the same inputs, with their outputs evaluated by a majority voter. This structure extends the concept from simple logic gates to complete processing units, illustrating how redundancy can be applied at the architectural level for fault tolerance.

While TMR is effective in large spacecraft, it is impractical for lightweight microcontroller platforms. Consequently, research shifted toward Software-Implemented Fault Tolerance (SIFT). Instead of duplicating hardware, SIFT uses watchdog timers, checkpointing, instruction-level redundancy, and error detection mechanisms to preserve correct execution (Reis et al., 2005).

Software-controlled redundancy reduces mass and power overhead while preserving resilience, an approach particularly suitable for distributed microcontroller systems.

In Arduino-to-Arduino redundancy, this philosophy translates into peer monitoring and dynamic role reassignment rather than hardware voting.

2.6.2 Radiation Effects and Microcontroller Vulnerability

One of the most significant threats to embedded processors in space is the Single Event Upset (SEU), a transient bit flip caused by ionizing radiation. (Baumann, 2005) provides a comprehensive analysis of radiation-induced soft errors in modern semiconductor devices.

In microcontrollers, SEUs may:

- Corrupt stack pointers
- Freeze communication peripherals
- Disrupt State Machines

Crucially, many SEUs are transient rather than permanent. If detected early, a reset or reinitialization can restore functionality. This reinforces the importance of heartbeat-based monitoring in distributed systems. A missed heartbeat may indicate a transient upset requiring recovery rather than catastrophic failure.

2.6.3 Watchdog Timers and Autonomous Recovery

Watchdog timers are among the simplest and most effective software-level resilience mechanisms. A watchdog forces a system reset if it fails to receive periodic refresh signals. (Zheng et al., 2021) analyzed such recovery strategies under stochastic models, demonstrating improved system availability.

In distributed systems, watchdog logic can be extended to mutual supervision, where nodes monitor each other rather than themselves.

For Arduino-based redundancy:

- Each node monitors its peer via heartbeat.
- Missed heartbeat triggers reinitialization.
- Recovery occurs autonomously, without ground intervention.

This mirrors software rejuvenation concepts adapted to a minimal microcontroller network.

2.6.4 Software Reconfiguration and Task Migration

Modern space processors increasingly rely on software-controlled reconfiguration. (Gaisler, 2002) introduced the LEON fault-tolerant SPARC architecture, supporting software-managed resilience and task redistribution.

Figure 16 shows how the LEON processor architecture integrates a SPARC V8 core with memory controllers, cache subsystems, and peripheral interfaces interconnected through AMBA buses. The design incorporates fault-tolerant features such as protected register files, error detection and correction (EDAC), and modular peripheral components, enabling both hardware-level reliability and software-managed recovery mechanisms. This modular architecture supports dynamic reconfiguration and task migration by allowing faulty components to be isolated while maintaining system functionality.

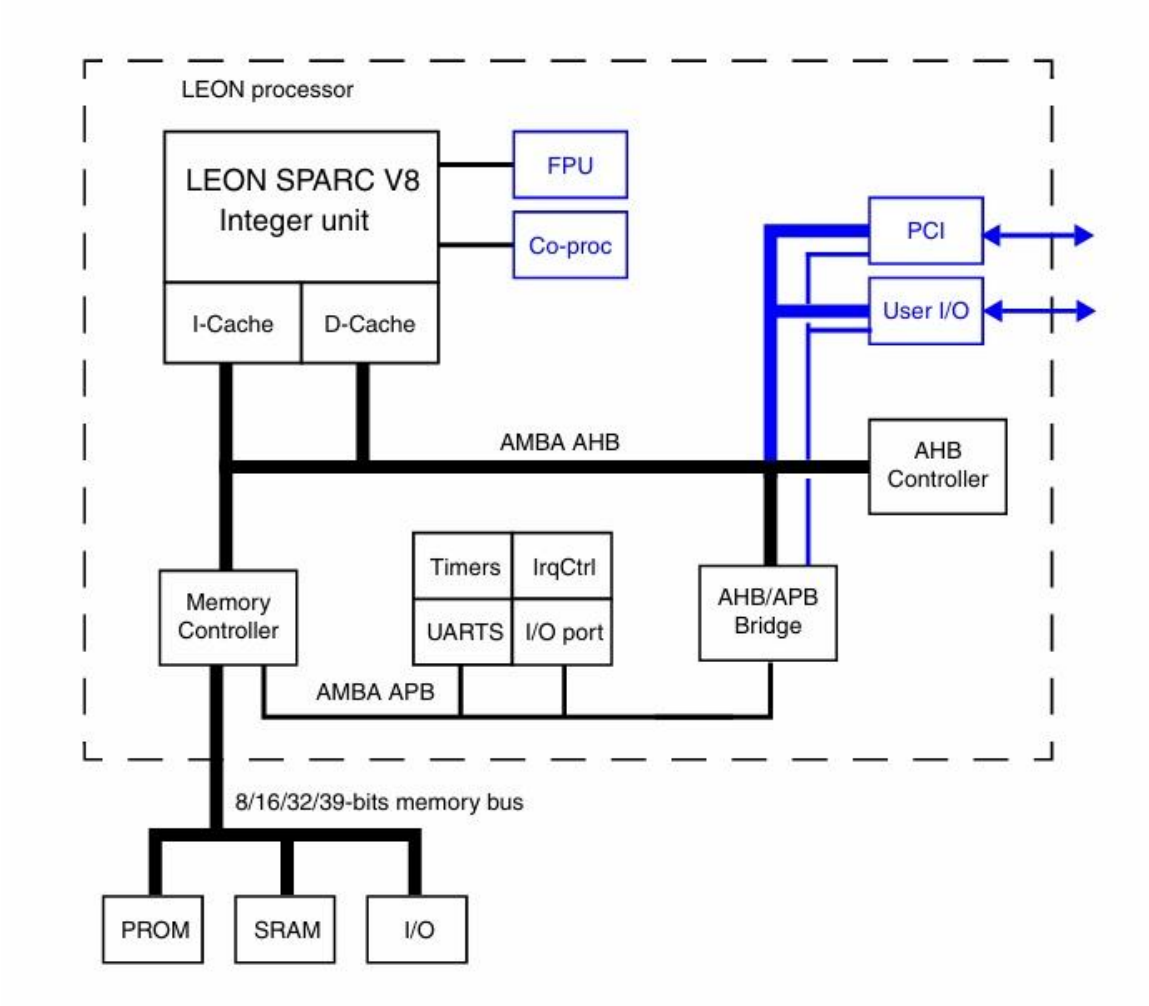

Figure 16. *LEON fault-tolerant processor architecture showing SPARC V8 core, memory and I/O subsystems, and AMBA bus interconnections supporting software-managed reconfiguration and resilience (Gaisler, 2002)*

It was essentially a radiation-tolerant SPARC V8 processor specifically designed for space applications. Unlike purely hardware-redundant approaches, LEON-FT integrates:

- TMR-protected register files
- On-chip EDAC (Error Detection and Correction)
- Pipeline restart mechanisms
- Forced cache miss recovery
- Software-visible fault handling support

The architecture emphasizes software-managed resilience, enabling task redistribution after fault detection, graceful recovery without full system halt and compatibility with standard SPARC toolchains. Heavy-ion radiation testing demonstrated full correction of transient SEUs without observable software-level disruption.

This work marked a shift from static hardware redundancy to configurable, software-cooperative fault tolerance, influencing modern European space processors (e.g., LEON3/LEON4, GR740).

In simplified Arduino systems, task migration reduces to: Node A controls, Node B monitors, Upon failure, node B assumes control. This is software reconfiguration scaled to microcontroller networks.

2.6.5 CubeSats and Software Reliability Gaps

CubeSat reliability studies reveal that many mission failures stem from inadequate fault tolerance rather than hardware limitations. Langer and Bouwmeester (2016) analyzed statistical CubeSat failures and highlighted the need for stronger onboard software resilience (Langer & Bouwmeester, 2016).

These findings underscore a critical point: smaller satellites cannot rely on traditional hardware-heavy redundancy and must instead leverage lightweight distributed software mechanisms.

2.6.6 Existing Limitations

Despite progress, most space-grade software redundancy systems assume:

- Radiation-hardened processors
- Real-Time Operating Systems (RTOS)
- Complex supervisory architectures

Such systems are not directly transferable to Arduino-class devices. They presume greater computational margins and structured communication buses.

A gap therefore remains between:

- High-end distributed space computing architectures
- Ultra-lightweight, peer-to-peer microcontroller redundancy networks.

2.6.7 Relevance to Arduino-to-Arduino Communication

Literature consistently supports three principles:

1. Hardware duplication increases mass and power costs.
2. Software monitoring and recovery can provide meaningful resilience.
3. Distributed peer supervision reduces single-point failure risks.

However, there is limited peer-reviewed exploration of microcontroller-level distributed redundancy built solely on direct inter-node communication without centralized control.

This thesis extends established space fault tolerance theory into the Arduino domain by implementing:

- Heartbeat-based failure detection
- Dynamic role switching
- Autonomous recovery without hardware voting

In doing so, it bridges the gap between space-grade software fault tolerance principles and practical, low-cost, Arduino-based embedded systems designed for harsh environments.

2.7 CubeSats: Evolution, Applications, and the Need for Embedded Redundancy

2.7.1 What Are CubeSats?

CubeSats are a class of nano satellites based on a standardized modular form factor first introduced in 1999 by Jordi Puig-Suari and Bob Twiggs. The fundamental unit, referred to as 1U, measures 10 × 10 × 10 cm and typically weighs less than 1.33 kg. Larger configurations such as 3U, 6U, and 12U are constructed by stacking these units (Puig-Suari et al., 2002). For deployment, CubeSats are loaded into standard dispenser hardware that can interface with a wide range of launch vehicles to fill spare payload space.

CubeSats were originally developed for educational purposes, enabling universities to access space at a fraction of the cost of traditional satellites. The standardized CubeSat form factor reduces the cost of satellite development and deployment, lowering the barrier to entry for educational projects, scientific research, technology demonstrations and commercial applications.

Every year through its CubeSat Launch Initiative (CSLI), NASA selects several projects that align with its research goals and covers all launch costs to deliver the satellites to space. Not all CubeSats launch through NASA's CSLI program, but the development process is similar regardless of mission details (Eric Olson, 2019).

CSLI enables NASA to develop public-private partnerships that provide a low-cost platform for NASA science missions, including planetary exploration, Earth observation, and fundamental Earth and space science. These efforts are a cornerstone in the development of cutting-edge NASA enabling technologies including laser communications, next generation avionics approaches, power generation, distributive sensor systems, satellite-to-satellite communications, and autonomous movement. Leveraging these missions for collaboration optimizes NASA's technology investments, fosters open innovation, and facilitates technology infusion. CubeSat missions are enabling the acceleration of flight-qualified technology assistance in raising Technology Readiness Levels, which aligns to NASA's objective of advancing the Nation's capabilities by maturing cross-cutting innovative space technologies (NASA CubeSat Launch Initiative, 2017).

Figure 17. *A 1U CubeSat was the first satellite developed at Cal Poly to qualify a reliable bus system, sensors and attitude control devices (left), a 3U CubeSat, was also designed at Cal Poly with space weather sensors from NASA's Goddard Space Flight Center (right). Source: [(NASA CubeSat Launch Initiative, 2017)]*

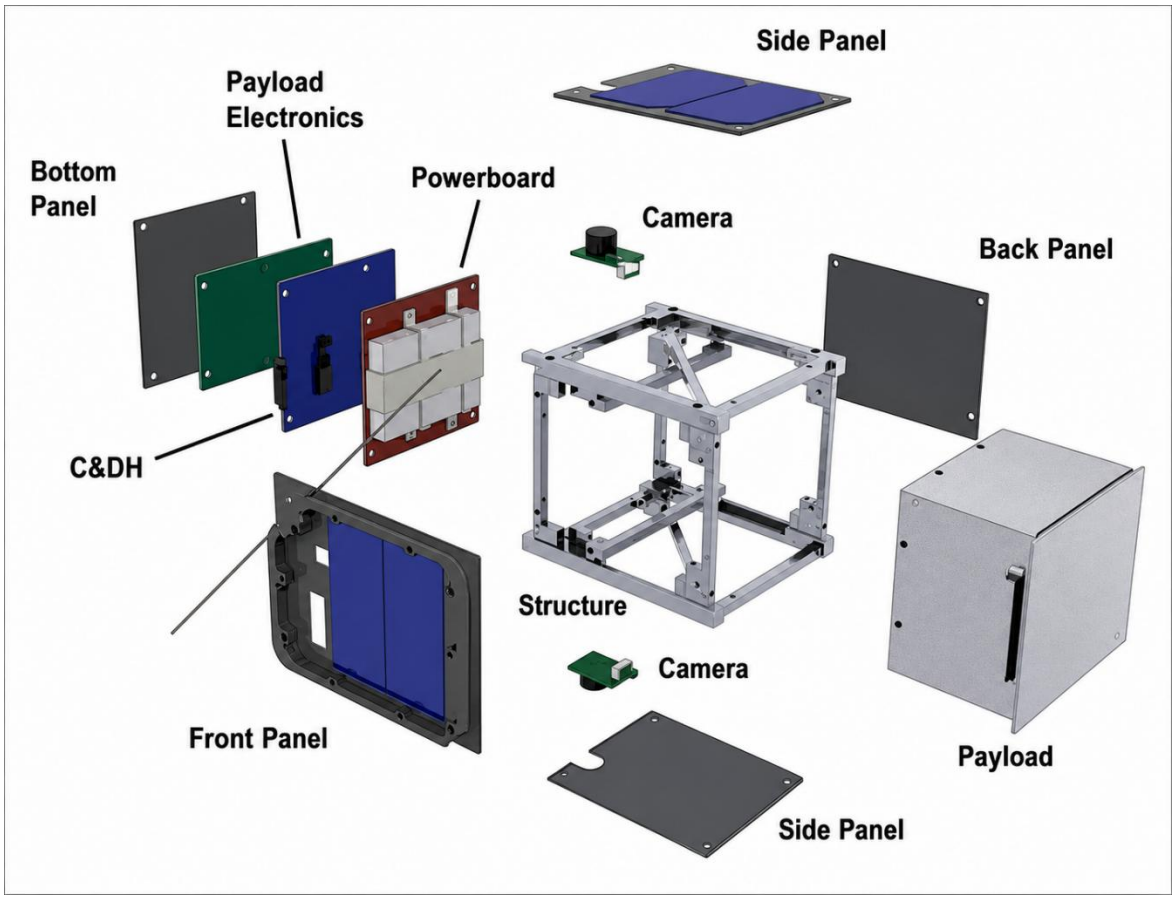

Figure 18. *An exploded view of the basic modules that comprise a typical CubeSat. Source: NASA*

2.7.2 Historical Evolution of CubeSats

The first successful CubeSat launches occurred in 2003. Since then, the number of CubeSats launched annually has grown exponentially. According to (Swartwout, 2013), early CubeSats had limited mission lifetimes and high failure rates, often due to inadequate power systems and insufficient fault tolerance.

By the mid-2010s, improvements in onboard computing, power regulation, and communication systems significantly increased mission reliability. CubeSats transitioned from simple technology demonstrators to platforms supporting Earth observation, interplanetary missions, and distributed sensing networks.

(Langer & Bouwmeester, 2016) conducted a comprehensive statistical reliability analysis of CubeSats, identifying software and electrically related subsystem failures as among the most frequent causes of mission termination. After first assessing overall system reliability, they applied both nonparametric and parametric reliability analyses to subsystem level failure data derived from the CubeSat Failure Database (CFD). To structure their evaluation, failures were categorized into six primary subsystems which are: Electrical Power System (EPS), On-Board Computer (OBC),

Communication System including antennas (COM), Attitude Determination and Control System (ADCS), Payload (PL), and Structure & Deployables (STR) along with an additional "unknown" category for cases in which no specific subsystem could be conclusively identified as the root cause.

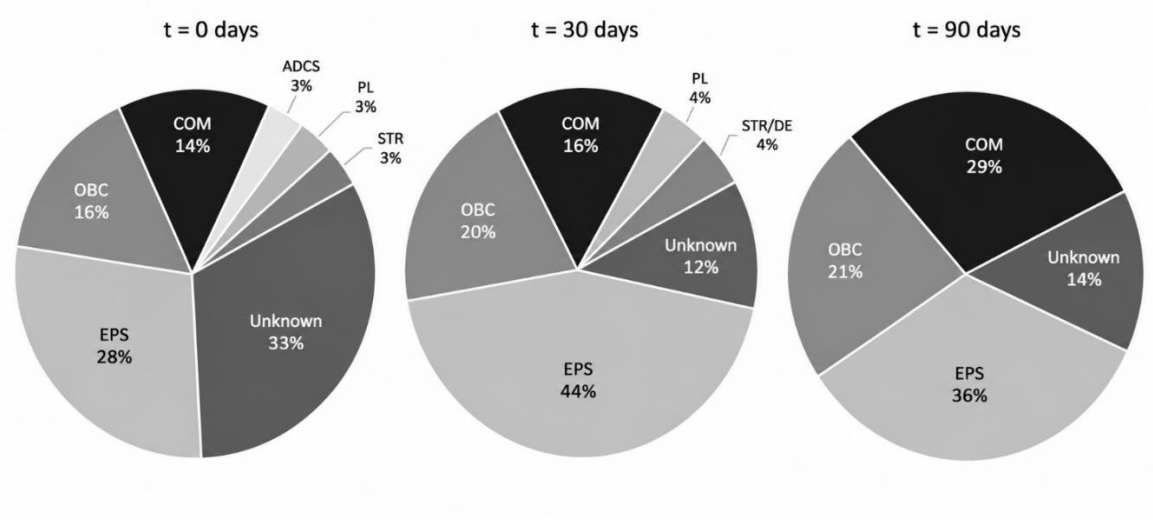

Figure 19 *Subsystem contributions to CubeSat failure after ejection (incl. DOA), 30 days and 90 days*

Their findings revealed distinct temporal and subsystem dependent failure trends. In the early mission phase, the "unknown" category constituted a major source of failure, particularly among dead-on-arrival (DOA) satellites for which communication could not be established. Developer interviews suggested that approximately half of DOA cases fell into this unidentified category, while the remainder often had suspected but unverified causes. As missions progressed, the Electrical Power System emerged as the dominant contributor to failure, accounting for more than 40% of all failures occurring after 30 days in orbit. Beyond 90 days, the communication subsystem became increasingly significant, representing nearly 30% of failures. In contrast, ADCS, payload, and structural subsystems collectively accounted for less than 10% of total mission losses. Overall, the three principal contributors to CubeSat failure are the OBC, EPS, and COM subsystems.

These results underscore that the most failure prone elements are precisely those responsible for power management, onboard processing, and communication, core functions that determine mission survivability. This finding directly supports the need for improved embedded redundancy strategies, particularly in electrical and computing architectures, to enhance CubeSat reliability and reduce early mission loss rates.

2.7.3 Applications in Modern Space Environments

Today, CubeSats are deployed for Earth observation, Climate monitoring, Communication constellations, Space weather monitoring and Deep-space exploration.

A notable milestone was NASA's MarCO CubeSats (Klesh & Krajewski, 2015), which accompanied the InSight Mars mission.

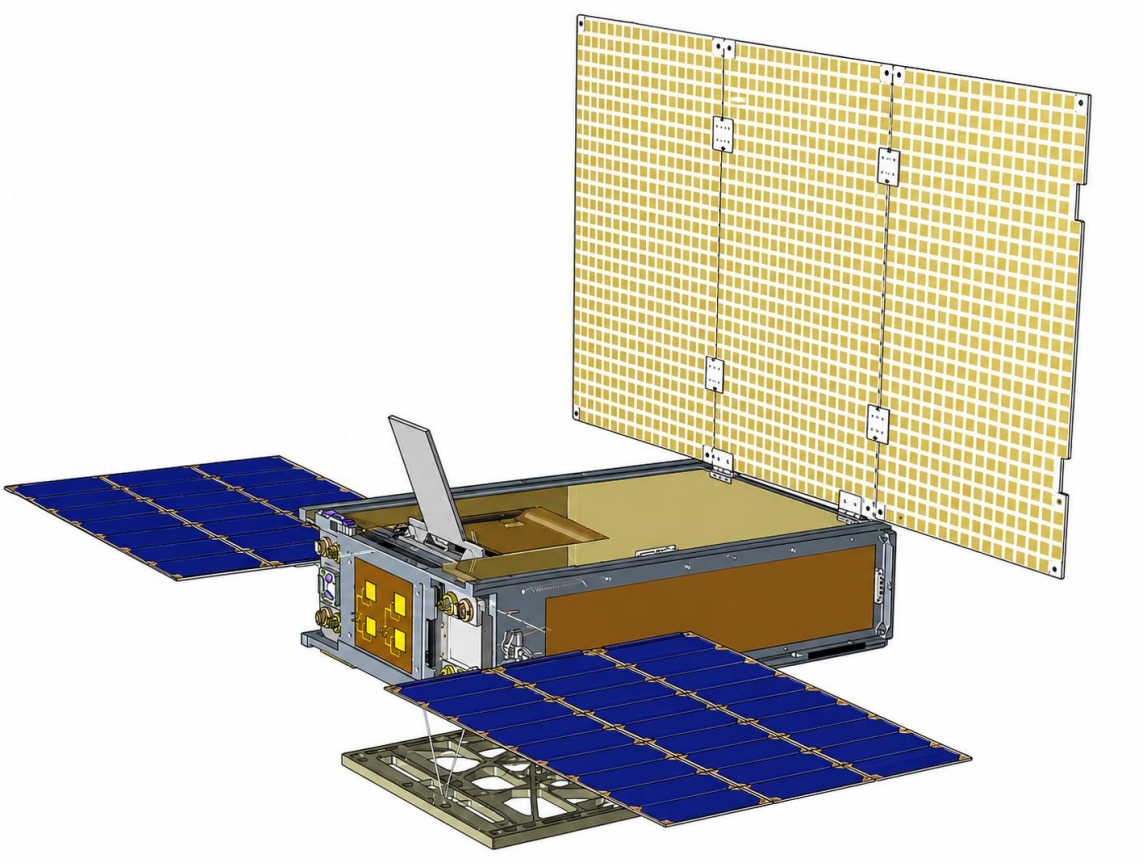

Figure 20. *CAD model rendering of the MarCO CubeSat. The large vertical panel represents the high-gain reflectarray antenna, designed to transmit data at 8 kbps from Mars to the Deep Space Network's 70-m ground station in Madrid, Spain. Courtesy of (Klesh & Krajewski, 2015).*

MarCO demonstrated that CubeSats could survive deep-space radiation environments and operate autonomously millions of kilometers from Earth. However, mission designers emphasized the importance of robust fault detection and recovery logic due to limited real-time ground intervention.

