A Development Platform to Evaluate UAV Runtime Verification Through Hardware-in-the-loop Simulation

Akhil A. Rafeeq

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Cameron D. Patterson, Chair

Changwoo Min

Matthew Hicks

May 12, 2020

Blacksburg, Virginia

Keywords: Runtime Verification, UAV, Hardware-in-the-loop Simulation, Hardware and Software Monitors, PSoC, FPGA

A Development Platform to Evaluate UAV Runtime Verification Through Hardware-in-the-loop Simulation

Akhil A. Rafeeq

(ABSTRACT)

The popularity and demand for safe autonomous vehicles are on the rise. Advances in semiconductor technology have led to the integration of a wide range of sensors with high-performance computers, all onboard the autonomous vehicles. The complexity of the software controlling the vehicles has also seen steady growth in recent years. Verifying the control software using traditional verification techniques is difficult and thus increases their safety concerns.

Runtime verification is an efficient technique to ensure the autonomous vehicle's actions are limited to a set of acceptable behaviors that are deemed safe. The acceptable behaviors are formally described in linear temporal logic (LTL) specifications. The sensor data is actively monitored to verify its adherence to the LTL specifications using monitors. Corrective action is taken if a violation of a specification is found.

An unmanned aerial vehicle (UAV) development platform is proposed for the validation of monitors on configurable hardware. A high-fidelity simulator is used to emulate the UAV and the virtual environment, thereby eliminating the need for a real UAV. The platform interfaces the emulated UAV with monitors implemented on configurable hardware and autopilot software running on a flight controller. The proposed platform allows the implementation of monitors in an isolated and scalable manner. Scenarios violating the LTL specifications can be generated in the simulator to validate the functioning of the monitors.

A Development Platform to Evaluate UAV Runtime Verification Through Hardware-in-the-loop Simulation

Akhil A. Rafeeq

(GENERAL AUDIENCE ABSTRACT)

Safety is one of the most crucial factors considered when designing an autonomous vehicle. Modern vehicles that use a machine learning-based control algorithm can have unpredictable behavior in real-world scenarios that were not anticipated while training the algorithm. Verifying the underlying software code with all possible scenarios is a difficult task.

Runtime verification is an efficient solution where a relatively simple set of monitors validate the decisions made by the sophisticated control software against a set of predefined rules. If the monitors detect an erroneous behavior, they initiate a predetermined corrective action. Unmanned aerial vehicles (UAVs), like drones, are a class of autonomous vehicles that use complex software to control their flight. This thesis proposes a platform that allows the development and validation of monitors for UAVs using configurable hardware. The UAV is emulated on a high-fidelity simulator, thereby eliminating the time-consuming process of flying and validating monitors on a real UAV. The platform supports the implementation of multiple monitors that can execute in parallel. Scenarios to violate rules and cause the monitors to trigger corrective actions can easily be generated on the simulator.

Dedication

To my family

Acknowledgments

First and foremost, I thank Almighty for blessing me with the knowledge and skills required to accomplish the thesis. I would like to thank Dr. Cameron Patterson for providing me the opportunity to work on this project and for his advice on academics and career. His constant guidance and support have made my journey through the Master's degree a smooth one. Completing the thesis on time would not have been possible without his prompt feedback. I aspire to acquire Dr. Patterson's quality of paying attention to detail and his ability to articulate complex ideas in uncomplicated words. I would also like to thank my committee members, Dr. Changwoo Min and Dr. Matthew Hicks. I enjoyed attending their classes, and I believe the lessons learned there would make my career fruitful.

I thank my parents and sisters for relentlessly supporting my decision to pursue a Master's degree in the United States. They have always been there when I wanted them. I am grateful to my cheerful nephews and niece, who have always lightened my mood during stressful times. Special thanks to my friends Mazher Ali Khan and Aditya Warnulkar for their encouraging words, Syed Azhar Ali and Parthkumar Modi for being wonderful flatmates, and Nahmed Nissar and Shashank Raghuraman for the fun times in Blacksburg. I would also like to thank Lakshman for his contributions to this project.

