Bridging Cross-Domain Communication in Asynchronous and Synchronous Digital Systems

Petros Petrou

Undergraduate Student

Supervised by:

Professor Julius Georgiou

A Thesis submitted in fulfilment of requirements for the degree of

Bachelor of Science in Computer Engineering

Department of Electrical and Computer Engineering

University of Cyprus

May, 2024

Abstract

This thesis investigates the dynamics of communication between synchronous and asynchronous digital circuits within semiconductor technologies to enhance their interoperability and efficiency. It begins by examining the evolution of semiconductor technology and its implications on current circuit designs, focusing on functionality, reliability, and performance enhancements. Through a comparative analysis, the operational characteristics and advantages of both synchronous and asynchronous systems are explored, highlighting the innovative solutions to minimize latency, maximize throughput, and ensure robust data integrity through advanced communication protocols and design strategies. The practical research phases involve comprehensive assessments of hardware and software aspects critical to the communication process. The hardware evaluation focuses on selecting appropriate technologies to serve as intermediaries in the communication process, while the software discussion revolves around the development of interfaces and middleware crucial for facilitating efficient data transfer. The implementation showcases a system designed to enable effective and reliable communication, significantly enhancing the efficiency and reliability of semiconductor systems through the strategic integration of synchronous and asynchronous elements, coupled with optimized hardware and software solutions. This research contributes to the field of asynchronous and synchronous systems by clarifying the complex interactions within integrated circuits and setting a foundation for future technological advancements.

Contents

Chapter 1

Introduction

1.1 Motivation

Technology, a broad and multidimensional phenomenon whose precise terminology can hardly be ascertained since it includes many different categories. As a simplistic definition, it can be attributed to the application of science aimed at the practical achievement of a function. It can also be paralleled to a modern-day steam engine, which is constantly moving at increasing speed towards the future. A future that is getting closer regardless of whether it seems utopian or dystopian.

In fact, technology is not just one tool or set of tools, but a dynamic and ever-changing web of interdependent and interacting elements that shape the way we live, work, and generally exist.

In general, technology starts with human ingenuity and the endless effort to improve existing living conditions. From the invention of the wheel to the creation of the Internet, technology has evolved from simple tools to complex systems and networks that are interconnected and communicate, even on a global scale. With its help, man was freed to a large extent from the domination of nature and utilized natural forces to his advantage. Thus, he improved his standard of living by having many comforts at his disposal. His diet, housing and clothing have been improved and he enjoys many things that make his life easier and better. At the same time, he built machines that increased production but also expanded his free time, which he could utilize with various forms of entertainment.

Its contribution to medical science is also a catalyst that has succeeded in

curing previously incurable diseases and increasing life expectancy. In addition, man has managed to enrich his knowledge in every field of science, to get rid of stereotypes, prejudices, and superstitions, and to acquire spiritual resources that help him to correctly assess social reality. Finally, he developed the means of transportation and mass communication resulting in the annihilation of distances that favored cultural exchanges, contact, and cooperation of peoples, laying the foundations for a peaceful global community.

When we refer to the modern achievements and era of technology, we mean a period where technology is not just present but dominant. In all this dominance and industry of modern technology, digital circuits stand as the fundamental building blocks, orchestrating the symphony of electrons that power our digital world. From ubiquitous smartphones in our pockets to sprawling data centers that power the cloud, digital circuits are the cornerstone upon which the edifice of contemporary computing is built. As we delve into this world, we encounter two protagonists: synchronous and asynchronous digital circuits, each playing a unique role in the vast domain of semiconductor technology.

Synchronous circuits, with their clock-driven rhythm, ensure a harmonious and predictable flow of data, reminiscent of a conductor leading an orchestra with precision. In contrast, asynchronous circuits, free from the constraints of a global clock, offer flexibility and efficiency by navigating the complexities of data flow with a dynamic cadence. The interaction between these two types of circuits is not just a technical dialogue but an of innovation and challenge in the semiconductor industry.

This thesis embarks on an exploratory journey through the digital circuit communication. In doing so, it illuminates the profound implications of their interaction, casting light on issues of timing, data coherence, and system reliability. As it navigates this landscape, it uncovers how the communication between these circuits shapes the performance and efficiency of modern semiconductor technology.

In pursuing this exploration, the thesis aims not only to dissect the technical intricacies of synchronous and asynchronous circuit communication but also to contribute to the broader discourse on advancing semiconductor technology.

The motivation for this thesis focuses on the important challenges of commu-

nication between synchronous and asynchronous digital circuits, which are key to improving semiconductor technology. This communication is essential for the smooth operation of systems that use both types of timing, crucial for making devices that are more efficient, faster, and smaller. By improving the way these circuits talk to each other, we aim to make systems more reliable and perform better, paving the way for future advances.

1.2 Research Objectives

The overarching objective of this thesis is to extensively study the theoretical background surrounding semiconductor technology in general and its evolution over the years. A comprehensive literature review is conducted with a focal point on the two fundamental philosophies and designs of digital circuits, synchronous and asynchronous. Furthermore, it highlights the key characteristics of these circuits, their design, and also their advantages and disadvantages when using them either separately or in collaboration.

The main purpose is to investigate the communication between synchronous and asynchronous digital circuits, which is a crucial aspect of digital circuit design. Following this, the technical part focuses on the variety of challenges that might occur during the application of the theory at an implementation level. To examine and test such challenges, a communication system is being implemented whose main purpose is the exchange of simple instructions and packetized messages between a synchronous interface and an asynchronous interface.

The vision of this system is summarized in Figure 1.1 with the main objectives covering the following:

- To establish stable and synchronized – in terms of timing - communication between a synchronous and an asynchronous circuit.
- To provide the programmer with a user-friendly and easy-to-use environment to send and receive the desired bit streams between the two digital circuits.
- To minimize any delays and increase the transferring latency.

- To monitor and regulate the appropriate voltage levels and guarantee internal compatibility.

1.2.1 Features of the Proposed System

The proposed system consists of three main parts:

- The first part of the system features an application software with a user-friendly graphical user interface which gives the ability to the user to input and send the desired data. In addition, the software is cross-platform and compatible with multiple systems.

- The second part contains an embedded system that acts as a bridge between the user interface and the actual asynchronous circuit. Its main purpose is to store the bit streams that come from the user software to its internal memory and also forward them to the asynchronous chip with a human command. Moreover, it is capable of making calls to the asynchronous chip, asking it to send its data for verification.

- The final part involves the connection that the embedded device establishes with the asynchronous circuit and the circuit itself. The asynchronous circuit can accept the incoming bit streams. Additionally, it can send acknowledge back to the embedded system.

Each area shown in Figure 1.1 is also a separate investigation which consists of the selection of the appropriate material based on the required conditions each time, as well as the methodology that will be followed to carry out the implementation.

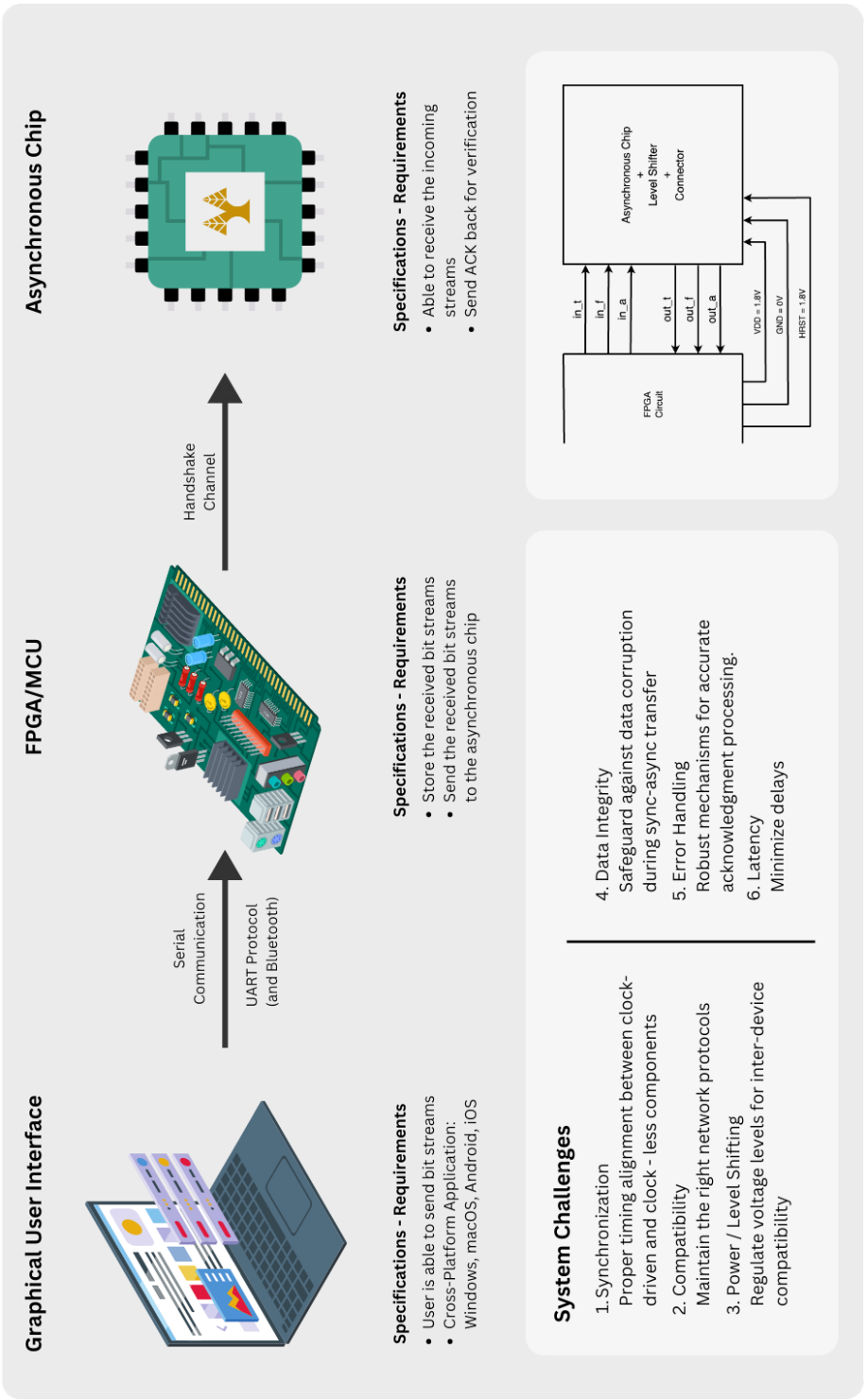


Figure 1.1 Overview of the proposed system

1.3 Thesis Structure

The thesis is divided into three main parts:

1.3.1 Part I - Foundational Concepts and Theoretical Background

Part 1 covers all the theoretical study and the literature review. It dives into the essential theories regarding the semiconductor technology and summarizes key concepts.

Ch 2: Semiconductor Technology and Foundations

Chapter 2 explores the principles of semiconductor technology, its evolution, and the impact on circuit design, laying the groundwork for understanding the technical aspects discussed in later chapters.

Ch 3: Synchronous Systems

Chapter 3 delves into the fundamentals of synchronous system design, including clock domains and reliability factors.

Ch 4: Asynchronous Systems

Chapter 4 introduces asynchronous design principles, communication protocols, advantages and disadvantages, and tools for asynchronous design.

Ch 5: Comparative Analysis between Synchronous and Asynchronous

Chapter 5 offers a detailed comparison of synchronous and asynchronous systems, focusing on process and mismatch variationa, and clock domain transitions.

1.3.2 Part II - Comparative Technical Exploration and Hardware Selection

Part II presents the comprehensive study conducted to construct the system's implementation, shown in Figure 1.1.

Ch 6: Selecting Hardware for Communication

Chapter 6 discusses the selection process for communication hardware which is used as bridge of the synchronous and the asynchronous parts, comparing Field-Programmable Gate Arrays(FPGAs) and Microcontrollers(MCUs) in terms of design and programming considerations, performance metrics, and reliability.

Ch 7: Reliable Communication Establishment

Chapter 7 focuses on finding reliable communication protocols for the two ends that they ensuring consistency and integrity.

Ch 8: The Engineering of the Software

Chapter 8 delves into the study for the user software and the selection of an appropriate software programming language and framework, starting with a clear outline of the software's requirements.

1.3.3 Part III - Practical Implementation, Evaluation and Synthesis

Part III focuses on the implementation and testing phases and also presents the study's results and the implications for future research and development in the field.

Ch 9: Implementation and Testing

Chapter 9 covers the practical aspects of the system design and implementation, shown in , including the graphical user interface, the device communication, coding, simulation, and reliability testing.

Ch 10: Results and Discussion

Chapter 10 presents the findings of the study and, more specifically, from the implementation, including performance evaluation and reliability analysis.

Part I

Part I: Foundational Concepts and Theoretical Background

Chapter 2

Semiconductor Technology and Foundations

2.1 Overview of Semiconductor Principles

Semiconductors are materials that possess electrical conductivity between that of conductors and insulators. This unique characteristic is derived from the energy band structure of the materials, where the energy difference between the valence band (fully occupied by electrons) and the conduction band (where electrons move freely) can be manipulated by doping—introducing impurities into the semiconductor's crystalline structure. Doping adjusts the electron and hole concentrations in the material, which directly influences the semiconductor's conductivity. The behavior of electrons and holes within these materials under various conditions forms the fundamental basis for semiconductor devices. Devices such as diodes, transistors, and integrated circuits exploit these properties for a variety of applications. Today, semiconductors play a pivotal role in the advancement of technology, powering everything from everyday electronic devices to sophisticated systems in communications, computing, and renewable energy [1] [2].

2.2 The Evolution of Semiconductor Technology

The development of semiconductor technology, characterized by the shrinking size of transistors and wires, is critical to various areas such as mobile technology,

communications, consumer electronics, medicine, and the automotive industry. Miniaturization allows chips to include more transistors, increasing their ability to perform demanding tasks. Smaller transistors and wires reduce resistance and capacitance, improving performance and reducing power consumption in integrated circuits.

2.2.1 Merits from the Semiconductor Technology Evolution

- **Miniaturization:** The main driving force. As transistors and wires get smaller, chips can hold more transistors, reaching transistor counts in the order of hundreds of billions. This increase in transistor density leads to quantitative changes in integrated circuit performance.
- **Communication Revolution:** Semiconductors are at the heart of modern communication systems, enabling high-speed internet, mobile phone networks, and the Global Positioning System (GPS), facilitating global connectivity and real-time information sharing.
- **Contribution in Healthcare:** Semiconductor technology has revolutionized healthcare with innovations such as medical imaging, DNA sequencing, and wearable health monitors, improving disease diagnosis, treatment, and patient care.
- **Industrial Automation:** Semiconductor growth has radically shifted manufacturing and industrial processes through automation and robotics, leading to increased efficiency, precision, and safety in various industries.

2.2.2 Challenges from the Semiconductor Technology Evolution

- **Fabrication Limitations:** As the physical limits of Moore's Law are approached, downsizing electronic components encounters fundamental barriers. Current printing technologies are reaching their limits, and new methods

such as ultraviolet scanning (EUV) or alternatives like quantum dots and nanotube applications must be developed.

- **Thermal Management:** Increasing transistor density raises heat dissipation on the chip. Proper heat management is crucial for reliability and performance. Designers must create cooling solutions that can withstand high heat loads without compromising the compactness of the device.
- **Signal Integrity:** As components become denser and operate at faster speeds, maintaining signal integrity becomes challenging. Crosstalk, electrical interference, and electrical noise affect circuit performance. Solutions involve design methods and rigid materials to protect or reduce interference.
- **Power Consumption:** Despite the lower power consumption of the transistors, the high inductance results in significant power consumption, especially in mobile and wearable technologies where battery life is critical. Industry must innovate in energy-efficient and efficient power management technologies.
- **Complexity of Integration:** Incorporating power electronic technologies into systems increases the complexity of design, testing, and end use. Ensuring compatibility with existing systems and achieving the required level of reliability requires a multidisciplinary approach combining knowledge from materials science, electrical engineering, and computer science.

2.3 Impact on Circuit Design

The relentless advancement in semiconductor technology has significantly influenced the way electronic circuits are designed today. This impact is seen not just in the capabilities of the circuits but in their entire creation process, from conceptualization to final product.

2.3.1 Design for Manufacturability (DFM)

Modern circuit design heavily incorporates DfM principles, ensuring that the designs are not only theoretically sound but also practical for manufacturing. This

includes adopting techniques that minimize the cost and complexity of production, enhance the robustness of designs, and ensure higher yields during the manufacturing phase. DfM principles are integral to modern circuit design, aiming to streamline production by standardizing components, simplifying assemblies, and optimizing design processes to increase manufacturability yields, thereby reducing costs without sacrificing performance [1].

2.3.2 Integration of Advanced Modeling Tools

Designers now routinely use sophisticated tools that allow for detailed simulations and testing of electronic designs before they reach the point of fabrication. This use of advanced CAD (computer-assisted design) and EDA (electronic design automation) tools means that potential problems can be identified and addressed earlier in the design process, reducing the risk of costly redesigns and ensuring a smoother path from conception to product [1].

2.3.3 Focus on System-Level Integration

As devices become more integrated, the emphasis shifts from individual components o system-level considerations. This means designing circuits that not only work in isolation but also function seamlessly when integrated with other system components. The move towards systems on a chip (SoC) and system in package (SiP) solutions exemplifies this trend, where multiple functions are integrated into a single module, saving space and improving performance [1].

2.3.4 Enhanced Reliability and Testing

2.3.4.1 Predictive Maintenance through Smart Design

In semiconductor circuit design, reliability is paramount. Modern designs often incorporate predictive maintenance capabilities, where circuits are not only robust but also capable of anticipating failures before they occur. This is achieved through the integration of smart sensors and diagnostic algorithms that monitor the health of the circuit, identifying potential issues. This proactive approach is crucial in industries where reliability is critical [1].

2.3.4.2 Advanced Testing Protocols

Along with design innovations, the testing protocols for semiconductor-based circuits have also evolved. Today's testing methodologies are far more comprehensive and automated compared to earlier practices. For example, the use of Automated Test Equipment (ATE) allows for rapid testing of complex ICs at various stages of the manufacturing process, ensuring that only fully functional units reach end users [1].

2.3.5 Challenges and Solutions

Despite these advancements, the increasing complexity of ICs presents new challenges such as thermal management, signal integrity, and power consumption. To address these, designers are using new materials, such as advanced silicon formulations and compound semiconductors, and innovative techniques such as 3D integration and multilayer PCBs (Printed Circuit Boards) [1].

Chapter 3

Synchronous Systems

3.1 Fundamentals of Synchronous System Design

Synchronous design has been the primary methodology since the early development of semiconductor technology in the 1960s. Typical synchronization systems, especially semi-custom synchronization application-specific integrated circuit (ASIC) operations, have combinational logic combined with banks of flip-flops that store data. This configuration ensures that the combinational block completes its operation within one clock cycle and stores the data within this cycle. It is worth mentioning that in order to calculate the actual frequency that the clock should perform on, it is taken into consideration the worst-case operating environment. The precision of the global clock in such systems is crucial [3].

3.1.1 Design Principles

The synchronous design paradigm involves tightly controlling the timing of signals. A synchronous circuit uses a clock signal to orchestrate the actions of digital circuits to ensure predictability and simplicity in handling data dependencies and timing analysis. The clock signal synchronizes the data movements across all parts of the circuit, ensuring that operations are performed in a pre-defined chronological order.

3.1.2 Timing Constraints

In synchronous systems, timing constraints are essential for ensuring that the circuit operates reliably under all conditions. These constraints include:

- **Setup Time**

  This refers to the minimum amount of time before the clock edge that the input data signals must be held stable to ensure they are correctly latched by the flip-flops. Not meeting setup times can result in setup violations in which data are not captured correctly, leading to unpredictable circuit behavior.

- **Hold Time**

  This defines the minimum time after the clock edge during which the input data must remain stable after being captured by a flip-flop. Hold time violations are less common, but can occur in high-performance designs where clock skew and data path delays cause data to change too quickly after the clock edge.

- **Clock-to-Output Delay**

  This is the time taken from the clock edge triggering a flip-flop to the time the output of the flip-flop stabilizes. This delay is critical in determining the maximum operating frequency of the synchronous circuit.

Timing analysis tools play a crucial role in the design process, employing static timing analysis (STA) to predict the worst-case timing scenarios without the need for simulation of every possible input condition. These tools help identify critical paths, the longest paths between two registers that determine the clock frequency.

3.1.3 Advantages of Synchronous Design

- **Advanced CAD ans EDA Tools**

  CAD and EDA tools are crucial in synchronous design, offering advanced features that optimize clock-driven processes. They automate verification and simulation, ensuring all parts of the circuit meet strict timing require-

ments. This reduces errors and speeds up the design process, making it more efficient and reliable.

- **Predictability and Simplified Debugging**
  Since all operations are controlled by the clock, it is easier to predict when things happen in the system, which simplifies testing and debugging.

- **Ease of Design**
  Synchronous design allows for easier scaling and integration of multiple subsystems because timing control is centralized through the clock. This modularity facilitates independent design and verification of sub-modules, which can then be integrated with reduced risk of integration issues.

3.2 Synchronous Clock Domains

Synchronous clock domains are fundamental to the design of complex digital systems, especially in large-scale integrated circuits, where different parts of the system operate at varying clock frequencies or phases. Managing multiple clock domains effectively is crucial to maintaining the integrity and performance of the system.

3.2.1 Definition and Importance

A clock domain is a region of a digital circuit where a single clock signal drives all sequential elements such as flip-flops and latches. In complex systems, multiple clock domains may exist, each synchronized by its own unique clock source. These are particularly important in systems where different modules need to operate at different frequencies or where power management strategies involve scaling the clock frequency dynamically.

3.2.2 Challenges of Multiple Clock Domains

The interaction between multiple clock domains introduces several challenges, mainly related to data transfer between domains operating at different frequencies or phases. These challenges include:

- **Clock Domain Crossing (CDC):** Signals passing between different clock domains require special handling to prevent data corruption caused by clock skew and phase variation. CDC issues are nontrivial and often a source of critical design bugs in synchronous systems.
- **Metastability:** When data cross from one clock domain to another, there is a risk that the data will not be stable during the clock edge in the receiving domain, leading to metastability. Metastability can cause unpredictable circuit behavior and system failures.
- **Jitter and Skew Management:** Different clock domains may experience varying amounts of jitter and skew. Care must be taken to ensure that these variations do not impact the reliability of the timing between interconnected domains.

3.2.3 Design Techniques for Clock Domain Management

To manage these challenges, several design techniques and methodologies are employed:

- **Synchronization Circuits:** The use of flip-flops or latches arranged in specific configurations (e.g., two-stage synchronizers) to safely pass signals between clock domains, thereby reducing the risks of metastability.
- **Asynchronous FIFOs:** These are used to handle data transfer between domains of differing frequencies, buffering data entries until they can be safely read by the receiving clock domain.
- **Clock Gating and Dynamic Frequency Scaling:** Techniques that control the activity and frequency of clock signals based on real-time operational requirements to manage power consumption and reduce cross-domain interaction complexities.
- **Static Timing Analysis for CDC:** Specialized tools and methods in static timing analysis are employed to analyze and verify the timing of signals that cross the clock domains, ensuring that all possible timing violations are identified and addressed during the design phase.

3.2.4 Best Practices

Some best practices in managing synchronous clock domains include:

- **Clear Domain Separation:** Clearly define and document the clock domains within the design specifications to simplify the management and verification processes.
- **Conservative Timing Margins:** Apply conservative timing margins for interactions in the domain of the product to account for variations in fabrication, temperature and operating conditions.
- **Regular Reviews and Testing:** Regular design reviews and rigorous testing strategies, including simulation at different levels (unit, module, system) to identify and resolve timing and synchronization problems early in the design process.

All the information in Chapter 3 is taken from [4].

