Enhancing Input/Output Correctness, Protection, Performance, and Scalability for Process Control Platforms

Ryan D. Burrow

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Engineering

Cameron D. Patterson, Chair
William O. Plymale
Patrick R. Schaumont

April 24, 2019
Blacksburg, Virginia

Keywords: digital control system, programmable system-on-chip, model checking, input/output processor, malware resilience

Enhancing Input/Output Correctness, Protection, Performance, and Scalability for Process Control Platforms

Ryan D. Burrow

(ABSTRACT)

Most modern control systems use digital controllers to ensure safe operation. We modify the traditional digital control system architecture to integrate a new component known as a trusted input/output processor (TIOP). TIOPs interface to the inputs (sensors) and outputs (actuators) of the system through existing communication protocols. The TIOPs also interface to the application processor (AP) through a simple message passing protocol. This removes any direct input/output (I/O) interaction from taking place in the AP. By isolating this interaction from the AP, system resilience against malware is increased by enabling the ability to insert run-time monitors to ensure correct operation within provided safe limits. These run-time monitors can be located in either the TIOP(s) or in independent hardware. Furthermore, monitors have the ability to override commands from the AP should those commands seek to violate the safety requirements of the system. By isolating I/O interaction, formal methods can be applied to verify TIOP functionality, ensuring correct adherence to the rules of operation. Additionally, removing sequential I/O interaction in the AP allows multiple I/O operations to run concurrently. This reduces I/O latency which is desirable in many control systems with large numbers of sensors and actuators. Finally, by utilizing a hierarchical arrangement of TIOPs, scalable growth is efficiently supported. We demonstrate this on a Xilinx Zynq-7000 programmable system-on-chip device.

Enhancing Input/Output Correctness, Protection, Performance, and Scalability for Process Control Platforms

Ryan D. Burrow

(GENERAL AUDIENCE ABSTRACT)

Complex modern systems, from unmanned aircraft system to industrial plants are almost always controlled digitally. These digital control systems (DCSes) need to be verified for correctness since failures can have disastrous consequences. However, proving that a DCS will always act correctly can be infeasible if the system is too complex. In addition, with the growth of inter-connectivity of systems through the internet, malicious actors have more access than ever to attempt to cause these systems to deviate from their proper operation. This thesis seeks to solve these problems by introducing a new architecture for DCSes that uses isolated components that can be verified for correctness. In addition, safety monitors are implemented as a part of the architecture to prevent unsafe operation.

To my family, friends, and all those who have supported me in my educational endeavors.

Acknowledgments

I would like to thank my faculty advisor, Dr. Cameron Patterson, who has guided and supported my work through this entire process. I would not have been able to complete my thesis in the time and to the standard that I wanted without his feedback.

I would also like to thank my committee members, Dr. Patrick Schaumont and Dr. William Plymale. Your knowledge and expertise have helped shape me through my early academic career, and the lessons I have learned classes and research with you will continue to guide me for years to come.

Finally, I would like to thank my CyberCorps advisors, Dr. Joseph Tront, Dr. Kira Gantt, and Dr. Ingrid Burbey, who have guided and supported me through the SFS program and my early career decisions.

Contents

List of Figures

List of Tables

List of Acronyms

AI artificial intelligence

AMBA Advance Microcontroller Bus Architecture

AP application processor

API application programming interface

BRAM Block RAM

BSP board support package

CLB configurable logic block

CPU central processing unit

D.C. direct current

DCS digital control system

FF flip-flop

FIFO first-in-first-out buffer

FPGA field-programmable gate array

FSM finite state machine

GPIO general purpose input/output

IC integrated circuit

I/O input/output

IOI I/O Intermediary

IoT internet of things

IP Intellectual Property

LTL linear temporal logic

LUT look-up table

MCU microcontroller unit

OS operating system

PC personal computer

PID proportional-integral-derivative

PLC programmable logic controller

PSoC programmable system-on-chip

RAM random-access memory

RCF recovery control function

ROM read-only memory

RTA run-time assurance

RTOS real-time operating system (OS)

SDK software development kit

seL4 secure embedded L4 microkernel

SM safety monitor

SoC system-on-chip

TIOP trusted input/output processor

TPM trusted platform module

UART universal asynchronous receiver/transmitter

UAS unmanned aircraft system

UML unified modeling language

List of Listings

List of Algorithms

Chapter 1

Introduction

Digital control systems (DCSes) are present everywhere in modern society. They are used to ensure correct and safe operation in many inherently unstable systems, from unmanned aircraft system (UAS) to critical infrastructure. Furthermore, with the growth of the internet of things (IoT), the presence of DCS, particularly embedded systems, has grown exponentially. IoT typically refers to an embedded device with an interface to the internet, usually for monitoring or control. Some examples of these include the Amazon assistant, Alexa, or the Google home control system, Nest, which can control your thermostat, unlock your doors, and more [8]. In addition, Microsoft, known for their software for traditional computing devices for over 3 decades, are even starting to develop their own embedded IoT devices, the Microsoft Azure Sphere [7]. In March of 2019, UPS announced its plan to begin using UASes to deliver medical samples to hospitals in North Carolina [11].

DCSes are comprised of several key components, including:

- *Application processor (AP)* – Any type of digital processor that is programmed to regulate a system.

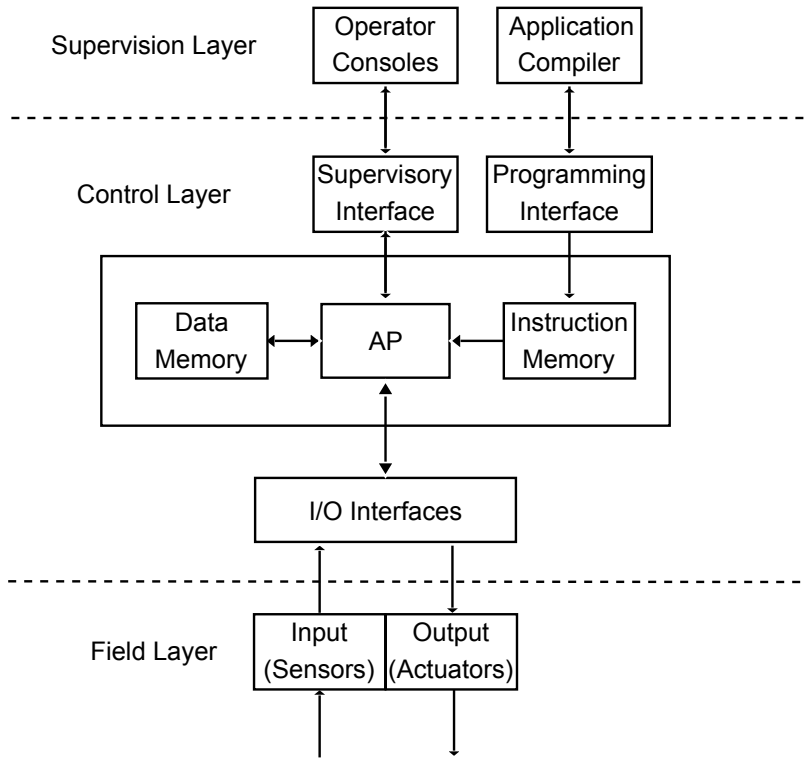

Figure 1.1: Generalized architecture of a DCS

- *Sensors* – Devices that measure the current state of the system being regulated. Data from these sensors are used as inputs to the control algorithm. Some examples include speed, altitude, temperature, or location.

- *Actuators* – Devices that convert a control signal into a physical change in the controlled system. These are used to change the state of the system. Common actuators include motors, valves, and pumps.

Figure 1.1 shows the typical architecture of a traditional DCS. The sensors and actuators together form the input/output (I/O) devices with which the AP must interface in order to run its control loop. There are many types of I/O interfaces that are used to connect the `AP` and the sensors/actuators. There are local network based protocols, such as Fieldbus [1] or Profibus [4], where all devices use the same interconnected medium to communicate.

There are also many point-to-point protocols like universal asynchronous receiver/transmitter (UART) where each device has a unique interface to the AP. DCSes also typically have an external network connection that allows operators to send setpoint commands as well as monitor the system remotely [22] [33]. The network connection connects the `Operator Consoles` to the `Supervisory Interface` which relays sensor and state data from the `AP` to the operator. It also can adjust the AP's setpoint value to change the system's mode of operation. The `Programming Interface` is used by the `Application Compiler` to load new instructions into the otherwise read-only `Instruction Memory` of the AP. This memory contains the executable instructions for the AP. The `Data Memory` contains any variables the AP needs in order to perform its computations and execute its program.

1.1 Objectives

One of the major differences between DCSes and traditional computers is their connection to sensors and actuators, as well as the requirement to operate in real-time. Real-time control does not only focus on the outputs generated by inputs, but also the time required to do so [67]. Sensors and actuators, which can be collectively referred to as I/O devices, interface electronics to the real world. These connections create several key issues which this thesis addresses:

- *Correctness* – Control software is complex and can have thousands of lines of code. Ensuring correct operation under all conditions grow becomes infeasible as the system's complexity and the number of inputs grow.

- *Malware resilience* – Technology, including DCSes, is rapidly becoming more accessible through networking. This gives malicious actors more opportunities to target systems.

In addition, due to their use to control physical processes, incorrect operation of safety-critical DCSes can be costly or even cause loss of human life. For the purposes of this thesis, the threat model observed focuses on attacks directed at the AP itself; mitigating sensor-based attacks is left to future work through the incorporation of redundancies.

- *Performance* – In high-speed DCSes, low latency for I/O communication is critical, as it affects real-time capabilities. Delayed state data can cause the controller function to act on a stale state, leading to inaccurate adjustments to the system.

- *Scalability* – Complex DCSes have hundreds if not thousands of I/O devices. For example, the Airbus A350 has over 50,000 sensors and generates terabytes of sensor data during a flight[19]. Effective architectures must be able to incrementally grow to accommodate additional sensors without incurring additional overheads or becoming infeasible to implement.

1.2 Contributions

In May of 2018, ASTM International, an international standards organization, released "Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions", Designation F3269-17. This standard details design practices that should be followed in order to create safe UAS for civilian/commercial use [15]. These guidelines have been addressed and incorporated into the design process of this thesis.

The fundamental basis of this thesis is an investigation into the effectiveness and potential of utilizing dedicated, isolated trusted input/output processors (TIOPs) to enhance correctness, malware resilience, performance and scalability. An architecture is proposed which isolates

I/O interaction in one or more TIOPs. These TIOPs also communicate with the AP and serve as an intermediary between the controller and peripherals. This architecture meets the objectives outlined in Section 1.1 in the following ways:

- Performance and scalability – The architecture proposed seeks to improve upon existing architectures without adding significant latencies. In fact, by utilizing one or more dedicated TIOPs, concurrency can be introduced to reduce communication latencies. In addition, the architecture proposed promotes scalability without adding complexity by enabling the introduction of secondary TIOPs.

- Isolation and correctness – Incorrect operation of safety-critical systems, whether by the actions of malicious actors or simply incorrect code, is unacceptable. The architecture proposed aims to eliminate these dangers by introducing safety monitors (SMs) which can detect unsafe operation and override commands to prevent failures. In addition, formal verification methods are applied to the TIOPs, ensuring correct operation at all times. This implementation of run-time assurance (RTA), discussed more in Section 3.1, utilizes isolation to allow trusted components to communicate with untrusted components without compromising the correctness of the system.

1.3 Organization

The remainder of this thesis follows the following structure. Chapter 2 provides background information on challenges with DCSes, as well as the platforms available for development. Chapter 3 outlines a high-level overview of the proposed solution and discusses related work. Then, Chapter 4 describes an implementation of this system and applies model checking to verify its correctness. Chapter 5 quantitatively discusses the data collected using the

implementation described in Chapter 4, comparing it to a reference implementation that is functionally equivalent. Finally, Chapter 6 reviews the problem addressed, as well as the proposed solution, and analyzes the effectiveness of the solution in addition to the future potential for improvement.

Chapter 2

Background

2.1 Operation of a Digital Control System

The control algorithm implemented in the AP of a DCS attempts to bring the state of the system to a statically or dynamically defined setpoint, such as the desired speed for a motor. One method of controlling direct current motors is through a voltage adjustment [34], as in Figure 2.1. In this diagram sensor values are read to determine the actual speed of the motor. This value is then compared to the setpoint by the controller, and an adjustment is made to the output voltage. This change in voltage attempts to bring the actual motor speed closer to the desired speed. External disturbances, such as increased friction or a change in load can also affect the actual speed, necessitating the feedback loop. Motor speed controllers are present in many DCSes, motivating their use in this paper.

In contrast to personal computers, which have a complex operating system to run hundreds of processes simultaneously, the AP in a DCS typically follows a simpler model known as cyclic execution. In this model, different tasks correspond to different blocks of code in a

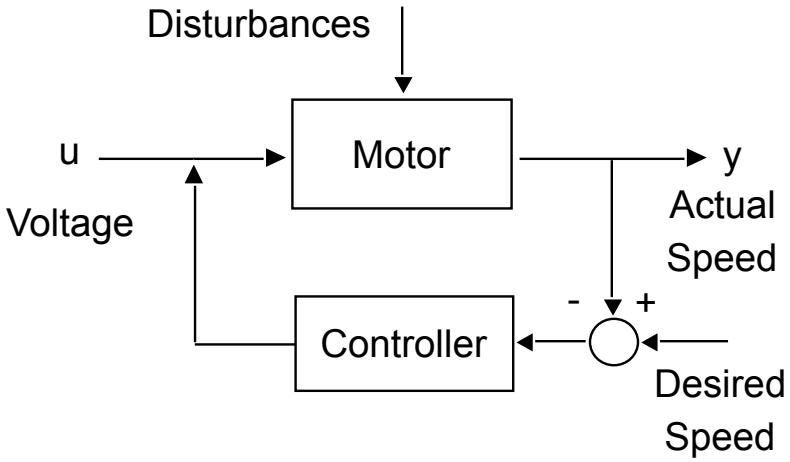

Figure 2.1: A motor speed controller

single overall loop [28]. More modern APs may also use a real-time OS (RTOS), which attempts to complete tasks in repeatedly consistent time periods [56]. They are used to manage several simultaneous control loops, using various scheduling algorithms. Each loop is initiated by a timer or other trigger. These control loops, whether managed by an RTOS or run as a cyclic executive, typically consist of the following steps [28]:

1. *Timer synchronization*: The loop waits for the next periodic timer event.
2. *Read data*: Data is collected from sensors.
3. *Compute error and correction*: The AP (controller) computes the error between the desired state (setpoint) and the actual state determined by the measured variables. It then calculates a correction, which is written to the manipulated variables. This is often done with a proportional-integral-derivative (PID) algorithm [28].
4. *Adjust outputs*: Commands are written to actuators.
5. Repeat from Step (1).

This model of execution is required for real-time operation, which is necessary for DCSes to

maintain stability in the systems they manage. Monitoring and interacting with I/O is one of the most time consuming operations for the controller, and high latencies in this interaction can greatly diminish the real-time capabilities of the system [67].

2.2 Digital Control System Platforms

There are many platforms available for DCSes. The following section is designed to provide some background information about these different systems, as well as their advantages and disadvantages.

2.2.1 Classical Microcontroller Unit

The term microcontroller unit (MCU) typically refers to a chip that follows a standard central processing unit (CPU) architecture but is not as highly optimized. While personal computers or devices provide 32-bit or 64-bit architectures, many MCUs run on as small as 8-bit systems, although some recent high-end chips have begun to use 64-bit architectures. MCUs are packaged on the same chip with peripheral controllers and internal random-access memory (RAM) and read-only memory (ROM), whereas a CPU's memory is usually external.

One of the reasons MCUs have become so popular in the modern era is their low cost combined with their high functionality. Part of this is due to the speed at which these controllers can run. MCUs are not typically as fast as the processors found in conventional computers. Instead, they prioritize I/O performance over computational performance.

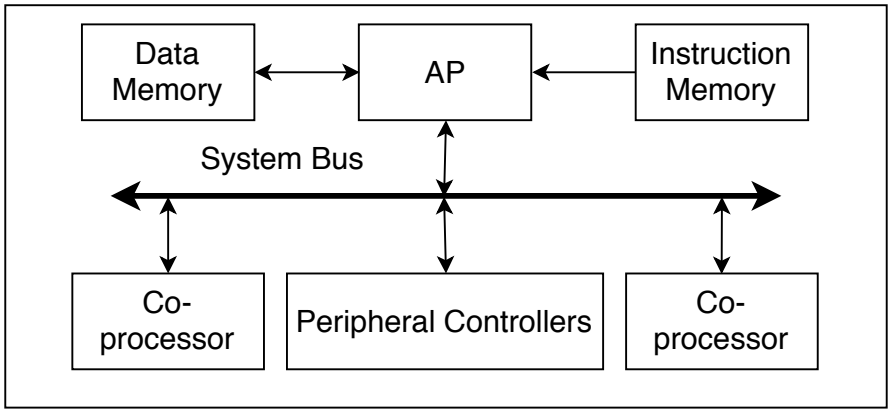

Figure 2.2: Generalized architecture of a SoC

2.2.2 System-on-Chip

As the systems controlled by DCSes grow in complexity and functionality, so do the platforms used to implement them. Figure 2.2 is an example of a system-on-chip (SoC). The SoC has an `AP`, typically a MCU, which has its own `data memory` and `instruction memory`. SoCs are designed to offload some of the operations of the AP to `co-processors`. These co-processors are typically designed to be especially efficient at a unique task, such as floating-point arithmetic or signal processing. Peripherals are the external devices that the controller communicates with, typically through a serial bus. The AP interfaces to these buses through the use of `peripheral controllers`. As the name implies, all components are packaged on a single chip, with the exception of bulk memory.

SoCs have been implemented in many different areas of industry, and they are even starting to arise in the area of IoT. One such device being developed by MediaTek will be able to interface to the Microsoft Azure cloud service [6]. This device uses two co-processors manufactured by ARM as I/O peripheral managers, offloading I/O interaction to them.

Another company using SoCs and co-processors for I/O control is Intel. One of their new

UAS devices, the Intel Aero, combines an SoC as well as an additional MCU to create a development platform for hobbyists and professionals alike. It does this by offloading computationally intensive tasks such as camera data analysis to the SoC, freeing up the separate ARM processor to serve as the autopilot [50].

2.2.3 Field-Programmable Gate Arrays

An alternative platform for DCSes is a field-programmable gate array (FPGA). An FPGA is a chip containing a tiled arrangement of logic resources, which can be configured to implement custom digital circuits; they are essentially hardware that can be rapidly modified to suit the needs of the current situation. This is fundamentally different from an MCU in that an MCU is fixed hardware that provides adaptability through software. In addition, MCUs can typically be run at a faster clock speed than FPGAs, however, their execution is almost entirely serial. Conversely, FPGAs have the ability to provide fine-grained parallelism, which in certain situations can be several orders of magnitudes faster than a sequential execution of the same set of data [25].

Figure 2.3 outlines the structure of an FPGA, which is made up of several components:

- *Configurable logic block (CLB)* – The basic building block of an FPGA, a CLB is made up of several discrete components such as *flip-flops (FFs)*, which is used to store state information. *Look-up tables (LUTs)* are custom truth tables that hold unique values at boot time. CLBs can be configured as various `logic` blocks, as shown in Figure 2.3
- *Block RAM (BRAM)* – The dedicated `memory` blocks found in Figure 2.3. They can be configured to various depths and widths.
- *I/O blocks* - The means by which the FPGA transmits and receives any sort of data

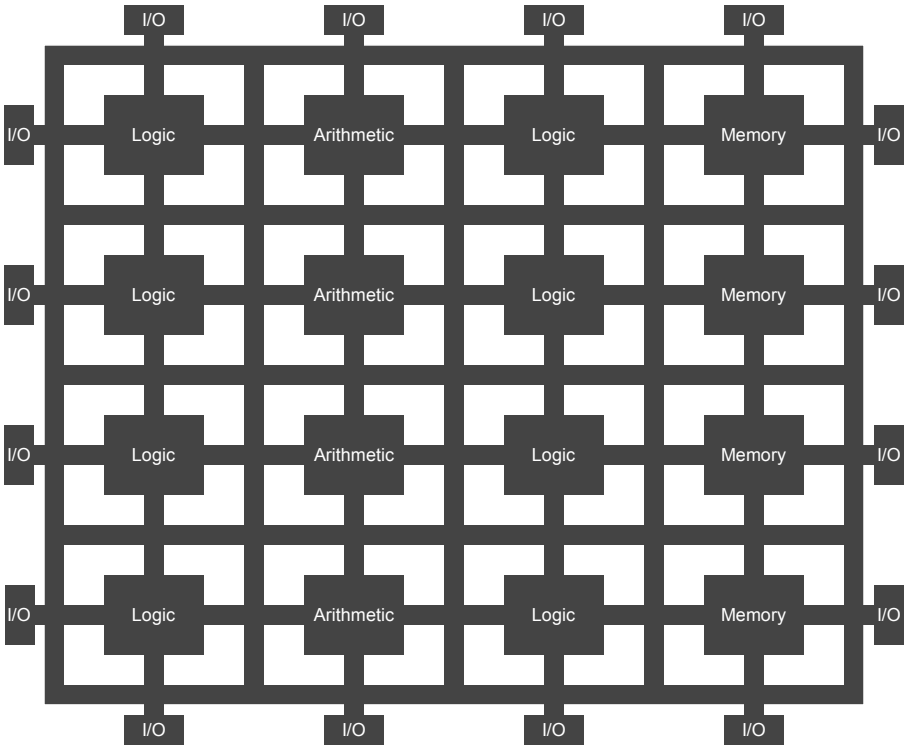

Figure 2.3: Structure of an FPGA

to or from external components. These can be configured as inputs, outputs, or bi-directional.

- *Arithmetic blocks* – Specialized components to efficiently perform digital signal processing functions more efficiently than CLBs. These are represented by the `arithmetic` blocks in Figure 2.3.

- *Interconnects* – System resources that can be reconfigured to connect any desired components. These are represented in Figure 2.3 by the lines connecting all blocks [29].