Similarly, distributed CubeSat constellations are being used for IoT connectivity and Earth imaging (Villela et al., 2019). These missions rely heavily on embedded communication systems, often implemented using low-power microcontrollers.

2.7.4 CubeSat Mission Lifetime: How Long Do They Stay in Space?

CubeSat operational lifetime depends primarily on Orbital altitude, Atmospheric drag, Power subsystem durability and electronic reliability

(Swartwout, 2013) conducted a statistical assessment of the first 100 CubeSats launched between 2000 and 2012 and found that operational lifetimes were typically short, averaging less than 200 days. Despite orbital lifetimes that often extended for several years, many missions ceased functioning well before atmospheric decay, highlighting a persistent gap between orbital persistence and functional reliability in early CubeSat missions.

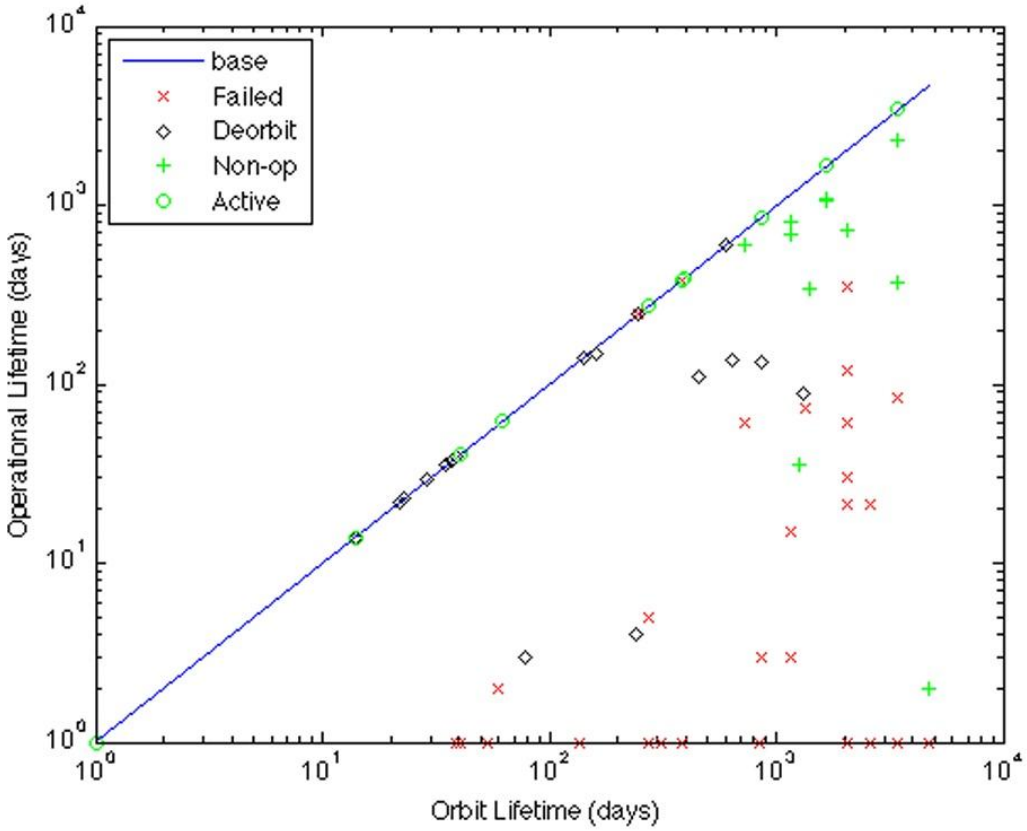

Figure 21. *Operational Lifetime vs. Orbit Lifetime, CubeSats Launched from 2000-2012, (Swartwout, 2013)*

(Langer & Bouwmeester, 2016) showed that CubeSats exhibit a high rate of early failure, with reliability strongly affected by dead-on-arrival cases and infant mortality during the first months of operation. Their subsystem analysis revealed that electrical power systems, onboard computers, and communication subsystems are the dominant contributors to mission failure, indicating that electronic subsystem reliability remains a key limitation in early CubeSat missions.

Spacecraft deployed at altitudes between 400 and 600 km may remain in orbit for several years because atmospheric density decreases rapidly with altitude, reducing drag-induced orbital decay. Orbital lifetime in this regime is strongly dependent on solar activity, atmospheric density modeling, and the spacecraft's ballistic coefficient ((Vallado & McClain, 2013), pp. 508–516). Mission engineering references further note that satellites in the lower portion of this altitude band (approx. 400 km) typically decay within a few years, whereas spacecraft near 500–600 km can persist for many years under moderate solar conditions ((Wertz et al., 2018), Ch. 30, pp. 937–945). For small satellites such as CubeSats, which generally possess relatively high area-to-mass ratios, decay times on the order of approximately 1–2 years at 400 km and 5 or more years near 500–600 km are consistent with standard drag-based lifetime estimates.

The key observation is that electronic reliability, not orbital mechanics, often determines usable mission duration.

2.7.5 Shortcomings in CubeSat Electronic Architectures

While CubeSats have democratized access to space, their electronic architecture remains inherently constrained by mass, volume, and cost limitations. These constraints directly influence how fault tolerance is implemented and more importantly, what cannot be implemented.

A defining characteristic of most CubeSats is their reliance on Commercial Off-The-Shelf (COTS) components. Unlike traditional spacecraft which employ radiation-hardened processors and custom-designed fault-tolerant circuits, CubeSats frequently use low-cost microcontrollers and subsystems originally developed for terrestrial applications. (Langer & Bouwmeester, 2016) show that although this approach dramatically reduces development cost and schedule, it also increases susceptibility to electrical and software failures in orbit.

One of the primary risks in Low Earth Orbit is radiation-induced disturbance. (Baumann, 2005) provides a detailed analysis of radiation-induced soft errors, particularly Single Event Upsets (SEUs), in advanced semiconductor technologies, defining SEUs as transient radiation-induced bit flips in memory cells, registers, or logic circuits that do not permanently damage the device. Such events occur when collected charge exceeds the critical charge of a storage node, causing a temporary state reversal. In COTS-based CubeSat onboard computers (OBCs), these transient faults may propagate into higher-level system behavior, potentially resulting in communication interruptions or processor anomalies.

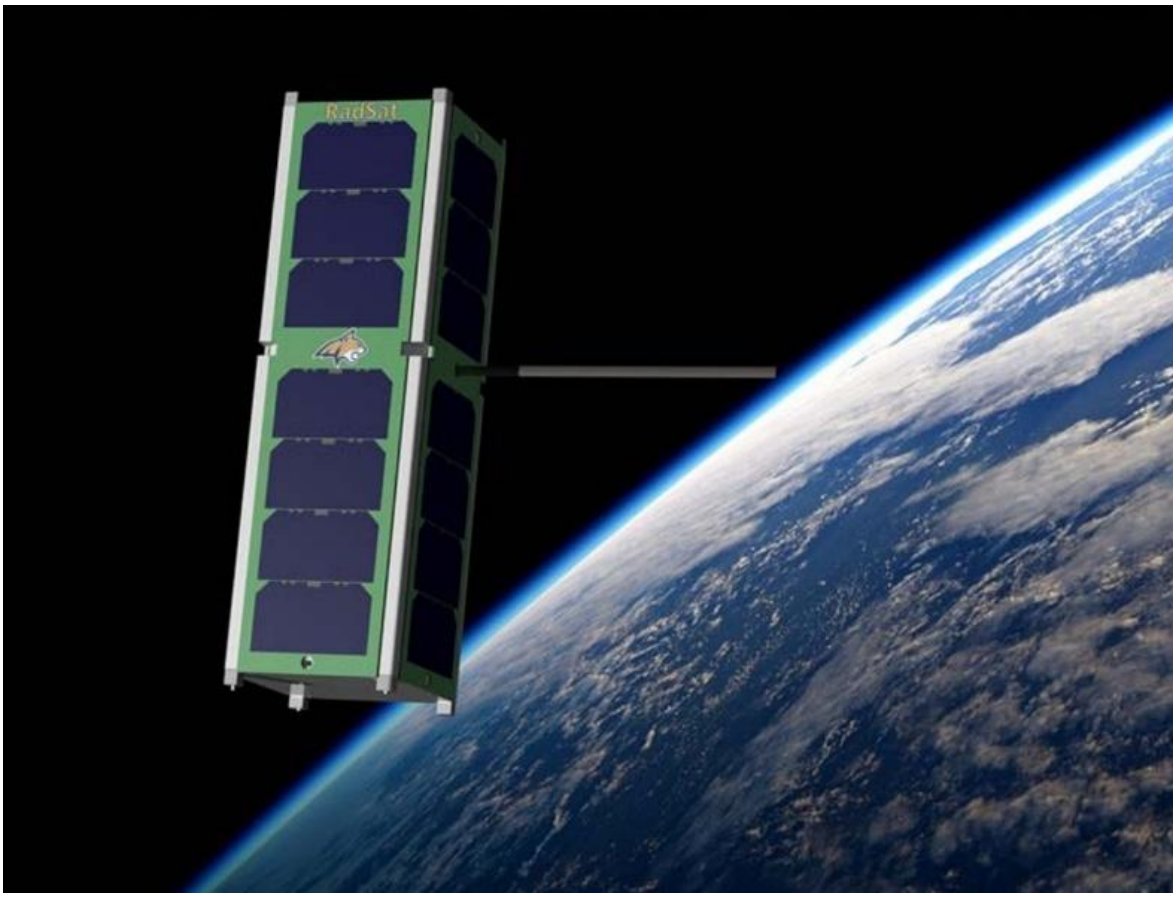

Figure 22. *ELaNa 23 RadSat-G CubeSat from Montana State University in orbit around Earth(NASA CubeSat Launch Initiative, 2017).*

Architecturally, many CubeSats rely on a single onboard computer (OBC) responsible for command handling, data processing, and subsystem coordination. This design minimizes power consumption and board complexity but introduces a critical single-point failure. As (Langer & Bouwmeester, 2016) report in their statistical review of CubeSat missions, a significant proportion of early mission failures are attributed to electrical and software faults rather than structural degradation.

In practice, most CubeSats do not implement full hardware-level redundancy such as Triple Modular Redundancy (TMR), primarily due to power and mass constraints. Instead, resilience is often achieved through watchdog timers, periodic resets, and software recovery routines. While effective against transient faults, watchdog-based recovery cannot mitigate permanent microcontroller failure or catastrophic bus lockups.

Consequently, CubeSat electronic architecture remains vulnerable to both transient and permanent faults. This limitation highlights the need for lightweight, software-driven redundancy

mechanisms capable of maintaining mission continuity without incurring the mass and power penalties associated with traditional hardware duplication.

2.7.6 The Case for Lightweight Electronic Redundancy

Traditional large satellites use hardware voting systems and triple-redundant processors. CubeSats cannot afford this level of duplication due to mass constraints, power budget limits and volume limitations.

Therefore, there is growing interest in distributed microcontroller architectures, where multiple low-power controllers monitor each other. Software-based redundancy enables peer health monitoring, heartbeat exchange, autonomous failover and role reassignment.

This approach aligns directly with the objective of this thesis to design a robust Arduino-to-Arduino redundancy protocol capable of detecting controller failure, transferring operational responsibility and extending mission productivity.

If onboard microcontroller failure can be mitigated through autonomous substitution, CubeSat mission lifetime could approach its full orbital lifetime rather than being prematurely terminated by electronics faults.

This translates into Extended data collection, Increased scientific return, Improved cost efficiency and Greater resilience in deep-space missions.

2.7.7 Research Gap and Thesis Positioning

The literature on CubeSat reliability consistently demonstrates that electronic and software subsystem failures are the dominant causes of mission degradation and premature termination. Statistical analyses of CubeSat missions reveal a recurring pattern. Although many satellites remain physically in orbit for several years, functional lifetimes frequently end within the first 6 to 18 months due to onboard computer or electrical subsystem faults. This discrepancy between orbital survivability and electronic survivability highlights a fundamental architectural limitation in current nanosatellite designs.

Conventional spacecraft mitigate such risks through hardware-intensive redundancy schemes such as triple modular redundancy (TMR), radiation-hardened processors, and voting logic. However, CubeSats operate under strict constraints of mass, power consumption, and volume. As a result, full hardware duplication is rarely feasible. Most CubeSat platforms therefore rely on a single primary onboard computer supported by watchdog timers and reset mechanisms. While these strategies can recover from transient anomalies, they offer limited protection against permanent controller failure.

Research in distributed embedded systems and wireless sensor networks demonstrates that decentralized architectures improve resilience through peer monitoring, heartbeat exchange, and adaptive failure detection. Yet these mechanisms have not been systematically translated into CubeSat-scale microcontroller architectures in a lightweight and deterministic manner. There

remains limited research on Lightweight microcontroller-to-microcontroller redundancy suitable for resource-constrained space platforms, Deterministic failover mechanisms tailored to CubeSat-scale embedded systems and Peer-based heartbeat arbitration in low-memory, low-power environments

Furthermore, while distributed computing increases resilience, it often introduces additional architectural complexity. Current CubeSat implementations typically prioritize communication functionality and payload integration over autonomous controller substitution. As a result, many missions remain dependent on a single onboard processing unit, creating a critical single point of failure.

This thesis addresses this gap by proposing a direct Arduino-to-Arduino redundancy protocol designed specifically for CubeSat-class embedded systems. The proposed approach demonstrates how low-cost, resource-constrained microcontrollers can monitor each other through structured heartbeat exchange, detect controller failure using deterministic timeout models and autonomously reassign operational roles without centralized supervision.

By introducing distributed, software-based redundancy at the microcontroller level, this research aims to narrow the gap between orbital lifetime and functional lifetime in nanosatellite missions. Extending operational continuity beyond the failure of a single controller has the potential to increase mission resilience, improve scientific return, and enhance cost efficiency in low-budget space platforms.

Having established the theoretical foundations and identified the architectural limitations in existing systems, the following chapter presents the methodology used to design, implement, and evaluate the proposed redundancy protocol.

3.0 Methodology

This chapter presents the design, development, and implementation methodology of the proposed redundant Arduino-to-Arduino communication system. Building on the limitations identified in Chapter 2, particularly the reliance on centralized controllers and hardware-intensive redundancy schemes, this work adopts a software-based, peer-to-peer redundancy approach aimed at improving system reliability while minimizing hardware overhead.

The methodology is centered on the design of a direct internode communication architecture, where multiple Arduino boards communicate without intermediary controllers. Unlike traditional controller-periphery or centrally coordinated systems, the proposed framework enables each node to function both as an operational controller and as a monitoring agent. This design directly addresses the research gap identified in the literature, the lack of lightweight, deterministic, and fully distributed redundancy mechanisms for Arduino-class microcontrollers.

At the core of the system is a bidirectional heartbeat monitoring protocol, through which each Arduino periodically transmits and observes status signals from its peers. These heartbeat messages provide a simple yet effective mechanism for real-time system health monitoring. Failure detection is achieved by evaluating the absence of expected heartbeats within a predefined time threshold, allowing nodes to autonomously identify faults without external supervision.

To ensure continuity of operation, the system incorporates a software-controlled failover mechanism based on deterministic role reassignment. When a node is detected as failed or unresponsive, control is automatically transferred to a standby node according to a predefined priority hierarchy. This approach combines distributed fault detection with hierarchical control execution, ensuring that while all nodes participate in monitoring, only one node assumes active control at any given time, thereby preventing conflicts and ensuring safe operation.

The development process followed a structured and incremental approach. Initial validation was performed using a simple two-node configuration to establish reliable communication and basic heartbeat detection. The system was then extended to a multi-node architecture, enabling the evaluation of scalability, coordination, and failover behavior under more complex conditions. This progressive refinement allowed for systematic testing of communication reliability, failure detection accuracy, and recovery latency.

Implementation decisions, including communication interface selection, timing thresholds, and failover logic, were guided by the constraints and insights identified in Chapter 2. In particular, the preference for serial communication, the need for deterministic timing, and the avoidance of shared-bus contention directly influenced architectural design. The resulting system emphasizes simplicity, predictability, and robustness, making it suitable for resource-constrained and fault-prone environments such as space-based embedded systems.

Overall, this methodology demonstrates how software-driven, peer-to-peer coordination can replace traditional hardware redundancy and centralized control, enabling a lightweight yet effective fault-tolerant architecture for Arduino-based networks.

3.1 System Architecture and Design Framework

As established in Chapter 2, existing embedded fault-tolerant architectures are commonly based on centralized supervision, hardware voting, or middleware-supported distributed control, all of which introduce overheads that are poorly suited to Arduino-class and CubeSat-constrained systems. To address this gap, the present work adopts a direct serial, peer-to-peer Arduino redundancy architecture based on bidirectional heartbeat monitoring and software-controlled failover. In this architecture, each Arduino functions as both an operational node and a monitoring node, enabling decentralized fault detection without intermediary controllers.

The implemented design combines distributed monitoring with hierarchical control execution. All nodes continuously exchange heartbeat signals and independently observe the status of their peers. However, to prevent simultaneous control of the payload, only one node is allowed to assume the active control role at any given time. This node operates as the primary controller, while the remaining nodes serve as standby nodes and are ordered by software-defined priority for deterministic failover. In this way, the system preserves the resilience benefits of peer-to-peer monitoring while maintaining safe and conflict-free control transfer during node failure events.

At the core of the architecture is a hierarchical control model in which one Arduino operates as the primary node, actively controlling the LED load circuit, while the remaining nodes function as standby nodes. These standby nodes continuously monitor the operational status of the primary node through heartbeat signals and remain ready to assume control in the event of a failure. This initial design was further enhanced by implementing a fully distributed monitoring approach, where all Arduinos simultaneously observe each other's heartbeat signals. This was achieved by transmitting detectable voltage signals, allowing each node to independently verify the status of others in the network.

To prevent conflicts such as multiple nodes attempting to control the load simultaneously which could potentially damage the microcontrollers, a predefined hierarchy is embedded within the system software. This logical prioritization ensures that only one Arduino assumes control at any given time, even during failover events.

From a hardware perspective, each Arduino is powered by an independent power supply to improve fault isolation and system resilience. However, all ground (GND) pins are interconnected to establish a common electrical reference, which is essential for stable and accurate serial communication. The system also incorporates multiple LEDs for monitoring and debugging purposes. Dedicated indicator LEDs are used to display which Arduino currently controls the

payload, while additional heartbeat LEDs visually represent the transmission and reception of heartbeat signals between nodes.

Failure detection is designed to be both functional and observable. When an Arduino fails regardless of whether it is the active controller its corresponding heartbeat LED ceases to illuminate, providing a clear visual indication of the fault condition.

The entire system was designed, implemented, and tested using Tinkercad Circuits, a simulation platform that enables real-time modeling of Arduino-based systems. Due to limitations in the simulation environment, specifically the inability to directly disconnect power from individual Arduinos, system failures were emulated using switches. These switches effectively interrupted the operation of a node, allowing the failover mechanisms and redundancy features of the system to be thoroughly evaluated under controlled conditions.

3.2 Failover Control System Using Dual Arduino Nodes

This system implements a simple but effective failover mechanism using two Arduino Uno boards, referred to as the Primary Node (Node A) and the Backup Node (Node B). The purpose of the design is to ensure that control of a set of LEDs is maintained even if the primary controller stops functioning. This is achieved through a heartbeat signaling method, where the primary continuously sends a signal to indicate it is operational, and the backup monitors this signal to decide whether to take control.

3.2.1 System Overview

The circuit consists of two Arduino boards connected through:

- A heartbeat communication line (digital pin 12 on both boards)
- A common ground connection to ensure proper signal referencing
- A breadboard with status LEDs and test LEDs

The primary Arduino is responsible for normal operation, including blinking LEDs and sending periodic heartbeat signals. The backup Arduino remains passive while monitoring the heartbeat signal. If the signal stops, the backup assumes that the primary has failed and takes over control of the LEDs.