Chapter 4

Asynchronous Systems

4.1 Introduction to Asynchronous Design

4.1.1 Asynchronous Design Basics

Asynchronous design, is described by the absence of a global clock. Instead, their designs rely on handshaking mechanisms and protocols to transfer data between functional blocks and also to ensure well-established communication between them. This data transfer is organized in channels that group a bundle of data wires using handshaking signals. These channels are unidirectional, meaning that bidirectional communication between blocks is able to be achieved. Asynchronous circuits do not adhere to a strict global clocking mechanism, which makes their design and operation completely different from synchronous systems. This approach has gained lately more traction due to its simplicity and due to the constantly growing complexities of synchronous designs. [3]

4.1.2 Advantages of Asynchronous Design

- **High Performance:** Asynchronous systems provide high performance by eliminating clock skew and operating on case-average performance. This improves data-dependent flows and delays because asynchronous circuits do not have global clock skew and can operate more efficiently in average scenarios than the worst-case scenarios often considered in synchronous systems. Asynchronous circuits can also automatically adapt to physical character-

istics such as the manufacturing process, temperature, and supply voltage variations, providing robust performance under changing conditions. In synchronous designs, this adaptability becomes more difficult due to the overall effect of clock variations. [3]

- **Low power consumption:** Asynchronous circuits use power more efficiently because they do not rely on the constant operation of a global clock. In a synchronous system, the global clock runs continuously, which consumes power even when parts of the circuit are idle. However, asynchronous designs reduce unnecessary data movement and reduce overall power consumption, making them especially beneficial for applications where energy efficiency is a priority.[3]

- **Reduced Electromagnetic Interference (EMI):** Unlike synchronous circuits, where nearly all energy is concentrated in narrow spectral bands around the clock frequency and its harmonics, leading to substantial electromagnetic noise, asynchronous circuits have a more distributed noise spectrum. This results in low peak noise levels and minimal interference with adjacent analog circuitry.[3]

- **Modularity:** Channel-based transmit and receive discipline in an asynchronous design promotes modularity and enables easy plug-and-play connectivity between blocks. This approach provides an immediate idea of flow control within the design and makes asynchronous circuits latency-insensitive. If a phase in the design is not ready to receive a new token, the sender will wait until the recipient is ready. This modularity reduces disruption when changing pipeline levels later in the design cycle, which is especially beneficial when incorporating long wires into the design.[3]

4.2 Asynchronous Communication Protocols

4.2.1 Bundled-data protocols

Bundled-data protocols involve data signals that use standard Boolean levels to encode information, along with separate request and acknowledge wires that are

bundled with the data signals [4].

- **4-Phase Bundled-data Protocol** [4]:
  - Consists of four communication actions:
    1. Sender issues data and sets request high.
    2. Receiver absorbs the data and sets acknowledge high.
    3. Sender takes request low, making data no longer valid.
    4. Receiver takes acknowledge low, signaling readiness for next cycle.
  - Known for its superfluous return-to-zero transitions, which can cost time and energy.
- **2-Phase Bundled-data Protocol** [4]:
  - Information on the request and acknowledge wires is encoded as signal transitions (either 0 to 1 or 1 to 0), representing a "signal event."
  - Aims to be faster than the 4-phase protocol by eliminating the return-to-zero phase.
  - Implementation complexity varies, and there's no definitive answer on which protocol (2-phase vs 4-phase) performs better overall.
- **Terminology Variations** [4]:
  - Different terms are used interchangeably:
    - "Bundled-data" might also be called "single-rail."
    - "4-phase handshaking" can be referred to as "return-to-zero (RTZ) signaling" or "level signaling."
    - "2-phase handshaking" might be known as "non-return-to-zero (NRZ) signaling" or "transition signaling."
- **Push and Pull Channels** [4]:
  - In push channels, the sender initiates data transfer.
  - In pull channels, the receiver initiates the request for data, and the roles of request and acknowledge signals are reversed.

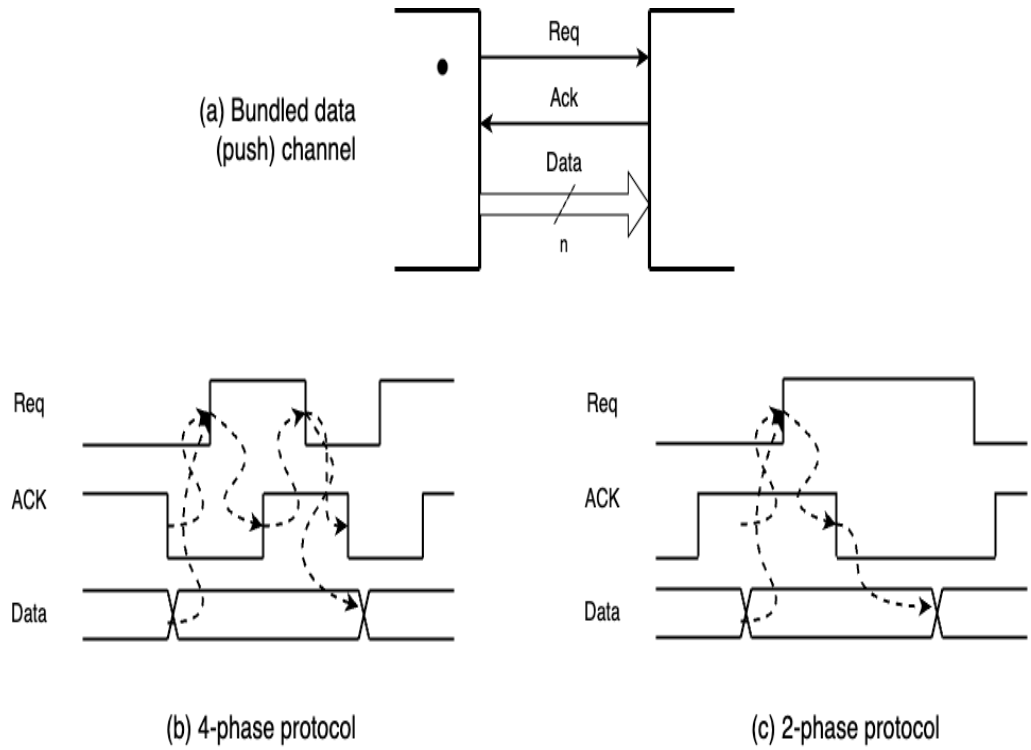


Figure 4.1 (a) A bundled-data channel. (b) A 4-phase bundled-data protocol. (c) A 2-phase bundled-data protocol. Adapted from [4]

4.2.2 The 4-phase dual-rail protocol

The 4-phase dual-rail protocol is a robust and enduring protocol to eliminate wire delays. It is considered delay-insensitive. In essence, it is a 4-phase protocol using two request wires per bit of information. In other words, each bit is represented by two request wires, one wire d.t for signaling true (or Logic 1) and another wire d.f for signaling false (or Logic 0). [4]

The schematic shown in Figure 4.2, illustrates the abstract architecture of the protocol. There is a bus that contains the two wires per bit, needed for the data transfering and the requests. There is also a single bit bus for the ACK.

In addition, Figure 4.2 shows the truth table for all the possible combinations.

Case 1: d.t = LOW, d.f = LOW
Represents "No Data"

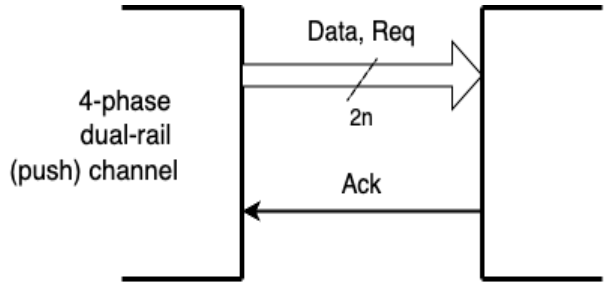
Case 2: d.t = LOW, d.f = HIGH
Represents "Valid Data - False - Logic 0"

Case 3: d.t = HIGH, d.f = LOW

Represents "Valid Data - True - Logic 1"

Case 4: d.t = HIGH, d.f = HIGH

Not Used Pair



	d.t	d.f
Empty (E)	0	0
Valid "0"	0	1
Valid "1"	1	0
Not Used	1	1

Figure 4.2 The 4-phase dual rail protocol - Protocol Schematic and Truth Table. Adapted from [4]

Figure 4.3 illustrates the the timing waveforms indicating the sequence of signals for the "Data" lines (d.t and d.f) and the "ACK" line (Acknowledge). Moreover, it illustrates the state diagram with the three possible states of the protocol.

From the figure, it is clear that between each valid bit, the protocol send an Empty bit(d.t = LOW, d.f = HIGH). This leads to 4-phase handshaking. [4]. In an abstract viewing, the handshaking consists in the following.

1. The Sender issues a valid codeword
2. The Receiver absorbs the codeword and sets acknowledge high
3. The sender responds by issuing the empty codeword
4. The receiver acknowledges this by taking acknowledge low

Adapted from [4]

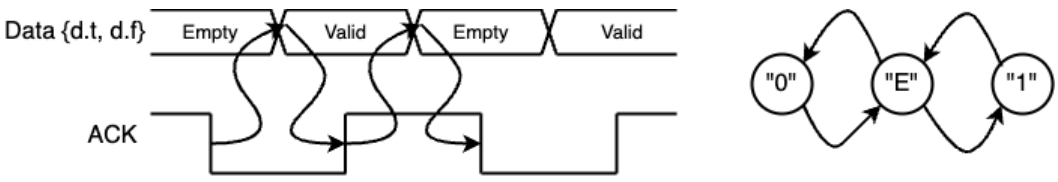


Figure 4.3 The 4-phase dual rail protocol - Time and State Diagram. Adapted from [4]

4.2.3 The 2-phase dual-rail protocol

The 2-phase dual-rail protocol uses 2 wires d.t, d.f per bit. A new codeword is received when exactly one wire in each of the N wire pairs has made a transition. The valid message is then acknowledged [4].

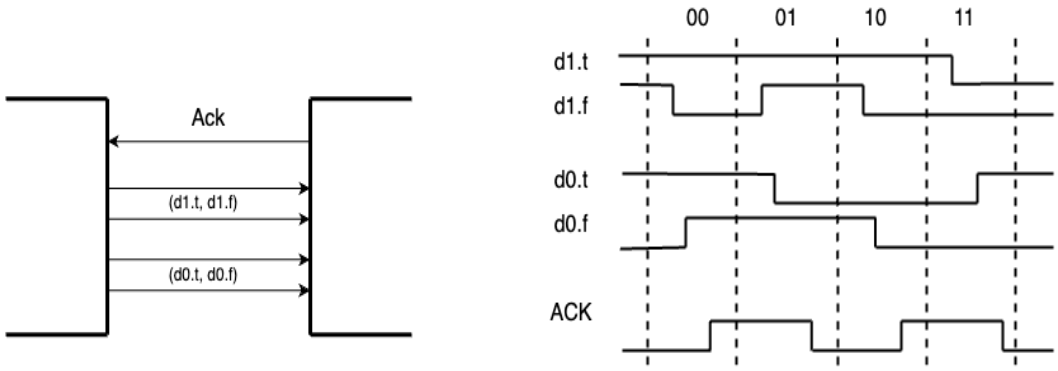


Figure 4.4 The 2-phase dual rail protocol - Handshake. Adapted from [4]

4.3 High-Level Languages and Tools in Asynchronous Circuit Design

4.3.1 Overview of Communicating Sequential Processes (CSP)

CSP is a hardware description language for asynchronous circuits that is crucial for modeling asynchronous circuits due to its focus on concurrency and inter-process communication. The language facilitates robust message passing mechanisms, essential for the effective design of modular asynchronous systems.

Key features of CSP include:

- **Concurrent processes**: Multiple processes operate simultaneously, reflecting the inherent parallelism in asynchronous systems.
- **Composition of processes**: CSP supports both sequential and concurrent compositions, allowing complex behaviors to be built from simpler ones.
- **Synchronous message passing**: Inter-process communication in CSP is explicitly managed through synchronous message exchanges over designated channels, ensuring tight synchronization between processes.

4.3.2 CSP Point-to-Point Communication Example

Figure 4.5 illustrates two processes, P1 and P2, connected by a channel named C. Process P1 sends data using the command `C!x` (where `!` indicates a send operation), which transmits the value of variable x over the channel. Process P2 receives this data using the command `C?y` (where `?` indicates a receive operation), and the received value is then stored in variable y. This synchronous transmission ensures that the data transfer is an atomic action, effectively synchronizing both processes at the point of communication.

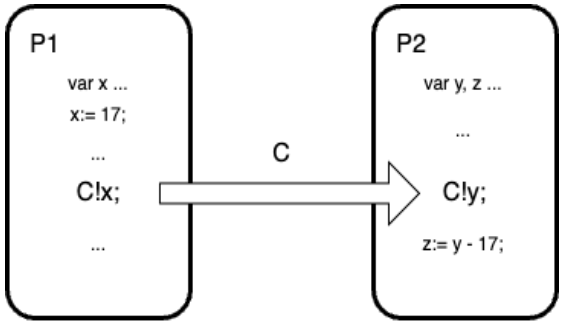


Figure 4.5 Two processes P1 and P2 in CSP connected by a channel C. Adapted from [4]

4.3.3 Influence of CSP on Other Languages

CSP's approach to managing concurrency has influenced several languages tailored for specific domains:

- **General concurrent systems**: Languages such as OCCAM and LOTOS extend CSP's concepts to broader applications.
- **Asynchronous circuit design**: Specific derivatives like Tangram and CHP (Communicating Hardware Processes) have been developed to cater directly to the needs of asynchronous circuit design, optimizing various aspects of implementation and simulation. Other notable languages include **Balsa**, which also uses CSP concepts to support the design of asynchronous circuits but with varying syntactic and functional focuses.

4.3.4 Martin's Translation Process for Asynchronous Circuit Design

Martin's translation process provides a structured methodology for transforming high-level asynchronous circuit designs into optimized hardware implementations. This method involves several detailed phases:

1. Process Decomposition
    - **Objective:** Decompose each complex process into simpler, interacting subprocesses.
2. Handshake Expansion
    - **Objective:** Translate abstract communication actions into explicit low-level signal transitions and wire implementations.
    - **Example:** The receive statement `C?y` can be expanded with:

$$\left[C_{\text{req}}\right]; y := \text{data}; C_{\text{ack}}\uparrow; \left[\neg C_{\text{req}}\right]; C_{\text{ack}}\downarrow$$

        (a) "Wait for request to go high (red)"
        (b) "Read the data (blue)"
        (c) "Drive acknowledge high (green)"
        (d) "Wait for request to go low (orange)"
        (e) "Drive acknowledge low (purple)"
    - This expansion explicitly manages signal transitions required for reliable data transfer, adhering to protocol and timing constraints.
3. Production Rule Expansion
    - **Objective:** Replace handshake expansions with production rules that define behavior of internal and output signals based on conditional actions.
    - **Example:** A rule such as `a \and b \to c` indicates that signal `c` is activated only when both `a` and `b` are true, implementing a logical AND operation.

4. Operator Reduction

   - **Objective:** Group production rules into clusters and map these to basic hardware components such as generalized C elements.
   - **Impact:** Optimizes performance and reliability by minimizing complexity.

4.3.5 Integration of VHDL in Asynchronous Design

Although traditionally used for synchronous circuit design, VHDL has been adapted to support asynchronous operations. By incorporating CSP-like constructs for message passing and process concurrency, VHDL can be extended to model and simulate asynchronous behaviors effectively.

All the information in Subsection 4.3 is taken from [4].

Chapter 5

Comparative Analysis between Synchronous and Asynchronous

5.1 Synchronous versus Asynchronous Systems

Aspect	**Synchronous Systems**	**Asynchronous Systems**
Timing	Operates with a global clock to synchronize all elements.	Operates without a global clock using handshaking protocols.
Design Complexity	Easier to design due to centralized timing control.	More complex due to decentralized timing.
Flexibility	Less flexible, as all parts are tied to the clock cycle.	More flexible, adapts to changes in workload and conditions.
Power Efficiency	Less power-efficient due to continuous clock operation.	More power-efficient, as components activate only when needed.
Reliability	Susceptible to timing issues such as jitter and skew.	More robust against timing issues, reliable under variable conditions.
Communication	Straightforward but can cause latency in complex systems.	Complex but allows for immediate communication without waiting.

Table 5.1: Comparison of Synchronous and Asynchronous Systems

Table 5.1 illustrates the fundamental distinctions between synchronous and asynchronous systems. Although synchronous systems benefit from simpler design and predictable behavior, they face limitations in efficiency and flexibility.

Asynchronous systems, conversely, offer superior power efficiency and adaptability, proving more robust under variable operational conditions. These differences highlight the importance of system selection based on specific application requirements, often leading to hybrid solutions that leverage the strengths of both to optimize performance and reliability [1][2][4].

5.2 Process and Mismatch Variations

5.2.1 Process Variations

Process variations in semiconductor manufacturing refer to deviations from intended design specifications that occur due to imperfections in the fabrication processes. These variations can significantly impact the performance, power consumption, and overall yield of semiconductor devices [5][6].

5.2.1.1 Types of Process Variations

- **Random Variations:**
  - **Description:** Random variations arise from inherent stochastic processes such as fluctuations in chemical concentrations or random dopant fluctuations in transistor channels. They are unpredictable and vary from device to device, even within a single batch.
  - **Impact:** These variations can cause significant shifts in electrical characteristics, such as threshold voltage and drive current, potentially leading to failures in meeting design specifications [6].
- **Systematic Variations:**
  - **Description:** Systematic variations are predictable deviations that occur due to non-uniformities in equipment, materials, or environmental conditions. These might include gradient effects across a wafer caused by equipment alignment issues or systematic pattern distortions during lithography [5][6].

  - **Impact:** Such variations typically manifest as spatial patterns across a wafer and can be somewhat mitigated by calibrating and adjusting manufacturing tools.

- **Environmental Variations:**

  - **Description:** Variations due to environmental factors like temperature and humidity during the fabrication process. These conditions can fluctuate over time and between different manufacturing batches [5][6].
  - **Impact:** Environmental variations can affect many steps in the semiconductor process, such as oxidation and diffusion rates, which could lead to inconsistent device performance.

5.2.1.2 Incorporating Variations into Semiconductor Designs

To mitigate the effects of process variations, semiconductor designs often incorporate several robust engineering practices.

- **Guard-Banding:**

  - **Description:** Guard-banding involves setting design specifications more stringently than the actual operational requirements. This method provides a safety margin that ensures that even if the device is subjected to the worst-case variation, it will still meet the necessary performance criteria.
  - **Application:** Typically applied in timing critical circuits, power management, and signal integrity specifications.

- **Adaptive Body Biasing:**

  - **Description:** Adaptive body biasing (ABB) is a technique used to adjust the threshold voltage of a transistor dynamically during operation based on the detected process variations.
  - **Benefits:** ABB helps in compensating for speed deficiencies or power inefficiencies that arise due to process variations. By tuning the performance of the chip in real time, it enhances both energy efficiency and performance stability.

- **Real-Time Adjustment Techniques:**
  - **Description:** Beyond ABB, other real-time adjustments might include dynamic voltage and frequency scaling (DVFS) which adjusts the power and speed according to the load requirements and operational conditions.
  - **Impact:** These adjustments help adapt to variations in the manufacturing process and environmental conditions, ensuring optimal performance under a variety of operating conditions.

5.2.2 Mismatch Variations

Mismatch variations refer to the discrepancies that occur between identically designed devices within the same integrated circuit. Unlike process variations that impact all devices globally across different wafers or batches, mismatch variations are localized differences affecting devices that are supposed to be identical [2].

5.2.2.1 Origins of Mismatch Variations

Mismatch variations arise primarily because of the inherent limitations in fabrication processes that affect the microscopic uniformity across a wafer. These can include variations in:

- **Dopant concentrations**: Slight differences in the amount and distribution of dopants in semiconductor devices can lead to variations in threshold voltage and current drive capabilities.
- **Line edge roughness**: Variations in the lithography process can lead to non-uniform gate lengths and widths.
- **Oxide thickness**: Variability in the oxide growth process can affect the capacitance and, consequently, the performance of transistors.

5.2.2.2 Impact of Mismatch Variations

The primary impact of mismatch variations is on the precision and performance of analog circuits, such as operational amplifiers and current mirrors, where closely

matched transistor characteristics are crucial. Mismatch can lead to offsets, gain errors, and non-ideal behaviors that degrade the overall performance of the circuit.

5.2.2.3 Modeling and Mitigation

To account for mismatch variations, designers use statistical models provided by foundries, which describe the expected variability in device characteristics. Monte Carlo simulations are particularly useful for assessing the impact of these variations. By simulating circuits with different instances of the same device model that have slightly varied parameters, designers can predict the variability in circuit performance and identify the worst-case scenarios.

In practical applications, strategies such as common-centroid layout and interdigitated designs are employed to minimize mismatch by ensuring that all critical components of a circuit are equally affected by any gradients in process parameters across the wafer.

5.2.3 Monte Carlo Simulation

Monte Carlo simulation is a computational technique that is used to understand the variability in system performance and predict future outcomes by simulating the process with input from random variables. This method is essential in fields where exact solutions are impossible or impractical to determine due to the complexity or the presence of uncertainty. In addition, Monte Carlo simulations play a crucial role in understanding the impact of Process Variations [7][8].

5.2.3.1 Definition and Purpose

Monte Carlo simulation allows the evaluation of complex systems or processes by using random sampling to obtain numerical results. The term originated from the Monte Carlo Casino in Monaco due to its association with randomness and chance. In engineering, it is used to see the effects of input variations on outputs, which are often too complicated to model analytically [7][8].

5.2.3.2 Implementation in Cadence Virtuoso

The implementation of a Monte Carlo simulation in Cadence Virtuoso involves several steps to set up and analyze a circuit design under the influence of component variability and manufacturing inconsistencies [7][8].

- **Setup the Environment**

  Initiate the Cadence Spectre Circuit Simulator within the Analog Design Environment (ADE). Prepare the schematic model of the circuit, including all necessary components and their nominal values.

- **Component Variation Modeling**

  Replace standard models of the components with their statistical counterparts that include variability and mismatch data. This may involve editing component properties to reflect manufacturing tolerances or specific variation parameters.

- **Simulation Configuration**

  Configure the Monte Carlo analysis settings in the ADE. Specify the number of runs, which determines how many times the circuit will be simulated with randomly varied parameters. Typically, at least 1000 runs are performed to ensure statistical significance of the results.

- **Running the Simulation**

  Execute the simulation to observe how the circuit's performance metrics such as gain, frequency response, and output levels vary with the input changes. Results are typically analyzed using histograms or probability distributions to understand the likelihood of different performance outcomes.

- **Analysis and Optimization**

  Based on the simulation results, identify components or parameters with the most significant impact on circuit performance and consider redesign or adjustment to meet design specifications.