The author gratefully acknowledges Ms. Susan J. DeGuzman, Director, USN/USMC Airworthiness and CYBERSAFE Office, Naval Air Systems Command. This research was funded by NAVAIR Contract # N00421-16-2-0001. The author also acknowledges Mr. Richard E. Adams, DoN Chief Airworthiness Engineer for UAS & Targets, USN/USMC Airworthiness & Cybersafe Office, Naval Air Systems Command for his technical advice and guidance.

This work has also been supported by the Center for Unmanned Aircraft Systems (C-UAS),

a National Science Foundation Industry/University Cooperative Research Center (I/UCRC) under NSF award No. CNS-1650465 along with significant contributions from C-UAS industry members.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Advances in the fields of semiconductor technology, material science, computer vision, and machine learning have accelerated the deployment of autonomous vehicles like cars, trucks, and UAVs. The onboard computers used in modern autonomous vehicles are high performance, low power, reliable and cost-effective. With the availability of higher processing power, sophisticated algorithms to control the vehicle can now run on these onboard computers.

The market for UAVs is expected to grow from USD 19.3 billion in 2019 to USD 45.8 billion by 2025 [3]. UAVs, which were once limited to military applications, are now used in many civil and commercial applications like inspection, package delivery, remote sensing, and entertainment. The adoption of UAV technology in these domains can be attributed to the abundant availability of feature-rich, low-cost, and easy-to-use UAVs. Improvements in Micro Electro-Mechanical Systems (MEMS), brushless motors, durable and light-weight materials, and high energy density batteries have accelerated the miniaturization of UAVs, making them more accessible.

To ensure the safe use of autonomous vehicles, the software controlling the vehicle needs to be thoroughly verified. An untested scenario can leave a software bug hidden in the system that

may eventually cause an accident, potentially leading to loss of life and property. Machine learning algorithms rely on training datasets to learn to control the vehicle autonomously. Training datasets are limited. Any real-world scenario that is significantly different from the training dataset may cause the machine learning algorithm to behave unpredictably.

Using traditional software verification techniques to verify the software code becomes a challenge as newer complex algorithms get integrated into the codebase. An alternative approach is to use runtime verification to monitor whether the autonomous vehicle conforms to a set of predefined, acceptable behaviors that are formally expressed as LTL specifications. With UAVs serving as an instance of autonomous vehicles, a platform is constructed that enables the development, implementation, and evaluation of runtime verification of UAVs using a programmable system-on-chip (PSoC). The monitors are implemented in field-programmable gate array (FPGA) resources of the PSoC and are scalable and isolated. The platform uses a hardware-in-the-loop (HITL) simulation technique and eliminates the need for a real UAV. The addition of the monitoring system has minimal timing impact on HITL simulations.

1.2 Contributions

The author's contributions to this project are:

1. Set up the Linux environment to run HITL simulations on two simulators with the autopilot software running on the flight controller hardware.

2. Implemented and verified the hardware needed to interface the simulator with the PSoC and the flight controller.

3. Created a new software protocol for inter-processor communication within the PSoC.

4. Developed the software that parses the sensor data sent by the simulator and communicates it to the monitors.

5. Optimized the overall latency, FPGA resource utilization, and power consumption of the UAV runtime verification platform.

1.3 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 presents a brief introduction to topics relevant to the thesis and describes prior work accomplished in UAV runtime verification. The high-level architectural choices available and the reasons for choosing a particular architecture are presented in Chapter 3. The hardware and software implementation of various components of the platform are discussed in detail in Chapter 4. Chapter 5 discusses the various metrics used for evaluating the platform and compares the results with the optimized implementation of the platform. Chapter 6 suggests future work and concludes the thesis.