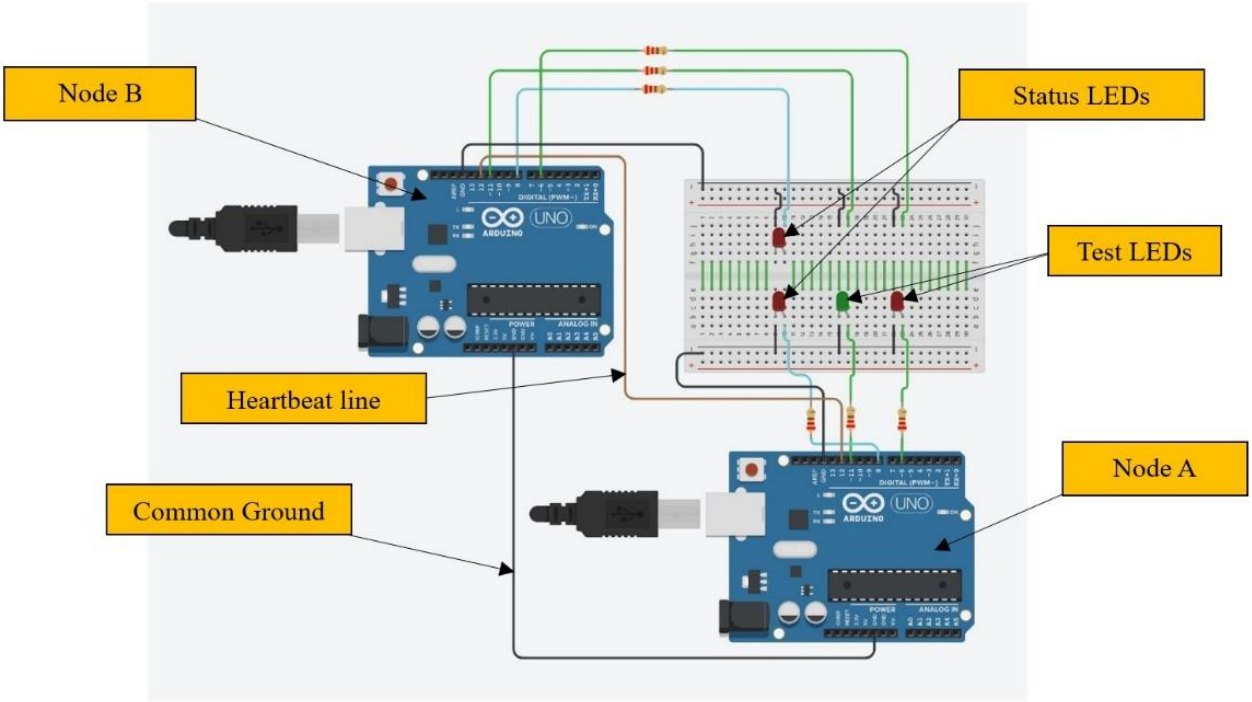

Figure 23. *Arduino redundancy, Uni-directional Setup*

3.2.2 Primary Node Operation (Node A)

The primary Arduino controls two LEDs connected to pins 6 and 11, a status LED on pin 8, and outputs a heartbeat signal on pin 12. At the start of execution, the system records the current time using the `millis()` function. The `millis()` function returns the number of milliseconds since the Arduino was powered on. This value is stored in the variable `startTime` function.

Node A Code:

```
// Primary Arduino
// Sketch to blink 2 LEDs
// Declare and initiallize LED pin variables
byte ledPin_1 = 6;
byte ledPin_2 = 11;
byte heartbeatPin = 12;  // output heartbeat signal
byte statusLed = 8;      // active status indicator
unsigned long startTime;
unsigned long activeDuration = 30000;  //active duration of 30 seconds
bool isActive = true;
void setup() {
  // put your setup code here, to run once:
  pinMode(ledPin_1, OUTPUT);
  pinMode(ledPin_2, OUTPUT);
  pinMode(heartbeatPin, OUTPUT);
```

```
  pinMode(statusLed, OUTPUT);
  startTime = millis();
}
void loop() {
  // put your main code here, to run repeatedly:
  // Turn LEDs on and off
  // Variable to specify the blink rate in milliseconds
  int blinkRateMilliseconds = 500;
  unsigned long currentTime = millis();
  if (isActive) {
    // stay active for a minute
    if (currentTime - startTime < activeDuration) {
      // indicate active status
      digitalWrite(statusLed, HIGH);
      // send heartbeat pulse
      digitalWrite(heartbeatPin, HIGH);
      delay(50);
      digitalWrite(heartbeatPin, LOW);
      // Turn on ledPin_1 and delay to see state
      digitalWrite(ledPin_1, HIGH);
      delay(blinkRateMilliseconds);
      // Turn off ledPin_1 and delay to see state
      digitalWrite(ledPin_1, LOW);
      delay(blinkRateMilliseconds);
      // Turn on ledPin_2 and delay to see state
      digitalWrite(ledPin_2, HIGH);
      delay(blinkRateMilliseconds);
      // Turn off ledPin_2 and delay to see state
      digitalWrite(ledPin_2, LOW);
      delay(blinkRateMilliseconds);
    } else {
      isActive = false;
      digitalWrite(statusLed, LOW);
      digitalWrite(heartbeatPin, LOW);  // stop sending heartbeat
      digitalWrite(ledPin_1, LOW);
      digitalWrite(ledPin_2, LOW);
    }
  }
}
```

The system is designed to remain active for a fixed duration of 30 seconds (`activeDuration = 30000 ms`). During this time:

- The status LED is turned ON, indicating the primary node is active.
- A heartbeat pulse is sent periodically, the heartbeat pin is set HIGH briefly (50 ms), then set LOW. This creates a detectable pulse for the backup node.
- The two LEDs blink alternately with a delay of 500 ms, providing a visible indication of normal operation.

The use of `millis()` allows the system to track how long it has been running without relying on blocking delays. The condition:

```
currentTime - startTime < activeDuration
```

ensures that the system remains active only for the specified duration. This method is preferred over directly comparing `millis()` to a fixed value because it remains reliable even if the timer overflows.
After 30 seconds, the system simulates a failure condition:

- The `isActive` condition is set to false
- The heartbeat signal stops
- All LEDs are turned OFF

This intentional shutdown allows the backup node to detect the absence of the heartbeat and takeover.

3.2.3 Backup Node Operation (Node B)

The backup Arduino monitors the heartbeat signal received on pin 12. It uses the `millis()` function to measure the time between heartbeat signals and determine whether the primary node is still functioning.
Three key variables are used:

- `lastHeartbeat`: stores the time the last heartbeat pulse was detected
- `currentMillis`: stores the current time from `millis()`
- `masterAlive`: indicates whether the primary node is considered operational

When a heartbeat signal is detected (pin reads HIGH), The current time is recorded in `lastHeartbeat`, the system marks that the primary node is alive A flag (`firstHeartbeatDetected`) ensures that monitoring only begins after the first valid signal. The backup continuously evaluates the condition: `currentMillis - lastHeartbeat > 5000.`

This checks whether more than 5 seconds have passed since the last heartbeat. If this condition is true, the system assumes that the primary node has failed.

Node B Code:

```
// backup Arduino
// Sketch to blink 2 LEDs
```

```
// Passive untill heartbeat detected from master
byte ledPin_1 = 6;
byte ledPin_2 = 11;
byte heartbeatPin = 12;  // input from main Arduino
byte statusLed = 8;      // active status indicator
unsigned long lastHeartbeat = 0;
bool masterAlive = false;
bool firstHeartbeatDetected = false;
void setup() {
  // put your setup code here, to run once:
  pinMode(ledPin_1, OUTPUT);
  pinMode(ledPin_2, OUTPUT);
  pinMode(heartbeatPin, INPUT);
  pinMode(statusLed, OUTPUT);
  // start in passive mode
  digitalWrite(ledPin_1, LOW);
  digitalWrite(ledPin_2, LOW);
  digitalWrite(statusLed, LOW);
}
void loop() {
  // put your main code here, to run repeatedly:
  // Turn LEDs on and off
  // Variable to specify the blink rate in milliseconds
  int blinkRateMilliseconds = 500;
  unsigned long currentMillis = millis();
  // check if heartbeat is received
  if (digitalRead(heartbeatPin) == HIGH) {
    lastHeartbeat = currentMillis;
    masterAlive = true;
    firstHeartbeatDetected = true;  // mark that master was detected
  }
  // determine master status
  if (firstHeartbeatDetected) {
    if (currentMillis - lastHeartbeat > 5000) {
      masterAlive = false;  // no heartbeat, master failed
    } else {
      masterAlive = true;
    }
  } else {
    masterAlive = true;  // haven't seen a heartbeat, stay passive
  }
  // behaviour on master status
  if (masterAlive) {
    digitalWrite(ledPin_1, LOW);
    digitalWrite(ledPin_2, LOW);
```

```
    digitalWrite(statusLed, LOW);
  } else {
    // take over
    digitalWrite(statusLed, HIGH);
    //Take control and blink LEDs
    // Turn on ledPin_1 and delay to see state
    digitalWrite(ledPin_1, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_1 and delay to see state
    digitalWrite(ledPin_1, LOW);
    delay(blinkRateMilliseconds);
    // Turn on ledPin_2 and delay to see state
    digitalWrite(ledPin_2, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_2 and delay to see state
    digitalWrite(ledPin_2, LOW);
    delay(blinkRateMilliseconds);
  }
}
```

3.2.4 Failover Behavior

The backup node operates in two modes:

1. **Passive Mode (Normal Operation)**
   - Activated when the primary node is alive
   - All LEDs remain OFF
   - The backup does not interfere with the system
2. **Active Mode (Failover)**
   - Triggered when no heartbeat is detected for more than 5 seconds
   - The backup node takes control of the LEDs
   - The status LED is turned ON to indicate takeover
   - The same blinking pattern used by the primary node is executed

This ensures continuity of operation without manual intervention.

3.2.5 Role of millis() in the System.

The `millis()` function is central to the system's timing and reliability. It enables:

- **Non-blocking time tracking**. The system can measure elapsed time without stopping execution.

- **Failure detection.** The backup node determines system health by measuring the time since the last heartbeat.
- **Controlled operation duration**. The primary node uses `millis()` to limit its active period.

Unlike `delay()`, which pauses program execution, `millis()` allows the system to remain responsive while performing timing checks. This is essential for real-time monitoring in failover systems.

3.2.6 Summary

This dual-Arduino system demonstrates a practical implementation of a fault-tolerant control system using a heartbeat mechanism. The primary node actively controls the system and signals its status, while the backup node passively monitors and intervenes only when necessary. The use of `millis()` enables accurate and reliable timing for both operation control and failure detection, making the system robust and suitable for applications requiring redundancy and high reliability.

3.3 Bidirectional Heartbeat Monitoring with Manual Failure Simulation

In this stage, the system was extended from a single-master configuration to a bidirectional redundancy architecture, in which both Arduino nodes were capable of acting as either the primary or backup controller. This design enabled mutual monitoring between nodes and allowed dynamic role switching based on system conditions. Additionally, push buttons were incorporated to simulate failure scenarios in a controlled manner.

3.3.1 System Architecture

The experimental setup consisted of two Arduino Uno boards connected via a shared breadboard. The layout included bidirectional communication lines for heartbeat signaling, a common ground connection, multiple LEDs for status indication, and push buttons with associated pull-down resistors. Each Arduino was configured identically in both hardware and software, ensuring that either node could assume control of the system when required. The LEDs, connected through 220Ω current-limiting resistors, provided a visual representation of system activity and node status (figure 25). The push buttons, supported by 10kΩ pull-down resistors, enabled manual simulation of node failure by forcing defined logic levels during operation.

3.3.2 Bidirectional Heartbeat Monitoring

In contrast to the previous stage, both nodes transmitted and monitored heartbeat signals simultaneously. Each Arduino periodically generated a digital pulse to indicate that it was operational, while also monitoring the incoming signal from the other node.

The presence of a heartbeat signal indicated that the corresponding node was functioning correctly. Conversely, the absence of a heartbeat for a specified duration was interpreted as a failure. This

approach eliminated the need for a permanently assigned controller node, allowing both devices to operate as equal peers within the system.

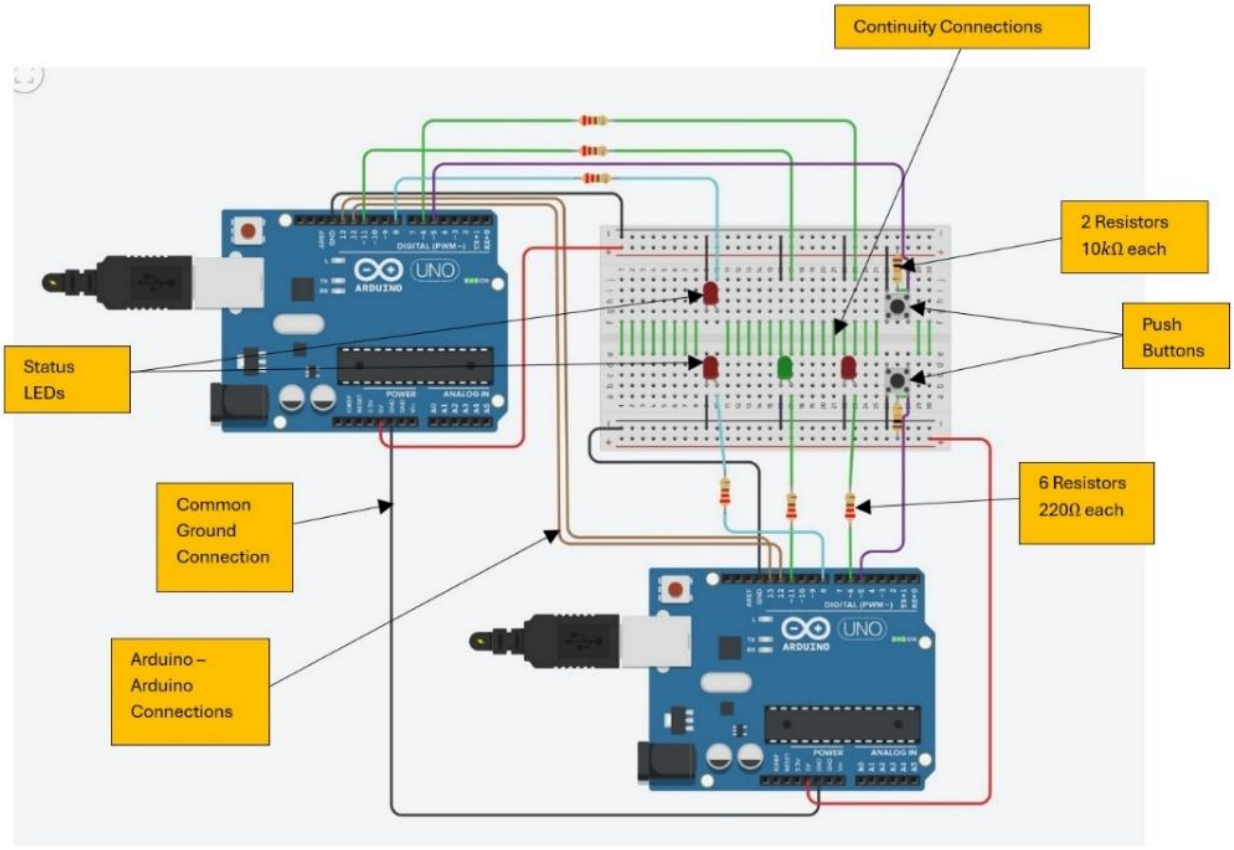

Figure 24. *Bidirectional Heartbeat monitoring layout*

3.3.3 Initial Implementation Error

During initial testing, the system experienced a critical failure due to incorrect wiring of the communication lines. Specifically, digital output pin 13 of one Arduino was directly connected to digital output pin 13 of the other Arduino. In addition, the intended input pin for heartbeat monitoring was incorrectly connected to another input pin rather than to a valid output signal. This configuration resulted in an output-to-output pin conflict, which is electrically unsafe in microcontroller systems.

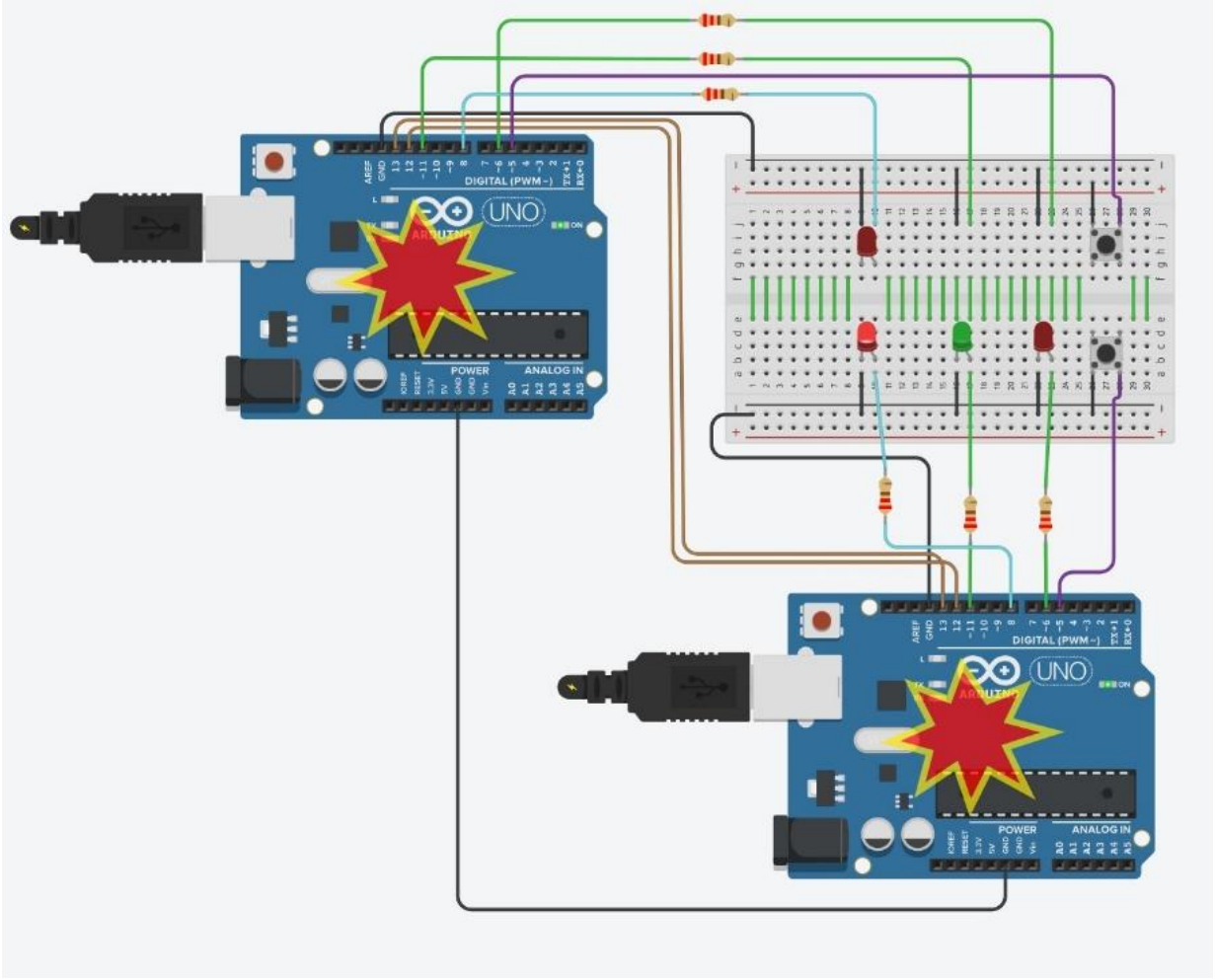

Figure 25. *Microcontroller crash due to conflicting signals*

From a technical perspective, digital output pins actively drive voltage levels. When configured as outputs, Arduino pins source or sink current to maintain either a HIGH (approximately 5 V) or LOW (0 V) state. Directly connecting two output pins creates a condition where both devices attempt to control the same electrical line. If one pin drives a HIGH signal while the other drives a LOW signal, a direct current path is established between the supply voltage and ground, resulting in excessive current flow.

This condition, commonly referred to as bus contention, led to system instability. Observed effects included erratic behavior, microcontroller resets, and eventual system failure. Furthermore, the incorrect input-to-input connection prevented valid signal transmission, rendering the heartbeat monitoring mechanism ineffective.

3.3.4 Corrected Circuit Design

The circuit was subsequently revised to eliminate these issues. The corrected design ensured that:

- Output pins were connected exclusively to input pins, preventing electrical contention.

- Separate communication lines were used for each direction of heartbeat transmission.
- Each heartbeat signal followed a clear path from a transmitting output pin to a receiving input pin.
- All input lines were properly referenced using pull-down resistors where required.

These modifications ensured reliable signal transmission and safe operation of the microcontrollers.

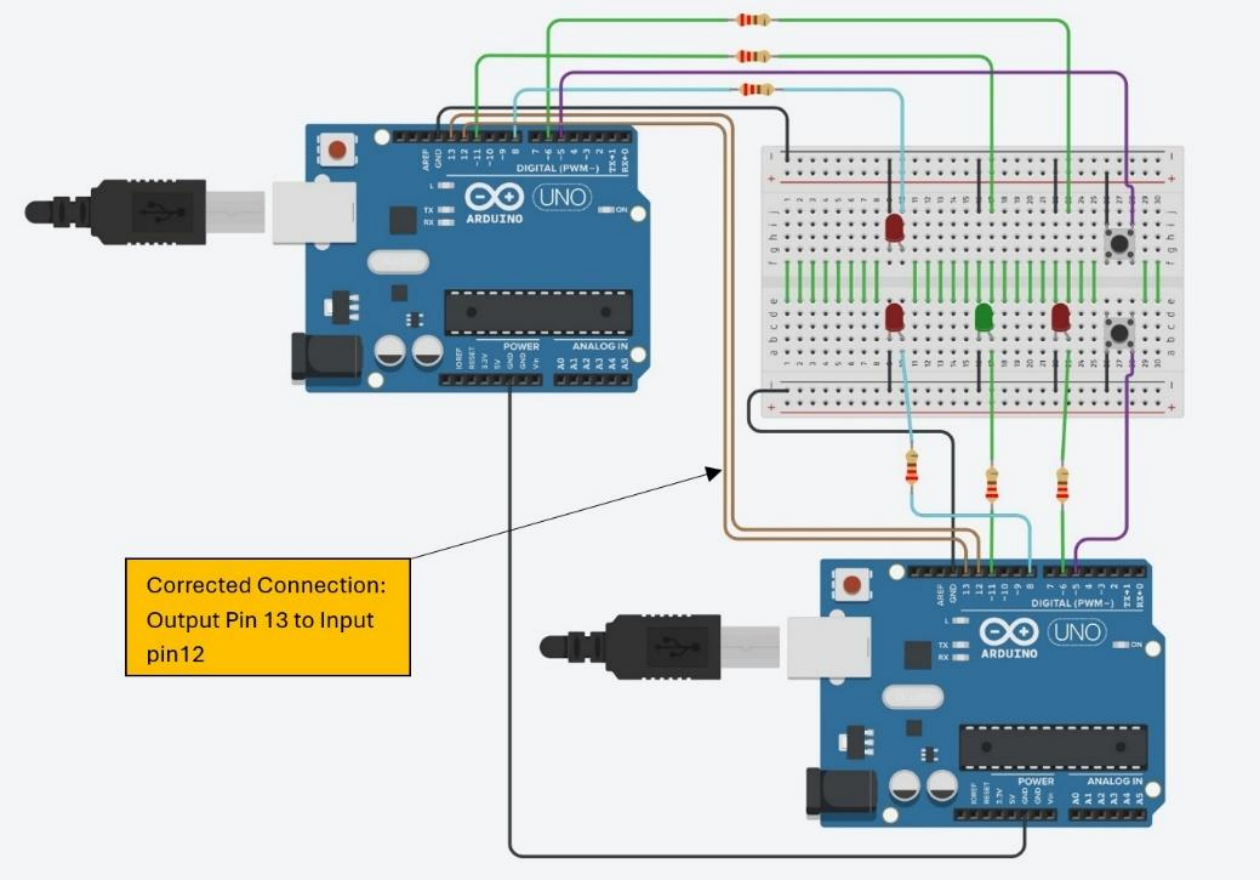

Figure 26. *Proper Connection: Output Pin13 - Input Pin 12*

3.3.5 Failure Simulation and System Response

Push buttons were used to simulate node failure conditions. When a button associated with a node was pressed, that node ceased transmitting its heartbeat signal and its output lines were forced into an inactive state. This effectively emulated a hardware or communication failure.