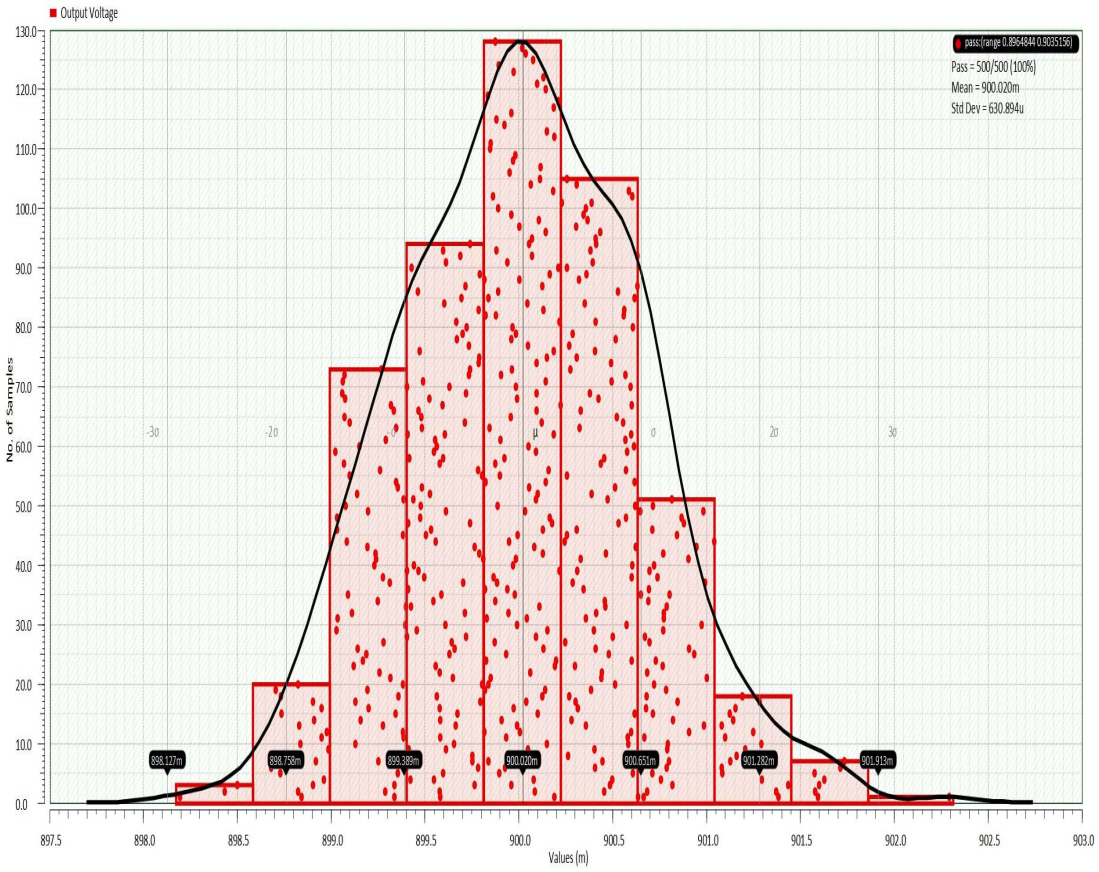


Figure 5.1 Monte Carlo Simulation Example [7]

5.2.3.3 Practical Example

Figure 5.1 illustrates the results of a Monte Carlo simulation, presented as a histogram. The histogram reflects the distribution of a specific parameter – in this case, frequency – across multiple simulation iterations considering the inherent process variability.

- The **X-axis** represents discrete bins of frequency values, with each bin corresponding to a range within the parameter's potential outcomes.
- The **Y-axis** quantifies the number of occurrences, indicating how many simulation runs resulted in a parameter value within each bin's interval.
- Different **colors or patterns** in the histogram bars may correspond to various categories within the simulation, such as different batches or components.

- **Overlayed curves** suggest theoretical distributions of the data. These fits, such as Gaussian distributions, help to understand the underlying statistical nature of the parameter variation.
- The **statistical information** provided includes:
  - **Number**: The total count of simulation runs included in the histogram.
  - **Mean**: The average value of the parameter, indicative of the central tendency across all simulations.
  - **Standard Deviation (Std Dev)**: This indicates the spread of values around the mean, signifying the degree of variation or uncertainty.
- The **legend** explains the dataset each color or pattern represents, facilitating differentiation between multiple variables or conditions. [7]

The interpretation of this histogram focuses on understanding the central tendency through the mean, variability through the standard deviation, and the overall shape of the distribution. The insights gleaned from this analysis guide decisions to improve the design's robustness and ensure it meets performance criteria despite manufacturing variations.

5.2.4 Corner Analysis

Corner Analysis is a deterministic simulation methodology used to assess the performance of an integrated circuit (IC) under extreme variations in the manufacturing process. Unlike the probabilistic approach of Monte Carlo simulations, Corner Analysis ensures that a circuit meets its design specifications even under the worst-case scenarios of manufacturing tolerances.

Manufacturers define 'process corners' to represent the extremes of these variations, which include:

- **FF (Fast-Fast):** Where both NMOS and PMOS transistors are at their fastest operational speeds.
- **SS (Slow-Slow):** Where both NMOS and PMOS transistors are at their slowest.

- **FS (Fast-Slow):** Where NMOS transistors are fast and PMOS transistors are slow.
- **SF (Slow-Fast):** Where NMOS transistors are slow and PMOS transistors are fast. [6]

During Corner Analysis, circuits are simulated with transistors set to these extreme values to evaluate if the design operates correctly under all conditions. If a circuit fails at any corner, it may require redesigning to ensure functionality across the full spectrum of manufacturing variances.

The figure below shows a two-dimensional grid mapping the performance spectrum for NMOS and PMOS transistors, with the process corners marked FF, SS, FS, and SF. The center point, TT (Typical-Typical), represents the standard operating condition.

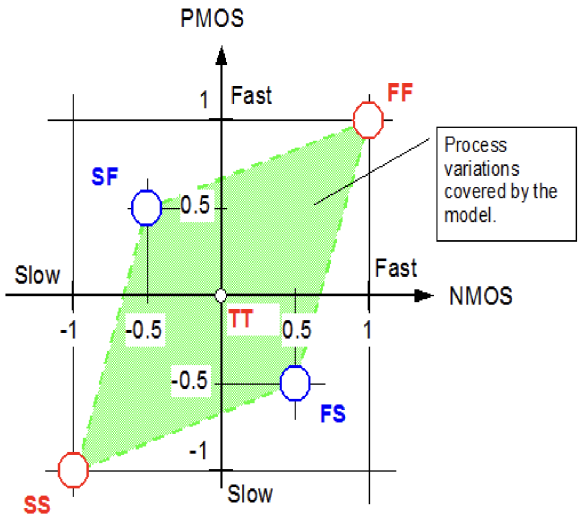


Figure 5.2 Corner Analysis Example [9]

This visual aid is crucial in identifying potential issues that may arise due to process variations and in designing circuits that are robust against the unpredictability of semiconductor manufacturing processes. Engineers use this figure as a guide to ensure high percentages of chip functionality even when subjected to the outer limits of process variations.

5.3 Clock Domain Transitions

5.3.1 Challenges in Clock Domain Transitions

In digital design, transitions between different clock domains represent a significant challenge, particularly when signals must be passed between synchronous circuits operating at different frequencies, or between synchronous and asynchronous circuits. These transitions require detailing attention to ensure data integrity and system reliability. Timely and accurate data transfer across these domains is critical for the seamless functionality of complex systems.

5.3.2 Synchronous to Asynchronous Transitions

Synchronous circuits operate under a common clock signal, ensuring coordinated state changes. However, interfacing these circuits with asynchronous systems—which do not share a common clock signal—can lead to timing issues. Asynchronous signals can change state at any time relative to the synchronous clock cycle, risking the introduction of errors if the transition coincides with the clock edge used to latch the data.

Example of a Synchronous-to-Asynchronous Transition

A synchronous system needs to send a control signal to an asynchronous subsystem. The control signal generated on the rising edge of the synchronous clock, must be accurately captured by the asynchronous system, which operates independently of the synchronous clock.

To manage this, a dual flip-flop synchronizer is often used:

1. First Flip-Flop: Captures the control signal on the rising edge of its clock cycle. This stage is susceptible to metastability if the control signal changes state close to the clock edge.

2. Second Flip-Flop: Acts to resolve any metastability from the first stage, ensuring that the output signal remains stable long enough to be safely used by the asynchronous system.

This method helps to mitigate the risks associated with clock domain transitions, although it introduces a delay equivalent to two clock cycles of the synchronous system.

5.3.3 Asynchronous to Synchronous Transitions

When signals from an asynchronous source are integrated into a synchronous circuit, careful synchronization is required to align the signal with the receiving system's clock domain. This is crucial to prevent the synchronous system from capturing incorrect data due to misalignment of the signal relative to the clock edge.

Example of an Asynchronous-to-Synchronous Transition

An asynchronous sensor emitting a status signal that must be read by a synchronous digital processor. Since the sensor updates its output based on external events rather than a clock, the signal might change state at any time relative to the processor's clock cycle.

To safely interface this signal into the synchronous domain, the following approach is adopted:

1. First Flip-Flop: The signal is fed into this flip-flop, which captures the signal state at the next clock edge, regardless of when the change occurs.

2. Second Flip-Flop: Further stabilization of the output from the first flip-flop provides a buffer period that ensures that the captured signal has resolved any potential metastability before being used in the synchronous logic circuit.

This two-stage buffering ensures that the processor only uses valid, stable inputs, thus maintaining data integrity across the clock domains.

All the information in Section 5.3 is taken from [10].

Part II

Part II: Comparative Technical Exploration and Hardware Selection

Chapter 6

Selecting Hardware for Communication

As previously stated, Part II consists of the necessary comprehensive study needed to design and implement the Thesis's communication system, as shown in Figure 1.1. It starts with Chapter 6, the hardware whose main purpose is to act as a bridge between the user's application software and the asynchronous chip. The main reason that external hardware has to exist between the two systems is because it gives the ability for easier configuration. In addition, with the use of an embedded system as an intermediary, there is easier control of the voltage levels that are being sent and the shifting of them.

For the selection of the embedded device, the study focuses specifically on two main categories, the FPGAs and the MCUs.

6.1 Field Programmable Gate Arrays (FPGAs)

FPGAs are a class of digital integrated circuits known for their versatility and reconfigurability. Their fundamental core consists of programmable blocks of logic along with configurable interconnects between these blocks. The most important characteristic that the FPGAs hold in their possession is that they can be programmed multiple times post-manufacturing, allowing designers to design and implement an enormous variety of functions ranging from simple logic operations to complex system designs. Such devices belong to the In-System Programmable

(ISP) category. [11]

Serving as a midpoint in the logic device spectrum, FPGAs provide a compromise between the simplicity of programmable logic devices and the tailored complexity of application-specific integrated circuits (ASICs). They are particularly favored in scenarios where design flexibility and the ability to quickly alter hardware functionalities are valued. This makes them suitable for a wide array of design applications, enabling large-scale enterprises and small teams to explore hardware innovations efficiently. [11]

6.1.1 Advantages of Using FPGAs

FPGAs are distinguished by several key characteristics and advantages that make them excel in their field:

1. **Reprogrammability**

   FPGAs can be programmed and reprogrammed multiple times, which distinguishes them from one-time-programmable devices. This reprogrammability offers flexibility for design engineers to adapt the FPGA to changing standards or to update its functionalities without the need for new hardware. For instance, some FPGAs can be reprogrammed repeatedly, while others may only be programmed once. [11]

2. **Parallel Processing Capabilities**

   The architecture of FPGAs allows them to be more focused in parallel processing. This means they can handle multiple processing tasks simultaneously, as opposed to serial-processing CPUs or other general-purposed CPUs that execute one operation at a time. This characteristic makes them particularly suitable for tasks that can be broken down into parallel processes, thus providing significant speed advantages in computational tasks such as signal processing, video encoding, and data analysis. [11]

3. **Customizability**

   FPGAs offer a high degree of customizability because their blocks of logic and interconnects are configurable. This allows engineers to tailor the hardware for specific applications, optimizing for speed and power consumption.

Unlike ASICs, which are custom-designed for a specific function and are not modifiable post-production, FPGAs can be customized even after deployment, enabling a more flexible approach to hardware design and optimizations. [11]

4. **Rapid Prototyping**

   With the ability to be reprogrammable and reusable, FPGAs are the perfect choice for prototyping purposes – especially for ASIC validation purposes. Before the tape out of an ASIC, it is important to determine whether the ASIC design is functioning and successfully achieves the purpose it has been designed for. ASICs are difficult and incredibly expensive to manufacture which means that if a chip needs modification, a considerable amount of time and money have to be invested in order to redesign the ASIC. With FPGAs, the reprogrammability feature allows us to perform test runs by manipulating the programming and determining the ideal configuration on one chip only. Once the prototyping is completed and the best solution is determined, the FPGA can be easily turned into a permanent ASIC. [12]

5. **Cost-Efficiency**

   Since FPGAs can be reprogrammed again and again, they prove to be extremely cost-effective in the long run, even though they may pose higher unit costs. They rid the designer of the need to cover recurring bug-related costs that he may get stuck paying if he were to opt for an ASIC. [12]

6.1.2 FPGA Implementation Strategies

In terms of designing logic and 'configuring' an FPGA, designers use Hardware Description Languages. Hardware Description Languages (HDLs) are high-level machine languages whose purpose is to describe the whole architecture of the designed digital circuit and implement it on the actual FPGA. At first, HDLs were used to simulate hardware circuits on a general-purpose processor with great success. Later on, people began to synthesize hardware, automatically generating the logic configuration for the specified device from the hardware description language. [13]

The most popular HDLs are:

1. **VHDL**

   VHDL, standing for VHSIC Hardware Description Language, is primarily used for the description and simulation of digital circuits. It allows for both structural and behavioral modeling, offering a comprehensive way to describe hardware functions and simulate their behavior before physical implementation. VHDL emphasizes a methodology that can include test benches for verifying the logic of the described hardware, supporting a wide range of digital circuit designs from single gates to complex programmable logic. [13]

2. **Verilog**

   Verilog is a hardware description language used for modeling electronic systems. It is less verbose than VHDL and is considered easier for those familiar with imperative programming languages like C/C++ and Java. Verilog enables both gate-level and behavioral modeling, making it versatile for designing a broad spectrum of digital circuits. It supports simulation and synthesis, allowing designers to test and implement their models on physical devices efficiently. The syntax and structure of Verilog are straightforward, promoting rapid learning and application in various digital design projects. [13]

3. **High-Level Synthesis (HLS)**

   High-Level Synthesis (HLS) is a transformative approach that enables designers to express their designs in high-level programming languages, such as C, C++, or SystemC. This methodology automatically translates these high-level descriptions into Register-Transfer Level (RTL) code, significantly improving design productivity and allowing for more rapid exploration of design alternatives. Through HLS, designers focus on algorithmic and architectural decisions rather than low-level implementation details, facilitating a more efficient design process that optimizes for both performance and area. [14]

6.2 Microcontrollers (MCUs)

Microcontrollers are integral components of embedded systems, designed to perform dedicated functions or tasks within larger systems, or even independently. They combine a processor with memory and input/output peripherals, all on a single chip, enabling them to execute a specific set of instructions efficiently and autonomously. This makes them ideal for applications where compactness, simplicity, and low power consumption are crucial, such as in household appliances, automotive electronics, medical devices, and a variety of other uses. [15]

The primary purpose of microcontrollers is to control the operation of electronic systems by responding to sensor inputs, processing data, and driving outputs accordingly. This allows them to interact with the physical world in a meaningful way, automating processes and enhancing the functionality of devices. For instance, in a smart thermostat, a microcontroller adjusts heating and cooling systems based on temperature reading, user inputs, and predefined schedules, demonstrating their role in bridging the digital and physical domains to create responsive and intelligent devices. [15]

In addition to their role in controlling devices, microcontrollers are also key to making technology more accessible and affordable. By integrating several functions and capabilities onto a single chip, they reduce the size and cost of electronics, making advanced technology available to a wider audience. This enables more people to innovate and create solutions for everyday problems. As microcontrollers become even more powerful and cost-effective, their impact on society is expected to grow, further transforming how we interact with the world around us. [15]

6.2.1 Advantages of Using MCUs

MCUs, just like FPGAs that have been discussed in the previous section, are distinguished by several key characteristics and advantages that make them excel in their field: [16]

1. **Easy Integration:** MCUs consolidate numerous functions into a single chip, significantly reducing size and complexity.

2. **Cost-effectiveness:** Their integrated nature reduces the overall cost of electronic systems.

3. **Power Efficiency:** Optimized for low-power operation, they are ideal for battery-powered devices and others.

4. **Ease of Programming:** A variety of development tools and high-level programming languages make programming and updates straightforward.

5. **Versatile Interfacing:** They support a wide array of sensors, accessories, and devices, ensuring wide applicability.

6.2.2 MCU Implementation Strategies

Programming microcontrollers can be approached through several methodologies, each with its distinct advantages: [17]

1. **Assembly Language**

   Assembly acts as a bridge between machine code and high-level languages, utilizing mnemonic abbreviations to represent binary instructions a microcontroller executes. This language allows precise control over the functionality of the microcontroller but is machine-dependent, so it is specific to each type of microcontroller, limiting its portability. It is favored for its stability, but requires a detailed understanding of the architecture of the microcontroller.

2. **C/C++**

   These languages offer a compromise between the direct hardware manipulation possible with the assembly language and the ease of use of high-level languages. While they are not as memory efficient or as fast as assembly language, they are easier to write and maintain. C/C++ programs must be converted into machine code through a compiler before the microcontroller can execute them. This process involves reorganizing the instructions from the high-level language into a form that the microcontroller can understand, making these languages more versatile and portable across different systems.

3. **Python**

   Python, through implementations like MicroPython and CircuitPython, brings the ease of a popular high-level programming language to microcontrollers. It emphasizes readability and maintainability but with the cost of trading some performance for accessibility. Python is particularly suited for educational purposes and IoT projects, where rapid development and prototyping are more valued than raw performance.

4. **Visual Programming Languages**

   Examples like Scratch for Arduino and Blockly introduce microcontroller programming through a graphical interface, where users connect blocks to form logic. This method is particularly effective for educational purposes, allowing beginners, and especially young learners, to grasp programming concepts without the complexity of traditional coding syntax.

6.3 Decision Criteria and Selection

6.3.1 System's Specific Requirements

Following the concise review of the two most popular embedded system categories, this section outlines the specific system requirements that the bridge hardware needs to meet. These are essential for the communication to be established correctly and to function efficiently in its intended environment. Covers both the core requirements and any additional considerations that ensure the system can adapt to various demands.

Embedded System Requirements

1. **Serial Communication:** The system shall establish serial communication with a Personal Computer.

2. **Data Reception:** The system shall be able to receive bitstreams and/or commands via serial communication.

3. **Data Storage:** The system shall store the received data in its non-volatile storage for later use.

4. **General-Purpose Input/Output (GPIO):** The system shall provide GPIO pins to establish a connection with the asynchronous chip.

5. **Power Management:** The GPIO pins must output 1.8V as this is the maximum that is accepted by the asynchronous circuit. If an embedded system with this output voltage is not available, a level shifter shall be added to the system to solve the problem.

Additional (Optional) Considerations

- **Wireless Communication:** The system shall be capable of establishing wireless communication with a Personal Computer.
- **Security:** Techniques for improving security shall be added.
- **Documentation and Community Support:** The embedded system may have comprehensive documentation and be supported by a large community.

Note: Detailed information on communication types and protocols will be provided in Chapter 7.

6.3.2 Hardware Findings Analysis

In view of the system requirements that have been stated above, extensive study has been conducted to identify suitable hardware devices that meet these specifications. The following tables present a curated selection of hardware options that have been considered. Table 6.1 shows all the chosen FPGAs and Table 6.2 all the Microcontrollers. Each table offers a comparison mainly of the chip model, the features on board and the price, ensuring that the final hardware choices align closely with the demands and requirements of the system. In addition, both tables include the manufacturer of the boards and a reference link to their official page.

This careful selection process ensures that the chosen hardware integrates seamlessly and also upholds the standards of reliability and efficiency.

Board Name/Model	Manufacturer	Chip Model	On-board Features	Price	Image Reference	Link
Boolean Board	Real Digital	Xilinx Spartan-7 XC7S50-CSGA324	USB, Bluetooth LE, 8-digit seven-segment display, 16 slide switches, 4 pushbuttons, 2 Pmod+ connectors	Academic: $87 / Commercial: $105		Link
Blackboard	Real Digital	Xilinx XC7007S ZYNQ	USB, WiFi, Bluetooth, 16 MB QSPI ROM, SD Card slot, 12 slide switches, 6 pushbuttons, LEDs, 3 Pmod connectors	Academic: $139 / Commercial: $174		Link
Basys 3	Digilent	Xilinx Artix-7 FPGA: XC7A35T-1CPG236C	USB-UART, VGA output, USB HID Host, 16 LEDs, 5 pushbuttons, 7-segment display, 4 Pmod ports	$165		Link
Cora Z7	Digilent	Zynq-7000	Arduino headers, Ethernet, ADC, USB, pmod peripheral connectors, microSD, small form factor with mounting holes	$149		Link
ZedBoard	Digilent	Zynq-7000 AP SoC	USB-UART, Ethernet, VGA, microSD, 7-segment display, 5 Pmod ports, IO output 3.3V	$589		Link

Table 6.1: FPGA Development Boards

Board Name/Model	Manufacturer	Chip Model	On-board Features	Price	Image Reference	Link
Raspberry Pi 5	Raspberry Pi	Quad Arm Cortex-A76	Dual HDMI, Raspberry Pi ISP, PCIe connector, Wi-Fi, Bluetooth 5.0, Gigabit Ethernet, USB 3.0 & 2.0, PoE support, GPIO	€72.90		Link
Arduino UNO R4	Arduino	Renesas RA4M1 (Arm® Cortex®- M4)	14 Digital I/O Pins (of which 6 provide PWM output), 6 Analog Input Pins, 16 MHz Quartz Crystal, UART, USB connection, a power jack, an ICSP header, and a reset button.	€25.00		Link
CSR101x Series	Qualcomm	16MHz 16MIPS XAP	UART, Bluetooth 4.1, GPIO, Operating Voltage 1.8 to 4.3V	$133		Link
ESP32-C6-DevKitM-1	Espressif	ESP32-C6	Wi-Fi, dual-mode Bluetooth, 32 I/O pins, UART, built-in antenna switches, balun, power-amplifier, power management modules	€7 - €15		Link

Table 6.2: Microcontrollers - Development Boards

6.3.3 Final Selection and Justification

After thorough analysis and comparison of the FPGA and microcontroller options listed in the provided tables, the final selection process focused on balancing the specific system requirements with the additional optional considerations for the bridge hardware. Taking into account the core requirements of serial communication, data reception and storage, GPIO capability, and power management, alongside optional features such as wireless communication and security enhancements, the following selections were made:

Final Selection and Justification

The selections are justified as follows:

FPGA Selection: Blackboard by Real Digital

The Blackboard, featuring the Xilinx XC7007S ZYNQ chip, emerges as the superior FPGA choice due to its extensive on-board features, including USB, WiFi, Bluetooth, 16 MB QSPI ROM, an SD Card slot, slide switches, pushbuttons, LEDs, and 3 Pmod connectors. At an attractive price of $139 for academic purposes and $174 for commercial use, it presents a versatile platform that excels in serial communication, data reception, and storage, fulfilling the core requirements. The integration of WiFi and Bluetooth not only caters to the optional wireless communication consideration but also enhances system flexibility and the potential for secure connections. This makes the Blackboard an ideal candidate.

Microcontroller Selection: ESP32-C6-DevKitM-1 by Espressif

Espressif ESP32-C6-DevKitM-1 has been selected as the optimal microcontroller, thanks to its ESP32-C6 chip, which facilitates Wi-Fi and dual mode Bluetooth capabilities, key for wireless communication. Offered at an affordable price range of €7 to €15, it stands out as a cost-effective, yet potent option to meet our system's serial communication, data reception, and storage needs. The board's 32 I/O pins and UART support address the requirement for versatile GPIO configurations and efficient data interaction with PCs. Moreover, its power management

capabilities and a flexible operating voltage range are in line with our system's need for efficient power usage and adaptability. When combined with the previously chosen Blackboard FPGA, this microcontroller underscores a robust and flexible foundation for developing a communication system that is both reliable and efficient within its intended setting.

In conclusion, the choices of the Blackboard FPGA and the ESP32-C6-DevKitM-1 microcontroller are justified by their comprehensive feature sets, which not only meet the essential system requirements but also provide additional capabilities that significantly bolster the bridge hardware's functionality, flexibility, and security.

Chapter 7

Reliable Communication Establishment

This chapter investigates the establishment of a reliable communication link between User Software and bridge hardware, a connection that is essential for the precise control and coordination of embedded systems. It goes beyond viewing this link as merely a conduit for data; it is a critical component for the synchronization and functionality of the system. In addition, it further explains the connection protocol that will be established between the Bridge Hardware and the asynchronous chip. The text focuses attention on the carefully selected protocols that enable this robust connection, highlighting their significant role in ensuring a reliable and orderly flow of information between the software and the hardware components.

7.1 Communication Protocols between User Software - Bridge Hardware

This section enumerates the communication protocols vital for ensuring reliable data exchange between User Software and bridge hardware. Each protocol is chosen for its proven robustness and ability to maintain consistent system performance.