As mentioned in Section 2.2.2, it can often be advantageous to offload work from a processor to co-processors. These co-processors can range from basic hardware multipliers to even a secondary MCUs. FPGA implementation may consist of designing several hardware co-

processors, and one of the main advantages of the reconfigurable logic is that co-processors can be quickly designed and optimized to suit a specific task. For example, in recent years, Microsoft has started implementing neural network-based Bing search engines using Intel FPGAs. These are able to do image recognition and searching at much faster rates than traditional implementations [5].

One use of an FPGA is to create what is known as a softcore processor. These are a specific type of hardware co-processor which mimics an MCU in that they can execute instructions, although this instruction set is often much more simplified than that of a "hard" core processor. These are very often suited to situations where basic processing is required that is overly complex to solve using hardware-based logic. One of the major advantages of these reconfigurable processors is the ability to create several that can each be dedicated to a unique task, thereby parallelizing the processing of data [52].

One disadvantage of FPGAs is they typically have higher power consumption than their counterparts due to the overheads incurred by configurable interconnect, although they may reduce power relative to MCUs for certain applications with high streaming data rates by removing instruction processing overheads [35]. This can be reduced through careful design and implementation, but this often comes at the cost of speed or concurrency, a tradeoff that is not ideal in real-time systems. For many industrial DCSes additional power consumption is a relatively minor cost, but for robotics and UASes addition power consumption has a larger impact. Either usability time is reduced since batteries are depleted faster, or additional power sources are needed, which adds weight and cost to the system.

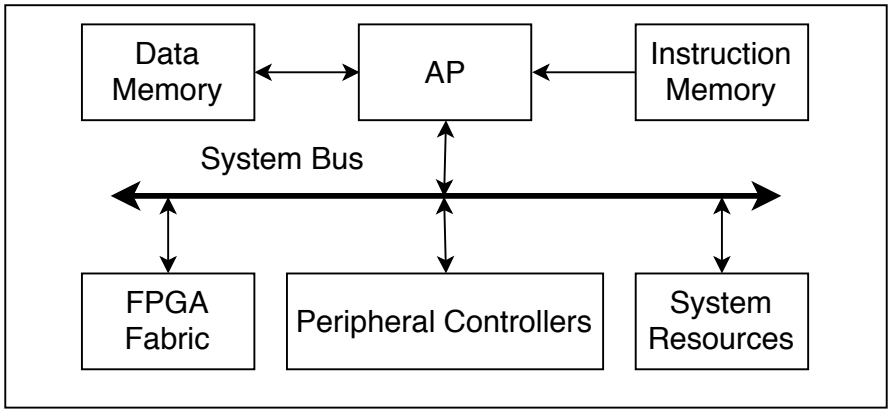

Figure 2.4: Generalized architecture of a PSoC

2.2.4 Programmable System-on-Chip

A more recent platform for DCSes is the programmable system-on-chip (PSoC) like the one seen in Figure 2.4. One of the major benefits of a PSoC is that they combine an FPGA and MCU together in one system. It is packaged with `peripheral controllers`, as well as a `system bus` which connects the different components. These factors combined enable the entire system to be rapidly prototyped and developed as one functional unit. Like the SoC, all of these components are packaged together in a single chip.

A PSoC is really just an adaptation to an SoC, replacing many of the integrated circuit (IC) co-processors with `FPGA fabric`. This makes the co-processors reconfigurable, while still providing the offloading capability of an SoC. In addition, the PSoC also provides the `AP` in the form of an MCU that can be missing in a pure FPGA implementation. Like the SoC and MCU, the AP has its own dedicated `data` and `instruction memories`. The system clock and external memories, found in `system resources`, can be configured so that different components have access to them, or can be passed through interfaces instantiated in the FPGA fabric to add functionality.

For many prototyping, design, and highly configurable systems, a PSoC provides many advantages to alternative systems. One such example of this type of system is the Xilinx Zynq-7000 [64]. This PSoC combines the configurable hardware of an FPGA with the processing power of a 32-bit ARM MCU, as well as bidirectional data transfer buses that allow for easy communication between hardware and software.

2.2.5 Comparing Platforms

While each new controller option has created additional functionality and improvements, there are appropriate uses for each type of device. Table 2.1 outlines the advantages and disadvantages of each platform in further detail.

Controller	Advantages	Disadvantages
MCU	High speed, ideal for simple cyclical programs Can use RTOS to support multiple control loops	Lack of options for optimizing processing with additional hardware
SoC	Provides additional hardware to optimize functionality	Hardware is fixed and may not be ideal for application Boards are often specific to the application
FPGA	Configurable hardware	Lacks hard processor to execute complex software Higher power consumption
PSoC	Configurable hardware + MCU	Higher power consumption

Table 2.1: Advantages and disadvantages of different DCS platforms

2.3 Issues with Digital Control Systems

The network connection present in most DCSes provides an attack vector for malicious actors and malware. Due to their use in industrial and infrastructure applications, deviation from

a stable operation can have catastrophic consequences. This problem is best illustrated by Stuxnet, a computer worm which reprogrammed programmable logic controllers (PLCs) responsible for controlling industrial centrifuges [44]. The new programming caused the centrifuges to rapidly vary their speed until they destroyed themselves. More recently, U.S. nuclear power plants, a New York dam, and the Ukranian power grid have all been targets of cyber attacks [17]. An unnamed critical infrastructure facility was also hacked, which forced the plant to cease operations [30]; further details about the location of the attack are not available for security reasons.

In high-speed DCSes, the AP may run at speeds several orders of magnitude faster than the I/O devices with which they interact in order to maintain real-time operation. This interaction typically involves interrupt handling and context switches, which require the AP to save its current state in order to execute a new set of commands, and adds significant latency overheads. In many complex control devices, such as robots or aircraft, PID controllers are employed to maintain stability [45]. Control models assume that the control loop steps (reading measured variables, performing calculations, and writing to manipulated variables) occur instantaneously, and any significant deviation from this assumption affects the control system's responsiveness to disturbances. The most important factor regarding speed in these systems is not throughput, but rather latency; data that is too old could cause the controller to respond to a stale system state.

As the number of I/O devices grows in a DCS, so does the complexity of the software required to control them, along with the overheads of communication. This causes many approaches which work at a small scale to be infeasible in large-scale systems.

2.4 Formal Methods

As mentioned previously, malfunctioning AP software on a DCS can have catastrophic consequences. To ensure correct operation of safety-critical software, a technique known as formal verification is sometimes used. Two commonly used methods of formal verification are *static analysis* and *runtime verification*. One form of static analysis is model checking, where the objective is to create a model of the system, formally define expected behavior, and systematically prove or disprove that the system always adheres to that behavior [20]. Model checking becomes infeasible if the model is too complex [24]. The cause of this is state explosion, which occurs when too many states cause model checking to require excessive time and/or memory. Interrupts in a process complicate the control flow of the program and can add many more states. Model checking can suit multi-threaded and distributed systems, as well as the communication between them. Runtime verification takes a different approach to ensure correctness. Instead of statically analyzing software, the verification tool actively monitors the software while it runs and ensures that it does not attempt to violate predetermined constraints [18].

Runtime verification also has the ability to ensure specific time intervals are maintained between events, such as a UAS holding its position for 30 seconds prior to continuing on its path. This is a fundamental difference between model checking and runtime verification; while model checking interprets time as an always advancing sequence of events. Both model checking and runtime verification are facilitated by the architecture discussed, however, this paper concentrates on the application of model checking to TIOP software, while runtime verification is left to future work.

Formal methods require an assumption of trust at some level. For this thesis, the tools used to test implemented models and to perform code translations are assumed to be correct.

They are open source and for years have been thoroughly vetted by academics who are well versed in formal methods. In addition, the compiler used to generate the binaries that run on the TIOPs is also assumed to be correct, otherwise model checking applied to code implementations may be invalid. Finally, for the purpose of this thesis, it is assumed that the model of the system and the safe states are correct. These would need to be designed and thoroughly vetted by a system expert in a more complex system.

Chapter 3

High-Level Design

I/O processors are becoming more prevalent in the embedded systems controller world. The main functionality of an I/O processor is to offload I/O, rather than bogging down the main processor with this slow task. Due to the nature of most I/O devices, most MCUs have to waste several thousand cycles waiting for data to be ready, or they have to have some sort of interrupt to be able to continue processing other data. Adding these interrupts makes any attempts at formally verifying the code executing significantly more difficult, if not impossible.

There are several examples of I/O processors making appearances in industry. One such example is the Zynq UltraScale+, which has a quad-core 64-bit application processor which is complemented by a dual-core, 32-bit, real-time I/O processor [65]. Another example of I/O processors is the aforementioned MediaTek embedded device [6].

Dedicated isolated processors are not just used for I/O processing though. There are several examples of isolation of trusted modules which are used for various situations. The Apple *Secure Enclave Processor*, present in most modern iPhones, is an example of this [40]. This

processor is used to authenticate a user prior to making purchases, using FaceID or TouchID. This processor is similar to what is often referred to as a trusted platform module (TPM). TPMs are used for cryptographic functions, often used to validate firmware or software to be run on the main processor [2]. They have a isolated boot sequence and cannot be modified by the main processor. This creates a root of trust which can be used to identify a compromised system. By using an isolated dedicated processor, TPMs are able to better protect trusted functions from malicious attacks.

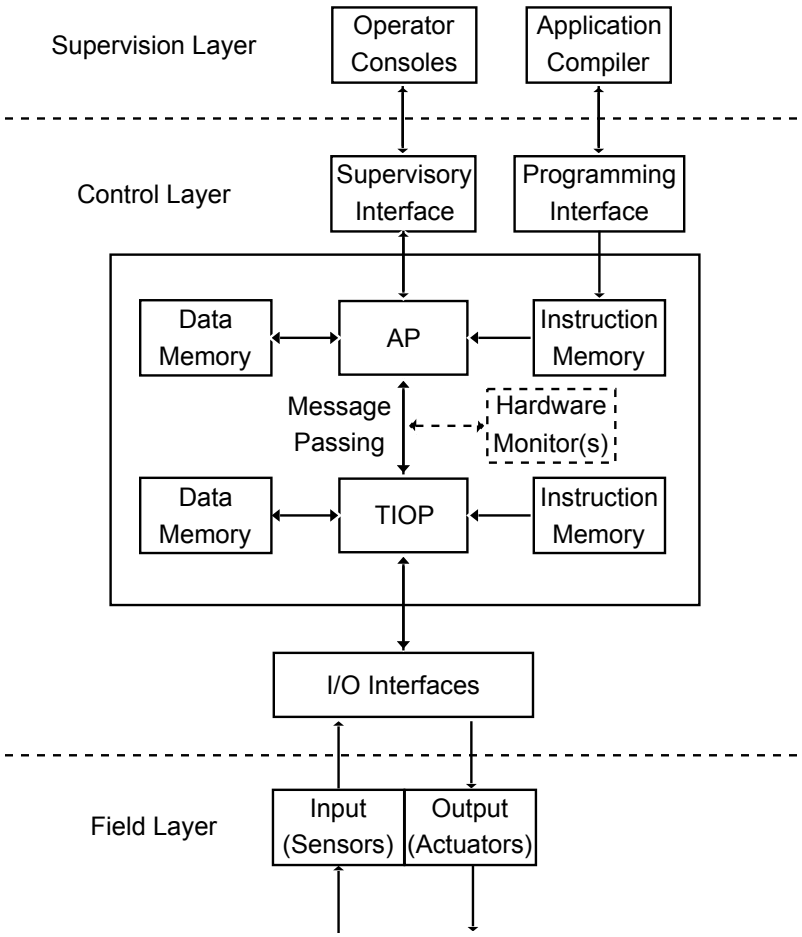

Figure 3.1: A modified DCS which uses an TIOP

Figure 3.1 outlines a modification to the classical DCS architecture shown in Figure 1.1. This architecture aims to isolate I/O interaction to one or more trusted I/O processors (TIOPs).

The `TIOPs` are very similar to the `AP` but are dedicated to a specific, low-level task: I/O interaction. The AP now gets sensor data from the TIOP and sends actuator commands to the TIOP rather than communicate directly with the I/O peripherals. The algorithms that run on the TIOPs are much simpler than interrupt-driven I/O driver software which would reside in the AP. Because the TIOPs have their own `instruction` and `data memory`, they are truly isolated from the AP and each another. In contrast to the AP, the TIOPs have no network connection and cannot be accessed remotely. This combination enables formal analysis to be applied to the TIOP, verifying the correctness of these devices, and by extension, all interaction with I/O devices. In addition, this architecture supports RTA, which will be discussed further in Section 3.1. The `hardware monitor` seen in Figure 3.1 is an optional component of this RTA architecture.

3.1 Run-Time Assurance

In May of 2018, ASTM International, an international standards organization, released *Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions*, Designation F3269 - 17 [15]. This standard contains guidance for designing a run-time assurance (RTA) compliant architecture for UASes. RTA requires several key components: safety monitors (SMs), recovery control functions (RCFs), a complex function and an RTA switch. Figure 3.2 illustrates an architecture that supports RTA. The `RTA inputs` block is comprised of the critical system sensors. These inputs are passed to the complex function, SM and RCF. The `complex function` is the controller code that dictates system responses in normal operational modes. Depending on the system, this code may be too complex for the application of formal verification methods. The `safety monitor` is system-specific and should be designed by an expert who is well versed in the

operational modes of the DCS. It analyses the RTA inputs and determines whether the system is approaching an unsafe state. If the system is stable, the SM uses the `RTA switch` to allow the complex function to continue controlling the system. However, if the SM determines that the system is becoming unstable, it attempts to mitigate this by activating one of the `RCFs` through the RTA switch. The RCF(s) are simpler control functions that are also system-specific and are designed to stabilize the system by moving towards a predetermined safe state. For example, a UAS could be implemented on an architecture supporting RTA to ensure it follows a pre-planned flight path or lands at one of several "safe zones". Under normal operation, the complex function running on the AP would ensure the UAS adheres to its flight plan. However, if the AP was somehow compromised or began to otherwise malfunction, the SM may try to initially activate an RCF to keep the UAS on its flight path. If unable to maintain this, a secondary RCF could be activated to force the UAS to land at a safe location, preventing damage to property or human injury.

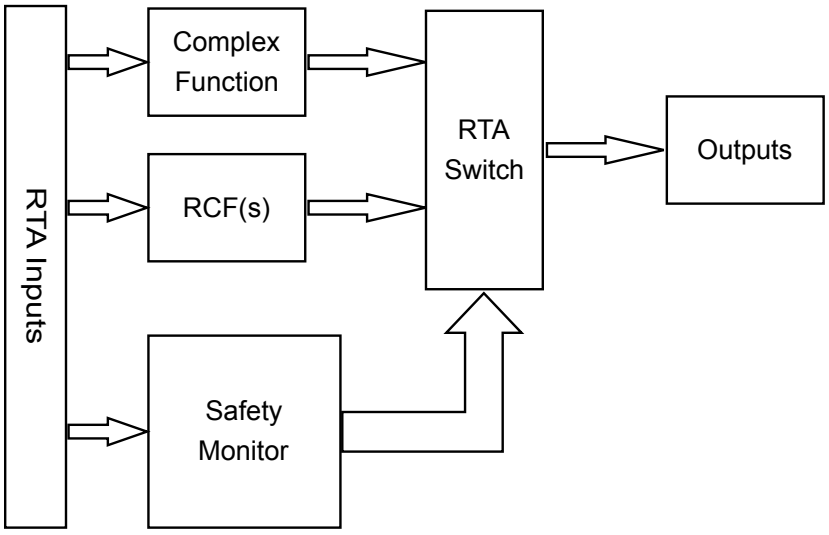

Figure 3.2: An architecture supporting RTA

One benefit of implementing an architecture which supports RTA through isolation is that

untrusted components can communicate with trusted components without compromising the integrity of the system. This is because the SMs are able to override the untrusted components with the RTA switch if they try to violate the constraints of the system, while the isolation prevents the SMs and RTA switch from being compromised. In the architecture proposed the complex function is implemented in the AP, while the SM, RCF, and RTA switch are implemented in formally verified TIOPs. The implementation of these TIOP ensures the adherence to requirements 5.1.2.1 and 5.1.2.2 of the ASTM guidance [15], which state:

- "The UAS manufacturer shall partition the complex function from the safety monitor, RTA switch, and recovery control function(s)." — The complex function is implemented in the AP, while the SM, RCF, and RTA switch are implemented in the TIOP(s).

- "The UAS manufacturer shall determine the hardware and software partitioning requirements based on the results of a system safety analysis, taking into account the likelihood of common mode failures and the hierarchy of response." — The RTA components are implemented in software in this architecture, however, I/O interaction is isolated through the implementation of the TIOPs. This enables these components to be instantiated in dedicated hardware instead, as illustrated by the hardware monitor(s) shown in Figure 3.1. Section 6.1 discusses these modifications in more detail.

3.2 Architecture and Variants

Figure 3.3 illustrates an architecture that has no TIOPs and is representative of a classical DCS. The AP communicates directly with the sensors and actuators through one or more

serial interfaces. These serial interfaces require the AP to wait repeatedly for outgoing data to finish transmitting before sending more, and must regularly be re-checked for incoming data. These blocking waits can be alleviated with interrupts, which alert the processor to these events. The processor then pauses its current task, handles the I/O, then resumes its original task, preventing busy waits. However, these interrupts add significant complexity to the code control flow since they can happen at any point. This complexity grows with each additional serial interface. An architecture like this was implemented as a reference to determine the effect TIOPs have on the system.

Figure 3.3: A block diagram of the implementation without a TIOP

Figure 3.4 introduces a TIOP to the communication protocol. This TIOP communicates directly with I/O devices and the AP, passing information between the two. This greatly reduces the complexity of code in the AP since the message passing protocol is much simpler and faster than multiple and possibly diverse serial interfaces. In addition, since the TIOP's primary task is to interact with I/O devices, busy waits are acceptable. This removes the need for interrupts and keeps the TIOP code simple which makes model checking more feasible.

The architecture shown in Figure 3.5 has a primary TIOP which communicates with the AP and delegates I/O interaction to one or more secondary TIOPs. By distributing this communication across several secondary TIOPs, I/O interaction can be parallelized through concurrency and latencies can be reduced. In addition, by reducing the number of serial

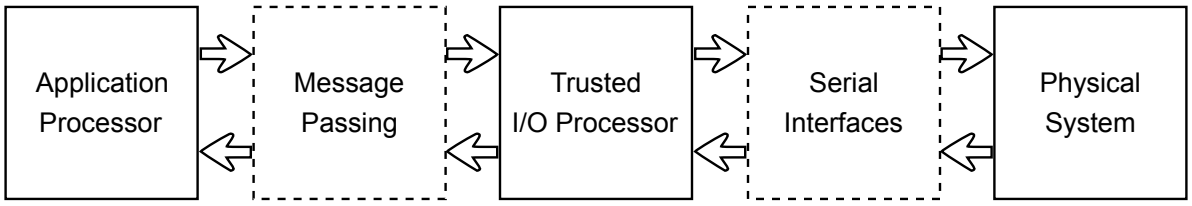

Figure 3.4: A block diagram of a system utilizing a single TIOP

interfaces connected to each TIOP, individual TIOP code complexity is again reduced. Although the system grows in components, the complexity of each component is reduced and the overall complexity of components required for RTA is reduced.

Both Figure 3.4 and Figure 3.5 are implemented in Section 4.1. The single TIOP architecture is the primary focus of the study; formal verification methods are applied to both its model of execution as well as its implementation. A multi-TIOP architecture is implemented to evaluate the effects of introducing secondary TIOPs, however, this architecture is not formally verified which is left to future work.

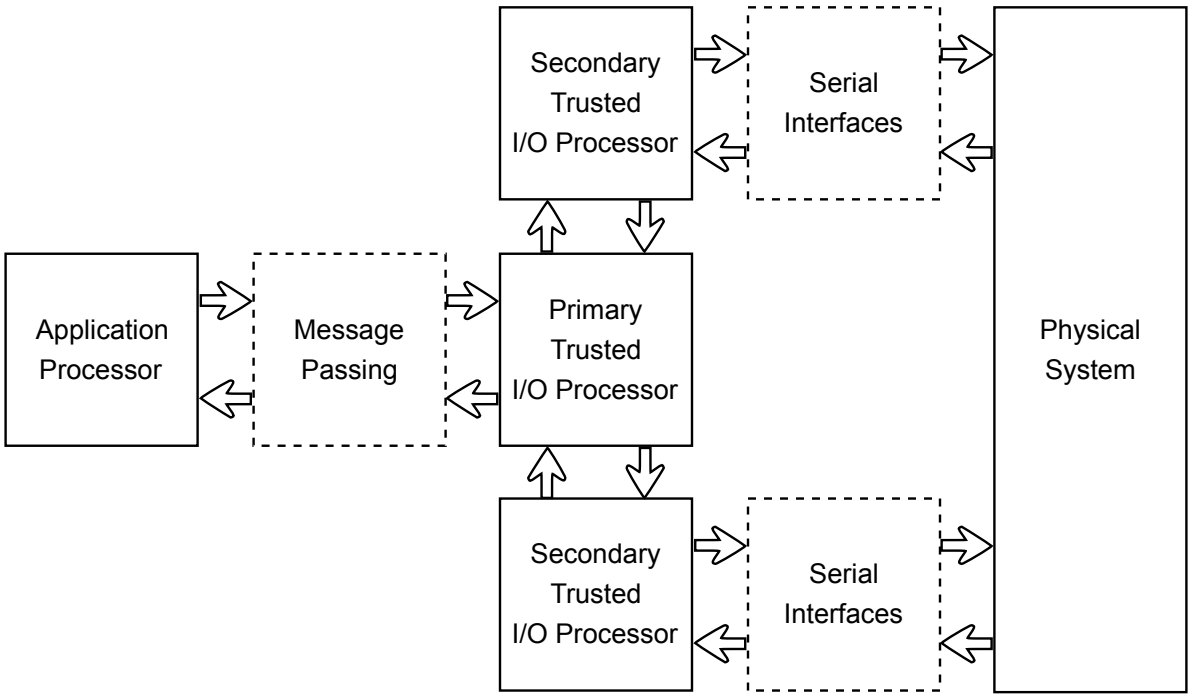

Figure 3.5: A block diagram of a system utilizing multiple TIOPs

3.3 Related Work

Kekec et al. propose a DCS architecture that allows for modular software and hardware in UAS design [42]. Additionally, Deng et al. describe an architecture for UASes that can be rapidly reconfigured depending on the requirements of the system [26]. Both of these proposals aim to abstract hardware and software to increase modularity and flexibility. However, neither system addresses security. TIOPs offer both modularity and flexibility and mitigate AP malware. With the use of a common message passing protocol to communicate between the AP and TIOPs, the AP need not be aware of the specific serial protocols used to interface with the I/O devices. In addition, new I/O devices can be added with new TIOPs (e.g. to add sensor redundancy with cross-checks performed in a TIOP) without impacting the AP or its software.