Under normal operation, both nodes detected valid heartbeat signals and remained in a coordinated state, with only one node actively controlling the LEDs. When a heartbeat signal was no longer detected within the defined timeout period, the monitoring node interpreted this as a failure and transitioned into active mode.

During takeover, the standby node illuminated its status LED and assumed control of the LED outputs, continuing the predefined blinking sequence. This transition occurred automatically without external intervention.

3.3.6 Outcome

Following correction of the wiring configuration, the system operated reliably and demonstrated effective bidirectional redundancy. The experiment confirmed that both nodes were capable of independently detecting failure conditions and assuming control when necessary. This stage demonstrated the successful implementation of a mutually redundant, fault-tolerant system using bidirectional heartbeat monitoring. The initial failure highlighted the importance of correct pin configuration and the risks associated with output-to-output connections in embedded systems. After rectification, the system achieved stable operation, validating the effectiveness of software-driven redundancy combined with proper electrical design.

3.4 Stage 3: Three-Arduino Network Communication

Following the successful implementation of bidirectional redundancy between two nodes, the system was further expanded into a three-node distributed network by introducing a third Arduino (Node C). This stage demonstrated enhanced fault tolerance through multi-node heartbeat communication, where each node monitored the operational status of the others and could autonomously assume control when required.

3.4.1 System Architecture

The experimental setup consisted of three Arduino Uno boards interconnected via a shared breadboard and communication lines. Each node was connected through dedicated heartbeat lines, enabling both transmission and reception of status signals. A common ground was maintained across all nodes to ensure consistent voltage referencing and reliable digital communication.

The circuit included multiple LEDs, each connected through 220 Ω resistors, to visually represent system states such as active control, standby mode, and fault conditions. Push buttons were maintained to simulate node failures.
In the initial design, each push button was paired with an external 10 kΩ pull-down resistor to ensure a defined LOW state when the button was not pressed. However, this configuration was later optimized by replacing the external resistors with the Arduino's internal pull-up functionality.

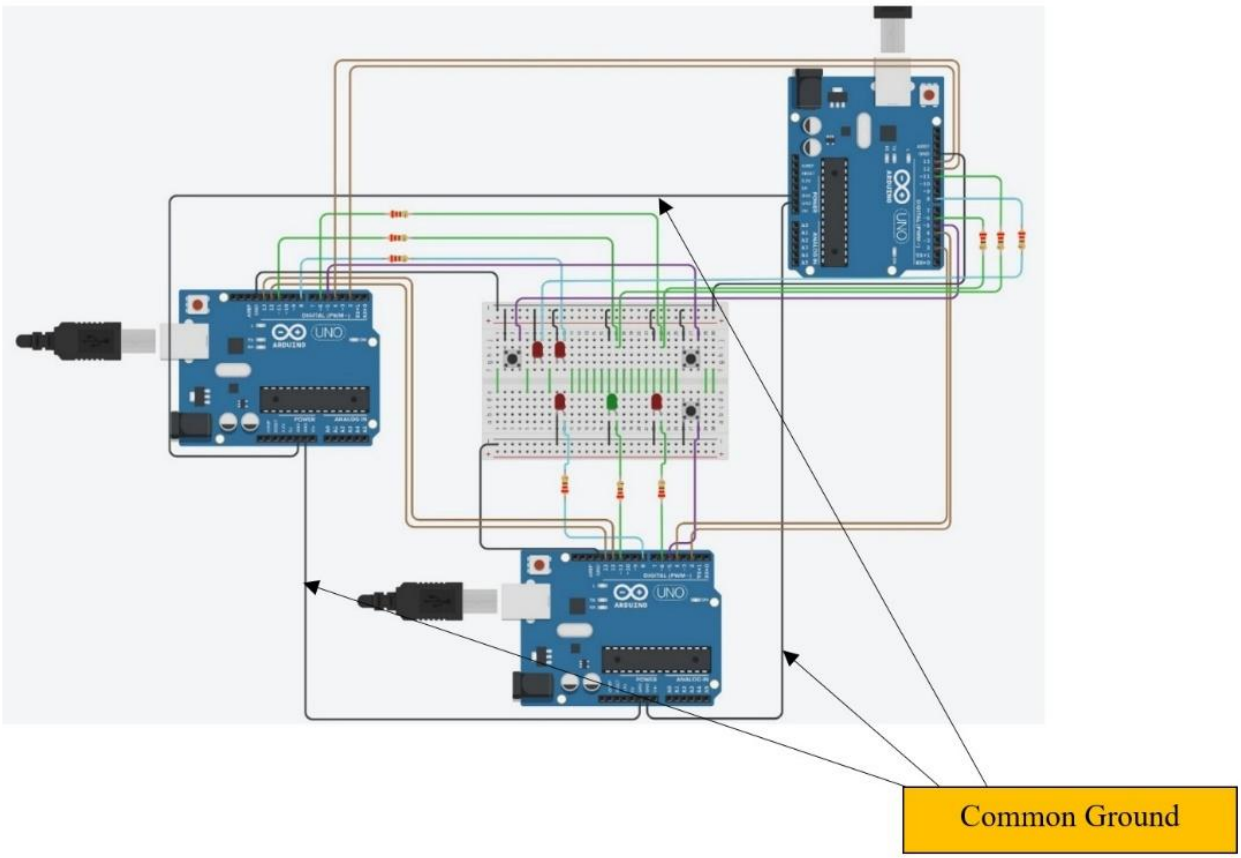

Figure 27. *3-Node configuration with push buttons*

3.4.2 Replacement of External Resistors with Internal Pull-Up Configuration

The external 10 kΩ pull-down resistors were replaced using Arduino's built-in feature:

```
pinMode(buttonPin_C, INPUT_PULLUP);
```

Similarly, for all three nodes (A, B, and C), the push buttons were connected to digital pin 5, for example, i.e. "`const byte buttonPin_C = 5;`"

Using INPUT_PULLUP enables an internal pull-up resistor within the Arduino microcontroller, eliminating the need for external resistors. This configuration ensures that the input pin is held at a stable HIGH logic level (5 V) when the button is not pressed. Pressing the button connects the pin directly to ground, producing a LOW signal. This results in an active-low configuration, where if the Push Button is not pressed maintains HIGH and when the Button is pressed creates a LOW

The advantages of this approach include Reduced component count, simplifying the circuit design, Improved reliability, as fewer external connections reduce the risk of wiring errors and a cleaner layout, especially beneficial in multi-node systems with increased wiring complexity

3.4.3 Multi-Node Heartbeat Communication

In this stage, each Arduino node periodically transmitted a heartbeat signal to the other two nodes while simultaneously monitoring incoming signals. This resulted in a fully interconnected monitoring network, where every node maintained continuous awareness of the operational status of the others.

Within this configuration, each node independently evaluated the presence or absence of heartbeat signals. The absence of a heartbeat from any node was interpreted as a potential failure. To achieve this, timing logic based on the `millis()` function was used to measure the elapsed time since the last received heartbeat. This approach allowed for accurate and non-blocking detection of communication loss.

By distributing monitoring responsibility across all nodes, the system eliminated reliance on a single controller. Instead, system awareness was shared, improving robustness and reducing the likelihood of total system failure.

3.4.4 Control Logic and Node Coordination

Although all three nodes could perform the same functions, only one Arduino actively controlled the LED circuit at any given time. The remaining nodes operated in a standby state while continuously monitoring system conditions.

3.4.4.1 Normal Operation

During normal operation, all nodes transmitted and received heartbeat signals. Each node confirmed that the others were functioning correctly. Based on the implemented logic, one node assumed the role of the active controller and drove the LED outputs. The other nodes remained passive but continued to monitor incoming heartbeat signals in real time.

3.4.4.2 Failure Detection

If a node stopped transmitting its heartbeat signal, the remaining nodes detected its absence after a predefined timeout period. This detection was achieved using time comparisons derived from the `millis()` function, allowing the system to monitor failures without interrupting ongoing processes.

3.4.4.3 Failover Mechanism

When the active node failed, one of the standby nodes automatically assumed control of the system. The newly active node began driving the LED outputs and maintained normal operation. This transition occurred seamlessly and did not require any manual reset or external intervention.

3.4.5 Failure Simulation Using Push Buttons

Push buttons were incorporated to simulate node failures in a controlled and repeatable manner. Each button was connected to a specific node (for example, digital pin 5) and configured using the `INPUT_PULLUP` mode. When a button was pressed, the corresponding node's input pin was pulled to a LOW state, which was interpreted in software as a failure condition. As a result, the node

stopped transmitting its heartbeat signal and appeared inactive to the rest of the network. The remaining nodes detected the absence of the heartbeat and initiated the failover process.

The use of `INPUT_PULLUP` created an active-low configuration, where the default state of the input was HIGH and transitioned to LOW only when the button was pressed. This eliminated the need for external pull-down resistors and simplified the circuit design.

3.4.6 Electrical Connections

Several key electrical design choices were implemented to ensure reliable system operation:

- 220 Ω resistors were used with each LED to limit current and protect both the LEDs and the Arduino output pins.
- The use of internal pull-up resistors replaced external 10kΩ resistors, reducing component count and simplifying wiring.
- A common ground connection was maintained across all nodes to ensure consistent voltage reference and accurate signal interpretation.
- Communication lines were carefully configured so that output pins were connected only to input pins, preventing electrical conflicts such as bus contention.

3.4.7 System Outcome

The three-node configuration successfully demonstrated a distributed and fault-tolerant system. The network could monitor multiple nodes simultaneously, detecting failures through the absence of heartbeat signals, and automatically reassigning control to a functioning node.

The use of push buttons confirmed that the system could respond dynamically to simulated failures. Furthermore, replacing external resistors with internal pull-up configurations improved both the simplicity and reliability of the circuit without affecting performance.

Push button layout code for Arduino A

```
// Primary Arduino A
// Sketch to blink 3 LEDs, show status of Arduino A, simulate failure, monitor
heartbeat of
// backup Arduino B
// Declare and initiallize LED pin variables
byte ledPin_1 = 6;
byte ledPin_2 = 11;
const byte heartbeatOut_B = 12;  // send heartbeat to Arduino B
const byte heartbeatIn_B = 13;   // receive heartbeat from Arduino B
const byte statusLed_A = 8;      // active status indicator
const byte buttonPin_A = 5;      // simulate failure
const byte heartbeatOut_C = 2;   // send heartbeat to Arduino C
const byte heartbeatIn_C = 4;    // receive heartbeat from Arduino C
```

```
unsigned long lastHeartbeat_B = 0;
unsigned long lastHeartbeat_C = 0;  // Arduino C
bool partnerAlive_B = false;
bool partnerAlive_C = false;
bool isActive_A = true;  // Arduino A starts as active
void setup() {
  // put your setup code here, to run once:
  pinMode(ledPin_1, OUTPUT);
  pinMode(ledPin_2, OUTPUT);
  pinMode(heartbeatOut_B, OUTPUT);
  pinMode(heartbeatIn_B, INPUT);
  pinMode(heartbeatOut_C, OUTPUT);  // Sets pin2 as an output to Arduino C
  pinMode(heartbeatIn_C, INPUT);    // Sets pin4 as an input for signal from
Arduino C
  pinMode(statusLed_A, OUTPUT);
  pinMode(buttonPin_A, INPUT_PULLUP);
}
void loop() {
  // Turn LEDs on and off
  // Variable to specify the blink rate in milliseconds
  int blinkRateMilliseconds = 300;
  bool buttonPressed = digitalRead(buttonPin_A) == LOW;  // simulate failure,
when pressed = low
  // send heartbeat pulse and simulate failure
  if (buttonPressed) {
    digitalWrite(heartbeatOut_B, LOW);  // stop heartbeat to B and simulate
failure
    digitalWrite(heartbeatOut_C, LOW);  // stop heartbeat to C and simulate
failure
    digitalWrite(statusLed_A, LOW);
    digitalWrite(ledPin_1, LOW);
    digitalWrite(ledPin_2, LOW);
    delay(10);
    return;  //skip rest of loop
  } else {
    digitalWrite(heartbeatOut_B, HIGH);
    digitalWrite(heartbeatOut_C, HIGH);
  }
  // Monitor Partner B Hearbeat
  if (digitalRead(heartbeatIn_B) == HIGH) {
    lastHeartbeat_B = millis();
    partnerAlive_B = true;
  }
  if (millis() - lastHeartbeat_B > 3000) {
    partnerAlive_B = false;
```

```
}
// Monitor Partner C Hearbeat
if (digitalRead(heartbeatIn_C) == HIGH) {
  lastHeartbeat_C = millis();
  partnerAlive_C = true;
}
if (millis() - lastHeartbeat_C > 3000) {
  partnerAlive_C = false;
}
//determine role
if (!partnerAlive_B) {
  // partner dead or off, take control
  isActive_A = true;
  // indicate active status
  digitalWrite(statusLed_A, HIGH);
} else {
  // partner alive, only one should run
  isActive_A = true;  // primary always continues unless its button is pressed
  digitalWrite(statusLed_A, HIGH);
}
if (!partnerAlive_C) {
  // partner dead or off, take control
  isActive_A = true;
  // indicate active status
  digitalWrite(statusLed_A, HIGH);
} else {
  // partner alive, only one should run
  isActive_A = true;  // primary always continues unless its button is pressed
  digitalWrite(statusLed_A, HIGH);
}
// Active blinking
if (isActive_A) {
  // Turn on ledPin_1 and delay to see state
  digitalWrite(ledPin_1, HIGH);
  delay(blinkRateMilliseconds);
  // Turn off ledPin_1 and delay to see state
  digitalWrite(ledPin_1, LOW);
  delay(blinkRateMilliseconds);
  // Turn on ledPin_2 and delay to see state
  digitalWrite(ledPin_2, HIGH);
  delay(blinkRateMilliseconds);
  // Turn off ledPin_2 and delay to see state
  digitalWrite(ledPin_2, LOW);
  delay(blinkRateMilliseconds);
}
```

```
}
```

Push button layout code for Arduino B

```
// Backup Arduino B
// Sketch to blink 3 LEDs, show status of Arduino B, Monitor heartbeat of Arduino
A and
// Simulate failure
// Declare and initiallize LED pin variables
byte ledPin_1 = 6;
byte ledPin_2 = 11;
const byte heartbeatOut_A = 12;  // send heartbeat to Arduino A
const byte heartbeatIn_A = 13;   // receive heartbeat from Arduino A
const byte statusLed_B = 8;      // active status indicator
const byte buttonPin_B = 5;      // simulate failure
const byte heartbeatOut_C = 2;   // send heartbeat to Arduino C
const byte heartbeatIn_C = 4;    // receive heartbeat from Arduino C
unsigned long lastHeartbeat_A = 0;
unsigned long lastHeartbeat_C = 0;
bool partnerAlive_A = false;
bool partnerAlive_C = false;
bool isActive_B = false;                // Arduino B starts in passive mode
bool firstHeartbeatDetected_A = false;  // Assume no heartbeat from Arduino A if
no heartbeat is received yet
bool firstHeartbeatDetected_C = false;  // Assume no heartbeat from Arduino C if
no heartbeat is received yet
void setup() {
  pinMode(ledPin_1, OUTPUT);
  pinMode(ledPin_2, OUTPUT);
  pinMode(heartbeatOut_A, OUTPUT);
  pinMode(heartbeatIn_A, INPUT);
  pinMode(heartbeatOut_C, OUTPUT);  // Sets pin2 as an output to Arduino C
  pinMode(heartbeatIn_C, INPUT);    // Sets pin4 as an input for signal from
Arduino C
  pinMode(statusLed_B, OUTPUT);
  pinMode(buttonPin_B, INPUT_PULLUP);
  // start in passive mode
  digitalWrite(ledPin_1, LOW);
  digitalWrite(ledPin_2, LOW);
  digitalWrite(statusLed_B, LOW);
}
```

```
void loop() {
  // Variable to specify the blink rate in milliseconds
  int blinkRateMilliseconds = 300;
  unsigned long currentMillis = millis();
  bool buttonPressed = digitalRead(buttonPin_B) == LOW;  // simulate failure,
button pressed = low
  // send heartbeat always unless button is pressed(failure simulated)
  if (buttonPressed) {
    digitalWrite(heartbeatOut_A, LOW);  // stop heartbeat to Arduino A and
simulate failure
    digitalWrite(heartbeatOut_C, LOW);  // stop heartbeat to Arduino C and
simulate failure
    digitalWrite(statusLed_B, LOW);
    digitalWrite(ledPin_1, LOW);
    digitalWrite(ledPin_2, LOW);
    delay(10);
    return;  //skip rest of loop
  } else {
    digitalWrite(heartbeatOut_A, HIGH);  // send heartbeat to Arduino A if button
not pressed
    digitalWrite(heartbeatOut_C, HIGH);  // send heartbeat to Arduino C if button
not pressed
  }
  // Monitor Partner A Hearbeat
  if (digitalRead(heartbeatIn_A) == HIGH) {
    lastHeartbeat_A = currentMillis;
    partnerAlive_A = true;
    firstHeartbeatDetected_A = true;
  }
  // Monitor Partner C Hearbeat
  if (digitalRead(heartbeatIn_C) == HIGH) {
    lastHeartbeat_C = currentMillis;
    partnerAlive_C = true;
    firstHeartbeatDetected_C = true;
  }
  // determine partner A status
  if (firstHeartbeatDetected_A) {
    // Already saw a heartbeat, check if stopped
    if (currentMillis - lastHeartbeat_A > 3000) {
      partnerAlive_A = false;  // no recent heartbeat detected, partner failed
    } else {
      partnerAlive_A = true;  // heartbeat present
    }
  } else {
    // never saw a hearbeat, wait a short time after start up
```

```
    if (currentMillis > 2000) {
      partnerAlive_A = false;  // before 1st detection, assume partner alive,
stay passive
    } else {
      partnerAlive_A = true;  // still within starting window
    }
  }
  // determine partner C status
  if (firstHeartbeatDetected_C) {
    // Already saw a heartbeat, check if stopped
    if (currentMillis - lastHeartbeat_C > 3000) {
      partnerAlive_C = false;  // no recent heartbeat detected, partner failed
    } else {
      partnerAlive_C = true;  // heartbeat present
    }
  } else {
    // never saw a hearbeat, wait a short time after start up
    if (currentMillis > 2000) {
      partnerAlive_C = false;  // before 1st detection, assume partner alive,
stay passive
    } else {
      partnerAlive_C = true;  // still within starting window
    }
  }
  //determine role
  if (!partnerAlive_A) {
    // partner A dead, take control
    isActive_B = true;
    digitalWrite(statusLed_B, HIGH);  // indicate active status
  } else {
    // partner alive, standby
    isActive_B = false;  // primary always continues unless its button is pressed
    digitalWrite(statusLed_B, LOW);
  }
  if (!partnerAlive_C) {
    // partner A dead, take control
    isActive_B = true;
    digitalWrite(statusLed_B, HIGH);  // indicate active status
  } else {
    // partner alive, standby
    isActive_B = false;  // primary always continues unless its button is pressed
    digitalWrite(statusLed_B, LOW);
  }
  // behaviour based on role
  if (isActive_B) {
```

```
    // Turn on ledPin_1 and delay to see state
    digitalWrite(ledPin_1, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_1 and delay to see state
    digitalWrite(ledPin_1, LOW);
    delay(blinkRateMilliseconds);
    // Turn on ledPin_2 and delay to see state
    digitalWrite(ledPin_2, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_2 and delay to see state
    digitalWrite(ledPin_2, LOW);
    delay(blinkRateMilliseconds);
  } else {
    // stay passive
    digitalWrite(ledPin_1, LOW);
    digitalWrite(ledPin_2, LOW);
    digitalWrite(statusLed_B, LOW);
  }
}
```