7.1.1 Serial Communication

Serial communication refers to the process of sending data one bit at a time, sequentially, over a communication channel or a computer bus. This method of data transmission has several advantages that make it popular for a wide range of applications, especially in embedded systems and for communication between devices. [18]

Advantages of Serial Communication:

1. Efficiency for Long-Distance Communication
2. Minimal Wiring
3. Broad Application Support
4. Wide Support Across Operating Systems and Microcontrollers

UART Protocol:

The Universal Asynchronous Receiver/Transmitter (UART) is important in asynchronous serial communication by converting between parallel and serial data, managing low-level details of serial communications. Embedded in PCs, microcontrollers, and external UART chips, it supports various word formats, including the commonly used 8-N-1 format, which consists of one start bit, eight data bits, and one stop bit. UARTs facilitate serial communication in devices lacking built-in serial ports through USB converters, supporting interfaces like RS-232 and RS-485. Beyond hardware, UART functionality extends to software, with programming libraries providing simplified access for microcontroller-based projects. This versatility makes UARTs fundamental to the implementation of serial communication across a wide array of devices and applications, highlighting their indispensable role in modern electronics. [18]

I2C Protocol:

The Inter-Integrated Circuit (I2C) protocol is a synchronous serial communication interface used primarily for short-distance communication within a device. Char-

acterized by its simplicity and efficiency, I2C facilitates communication between multiple slave devices and a single or multiple master devices over a two-wire interface, comprising Serial Data (SDA) and Serial Clock (SCL) lines. This design allows a straightforward and cost-effective way to build complex communication networks between various components within an electronic system. In particular, I2C supports device addressing, allowing multiple devices to share the same bus without conflict, and offers different speed modes ranging from standard to high-speed operation. The ability of the protocol to perform bus arbitration and collision detection ensures reliable data transfer, making I2C a widely adopted solution for embedded systems and microcontroller-based projects for effective component-to-component communication. [18]

SPI Protocol:

The Serial Peripheral Interface (SPI) is a synchronous serial communication protocol known for its high-speed data transfer capabilities, which makes it a popular choice to connect microcontrollers to peripheral devices such as sensors, SD cards and LCD displays. Distinctive for its master-slave architecture, SPI operates over a four-wire interface, consisting of a Serial Clock (SCK), Master Out Slave In (MOSI), Master In Slave Out (MISO), and a separate Slave Select (SS) for each device. This setup enables full-duplex communication, allowing data to be simultaneously transmitted and received, thus facilitating efficient data exchange. SPI's simplicity in connectivity, coupled with its absence of a strict protocol for data exchange, grants designers the flexibility to implement custom communication protocols suited to their specific needs. Moreover, the ability to run at very high clock speeds allows for rapid data transactions, making SPI an ideal protocol for applications that require fast and reliable communication between a microcontroller and peripheral devices. [18]

7.1.2 Bluetooth Low Energy (BLE)

Bluetooth technology, particularly Bluetooth Low Energy (BLE), is a revolutionary wireless communication standard designed for significantly reduced power consumption and cost while maintaining communication capabilities over short dis-

tances. BLE, unlike classic Bluetooth, is optimized for small, low-power devices, enabling a wide range of applications, from fitness trackers to home automation devices. Key aspects of BLE include its ability to operate on tiny amounts of power. This efficiency is achieved through advanced techniques like adaptive frequency hopping for robustness in congested environments, low-energy physical layers for reduced power consumption, and simplified protocols that minimize data transfer and operational complexity. Furthermore, BLE supports an expansive ecosystem, providing a universal platform for a myriad of connected applications. This scalability, along with its low cost and energy requirements, positions BLE as a cornerstone technology in the expanding universe of the Internet of Things (IoT). [19]

7.1.3 WiFi

WiFi technology allows devices like smartphones and computers to connect to the internet wirelessly, without needing cables. It works by using radio signals to transmit data over short distances. This makes it possible for people to access the Internet from anywhere within the range of a WiFi network, be it at home, at work, or in public places like cafes and libraries. WiFi is convenient because it supports fast internet speeds and can connect multiple devices at the same time. As technology improves, newer versions of WiFi offer even faster speeds and better connections, making it easier and more reliable for everyone to use the Internet. [20]

7.1.4 Selected Communication Method

After careful consideration of the various communication protocols presented, the decision has been made to primarily utilize Serial Communication with a focus on the UART protocol for the bridge hardware interface. This choice is driven by UART's simplicity and efficiency, particularly in scenarios requiring minimal wiring and lower complexity. The UART protocol's widespread support across operating systems and microcontrollers further enhances its suitability for our application, ensuring reliable and straightforward communication between the User Software and bridge hardware. Its ability to facilitate serial communication

through USB converters expands its compatibility, making it an optimal solution for ensuring robust and efficient data exchange while maintaining system performance and reliability.

7.2 Communication Protocol between Bridge Hardware - Asynchronous Chip

The communication protocol that will be used to transfer the bitstreams from the Bridge Hardware to the asynchronous chip is the 4-phase dual-rail protocol. This protocol was discussed in detail in Chapter 4, Section 4.2.

7.3 Data Packet Architecture Overview

This section focuses on the format of the protocol's packet needed in order for the Asynchronous Chip to get the appropriate data from the Bridge Hardware and therefore from the user itself. The Figure 7.1 illustrates in an abstract view the way a data packet should be constructed for it to be consistent with the format that the asynchronous chip accepts. A more detailed description follows the figure below.

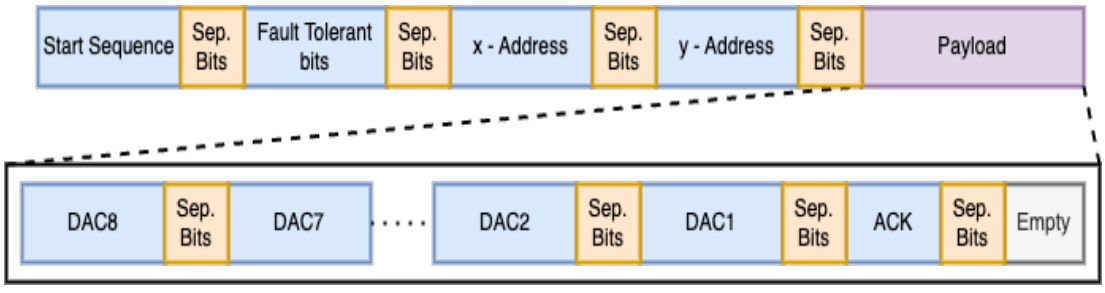


Figure 7.1 Data Packet Architecture. Adapted from [21].

- Header Section
  - Start Sequence (11)

    The Start Sequence is always initialised to the binary number$(10101010101)_2$.

This sequence is used to ensure that any packet starts with this and there is no chance of detecting it anywhere else.

  - Fault-Tolerant bits (2)
    Exploited by the gateway to force lock-on to one of the outputs. [21]
  - x-Address Bits (9)
    This block contains the Addresse's X coordinates in the network grid.
  - y-Address Bits (9)
    This block contains the Addresse's Y coordinates in the network grid.

- Separation bits (2)
  Null bits, always initialized as 00_2, are used only to separate each block from the previous and the following.

- Payload Section
  - dac8 - dac3 (8)
  - dac2 - dac1 (9)
  - ACK (3)
  - Empty (2)

  Note: The numbers in the parenthesis next to each block indicate the number of bits of the block.

A typical packet example based on the above format will be as follows:

10101010101 00 000000000 00 000000000 00 dac8 00 dac7 00 dac6 00 dac5 00 dac4 00 dac3 00 dac2 00 dac1 00 000 00 00

Chapter 8

The Engineering of the Software

8.1 Overview

This chapter delves into the systematic process of the User's software development. It begins by providing an overview of various software development models and methodologies. As the chapter progresses, it addresses the critical stages of defining and analyzing software requirements. These stages are pivotal for grasping the intended functionality and objectives of the software, ensuring it meets the users' needs. Following these, it shifts towards software design. Additionally, the chapter explores the evaluation of different frameworks, detailing the selection process for choosing the most suitable framework for the project.

8.2 Software Development Models

In the field of software development, there is a huge variety of ways to define, visualise, design and represent the whole process of developing a software. Those are called "Software Process Models" [22].

Each model has the following important characteristics, despite any extra variation that it might have.

- **Software Specification**: The description of the functionalities and the operations of the software
- **Software Design and Implementation**: The production of the software
- **Software Validation**: The testing phase
- **Software Evolution**: The ability to change and adapt to the user's needs

The following list states, at an abstraction level, the most popular and useful ones that cover most of the cases and approaches.

- **The Waterfall Model**: Linear, with sequential phases where each phase must be completed before the next begins.
- **Incremental Development**: Develops the system in versions, gradually adding functionality.
- **Reuse-oriented Software Engineering**: Focuses on integrating existing components into new systems.
- **Prototyping**: Builds an initial version to refine requirements through user feedback.
- **Incremental Delivery**: Delivers system parts to customers for feedback and use.
- **Boehm's Spiral Model**: Iterative, risk-driven, and combines sequential and iterative elements.

All models and characteristics are based on the descriptions found in "Software Engineering, 9th Edition" by Ian Sommerville [22].

8.2.1 Selected Software Development Model for User Software

Focusing exclusively on the above fundamental models, the methology that is going to be used for the development of the User Software is the traditional Waterfall Model. The justifications behind this decision are:

1. **The linear philosophy**: With this approach, each steps must be completed in order to proceed to the next one, so it reduces the chances of ambiguities or errors.

2. **Simplicity**: The software will be a simple one without the need of complicated algorithm implementations.

3. **Parallel System Development**: Due to the nature of the Thesis' System, there are more that one designs needed to be implemented in a small period of time. The Waterfall Model is the most suitable when having to deal with multiple implementations, especially when hardware requirements are involved.

4. **Thesis is more "Communication Focused"**: The main topic is the communication between the digital circuits, so there is no need of a complicated user software.

8.2.1.1 Modified Waterfall Model Steps

Figure 8.1 presents all the steps that are going to be followed in the selected Waterfall Model.

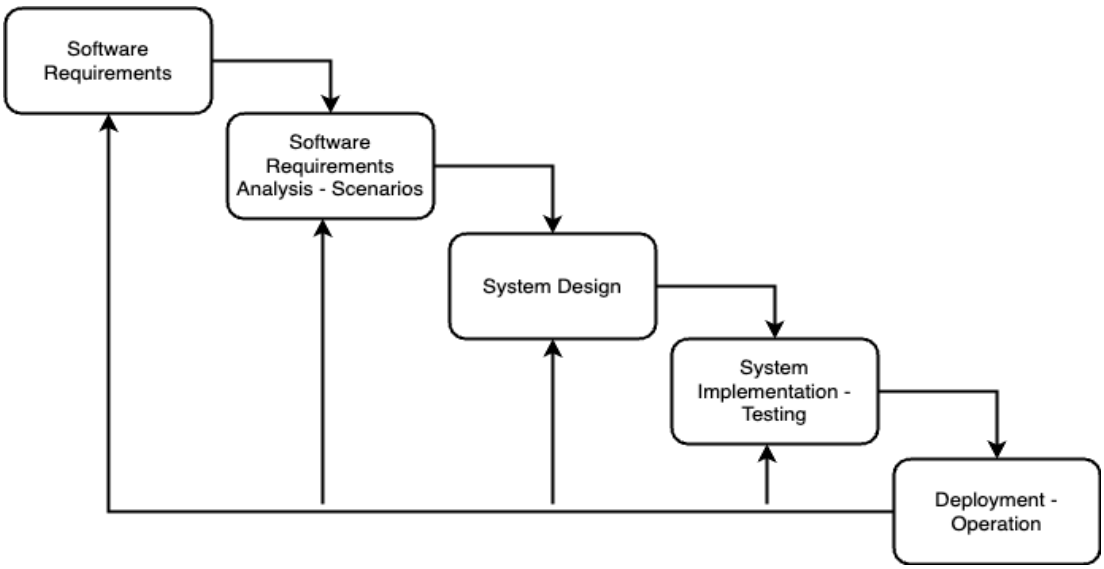


Figure 8.1 Modified Waterfall Model. Adapted from [22].

1. **Software Requirements**: Establish what the application should do and define the resources needed to build the project.

2. **Software Requirements Analysis - Scenarios**: Analyze the requirements through detailed use cases or scenarios to understand functional behavior.

3. **System Design**: Design the overall system architecture and define the software components, interfaces, and data models.

4. **System Implementation - Testing**: Develop the software according to the design documents and conduct systematic testing to ensure it meets the specified requirements.

5. **Deployment - Operation**: Export the final product for use and perform ongoing changes or updates.

8.3 Software Requirements

This section consists the beginning of the Software Development analysis and enumerates all the Functional and Non - functional requirements that the User Software must have.

8.3.1 Purpose of the System

The primary purpose of the system is to give the ability to the user to send data packets through a user-friendly environment to the bridge hardware and therefore to the Asynchronous chip.

8.3.2 Existing System

There is no other existing system that is able to form the specific data packet format that the Asynchronous chip accepts and also have a user friendly environment.

8.3.3 Functional Requirements

1. **Add Input Data**: The user have to fill all the input fields with the desired data to form the packet that it will be transferred to the Asynchronous chip.
   Involving Steps:
   (a) Insert the Start Sequence bits.
   (b) Insert the Fault Tolerant bits.
   (c) Insert the x-Address bits.
   (d) Insert the y-Address bits.
   (e) Insert the bits for dac8 - dac3 to the appropriate fields.
   (f) Insert the bits for dac2 - dac1 o the appropriate fields.
   (g) Insert the ACK bits.
   Notes:
   i. The type that all the fields can accept is Binary Integers.
   ii. All the fields, except the fields for the dac, will be pre-filled with 0s. Despite this, the user will have the ability to change them.

2. **Start Sending Bits Function**: The user can click a button and the data from the input field will be start transferring to the Bridge Hardware.
   Note: All the input fields must be filled in order for the operation to start. In case a field is empty, a message will be prompted to the user.

3. **Structure the Data Packet**: The first and most important thing that needs to be executed as soon as the user clicks the "Start Sending Bits" button is the correct formation of the data packet format, just like the one that discussed in Chapter 7.

4. **Serial Connection Establishment**: The software must be able to detect the Bridge Hardware - which is connected with a serial cable to the personal computer - and establish the the serial connection channel for the data to be sent.

8.3.4 Non - functional Requirements

1. **User-Friendly Operational Environment**: A software that the user experience is as important as the functionality. The user must be able to navigate and operate the program seamlessly without any optical flaws or errors.

2. **Terminal Integration**: A window that includes a terminal session. This can be useful for monitoring the operations that are being executed on the Bridge Hardware and also monitor the later process which is the sending of the bits to the Asynchronous chip with the 4-phase dual-rail protocol.

3. **Log Activity - History**: After the completion of a sending, the software can present into a window a history of the data being sent so far.

4. **Cross-Platform Compatibility**: The software can be ported to multiple platforms to ensure universal compatibility. Can be ported to the following systems: Windows, macOS, iOS, Android.

8.3.5 Use Case Diagram

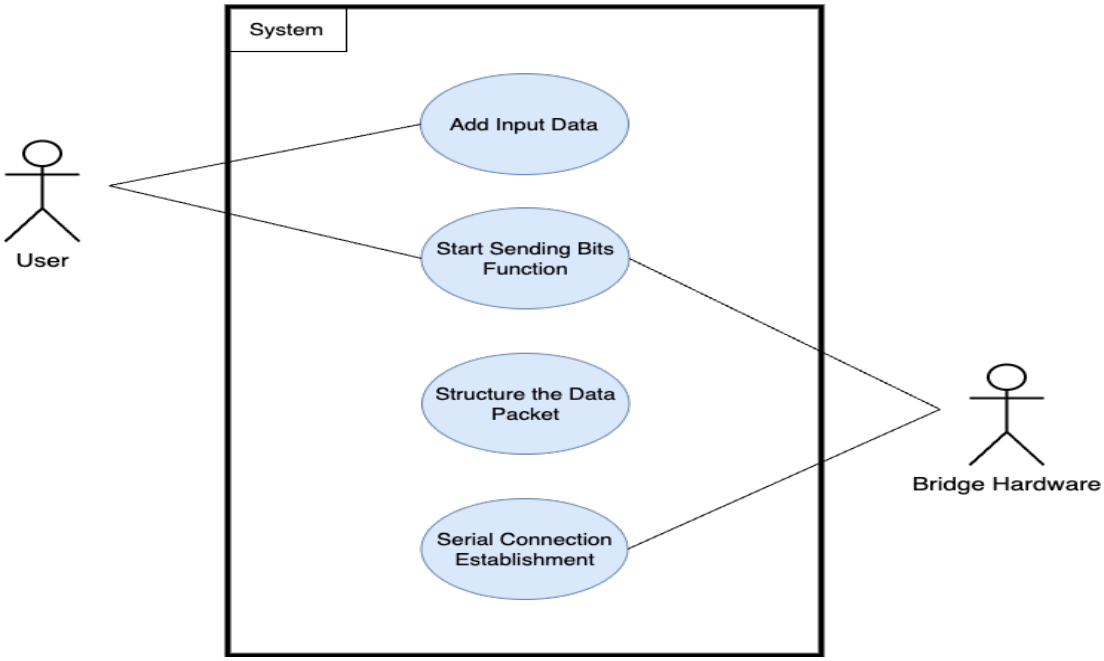


Figure 8.2 Software's Use Case Diagram

8.3.6 Software Wireframe

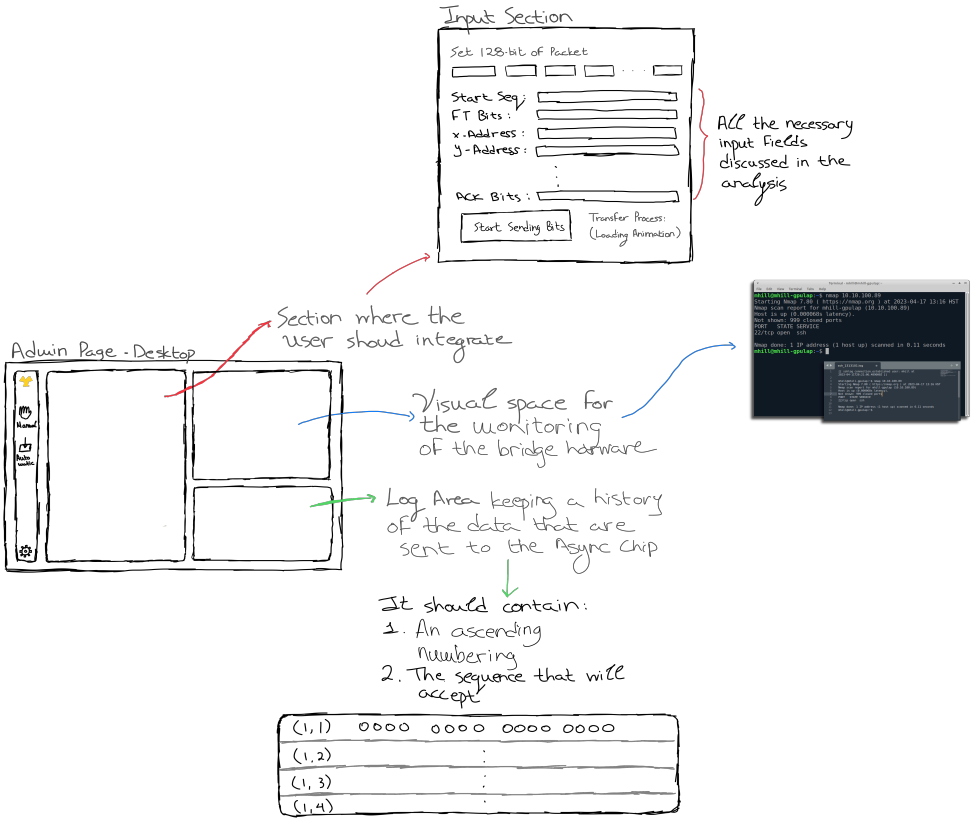


Figure 8.3 User Software Wireframe

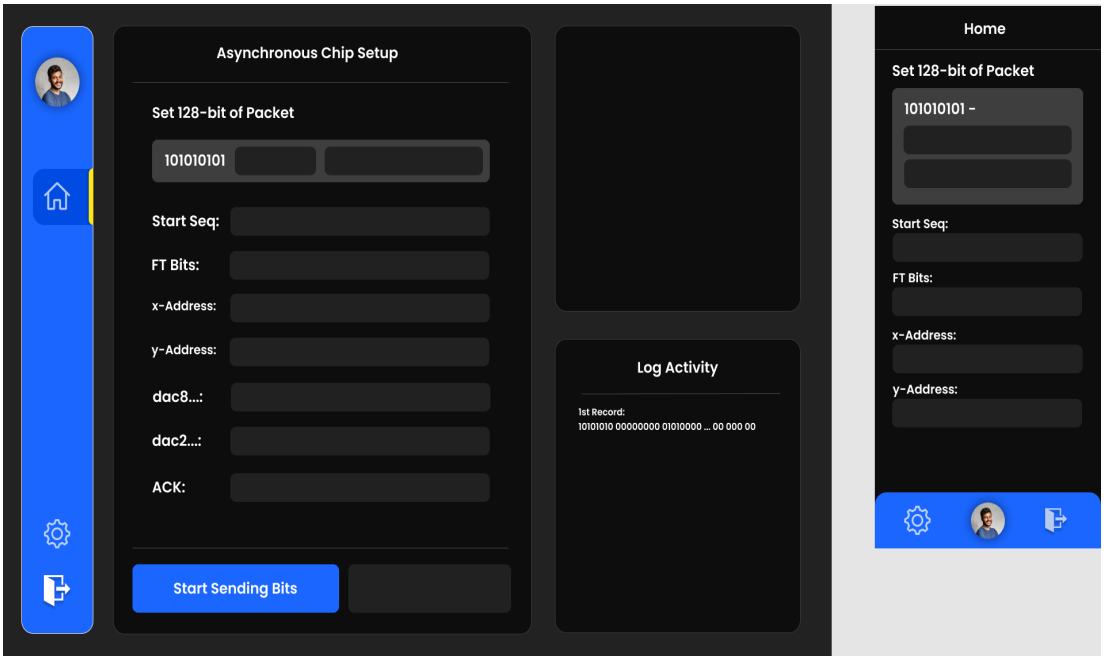


Figure 8.4 User Software Mockup Wireframe - Designed in Figma

8.4 Software Requirements Analysis

This section covers the analysis of the system requirements stated in Section 8.3. The analysis includes some of the possible scenarios - Normal or Exception - that might happen based on the functional requirements. In addition, a class diagram is provided to present the object - oriented structure of the back-end of the software.

8.4.1 Operational Scenarios

8.4.1.1 Requirement 1: Add Input Data

Normal Scenario:
The user wants to send a packet to the asynchronous chip. He starts by filling in the input fields with the correct values in binary format.
Exception Scenario:
The user wants to send a packet to the asynchronous chip. He starts by filling in the input fields, but instead of using binary format, he tries to add letters of integers in decimal format. In this case, the input will not be able to accept incompatible formats and a message will be prompt to the user.

8.4.1.2 Requirement 2: Start Sending Bits

Normal Scenario
As soon as the user finishes filling in **all** the input fields, the user clicks the Start Sending Bits button and the process of sending the bits starts.
Exception Scenario
The user clicks the Start Sending Bits button, but there is one or more fields that are empty. In this case, the process is unable to start and a message will be prompt to the user.

8.4.1.3 Requirement 3: Serial Connection Establishment

Normal Scenario
The bridge hardware is connected to the personal computer with a serial cable. The software recognises it and a connection is established.
Exception Scenario

The bridge hardware is not connected to the personal computer with a serial cable. The software is unable to find the hardware to establish a connection, and a message will be prompt to the user.