Chapter 2

Background

This chapter presents a brief discussion of digital systems and their verification techniques. Hardware options available for implementing runtime verification are presented. The HITL simulation technique and its advantages are discussed. Various open source tools and subsystems used in this thesis are also described.

2.1 Digital Systems and Verification

Embedded digital systems are increasingly ubiquitous in modern societies. The two primary components of such systems are the digital hardware and the software that runs on this hardware. The processor, memory, and platform-specific peripheral circuitry constitute the hardware. Considerations such as power, area, and performance dictate the choice of the processor. The power-efficient Arm Cortex-M series processors are more suited to applications requiring little processing, such as microcontrollers for embedded systems. Applications like general-purpose computers and mobile phones that require more substantial processing power use Arm Cortex-A series processors or x86-based Intel processors. The software that runs on these processors is generally more complex.

With advancements in communication technology like 4G LTE, frequent upgrades to the software components are possible. Thus, the possibility of faulty and incompletely verified software code entering the system have also increased. In real-time embedded applications

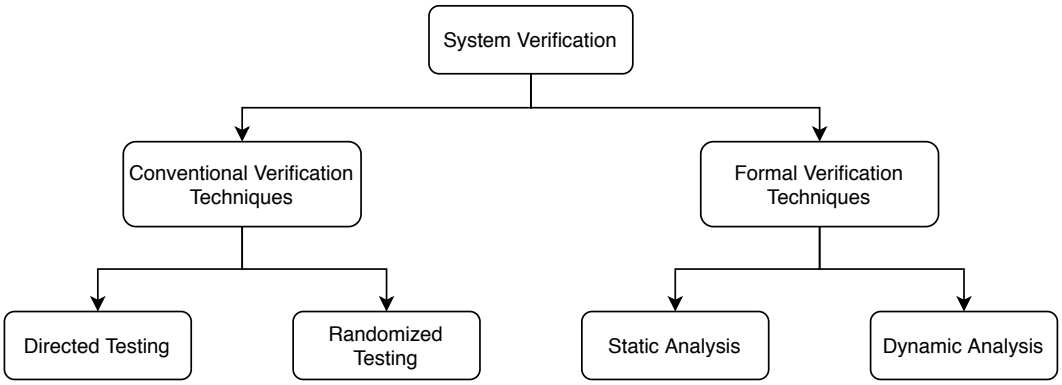


Figure 2.1: Hierarchy of verification techniques

like a UAV, it is crucial that the software controlling the UAV functions in a correct, secure, and reliable manner. Various measures are taken at different stages of product development to ensure the end product, comprising the hardware and software, is free of bugs. A *system under test* in this context refers to the hardware or software being verified. Figure 2.1 shows the hierarchy of verification techniques available for verifying the functionality of hardware and software. The following sections briefly discuss these techniques.

2.1.1 Conventional Verification Techniques

In conventional verification techniques, test inputs are generated and applied to the system under test. The same test inputs are provided to a functionally accurate reference model to obtain the reference results. The response generated by the system under test is compared with the reference results. The test is declared as a passing test if the results match or a failing test if the results do not match. A verification metric called coverage measures the extent to which the system under test is verified. System designers strive to get 100% coverage, which may not be easy for larger systems.

- **Directed Testing [30]:** In directed testing, the test inputs target specific blocks of

the system under test. For example, while verifying a hardware arithmetic-logic unit (ALU), a directed test can exercise the special case of division by zero.

- **Randomized Testing:** Creating a directed test for all combinations of inputs becomes difficult as the number of inputs increase. Hence, randomized testing is used where the test inputs are randomly generated. In some cases, completely random inputs are not desirable either. In such cases, a technique called constrained random verification is used where randomization occurs within a set of constraints [19].