Push button layout code for Arduino C

```
// Backup Arduino C
// Sketch to blink 3 LEDs, show status of Arduino B, Monitor heartbeat of Arduino
A and
// Simulate failure
// Declare and initiallize LED pin variables
byte ledPin_1 = 6;
byte ledPin_2 = 11;
const byte heartbeatOut_B = 12;  // send heartbeat to Arduino B
const byte heartbeatIn_B = 13;   // receive heartbeat from Arduino B
const byte statusLed_C = 8;      // active status indicator
const byte buttonPin_C = 5;      // simulate failure
const byte heartbeatOut_A = 2;   // send heartbeat to Arduino A
const byte heartbeatIn_A = 4;    // receive heartbeat from Arduino A
unsigned long lastHeartbeat_A = 0;
unsigned long lastHeartbeat_B = 0;
bool partnerAlive_A = false;
bool partnerAlive_B = false;
bool isActive_C = false;                // Arduino C starts in passive mode
bool firstHeartbeatDetected_A = false;  // Assume no heartbeat from Arduino A if
no heartbeat is received yet
bool firstHeartbeatDetected_B = false;  // Assume no heartbeat from Arduino B if
no heartbeat is received yet
```

```
void setup() {
  pinMode(ledPin_1, OUTPUT);
  pinMode(ledPin_2, OUTPUT);
  pinMode(heartbeatOut_B, OUTPUT);
  pinMode(heartbeatIn_B, INPUT);
  pinMode(heartbeatOut_A, OUTPUT);  // Sets pin2 as an output to Arduino C
  pinMode(heartbeatIn_A, INPUT);    // Sets pin4 as an input for signal from
Arduino C
  pinMode(statusLed_C, OUTPUT);
  pinMode(buttonPin_C, INPUT_PULLUP);
  // start in passive mode
  digitalWrite(ledPin_1, LOW);
  digitalWrite(ledPin_2, LOW);
  digitalWrite(statusLed_C, LOW);
}
void loop() {
  // Variable to specify the blink rate in milliseconds
  int blinkRateMilliseconds = 300;
  unsigned long currentMillis = millis();
  bool buttonPressed = digitalRead(buttonPin_C) == LOW;  // simulate failure,
button pressed = low
  // send heartbeat always unless button is pressed(failure simulated)
  if (buttonPressed) {
    digitalWrite(heartbeatOut_A, LOW);  // stop heartbeat to Arduino A and
simulate failure
    digitalWrite(heartbeatOut_B, LOW);  // stop heartbeat to Arduino B and
simulate failure
    digitalWrite(statusLed_C, LOW);
    digitalWrite(ledPin_1, LOW);
    digitalWrite(ledPin_2, LOW);
    delay(10);
    return;  //skip rest of loop
  } else {
    digitalWrite(heartbeatOut_A, HIGH);  // send heartbeat to Arduino A if button
not pressed
    digitalWrite(heartbeatOut_B, HIGH);  // send heartbeat to Arduino B if button
not pressed
  }
  // Monitor Partner A Hearbeat
  if (digitalRead(heartbeatIn_A) == HIGH) {
    lastHeartbeat_A = currentMillis;
    partnerAlive_A = true;
    firstHeartbeatDetected_A = true;
  }
  // Monitor Partner B Hearbeat
```

```
  if (digitalRead(heartbeatIn_B) == HIGH) {
    lastHeartbeat_B = currentMillis;
    partnerAlive_B = true;
    firstHeartbeatDetected_B = true;
  }
  // determine partner A status
  if (firstHeartbeatDetected_A) {
    // Already saw a heartbeat, check if stopped
    if (currentMillis - lastHeartbeat_A > 3000) {
      partnerAlive_A = false;  // no recent heartbeat detected, partner failed
    } else {
      partnerAlive_A = true;  // heartbeat present
    }
  } else {
    // never saw a hearbeat, wait a short time after start up
    if (currentMillis > 2000) {
      partnerAlive_A = false;  // before 1st detection, assume partner alive,
stay passive
    } else {
      partnerAlive_A = true;  // still within starting window
    }
  }
  // determine partner B status
  if (firstHeartbeatDetected_B) {
    // Already saw a heartbeat, check if stopped
    if (currentMillis - lastHeartbeat_B > 3000) {
      partnerAlive_B = false;  // no recent heartbeat detected, partner failed
    } else {
      partnerAlive_B = true;  // heartbeat present
    }
  } else {
    // never saw a hearbeat, wait a short time after start up
    if (currentMillis > 2000) {
      partnerAlive_B = false;  // before 1st detection, assume partner alive,
stay passive
    } else {
      partnerAlive_B = true;  // still within starting window
    }
  }
  //determine role
  if (!partnerAlive_A) {
    // partner A dead, do not take control unless partner B is also dead
    isActive_C = false;
    digitalWrite(statusLed_C, LOW);  // stay passive
  } else {
```

```
    // partner alive, standby
    isActive_C = false;  // primary always continues unless its button is pressed
    digitalWrite(statusLed_C, LOW);
  }
  if (!partnerAlive_B) {
    // partner B dead, take control
    isActive_C = true;
    digitalWrite(statusLed_C, HIGH);  // indicate active status
  } else {
    // partner alive, standby
    isActive_C = false;  // primary always continues unless its button is pressed
    digitalWrite(statusLed_C, LOW);
  }
  // behaviour based on role
  if (isActive_C) {
    // Turn on ledPin_1 and delay to see state
    digitalWrite(ledPin_1, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_1 and delay to see state
    digitalWrite(ledPin_1, LOW);
    delay(blinkRateMilliseconds);
    // Turn on ledPin_2 and delay to see state
    digitalWrite(ledPin_2, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_2 and delay to see state
    digitalWrite(ledPin_2, LOW);
    delay(blinkRateMilliseconds);
  } else {
    // stay passive
    digitalWrite(ledPin_1, LOW);
    digitalWrite(ledPin_2, LOW);
    digitalWrite(statusLed_C, LOW);
  }
}
```

3.4.8 Fault Injection Mechanism Using DIP Switches

In the final stage of system development, the fault injection mechanism was refined through the integration of a DIP switch module, replacing the previously implemented pushbutton-based interface. This modification was introduced to address limitations observed in earlier simulations and to enable more stable and reproducible fault conditions.

In prior iterations, momentary pushbuttons were used to emulate node failures. While this approach provided basic functionality, it imposed practical constraints, as continuous manual

actuation was required to maintain a failure state. This dependency introduced inconsistencies in test execution and limited the ability to observe system behaviour over extended durations. To overcome these limitations, a DIP switch module was adopted, providing a bistable interface capable of maintaining a persistent state without user intervention.

From an implementation standpoint, the heartbeat control signal of each node was routed through an individual channel of the DIP switch module. When a switch was toggled to the “on” position, the corresponding signal path was interrupted, effectively simulating a node failure through the absence of heartbeat transmission. This method enabled deterministic control over fault conditions while preserving the simplicity and modularity of the circuit design.

The integration of the DIP switch resulted in a more controlled and repeatable testing environment. It allowed failure scenarios to be sustained over prolonged periods, thereby supporting more comprehensive observation and analysis of system response under fault conditions.

3.4.8.1 Circuit Architecture and Interconnection

The circuit configuration shown in Figure 29 represented the final iteration of the hardware design and constituted an evolution from the earlier pushbutton-based layout illustrated in Figure 27. In the initial implementation, three pushbuttons were incorporated to simulate node failures by manually interrupting the signal pathways. However, this approach required sustained physical interaction and limited consistency of fault simulation.

In the revised design, these pushbuttons were replaced with a DIP switch module, as shown in the updated circuit diagram. This modification enabled each fault condition to be maintained without continuous user input, thereby improving stability and repeatability during testing. The replacement of pushbuttons with a switch-based interface therefore represented a refinement of the fault injection mechanism, aligning the hardware implementation with the requirements for controlled and persistent failure simulation.

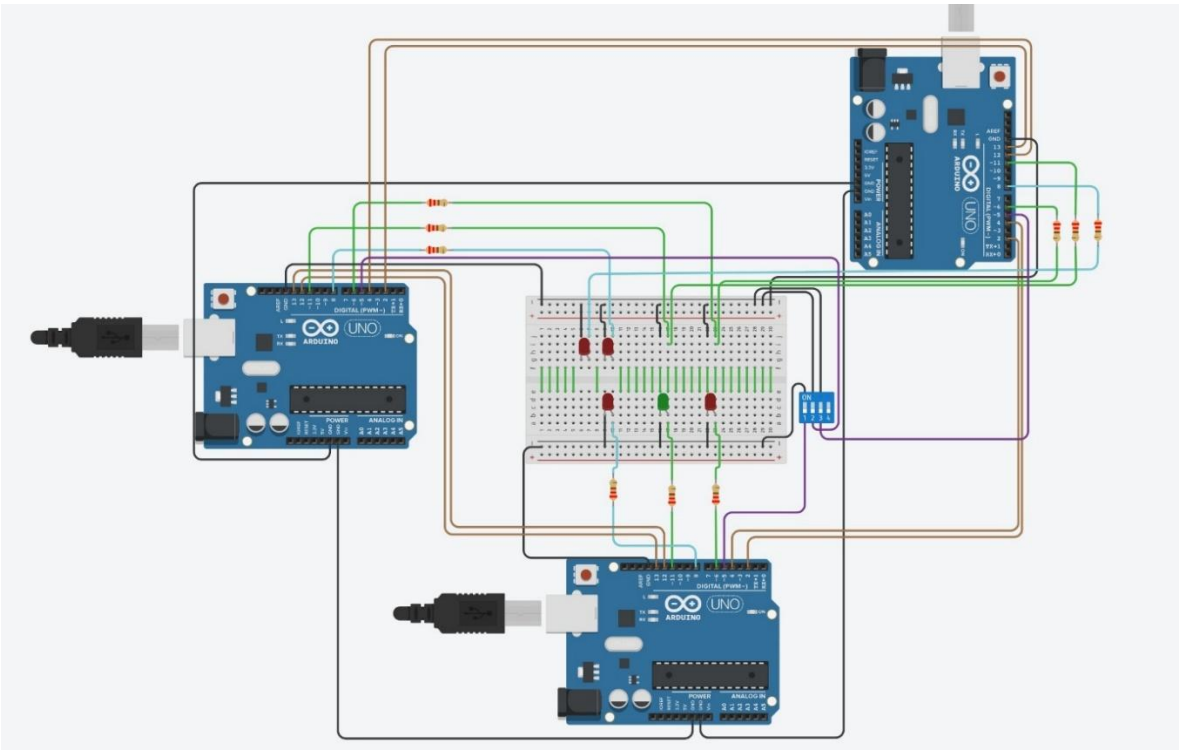

Figure 28. *3-Node configuration with DIP Switch*

3.4.8.2 Software Architecture and Communication Strategy

A uniform software architecture was implemented across all nodes, ensuring consistent system behaviour and simplifying the overall design. Each microcontroller executed an identical program, enabling dynamic role allocation based on system state rather than fixed configuration.

The communication strategy was based on a periodic heartbeat mechanism, whereby each node transmitted its operational status at predefined time intervals. Simultaneously, nodes monitored incoming heartbeat signals from their peers, maintaining an internal representation of network availability. Failure detection was achieved using a counter-based approach, in which consecutive missed heartbeat messages incremented a failure metric. When this metric exceeded a predefined threshold, the corresponding node was classified as inactive.

Upon detection of a node failure, a standby node automatically transitioned to an active state and assumed control responsibilities. This transition was reflected both logically within the system and visually through the LED indicators, providing immediate feedback on network status. When a previously failed node resumed operation, its status was reassessed and it was reintegrated into the network as a standby node.

3.5 Summary of System Development

This chapter presented the design and implementation of a fault-tolerant communication system based on interconnected Arduino nodes. The system evolved from a basic redundancy model into a distributed architecture capable of autonomous failure detection and recovery.

The results demonstrated that direct communication between microcontrollers was achievable without intermediary devices, and that heartbeat-based monitoring provided an effective mechanism for detecting node failures. The implementation of automatic failover ensured continued system operation in the presence of faults. Furthermore, the integration of DIP switch-based fault injection enhanced the precision and repeatability of simulation scenarios.

Overall, the developed system provided a practical framework for investigating redundancy and fault tolerance in embedded networks. The following chapter presents a detailed evaluation of system performance, focusing on timing characteristics, reliability, and failover effectiveness under varying conditions.

4.0 DISCUSSION OF RESULTS

This chapter presents and critically analyzes the physical results obtained from the implementation of the proposed redundancy system. The discussion is based on both simulation outcomes and practical hardware experimentation. The aim is to evaluate the effectiveness of the redundancy strategy, identify challenges encountered during implementation, and describe the modifications introduced to achieve a stable and reliable system.

The experimental setup consisted of three microcontroller units configured in a hierarchical redundancy architecture, where one unit operates as the primary controller and the others serve as backup controllers. The system was first validated using simulation tools before being physically implemented on a breadboard. Figures 29 to 40 in the Appendix show the individual components used for simulation.

4.1 Simulation Results and Identified Issues

Prior to the construction of the physical circuit, simulations were conducted to evaluate system behavior under different operating conditions. These simulations revealed several potential issues that could affect system performance and reliability.

4.1.1 Dimming of LEDs

One of the earliest issues observed during simulation was the dimming of the LEDs in the circuit controlled. The LEDs, which were intended to blink at full brightness when driven by the active Arduino, exhibited reduced intensity under certain conditions.

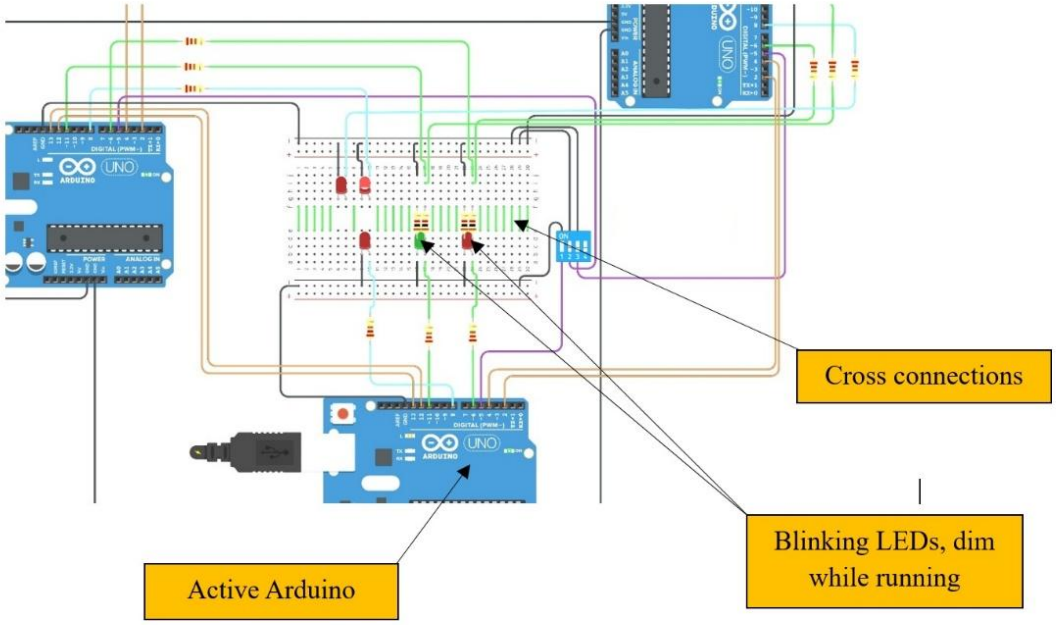

Figure 29. *Reduced intensity of LEDs due to Cross connections*

Upon investigation, it was determined that this issue was caused by unintended current flow between the interconnected Arduino units. Due to the shared control lines required for redundancy, passive Arduinos were inadvertently drawing current from the active Arduino. This resulted in a reduction of the current available to drive the LEDs, thereby causing the observed dimming effect.

The circuit design required cross-connections between the Arduinos, Figure 28, to allow any of them to assume control of the target circuit. While this approach enabled redundancy, it also created unintended current paths between the microcontrollers. It is important to note that such cross-connections were not necessary for the status LEDs, as each Arduino maintained an independent status indication circuit.

4.1.2 Circuit Improvement Using Zener Diodes

To address the issue of unintended current flow, several solutions were explored. These included the use of higher-rated resistors and NPN transistors; however, these approaches did not produce satisfactory results in terms of isolating the current paths between the Arduinos.

The most effective solution was found to be the introduction of 5.1 V Zener diodes. These diodes were used to enforce directional current flow, thereby preventing reverse current from flowing into passive Arduinos. By incorporating Zener diodes into the interconnections between each Arduino and the target circuit, it became possible to ensure that only the active Arduino supplied current to the load. This modification significantly improved the circuit behavior by eliminating unintended current sharing and restoring the expected brightness of the LEDs.

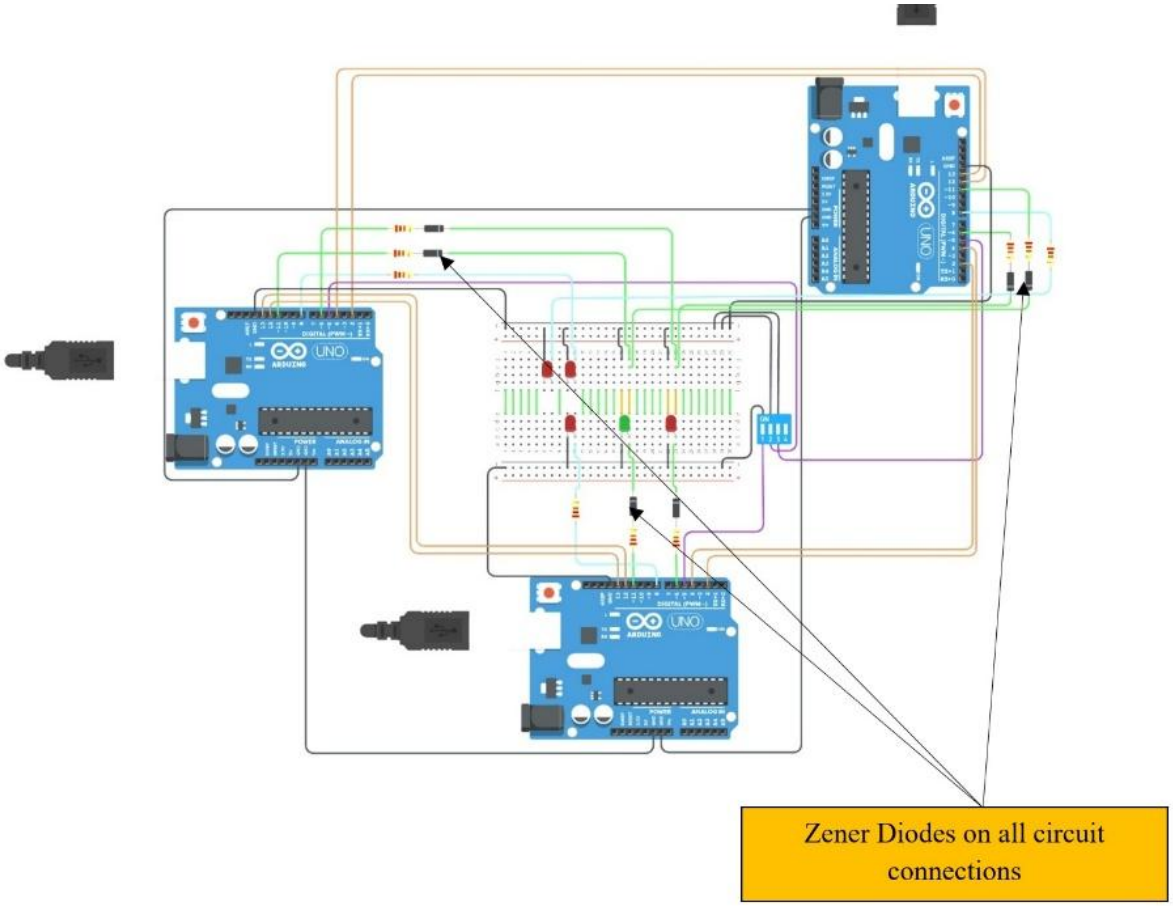

Figure 30. *Diodes on heartbeat lines*

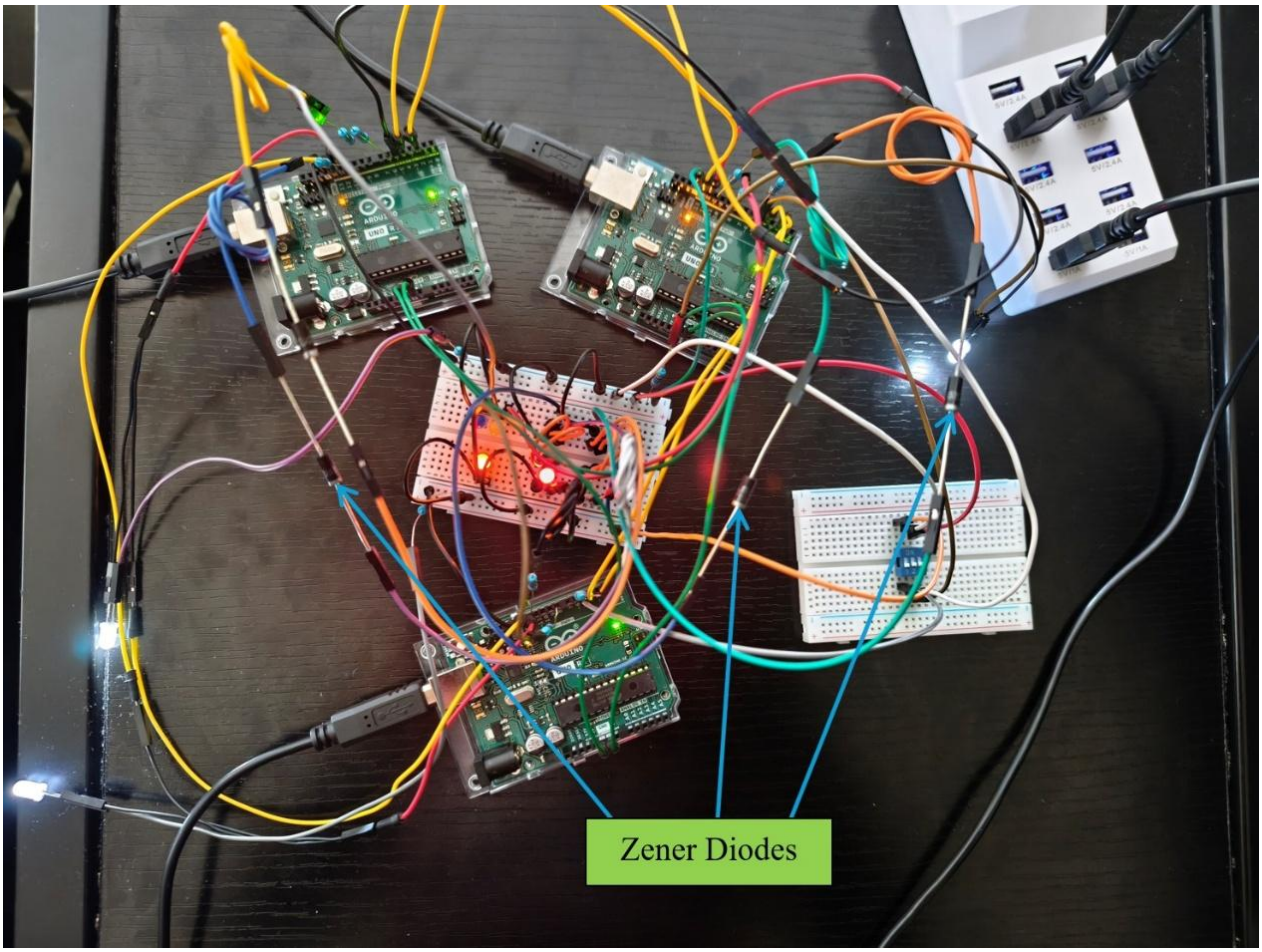

Figure 31. *Physical Zener Diodes on Heartbeat Lines*

4.2 Heartbeat Visualization

To enhance system observability and facilitate debugging, additional LEDs were introduced on the heartbeat signal lines. These LEDs served as visual indicators of heartbeat transmission between the Arduinos.