8.4.2 Class Diagram

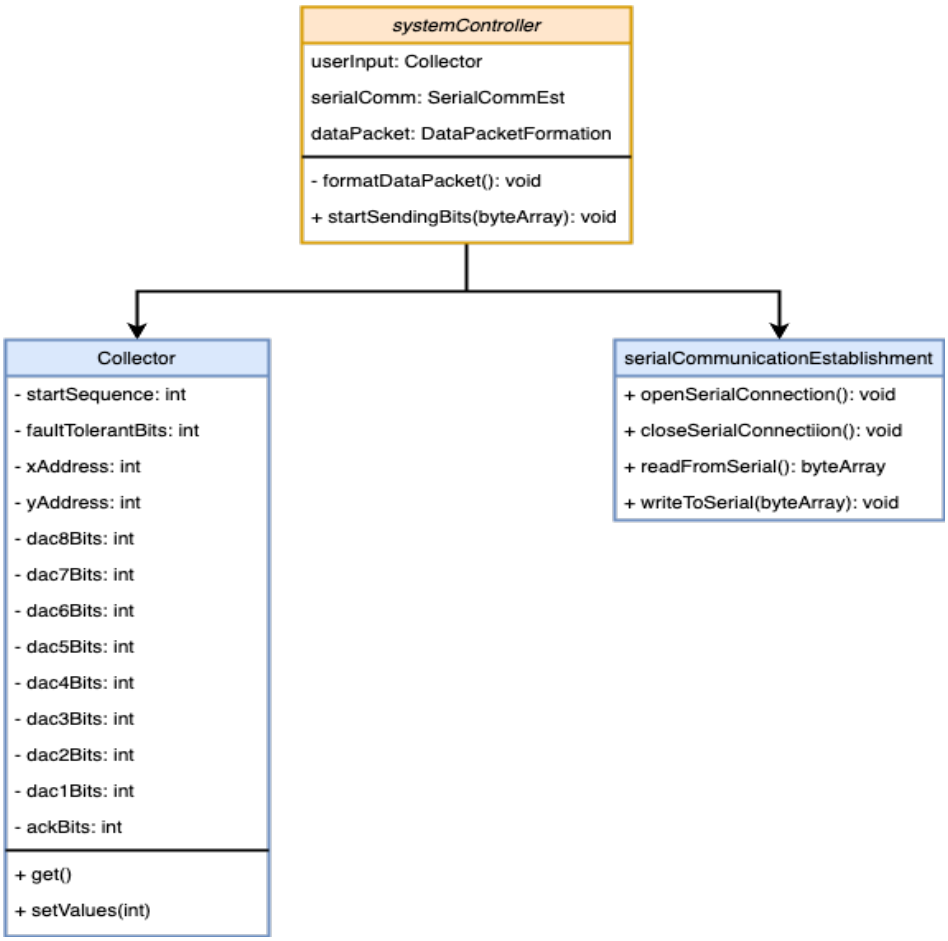


Figure 8.5 User Software Class Diagram

8.5 Implementation Frameworks

There are numerous frameworks that offer graphical user development. Selecting the right framework for app development is crucial for achieving optimal performance, maintainability, and optionally the cross platform compatibility that stated as a non-functional requirement in the Section 8.3. Table 8.1 presents a comprehensive comparison between the latest and most popular all-in-one frameworks.

Feature	.NET MAUI	Kivy	Qt
Programming Language	C#	Python	C++
Cross-Platform	Yes, including Android, iOS, macOS, Windows	Yes, including Android, iOS, Linux, Windows, macOS	Yes, across numerous platforms including embedded devices
UI Design	XAML-based declarative UI	Custom widgets and OpenGL	QML and QtWidgets for UI design
Performance	High, with native performance on each platform	Depends on Python performance	High, with emphasis on runtime performance
Community and Support	Extensive support from Microsoft and community	Limited compared to others, somewhat less corporate backing	Strong commercial and community support, extensive documentation
Use Case	Modern applications, focus on mobile and desktop integration	Applications requiring rapid development	High-performance applications, suitable for embedded systems
Development Experience	Streamlined with Visual Studio integration	Flexible and dynamic	Comprehensive IDE support, including Qt Creator for integrated development
Ecosystem Integration	Seamless integration with .NET libraries, and Microsoft services	Supports numerous third-party Python libraries	Extensive libraries and modules for a wide range of functionalities, with C++
Mobile Development	First-class support, with native controls and look-and-feel	Mobile-friendly through build tools	Strong support for mobile development, especially with Qt Quick(Dynamic UIs)
Desktop Development	Robust desktop app development	Capable of desktop application development, though may require more work for native look	Powerful tools for desktop development, with a long history of creating complex desktop applications
Embedded Systems	Emerging support for embedded systems	Limited direct support	Exceptionally strong in embedded systems
Documentation Link	Link	Link	Link

Table 8.1: Comprehensive Comparison of the Implementation Frameworks

8.6 Selected Framework

Upon evaluating the frameworks in Table 8.1, Qt has been selected for the implementation of the User Software. Qt's adept use of C++ offers fine-grained control over system operations, making it highly reliable for tasks that involve direct hardware interaction. Its expansive cross-platform capabilities ensure the User Software remains adaptable to various environments, an asset for compatibility with embedded systems. The mature Qt ecosystem, coupled with strong community and commercial support, equips developers with a robust foundation for creating versatile and efficient applications.

Part III

Part III: Practical Implementation, Evaluation and Synthesis

Chapter 9

Implementation and Testing

Part III consists of the whole phase of implementing and extracting results from the Communication System of the thesis.

This section explains in detail the whole process of designing and implementing the communication system. In addition, it includes various testing methods to verify the integrity of the system.

9.1 Bridge Hardware Implementation

9.1.1 Selected Device for Communication

As discussed in Chapter 6, during the process of selecting the most suitable device for the communication needs of the system, two of the most prevalent devices available on the market were evaluated. After a thorough comparison and consideration of the specific requirements for the project, it was decided to proceed with the ESP32-C6-DevKitM-1 by Espressif. This decision was influenced not only by its compatibility with the system design but also by the extensive documentation available and the familiarity with the C programming language, which is used primarily in the project.

9.1.2 System Design

Figure 9.1 illustrates abstractly how the system of the bridge hardware should be structured in order to be able to execute all the desired functionalities.

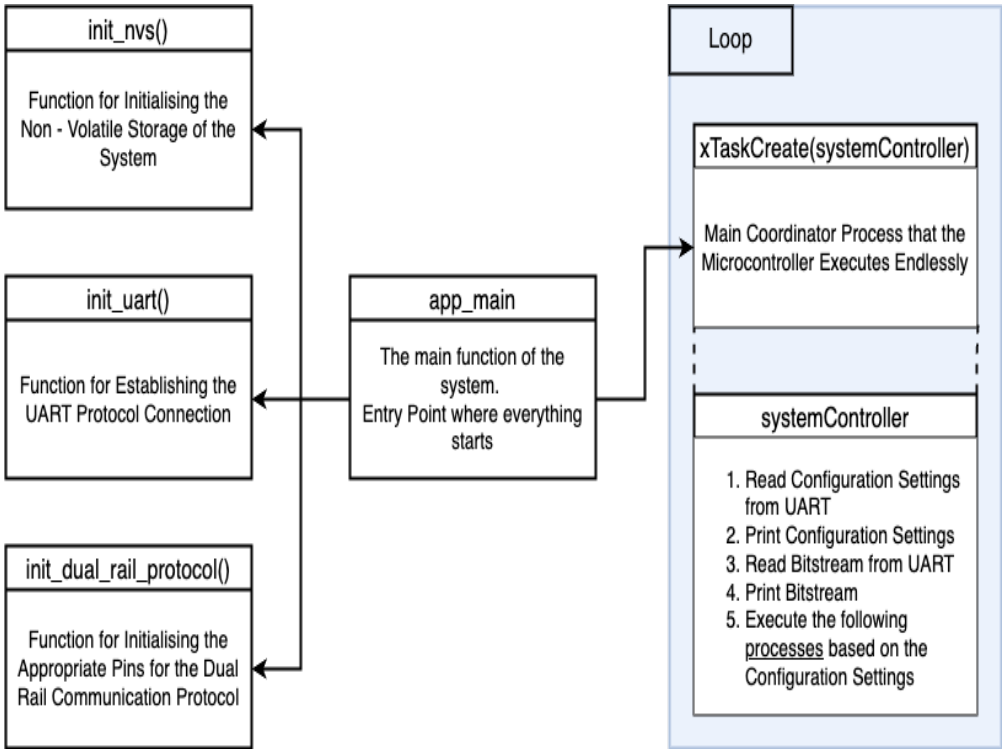


Figure 9.1 Abstract System Design Bridge Hardware

There is the main function called "app_main()" where it is the primary coordinator of the entire system. Within main, there will be the initialisation of the three main functionalities of the system, the Non-Volatile Storage, the UART Serial Communication and finally the initialisation of the communication protocol. After the initialisation, an endless loop is followed where the system is able to receive data from the UART and process them or send them to the Asynchronous chip, based on the configuration settings that the user will give from the User Software.

Figure 9.2 shows a flow chart showing how the program will continue based on the user configuration settings.

9.1.3 Selected Implementation Framework

Espressif gives multiple options in terms the development on its systems. Those options are:

- ESP-IDF: This is the company's official development framework and is primarily built on C.

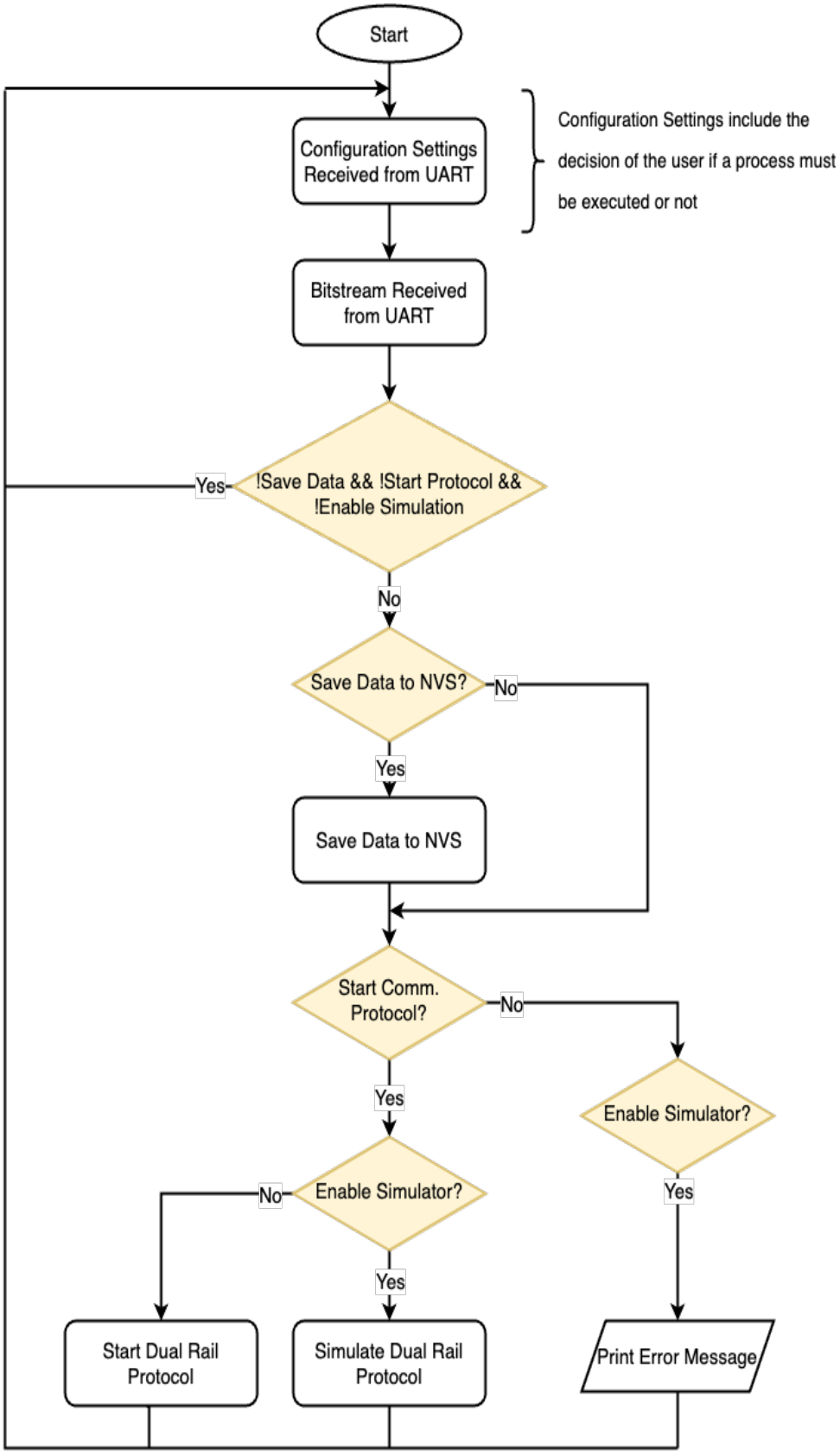


Figure 9.2 Flowchart that shows the flow of the executions processed after the bitstream reception.

- Arduino framework.
- MicroPython: An optimized version of Python able to run on microcontrollers.

The selected framework for the implementation of the system is the ESP-IDF. The main reason for this selection is the familiarity with the C programming language. In addition, the ESP32-C6-DevKitM-1 is one of the latest models, and the company has not yet added support of the other two frameworks.

9.1.4 Coding

9.1.4.1 Part I: Environment Setup

Despite the fact that ESP-IDF can run as a standalone framework using Visual Studio Code, it can also run within a larger ecosystem called PlatformIO which is the tool that is used in this case. PlatformIO is an open-source ecosystem that offers a user-friendly and extensible integrated development environment with a set of professional development instruments [23]. Basically, it enhances the existing framework with powerful debugging and monitoring features.

The chosen integrated development environment is the CLion by JetBrains. PlatformIO is installed on CLion as an extra add-on.

9.1.4.2 Part II: Project Creation

New Project → PlatformIO Add-on → Set Location → Select MCU

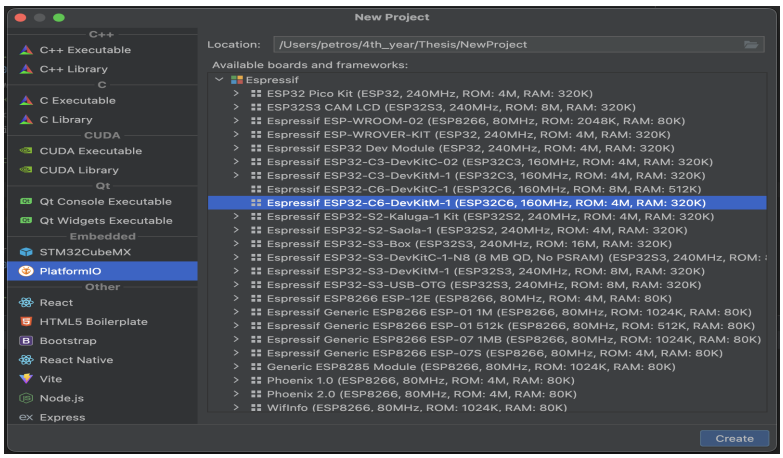


Figure 9.3 Create Project Process.

9.1.4.3 Part III: Main Function - app_main()

```
// Main application - Entry Point
void app_main(void) {

    init_nvs(); // Initialise Non-Volatile Storage(NVS)
    init_uart(); // Initialize UART Protocol
    init_dual_rail_protocol(); // Initialise Async
       Communication Protocol

    // Check for and print any previously stored data
    read_and_print_data_from_nvs();

    // Start the whole Process
    xTaskCreate(system_controller, "system_controller",
       4096, NULL, 10, NULL);
}
```

Listing 9.1: Main application entry point

Explanation of the xTaskCreate():

The `xTaskCreate()`, is part of the FreeRTOS, which is a real-time operating system kernel for embedded devices. `xTaskCreate()` is used to create a new task, able to be executed endlessly [24].

Function Prototype

```
BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,
                       const char *const pcName,
                       const uint16_t usStackDepth,
                       void *const pvParameters,
                       UBaseType_t uxPriority,
                       TaskHandle_t *const pvCreatedTask
                          );
```

Arguments Explained

- **TaskFunction_t pvTaskCode**: A pointer to the function that implements the task. This case: system_controller
- **const char *const pcName**: A descriptive name for the task for debugging purposes.
- **const uint16_t usStackDepth**: This specifies the size of the task stack in words, not bytes.
- **void *const pvParameters**: Allows to pass a single value or pointer to the task being created. Useful for passing configuration information to the task.
- **UBaseType_t uxPriority**: Tasks are scheduled according to their priority, with higher numbers representing higher priorities.
- **TaskHandle_t *const pvCreatedTask**: Used to pass back a handle by which the created task can be referenced.

Return Value:

`xTaskCreate()` returns a `BaseType_t`, which is typically used for error checking. A value of `pdPASS` indicates success, while any other value indicates that the task was not created.

All the information for `xTaskCreate()` adapted from [24]

9.1.4.4 Part IV: Initialisation Functions

Initialise NVS:

```
void init_nvs(void) {
    esp_err_t ret = nvs_flash_init();
}
```

Listing 9.2: Initialize Non-Volatile Storage (NVS)

Initialize UART Serial Communication:

```
1  #define BUF_SIZE (1024)
2  void init_uart(void) {
3      const uart_config_t uart_config = {
4              .baud_rate = 115200,
5              .data_bits = UART_DATA_8_BITS,
6              .parity = UART_PARITY_DISABLE,
7              .stop_bits = UART_STOP_BITS_1,
8              .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
9      };
10     uart_driver_install(UART_NUM_0, BUF_SIZE * 2, 0, 0, NULL
           , 0);
11     uart_param_config(UART_NUM_0, &uart_config);
12 }
```

Listing 9.3: Initialize UART

Code Explanation

Line 3: Declares a constant variable `uart_config` of type `uart_config_t`. This structure is used to configure the UART parameters.

Line 4: Sets the baud rate to 115200 bits per second.

Line 5: Configures the number of data bits in each character to 8.

Line 6: Disables parity checking.

Line 7: Sets the number of stop bits to 1 bit.

Line 8: Disables hardware flow control.

Line 10: Installs the UART driver into the application. UART_NUM_0 specifies using the first UART port (UART0).

Line 11: Applies the UART configuration (uart_config) to the UART controller specified by UART_NUM_0 (UART0). This command activates the configuration settings made earlier.

Initialize Dual Rail Protocol:

```
#define IN_F_0_PIN GPIO_NUM_7
#define IN_T_1_PIN GPIO_NUM_14
#define IN_A_ACK_PIN GPIO_NUM_6
#define OUT_F_0_PIN GPIO_NUM_3
#define OUT_T_1_PIN GPIO_NUM_4
#define OUT_A_ACK_PIN GPIO_NUM_5

void init_dual_rail_protocol() {
    gpio_set_direction(OUT_T_1_PIN, GPIO_MODE_INPUT);
    gpio_set_direction(OUT_F_0_PIN, GPIO_MODE_INPUT);
    gpio_set_direction(OUT_A_ACK_PIN, GPIO_MODE_OUTPUT);

    gpio_set_direction(IN_T_1_PIN, GPIO_MODE_OUTPUT);
    gpio_set_direction(IN_F_0_PIN, GPIO_MODE_OUTPUT);
    gpio_set_direction(IN_A_ACK_PIN, GPIO_MODE_INPUT);
}
```

Listing 9.4: Initialize Dual-Rail Protocol

This function initialises the GPIO Pins that are used in the process of sending and receiving data with the asynchronous chip or when simulating the communication protocol.

The pins are defined as constants based on the pin layout diagram of the MCU and the `gpio_set_direction` is used to set the behaviour of those pins, either accepting or sending.

9.1.4.5 Part V: System Controller Function

```
// Task to handle UART communication
void system_controller(void *pvParameters) {
    while (1) {
        // Read three boolean values from UART
        uint8_t boolValues[3];
        int bytesRead = uart_read_bytes(UART_NUM_0,
```

```
            boolValues, 3, pdMS_TO_TICKS(100));
7
8           // Convert bytes to boolean values
9           bool saveData = boolValues[0] != 0;
10          bool startProtocol = boolValues[1] != 0;
11          bool enableSimulation = boolValues[2] != 0;
12
13          // Read data from UART
14          int len = uart_read_bytes(UART_NUM_0, data, BUF_SIZE
                , pdMS_TO_TICKS(100));
15          if (len > 0) {
16          // Print the received string
17          // Processes after the received data from the UART
                based on the boolean values received
18      }
19  }
```

Listing 9.5: Pseudocode of the system_controller

Code Explanation

Line 3: while(1) ensures the endless execution of the loop

Line 6: Reads the boolean function from UART.

The `uart_read_bytes` is the main function for receiving data from UART. Its arguments are: 1. From which UART should accept data, 2. In which variable to save, 3. The amount of data will accept, 4. the time space that it has to accept those data each time.

Line 9-11: Converts the received data to boolean variables.

Line 14: Reads the bitstream from the UART

From this point and on: Prints the received bitstream and follows with the rest of the process.

9.1.4.6 Part VI: Save Data to NVS Function

```
1 void save_data_to_NVS(nvs_handle_t handle, const char *key,
    const void *data, size_t len) {
2     esp_err_t err = nvs_set_blob(handle, key, data, len);
3 }
```

Listing 9.6: Save data to NVS Function

The `nvs_set_blob(handle, key, data, len)` call attempts to store the blob of data into the NVS.

Arguments Explained

- `nvs_handle_t handle`: An open reference to a specific NVS namespace. Namespaces help organize data in NVS.
- `const char *key`: A pointer to a null-terminated string that serves as the key under which the data will be stored in NVS. Keys are unique identifiers within a namespace for accessing data values.
- `const void *data`: A pointer to the data to be stored. Being of type `void*`, it allows any type of data (e.g., integers, floats, structures) to be passed to the function.
- `size_t len`: Represents the length of the data in bytes.

9.1.4.7 Part VII: The Dual Rail Protocol Implementation Function

```
1 void send_data_dual_rail_protocol(char* bitString, size_t
    size) {
2     power_up_async_chip(); // Powers up the asynchronous
        chip
3
4     clock_gettime(CLOCK_MONOTONIC, &start); // Start timing
        the transmission
5
6     // Loop through each bit in the string
```

```
7       // Start from the last character in the bitstream ( LSB
            ) and move towards the first (MSB)
8       for (int i = size - 1; i >= 0; i--) {
9           // Check if the current character represents a '1' or
                a '0'
10          if (bitString[i] == '1') {
11              // Set IN_T = 1 and IN_F = 0
12          }
13          else if (bitString[i] == '0') {
14              // Set IN_T = 0 and IN_F = 1
15          }
16
17          // Wait for ACK Signal to go High
18          if (xSemaphoreTake(ackSemaphore, pdMS_TO_TICKS(
                TIMEOUT_MICROSECONDS / 1000)) == pdTRUE) {
19              // ACK high received
20          } else {
21              // Skip waiting for ACK low if timeout occurred
22          }
23
24          // Reset the GPIO pins to low after receiving high
                ACK
25          gpio_set_level(IN_T_1_PIN, 0);
26          gpio_set_level(IN_F_0_PIN, 0);
27
28          // Wait for ACK to go Low before sending the next
                bit
29          while (gpio_get_level(IN_A_ACK_PIN) == 1) {
30              vTaskDelay(pdMS_TO_TICKS(10));
31          }
32      }
33
34      // Measure the elapsed time for the transmission
35      clock_gettime(CLOCK_MONOTONIC, &end);
```

```
36      elapsed_time = (end.tv_sec - start.tv_sec) * 1000.0 + (
            end.tv_nsec - start.tv_nsec) / 1000000.0;
37  }
```

Listing 9.7: Pseudocode of the Dual Rail Protocol Function

Event Driven Approach for ACK

After setting the pin levels based on the bit and sending them to the asynchronous chip, the asynchronous must set the ACK High (IN_A) to indicate that it received the bit. On the other side, the MCU must stay in a wait state to receive this ACK High. Firstly, a busy wait loop had implemented that the program stayed in a loop until it receives the change. Such an implementation is very CPU heavy, leading to MCU crushes.

Instead, an event driven approach was implemented with an interrupt routine for waiting to prevent such crashes. The solution involves semaphores [25].

The semaphores are crucial for synchronizing the interrupt service routine (ISR) with a task managing data transmission, ensuring reliable and orderly communication. This synchronization prevents data collision and ensures that each piece of data is properly acknowledged before proceeding, maintaining the integrity and reliability of the communication protocol.