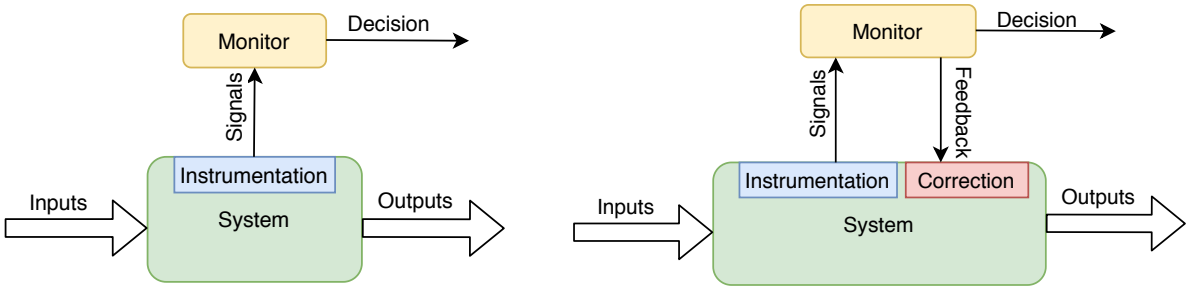
2.1.2 Formal Verification Techniques

Formal verification techniques aim at proving or disproving the correctness of systems with respect to a certain formal specification or property.

- **Static Analysis:** The static analysis includes formal methods like theorem proving and model checking, where the correctness of the hardware or software design is proven using rigorous mathematical and logical reasoning. In theorem proving, also known as deductive verification, a set of mathematical formulae are generated from the system under test and its specifications that need a mathematical proof to establish the correctness of the system under test [13]. In model checking, the model of the system under test is exhaustively explored to check whether the specifications hold in all states and transitions of the model [5]. The main difference between theorem proving and model checking is that theorem proving is largely a manual technique, whereas model checking is performed by algorithms. Static analysis is performed at design time, without actually executing the system.

- **Dynamic Analysis:** Dynamic analysis or runtime verification is very similar to static analysis, except that the analysis is carried out when the system is under execution. In-

strumentation codes called monitors are generated that continuously examine whether a formally specified property is true. As shown in Figure 2.2a, monitors by themselves can only detect an incorrect behavior called a *fault* and typically do not influence or correct the behavior of the system. A robust system can be built using the results of the monitor to perform a corrective action, as shown in Figure 2.2b. The corrective action can either reverse the effects of the fault if feasible or minimize the consequences of the fault.



(a) Simple monitoring system (b) Monitoring system with correction

Figure 2.2: Block diagram of runtime verification systems

A system is said to be *complex* if its behavior cannot be easily described. When a system contains many subsystems and has numerous interactions between these subsystems, describing the behavior of the whole system is nontrivial. The total number of inputs, the interaction between the sub-systems, and the number of previous inputs that determine the current output gives a measure of the *complexity* of the system. As the complexity of the system grows, static analysis techniques become difficult due to a problem known as the state explosion problem [39]. Hence static analysis is mostly used to verify isolated software or hardware components. The floating-point unit (FPU) within a CPU is an ideal candidate for verification using static analysis [17]. Runtime verification, on the other hand, can still be used with complex systems like UAVs [36].

The application of runtime verification on real-time embedded systems has been studied by

[2], [40], [4], [31], [35]. Leucker et al. give a brief account of runtime verification, contrast it with other techniques like model checking and randomized testing, and extend the idea of runtime verification to monitor-oriented programming [20]. Instrumentation techniques and the problem of monitorability are presented in [2]. Watterson et al. present the advantages and limitations of using internal hardware probes, instrumentation code in software, and their combination with regards to non-intrusion and processing overhead [40]. Cassar et al. present and classify various instrumentation methodologies into monitoring categories that are entirely asynchronous, entirely synchronous, and a combination of the two [4].