Chiluvuri et al. propose a DCS architecture for safety-critical systems known as TAIGA [22] [21]. This architecture describes an I/O Intermediary (IOI) softcore processor that reads sensors and writes to actuators in a similar manner to TIOPs. This IOI uses conventional interrupt-driven I/O rather than the TIOP's polling. TAIGA also has an RCF, although this is implemented as a backup controller instantiated as another softcore processor. Similarly, Kini creates a UAS architecture that uses a PSoC platform to implement isolated I/O co-processors [43]. These co-processors do not implement a monitor nor an RCF, but only validate sensor data. Neither of these I/O subsystems are formally verified, unlike TIOPs.

Many approaches to try to make systems resilient to malware and malicious actors. One modern method of achieving this goal is through the use of verified, isolated hardware that serves as a root of trust for the rest of the system. An example is the *Secure Enclave Processor*, which is present in most modern Apple products [40]. This isolated processor is validates a user's identity through Face ID or Touch ID. The software running on this processor has a separate secure boot process and is independent of the AP [46]. This prevents software vulnerabilities on the main processor from affecting the device's ability to accurately authenticate a user for purchases or other secure interactions. Using isolation to enhance malware resilience is similar to the use of TIOPs. While the secure enclave safeguards information, TIOPs safeguard a physical process.

Ensuring the correct functionality of software running on DCSes is especially important because they protect physical processes that may be part of critical infrastructure (e.g. power plants). Microsoft Windows used to be vulnerable to crashes arising from errors in device drivers. As a result, Microsoft required all device drivers to be formally verified through static analysis [39]. This has significantly reduced the number of crashes in newer versions of Windows [36]. Device drivers are suited to formal analysis, as they are usually structured as state machines. TIOPs are also well suited to this kind of analysis since the

algorithms that run on them have a simple state machine structure.

Formal verification methods do not need to be applied to an entire system, and can instead be applied to critical subsystems. This has been shown in the secure embedded L4 microkernel (seL4) project. The seL4 project is a family of microkernels verified using theorem proving in a way that allows the rest of the system to remain untrusted [48]. seL4 does not include device drivers as they would have complicated the model, increasing the verification difficulty. Instead, the drivers are implemented as conventional application tasks. Similarly, our DCS architecture partitions the system to allow the TIOPs to be formally verified. Rigorous verification of device drivers is facilitated by implementing them on isolated and dedicated processors. In addition, formally verified monitors can check and possibly override the messages passed between the AP and primary TIOP, as shown in Figure 3.1. This partition allows the software on the AP to be untrusted while still offering overall system assurances. With the growing use of AI and machine learning in autonomous systems [19], the need to rigorously analyze these systems is growing. Because machine learning algorithms can react unexpectedly if they encounter scenarios not included in the training data, it is desirable to add run-time monitors that can invoke RCFs.

If the physical system state can change quickly, the DCS requires low latency to avoid reacting to stale state data. One of the biggest causes of DCS latencies is overheads associated with communication between devices, which can be further increased by security measures. The Ptolemy tool is used to model real-time systems [49] and is useful when designing DCSes since the time required to perform a task affects the correct functioning of the system [45]. One of the most important considerations is ensuring that TIOPs do not introduce significant latencies into the system. In fact, TIOPs may even reduce latencies relative to the reference implementation.

Stamenkovich et al. explore utilizing hardware monitors for autonomous systems [55]. First,

formal specifications are captured utilizing linear temporal logic (LTL) formulas. These formulas are then converted to an infinite input state machine. This state machine is translated to C code, followed by the application of High-Level Synthesis which transforms the C code into digital circuits. The digital circuits are instantiated in an FPGA and are checked for correctness during each step of translation, ensuring the final implementation matches the original specification.

Chapter 4

Implementation

4.1 Architecture

The goal of this research is to provide guidelines for creating a DCS platform on a PSoC that can reliably adhere to a set of predefined rules while still providing the ability to run unverified applications with minimal modifications. Three architectures are implemented and tested in this thesis. The first uses no TIOPs and is used as a reference for examining the effectiveness of the TIOPs. The second architecture introduces a single, formally verified TIOP which implements the SM and RCF while communicating with the AP and I/O devices. The third architecture adds a secondary TIOP. Secondary TIOPs handle I/O while the primary TIOP communicates with the AP and implements the SM and RCF. Timing and resource usage information are extracted from this implementation to quantify the overheads of secondary TIOPs and determine scalability.

4.1.1 Platform and Tools

The platform selected for implementation is a Xilinx Zynq-7000 PSoC [64]. This chip contains an FPGA with 6.6 million logic cells and a 32-bit ARM MCU, as well as bidirectional data transfer buses that allow for easy communication between hardware and software. The PSoC is mounted on the Digilent Zybo Z7-20 development board [27]. This development board provides the hardware for connecting external peripherals, such as the UART-to-serial devices used to interface the board to the MATLAB simulated motor. The FPGA is programmed using the Xilinx tool Vivado [12]. This tool is used to create and connect different Intellectual Property (IP) blocks, such as UART blocks, general purpose input/output (GPIO) blocks, or softcore processors. The softcore processors utilized in these architectures are Xilinx MicroBlazes [52]. The IP blocks are connected using the Advance Microcontroller Bus Architecture (AMBA) AXI bus protocol [9]. This protocol is a multi-master multi-slave protocol which is designed for high speed communication between connected devices in an embedded system environment.

4.1.2 Design

In all three architectures, shown in Figures 4.1, 4.2, and 4.4, the AP is connected to GPIO pins (`axi_gpio_0`). These pins are used to collect timing data by identifying the current state of the control loop running in the AP. All architectures also have two `AXI Uartlite` IPs, which implement the UART protocol. `axi_uartlite_0` is used for transmitting actuator commands, while `axi_uartlite_1` is used for receiving sensor data. The TX pin of `axi_uartlite_0` is connected to two output pins, allowing the transmission of commands to be more easily monitored when collecting timing data. Timing data is discussed more in Section 5.2.

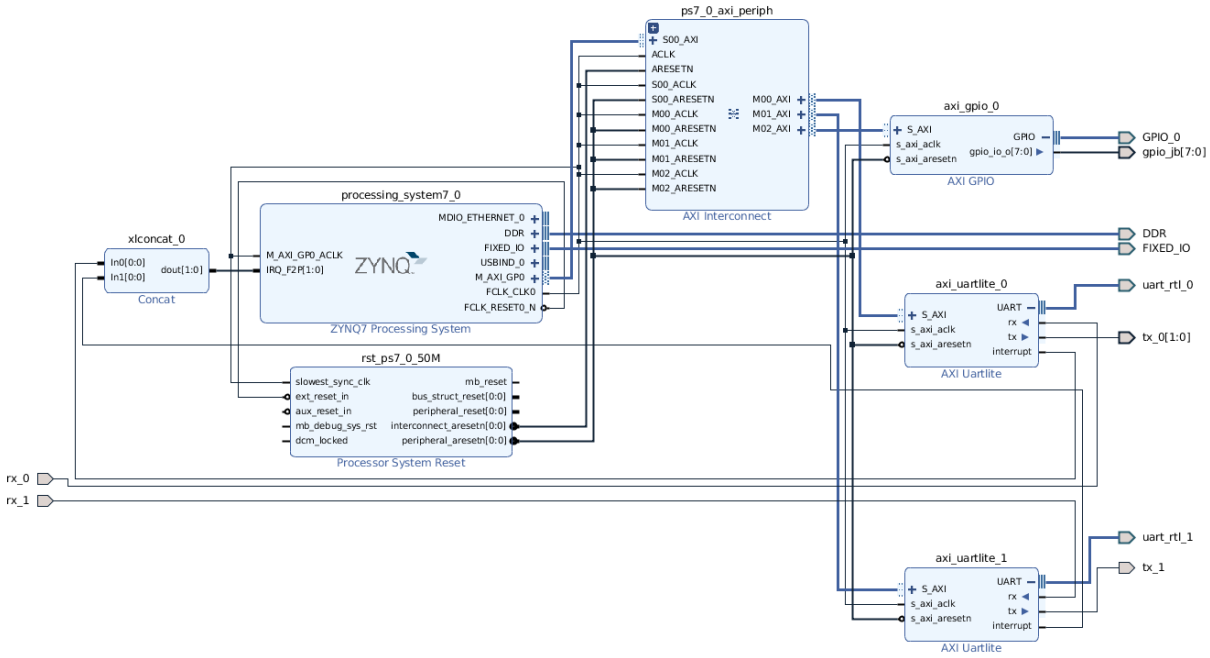

Figure 4.1: Vivado block diagram for the TIOP-less architecture

Figure 4.1 is the Vivado block diagram of the reference architecture which uses no TIOPs. The ZYNQ7 processing system (`processing_system7_0`) is the hardware block containing the AP, its interfaces, and the system clock. The AP is directly connected to the two UART devices. These connections are made via an `AXI Interconnect` IP (`ps7_0_axi_periph`) which allows two devices to communicate data using the AMBA AXI bus protocol [62].

In the early stages of development, sensor and actuator control used GPIO. However, very few modern DCSes use direct I/O interaction and most communication goes through a serial bus. The UART protocol is a relatively simple serial communication interface and is used by many devices. There are also many UART to serial interface boards available, which simplified interfacing to the UART device with a computer to generate and receive data. A UART was chosen for these reasons.

Xilinx MicroBlazes are used to implement the TIOPs [52]. The AP and TIOPs communicate through `AXI-Stream first-in-first-out buffer (FIFO)` IPs. These streaming FIFOs

allow for large amounts of data to be passed between two devices quickly with little overhead. These FIFOs can be bi-directional, but for simplicity and isolation two independent sets are implemented, one for sensor data (`tx_data_buf` and `rx_data_buf`), the other for actuator commands (`tx_cmds_buf` and `rx_cmds_buf`). Xilinx `Mailbox` IPs were another option that was considered for communication between the AP and TIOPs [63]. These devices were used by another implementation of I/O processors [43]. They function in a very similar manner to FIFOs but add some unnecessary complexity in configuration. It was decided that generic FIFOs would suit the task just as well. It was also simpler to configure the FIFOs and to interface to them.

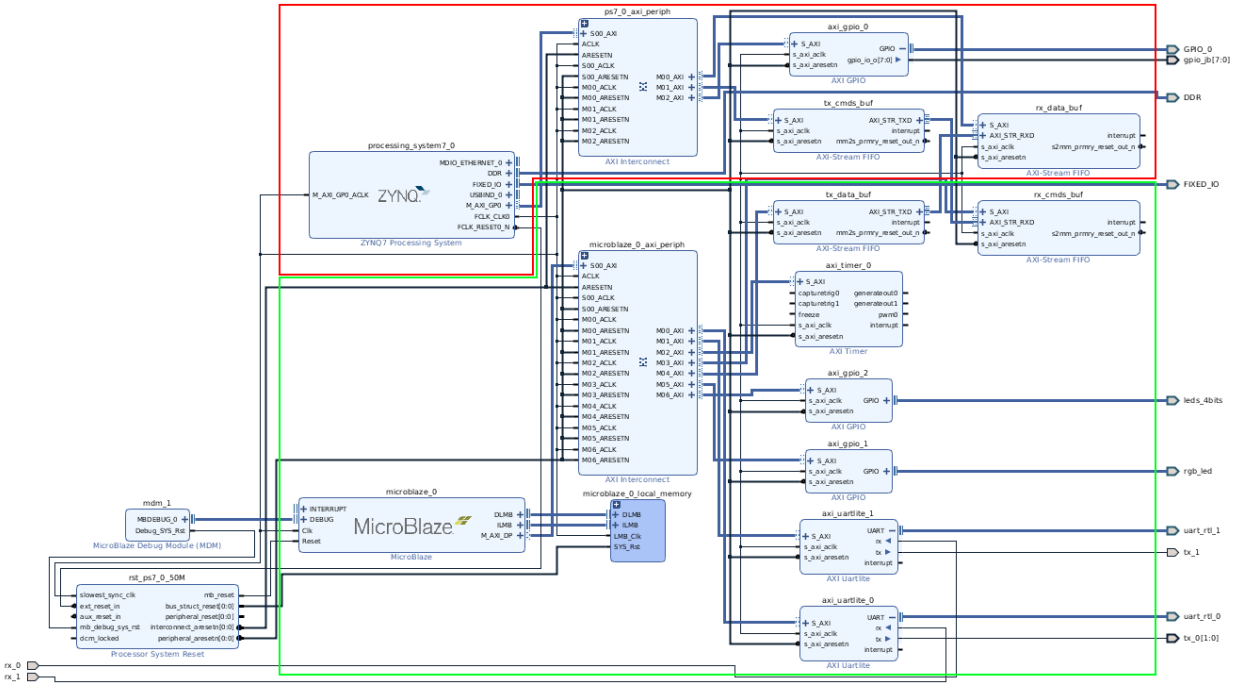

Figure 4.2: Vivado block diagram for the single TIOP architecture

In Figure 4.2 a single TIOP is introduced to the architecture. All devices that are included in the green boxed area are connected to the TIOP (`microblaze_0`). The red box illustrates the devices that are connected to the AP. The Uartlite devices have no connection to the AP and instead are connected to the TIOP through a second AXI interconnect

(`microblaze_0_axi_periph`). This TIOP has its own set of instruction and data memory and is also connected to two new GPIO devices (`axi_gpio_1` and `axi_gpio_2`) which are used for debugging purposes. The TIOP is also connected to an `AXI Timer` IP (`axi_timer_0`) which is used to implement a watchdog timer, discussed more in Section 4.2.2. FIFOs are used to connect the TIOP and AP. To communicate sensor data the TIOP writes to `tx_data_buf`. After finishing these writes, the FIFO transmits the data to `rx_data_buf` where it is available for the AP to read. The same process is reversed when the AP needs to write actuator commands, but `tx_cmds_buf` and `rx_cmds_buf` are used instead.

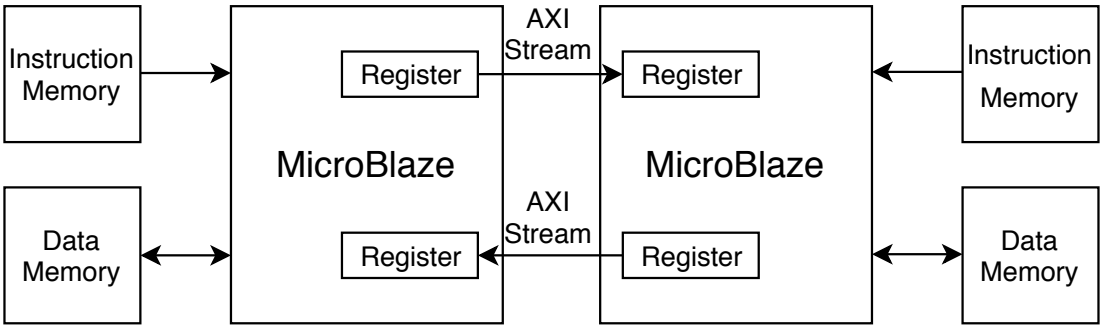

Figure 4.3: AXI-Stream interface between two MicroBlazes

Figure 4.3 outlines communication between two MicroBlazes through the use of an AXI-Stream interface. This interface uses direct transfer between registers in the two MicroBlazes and has no need for shared memory or an associated mutual exclusion mechanism. Communication on this interface can be non-blocking or blocking. The multi-TIOP architecture implemented below uses the blocking method which causes the receiving processor to stall until data comes available.

Figure 4.4 illustrates the final architecture. Here a secondary TIOP is added, again with its own instruction and data memory, as well as its own AXI Interconnect. The secondary TIOP (`microblaze_1`) and its connected peripherals are outlined in the new blue box. The AP is

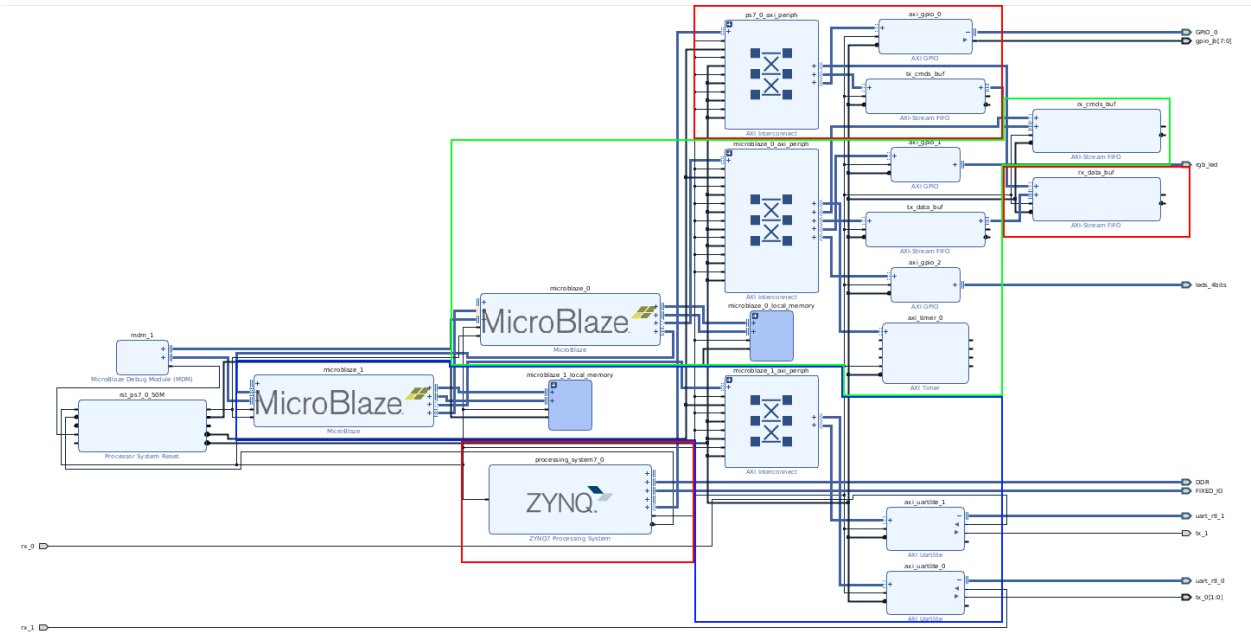

Figure 4.4: Vivado block diagram for the multi-TIOP architecture

again outlined in red and the primary TIOP is still outlined in green. In this architecture, the UART devices are connected to the secondary TIOP, while the communication via FIFOs remains between the AP and primary TIOP. To exchange sensor data and actuator commands between the primary and secondary TIOP, a high-speed `AXI-Stream` interface is used [59]. All other connections, including GPIO and the AXI Timer, remain the same as in the single TIOP architecture. This architecture could be scaled to include more secondary TIOP as needed, with each new processor interfacing to a unique serial interface. Each additional secondary TIOP would also need a new connection to the primary TIOP, which can support up to 16 AXI-Stream interfaces [59].

Section A.1 provides step-by-step instructions to recreate any of the desired architectures in Vivado.

4.2 Algorithms

4.2.1 Application Processor

Algorithm 1 Motor control algorithm

```
1: init_timer()
2: init()
3: while true do
4:    wait_period()
5:    recv_data()
6:    pid()
7:    send_cmds()
8: end while
```

To determine the effectiveness of the TIOP a motor controller is implemented on the AP using a modification of a floating-point PID controller provided by Xilinx [16]. The measured variables are generated using a modification of a MATLAB script that accompanies the controller code, and the manipulated variables are compared to their expected values to check whether the algorithm functions as expected. This algorithm performs a series of application programming interface (API) calls, as shown in Algorithm 1. While the implementation of the API calls is specific to the architecture being used, the function summaries are:

- `init_timer()`: Initializes a periodic timer which used by the `wait_period()` function.
- `init()`: Initializes the communication devices used by the implementation. In the two examples used, these are the UART drivers and file I/O or a pair of buffers implemented in the FPGA fabric.
- `wait_period()`: Returns once a time period has elapsed.
- `recv_data()`: Retrieves sensor data from the interface that has been implemented. If utilizing a TIOP this gets data written to the FIFO by the TIOP's `send_data()`

function.

- `pid()`: Runs a PID control algorithm on the provided data and returns the output values for converging to the desired state.

- `send_cmds()`: Transmits actuator commands on the implemented interface. If a TIOP is implemented, it retrieves the commands written to the FIFO with its `recv_cmds()` function.

`recv_data()` and `send_cmds()` are used as setpoints on the program to collect timing data and compare the latencies of different implementations, as shown in Section 5.2. A guide to working with PetaLinux and programming the AP can be found in Section A.2.

4.2.2 Trusted Input/Output Processors

The basic control flow of the TIOP in a single TIOP architecture is shown in Algorithm 2.

Algorithm 2 Single TIOP routine

```
1: init:
2:     Initialize UARTs uart_data, uart_cmds
3:     Initialize buffers data_buf, cmds_buf
4: while true do
5:     recv_data:
6:         data ← uart_data
7:     data_error ← monitor(data)
8:     send_data:
9:         data_buf ← data
10:    start watchdog timer
11:    recv_cmds:
12:        (cmds, wd_triggered) ← cmds_buf
13:    cmds_error ← monitor(cmds)
14:    if data_error or wd_triggered or cmds_error then
15:        rcf:
16:            recovery_control(cmds)
17:    end if
18:    send_cmds:
19:        uart_cmds ← cmds
20: end while
```

In `recv_data()` the TIOP waits for data from the sensors to become available. It then processes this data with the SM in line 7. The next step is to transfer the data to the AP through the AXI FIFO in `send_data()`. The TIOP then begins a timer before waiting to receive actuator commands from the AP. If no commands are received in `recv_cmds()` before

the timer runs out, the flag `wd_triggered()` is set and the TIOP moves on. If commands are received, line 13 checks their validity. If the sensor data or actuator commands indicate the system is approaching an unsafe state, or the watchdog timer expired, line 14 causes the activation of the RCF. The RCF modifies the actuator commands to bring the system back to a safe state. The final event to take place in the TIOP is for the actuator commands to be sent out in `send_cmds()`. The system then returns to `recv_data()` and waits for more sensor data.