Visual Representation of the Solution

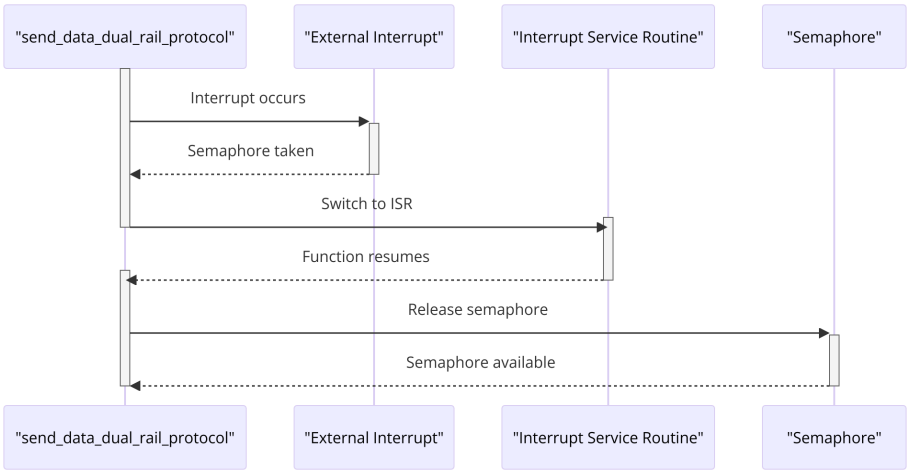


Figure 9.4 Visualisation of the Event Driven Approach

Code Implementation

Semaphore Declaration

```
SemaphoreHandle_t ackSemaphore;
```

Semaphore Creation

```
ackSemaphore = xSemaphoreCreateBinary();
if (ackSemaphore == NULL) {
    // Failed to Create Semaphore
    abort();
}
```

`xSemaphoreCreateBinary()` creates a binary semaphore. The binary semaphore can only take values 0 or 1, ideal for signaling events like an interrupt. The semaphore is initially 'taken' or 'not available'. If creation fails, it logs an error and aborts execution.

Interrupt Service Routine (ISR) for waiting

```
static void IRAM_ATTR ack_isr_handler(void* arg) {
    BaseType_t higherPriorityTaskWoken = pdFALSE;
    int currentLevel = gpio_get_level(IN_A_ACK_PIN);
    if (currentLevel == 1) {
        xSemaphoreGiveFromISR(ackSemaphore, &
           higherPriorityTaskWoken);
    }
}
```

The ISR, `ack_isr_handler`, is triggered by a GPIO interrupt. When triggered, it checks if the interrupt is because the ACK pin (`IN_A_ACK_PIN`) has gone high. If so, it "gives" the semaphore using `xSemaphoreGiveFromISR`, which sets the semaphore to available.

Semaphore Usage

```
if (xSemaphoreTake(ackSemaphore, pdMS_TO_TICKS(
   TIMEOUT_MICROSECONDS / 1000)) == pdTRUE) {
```

```
2       // ACK Received
3   } else {
4       // Timeout Occured. Continue
5   }
```

In the task `send_data_dual_rail_protocol`, the semaphore is "taken". This task waits for the semaphore to become available before proceeding, ensuring that an ACK was received after a data bit was sent. The task waits up to the specified timeout period. If the semaphore isn't given in that time, it concludes that a timeout occurred, likely indicating an issue in the communication process or a missed interrupt.

9.1.5 Simulation and Testing

9.1.5.1 Serial Connection Monitoring

In order to fully test the behavior of the bridge hardware, a serial connection monitoring must be established. With this way, every message or error can be printed into the console of the connected computer.

For monitoring, two methods have been used:

- PlatformIO's Integrated Monitoring Feature

  `platformio device monitor`

  By executing the above terminal command within the project's directory, PlatformIO automatically detects the serial port that the device is connected and goes into a state where is ready to monitor.

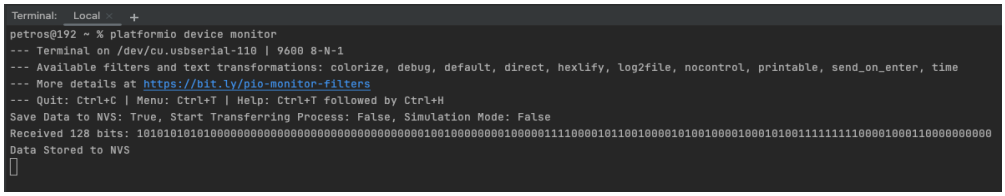


Figure 9.5 PlatformIO Device Monitoring

- Using the "screen" terminal command

`screen /dev/tty.Test BaudRate`

The more manual approach where the user must specify the port and the baudrate.

Figure 9.6 "screen" Terminal Command

9.1.5.2 Complimentary Printing Functions

For convenience and management, there are complimentary functions throughout the procedure to print messages or data to the serial monitoring connection.

`void print_byte_in_binary(uint8_t byte)`

`read_and_print_data_from_nvs()`

9.1.5.3 Dual Rail Protocol Simulator Function

`void simulate_send_data_dual_rail_protocol(char* bitString, size_t size)`

This function is dedicated to the simulation of the Dual Rail Protocol Implementation which was discussed in Subsection 9.1.4.7. A minor difference is that in this function, the event driven approach is not present because the ACK signal is simulated by manually changing the value of the corresponding pin and not waiting it from the asynchronous chip.

9.2 Graphical User Interface Implementation

9.2.1 Coding

9.2.1.1 Part I: Environment Setup

For installing the Qt Development Environment, the Qt company offers an all-in-one installer. The version used for the project is the Qt 5.15.16 under the Qt

Creator IDE. In addition, the Qt Design Studio 4.4.0 is used for designing the front-end interface.

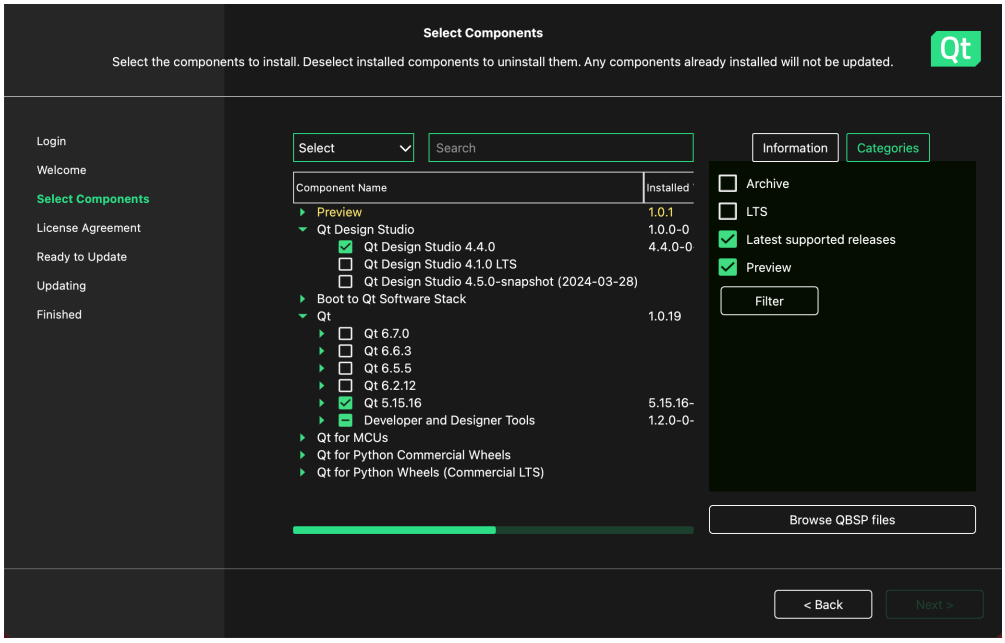


Figure 9.7 Qt Environment Configuration Settings

9.2.1.2 Part II: Project Creation

Create Project → Application (Qt) → Qt Quick Application → Project Location → Kit Selection

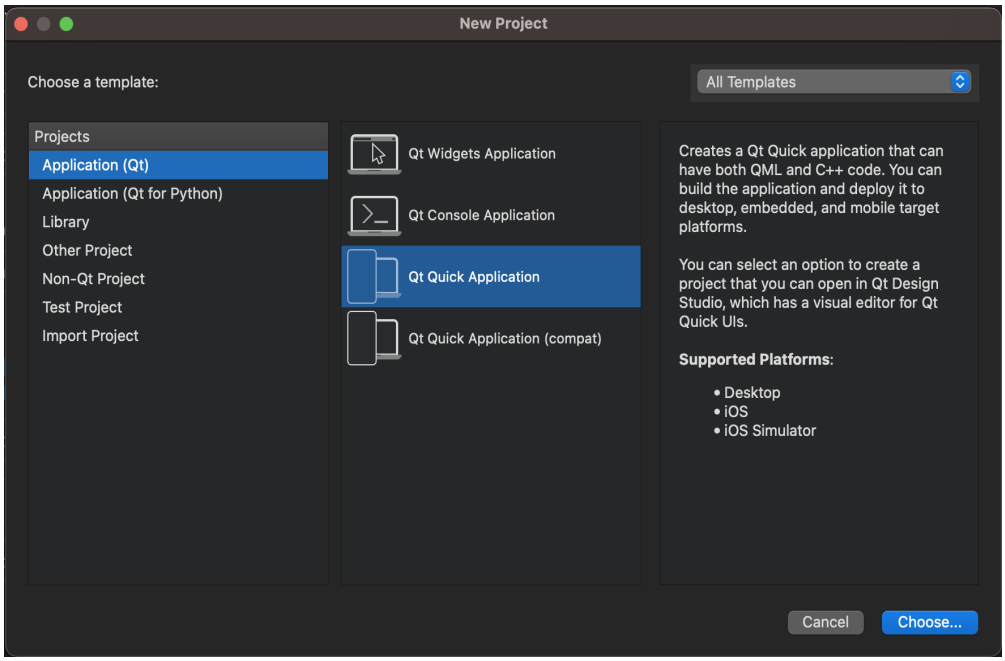


Figure 9.8 Qt Choose Template

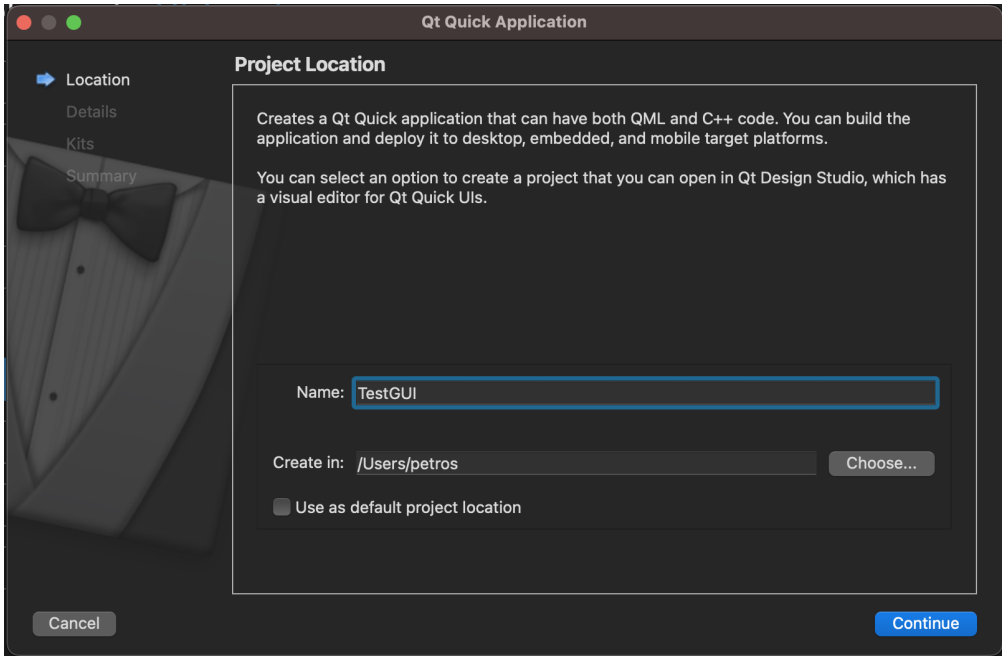


Figure 9.9 Qt Project Location

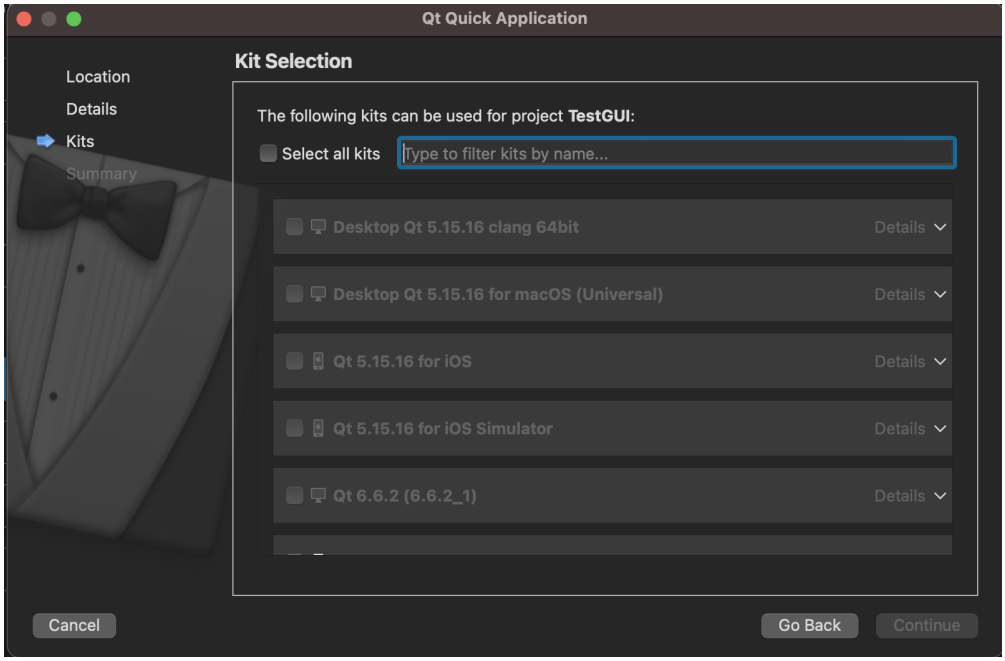


Figure 9.10 Qt Project Kit Selection

Terminology Explanation

QtQuick: This is the latest user interface technology under the Qt framework dedicated for high performance GUI rendering across all platforms. It has QML, a markup language built on Javascript. Basically, it integrates the Javascript language within the C++ Development.

Kit Selection: Kits in Qt are comprehensive packages contain all the set of tools,

libraries, and configurations necessary for developing and building applications for a specific target platform.

9.2.1.3 Part III: Collector Class - Methods

```
/* The purpose of this class is to collect all the input
   user data from the application
   The data is coming from InputParam.qml */

class Collector : public QObject {
    Q_OBJECT
public:
    struct Data {
        string m_startSequence;
        string m_faultTolerantBits;
        string m_xAddress;
        string m_yAddress;
        string m_dac8;
        string m_dac7;
        string m_dac6;
        string m_dac5;
        string m_dac4;
        string m_dac3;
        string m_dac2;
        string m_dac1;
        string m_ack;
    };

    Q_INVOKABLE void collectData(/*User Input Parameters*/);
    Q_INVOKABLE Data getAllData() const;
private:
    Data userData;
};
```

Listing 9.8: Pseudocode of Collector class

Class Explanation

- `Data userData`: A field with type struct Data. It is responsible to hold the user input.
- `void collectData`: Method responsible for converting the user inputs to string format and saving them to the corresponding field in the `userData` struct.
- `Data getAllData`: Method responsible to return the `userData` field when is called from another class, since the `userData` is a private field.

Note: The command `Q_INVOKABLE` is used to enable the specific method to be called within a QML file.

9.2.1.4 Part IV: Serial Communication Establishment Class - Methods

```
class SerialCommunicationEstablishment : public QObject {
public:
    int getSerialPort();
    void openSerialPort();
    void configurePort(int serialPort);
    void sendBooleanValues(int serialPort, bool bool1, bool
       bool2, bool bool3);
    void sendData(int serialPort, const std::string &data);
    void closeSerialPort(int serialPort);
private:
    int serialPort;
}
```

Listing 9.9: Pseudocode of SerialCommunicationEstablishment class

Further Analysis of the Methods within the Class

openSerialPort() Method

```
void openSerialPort() {
    glob_t glob_result;
```

```
3       // Use glob to search for all serial devices matching
            the pattern "/dev/cu.usbserial-*".
4       glob("/dev/cu.usbserial-*", GLOB_TILDE, NULL, &
            glob_result);
5
6       const char* portName = glob_result.gl_pathv[0];
7
8       // Opening the serial port
9       int serialPort = open(portName, O_RDWR | O_NOCTTY |
            O_SYNC);
10
11      // Free the resources allocated by glob to avoid memory
            leaks.
12      globfree(&glob_result);
13  }
```

Listing 9.10: Snippet of the openSerialPort() method

Explanation: The method is able to automatically detect the serial port that the bridge hardware is connected. After the detection, the port opens and saves a non-negative value to the serialPort integer to be used at a later step.

sendBooleanValues() Method

```
1   void sendBooleanValues(int serialPort, bool bool1, bool
        bool2, bool bool3) {
2       unsigned char boolBytes[3];
3       boolBytes[0] = bool1 ? 0xFF : 0x00; // Convert bool1 to
            byte
4       boolBytes[1] = bool2 ? 0xFF : 0x00; // Convert bool2 to
            byte
5       boolBytes[2] = bool3 ? 0xFF : 0x00; // Convert bool3 to
            byte
6
7       write(serialPort, boolBytes, 3); // Send the boolean
            bytes
```

```
}
```

Listing 9.11: Snippet of the sendBooleanValues() method

Explanation: This method receives the boolean configuration settings, saves them into a char array by converting them to byte values and sends them to the bridge hardware.

sendData() Method

```
void sendData(int serialPort, const std::string &data) {
    write(serialPort, data.data(), data.size());
}
```

Listing 9.12: Snippet of the sendData() method

Explanation: This method sends the bitstream to the bridge hardware.

closeSerialPort() Method

```
void closeSerialPort(int serialPort) {
    close(serialPort); // Close the serial port
}
```

Listing 9.13: Snippet of the closeSerialPort method

Explanation: This method is used to close the serial connection.

9.2.1.5 Part V: System Controller Class - Methods

```
#include "serialcommunicationestablishment.h"

class SystemController : public QObject
{
    Q_OBJECT

    \\ Creation of the objects from the other classes
    Collector collector;
    DataPacketFormation dataPacketFormation;
    SerialCommunicationEstablishment serial;
```

```
11 public:
12     Q_INVOKABLE void startSendingBitsProcess(bool saveData,
          bool startProtocol, bool enableSimulation, const
          QVariant &startSequence, const QVariant &
          faultTolerantBits, const QVariant &xAddress, const
          QVariant &yAddress, const QVariant &dac8, const
          QVariant &dac7, const QVariant &dac6, const QVariant
          &dac5, const QVariant &dac4, const QVariant &dac3,
          const QVariant &dac2, const QVariant &dac1, const
          QVariant &ack);
13
14 private:
15     string formattedData;
16     void formatDataPacket();
17 }
```

Listing 9.14: Snippet of the System Controller Class

Note: The command `Q_OBJECT` is used to enable the specific class to be called within a QML file.

Further Analysis of the Methods within the Class

formatDataPacket() Method

```
1  void formatDataPacket() {
2      Collector::Data my_data = collector.getAllData();
3
4      formattedData = my_data.m_startSequence + "00"
5                      + my_data.m_faultTolerantBits + "00"
6                      + my_data.m_xAddress + "00"
7                      + my_data.m_yAddress + "00"
8                      + my_data.m_dac8 + "00"
9                      + my_data.m_dac7 + "00"
10                     + my_data.m_dac6 + "00"
11                     + my_data.m_dac5 + "00"
12                     + my_data.m_dac4 + "00"
```

```
13                          + my_data.m_dac3 + "00"
14                          + my_data.m_dac2 + "00"
15                          + my_data.m_dac1 + "00"
16                          + my_data.m_ack + "00" + "00";
17  }
```

Listing 9.15: Snippet of the formatDataPacket() method

Explanation: This method gets all the input data and concatenates them into a single bitstream string formation. It also adds the separation bits between each input field.

startSendingBitsProcess() Method

```
1  void startSendingBitsProcess(/* User input data and
       configuration settings */) {
2      collector.collectData(/* User input data */);
3
4      formatDataPacket();
5
6      serial.openSerialPort();
7      serial.configurePort(serial.getSerialPort());
8      serial.sendBooleanValues(serial.getSerialPort(),
           saveData, startProtocol, enableSimulation);
9      serial.sendData(serial.getSerialPort(), formattedData);
10     serial.closeSerialPort(serial.getSerialPort());
11 }
```

Listing 9.16: Snippet of the startSendingBits() method

Explanation: This is the main method that orchestrates all the process and the main connection between front-end and back-end.

9.2.1.6 Part VI: Front-End Interface

Figure 9.11 illustrates the implemented Graphical User Interface which was based on the mockup shown in Figure 8.4. It is divided into four separated sections(boxes), each one for a different purpose.

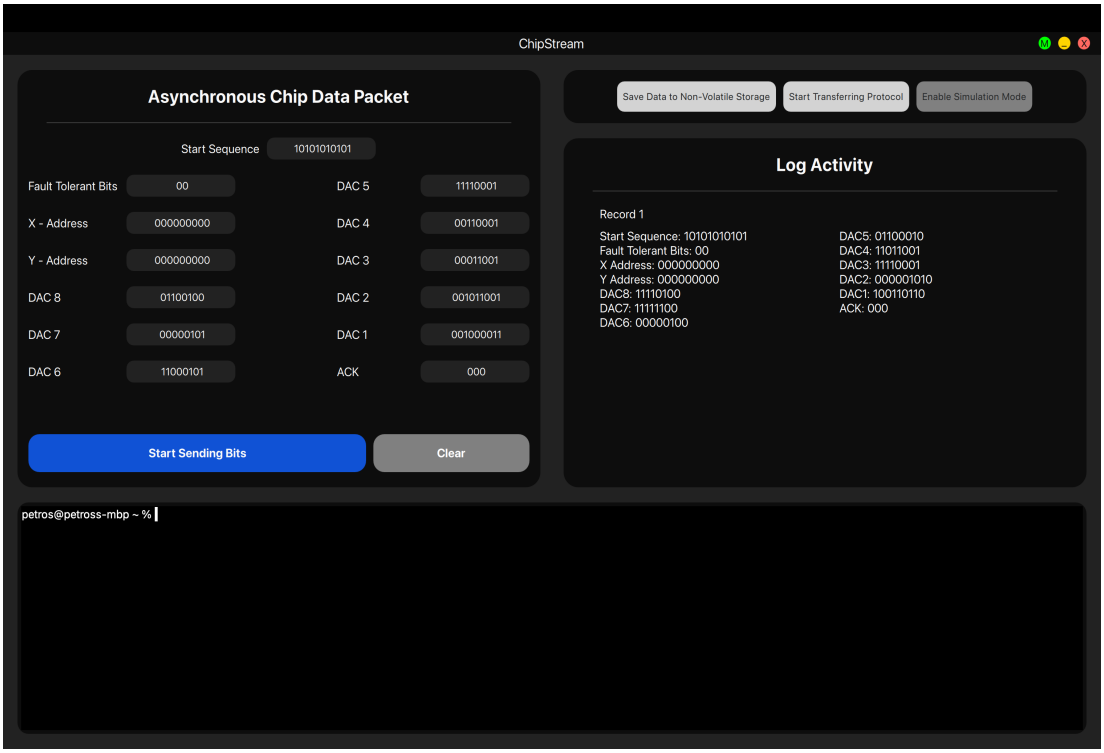


Figure 9.11 Graphical User Interface

Explanation of the GUI

- **Top Left Section - "Asynchronous Chip Data Packet":** This section contains all the user inputs that the user must fill with binary values. It also contains an "Auto-Fill - Clear" button which fills the inputs with random values and also the main "Start Sending Bits" button where it initiates the sending process.
- **Top Right Section:** There are three - "ON/OFF" - buttons so that the user can set the configuration settings.
- **Mid-Right Section - "Log Activity":** A viewing section where it shows previous records and data that being sent previously.
- **Bottom Section:** This section is a terminal session, used for integration of the bridge hardware's monitoring within the GUI.