The inclusion of heartbeat indicators provided valuable insight into the internal communication of the system. It allowed for real-time observation of which Arduino was actively transmitting heartbeat signals and which ones were in a passive monitoring state. This proved particularly useful in identifying synchronization issues and unintended signal behavior during both simulation and physical testing.

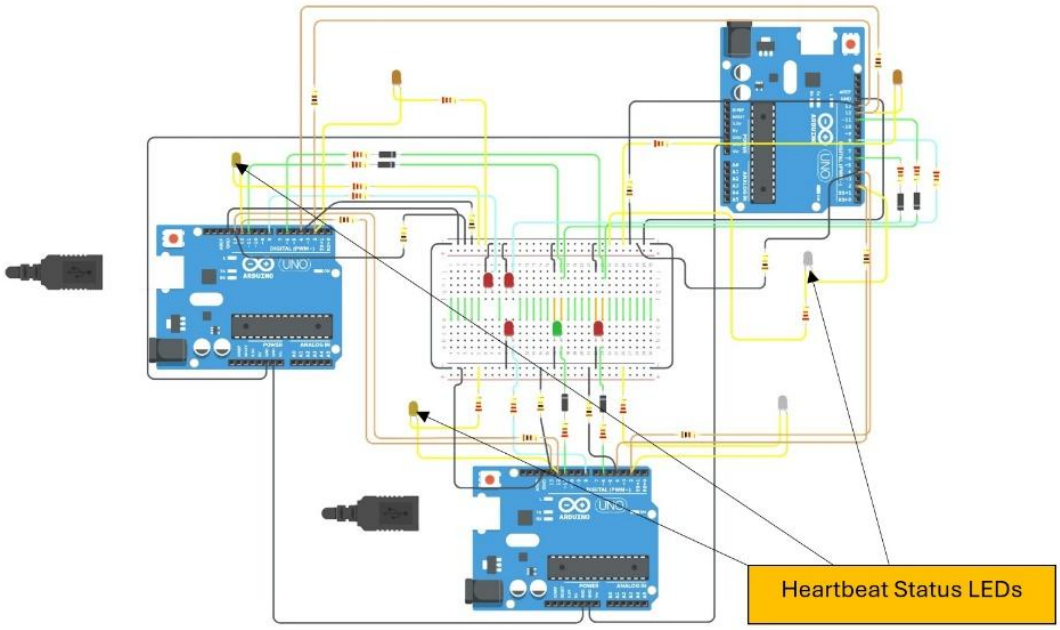

Figure 32. *LEDs showing active Heartbeat status for each Arduino*

4.3 Physical Circuit Layout

The physical implementation of the system consisted of three Arduino units interconnected through shared communication lines and a common ground reference. The circuit layout ensured that all Arduinos could both transmit and receive heartbeat signals from one another.

In the implemented design, white LEDs were used to indicate heartbeat transmission, while colored LEDs were used to represent the active status of each Arduino. Due to space limitations on the primary breadboard, certain components, such as the failure simulation switch, were placed on a secondary breadboard.

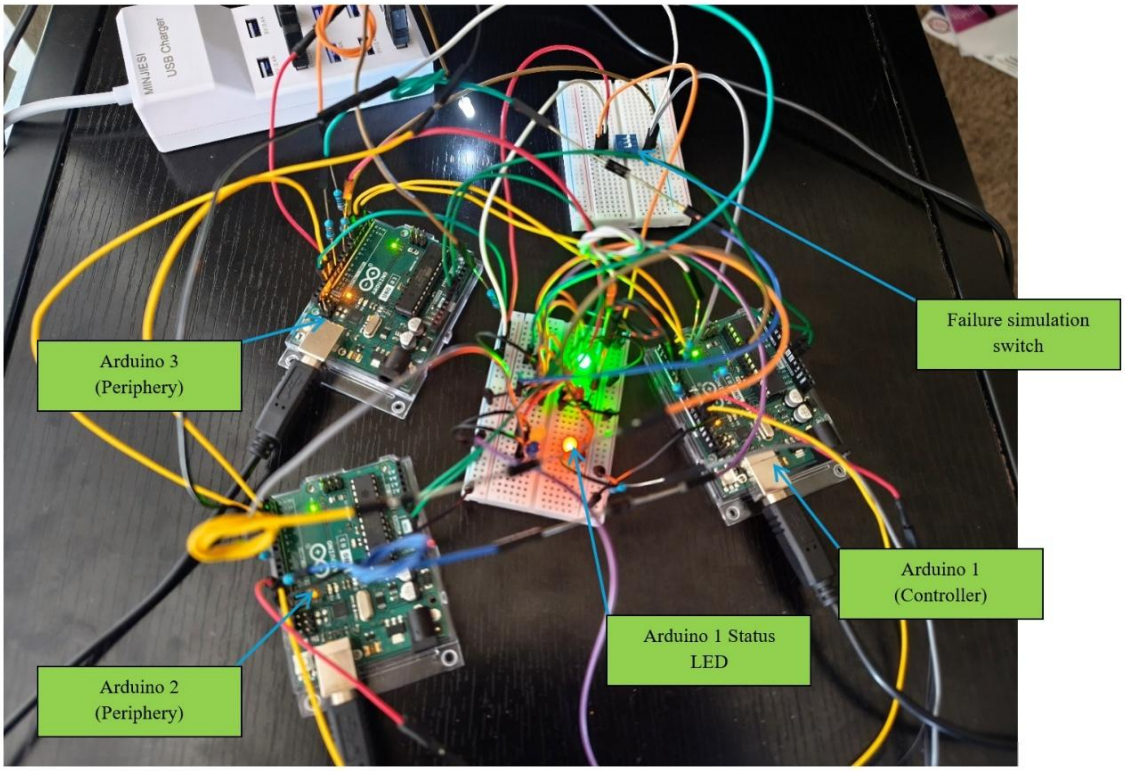

Figure 33. *Initial Circuit Layout with Failure Simulation Switch*

Initially, failure conditions were simulated using a DIP switch. While this method was effective in the simulation environment, it proved impractical in the physical setup. The physical activation of the switch introduced disturbances in the wiring, leading to inconsistent and unpredictable system behavior.

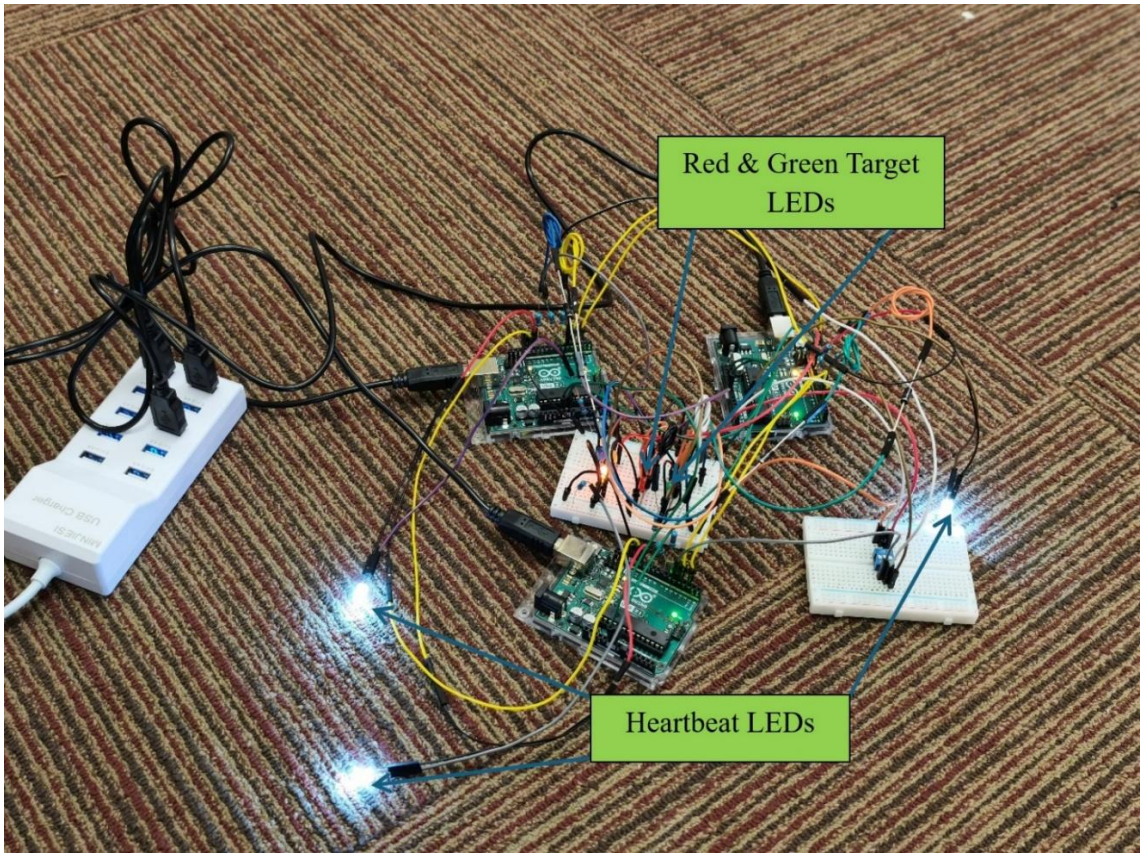

Figure 34. *Lit Status LEDs show that all Arduinos are healthy sending and receiving heartbeat signals to each other but only one Arduino controls the circuit since only one status LED is Lit*

4.4 Revision of Failure Simulation Approach

To address the limitations of switch-based failure simulation, the system was redesigned to eliminate the use of the DIP switch. Instead, failure conditions were simulated by physically disconnecting power from the Arduino units.

This modification aligned the testing process more closely with real-world scenarios, where failures are typically caused by power loss or hardware faults rather than controlled switch inputs. Correspondingly, the control logic was updated to detect the absence of heartbeat signals as an indication of failure.

The revised approach resulted in a more realistic and robust system, capable of responding appropriately to actual hardware failures.

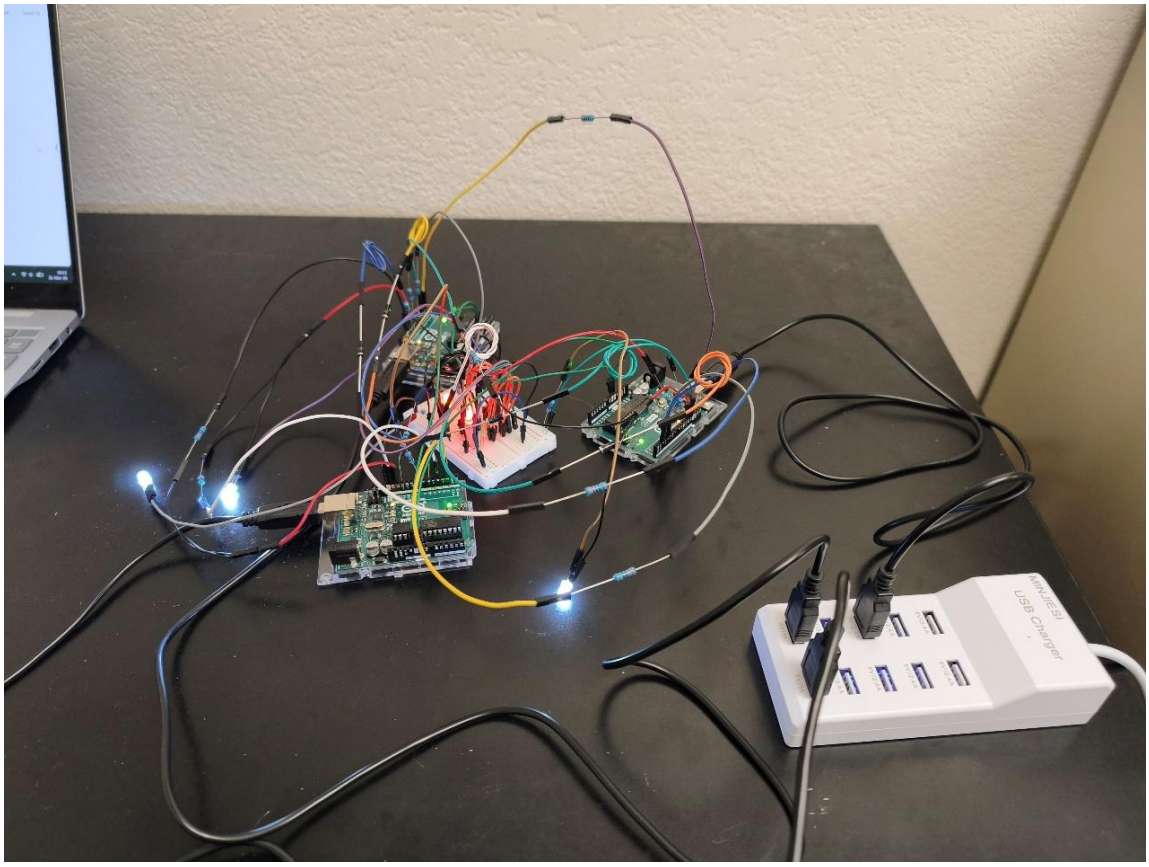

Figure 35. *Upgraded circuit with failure simulation switch removed*

4.5 Backflow Current Issue and Resolution

Following the removal of the switch and implementation of power-based failure simulation, a new issue emerged. When one Arduino was powered, it was observed that current flowed through the interconnecting lines into the other, unpowered Arduinos. As a result, the passive Arduinos exhibited partial activation, including weak heartbeat signals indicated by faintly lit LEDs.
This phenomenon posed a risk to microcontrollers, as backflow current can potentially damage sensitive components.
Further analysis revealed that the backflow occurred primarily through the heartbeat communication lines. To mitigate this issue, the circuit was enhanced with the following components:

- 1k Ω series resistors on each heartbeat output line (Pins 12 and 2)
- 10k Ω pull-down resistors on each heartbeat input line

The series resistors limited the current flowing between interconnected pins, while the pull-down resistors ensured that input lines remained at a defined LOW state when not actively driven. Together, these modifications effectively eliminated backflow current and stabilized the system.

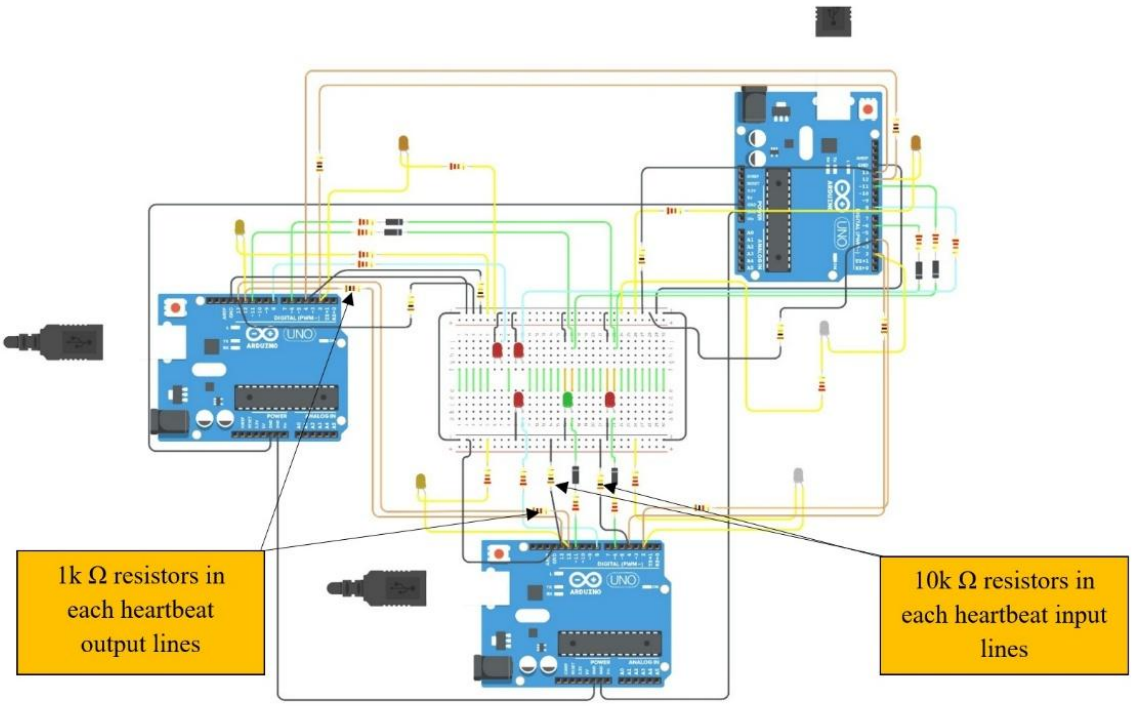

Figure 36. *Resistors in heartbeat output lines protected circuit from unwanted current flows between Arduinos*

4.5.1 System Performance After Improvements

With the improved circuit design, the system demonstrated reliable failover behavior:

- When the primary Arduino (Arduino A) lost power, Arduino B successfully assumed control.
- When Arduino B subsequently lost power, Arduino C took over control of the circuit.

This confirmed that the redundancy mechanism was functioning correctly in the forward failover direction.

4.5.2 Challenges in Reverse Failover

Despite successful forward failover, challenges were encountered during reverse failover. Specifically:

- When Arduino B was restored after failure, it successfully regained control from Arduino C.
- However, when Arduino A was restored, it did not immediately regain control from Arduino B or C.

This issue indicated a limitation in the heartbeat detection mechanism, particularly in how returning controllers were recognized by the backup units.

4.6 Improvement of Heartbeat Signal Design

The initial implementation of the heartbeat signal used a constant HIGH output. This approach proved inadequate for reliable detection, as it did not provide a distinguishable signal pattern for synchronization.
To address this, the heartbeat signal was modified to alternate HIGH and LOW states, introducing a pulsed signal. Initially, equal durations were used for both states (e.g., delay (40) for HIGH and LOW). However, this resulted in synchronization issues, as all Arduinos operated with identical timing, making it difficult for backup units to reliably detect the primary controller's return.
A more effective solution was achieved by introducing asymmetric pulse durations:

- Arduino A (Primary): HIGH = 250 ms
- Arduino B: HIGH = 100 ms
- Arduino C: HIGH = 50 ms
- LOW duration: 20 ms for all units

By assigning longer HIGH durations to higher-priority controllers, the system established a clear hierarchy in signal visibility. This ensured that when the primary Arduino (Arduino A) returned online, its heartbeat signal was more easily detected by the backup units.

```
digitalWrite(heartbeatOut_B, HIGH);
digitalWrite(heartbeatOut_C, HIGH);
delay(250);
digitalWrite(heartbeatOut_B, LOW);
digitalWrite(heartbeatOut_C, LOW);
delay(20);
```

Figure 37. *Modified Code for Arduino A Heartbeat*

```
digitalWrite(heartbeatOut_A, HIGH);
digitalWrite(heartbeatOut_C, HIGH);
delay(100);
digitalWrite(heartbeatOut_A, LOW);
digitalWrite(heartbeatOut_C, LOW);
delay(20);
```

Figure 38. *Modified Code for Arduino B Heartbeat*

```
digitalWrite(heartbeatOut_A, HIGH);  // send heartbeat to Arduino A
digitalWrite(heartbeatOut_B, HIGH);  // send heartbeat to Arduino B
delay(50);
digitalWrite(heartbeatOut_A, LOW);  // send heartbeat to Arduino A
digitalWrite(heartbeatOut_B, LOW);  // send heartbeat to Arduino B
delay(20);
```

Figure 39. *Modified Code for Arduino C Heartbeat*

This modification significantly improved system performance, enabling proper reverse failover behavior where control was correctly returned to the primary controller upon recovery.

These experimental results demonstrated that the proposed redundancy system is capable of achieving reliable failover and recovery under realistic operating conditions. Through iterative design improvements, key challenges such as current backflow, signal instability, and failure

detection were successfully addressed. The final system incorporates both electrical and logical safeguards, including current-limiting resistors, pull-down stabilization, and prioritized heartbeat signaling. These enhancements collectively contribute to a robust and fault-tolerant architecture suitable for practical applications.