In a multi-TIOP architecture, the primary TIOP follows the same algorithm with a slight modification. The UARTs are replaced with the AXI-Stream interfaces discussed in Section 4.1. The secondary TIOP follows Algorithm 3. It starts by initializing the UART and AXI-Stream devices (lines 2 and 3). It then enters an infinite loop of receiving data, sending that data to the primary TIOP, receiving commands from the primary TIOP, and sending those commands out to the actuators in lines 6, 8, 10, and 12, respectively.

Algorithm 3 Secondary TIOP routine

```
1: init:
2:     Initialize UARTs uart_data, uart_cmds
3:     Initialize AXI-Stream data_stream, cmds_stream
4: while true do
5:     recv_data:
6:         data ← uart_data
7:     send_data:
8:         data_stream ← data
9:     recv_cmds:
10:        cmds ← cmds_stream
11:    send_cmds:
12:        uart_cmds ← cmds
13: end while
```

A guide to programming the MicroBlazes can be found in Section A.3.

4.3 Memory-mapped Peripherals

On the Zynq platform, all peripheral devices are mapped to a section of memory of the controller they are connected to. This means that writing to or reading from those addresses will result in a data transfer on the AXI bus to the corresponding peripheral. The following section outlines how this memory mapping is implemented for each architecture.

Figure 4.5 captures the memory addresses for the different peripherals connected to the AP in the TIOP-less architecture. The first column identifies the device that is connected to the

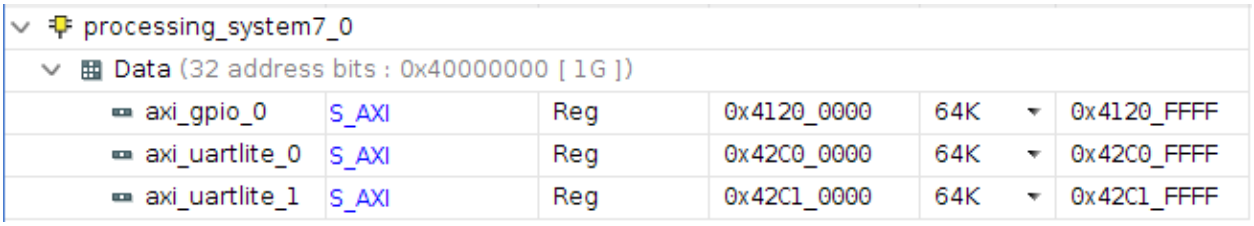

Figure 4.5: Address spaces for the TIOP-less architecture

AP. Column 2 lists the interface connection type, in this case, all interfaces are AXI slaves (**S_AXI**). A value of **Reg** in the third column indicates that the interface is connected to a register. The final three columns list the starting value, size, and ending value of the address space associated with each device. By reading from and writing to these addresses with device-specific offsets the AP (**processing_system7_0**) is able to control these devices. The AP runs a minimal version of Linux known as PetaLinux [66]. Because of this, access to these memory locations is not as simple as on a bare metal application. To control the GPIO pins the `mmap` command is used to permit access to the AXI interface [32]. AXI GPIO (**axi_gpio_0**) registers are then manipulated as described in [61] to drive pin values to a value of one or zero. Each function in Algorithm 1 drives a unique pin to a value of one, and all pins are reset to zero after the control loop finishes executing. The UART devices (**axi_uartlite_0** and **axi_uartlite_1**) are controlled using Linux drivers which use Linux file I/O and the `termios` library [58].

Figure 4.6 shows the connections between the AP, TIOP, and peripherals in the single TIOP architecture. The layout of the data values is the same as in the TIOP-less architecture, although some new values are present. **SLMB** in column two indicates that the interface is not an AXI slave, but rather a BRAM interface controller. Furthermore, **Mem** and **Mem0** indicate this interface is connected to some form or memory rather than a register. The FIFOs are shown as memory because this is how they are implemented in hardware. As discussed in Section 4.1, the TIOP (**microblaze_0**) is connected to an AXI timer (**axi_timer_0**)

processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
rx_data_buf	S_AXI	Mem0	0x4100_0000	64K	0x4100_FFFF
tx_cmds_buf	S_AXI	Mem0	0x4000_0000	64K	0x4000_FFFF
microblaze_0					
Data (32 address bits : 4G)					
axi_gpio_1	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_gpio_2	S_AXI	Reg	0x4001_0000	64K	0x4001_FFFF
axi_timer_0	S_AXI	Reg	0x41C0_0000	64K	0x41C0_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
axi_uartlite_1	S_AXI	Reg	0x4061_0000	64K	0x4061_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	16K	0x0000_3FFF
rx_cmds_buf	S_AXI	Mem0	0x44A0_0000	64K	0x44A0_FFFF
tx_data_buf	S_AXI	Mem0	0x44A1_0000	64K	0x44A1_FFFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	16K	0x0000_3FFF

Figure 4.6: Address spaces for the single TIOP architecture

which is used as the watchdog timer outlined in Algorithm 2. It is also connected to the two UART devices and two GPIO devices (**axi_gpio_1** and **axi_gpio_2**) which are used for debugging. The final two connections are to the FIFOs (**rx_cmds_buf** and **tx_data_buf**) used for communication with the AP. All of these devices are controlled through the use of Xilinx provided APIs [47]. The AP interacts with the GPIO in the same way as in the reference architecture but is now connected to the AXI-Stream FIFOs (**tx_cmds_buf** and **rx_data_buf**) instead of the UARTs. The FIFOs are accessed with the `mmap` command and are controlled by reading and writing 32-bit words of data along with several control registers [60].

Figure 4.7 adds the addresses for the secondary TIOP, **microblaze_1**. The layout is the same as described for the single TIOP architecture. In this architecture, the UART communication moves to the secondary TIOP but not much else changes from the single TIOP architecture. Because the primary and secondary TIOPs have a direct register to register communication, no memory addresses are needed to connect these two devices, unlike the FIFOs that connect the primary TIOP and the AP.

processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
rx_data_buf	S_AXI	Mem0	0x4100_0000	64K	0x4100_FFFF
tx_cmds_buf	S_AXI	Mem0	0x4000_0000	64K	0x4000_FFFF
microblaze_0					
Data (32 address bits : 4G)					
axi_gpio_1	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_gpio_2	S_AXI	Reg	0x4001_0000	64K	0x4001_FFFF
axi_timer_0	S_AXI	Reg	0x41C0_0000	64K	0x41C0_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	16K	0x0000_3FFF
rx_cmds_buf	S_AXI	Mem0	0x44A0_0000	64K	0x44A0_FFFF
tx_data_buf	S_AXI	Mem0	0x44A1_0000	64K	0x44A1_FFFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	16K	0x0000_3FFF
microblaze_1					
Data (32 address bits : 4G)					
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
axi_uartlite_1	S_AXI	Reg	0x4061_0000	64K	0x4061_FFFF
microblaze_1_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	16K	0x0000_3FFF
Instruction (32 address bits : 4G)					
microblaze_1_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	16K	0x0000_3FFF

Figure 4.7: Address spaces for the multi-TIOP architecture

4.4 Revisions to the Design

In the first stage of design and testing, data was transferred as null-terminated strings. This added significant overheads for transforming the floating-point values into strings. In addition, the SM and RCF were not initially included in the TIOP. These were added while still using the string data format, so monitoring required a conversion back to floating-points which added more latency. After converting to 16-bit fixed-point values, these latencies were drastically reduced. This formatting change almost halved latencies in the TIOP-less architecture, and latencies were reduced by almost a factor of ten in the architectures utilizing the TIOPs.

An important change was to use full optimization on the code running on the MicroBlazes. Initially debug mode was used, which does not take advantage of hardware registers, and stores all variables in memory. Once "release" mode was used with the `-O3` (highest opti-

mization) flags, the latencies related to overheads of adding a secondary TIOP were reduced from around $2\mu s$ to around $70ns$.

Another major change was modifying the speed of the MicroBlazes and FPGA fabric. To begin with, these were set to 50MHz, the default value. Once these values were changed to 100MHz, the latencies introduced by a secondary TIOP increased from $70ns$ to the $290ns$ identified in Section 5.2; however, the overall performance of the TIOP architectures improved. Initially, adding a single TIOP added a transmission latency of around $5\mu s$, but after applying the optimizations and increasing the clock frequency the TIOP actually reduced the transmission latency by around $3\mu s$. This can be attributed to the fact that the TIOPs are now running at almost double the original speed, so their timing data was almost halved in all areas. Because the ARM processor runs at 667MHz, it seemed appropriate to raise the speed of the TIOPs to be competitive and the timing data supports this decision.

Chapter 5

Implementation Analysis

5.1 Formal Verification

t The Spin and Modex tools were selected for this thesis due to a number of resources and examples available. Spin and Modex are also free and open-source, which encourages review by others to ensure the tools are implemented correctly. seL4 uses another model checker known as FDR4 which has similar functionality to Spin, but was not really investigated until later in the design process [10]. Both the Modex and Promela code used in this thesis can be found on Gitlab[1].

The control algorithms that run on the APs are often too complex to model check due to context switching, interrupts, and the sheer number of states. However, model checking can be used to verify that the communication algorithms executing on the TIOPs will always satisfy certain properties. The Promela language and associated tools Spin [37] and Modex [54] are used to perform the model checking. To do this, Algorithm 2 is modeled as the

[1] https://gitlab.com/zybo-secure/spin-thesis

finite state machine (FSM) shown in Figure 5.1. As in Algorithm 2, the machine enters the `recv_data` state after initialization, then moves to the monitor state. The main difference between this FSM and the algorithm is that there is only one monitor state. The transition out of this state is determined by the previous state (`prev`) and whether an error has been encountered (`ERR`). This error value is a logical-or between the monitoring of data and commands, and it is reset upon returning to `recv_data`.

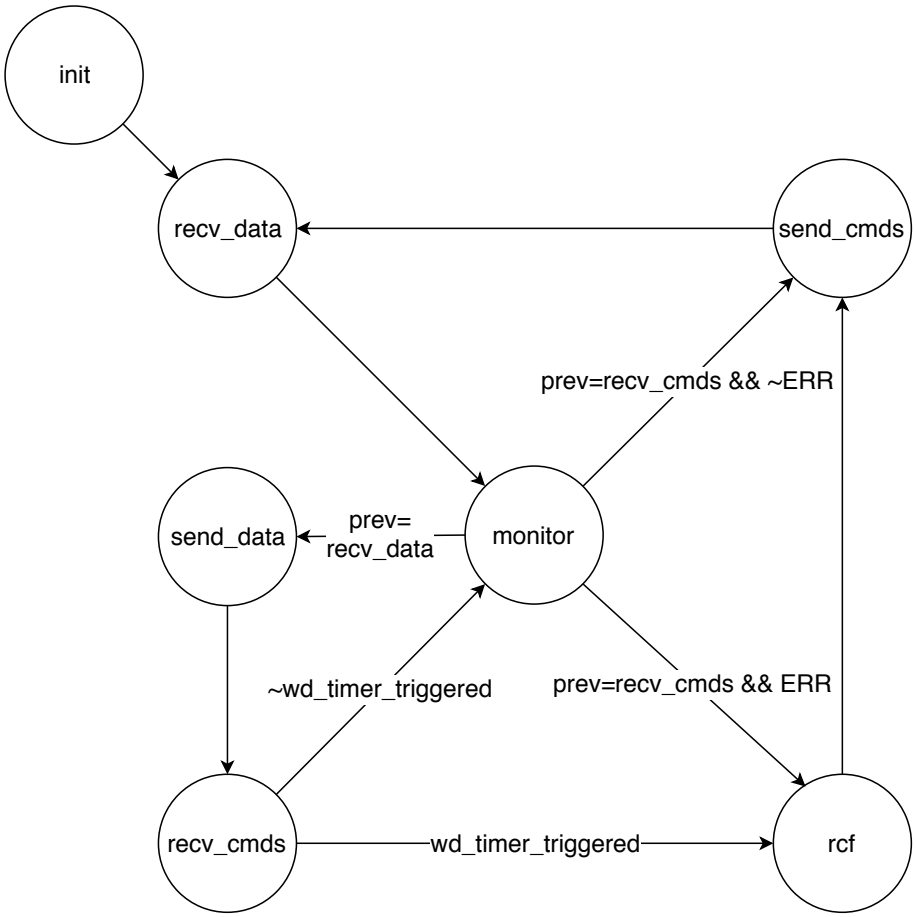

Figure 5.1: State machine model of a TIOP process

The state machine described is then encoded using Promela syntax. After translation, several LTL formulas are developed to describe the expected behavior of the algorithms in all executions. LTL is a formal language for expressing properties over time [53] and in-

corporates several operators such as `Always ([])`, `Eventually (<>)`, and `Implies (->)`. LTL formulas can also use the logical operators `and (&&)`, `or (||)`, and `not (!)`. These formulas capture aspects of system *correctness* and *liveness*. While correctness ensures the system does not violate properties, liveness ensures the system always makes progress [51] [23]. Listing C.1 is the Promela code used to model the system. Listing 5.1 is an excerpt of this code that has been slightly modified for readability and shows the Promela LTL statements that capture correct behavior. Formulas 1 to 3 ensure the safety of the system when one of the three error conditions arise. Formulas 4 and 5 also ensure safety by asserting that the state machine never enters an invalid state, nor is the RCF activated if there is no error. Finally, formula 6 ensures liveness by stating that whenever sensor data becomes available, actuator commands will eventually be sent. The Spin tool converts the Promela code into an executable program that analyzes all possible event and state sequences. A counter-example is generated if a property is false.

Listing 5.1: Basic LTL formulas

```
1  [](( !data_safe) -> <>(rcf_activated))
2  [](( !ap_running) -> <>(rcf_activated))
3  [](( !cmds_safe) -> <>(rcf_activated))
4  [](( data_safe && ap_running && cmds_safe) -> [](!rcf_activated))
5  [](!error_state)
6  [](( data_received) -> <>(commands_sent))
```

Once all LTL formulas are validated for the model, the next step is to ensure the C implementation of the model is correct. Modex translates C code into a Promela model by converting as much of the codes as possible into Promela syntax. Any portions of code that it cannot be converted are embedded within the model utilizing Spin's built-in `c_code` and

`c_expr` functions [38]. `c_code` is used to allow the code to modify parts of the system that cannot be modeled with Promela. `c_expr` determines the value of expressions related to the same parts of the system that cannot be modeled. One requirement of `c_expr` is that the expression cannot have any side effects and must only evaluate a statement without modifying the system. To accommodate these requirements, the original C code running on the TIOP needed to be tweaked slightly. Several checks on the return value of a function were done when assigning that value, like those listed as invalid in Listing 5.2. This had to be changed to an assignment operation followed by a evaluation, which can be handled by separate `c_code()` and `c_expr()` functions as shown in Listing 5.2. One limitation of the Modex tool is that all code must be ANSI-C compliant or the compilation will fail. Additionally, all variables in functions appear to be treated as static variables. If variables are not set after initialization in a function call, they retain the same value as at the end of the previous call to the function. This led to a state explosion in initial testing as some counters were not reset between calls.

Listing 5.2: Valid and invalid C code for Promela translation

```
/* Invalid */
if ((retval = foo())) {
  bar(retval);
}

/* Valid */
retval = foo(); /* c_code() */
if (retval) { /* c_expr() */
  bar(retval);
}
```

Part of the translation requires a user to define which functions are under test. These definitions are done in a `.prx` file, like the one shown in Listing C.2. This file is used to select which files to convert and defines any user types, as well as how to translate more complex C code into Promela-like statements. To simplify the analysis, data-type conversion when receiving or sending information via the FIFOs and UARTs is ignored and instead modeled as a common data-type message passing. This conversion, while easy in C, adds unnecessary complexity to the Promela model, and is verified through conventional testing. This substitution is another example of an assumption related to the formal verification process.

5.2 Timing

To collect accurate timing data on the system transitions as well as data transmission, a logic analyzer [57] is connected to several I/O pins located on Jumper B of the Zybo-Z7-20. These pins are driven to a value of 1 at different points of the Algorithm 1 execution flow. Figure 5.2 shows the timing results of both the reference implementation and the implementation utilizing a TIOP. In addition, to test the impact of incorporating secondary TIOPs, a simplified version of the architecture discussed in Figure 3.5 is implemented. The timing results for this architecture are also found in Figure 5.2. This system only implements a single secondary TIOP. `recv_data()` represents the time between the AP waking up and receiving all data from sensors. The time taken to complete the PID algorithm is identified by `pid()`. `send_cmds()` indicates the delay between the PID finishing and the AP communicating all commands. The time from the AP waking up to the first byte of data being transmitted via the UART is represented by `TX`. This is detected by connecting to an extra pin on Jumper

A, the connector containing the actuator UART. The sensor UART is connected to Jumper C. Pictures of this setup are found in Chapter B. In this implementation the MicroBlazes (TIOPs) have a clock rate of 100MHz, the ARM (AP) has a 667MHz clock frequency, and the control loop period is $50ms$. This period was determined by matching the computational period of the MATLAB script used to simulate the motor.

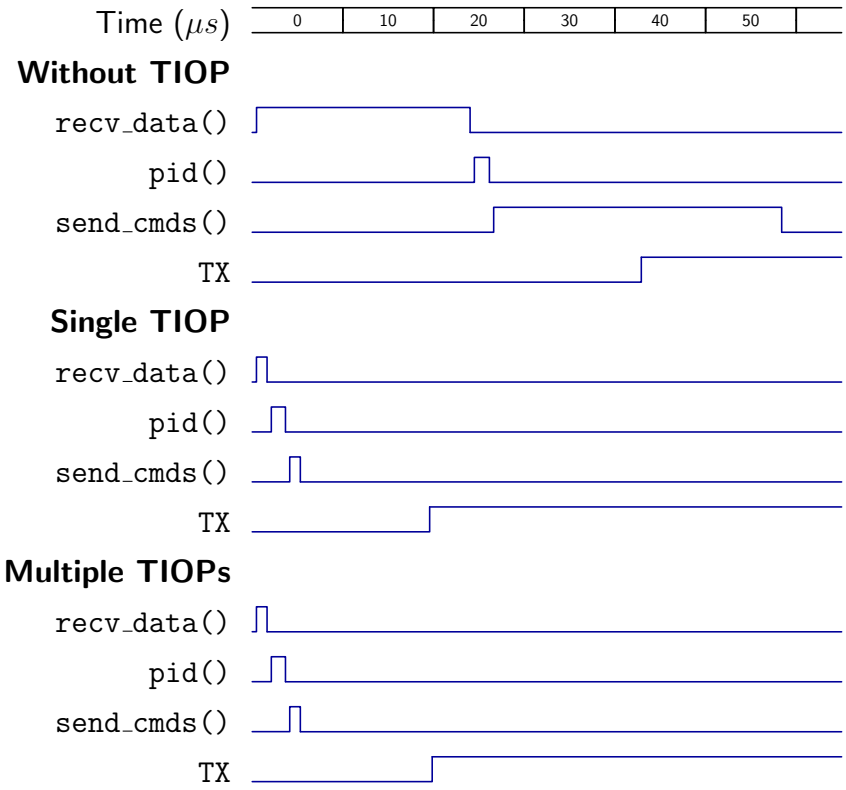

Figure 5.2: Timing diagram comparing architectures

Figures 5.3 to 5.5 are UML diagrams showing the sequence of events that takes place in each architecture from the start of the control loop until data is transmitted [41]. As shown in Figure 5.3, the AP in the reference implementation must wait until the `wait_period()` function completes before data can be read from the UART devices. Meanwhile, in both architectures containing TIOPs (Figures 5.4 and 5.5) this data can be read, validated, and written to a buffer by the TIOPs prior to the AP awaking. In addition, the overheads related to memory mapped (`mmap`) reads and writes of the FIFOs are significantly less than