2.2 Hardware Options for Runtime Verification

Monitors for runtime verification can be implemented as purely software monitors, purely hardware monitors, or a combination of software and hardware monitors. The hardware for implementing monitors can be mainly classified into two categories: non-configurable hardware such as a microcontroller and system-on-chip (SoC), and configurable hardware such as an FPGA and PSoC. Some products are available which have both the non-configurable and configurable hardware. Only software monitors can be implemented on non-configurable hardware. The following sections describe these options.

2.2.1 Microcontroller

Microcontrollers (MCUs) are a class of computers that are highly optimized for applications requiring less computation and are often used in simple embedded applications. Typically an MCU contains a single-core CPU, a small on-chip memory, and I/O peripherals. While modern CPUs support 64-bit operations, MCUs generally come with an 8-bit to 32-bit CPU.

The operating frequency of MCU is typically less than 100 MHz. Another common feature of an MCU is its native support for bit operations. Due to their low cost, it is easy to find products with more than one MCU (such as automobiles) where each MCU performs a dedicated operation. Software monitors can be implemented on a dedicated MCU. The signals from the system under test need to be instrumented and communicated to the MCU which uses these signals to evaluate the monitors.

2.2.2 SoC

A SoC is a more powerful version of an MCU. An SoC typically includes multiple processing cores, internal and external memories, and peripherals, all of which are connected through a central interface called a network-on-chip (NoC). Figure 2.3 shows the block diagram of a typical SoC. The processing cores can be one or more microcontroller, microprocessor (CPU), graphics processing unit (GPU), digital signal processing (DSP) core, or an application-specific instruction-set processor (ASIP) core. Depending on the application, the memory can be internal, external, or both. SoCs can contain a large number of peripheral interfaces such as USB, Ethernet, SPI, HDMI, I2C, UART, eMMC, NOR and NAND flash controller, analog-to-digital and digital-to-analog converters, and Wi-Fi. SoCs are used in devices that support running general-purpose applications like mobile phones. SoCs consume significantly higher power when compared to MCUs and can operate at frequencies in excess of 1 GHz.

In a multi-core SoC, software monitors can be implemented on a dedicated core. Signals from one or more cores can be instrumented and sent to the core running the monitor. Other peripheral interfaces present on the SoC can also be used to monitor code running outside the SoC. Kane et al. present an algorithm called EgMon that passively monitors the broadcast bus of an autonomous research vehicle (ARV) and checks for violations of properties written

in future-bounded, propositional metric temporal logic [18]. The software monitors run on an Arm development board that shares the CAN bus present in the ARV.