Explanation - Snippets of the GUI Implementation

Note: The GUI implementation is constructed with four main QML files, one for each section, where those files are bound to the main.cpp file.

- **InputParam.qml**

  Snippet 1:

```
    TextField {
        id: textFieldFaultTolerantBits
        width: 150
        placeholderText: qsTr("Enter value")

        background: Rectangle {
            color: Properties.baseColor // Dark
               background color
            radius: 10 // Rounded corners
        }

        // Text color
        color: Properties.textColor // Set text color to
            white for contrast
        font.pixelSize: 14
        horizontalAlignment: Text.AlignHCenter
        // Enforce number only input
        inputMethodHints: Qt.ImhDigitsOnly
        validator: RegExpValidator { regExp: /[0-9]+/ }
    }
```

Listing 9.17: Snippet for a Row Input

  Snippet 2:

```
    RoundButton {
        // When button is clicked, it executes the
           following functions
        onClicked: {
            systemController.startSendingBitsProcess(/*
               User input data and configuration
               settings */);

            // Display data to the Log Activity ection
```

```
7               logManager.displayData(/* User input data */
                    );
8
9               // Javascript function that clear the input
                    fields
10              initialisationState();
11          }
12      }
```

Listing 9.18: Snippet of the "Start Sending Bits" Button

Snippet 3:

```
1       function generateRandomBinary(length) {
2           var result = '';
3           for (var i = 0; i < length; i++) {
4               result += Math.floor(Math.random() * 2).
                    toString();
5           }
6           return result;
7       }
8
9       for (var i = 0; i < fields.length; i++) {
10          fields[i].id.text = generateRandomBinary(fields[
                i].length);
11      }
12      }
```

Listing 9.19: Snippet of the function that generates random binary values

- **ConigurationSettings.qml**

  This QML file contains the implementation of the three - "ON/OFF" - buttons for the configuration settings.

- **LogActivity.qml**

```
1           ListView {
2               id: logListView
```

```
3               anchors.fill: parent
4               anchors.margins: 10
5               // The model command triggers the
                   logMessages method from the back-end
6               model: logManager.logMessages  of strings
7               ScrollBar.vertical: ScrollBar { active: true
                    }
8
9               // More formating attributes...
10          }
```

Listing 9.20: Snippet of LogActivity Display Section

```
1       void LogManager::addLog(const QString &logMessage) {
2           m_logMessages.append(logMessage);
3       }
```

Listing 9.21: Snippet of Log Messages

- **Terminal.qml**

```
1       // Shortcut that enables the Copy functionality
2       Action{
3           onTriggered: terminal.copyClipboard();
4           shortcut: "Ctrl+Shift+C"
5       }
6       // Shortcut that enables the Paste functionality
7       Action{
8           onTriggered: terminal.pasteClipboard();
9           shortcut: "Ctrl+Shift+V"
10      }
11
12      QMLTermWidget {
13          // Component that contains the styling
               configurations of the terminal
14      }
```

Listing 9.22: Snippet of the Teminal.qml

9.3 System Troubleshoot

9.3.1 Troubleshoot of Bridge Hardware

9.3.1.1 Improper Default Initialization of Flash Size

Problem Statement

The software was unable to be uploaded to the MCU and the following message was shown to the console.

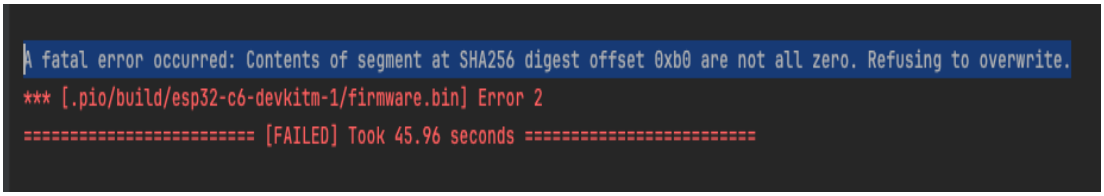


Figure 9.12 Demonstration of the Issue

9.3.1.2 Detection of the Issue

After some research on the PlatformIO community and the official github repository of the Espressif systems, there were a few developers stating that there is a misalignment to the Flash Memory configurations in an sdkconfig file. More precisely, when the project was first created with the PlatformIO IDE, the ESP32 flash memory was initialized to 2MB. Such thing created a conflict when uploading a project to the MCU, because it has 4MB of flash memory [26] [27].

9.3.1.3 Solution

The solution is to change the Flash Memory size form 2MB to 4MB. The steps of doing this are:

1. Run the following terminal command to the project directory

   pio run -t menuconfig

   This command will open a graphical user interface for managing the configurations of the project.

2. Change the Flash size

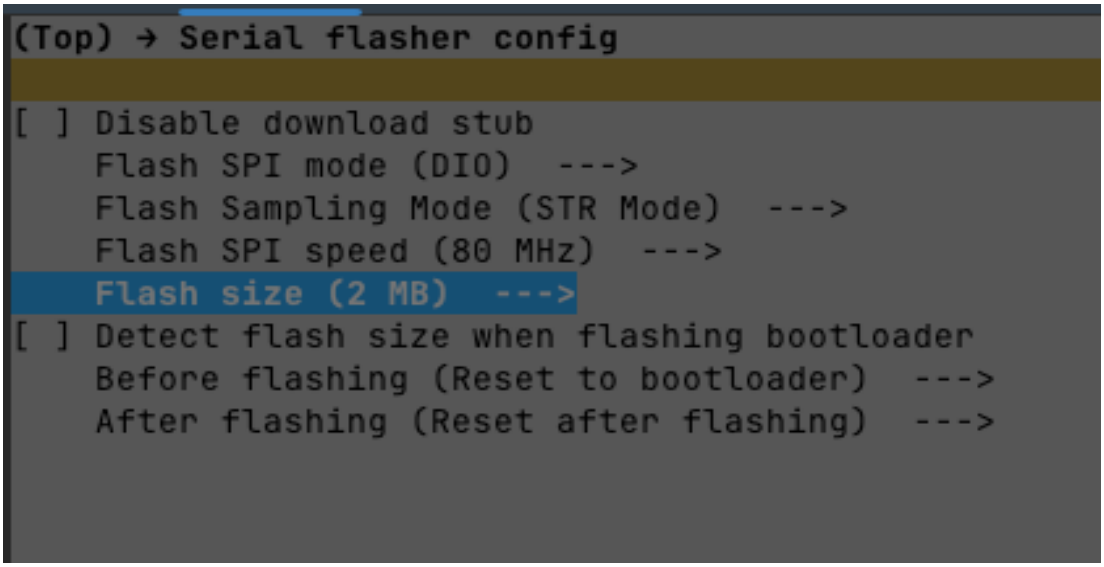


Figure 9.13 Configuration Menu

Figure 9.14 Change the Flash size

9.3.1.4 ESP32 Core Dump

Problem Statement

Upon execution of the function that is responsible for the implementation of the communication protocol between the MCU and the asynchronous chip, a software crash had been occurring and an error was showing. The error was stating "Core Register Dump."

```
Core <0 register dump:
MEPC    : 0x42000fec  RA      : 0x4200033a  SP      : 0x4087e920  GP      : 0x4080b790
TP      : 0x408762ac  T0      : 0x40028192  T1      : 0x4087e57c  T2      : 0x30303030
S0/FP   : 0x0000007f  S1      : 0x00000080  A0      : 0x00000006  A1      : 0x4087e8f0
A2      : 0x00000000  A3      : 0x0000000a  A4      : 0x4080d000  A5      : 0x15232300
A6      : 0x00000001  A7      : 0x0000000a  S2      : 0x00000001  S3      : 0x4087e9eb
S4      : 0x4087e96c  S5      : 0x00000000  S6      : 0x00000000  S7      : 0x00000000
S8      : 0x00000000  S9      : 0x00000000  S10     : 0x00000000  S11     : 0x00000000
T3      : 0x00000000  T4      : 0x31313130  T5      : 0x30313031  T6      : 0x31313030
MSTATUS : 0x00000001  MTVEC   : 0x4087ee30  MCAUSE  : 0x4080d990  MTVAL   : 0x400281a0
```

Figure 9.15 Demonstration of the Core Register Dump Issue

Note: A core dump is created by a panic handler during a software crash, capturing a detailed snapshot of the system's state. This helps in identifying the reasons behind the crash by preserving crucial information like task states and memory details [28].

Detection of the Issue

The issue lies within the busy wait loop implementation when the MCU waits for the asynchronous chip to send an ACK back, indicating the it received the sending bit.

```
1 while (gpio_get_level(IN_A_ACK_PIN) == 0) {
2     // Busy Wait
3 }
```

Solution

The solution to this problem was to add to the event driven approache with the semaphores that was discussed in the Section 9.1.4.7.

9.3.2 Troubleshoot GUI - Bridge Hardware

9.3.2.1 Improper Data Reception and Handling from Bridge Hardware

Problem Statement

During serial communication between a GUI application and the ESP32 microcontroller, inconsistent data handling was observed. The ESP32 sometimes read and saved data to the boolValues buffer as expected, but at other times, it incorrectly saved all incoming data into the data buffer. This behavior suggested a timing and synchronization issue between the transmission of data from the GUI and the reception of data by the ESP32.

Figure 9.16 Demonstration of the Issue

Possible Reasons of the Issue

- **Data Overwriting:** Investigated potential overlap between reading `boolValues` and `data` due to the UART buffer being filled faster than it is being read, or because two `uart_read_bytes` calls are too close together without proper handling of the state between reads.
- **Incorrect Printing:** Evaluated the correctness of the `printf` statement for `boolValues`. The use of `boolValues` with `%s` in `printf` suggests an expectation of a null-terminated string, which is not guaranteed for an array of `uint8_t`, particularly when bytes received are `0xFF`. This can cause unexpected output.

Detection of the Issue

It turns out that the issue was with how the MCU's UART buffer managed the data. The two commands that read the data were executed too closely together

in sequence. This meant that they were both trying to access the same buffer, which is a critical region of the system, without giving enough time to process the previous data properly.

Solution

A state machine was designed to manage two specific types of data input: boolean control signals and standard data bytes(char type). The machine utilized a control flag named `expectBoolValues`, indicating the nature of the next data input to be read from the UART buffer.

State Machine Description

States:

- `AWAITING_BOOL`: The system awaits boolean values.
- `AWAITING_DATA`: The system awaits regular data bytes.

Transitions:

- Transition from `AWAITING_BOOL` to `AWAITING_DATA` after processing 3 bytes of boolean values.
- Transition from `AWAITING_DATA` to `AWAITING_BOOL` after processing regular data bytes.

Actions:

- In `AWAITING_BOOL`, read and process 3 bytes, then transition to `AWAITING_DATA`.
- In `AWAITING_DATA`, read and process up to `BUF_SIZE` bytes, then transition to `AWAITING_BOOL`.

State Machine Diagram

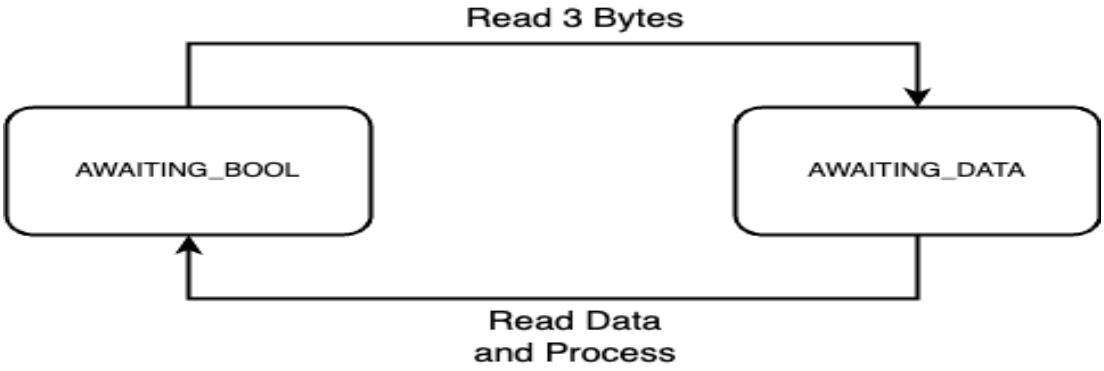


Figure 9.17 Visualisation of the Proposed State Machine Diagram

Code Snippets

State Management Initialization:

```
bool expectBoolValues = true; // State to track what is
    expected to read
```

Listing 9.23: Boolean Flag

State Machine Implementation within system_controller:

```
// Inside the system_controller loop:
if (expectBoolValues) {
    // Read boolean values and switch state
    expectBoolValues = false;
} else {
    // Read and process regular data and switch state
    expectBoolValues = true;
}
```

Listing 9.24: Conditional Statement for Avoiding Overlapping

Buffer Reading and Processing:

```
// Clearing the buffer and reading boolean values:
memset(boolValues, 0, sizeof(boolValues));
int bytesRead = uart_read_bytes(UART_NUM_0, boolValues,
    sizeof(boolValues), pdMS_TO_TICKS(100));
```

```
// ... processing of boolean values ...

// Clearing the buffer and reading regular data:
memset(data, 0, sizeof(data));
int len = uart_read_bytes(UART_NUM_0, data, BUF_SIZE,
    pdMS_TO_TICKS(100));
// ... processing of regular data ...
```

Listing 9.25: Buffer Reading and Processing

Ensuring Proper Timing:

```
// Delay to prevent data overlap and to maintain
    synchronization:
vTaskDelay(pdMS_TO_TICKS(100));
```

Listing 9.26: Delay Command

The implementation of this state machine architecture has allowed the ESP32 to reliably distinguish between control signals and data bytes, ensuring each is handled according to its specific processing requirements. Delays are incorporated to prevent premature data reads, thereby avoiding buffer overflow and ensuring synchronization with the GUI's data transmission.

9.4 Setup of the System

9.4.1 Integration of a Level Shifter

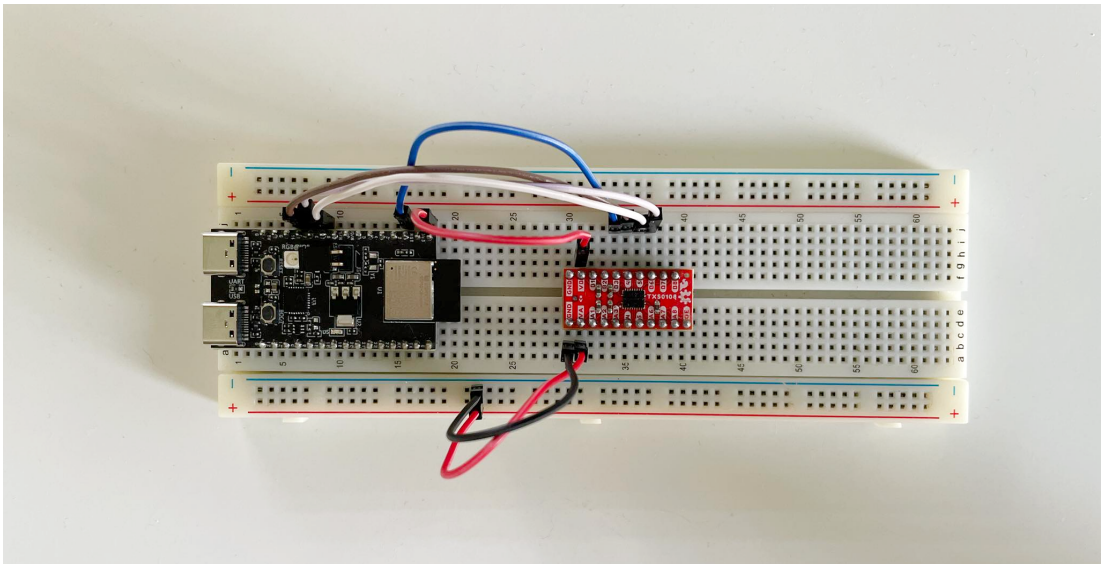


Figure 9.18 Setup of the System - Step 1

A voltage level shifter must be integrated into the setup. The reason behind this is because the MCU's minimum voltage output is 3.3V and the asynchronous chip was designed for maximum 1.8V. The model used was the 8-channel TXS0108E from Sparkfun.

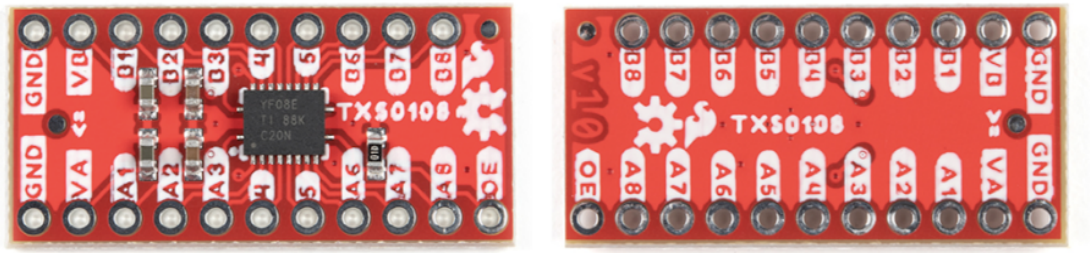


Figure 9.19 Sparkfun Level Shifter - 8-Channel TXS0108E

Figure 9.19 illustrates the Level Shifter where it converts from 3.3V (Side B) to 1.8V (Side A). In order for the Shifter to work, it must supplied with 3.3V from the one side and 1.8V to the other.

9.4.2 Connecting the Asynchronous Chip

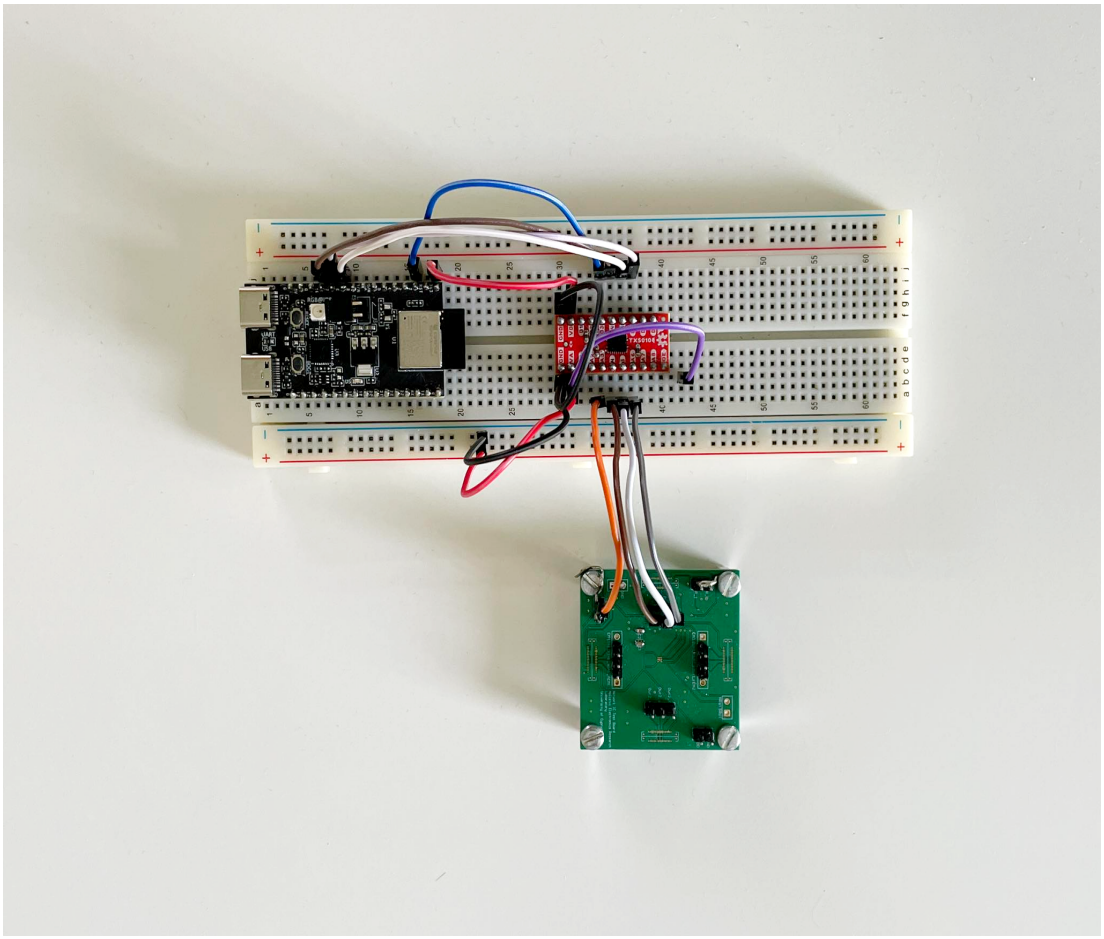


Figure 9.20 Setup of the System - Step 2

Figure 9.20 demonstrated how the asynchronous chip has connected. Each one of the pins (HRST, IN_F, INT_F, IN_A) have connected to specific places on the breadboard in order to be aligned with the conversion of the level shifter from the corresponding MCU pins.

9.4.3 Final Setup

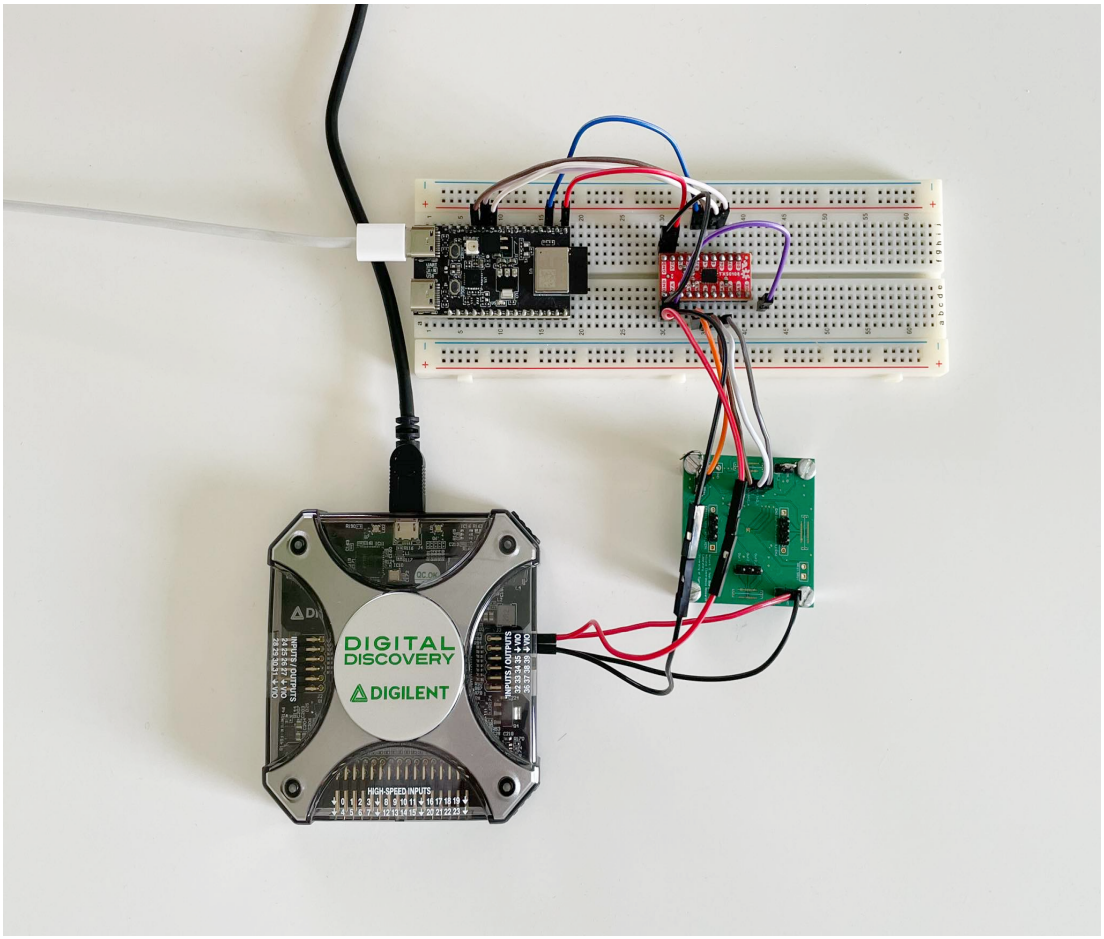


Figure 9.21 Setup of the System - Step 3

Upon finalising the test setup, a voltage supply unit must be added to give 1.8V to the level shifter and also power up the asynchronous chip. The supply which was used is the 'Digital Discovery' module by Digilent™. It is worth mentioning that the level shifter receives the 3.3V for side B from the MCU.