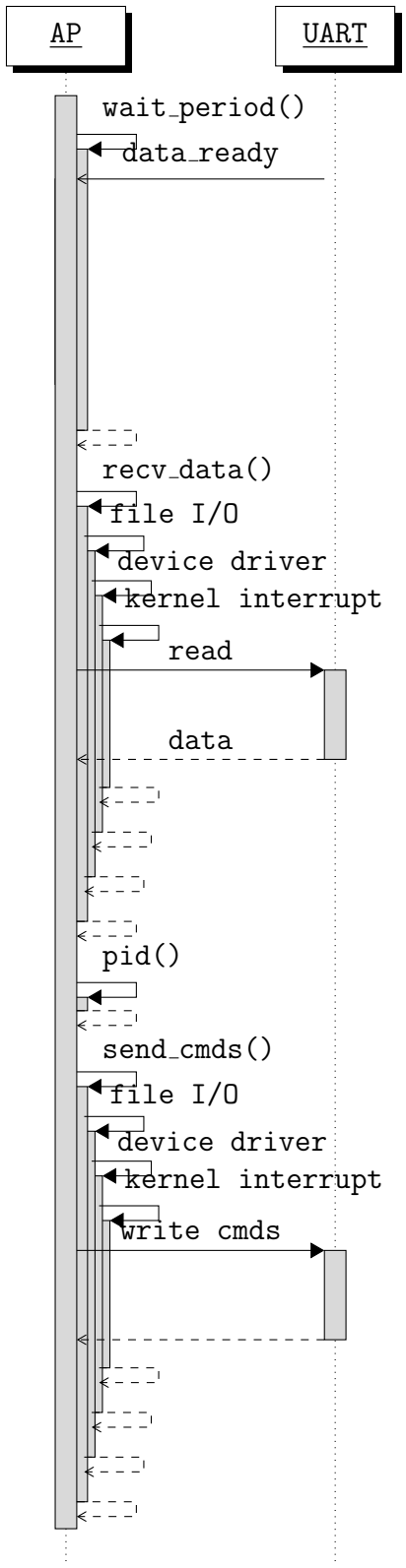

Figure 5.3: UML for reference architecture

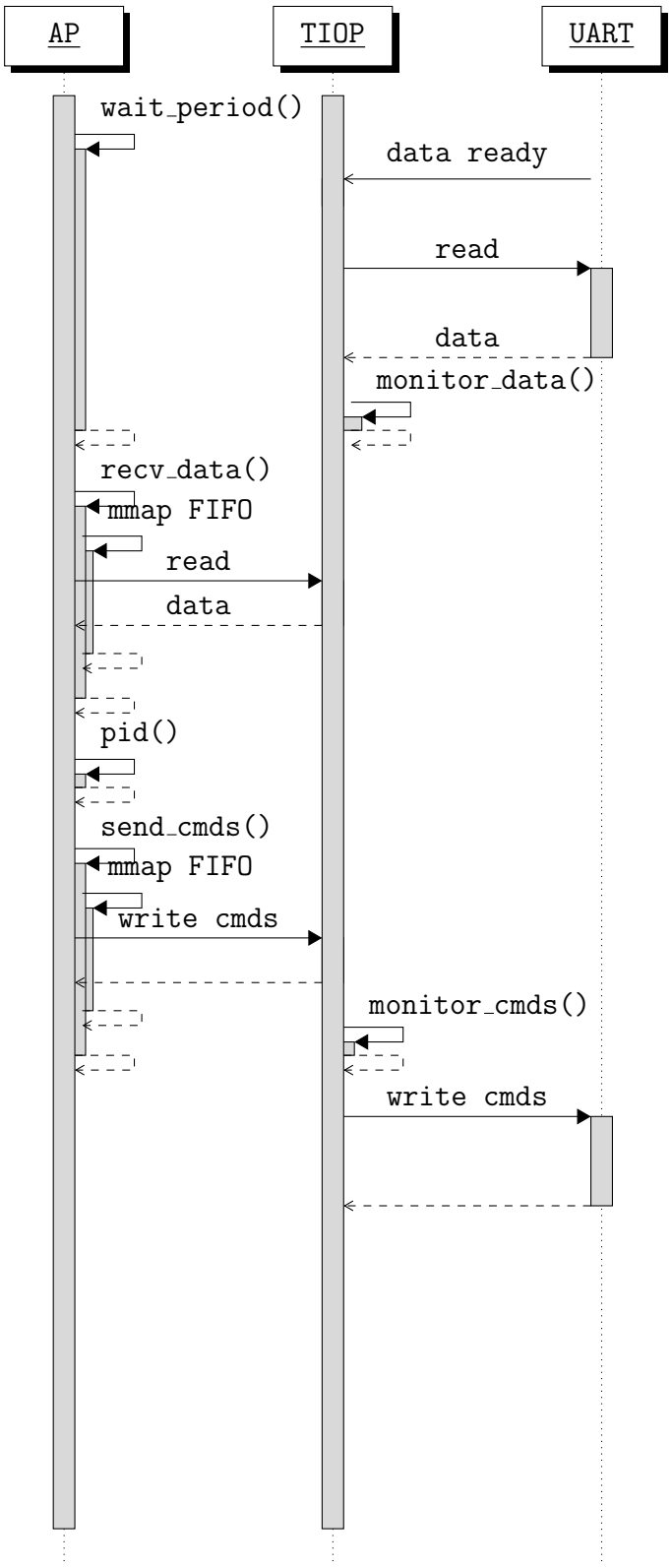

Figure 5.4: UML for single TIOP architecture

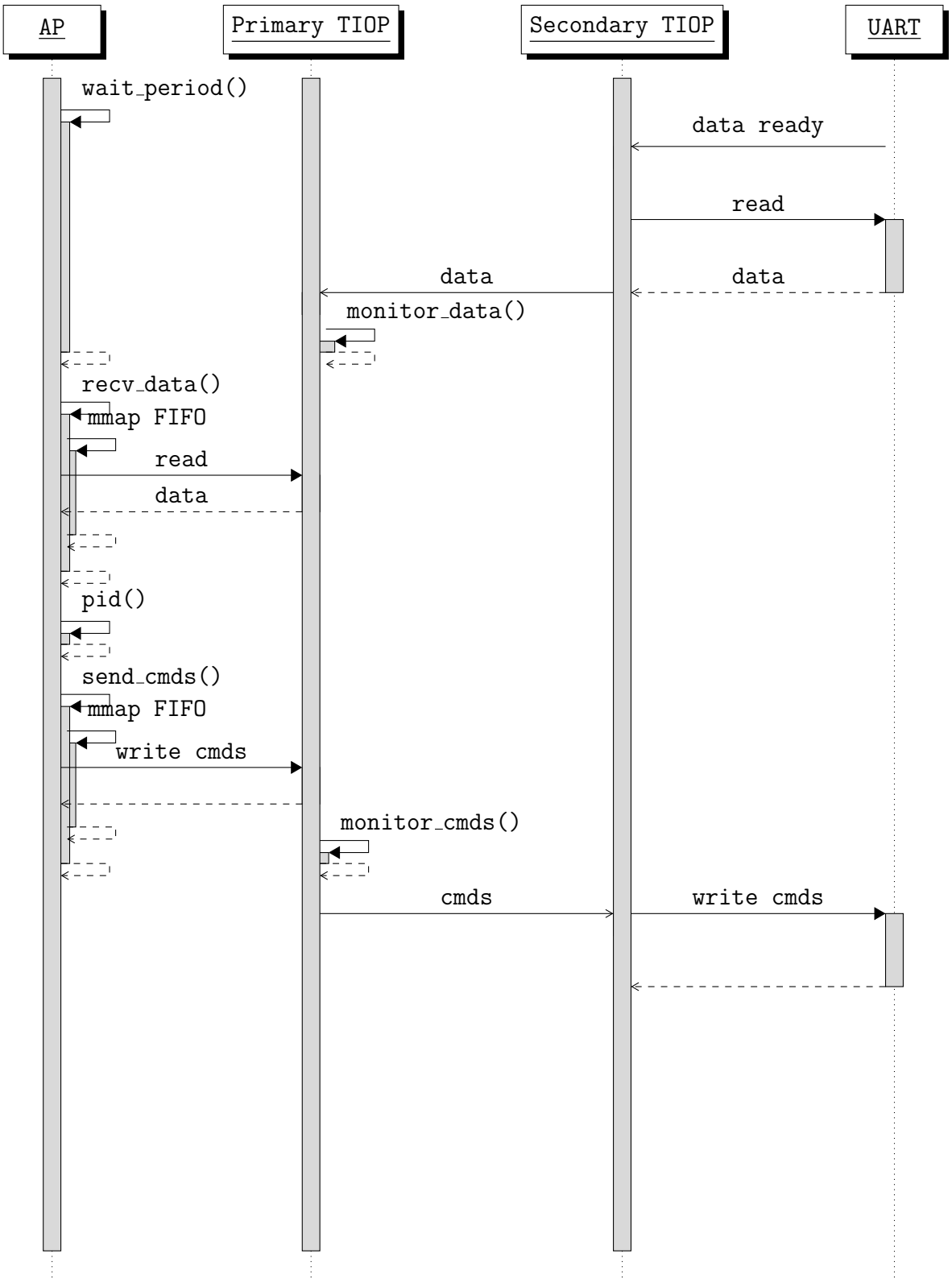

Figure 5.5: UML for mutli-TIOP architecture

the overheads associated with direct UART interaction through the Linux file I/O method. These combine to greatly reduce the control loop latency for both receiving sensor data, as well as sending actuator commands.

In Table 5.1 the timing values correspond to lines 5 to 7 of Algorithm 1, respectively. The `TX` value is defined as the time from the start of the `send_cmds()` function to first byte being transmitted via the UART. The total value is the total time from the end of line 4 of Algorithm 1 to either the `TX` event or the end of the `send_cmds()` function, depending on which comes last. These results were analyzed from a collection of 500 samples.

Table 5.1: Timing data (in μsec) for a TIOP-less architecture

Operation	**Min**	**Max**	**Mean**	**StdDev**
`recv_data()`	22.72	38.22	23.55	0.831
`pid()`	1.49	2.54	1.66	0.08
`send_cmds()`	30.78	43.79	31.76	0.70
`TX`	7.33	26.57	16.28	5.02
Total	56.78	85.95	58.39	1.53

As shown by Table 5.2, data transfer has a lower latency when a TIOP is used because the sensor data can be copied to a buffer while the AP is waking. Introducing a TIOP also reduces transmit latency by replacing complex driver code with fast sequential memory writes. As shown in the two tables, the overheads of the UART driver code is about $3.5\mu s$ greater than the overheads of transmitting the data through the TIOP, which also checks if the commands are safe prior to sending them. This combination of reductions greatly reduces overall latency. The maximum total latency for the single TIOP architecture is almost half of the minimum total latency of the TIOP-less latency. The mean latency for the single TIOP architecture is almost a third of the TIOP-less architecture.

As described in Section 4.1, communication between the primary and secondary TIOPs uses a high-speed AXI stream interface [59]. The timing data retrieved from this architecture can

Table 5.2: Timing data (in μsec) for single a TIOP architecture

Operation	Min	Max	Mean	StdDev
`recv_data()`	1.14	1.20	1.17	0.01
`pid()`	1.46	1.89	1.56	0.05
`send_cmds()`	1.16	1.20	1.17	0.01
`TX`	4.78	25.72	15.43	5.17
Total	9.09	29.97	19.57	5.17

be found in Table 5.3, and the added overheads are shown in Table 5.4. Adding secondary TIOPs has no effect on the reception of sensor data, as it is still buffered before the AP has finished its wait cycle. The introduction of a secondary TIOP adds around $290ns$ (29 clock cycles) to the transmission of actuator commands. These overheads are negligible when considering the added functionality of multiple secondary TIOPs which can concurrently interface with twice as many I/O.

Table 5.3: Timing data (in μsec) for a multi-TIOP architecture

Operation	Min	Max	Mean	StdDev
`recv_data()`	1.15	1.33	1.17	0.01
`pid()`	1.45	1.92	1.56	0.06
`send_cmds()`	1.15	1.34	1.17	0.01
`TX`	5.16	26.11	15.72	5.24
Total	9.30	30.25	19.85	5.24

Table 5.4: Overheads (in μsec) added by a secondary TIOP

Operation	Min	Max	Mean
`recv_data()`	0.01	0.13	0.00
`pid()`	-0.01	0.04	0.00
`send_cmds()`	-0.01	0.14	0.00
`TX`	0.38	0.28	0.29
Total	0.21	0.28	0.28

In all three architectures the latency for `TX` has a high standard deviation, and it is relatively

similar. This likely arises from the actual hardware that is transmitting the data and is related to latencies internal to that hardware.

5.3 Resource Utilization

Table 5.5: Resource utilization for a single TIOP architecture

Resource	Utilization	Available	% Utilization
LUT	3965	53200	7.45
LUTRAM	333	17400	1.91
FF	4002	106400	3.76
BRAM	8	140	5.71

Table 5.6: Resource utilization for a multi-TIOP architecture

Resource	Utilization	Available	% Utilization
LUT	5293	53200	9.95
LUTRAM	455	17400	2.61
FF	5116	106400	4.81
BRAM	12	140	8.57

MicroBlaze soft processors have variants optimized for speed or area, and the high performance version is used in our experiments. Speed-optimization would likely improve timing slightly but uses almost ten times as many FPGA resources, impacting scalability. This increased resource utilization results from a restructured architecture that attempts to take advantage of high levels of parallelization through additional instruction pipelining. An area-optimized MicroBlaze reduces resource usage but impacts performance and would affect timing. The performance-optimized MicroBlaze strikes a balance of resource usage and performance.

As shown by Tables 5.5 and 5.6, the addition of a secondary TIOP only increases LUT usage by 2.2%, LUTRAM usage by 0.5%, FF usage by 0.9% and BRAM usage by 2.8%.

These overheads would limit a system on this device to approximately 32 secondary TIOPs. Secondary TIOPs could be all connected to the primary TIOP which can have up to 32 AXI-Stream interfaces [59]. They could also be implemented in a hierarchical manner, with several secondary TIOPs controlling and monitoring their own subset of ternary TIOPs. Figure 5.6 illustrates one such hierarchy.

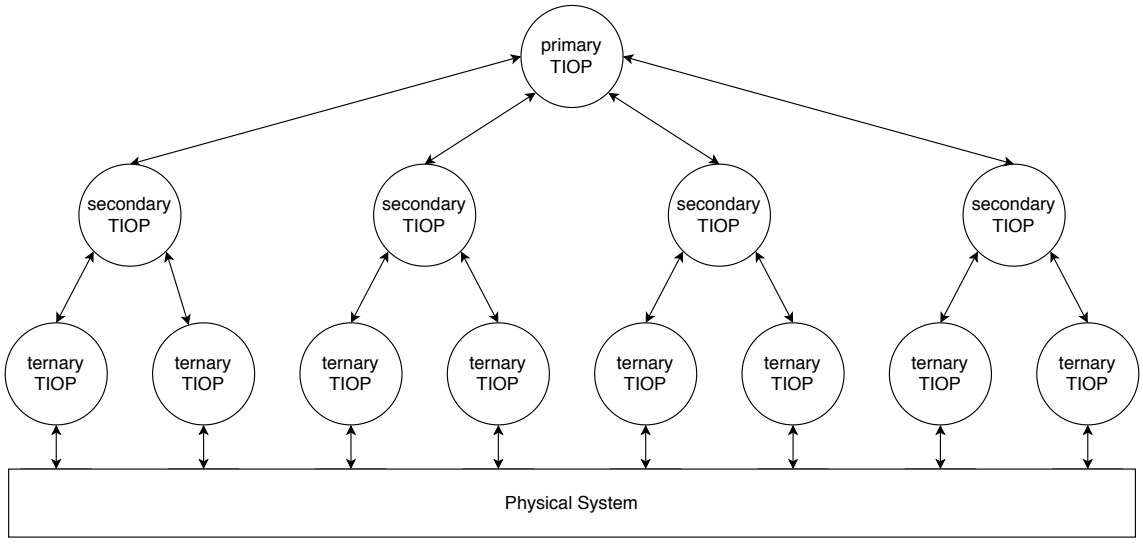

Figure 5.6: A hierarchical DCS containing several TIOPs

Each MicroBlaze has 16KB of memory available, and Table 5.7 outlines the memory usage of the different architectures. The *text* segment of memory contains read-only memory including any constant values and the executable instructions. The *data* section includes any variables that are initialized at the beginning of the program's execution. Finally, the *bss* segment contains data that is uninitialized or initialized to zero at the start of the program. The total column is the total size of the executable which is just the sum of the three previous segments.

Table 5.7: Memory usage of TIOP executables (bytes)

TIOP	**text**	**data**	**bss**	**Total**
Single	10520	350	2098	12968
Primary	7632	318	3122	11072
Secondary	5016	304	3122	8432

Chapter 6

Conclusions

This thesis designs, implements, and evaluates an architecture which uses dedicated and isolated processors for I/O processing. By isolating this interaction, formal methods can be applied to code critical to proper operation of a DCS. The architecture promotes the insertion of SMs and RCFs, further safeguarding the system from critical failures. Furthermore, no major latency is introduced to the system, and scalability to more complex systems is supported.

Frequently designers have to trade off the objectives of malware resilience, correctness, low latency, and scalability. This is because security often adds overheads, which increase latency or complexity, which makes verifying correctness more difficult. Sometimes a proposed solution is only practical for small systems. The architecture described here simultaneously addresses all four of these objectives. Dedicated I/O processors similar to those used in this architecture are already used in various commercial systems, from aircraft avionics [14] to the IBM Blue Gene supercomputer [13].

6.1 Future Work

The implementation of the proposed architecture is evaluated with a simple simulation of a motor controller. An architecture instantiating a single TIOP was tested. This was followed by the implementation and testing of a tiered TIOP system with a primary and secondary TIOP to evaluate the impact of secondary TIOPs. Future systems could include additional secondary TIOPs to accommodate sensor redundancy and cross-checking. Lack of sensor redundancy was a potential factor in the Boeing 737 MAX crashes in 2018 and 2019 [31]. While model checking is applied to the single TIOP architecture's software model and implementation, no formal methods are applied to the multi-TIOP architecture. Modeling and verifying distributed software processes is well supported by Promela, and is underway.

Stamenkovich et al. have implemented monitors like those found in the TIOP as hardware in the FPGA [55]. Merging these two works by migrating the SM and RCF out of the TIOP and into hardware will be a desirable step in moving forward.

Bibliography

[1] FieldComm Group, Apr 1970. URL https://fieldcommgroup.org/.

[2] Trusted Platform Module. *Trusted Computing Group*, Apr 2008. URL https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary.

[3] PetaLinux Tools Documentation Reference Guide. Technical Report UG1144 (v2017.4), Xilinx, Inc., Dec 2017. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1144-petalinux-tools-reference-guide.pdf.

[4] PROFIBUS, 2017. URL https://www.profibus.com/.

[5] Bing launches new intelligent search features, powered by AI, 2018. URL https://blogs.bing.com/search/2017-12/search-2017-12-December-AI-Update. [Accessed: 12-June-2018].

[6] MT3620, 2018. URL https://www.mediatek.com/products/azureSphere/mt3620. [Accessed: 23-May-2018].

[7] Microsoft Azure Sphere - a solution for creating highly-secured, connected MCU powered devices, 2018. URL https://www.microsoft.com/en-us/azure-sphere. [Accessed: 01-June-2018].

[8] Google Nest, 2018. URL https://nest.com. [Accessed: 25-May-2018].

[9] AMBA AXI4 Interface Protocol, 2019. URL https://www.xilinx.com/products/intellectual-property/axi.html.

[10] FDR4 - The CSP Refinement Checker, Feb 2019. URL https://www.cs.ox.ac.uk/projects/fdr/.

[11] UPS Partners With Matternet To Transport Medical Samples Via Drone Across Hospital System In Raleigh, N.C., Mar 2019. URL https://pressroom.ups.com/pressroom/ContentDetailsViewer.page?ConceptType=PressReleases&id=1553546776652-986.

[12] Vivado Design Suite, 2019. URL https://www.xilinx.com/products/design-tools/vivado.html.

[13] George Almási, Ralph Bellofatto, José Brunheroto, Călin Caşcaval, José G. Castaños, Luis Ceze, Paul Crumley, Christopher C. Erway, Joseph Gagliano, Derek Lieber, Xavier Martorell, José E. Moriera, Alda Sanomiya, and Karin Strauss. An Overview of the Blue Gene/L System Software Organization. *Euro-Par 2003 Parallel Processing Lecture Notes in Computer Science*, page 543–555, 2003. doi: 10.1007/978-3-540-45209-6_79.

[14] Aspen Avionics, Inc. EFD1000 PFD Pilot's Guide. EFD1000_PFD_Pilots_Guide_V1.1.pdf, June 2009. URL https://aspenavionics.com/pdfs/EFD1000_PFD_Pilots_Guide_V1.1.pdf.

[15] ASTM F3269 - 17. Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions. *ASTM International*, May 2018. doi: 10.1520/f3269-17. URL www.astm.org.

[16] Daniele Bagni and Duncan Mackay. Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP. Technical Report XAPP1163 (v1.0), Xilinx, Inc.,

Jan 2013. URL https://www.xilinx.com/support/documentation/application_notes/xapp1163.pdf.

[17] Tom Ball, Ed Targett, Conor Reynolds, and James Nunns. Top 5 critical infrastructure cyber attacks, Jan 2018. URL https://www.cbronline.com/cybersecurity/top-5-infrastructure-hacks/.

[18] Ezio Bartocci and Yliès Falcone. *Lectures on Runtime Verification: Introductory and Advanced Topics*. Springer International, 2018.

[19] Alyson Behr. More than an auto-pilot, AI charts its course in aviation, Dec 2018. URL https://arstechnica.com/information-technology/2018/12/unite-day1-1/.

[20] Mordechai (Moti) Ben-Ari. A primer on model checking. *ACM Inroads*, 1(1):40, Jan 2010. doi: 10.1145/1721933.1721950.

[21] N Teja Chiluvuri, Omkar A Harshe, Cameron D Patterson, and William T Baumann. Using heterogeneous computing to implement a trust isolated architecture for cyber-physical control systems. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, pages 25–35. ACM, 2015.

[22] Nayana Teja Chiluvuri. A Trusted Autonomic Architecture to Safeguard Cyber-Physical Control Leaf Nodes and Protect Process Integrity. Master's thesis, Virginia Polytechnic and State University, 2015.

[23] E. M. Clarke. My 27-year Quest to Overcome the State Explosion Problem. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 3–3, Aug 2009. doi: 10.1109/LICS.2009.42.

[24] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2008.

[25] Joe Delaere and Stefano Zammattio. Top 7 Reasons to Replace Your Microcontroller with a MAX®10 FPGA. *White Paper*, 2017.

[26] Z. Deng, C. Ma, and M. Zhu. A reconfigurable flight control system architecture for Small Unmanned Aerial Vehicles. In *2012 IEEE International Systems Conference SysCon 2012*, pages 1–4, March 2012. doi: 10.1109/SysCon.2012.6189451.

[27] Digilent, Inc. Zybo Z7: Zynq-7000 ARM/FPGA SoC Development Board, 2019. URL https://store.digilentinc.com/zybo-z7-zynq-7000-arm-fpga-soc-development-board/.

[28] Bruce Powell. Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley, 2003.

[29] Nate Eastland. Structure of an FPGA, Aug 2015. URL https://blog.digilentinc.com/structure-of-an-fpga/.

[30] Jim Finkle. Hackers halt plant operations in watershed cyber attack, Dec 2017. URL https://www.reuters.com/article/us-cyber-infrastructure-attack-idUSKBN1E8271.