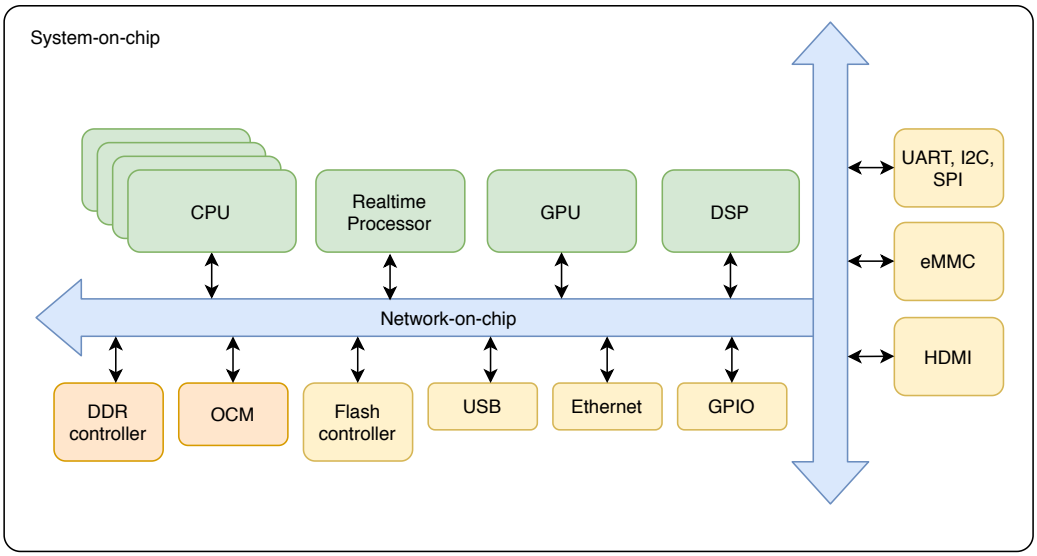


Figure 2.3: Block diagram of an SoC

2.2.3 FPGA

FPGA are semiconductor chips whose hardware components and connections can be customized for different application requirements. The reconfigurability is achieved by placing a large number of configurable logic blocks (CLBs), in a regular pattern, between a grid of configurable interconnects. Figure 2.4 shows the block diagram of a modern Xilinx FPGA. A CLB contains combinational logic to implement combinatorial functions, sequential logic (flip-flops or latches), and programmable routing controls (multiplexers). Modern FPGAs are also equipped with specialized hardware optimized for high-speed arithmetic operations and DSP applications. This hardware block is called a DSP slice. Although a CLB can be configured to function as a memory (RAM and ROM), a specialized component called block RAM (BRAM) is also integrated into the FPGA to efficiently implement larger RAM or ROM blocks. A specific component on the FPGA called the clock management tile (CMT),

or digital clock manager (DCM) generates the clock signal required by all the sequential elements. Input-output blocks (IOBs) implement the logic needed to send and receive data from devices external to the FPGA. Digital signals from various blocks are routed to each other over the interconnect fabric through a programmable switch matrix (PSM). Special routing resources are used to route clock signals.