5.0 CONCLUSIONS AND RECOMMENDATIONS

This chapter presents the key conclusions derived from the design, simulation, and physical implementation of the proposed Arduino-to-Arduino redundancy communication system. The study aimed to develop a distributed embedded architecture capable of autonomous failure detection and recovery without reliance on intermediary control units. By enabling direct communication between microcontrollers, the system reduces single points of failure while improving overall operational reliability.

In addition to summarizing the main findings, this chapter highlights the broader significance of the work, particularly in the context of mission-critical and resource-constrained environments such as CubeSat systems and remote embedded platforms. Recommendations for future improvements are also provided, with emphasis on enhancing system robustness and extending fault detection beyond controller-level monitoring.

5.1 Conclusions

The results obtained from both simulation and physical testing confirm that the research objectives outlined in this thesis have been successfully achieved. Specifically, the study addressed the key limitations identified in Chapter 2 regarding centralized control, hardware-intensive redundancy, and the lack of lightweight fault-tolerant solutions for Arduino-class embedded systems.

A primary contribution of this work is the elimination of reliance on centralized or intermediary controllers, which were identified as single points of failure in traditional architectures. By implementing a direct microcontroller-to-microcontroller communication framework, the proposed system enables fully decentralized operation, where each Arduino node independently monitors system health and participates in decision-making. This directly resolves the structural weakness associated with centralized coordination in distributed embedded systems.

In addition, the research addresses the limitations of hardware-based redundancy approaches, such as Triple Modular Redundancy (TMR), which, while effective, are impractical for resource-constrained platforms due to increased mass, power consumption, and system complexity. The proposed system demonstrates that software-based redundancy using heartbeat monitoring and deterministic failover can achieve reliable fault tolerance without requiring hardware duplication. This significantly reduces system overhead while maintaining functional resilience.

The study further resolves the identified gap in lightweight fault detection mechanisms for low-cost microcontrollers. Through the implementation of a bidirectional heartbeat-based monitoring protocol, each node is capable of autonomously detecting failures in real time. The use of time-out-based detection combined with distributed observation ensures that faults are identified promptly without the need for complex middleware, operating systems, or high computational resources.

Another key outcome is the successful implementation of deterministic, software-controlled failovers, addressing the lack of simple yet reliable recovery strategies in existing Arduino-based systems. The system demonstrated stable failover behavior across both two-node and three-node configurations, with a clearly defined hierarchical structure ensuring conflict-free control reassignment. This confirms that distributed monitoring combined with prioritized control execution is an effective approach for maintaining system continuity.

The transition from simulation to physical implementation further validated the practicality of the proposed approach. Challenges such as unintended current paths, signal interference, and synchronization issues were identified and mitigated through simple hardware enhancements, including current-limiting and pull-down resistors. These findings reinforce the importance of integrating minimal hardware safeguards alongside software redundancy in real-world deployments.

Importantly, this research also addresses the constraints of CubeSat and space-based embedded systems, where strict limitations on size, weight, power, and computational resources make traditional redundancy approaches unsuitable. The proposed architecture demonstrates that reliable fault tolerance can be achieved using minimal wiring, low-power microcontrollers, and simple communication protocols, making it highly applicable to small satellite platforms and other resource-constrained environments.

Furthermore, the evolution of the heartbeat signaling strategy from a constant HIGH signal to a pulsed signal with asymmetric timing resolved synchronization and prioritization challenges observed during testing. This refinement highlights the importance of signal design in distributed systems and contributes to improved robustness and deterministic behavior.

From a broader perspective, this work demonstrates that software-driven, peer-to-peer redundancy architectures can effectively replace traditional hardware-heavy solutions in certain classes of embedded systems. By minimizing additional components and avoiding centralized dependencies, the proposed system reduces potential points of failure while maintaining scalability and adaptability.

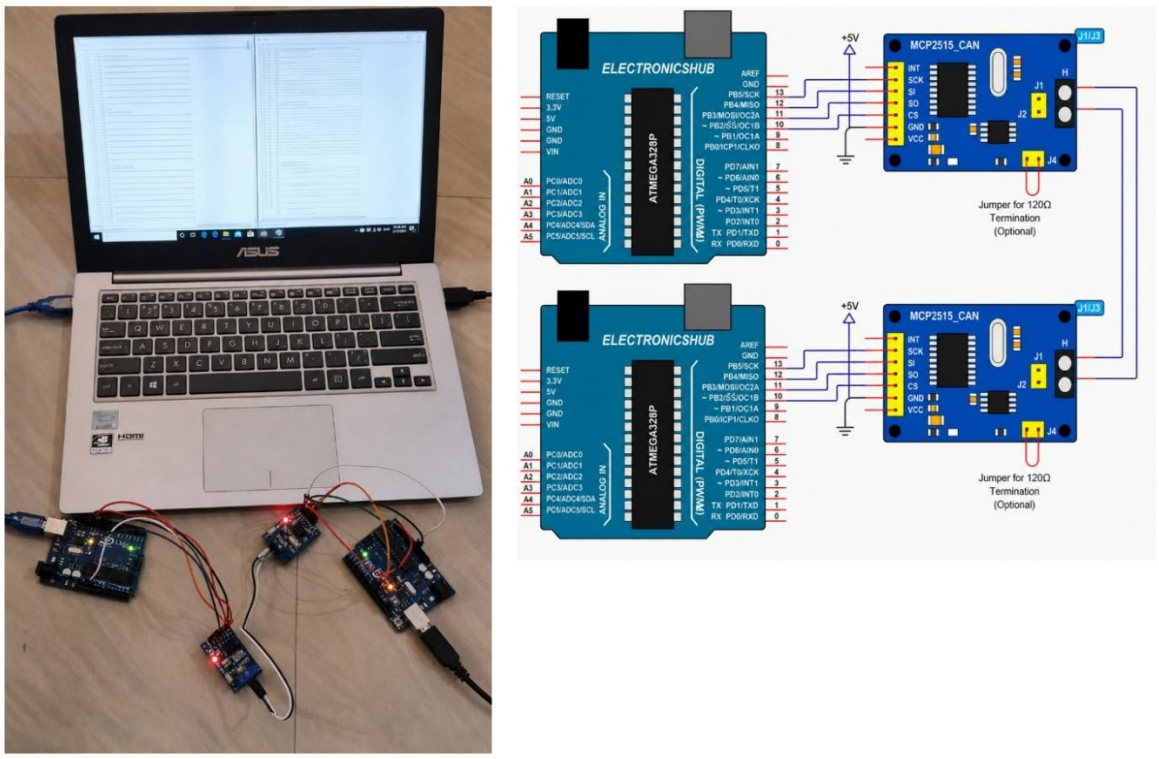

Figure 40. *Arduino-to-Arduino communication using CAN interface modules (MCP2515), illustrating structured peer-to-peer communication through dedicated hardware transceivers. courtesy (Lakshmi & Kumar, 2022)*

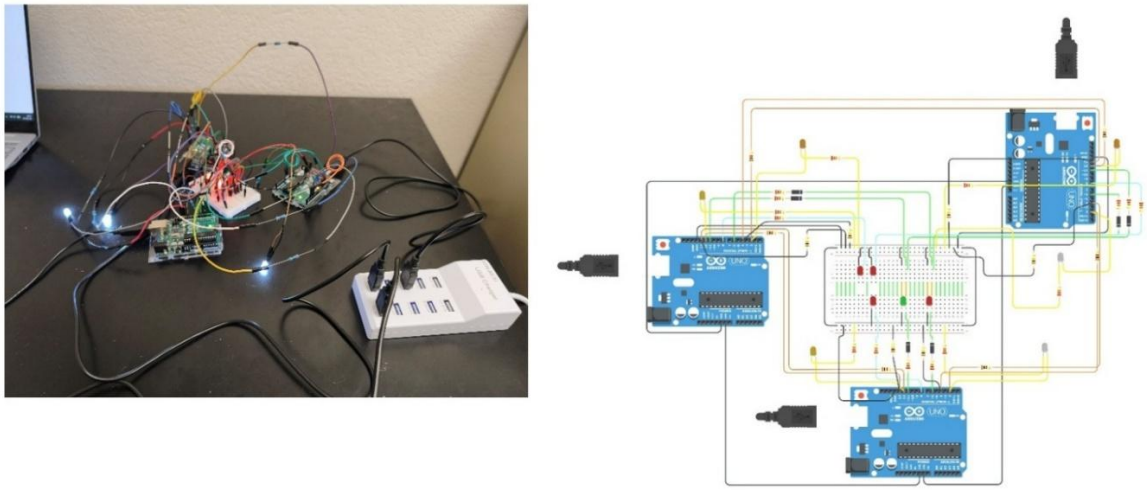

Figure 41. *Expanded multi-node Arduino testbed implementing peer monitoring and redundancy logic, with corresponding circuit schematic for distributed fault-tolerant operation.*

Figures 40 and 41 illustrate the evolution of Arduino-to-Arduino communication toward a more structured and scalable implementation. Figure 40 demonstrates the use of CAN interface modules

(MCP2515) to establish reliable peer-to-peer communication between microcontrollers through dedicated hardware transceivers. Building on this, Figure 41 presents a fully developed multi-node testbed incorporating distributed monitoring and redundancy logic, alongside its corresponding circuit schematic. This progression reflects the transition from basic communication concepts to a robust architecture capable of supporting fault-tolerant, peer-based operation.

5.2 Recommendations

While the developed system demonstrates strong performance in detecting controller-level failures and enabling reliable failovers, there remains an opportunity to further enhance robustness by extending fault detection beyond the microcontrollers themselves. A key recommendation arising from this study is the incorporation of target circuit connectivity monitoring. In the current implementation, an Arduino node is considered healthy as long as it remains powered and capable of transmitting heartbeat signals. However, this assumption does not account for scenarios in which the controller is operational but unable to effectively drive the target circuit due to issues such as broken connections, faulty wiring, or load disconnection.

Future interventions of the system could therefore include mechanisms for verifying circuit completeness and output integrity. This can be achieved by introducing feedback pathways from the controlled circuit back to the Arduino. By comparing the expected output state with the actual state observed at the load, each controller can assess not only its operational status but also its ability to perform its intended function.

In such an enhanced design, if an active Arduino detects that it cannot successfully drive the target circuit, it should classify itself as faulty and cease heartbeat transmission. This would enable the remaining nodes in the network to interpret the failure and initiate controlled failovers to another controller. Incorporating this additional layer of verification would significantly improve system reliability by enabling functional redundancy, rather than relying solely on node availability.

Additionally, future work should involve testing the system under more realistic environmental conditions, including variations in power supply, temperature, and electromagnetic interference. Such testing would provide deeper insight into the system's suitability for real-world deployment, particularly in aerospace and industrial applications.

5.3 Final Remarks

This study demonstrates that reliable and scalable redundancy can be achieved in simple embedded systems through the thoughtful integration of hardware and software design principles. By combining direct communication, intelligent signal design, and decentralized decision-making, the proposed system establishes a strong foundation for low-cost fault-tolerant architectures.

The progression from simulation to physical implementation not only validated the feasibility of the proposed approach but also revealed important practical considerations necessary for real-world deployment. The insights gained throughout this process contribute meaningfully to the field of embedded system redundancy and provide a basis for further development and innovation.

Ultimately, this work emphasizes the importance of designing systems that are not only functionally correct but also resilient, capable of adapting to failures and maintaining operation with minimal disruption. This principle is particularly critical in applications such as CubeSat missions, where reliability and longevity are paramount and opportunities for physical intervention are extremely limited.

References

Afonso, F., Silva, C., Tavares, A., & Montenegro, S. (2008). Application-level fault tolerance in real-time embedded systems. *2008 International Symposium on Industrial Embedded Systems*, 126–133. https://doi.org/10.1109/SIES.2008.4577690

Balasubramanian, P. (2016). ASIC-based design of NMR system health monitor for mission/safety–critical applications. *SpringerPlus*, *5*(1). https://doi.org/10.1186/s40064-016-2249-7

Balasubramanian, P., & Prasad, K. (2016). *A Fault Tolerance Improved Majority Voter for TMR System Architectures*.

Barbosa, Raul. (2008). *Layered fault tolerance for distributed embedded systems*. Chalmers University of Technology.

Baumann, R. C. (2005). Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, *5*(3), 305–316. https://doi.org/10.1109/TDMR.2005.853449

Bowoto, O. K., Oladapo, B. I., Nimako, P. A., Omigbodun, F. T., & Emenuvwe, O. A. (2019). *Arduino Fault Tolerant System, Internet of Things (IoT)*. https://doi.org/10.20944/preprints201906.0034.v1

Brambilla, M., Ferrante, E., Birattari, M., & Dorigo, M. (2013). Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, *7*(1), 1–41. https://doi.org/10.1007/s11721-012-0075-2

Bucciero, M., Walters, J. P., & French, M. (2011). Software fault tolerance methodology and testing for the embedded PowerPC. *2011 Aerospace Conference*, 1–9. https://doi.org/10.1109/AERO.2011.5747460

Csaba, S. (2025). *Mathematical Model and Digital Control of a Twin Power Converter*.

Cui, X., Gao, Q., Wang, R., Liu, L., Liang, J., & Peng, Y. (2019). Fault-tolerant method for anti-SEU of embedded system based on dual-core processor. *The Journal of Engineering*, *2019*(23), 8755–8759. https://doi.org/10.1049/joe.2018.9099

Dhrubo, A. F. H., & Qayum, M. A. (2025). *STM32-Based IoT Framework for Real-Time Environmental Monitoring and Wireless Node Synchronization*. http://arxiv.org/abs/2506.17295

Eric Olson. (2019, June 21). *How to develop a CubeSat: From concept to Earth orbit*. https://insights.globalspec.com/article/11912/how-to-develop-a-cubesat-from-concept-to-earth-orbit?utm

Fayyaz, M. (2015). *Task Oriented Fault-Tolerant Distributed Computing for Use on Board Spacecraft*.

Fuchs, C. M., Stefanov, T. P., Murillo, N. M., & Plaat, A. (2017). Bringing Fault-Tolerant GigaHertz-Computing to Space: A Multi-stage Software-Side Fault-Tolerance Approach for Miniaturized Spacecraft. *2017 IEEE 26th Asian Test Symposium (ATS)*, 100–107. https://doi.org/10.1109/ATS.2017.30

Gaisler, J. (2002). A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. *Proceedings International Conference on Dependable Systems and Networks*, 409–415. https://doi.org/10.1109/DSN.2002.1028926

Giovanni Blu Mitolo. (2022, February 10). *PJON 13.1*. https://github.com/gioblu/PJON

Grün, C. (2025). *A Realistic Simulator for Regression Testing within an Embedded Defence System*.

Györök, G., & Beszédes, B. (2017). *Fault-tolerant Software Solutions in Microcontroller Based Systems*.

Hadifur Rahman, M. (2017). A Fault Tolerant Voter Circuit for Triple Modular Redundant System. *Journal of Electrical and Electronic Engineering*, *5*(5), 156. https://doi.org/10.11648/j.jeee.20170505.11

Jim Blom. (n.d.). *Serial Communication*. Spark Fun Electronics. Retrieved February 14, 2026, from https://learn.sparkfun.com/tutorials/serial-communication#uarts

Khatri, A. R. (2020). Overview of Fault Tolerance Techniques and the Proposed TMR Generator Tool for FPGA Designs. *International Journal of Advanced Computer Science and Applications*, *11*(4). https://doi.org/10.14569/IJACSA.2020.0110497

Klesh, A., & Krajewski, J. (2015). *MarCO: CubeSats to Mars in 2016*. https://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=3180&context=smallsat

Lakshmi, S., & Kumar, R. H. (2022). Secure Communication between Arduinos using Controller Area Network(CAN) Bus. *2022 IEEE International Power and Renewable Energy Conference, IPRECON 2022*. https://doi.org/10.1109/IPRECON55716.2022.10059518

Langer, M., & Bouwmeester, J. (2016). *Reliability of CubeSats – Statistical Data, Developers' Beliefs and the Way Forward*.

Lauer, M., Amy, M., Fabre, J.-C., Roy, M., Excoffon, W., & Stoicescu, M. (2016). Engineering Adaptive Fault-Tolerance Mechanisms for Resilient Computing on ROS. *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, 94–101. https://doi.org/10.1109/HASE.2016.30

Leens, F. (2009). An introduction to I2C and SPI protocols. *IEEE Instrumentation & Measurement Magazine*, *12*(1), 8–13. https://doi.org/10.1109/MIM.2009.4762946

Lyons, R. E., & Vanderkulk, W. (1962). *The Use of Triple-Modular Redundancy to Improve Computer Reliability*. https://doi.org/https://doi.org/10.1147/rd.62.0200

Mahmood, H. A., & Khairullah, S. S. (2025). Design and Simulation of a Dependable Architecture Using Triple Modular Redundancy for Embedded Cyber-Physical Systems. *Journal of Electronic Testing*, *41*(1), 63–74. https://doi.org/10.1007/s10836-025-06159-5

Mike Grusin. (n.d.). *Serial Peripheral Interface (SPI)*. Spark Fun Electronics. Retrieved February 14, 2026, from https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi

Mishra, S., & Khilar, P. M. (2012). *Heartbeat Based Error Diagnosis Framework For Distributed Embedded Systems* (Z. Zeng & Y. Li, Eds.; pp. 834935-834935–834937). https://doi.org/10.1117/12.920945

NASA CubeSat Launch Initiative. (2017). *CubeSat 101:Basic Concepts and Processes for First-Time CubeSat Developers*.

New Haven Display International. (2025, September 26). *Serial vs Parallel Communication*. https://newhavendisplay.com/blog/serial-vs-parallel-communication/

Powell, D. (Ed.). (2001). *A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems*. Springer US. https://doi.org/10.1007/978-1-4757-3353-2

Puig-Suari, J., Turner, C., & Ahlgren, W. (2002). Development of the standard CubeSat deployer and a CubeSat class PicoSatellite. *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, 1/347-1/353. https://doi.org/10.1109/AERO.2001.931726

Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., & August, D. I. (2005). SWIFT: Software Implemented Fault Tolerance. *International Symposium on Code Generation and Optimization*, 243–254. https://doi.org/10.1109/CGO.2005.34

Rennels, D. A., Caldwell, D. W., Hwang, R., & Mesarina, M. (1997). A fault-tolerant embedded microcontroller testbed. *Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems*, 7–14. https://doi.org/10.1109/PRFTS.1997.640118

Rogenmoser, M., Tortorella, Y., Rossi, D., Conti, F., & Benini, L. (2025). Hybrid Modular Redundancy: Exploring Modular Redundancy Approaches in RISC-V Multi-core Computing Clusters for Reliable Processing in Space. *ACM Transactions on Cyber-Physical Systems*, *9*(1), 1–29. https://doi.org/10.1145/3635161

SFE. (n.d.). *12C*. Spark Fun Electronics. Retrieved February 14, 2026, from https://learn.sparkfun.com/tutorials/i2c

Sharma, G., & Yadav, A. (2018). Fault Tolerance in Real Time Distributed System. *Review of Computer Engineering Research*, *5*(2), 20–24. https://doi.org/10.18488/journal.76.2018.52.20.24

Sharpe, T., Thomas, E., Coyle, J., Neff, J., Salvinelli, C., & Whiting, G. (n.d.). *Design and Validation of Low-Cost Environmental Monitoring Systems to Support Basic and Ecosystem Service Delivery*.

Shukla, S., & Ray, K. C. (2022). A Low-Overhead Reconfigurable RISC-V Quad-Core Processor Architecture for Fault-Tolerant Applications. *IEEE Access*, *10*, 44136–44146. https://doi.org/10.1109/ACCESS.2022.3169495

Solouki, M. A., Angizi, S., & Violante, M. (2024). Dependability in Embedded Systems: A Survey of Fault Tolerance Methods and Software-Based Mitigation Techniques. *IEEE Access*, *12*, 180939–180967. https://doi.org/10.1109/ACCESS.2024.3509633

Swartwout, M. (2013). *The First One Hundred CubeSats: A Statistical Look*. www.JoSSonline.com

Vallado, D. A. ., & McClain, W. D. . (2013). *Fundamentals of astrodynamics and applications*. Microcosm Press.

Villela, T., Costa, C. A., Brandão, A. M., Bueno, F. T., & Leonardi, R. (2019). Towards the Thousandth CubeSat: A Statistical Overview. *International Journal of Aerospace Engineering*, *2019*, 1–13. https://doi.org/10.1155/2019/5063145

Wertz, J. Richard., Everett, D. F. ., & Puschell, J. John. (2018). *Space mission engineering : the new SMAD*. Microcosm Press.

Zheng, J., Okamura, H., & Dohi, T. (2021). Availability Analysis of Software Systems with Rejuvenation and Checkpointing. *Mathematics*, *9*(8), 846. https://doi.org/10.3390/math9080846

Appendix

This appendix provides the Arduino source code used to implement the proposed redundancy system. The code defines the behavior of each Arduino node within the network, including heartbeat transmission, failure detection, and automatic role switching.

Each Arduino periodically sends a heartbeat signal to indicate that it is operational while simultaneously monitoring incoming heartbeat signals from other nodes. If a node fails to detect a heartbeat from a peer within a specified timeout period, it classifies that node as inactive.

Based on the status of other nodes, the code implements a predefined priority logic to determine which Arduino should assume the active control role. Only one node is allowed to control the target circuit at any given time, while the others remain in standby mode. If the active node fails, a standby node automatically takes over control, ensuring continuous system operation.

The code also manages output behavior, such as activating or deactivating LEDs, to reflect the current role (active or standby) of each node.