[31] Todd C. Frankel. Sensor cited as potential factor in Boeing crashes draws scrutiny, Mar 2019. URL https://www.washingtonpost.com/business/economy/sensor-cited-as-potential-factor-in-boeing-crashes-draws-scrutiny/2019/03/17/5ecf0b0e-4682-11e9-aaf8-4512a6fe3439_story.html?noredirect=on&utm_term=.95b922343d32.

[32] B. O. Gallmeister. *mmap(2) - Linux Manual Page*. The Linux man-pages project, 5.00 edition, Feb 2019.

[33] Philippe Gayet, Renaud Barillere, et al. UNICOS a framework to build industry-like control systems, Principles and Methodology. *10th ICALEPCS*, 2005.

[34] Irving M. Gottlieb. *Electric motors & control techniques*. TAB Books, 2 edition, 1994.

[35] Daniel Harris. Reducing FPGA Power Consumption, Dec 2012. URL https://www.electronicdesign.com/fpgas/reducing-fpga-power-consumption.

[36] Nick Heath. Why the blue screen of death no longer plagues Windows users, Sep 2013. URL https://www.zdnet.com/article/why-the-blue-screen-of-death-no-longer-plagues-windows-users/.

[37] Gerard Holzmann. Verifying Multi-threaded Software with Spin, 2019. URL http://spinroot.com/spin/whatispin.html.

[38] Gerard J. Holzmann. *The Spin model checker: primer and reference manual*, chapter 17. Addison-Wesley, 2008.

[39] Ted Hudek. Analyzing a Driver Using Code Analysis and Verification Tools - Windows drivers, Jul 2008. URL https://docs.microsoft.com/en-us/windows-hardware/drivers/develop/analyzing-driver-quality-by-using-code-analysis-tools.

[40] Apple Inc. iOS Security. *White Paper*, Jan 2018. URL https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[41] Ankit Jain. Unified Modeling Language (UML) — Sequence Diagrams, Feb 2018. URL https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/.

[42] Taygun Kekec, Baris Can Ustundag, Mehmet Ali Guney, Alper Yildirim, and Mustafa Unel. A modular software architecture for UAVs. In *Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE*, pages 4037–4042. IEEE, 2013.

[43] Akshatha Jagannath Kini. Implementation of a Trusted I/O Processor on a Nascent SoC-FPGA Based Flight Controller for Unmanned Aerial Systems. Master's thesis, Virginia Polytechnic and State University, 2017.

[44] David Kushner. The Real Story of Stuxnet, Feb 2013. URL https://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet/.

[45] Edward A. Lee and Sanjit A. Seshia. *Introduction to embedded systems: a cyber-physical systems approach*. MIT Press, 2017.

[46] Tarjei Mandt, Mathew Solnik, and David Wang. Demystifying the Secure Enclave Processor, 2016. URL https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf.

[47] Terry O'Neal. Baremetal Drivers and Libraries, Feb 2019. URL https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841745/Baremetal+Drivers+and+Libraries.

[48] Daniel Potts, Rene Bourquin, Leslie Andresen, June Andronick, Gerwin Klein, and Gernot Heiser. Mathematically Verified Software Kernels: Raising the Bar for High Assurance Implementations. Technical report, NICTA, Sydney, Australia, July 2014.

[49] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL http://ptolemy.org/books/Systems.

[50] Renee S. Intel ®Aero Platform for UAVs, 2018. URL https://software.intel.com/en-us/aero/drone-kit. [Accessed: 01-June-2018].

[51] Julien Schmaltz. *2IX20 Software Specification*, chapter 9. Eindhoven University of Technology, 2017.

[52] Srikanth Seely, Joel Erusalagandi and Jayson Bethurem. The MicroBlaze Soft Processor: Flexibility and Performance for Cost-Sensitive Embedded Designs. Technical report, Xilinx, Apr 2017. URL https://www.xilinx.com/support/documentation/white_papers/wp501-microblaze.pdf.

[53] Sanjit A. Seshia. *Introduction to Temporal Logic*. UC Berkeley Department of Electrical Engineering and Computer Sciences, 2014. URL https://people.eecs.berkeley.edu/~sseshia/fmee/lectures/TemporalLogicIntro.pdf.

[54] spinroot. Modex 2.8 User Guide, March 2015. URL http://spinroot.com/modex/MANUAL.html.

[55] Joseph Stamenkovich, Lakshman Maalolan, and Cameron Patterson. Formal Assurances for Autonomous Systems Without Verifying Application Software. Technical report, Virginia Tech Bradley Department of Electrical and Computer Engineering, 2019.

[56] Andrew S. Tanenbaum. *Modern operating systems*. Pearson Prentice Hall, 2009.

[57] TechTools. DigiView DV3100 Logic Analyzer, Sep 2018. URL http://www.tech-tools.com/DV3100-logic-analyzer.htm.

[58] *termios(3) - Linux Manual Page*. The Linux man-pages project, 5.00 edition, March 2019.

[59] Xilinx, Inc. MicroBlaze Processor Reference Guide. Technical report, april 2014. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug984-vivado-microblaze-ref.pdf.

[60] Xilinx, Inc. AXI4-Stream FIFO v4.1 LogiCORE IP Product Guide. Technical report, April 2016. URL https://www.xilinx.com/support/documentation/ip_documentation/axi_fifo_mm_s/v4_1/pg080-axi-fifo-mm-s.pdf.

[61] Xilinx, Inc. AXI GPIO v2.0 LogiCORE IP Product Guide. Technical report, October 2016. URL https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf.

[62] Xilinx, Inc. AXI Interconnect v2.1 LogiCORE IP Product Guide. Technical report, December 2017. URL https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf.

[63] Xilinx, Inc. Mailbox v2.1 LogiCORE IP Product Guide. Technical report, April 2018. URL https://www.xilinx.com/support/documentation/ip_documentation/mailbox/v2_1/pg114-mailbox.pdf.

[64] Xilinx, Inc. Zynq-7000 SoC Data Sheet: Overview. Technical report, July 2018. URL https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[65] Xilinx, Inc. Zynq UltraScale+ MPSoC Data Sheet: Overview. Technical report, November 2018. URL https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[66] Xilinx, Inc. PetaLinux Tools, 2019. URL https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html.

[67] Peng Zhang. Chapter 1 - industrial control systems. In Peng Zhang, editor, *Advanced Industrial Control Technology*, pages 3 – 40. William Andrew Publishing, Oxford, 2010. ISBN 978-1-4377-7807-6. doi: https://doi.org/10.1016/B978-1-4377-7807-6.10001-4. URL http://www.sciencedirect.com/science/article/pii/B9781437778076100014.

Appendices

Appendix A

Using Vivado and Petalinux

A.1 Building the FPGA Hardware

Follow the steps found on Digilent's website[1] to install Vivado and the Digilent board files.

If building a TIOP-less architecture, skip steps 3, 4, 7 to 9, and 11, if not, skip step 6. If building a single TIOP architecture, skip step 4. If debugging GPIO for the primary TIOP is not desired, skip step 11. If timing data is not needed, skip step 10.

1. Launch Vivado and create a new project
   (a) Enter the desired name and project location
   (b) Select `RTL project`
   (c) Skip adding source files
   (d) Add the `Zybo-Z7-Master.xdc` file found on Digilent's Github[2]

[1] https://reference.digilentinc.com/vivado/installing-vivado/start
[2] https://github.com/Digilent/digilent-xdc/blob/master/Zybo-Z7-Master.xdc

(e) Make sure `Copy constraints files into project` is checked

(f) Select `Zybo-Z7-20` from the boards tab

(g) Select `Finish` to create the project

(h) Select `Create Block Design` from the `Flow Navigator`, pick a design name and directory

2. Add AP

(a) Add a `ZYNQ7 Processing System` IP

(b) Run `Block Automation`

- `Apply Board Preset`: Checked
- `Cross Trigger In`: Disable
- `Cross Trigger Out`: Disable

(c) Set the `PL system clock` to 100MHz

- Double click on the Zynq IP block to open `Re-customize IP` menu
- Navigate to `Clock Configuration`
- Open `PL Fabric Clocks`
- Change `FCLK_CLK0` from 50 to 100

3. Add Primary TIOP

(a) Add `MicroBlaze` IP

(b) Run `Block Automation`

- `Preset`: None
- `Local Memory`: 16KB
- `Local Memory ECC`: None

- `Cache Configuration`: None
- `Debug Module`: Debug Only
- `Peripheral AXI Port`: Enabled
- `Interrupt Controller`: Unchecked
- `Clock Connection`: `processing_system7_0/FCLK_CLK0`

(c) `Regenerate Layout`

(d) Block diagram should resemble Figure A.1

4. Add secondary TIOP(s) (optional)

(a) Double click primary TIOP to open `Re-customize IP Menu`

(b) Navigate to page 4

(c) Set `Number of Stream Interfaces` to number of secondary TIOPs, select `OK`

(d) Add `MicroBlaze` IP

(e) Run `Block Automation`

- `Preset`: None
- `Local Memory`: 16KB
- `Local Memory ECC`: None
- `Cache Configuration`: None
- `Debug Module`: Debug Only
- `Peripheral AXI Port`: Enabled
- `Interrupt Controller`: Unchecked
- `Clock Connection`: `processing_system7_0/FCLK_CLK0`

(f) Double click newly added TIOP to open `Re-customize IP Menu`

(g) Navigate to page 4

(h) Set `Number of Stream Interfaces` to 1, select `OK` (Figure A.6)

(i) Connect `S0_AXIS` on secondary TIOP to `M{N}_AXIS` on primary, where N is secondary TIOP number

(j) Connect `S{N}_AXIS` on primary TIOP to `M0_AXIS` on secondary, where N is secondary TIOP number

(k) Repeat step 4d to step 4j for all additional secondary TIOPs

(l) `Regenerate Layout`

(m) Block diagram for 1 secondary TIOP should resemble Figure A.7

(n) Optionally a `AXI Stream-Data FIFO` can be inserted between each pair of stream interfaces.

5. Add UART interfaces

(a) Add two `AXI Uartlite` IP, one for sensors, one for actuators

(b) Run `Connection Automation`

(c) Check all

(d) `S_AXI` settings:

i. `Master`: AP, or Primary/Secondary TIOP, depending on architecture

ii. `Interconnect IP`: New AXI Interconnect

iii. `Crossbar clock source of Interconnect IP`: Auto

iv. `Clock source for Master interface`: Auto

v. `Clock source for Slave interface`: Auto

(e) Rename `uart_rtl_0` to `uart_rtl_1` and `uart_rtl` to `uart_rtl_0` with the `External Interface Properties` menu

(f) Click on the plus icon next to UART on both AXI UART IP blocks

(g) Right click on `rx` and `tx` and select `make external` for both `AXI UART` IP blocks

6. Add UART interrupts (TIOP-less architecture only)

(a) Add `Concat` IP

(b) Double click on Zynq IP to open `Re-customize IP` menu

(c) Select `Interrupts` tab

(d) Check `Fabric Interrupts`, `PL-PS Interrupt Ports`, `IRQ_F2P[15:0]` (Figure A.2)

(e) Connect `Concat dout` port to `Zynq IRQ_F2P` port like shown in Figure A.2

(f) Connect `AXI Uartlite interrupt` ports to `Concat in` ports like shown in Figure A.2

7. Add AXI timer

(a) Add `AXI Timer` IP

(b) Run `Connection Automation`

(c) Check `axi_timer_0`

(d) `S_AXI` settings:

- `Master`: Primary TIOP
- `Interconnect IP`: If single TIOP architecture - primary TIOP AXI interconnect, if multi-TIOP architecture - `New AXI Interconnect`
- `Crossbar clock source of Interconnect IP`: Auto
- `Clock source for Master interface`: Auto
- `Clock source for Slave interface`: Auto

8. Add FIFOs

(a) Add `AXI-Stream FIFO` IP

(b) Double click new FIFO to open `Re-customize IP`

- Disable `Transmit Data`
- Disable `Transmit Control`
- Select `OK` to finish (Figure A.11)

(c) Rename FIFO with `Block Properties` tab to `rx_data_buf` (Figure A.4)

(d) Copy and paste `rx_data_buf`, renaming the new FIFO to `rx_cmds_buf`

(e) Add `AXI-Stream FIFO` IP

(f) Double click new FIFO to open `Re-customize IP`

- Disable `Receive Data`
- Disable `Transmit Control`
- Select `OK` to finish (Figure A.10)

(g) Rename FIFO with `Block Properties` tab to `tx_data_buf` (Figure A.4)

(h) Copy and paste `tx_data_buf`, renaming the new FIFO to `tx_cmds_buf`

(i) Connect `AXI_STR_TXD` of `tx_data_buf` to `AXI_STR_RXD` to `rx_data_buf`

(j) Connect `AXI_STR_TXD` of `tx_cmds_buf` to `AXI_STR_RXD` to `rx_cmds_buf`

(k) Buffers should be connected like in Figure A.5

9. Connect AP and primary TIOP

(a) Run `Connection Automation`

(b) Check `rx_data_buf` and `tx_cmds_buf`

(c) `S_AXI` settings:

- `Master`: `processing_system7_0` (AP)

- `Interconnect IP`: New AXI Interconnect
- `Crossbar clock source of Interconnect IP`: Auto
- `Clock source for Master interface`: Auto
- `Clock source for Slave interface`: Auto

(d) Select `OK`

(e) Run `Connection Automation`

(f) Check `rx_cmds_buf` and `tx_data_buf`

(g) `S_AXI` settings:

- `Master`: Primary TIOP
- `Interconnect IP`: Primary TIOP AXI interconnect
- `Crossbar clock source of Interconnect IP`: Auto
- `Clock source for Master interface`: Auto
- `Clock source for Slave interface`: Auto

(h) Select `OK`

(i) `Regenerate Layout`

10. Add timing GPIO (optional)

(a) Add `AXI GPIO` IP

(b) Run `Connection Automation`

(c) GPIO settings:

- `Select Board Part Interface`: `Custom`

(d) `S_AXI` settings:

- `Master`: `processing_system7_0` (AP)

- `Interconnect IP`: AP AXI Interconnect
- `Crossbar clock source of Interconnect IP`: Auto
- `Clock source for Master interface`: Auto
- `Clock source for Slave interface`: Auto

(e) Select OK

(f) Double click the GPIO IP to open `Re-customize IP`

(g) Navigate to the `IP Configuration Tab`

- `All inputs`: Unchecked
- `All outputs`: Checked
- `GPIO Width`: 8
- `Default Output Value`: 0x00000000
- `Enable Dual Channel`: Unchecked
- `Enable Interrupt`: Unchecked

(h) Select `OK`

(i) Click the plus next to `GPIO` on the `AXI GPIO IP`

(j) Rename port `gpio_rtl` to `GPIO_0` (Figure A.8)

(k) Right click on `gpio_io_o[7:0]`, select `make external`

(l) Rename the newly created port `gpio_jb`

(m) GPIO should now resemble Figure A.9

(n) Change external port `tx_0` to be two bits wide

i. Delete `tx_0`

ii. Right click anywhere on the diagram and select `create port`

- `Port Name`: `tx_0`

- `Direction`: Output
- `Type`: Other
- `Create vector`: Checked, from 1 to 0

iii. Select `OK` then connect the new port to the `tx` pin on `axi_uartlite_0`

11. Add debug GPIO (optional)
    (a) Add 2 AXI GPIO IP
    (b) Run `Connection Automation`
    (c) set `axi_gpio_1 GPIO` to `rgb_led`
    (d) set `axi_gpio_2 GPIO` to `leds_4bits`
    (e) `S_AXI` settings (both):
        - `Master`: Primary TIOP
        - `Interconnect IP`: Primary TIOP AXI Interconnect
        - `Crossbar clock source of Interconnect IP`: Auto
        - `Clock source for Master interface`: Auto
        - `Clock source for Slave interface`: Auto
    (f) Select OK
12. Assign addresses
    (a) Navigate to the `Address Editor` tab
    (b) Set the values for each `Data address` for each processor to match those in Section 4.3, whichever is appropriate
    (c) Return to the `Diagram` tab and select `Validate Design`
13. Modify constraints file to connect signals to ports

(a) Navigate to the sources tab of Hierarchy, open Zybo-Z7-Master.xdc under constraints folder

(b) Scroll to the section commented `Pmod Header JA (XADC)`

(c) Uncomment and rename ports `ja[2]`, `ja[5]`, and `ja[6]` to `tx_0[0]`, `rx_0`, and `tx_0[1]` respectively

(d) Scroll to the section commented `Pmod Header JB (Zybo Z7-20 only)`

(e) Uncomment each line and change the port name from `jb[#]` to `gpio_jb[#]` to match the port name from the diagram

(f) Scroll to the section commented `Pmod Header JC`

(g) Uncomment and rename ports `jc[5]` and `jc[6]` to `rx_1` and `tx_1`

14. Build the design

(a) Right click on the design file `(file_name.bd)` and select `create HDL Wrapper...`

(b) Let Vivado manage wrapper and auto-update and select `OK`

(c) `Generate Bitstream`

Step 13 sets the GPIO pins used by the AP and the UART TX and RX pins to the external jumpers on the board. These connections can be found in Chapter B in Figures B.1 and B.3.

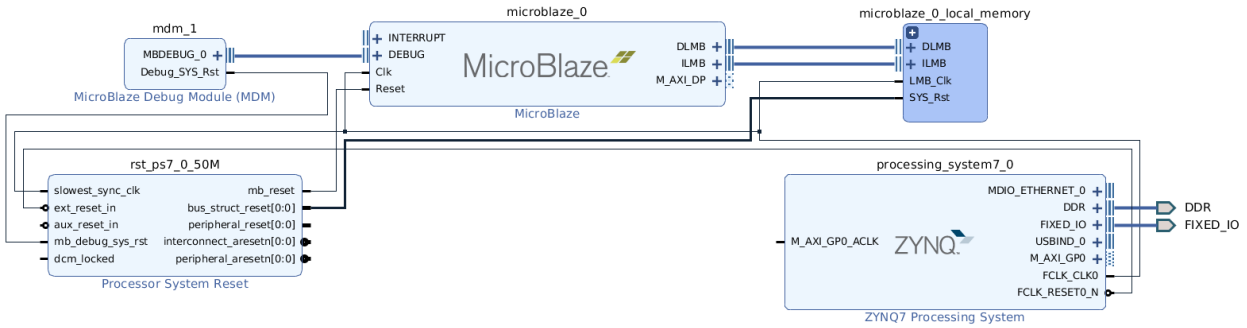

Figure A.1: Vivado block diagram for AP with a single TIOP

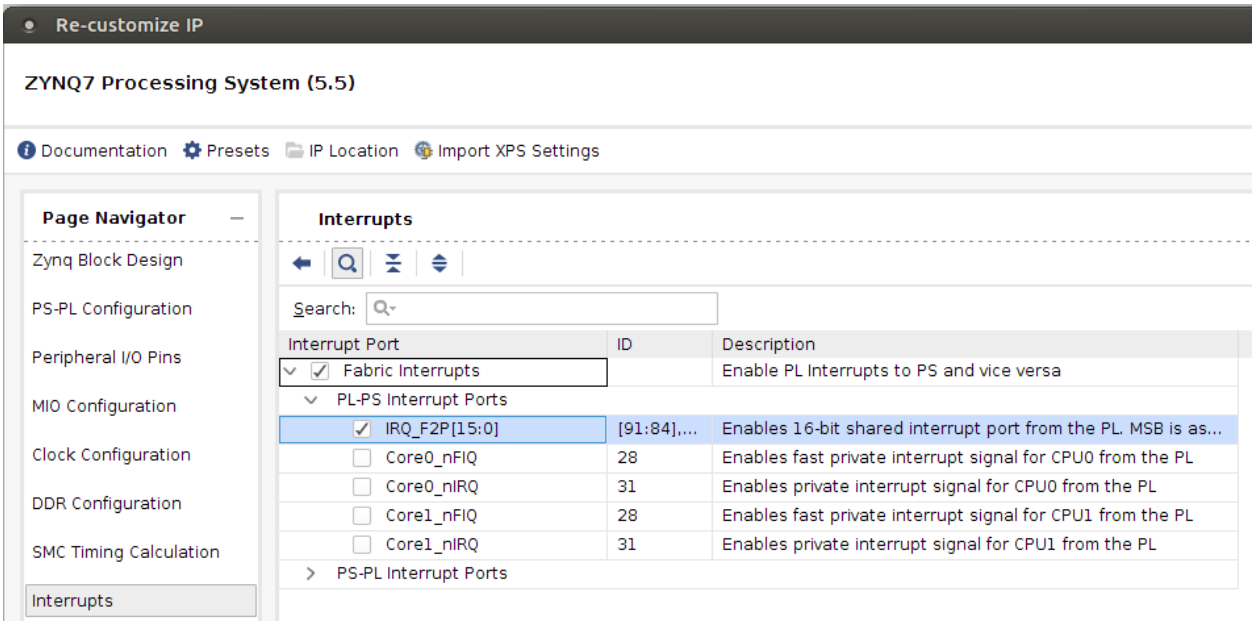

Figure A.2: Vivado menu enabling the AP to receive interrupts from the UART devices