Chapter 10

Performance Evaluation and Discussion

10.1 Communication Latency GUI - MCU

To calculate the time that the GUI needs to send the data to the MCU through the serial port, the time library **std::chrono** was used to capture the necessary timestamps [29]. Then, a sample of total 50 measures has been taken and stored into a CSV file in order for it to go under data analysis with a Python script.

10.1.1 Usage of the std::chrono

For a clock, the **high_resolution_clock** was used which is the clock with the shortest tick period [30].

10.1.1.1 Detection of the clock period

To define the exact clock period that the **high_resolution_clock** uses, the member type **period** was used which is included in the **high_resolution_clock** [30].

```
1    std::cout << "Clock␣period:␣" << std::chrono::
       high_resolution_clock::period::num << "/" << std::
       chrono::high_resolution_clock::period::den << "
       seconds." << std::endl;
```

Listing 10.1: C++ Use of the member type period

- `std::chrono::high_resolution_clock::period::num`: This retrieves the numerator of the ratio that defines the period of the clock. It represents the number of ticks per second [30].
- `std::chrono::high_resolution_clock::period::den`: This retrieves the denominator of the ratio. When combined with the numerator, this gives the duration of one tick in seconds [30].

The clock period, which is the output of this command is **1/1000000000 seconds**.

10.1.1.2 Capture of starting and ending timestamps

The **now()** method from the **high_resolution_clock** was used to capture the exact timestamps before the start of the communication and right after the end of the communication [30].

```
1  // Start the timer
2  auto start = std::chrono::high_resolution_clock::now();
3
4  serial.openSerialPort();
5  serial.configurePort(serial.getSerialPort());
6  serial.sendBooleanValues(serial.getSerialPort(), saveData,
       startProtocol, enableSimulation);
7  serial.sendData(serial.getSerialPort(), formattedData);
8  serial.closeSerialPort(serial.getSerialPort());
9
10 // Stop the timer
11 auto end = std::chrono::high_resolution_clock::now();
```

Listing 10.2: C++ Timestamps capturing using the std::chrono

10.1.2 Latency Results

The following are the calculations that were taken after the completion of the data analysis:

- **Minimum Value:** 22.301 ms

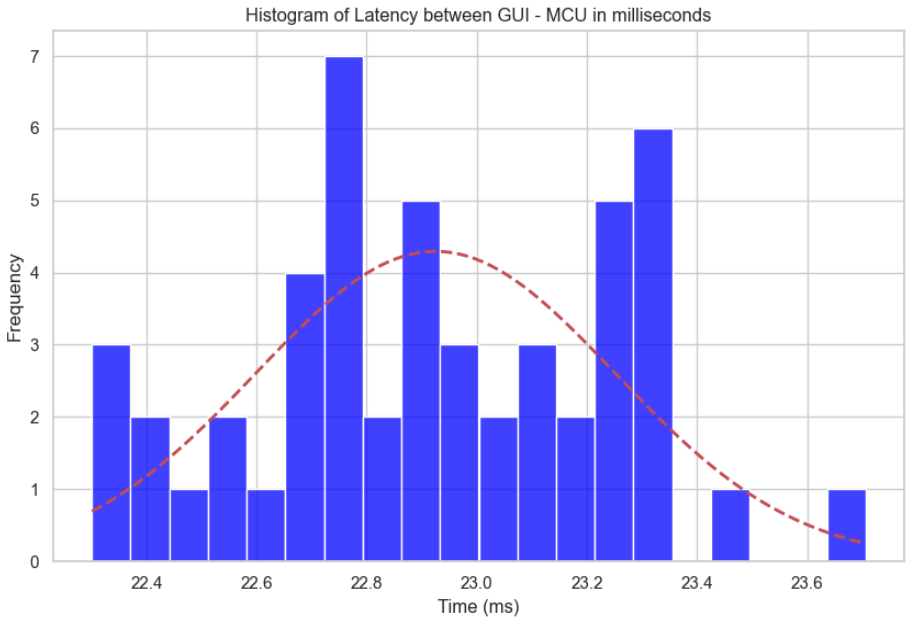


Figure 10.1 Histogram of Latency between GUI - MCU

- **Maximum Value:** 23.706 ms
- **Average (Mean):** 22.925 ms
- **Standard Deviation:** 0.326 ms

The data was also fitted to a Gaussian distribution, with the fit parameters:

- **Mean (μ):** 22.925 ms
- **Standard Deviation (σ):** 0.323 ms

10.2 Communication Throughput GUI - MCU

To calculate the throughput for the communication between the GUI and the MCU, the following formula was used:

$$\text{Throughput} = \frac{\text{Data Amount (in bits)}}{\text{Time (in seconds)}}$$

Data Amount:

- 128 bits of data

- 3 boolean values. Since each boolean value can be represented by 1 bit, the total is extra 3 bits.
- Total amount of data: 131 bits

The throughput was calculated for each time measure of the dataset.

- **Minimum Throughput:** 5526.03 bits per second (bps)
- **Maximum Throughput:** 5874.18 bps
- **Mean Throughput:** 5715.41 bps
- **Standard Deviation:** 81.44 bps

10.3 Communication Latency MCU - Asynchronous Chip

To calculate the time that the MCU needs to send the data to the asynchronous chip, the time library **time.h** was used to capture the necessary timestamps and more specifically the function **clock_gettime()** [31]. Then, similarly to the GUI-MCU latency calculation, a sample of total 50 measures has been taken and stored into a CSV file in order for it to go under data analysis with a Python script.

10.3.1 Usage of the clock_gettime() function

```
1    clock_gettime(CLOCK_MONOTONIC, &start);
2    // Execution of the communication protocol
3    clock_gettime(CLOCK_MONOTONIC, &end);
```

10.3.1.1 Function Prototype - Explanation

```
1 clock_gettime(clockid_t clk_id, struct timespec *tp);
```

Arguments

- `clockid_t clk_id`: Specifies the clock type. Common clock IDs include:

  - `CLOCK_REALTIME` – System-wide real-time clock.
  - `CLOCK_MONOTONIC` – Monotonic clock that represents the elapsed time since an unspecified point in the past. This is the one used in the current implementation.
  - `CLOCK_PROCESS_CPUTIME_ID` – High-resolution timer that measures CPU time consumed by all threads in the calling process.
  - `CLOCK_THREAD_CPUTIME_ID` – Timer that measures CPU time consumed by the calling thread [31].

- `struct timespec *tp`: A pointer to a `struct timespec` structure where the function stores the time [31]. This structure is defined as:

```
1    struct timespec {
2        time_t tv_sec;  // seconds
3        long   tv_nsec; // nanoseconds
4    }
```

10.3.1.2 Clock Period

For detecting the clock period, the function **clock_getres()** was used.

```
if (clock_getres(CLOCK_MONOTONIC, &res) == 0) {
    printf("Resolution of CLOCK_MONOTONIC: %ld nanoseconds\n",
           res.tv_nsec);
}
```

This function finds the resolution (precision) of the specified clock clk_id, and, if res is non-NULL, stores it in a struct timespec [31].

In the implementation, the clock period based on this function is 1000 nanoseconds.

10.3.2 Latency Results

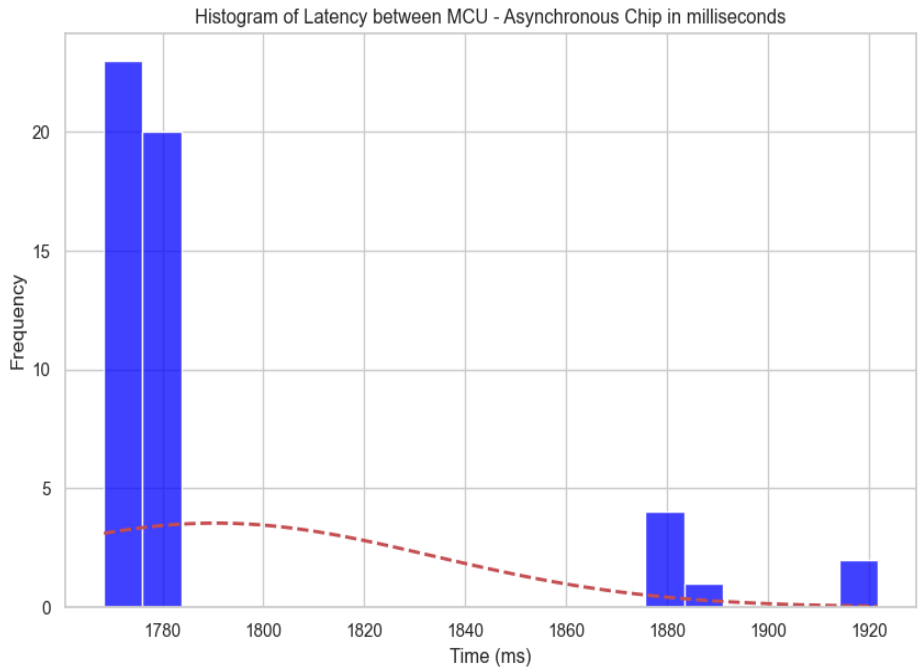


Figure 10.2 Histogram of Latency between GUI - MCU

- **Minimum Value:** 1768.397 ms
- **Maximum Value:** 1921.751 ms
- **Average (Mean):** 1790.476 ms
- **Standard Deviation:** 43.274 ms

The Gaussian fit to this dataset gives:

- **Mean (μ):** 1790.476 ms
- **Standard Deviation (σ):** 42.839 ms

10.4 Communication Throughput MCU - Asynchronous Chip

Similarly, the throughput for the communication between the MCU and the Asynchronous Chip, the same formula was used:

$$\text{Throughput} = \frac{\text{Data Amount (in bits)}}{\text{Time (in seconds)}}$$

Data Amount:

- Total of 128 bits of data

The throughput was calculated for each time measure of the dataset.

- **Minimum Throughput:** 66.61 bits per second (bps)
- **Maximum Throughput:** 72.38 bps
- **Mean Throughput:** 71.53 bps
- **Standard Deviation:** 1.64 bps

Chapter 11

Conclusion

11.1 Summary of Findings

This thesis has effectively addressed the challenges of bridging synchronous and asynchronous digital circuits. The initial sections laid the foundation for understanding the essential principles and challenges of semiconductor technology and system design. Based on this foundation, the following parts of the thesis carefully searched for the optimal setup, focusing on hardware selection (especially on microcontrollers), communication protocols, and software engineering strategies to develop a robust system architecture.

In the practical implementation phase, specific timing and synchronization strategies were applied, utilizing serial communication and MCU-based systems to ensure seamless data transfer and system reliability. The success of these strategies was evident in performance evaluations and system tests, demonstrating significant improvements in system synchronization and timing precision. Notably, the evaluations highlighted consistent latency and throughput measurements, confirming the effectiveness of the implemented communication protocols between the GUI, MCU, and the asynchronous chip.

The cumulative results of these comprehensive studies have provided new insights and methodologies that enhance the interoperability and efficiency of mixed-timing digital circuit systems, showcasing a successful application of theoretical knowledge to practical challenges.

11.2 Contributions to the Field

This study significantly contributes to the field of asynchronous and synchronous digital communication by addressing critical communication challenges between the two categories of digital circuits. Notably, the implementation of the 4-phase dual-rail protocol in a high-level programming language on a microcontroller unit marks a significant advancement.
Key contributions include:

- **Practical Implementation of an Asynchronous Protocol**
  This thesis successfully implements the 4-phase dual-rail protocol, an asynchronous communication protocol, on an MCU. The choice of C as the programming language enhances the accessibility and modifiability of the system.

- **User-Friendly Interface Development**
  The system includes a straightforward and user-friendly interface that facilitates easy control and manipulation of the protocol functions. This interface effectively simplifies interactions with the system, making it more accessible to users. It is designed to support crucial features such as sending bitstreams, monitoring responses, and logging data. These capabilities ensure that users can not only execute the necessary commands efficiently, but also track and analyze the system performance and outputs seamlessly. This enhances the practical utility of the system for both educational and research applications.

- **Bridging Theoretical Research with Practical Application**
  The implementation of the protocol in a practical, accessible form bridges a crucial gap in the research landscape. This contribution is particularly valuable for advancing the design and optimization of mixed-timing systems in the semiconductor industry, enhancing the ability to deploy these systems more effectively across various applications.

These contributions not only advance the technical knowledge and practical capabilities in handling asynchronous communication in digital circuits but also provide a robust framework for further exploration and innovation in the field.

11.3 Future Work and Recommendations

While this study has made significant road in understanding and improving mixed-timing communication systems, several avenues for future work remain:

- **Scalability of Solutions**
  Future studies should explore the scalability of the proposed solutions across different platforms and more complex asynchronous systems.

- **Advanced Error Checking Mechanisms**
  Developing advanced error checking mechanisms to further enhance data integrity in high-speed communication environments.

- **Integration with Emerging Technologies**
  A promising direction involves integrating wireless technologies, such as Bluetooth. This integration could facilitate the remote operation and data transmission from the GUI, enhancing the flexibility and applicability of the system in mobile and distributed environments.

In conclusion, the findings and contributions of this thesis not only advance our understanding of synchronous and asynchronous system interfaces but also lay a solid foundation for future innovations in semiconductor technology. By addressing both theoretical and practical aspects, this work paves the way for designing more efficient and robust digital systems.

References

[1] A. Balasinski, *Semiconductors: Integrated Circuit Design for Manufacturability*. CRC Press, 2018.

[2] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic, *Digital integrated circuits*. Prentice hall Englewood Cliffs, 2002, vol. 2.

[3] P. A. Beerel, R. O. Ozdag, and M. Ferretti, *A designer's guide to asynchronous VLSI*. Cambridge University Press, 2010.

[4] J. Sparsø, *Introduction to Asynchronous Circuit Design*. DTU Compute, Technical University of Denmark, 2020.

[5] R. M, *A View On: The Types of Variation — chipedge.com*, `https://chipedge.com/a-view-on-the-types-of-variation/`, [Accessed 21-04-2024].

[6] R. Garg, *Analysis and design of resilient VLSI circuits: mitigating soft errors and process variations*. Springer Science & Business Media, 2009.

[7] A. L., *A Monte Carlo Simulation in Cadence Virtuoso Step by Step — miscircuitos.com*, `https://miscircuitos.com/monte-carlo-simulation-cadence-virtuoso/`, [Accessed 21-04-2024].

[8] C. C. Wai Wong, *ENGI 5131 — Monte Carlo Analysis*, `https://vision.lakeheadu.ca/eele5131/tut/montecarlo.pdf`, [Accessed 21-04-2024].

[9] *Keysight Technologies, Inc*, `https://edadownload.software.keysight.com/eedl/iccap/2011_01/pdf/corner.pdf`.

[10] J. F. Wakerly, *Digital Design: Principles and Practices, 4/E*. Pearson Education India, 2008.

[11] C. Maxfield, *The design warrior's guide to FPGAs: devices, tools and flows*. Elsevier, 2004.

[12] hardwarebee, *Top 10 FPGA Advantages - HardwareBee — hardwarebee.com*, `https://hardwarebee.com/top-10-fpga-advantages/`, [Accessed 27-02-2024].

[13] R. Sass and A. G. Schmidt, *Embedded systems design with platform FPGAs: principles and practices*. Morgan Kaufmann, 2010.

[14] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "High-level synthesis: From algorithm to digital circuit", in *ch. AutoPilot: A Platform-Based ESL Synthesis System*, Springer Netherlands, 2008, pp. 99–112.

[15] S. F. Barrett and D. J. Pack, *Microcontrollers fundamentals for engineers and scientists*. Springer Nature, 2022.

[16] *Advantages and Disadvantages of Microcontroller - GeeksforGeeks — geeksforgeeks.org*, https://www.geeksforgeeks.org/advantages-and-disadvantages-of-microcontroller/, [Accessed 29-02-2024].

[17] S. R. Rizvi, *Microcontroller programming: an introduction*. Crc Press, 2016.

[18] J. Axelson, *Serial port complete: COM ports, USB virtual COM ports, and ports for embedded systems*. Lakeview Research, 2007.

[19] R. Heydon, *Bluetooth Low Energy: The Developer's Handbook*. Prentice Hall, 2012, ISBN: 013288836X,9780132888363.

[20] J. Ross, *The book of wireless: A painless guide to wi-fi and broadband wireless*. No Starch Press, 2008.

[21] L. P. Petrou, "Asynchronous chiplets for reconfigurable metasurfaces", 2023.

[22] I. Sommerville, *Software engineering (ed.9)*. 2011.

[23] PlatformIO, *PlatformIO: Your Gateway to Embedded Software Development Excellence — platformio.org*, https://platformio.org/, [Accessed 03-04-2024].

[24] *This page describes the RTOS xTaskCreate() FreeRTOS API function which is part of the RTOS task control API. FreeRTOS is a professional grade, small footprint, open source RTOS for microcontrollers. — freertos.org*, https://www.freertos.org/a00125.html, [Accessed 03-04-2024].

[25] A. S. Tanenbaum, *Modern operating systems*. China-Pub-Com, 2002.

[26] *ESP32-C6, espidf, error when using components folder, led*$_s$*trip, h − − − community.platformio.org*, https://community.platformio.org/t/esp32-c6-espidf-error-when-using-components-folder-led-strip-h/37006, [Accessed 21-04-2024].

[27] *GitHub - espressif/esp-idf: Espressif IoT Development Framework. Official development framework for Espressif SoCs. — github.com*, https://github.com/espressif/esp-idf, [Accessed 21-04-2024].

[28] *Core Dump - ESP32 - ESP-IDF Programming Guide v5.2.1 documentation — docs.espressif.com*, https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/core_dump.html, [Accessed 22-04-2024].

[29] *Cplusplus.com*, https://cplusplus.com/reference/chrono/, [Accessed 10-05-2024].

[30] *Cplusplus.com*, https://cplusplus.com/reference/chrono/high_resolution_clock/, [Accessed 10-05-2024].

[31] https://linux.die.net/man/3/clock_gettime, [Accessed 11-05-2024].

[32] C. J. Myers, *Asynchronous circuit design*. John Wiley & Sons, 2001.

[33] L. Lavagno and A. Sangiovanni-Vincentelli, *Algorithms for synthesis and testing of asynchronous circuits*. Springer Science & Business Media, 1993, vol. 232.

[34] C. A. R. Hoare *et al.*, *Communicating sequential processes*. Prentice-hall Englewood Cliffs, 1985, vol. 178.

[35] *ESP32-C6-DevKitM-1 - - &x2014; esp-dev-kits latest documentation — docs.espressif.com*, https://docs.espressif.com/projects/espressif-esp-dev-kits/en/latest/esp32c6/esp32-c6-devkitm-1/user_guide.html, [Accessed 21-04-2024].

[36] *ESP-IDF Programming Guide - ESP32-C6 - &x2014; ESP-IDF Programming Guide v5.2.1 documentation — docs.espressif.com*, https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32c6/index.html, [Accessed 21-04-2024].

[37] *Universal Asynchronous Receiver/Transmitter (UART) - ESP32-C6 - &x2014; ESP-IDF Programming Guide v5.2.1 documentation — docs.espressif.com*, https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32c6/api-reference/peripherals/uart.html, [Accessed 21-04-2024].

[38] *GPIO & RTC GPIO - ESP32-C6 - &x2014; ESP-IDF Programming Guide v5.2.1 documentation — docs.espressif.com*, https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32c6/api-reference/peripherals/gpio.html, [Accessed 21-04-2024].

[39] *Qt Documentation — Home — doc.qt.io*, https://doc.qt.io/, [Accessed 21-04-2024].

[40] *GitHub - Swordfish90/qmltermwidget: QML port of qtermwidget — github.com*, https://github.com/Swordfish90/qmltermwidget, [Accessed 21-04-2024].

Appendix A

Data Analysis Python Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm

# Load your data
data = pd.read_csv('Thesis Measures_MCU-Async Latency.csv')

# Descriptive statistics for latency
min_value = data['Time (ms)'].min()
max_value = data['Time (ms)'].max()
mean_value = data['Time (ms)'].mean()
std_dev = data['Time (ms)'].std()

# Convert time from milliseconds to seconds for throughput
    calculation
data['Time (s)'] = data['Time (ms)'] / 1000

# Data amount in bits (128 bits are being sent)
data_amount_bits = 128

# Calculate throughput (bits per second)
data['Throughput (bps)'] = data_amount_bits / data['Time (s)
    ']

# Plotting the histogram and Gaussian fit
sns.set(style='whitegrid')
plt.figure(figsize=(10, 6))
sns.histplot(data['Time (ms)'], kde=False, color='blue',
    bins=20)
x = np.linspace(min_value, max_value, 100)
y = norm.pdf(x, mean_value, std_dev) * len(data['Time (ms)'
    ]) * (max_value - min_value) / 20
plt.plot(x, y, 'r--', linewidth=2)
plt.title('Histogram of Latency between MCU - Asynchronous
    Chip in milliseconds')
```

```
plt.xlabel('Time (ms)')
plt.ylabel('Frequency')
plt.show()
plt.savefig('latency_histogram.png')

# Output the throughput statistics
print(f'Minimum Value: {min_value}')
print(f'Maximum Value: {max_value}')
print(f'Mean Value: {mean_value}')
print(f'Standard Deviation: {std_dev}')
print('Throughput Statistics:')
print(f'Minimum Throughput: {data["Throughput (bps)"].min()}
    bps')
print(f'Maximum Throughput: {data["Throughput (bps)"].max()}
    bps')
print(f'Mean Throughput: {data["Throughput (bps)"].mean()}
    bps')
print(f'Standard Deviation of Throughput: {data["Throughput
    (bps)"].std()} bps')
```

Appendix B

Timing Measures

B.1 Data from GUI-MCU Latency Measurements

Table B.1: Latency measurements between the GUI and MCU

Measure #	Time (ms)
1	23.191
2	23.299
3	22.765
4	22.913
5	23.231
6	22.472
7	23.334
8	23.166
9	22.903
10	23.129
11	23.460
12	23.326
13	23.308
14	22.883
15	22.326
16	22.719
17	22.544
18	23.226

Table B.1 continued from previous page

Measure #	Time (ms)
19	22.979
20	23.328
21	22.399
22	22.545
23	22.976
24	23.093
25	23.287
26	22.762
27	22.812
28	22.853
29	22.718
30	22.301
31	22.741
32	23.013
33	22.610
34	22.653
35	22.771
36	22.931
37	22.330
38	23.052
39	22.425
40	22.792
41	22.714
42	22.908
43	22.773
44	23.706
45	23.111
46	23.260
47	23.223

Table B.1 continued from previous page

Measure #	Time (ms)
48	23.284
49	22.728
50	22.979

B.2 Data from MCU-Async Latency Measurements

Table B.2: Latency measurements for MCU-Asynchronous Chip

Measure #	Time (ms)
1	1777.807
2	1769.618
3	1777.809
4	1769.756
5	1777.809
6	1769.709
7	1777.809
8	1769.758
9	1777.809
10	1769.781
11	1769.630
12	1769.838
13	1921.750
14	1777.807
15	1769.569
16	1883.277
17	1777.809
18	1883.288
19	1777.810

Table B.2 continued from previous page

Measure #	Time (ms)
20	1777.809
21	1883.277
22	1777.810
23	1769.830
24	1883.264
25	1777.810
26	1769.700
27	1777.807
28	1769.648
29	1777.809
30	1769.418
31	1777.809
32	1769.836
33	1888.809
34	1768.397
35	1769.840
36	1777.809
37	1777.809
38	1769.762
39	1769.758
40	1777.809
41	1777.809
42	1769.707
43	1921.751
44	1777.807
45	1769.573
46	1769.580
47	1769.843
48	1777.809

Table B.2 continued from previous page

Measure #	Time (ms)
49	1769.825
50	1769.839