Code for Controller Arduino A

```
// Primary Arduino A
// Sketch to blink 2 LEDs, show status of Arduino A, simulate failure, monitor
heartbeat of
// backup Arduinos B and C
// Declare and initiallize LED pin variables
byte ledPin_1 = 6;
byte ledPin_2 = 11;
const byte heartbeatOut_B = 12;  // send heartbeat to Arduino B
const byte heartbeatIn_B = 13;   // receive heartbeat from Arduino B
const byte statusLed_A = 8;      // active status indicator
//const byte buttonPin_A = 5;      // simulate failure
const byte heartbeatOut_C = 2;      // send heartbeat to Arduino C
const byte heartbeatIn_C = 4;       // receive heartbeat from Arduino C
unsigned long lastHeartbeat_B = 0;  // Arduino B
unsigned long lastHeartbeat_C = 0;  // Arduino C
bool partnerAlive_B = false;
bool partnerAlive_C = false;
bool isActive_A = true;  // Arduino A starts as active
void setup() {
  // put your setup code here, to run once:
  pinMode(ledPin_1, OUTPUT);
```

```
  pinMode(ledPin_2, OUTPUT);
  pinMode(heartbeatOut_B, OUTPUT);
  digitalWrite(heartbeatOut_B, LOW);  //Sets the default output volatage to zero
  pinMode(heartbeatIn_B, INPUT);
  pinMode(heartbeatOut_C, OUTPUT);    // Sets pin2 as an output to Arduino C
  digitalWrite(heartbeatOut_C, LOW);  //Sets the default output volatage to zero
  pinMode(heartbeatIn_C, INPUT);      // Sets pin4 as an input for signal from
Arduino C
  pinMode(statusLed_A, OUTPUT);
}
void loop() {
  // Turn LEDs on and off
  // Variable to specify the blink rate in milliseconds
  int blinkRateMilliseconds = 90;
  unsigned long currentMillis = millis();
  digitalWrite(heartbeatOut_B, HIGH);
  digitalWrite(heartbeatOut_C, HIGH);
  delay(250);
  digitalWrite(heartbeatOut_B, LOW);
  digitalWrite(heartbeatOut_C, LOW);
  delay(20);
  // Monitor Partner B Heartbeat
  if (digitalRead(heartbeatIn_B) == HIGH) {
    lastHeartbeat_B = currentMillis;
    partnerAlive_B = true;
  }
  if (currentMillis - lastHeartbeat_B > 1000) {
    partnerAlive_B = false;
  }
  // Monitor Partner C Heartbeat
  if (digitalRead(heartbeatIn_C) == HIGH) {
    lastHeartbeat_C = currentMillis;
    partnerAlive_C = true;
  }
  if (currentMillis - lastHeartbeat_C > 1000) {
    partnerAlive_C = false;
  }
  //determine role
  if (partnerAlive_B && partnerAlive_C) {
    // either partner on, stay active
    isActive_A = true;
  } else if (partnerAlive_B && !partnerAlive_C) {
    // partner B alive, stay active
    isActive_A = true;  // stay active nonetheless
  } else if (!partnerAlive_B && partnerAlive_C) {
```

```
    isActive_A = true;  // stay active
  } else if (!partnerAlive_B && !partnerAlive_C) {
    // as long as partner A is dead, stay active
    isActive_A = true;  // stay active
  } else {
    // partner alive, only one should run
    isActive_A = true;  // primary always active
  }
  // Active blinking
  if (isActive_A) {
    digitalWrite(statusLed_A, HIGH);
    // Turn on ledPin_1 and delay to see state
    digitalWrite(ledPin_1, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_1 and delay to see state
    digitalWrite(ledPin_1, LOW);
    delay(blinkRateMilliseconds);
    // Turn on ledPin_2 and delay to see state
    digitalWrite(ledPin_2, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_2 and delay to see state
    digitalWrite(ledPin_2, LOW);
    delay(blinkRateMilliseconds);
  } else {
    // stay passive
    digitalWrite(ledPin_1, LOW);
    digitalWrite(ledPin_2, LOW);
    digitalWrite(statusLed_A, LOW);
  }
}
```

Code for Backup Arduino B

```
// Backup Arduino B
// Sketch to blink 2 LEDs, show status of Arduino B, Monitor heartbeat of Arduino
A and
// Simulate failure
// Declare and initiallize LED pin variables
byte ledPin_1 = 6;
byte ledPin_2 = 11;
const byte heartbeatOut_A = 12;  // send heartbeat to Arduino A
const byte heartbeatIn_A = 13;   // receive heartbeat from Arduino A
const byte statusLed_B = 8;      // active status indicator
//const byte buttonPin_B = 5;      // simulate failure
```

```
const byte heartbeatOut_C = 2;  // send heartbeat to Arduino C
const byte heartbeatIn_C = 4;   // receive heartbeat from Arduino C
unsigned long lastHeartbeat_A = 0;
unsigned long lastHeartbeat_C = 0;
bool partnerAlive_A = false;
bool partnerAlive_C = false;
bool isActive_B = false;                // Arduino B starts in passive mode
bool firstHeartbeatDetected_A = false;  // Assume no heartbeat from Arduino A if
no heartbeat is received yet
bool firstHeartbeatDetected_C = false;  // Assume no heartbeat from Arduino C if
no heartbeat is received yet
void setup() {
  pinMode(ledPin_1, OUTPUT);
  pinMode(ledPin_2, OUTPUT);
  pinMode(heartbeatOut_A, OUTPUT);
  digitalWrite(heartbeatOut_A, LOW);  //Sets the default output volatage to zero
  pinMode(heartbeatIn_A, INPUT);
  pinMode(heartbeatOut_C, OUTPUT);    // Sets pin2 as an output to Arduino C
  digitalWrite(heartbeatOut_C, LOW);  //Sets the default output volatage to zero
  pinMode(heartbeatIn_C, INPUT);      // Sets pin4 as an input for signal from
Arduino C
  pinMode(statusLed_B, OUTPUT);
  // start in passive mode
  digitalWrite(ledPin_1, LOW);
  digitalWrite(ledPin_2, LOW);
  digitalWrite(statusLed_B, LOW);
}
void loop() {
  // Variable to specify the blink rate in milliseconds
  int blinkRateMilliseconds = 90;
  unsigned long currentMillis = millis();
  digitalWrite(heartbeatOut_A, HIGH);  // send heartbeat to Arduino A
  digitalWrite(heartbeatOut_C, HIGH);  // send heartbeat to Arduino C
  delay(100);
  digitalWrite(heartbeatOut_A, LOW);  // send heartbeat to Arduino A
  digitalWrite(heartbeatOut_C, LOW);  // send heartbeat to Arduino C
  delay(20);
  // Monitor Partner A Heartbeat
  if (digitalRead(heartbeatIn_A) == HIGH) {
    lastHeartbeat_A = currentMillis;
    partnerAlive_A = true;
    firstHeartbeatDetected_A = true;
    // A is back, immediately give up control
    isActive_B = false;
    digitalWrite(ledPin_1, LOW);
```

```
    digitalWrite(ledPin_2, LOW);
    digitalWrite(statusLed_B, LOW);
  }
  // Monitor Partner C Heartbeat
  if (digitalRead(heartbeatIn_C) == HIGH) {
    lastHeartbeat_C = currentMillis;
    partnerAlive_C = true;
    firstHeartbeatDetected_C = true;
  }
  // determine partner A status
  if (firstHeartbeatDetected_A) {
    // Already saw a heartbeat, check if stopped
    if (currentMillis - lastHeartbeat_A > 1000) {
      partnerAlive_A = false;  // no recent heartbeat detected, partner failed
    } else {
      partnerAlive_A = true;  // heartbeat present
    }
  } else {
    // never saw a hearbeat, wait a short time after start up
    if (currentMillis > 1000) {
      partnerAlive_A = false;  // before 1st detection, assume partner alive,
stay passive
    } else {
      partnerAlive_A = true;  // still within starting window
    }
  }
  // determine partner C status
  if (firstHeartbeatDetected_C) {
    // Already saw a heartbeat, check if stopped
    if (currentMillis - lastHeartbeat_C > 1000) {
      partnerAlive_C = false;  // no recent heartbeat detected, partner failed
    } else {
      partnerAlive_C = true;  // heartbeat present
    }
  } else {
    // never saw a hearbeat, wait a short time after start up
    if (currentMillis > 1000) {
      partnerAlive_C = false;  // before 1st detection, assume partner alive,
stay passive
    } else {
      partnerAlive_C = true;  // still within starting window
    }
  }
  //determine role
  if (partnerAlive_A && partnerAlive_C) {
```

```
    // partner A alive and C alive, stay passive
    isActive_B = false;
  } else if (partnerAlive_A && !partnerAlive_C) {
    // partner A alive, stay passive
    isActive_B = false;  // stay passive if A is active
  } else if (!partnerAlive_A && !partnerAlive_C) {
    // partner A dead, take over
    isActive_B = true;  // Take over if A is dead
  } else if (!partnerAlive_A && partnerAlive_C) {
    // as long as partner A is dead, stay active
    isActive_B = true;  // stay active even if C is on
  }
  // behaviour based on role
  if (isActive_B) {
    digitalWrite(statusLed_B, HIGH);
    // Turn on ledPin_1 and delay to see state
    digitalWrite(ledPin_1, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_1 and delay to see state
    digitalWrite(ledPin_1, LOW);
    delay(blinkRateMilliseconds);
    // Turn on ledPin_2 and delay to see state
    digitalWrite(ledPin_2, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_2 and delay to see state
    digitalWrite(ledPin_2, LOW);
    delay(blinkRateMilliseconds);
  } else {
    // stay passive
    digitalWrite(ledPin_1, LOW);
    digitalWrite(ledPin_2, LOW);
    digitalWrite(statusLed_B, LOW);
  }
}
```

Code for Backup Arduino C

```
// Backup Arduino C
// Sketch to blink 2 LEDs, show status of Arduino B, Monitor heartbeat of Arduino
A and
// Simulate failure
// Declare and initiallize LED pin variables
byte ledPin_1 = 6;
byte ledPin_2 = 11;
```

```
const byte heartbeatOut_B = 12;  // send heartbeat to Arduino B
const byte heartbeatIn_B = 13;   // receive heartbeat from Arduino B
const byte statusLed_C = 8;      // active status indicator
//const byte buttonPin_C = 5;      // simulate failure
const byte heartbeatOut_A = 2;  // send heartbeat to Arduino A
const byte heartbeatIn_A = 4;   // receive heartbeat from Arduino A
unsigned long lastHeartbeat_A = 0;
unsigned long lastHeartbeat_B = 0;
bool partnerAlive_A = false;
bool partnerAlive_B = false;
bool isActive_C = false;                // Arduino C starts in passive mode
bool firstHeartbeatDetected_A = false;  // Assume no heartbeat from Arduino A if
no heartbeat is received yet
bool firstHeartbeatDetected_B = false;  // Assume no heartbeat from Arduino B if
no heartbeat is received yet
void setup() {
  pinMode(ledPin_1, OUTPUT);
  pinMode(ledPin_2, OUTPUT);
  pinMode(heartbeatOut_B, OUTPUT);
  digitalWrite(heartbeatOut_B, LOW);  //Sets the default output volatage to zero
  pinMode(heartbeatIn_B, INPUT);
  pinMode(heartbeatOut_A, OUTPUT);    // Sets pin2 as an output to Arduino C
  digitalWrite(heartbeatOut_A, LOW);  //Sets the default output volatage to zero
  pinMode(heartbeatIn_A, INPUT);      // Sets pin4 as an input for signal from
Arduino C
  pinMode(statusLed_C, OUTPUT);
  // start in passive mode
  digitalWrite(ledPin_1, LOW);
  digitalWrite(ledPin_2, LOW);
  digitalWrite(statusLed_C, LOW);
}
void loop() {
  // Variable to specify the blink rate in milliseconds
  int blinkRateMilliseconds = 90;
  unsigned long currentMillis = millis();
  digitalWrite(heartbeatOut_A, HIGH);  // send heartbeat to Arduino A
  digitalWrite(heartbeatOut_B, HIGH);  // send heartbeat to Arduino B
  delay(50);
  digitalWrite(heartbeatOut_A, LOW);  // send heartbeat to Arduino A
  digitalWrite(heartbeatOut_B, LOW);  // send heartbeat to Arduino B
  delay(20);
  // Monitor Partner A Heartbeat
  if (digitalRead(heartbeatIn_A) == HIGH) {
    lastHeartbeat_A = currentMillis;
    partnerAlive_A = true;
```

```
    firstHeartbeatDetected_A = true;
    // A is back, immediately give up control
    isActive_C = false;
    digitalWrite(ledPin_1, LOW);
    digitalWrite(ledPin_2, LOW);
    digitalWrite(statusLed_C, LOW);
  }
  // Monitor Partner B Heartbeat
  if (digitalRead(heartbeatIn_B) == HIGH) {
    lastHeartbeat_B = currentMillis;
    partnerAlive_B = true;
    firstHeartbeatDetected_B = true;
  }
  // determine partner A status
  if (firstHeartbeatDetected_A) {
    // Already saw a heartbeat, check if stopped
    if (currentMillis - lastHeartbeat_A > 1000) {
      partnerAlive_A = false;  // no recent heartbeat detected, partner failed
    } else {
      partnerAlive_A = true;  // heartbeat present
    }
  } else {
    // never saw a hearbeat, wait a short time after start up
    if (currentMillis > 1000) {
      partnerAlive_A = false;  // before 1st detection, assume partner alive,
stay passive
    } else {
      partnerAlive_A = true;  // still within starting window
    }
  }
  // determine partner B status
  if (firstHeartbeatDetected_B) {
    // Already saw a heartbeat, check if stopped
    if (currentMillis - lastHeartbeat_B > 1000) {
      partnerAlive_B = false;  // no recent heartbeat detected, partner failed
    } else {
      partnerAlive_B = true;  // heartbeat present
    }
  } else {
    // never saw a hearbeat, wait a short time after start up
    if (currentMillis > 1000) {
      partnerAlive_B = false;  // before 1st detection, assume partner alive,
stay passive
    } else {
      partnerAlive_B = true;  // still within starting window
```

```
    }
  }
  //determine role
  if (partnerAlive_A && partnerAlive_B) {
    // partner A and B alive, stay passive
    isActive_C = false;
  } else if (partnerAlive_A && !partnerAlive_B) {
    // partner A alive, stay passive
    isActive_C = false;  // stay passive if A is active
  } else if (!partnerAlive_A && partnerAlive_B) {
    // partner A dead but partner B alive, stay passive
    isActive_C = false;  // stay passive if B is active
  } else if (!partnerAlive_A && !partnerAlive_B) {
    // both partners are dead, take over
    isActive_C = true;  // only take over if both are dead
  }
  // behaviour based on role
  if (isActive_C) {
    digitalWrite(statusLed_C, HIGH);
    // Turn on ledPin_1 and delay to see state
    digitalWrite(ledPin_1, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_1 and delay to see state
    digitalWrite(ledPin_1, LOW);
    delay(blinkRateMilliseconds);
    // Turn on ledPin_2 and delay to see state
    digitalWrite(ledPin_2, HIGH);
    delay(blinkRateMilliseconds);
    // Turn off ledPin_2 and delay to see state
    digitalWrite(ledPin_2, LOW);
    delay(blinkRateMilliseconds);
  } else {
    // stay passive
    digitalWrite(ledPin_1, LOW);
    digitalWrite(ledPin_2, LOW);
    digitalWrite(statusLed_C, LOW);
  }
}
```

Individual Circuit Components

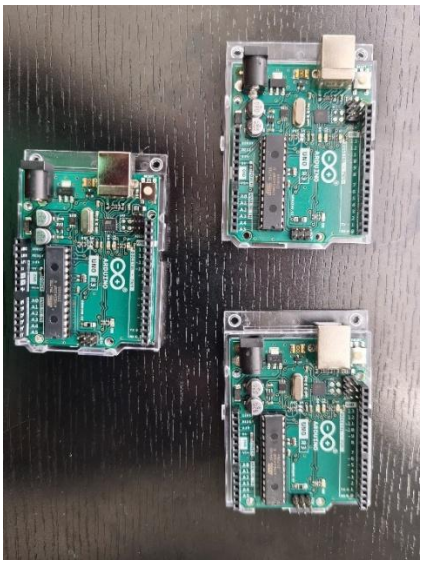

Figure 42. *Arduino UNO REV Boards*

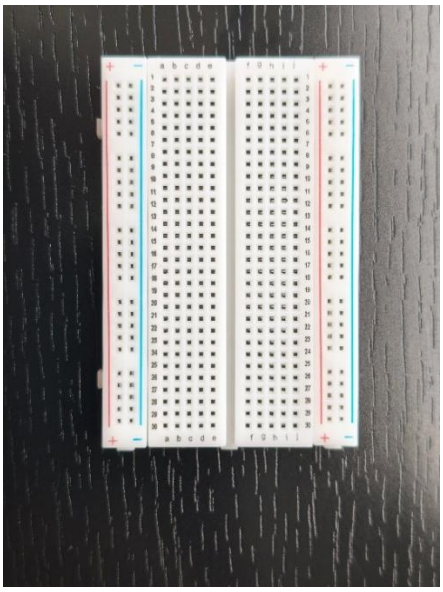

Figure 43. *Solderless Breadboard*

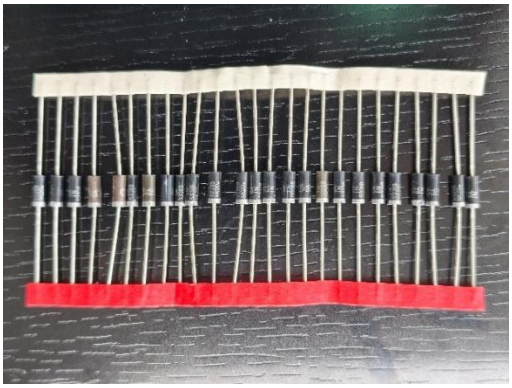

Figure 44. *1N5338B Zener Diodes. 5.1V, 5W*

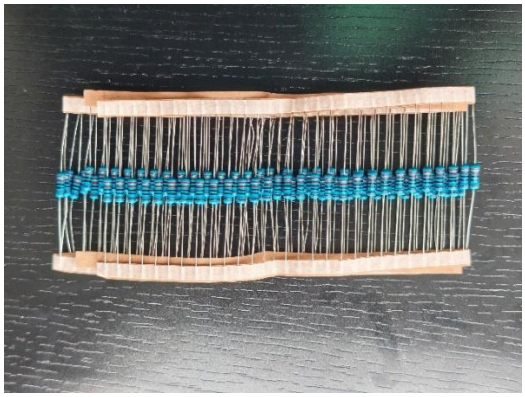

Figure 45. *220 Ohm Resistors*

Figure 46. *USB Data Sync Cables for Arduino*

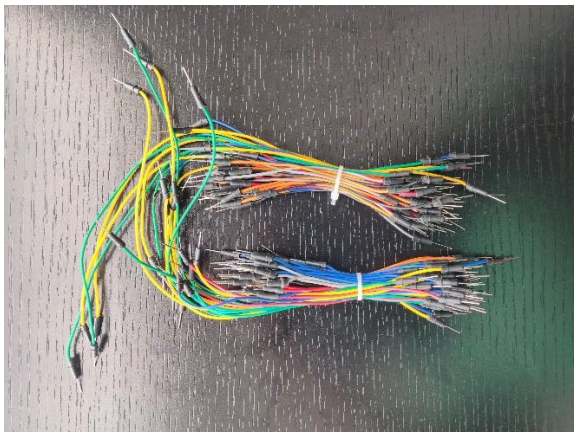

Figure 47. *Male-Male Cables*

Figure 48. *Single Female to Dual Female Jumpers*

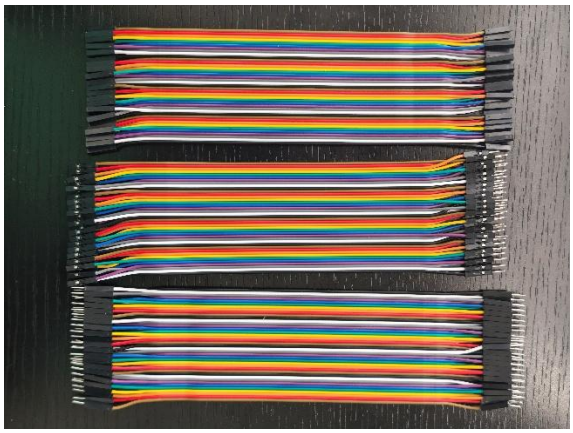

Figure 49. *Multi Coloured DuPont Jumper wires*

Figure 50. *10 Port USB Multi Charging port, 50W, 10A*

Figure 51. *Digital Multimeter*

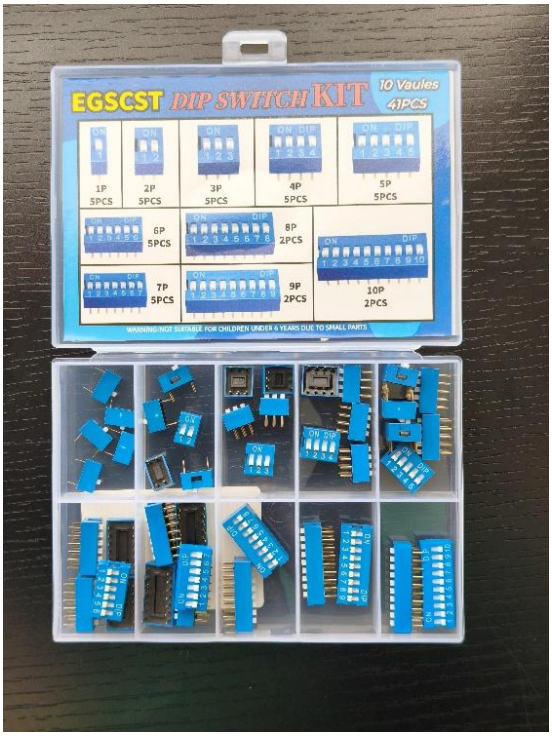

Figure 52. *DIP Switch Assortment Kit*

Figure 53. *MultiColoured LED Diode Assortment*