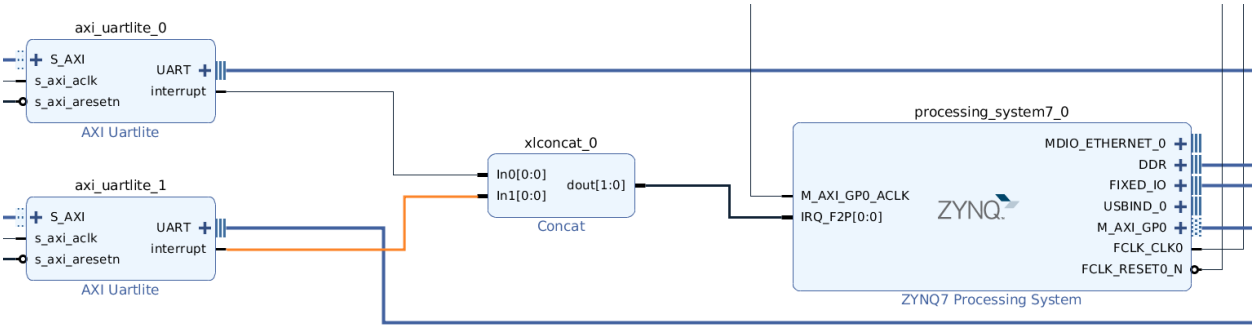

Figure A.3: Connecting the AP to receive interrupts from the UART devices

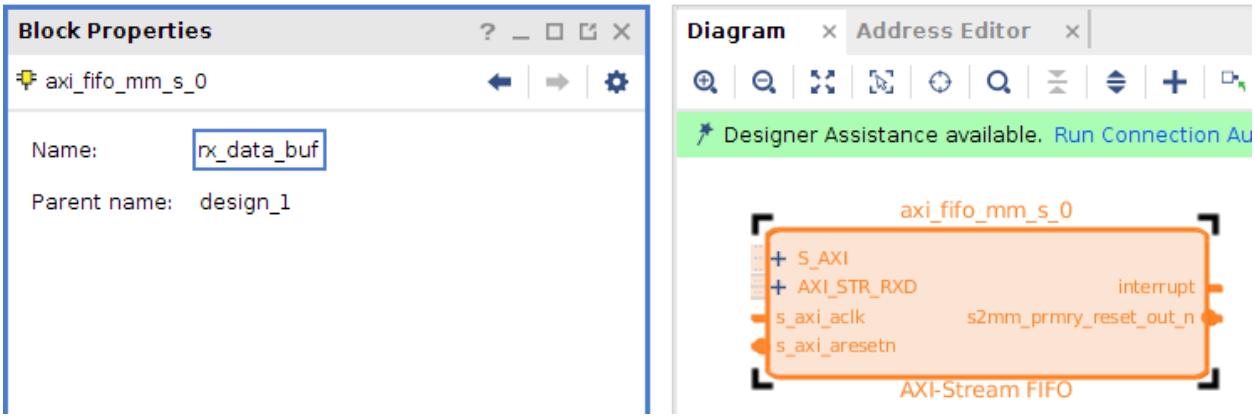

Figure A.4: Vivado menu to rename a FIFO

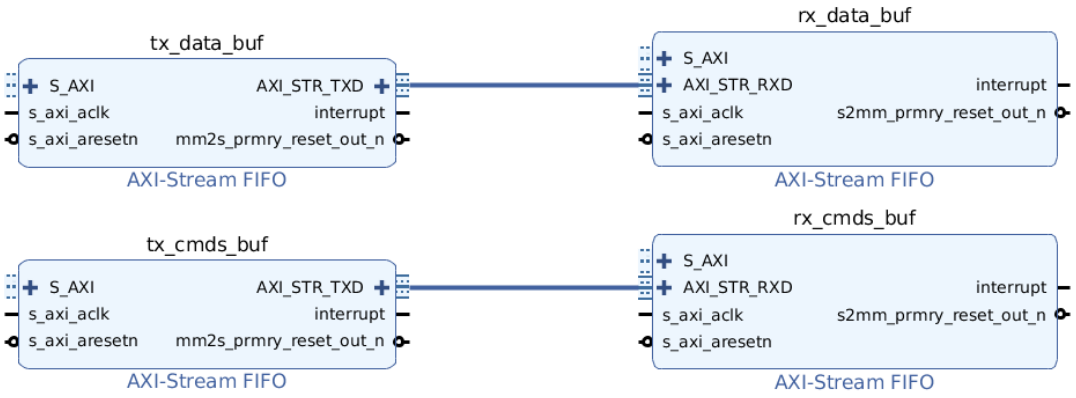

Figure A.5: Connected FIFOs

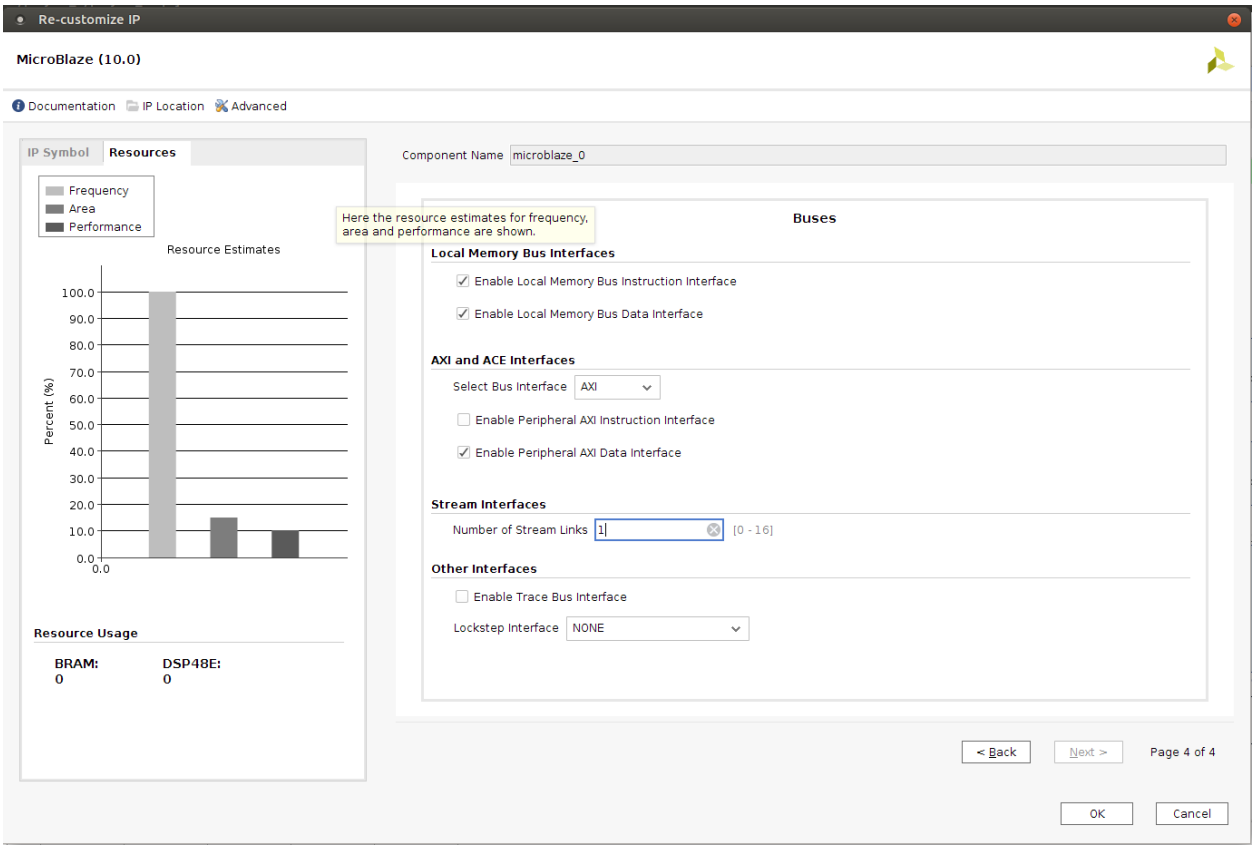

Figure A.6: Setting the MicroBlaze stream interface

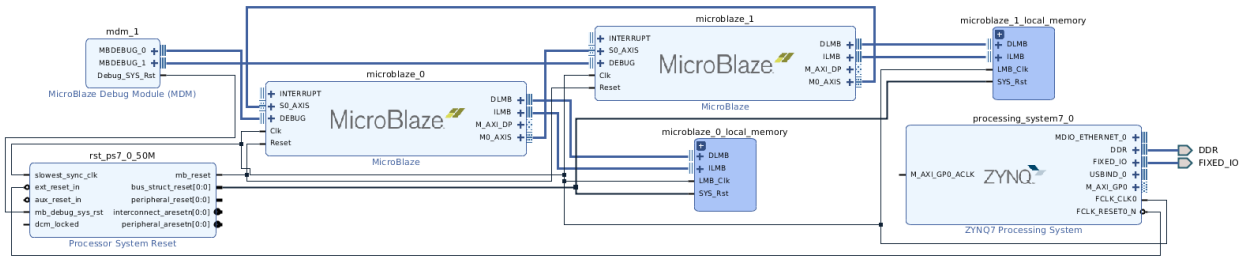

Figure A.7: Vivado block diagram for AP two TIOPs

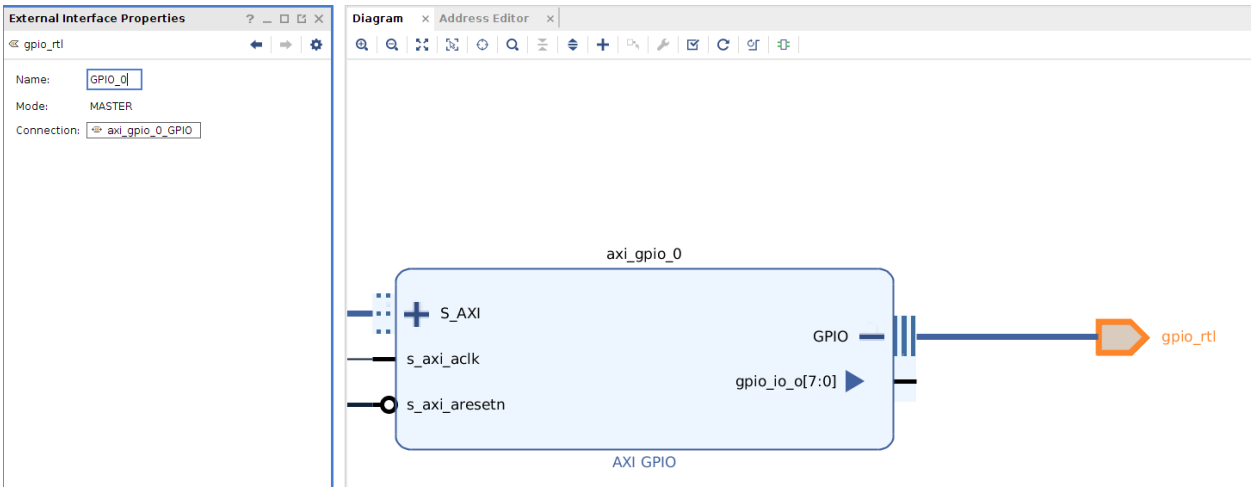

Figure A.8: Rename GPIO port with External Interface Properties Menu

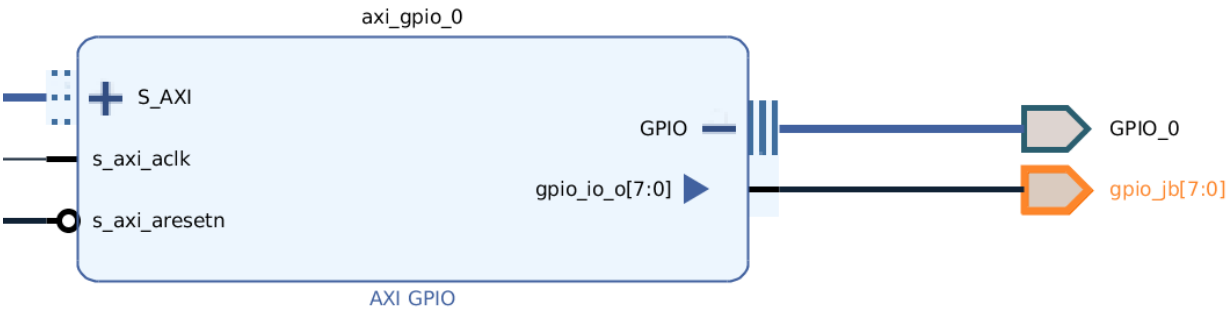

Figure A.9: Timing GPIO setup in Vivado

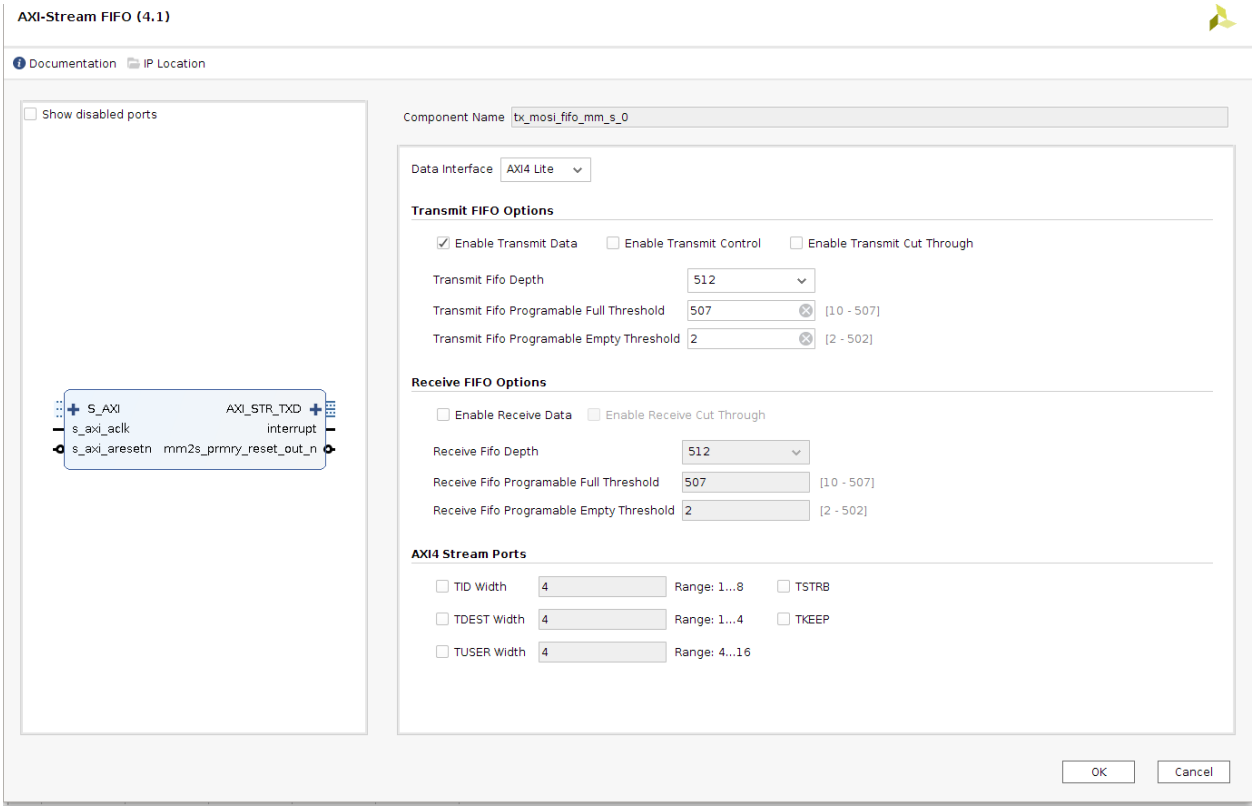

Figure A.10: Vivado AXI TX FIFO settings

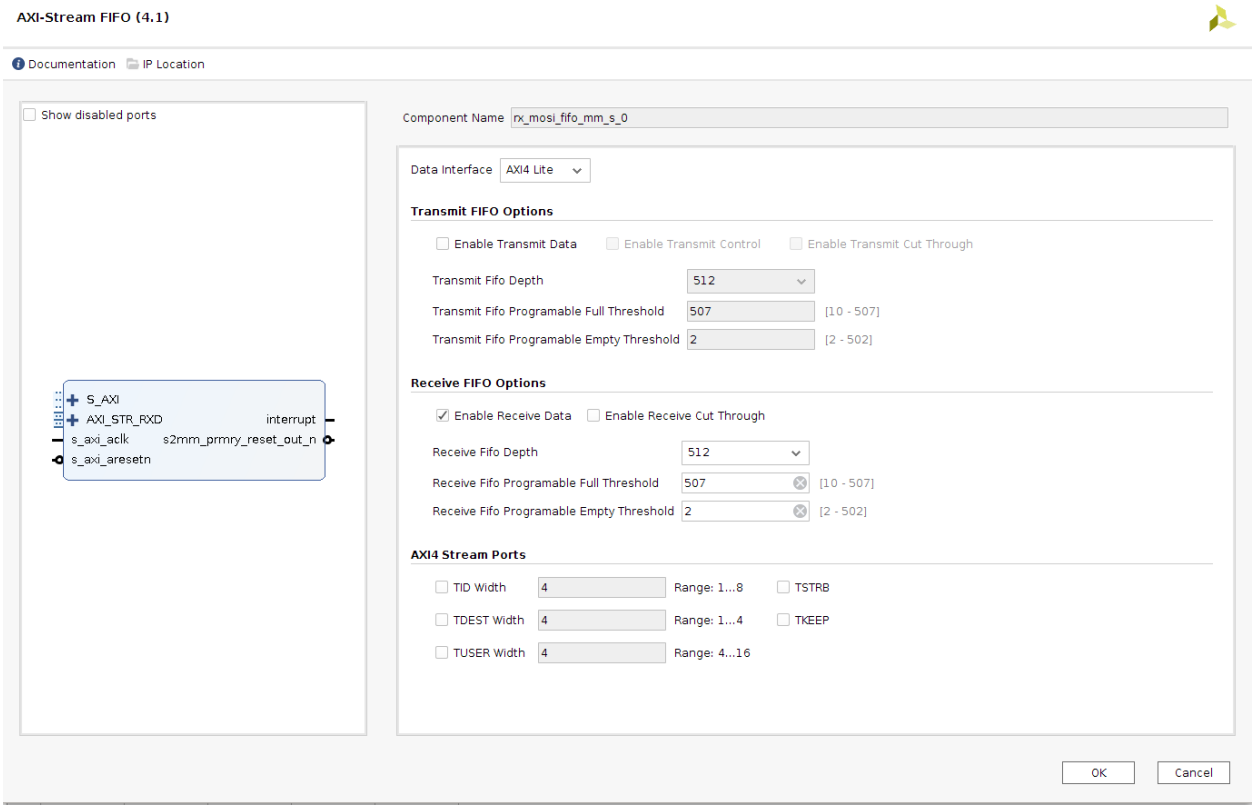

Figure A.11: Vivado AXI RX FIFO settings

A.2 Working with PetaLinux

Several steps are required to set up the development environment for PetaLinux. First, install the PetaLinux tools following the instructions in the *Petalinux Tools Reference Guide* [3]. Next create a new petalinux project with `petalinx-create -t project -n [project_name] --template zynq`. Now import the hardware specification created by Vivado. To do this use the command `petalinux-config --get-hw-description=[path_to_exported_hardware]`. The settings used for this thesis were to set the root filesystem type to SD. The project must now be built with the `petalinux-build` command. This can take some time. To add a custom application, run the command `petalinux-create -t apps -n [app_name] --enable`. This will create a custom application directory in `project-spec/meta-user/recipes-apps`. This directory contains a `.bb` file which tells the development tools which files to use to build, and where to install the program in linux. The petalinux directories and files used for each of the architectures discussed in this thesis are available on Gitlab[3].

The code for the linux application, known as controller, can be found on Gitlab[4]. To use this, copy all relevant `.c` and `.h` files, the `Makefile`, and the `build.config` file. Modify the `build.config` file to use the appropriate `#defines` and rebuild.

A.3 Programming MicroBlazes

To implement this algorithm and program the TIOP, the Vivado software development kit (SDK) must be used. To use this tool, the hardware architecture designed in Vivado must first be exported. This is done with the `File/Export/Export Hardware` command with the `Include bitstream` value checked (Figure A.12). The next step is to launch the

[3] https://gitlab.com/zybo-secure/petalinux
[4] https://gitlab.com/zybo-secure/arm-software

Vivado SDK with the `File/Launch SDK` command, making sure to select the location of exported hardware. This will generate a platform wrapper that will allow you to create new applications for the TIOPs as well as the AP.

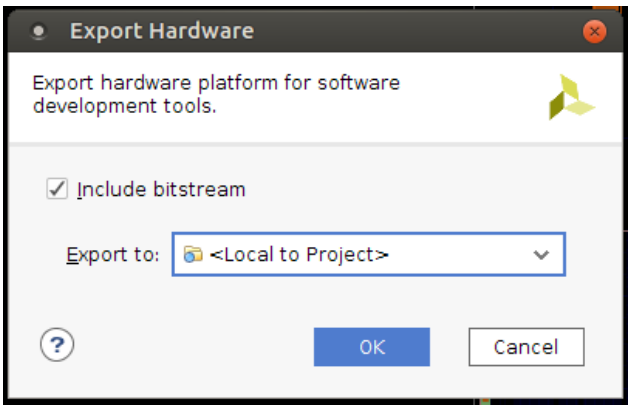

Figure A.12: Vivado export hardware to program APs

The process for creating a new TIOP application is as follows:

1. Select `File/New/Application Project`
    - `Project Name`: Desired project name
    - `Use Default Location`: Checked
    - `OS Platform`: Standalone
    - `Hardware Platform`: Name of exported hardware platform (default selection)
    - `Processor`: Name of TIOP
    - `Language`: User's choice, although `C` will be used as an example
    - `Board Support Package`: Create New
2. Select next, and set the template to `Empty Application`, then `Finish`

This will generate all the required code needed to interface with the different IP blocks. This is done through a Xilinx API which is documented in the board support package (BSP). The

next step is to create a `main.c` file to implement the code, to do this open the project in `project explorer`, navigate to the `src` folder, then right click and `add new C source file`. Name this file `main.c` and select `Finish`. Repeat this process for all TIOPs. Once the desired code has been implemented and built, the final step is to add the executable to the bitstream. This will load the executable into the TIOPs every time the system boots. To do this run the `Program FPGA` command; under `Software Configuration`, select the applicable `.elf` for each TIOP like shown in Figure A.13. This tool will "fail" if an FPGA is not connected and powered, but the new bitstream will still be created, and can be found in the software project directory under `[board-name]_wrapper_hw_platform_0/download.bit`.

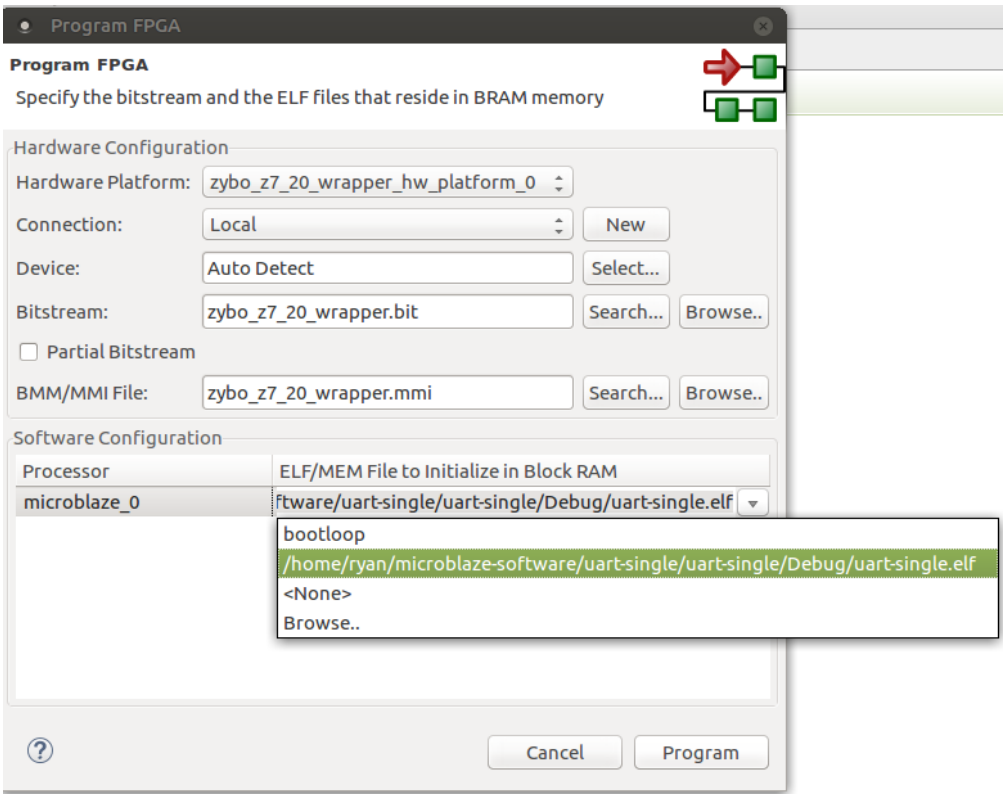

Figure A.13: Program the bitstream to include TIOP executables

The code used for either the single or multi-TIOP architectures can be found on Gitlab[5].

[5] https://gitlab.com/zybo-secure/microblaze-software

Appendix B

Components and Testbench Pictures

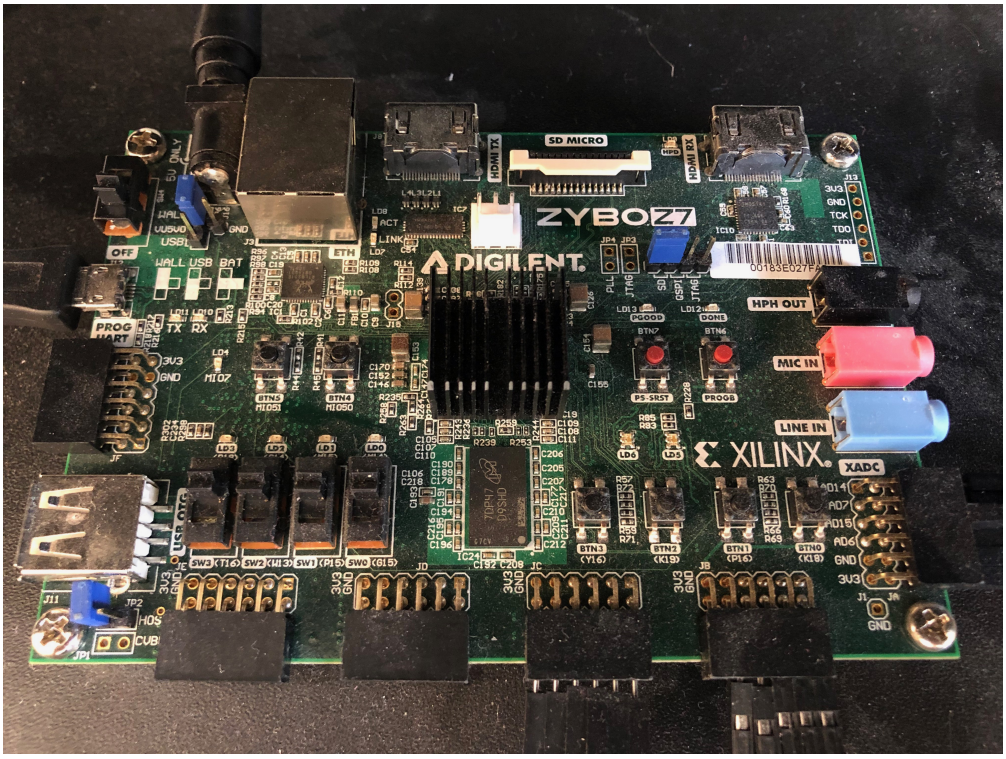

Figure B.1: Zybo Z7-20

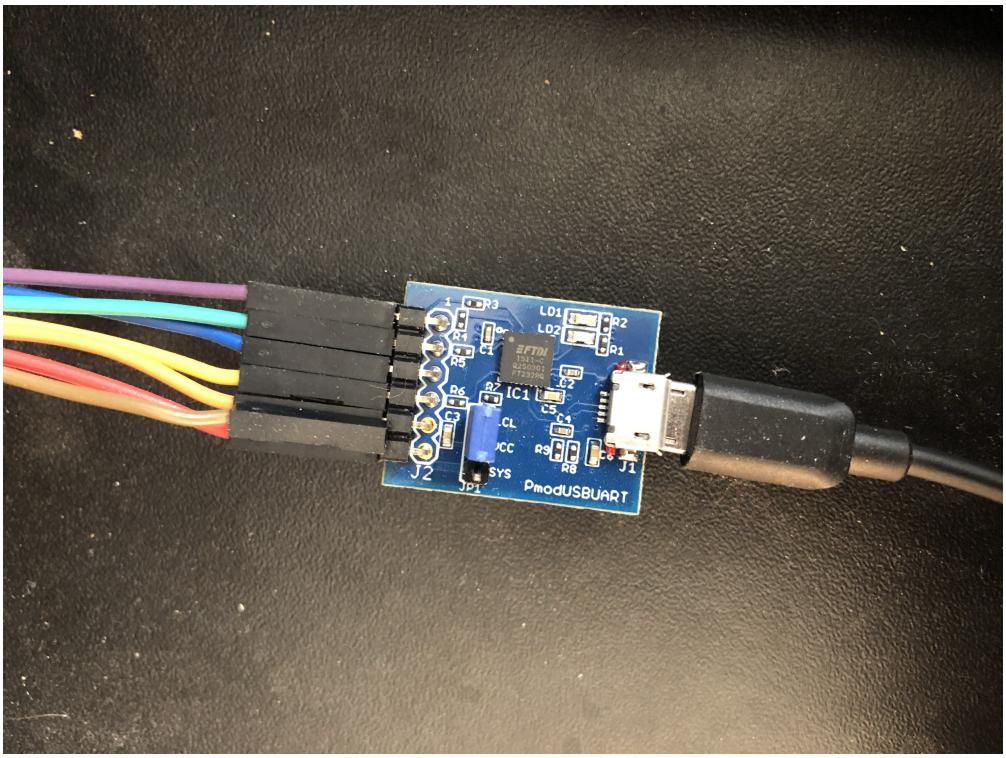

Figure B.2: UART to serial converter

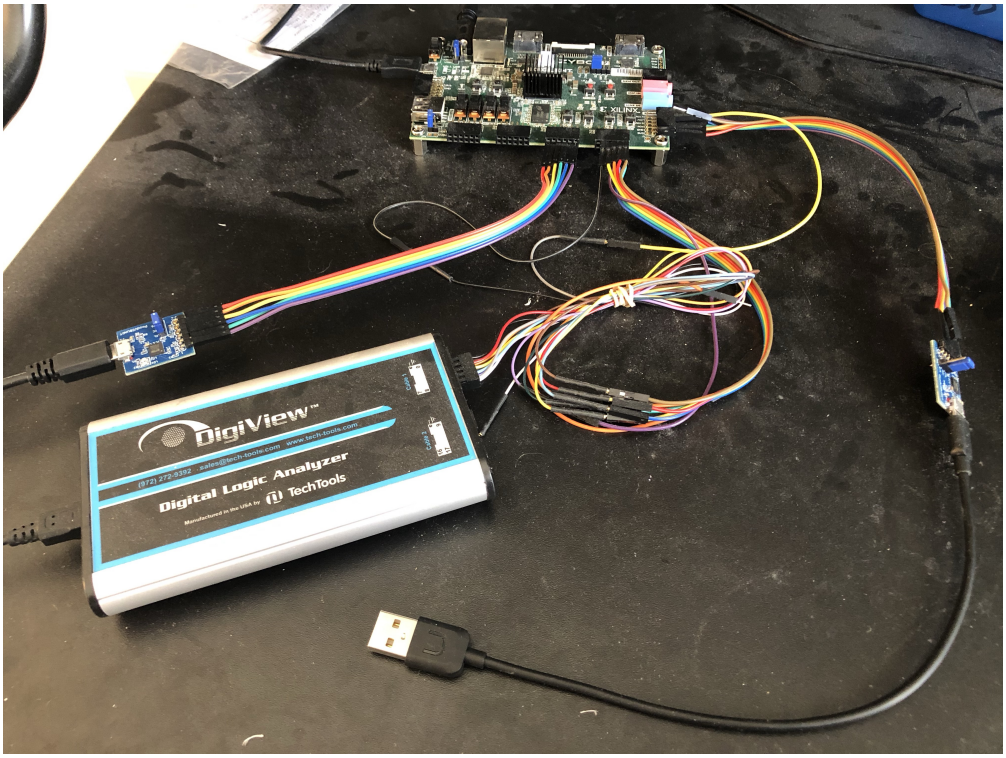

Figure B.3: Testbench setup

Figure B.4: DigiView logic analyzer

Appendix C

Code Listings

Listing C.1: Promela code for TIOP model

```
/* single i/o processor psuedocode + application processor to send
    data */

/* data -> sensor information */
/* cmds -> actuator commands */

/* c = controller, p = peripherals */
mtype:TIOP_STATE = {tiop_init, recv_data, send_data, recv_cmds,
        send_cmds, monitor, rcf, error};

/* Booleans for different tests */
/* Test the application is running, if not wd should trigger */
bool AP_RUNNING;
/* Test the commands comming from the AP are valid */
bool CMDS_SAFE;
/* Test the data coming from the sensors is OK */
bool DATA_SAFE;

/* LTL Statements */
```

```
/* Safety */
/* Test tiop never enters error state */
ltl p1 { [](!tiop@STATE_ERROR) }

/* Test if all good, nothing bad happens */
ltl p2 { []((AP_RUNNING && CMDS_SAFE && DATA_SAFE) ->
        [](!tiop@RCF_ACTIVATED)) }

/* Test all bad settings cause errors */
ltl p3 { []((!AP_RUNNING) -> <>(tiop@RCF_ACTIVATED)) }

ltl p4 { []((!CMDS_SAFE) -> <>(tiop@RCF_ACTIVATED)) }

ltl p5 { []((!DATA_SAFE) -> <>(tiop@RCF_ACTIVATED)) }

/* Liveness */
/* Test that if sensor data is available, actuator commands will
   be sent */
ltl p6 { []((tiop@DATA_RECEIVED) -> <>(tiop@COMMANDS_SENT)) }

#define MAX_DATA 2016
#define MIN_DATA -2048
#define MAX_CMDS 2016
#define MIN_CMDS -2048
#define SAFETY_VAL 0

#define NUM_VARS 2

chan uart_data = [NUM_VARS] of {short};
chan uart_cmds = [NUM_VARS] of {short};
chan fifo_data = [NUM_VARS] of {short};
chan fifo_cmds = [NUM_VARS] of {short};

active proctype tiop() {
```

```
    mtype:TIOP_STATE state = tiop_init;
    mtype:TIOP_STATE prev_state;
    /* tx byte, rx byte and counter for timer */
    short data[NUM_VARS], cmds[NUM_VARS];
    bool data_error, cmds_error, wd_triggered;
    int i;
    do
    :: (state == tiop_init) ->
        prev_state = tiop_init;
        state = recv_data;
        data_error = false;
        cmds_error = false;
        wd_triggered = false;
    :: (state == recv_data) ->
        for (i : 0 .. (NUM_VARS - 1)) {
            uart_data?data[i];
        }
DATA_RECEIVED:
        prev_state = recv_data;
        state = monitor;
    :: (state == send_data) ->
        for (i : 0 .. (NUM_VARS - 1)) {
            fifo_data!data[i];
        }
        prev_state = send_data;
        state = recv_cmds;
    :: (state == recv_cmds) ->
        prev_state = state;
        for (i : 0 .. (NUM_VARS - 1)) {
            if
            :: fifo_cmds?cmds[i] ->
                wd_triggered = false;
                state = monitor;
            :: timeout ->
```

```
                wd_triggered = true;
                state = rcf;
                break;
            fi
        }
    :: (state == send_cmds) ->
        for (i : 0 .. (NUM_VARS - 1)) {
            uart_cmds!cmds[i];
        }
COMMANDS_SENT:
        prev_state = send_cmds;
        state = recv_data;
    :: (state == monitor) ->
        if
        :: (prev_state == recv_data) ->
            state = send_data;
            for (i : 0 .. (NUM_VARS - 1)) {
                if
                :: (data[i] > MAX_DATA || data[i] < MIN_DATA) ->
                    data_error = true;
                    break;
                :: else ->
                    data_error = data_error | false;
                fi
            }
        :: (prev_state == recv_cmds) ->
            for (i : 0 .. (NUM_VARS - 1)) {
                if
                :: (cmds[i] > MAX_CMDS || cmds[i] < MIN_CMDS) ->
                    cmds_error = true;
                    break;
                :: else ->
                    cmds_error = false;
                fi
```

```
            }

            if
            :: (wd_triggered || data_error || cmds_error) ->
                state = rcf;
            :: else ->
                state = send_cmds;
            fi
        :: else ->
            state = error;
        fi
        prev_state = monitor;
    :: (state == rcf) ->
RCF_ACTIVATED:
        for (i : 0 .. (NUM_VARS - 1)) {
            cmds[i] = SAFETY_VAL;
        }
        prev_state = rcf;
        state = send_cmds;
    :: (state == error) ->
STATE_ERROR:
        prev_state = state;
        break;
    od;
}

mtype:AP_STATE = {ap_init, ap_recv, ap_pid, ap_send, ap_error};

active proctype ap() {
    short data[NUM_VARS], cmds[NUM_VARS];
    int i;
  mtype:AP_STATE ap_state = ap_init;
    do
    :: (ap_state == ap_init) ->
```

```
        ap_state = ap_recv;
    :: (ap_state == ap_recv) ->
        for (i : 0 .. (NUM_VARS - 1)) {
            fifo_data?data[i];
        }
        ap_state = ap_pid;
    :: (ap_state == ap_pid) ->
        for (i : 0 .. (NUM_VARS - 1)) {
            if
            :: (CMDS_SAFE) ->
                    cmds[i] = data[i];
            :: else ->
                if
                :: cmds[i] = MAX_CMDS + 1;
                :: cmds[i] = MIN_CMDS - 1;
                fi
            fi
        }
        if
        :: (AP_RUNNING) -> /*| \label{tiop-lst:no-ap} |*/
            ap_state = ap_send;
        }
        :: else ->
            ap_state = ap_recv;
        fi
    :: (ap_state == ap_send) ->
        for (i : 0 .. (NUM_VARS - 1)) {
            fifo_cmds!cmds[i];
        }
        ap_state = ap_recv;
    :: (ap_state == ap_error) ->
        break;
    od
}
```

```
active proctype sensors() {
    short data[NUM_VARS];
    int i;
    do
    :: true ->
        for (i : 0 .. (NUM_VARS - 1)) {
            if
            :: (DATA_SAFE) ->
                data[i] = SAFETY_VAL;
            :: else ->
                if
                :: data[i] = MAX_DATA + 1;
                :: data[i] = MIN_DATA - 1;
                fi
            fi
            uart_data!data[i];
        }
    od
}

active proctype actuators() {
    short cmds[NUM_VARS];
    int i;
    do
    :: (true) ->
        for (i : 0 .. (NUM_VARS - 1)) {
            uart_cmds?cmds[i];
        }
    od
}
```

Listing C.2: Modex definition file for TIOP code

```
%X -L main.lut -a main
```

```
%X -e Monitor
%X -e RCF
%H
typedef short int16_t;
typedef signed char int8_t;
typedef unsigned short uint16_t;
typedef unsigned long u32;
typedef unsigned char u8;
typedef int XUartLite;
typedef int XLlFifo;
typedef int XTmrCtr;
%%
%D
typedef short int16_t;
typedef signed char int8_t;
typedef unsigned short uint16_t;
typedef unsigned long u32;
typedef unsigned char u8;
typedef int XUartLite;
typedef int XLlFifo;
typedef int XTmrCtr;
%%
%L
LEDSet(...  comment
retval=UartSetup(...  retval = 0
retval=FifoSetup(...  retval = 0
XTmrCtr_Initialize(...  comment
XTmrCtr_SetOptions(...  comment
XTmrCtr_Start(... comment
XLlFifo_RxReset(... comment
XLlFifo_IntClear(...  comment
%%
%L main
Declare int  j  main
```

```
Declare short tmp    main
RecvData(...   for (j : 0 .. (NUM_DATA_VALS - 1)) { \
          uart_data?tmp; \
          c_code { Pp_main->data[Pp_main->j] = Pp_main->tmp; }; \
        }
SendData(...   for (j : 0 .. (NUM_DATA_VALS - 1)) { \
          c_code { Pp_main->tmp = Pp_main->data[Pp_main->j]; }; \
          fifo_data!tmp; \
        }
wd_triggered=RecvCmds(... c_code { Pp_main->wd_triggered = 0; }; \
        for (j : 0 .. (NUM_CMDS_VALS - 1)) { \
          if \
          ::fifo_cmds?tmp -> \
            c_code { Pp_main->cmds[Pp_main->j] = Pp_main->tmp; }; \
               \
          ::timeout -> \
            c_code { Pp_main->wd_triggered++; }; \
          fi \
        }
SendCmds(...   for (j : 0 .. (NUM_CMDS_VALS - 1)) { \
          c_code { Pp_main->tmp = Pp_main->cmds[Pp_main->j]; }; \
          uart_cmds!tmp; \
        }
%%
%P
#define NUM_DATA_VALS 2
#define NUM_CMDS_VALS 2
#define MAX_DATA 2016
#define MIN_DATA -2048
#define MAX_CMDS 2016
#define MIN_CMDS -2048
#define SAFETY_VAL 0

/* Test the application is running, if not wd should trigger */
```

```
bool AP_RUNNING;
/* Test the commands comming from the AP are valid */
bool CMDS_SAFE;
/* Test the data coming from the sensors is OK */
bool DATA_SAFE;

/* LTL Statements */
/* Safety */
/* Test tiop never enters error state */
ltl p1 { [](!p_main@STATE_ERROR) }

/* Test if all good, nothing bad happens */
ltl p2 { []((AP_RUNNING && CMDS_SAFE && DATA_SAFE) ->
        [](!p_main@RCF_ACTIVATED)) }

/* Test all bad settings cause errors */
ltl p3 { []((!AP_RUNNING) -> <>(p_main@RCF_ACTIVATED)) }

ltl p4 { []((!CMDS_SAFE) -> <>(p_main@RCF_ACTIVATED)) }

ltl p5 { []((!DATA_SAFE) -> <>(p_main@RCF_ACTIVATED)) }

/* Liveness */
/* Test that if sensor data is available, actuator commands will
   be sent */
ltl p6 { []((p_main@DATA_RECEIVED) -> <>(p_main@COMMANDS_SENT)) }

chan uart_data = [NUM_DATA_VALS] of {short};
chan uart_cmds = [NUM_CMDS_VALS] of {short};
chan fifo_data = [NUM_DATA_VALS] of {short};
chan fifo_cmds = [NUM_CMDS_VALS] of {short};

mtype:AP_STATE = {ap_init, ap_recv, ap_pid, ap_send, ap_error};
```

```
active proctype ap() {
    short data[NUM_DATA_VALS], cmds[NUM_CMDS_VALS];
    int i;
        mtype:AP_STATE ap_state = ap_init;
    do
    :: (ap_state == ap_init) ->
        ap_state = ap_recv;
    :: (ap_state == ap_recv) ->
        for (i : 0 .. (NUM_DATA_VALS - 1)) {
            fifo_data?data[i];
        }
        ap_state = ap_pid;
    :: (ap_state == ap_pid) ->
        for (i : 0 .. (NUM_CMDS_VALS - 1)) {
            if
            :: (CMDS_SAFE) ->
                    cmds[i] = data[i];
            :: else ->
                if
                :: cmds[i] = MAX_CMDS + 1;
                :: cmds[i] = MIN_CMDS - 1;
                fi
            fi
        }
        if
        :: (AP_RUNNING) ->
            ap_state = ap_send;
        :: else ->
            ap_state = ap_recv;
        fi
    :: (ap_state == ap_send) ->
        for (i : 0 .. (NUM_CMDS_VALS - 1)) {
            fifo_cmds!cmds[i];
        }
```

```
        ap_state = ap_recv;
    :: (ap_state == ap_error) ->
        break;
    od
}

active proctype sensors() {
    short data[NUM_DATA_VALS];
    int i;
    do
    :: true ->
        for (i : 0 .. (NUM_DATA_VALS - 1)) {
            if
            :: (DATA_SAFE) ->
                data[i] = SAFETY_VAL;
            :: else ->
                if
                :: data[i] = MAX_DATA + 1;
                :: data[i] = MIN_DATA - 1;
                fi
            fi
            uart_data!data[i];
        }
    od
}

active proctype actuators() {
    short cmds[NUM_CMDS_VALS];
    int i;
    do
    :: (true) ->
        for (i : 0 .. (NUM_CMDS_VALS - 1)) {
            uart_cmds?cmds[i];
        }
```

```
        od
}
%%
```