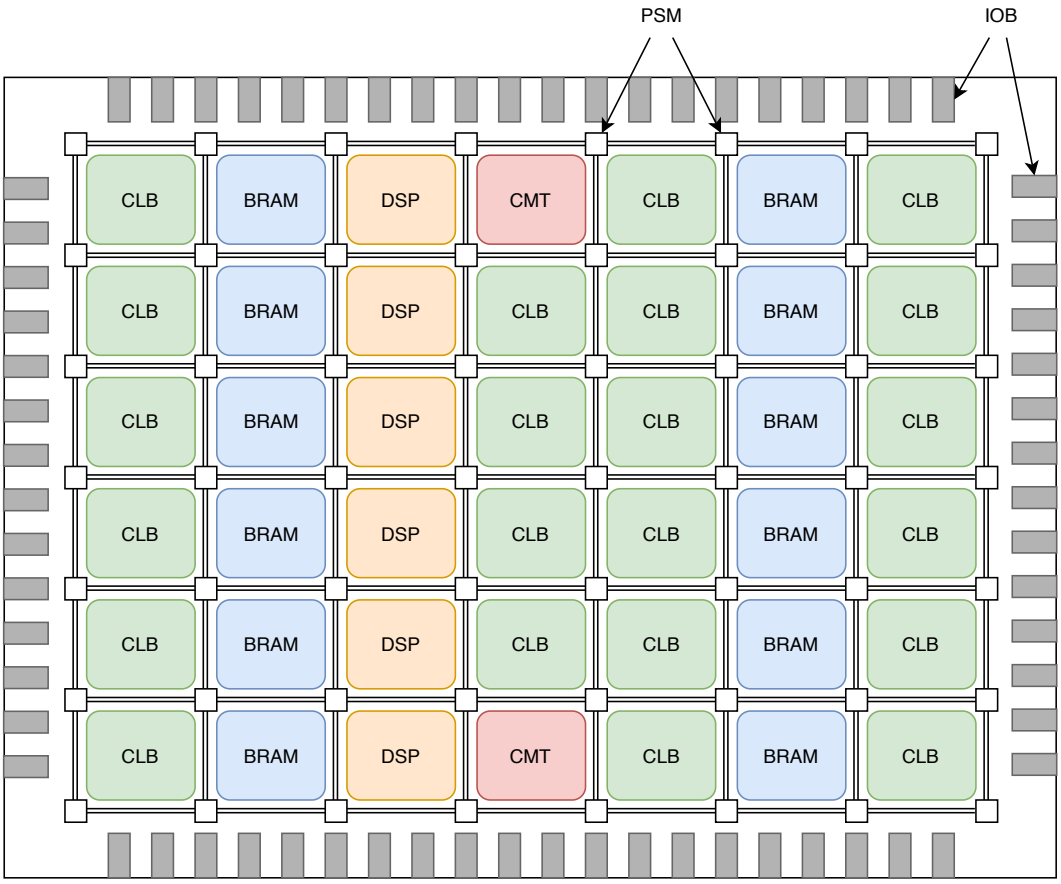


Figure 2.4: Block diagram of an FPGA

FPGA vendors provide tools required to design and implement digital circuits on the FPGA. The hardware design can either be written in an HDL like Verilog or can be synthesized from models or high-level languages. After the design is mapped to hardware resources, the tool produces a bitstream that contains all the information needed to configure the programmable components of the FPGA.

FPGAs are well suited to applications that can benefit from hardware acceleration and when the hardware design evolves quickly. Recently Microsoft and Amazon have invested in servers containing FPGAs to optimize application performance. Microsoft accelerates image searches using neural networks implemented on the FPGA [24]. Amazon provides FPGA resources to customers who train and run machine learning algorithms on AWS [34].

Another use of FPGAs is in applications that require multiple lightweight processors. FPGAs can implement a *soft-core* processor that can execute programs just like an ASIC hard (fixed in silicon) processor does. The soft-core processor can either have its own instruction and data memory or can share the memory with other soft-core processors. FPGAs with abundant resources can implement multiple soft-core processors. Having multiple processors operate on independent tasks provides parallelism and isolation, which is a crucial aspect in designing reliable and secure systems. Dedicated soft-core processors can be created to evaluate software monitors.

FPGAs have previously been used to implement monitors [31], [23], [33]. A methodology for on-line monitoring of past-time metric temporal logic (ptMTL) formulas on FPGA is presented by Reinbacher et al. [31]. A framework called BusMop is proposed for verifying commercial of the shelf (COTS) components [23]. Hardware monitors synthesized from high-level specifications are implemented on an FPGA and get plugged into the PCI peripheral bus. Monitors on the FPGA snoop the PCI bus for data packets that are sent and received by various COTS peripherals sharing the PCI bus. R2U2 is a framework presented for monitoring security properties and diagnosing threats to UAS [33]. The hardware monitors are implemented in an FPGA and use GPS, GCS, sensor data, actuator commands, and the flight software status.

2.2.4 PSoC

A PSoC is a new and advanced category of a system-on-chip. Along with the standard components of an SoC, a PSoC contains an integrated FPGA. Popular examples of a PSoC are the Xilinx Zynq family [45]. Figure 2.5 shows the block diagram of a modern Xilinx Zynq product. In this architecture, the fixed-function circuitry, which is the SoC part of the PSoC, is called the processing system (PS). The FPGA part of the PSoC is called the programmable logic (PL). A PSoC incorporates the advantages of both SoCs and FPGAs. The PS provides large amounts of processing power with multiple choices of peripherals, while the PL provides configurable hardware that can achieve acceleration, parallelism, and isolation. Some important drawbacks of a PSoC are its large die size, which increases cost and power consumption. In most cases, the benefits of a PSoC outweigh its disadvantages and are hence a popular choice in the research and product development community.

In [35], a PSoC implements the monitors synthesized from specifications expressed in past-time linear temporal logic (ptLTL) formulas. Minimal instrumentation code is added to the software running on the microcontroller to generate events that are processed by the monitors implemented in the programmable part of the PSoC.

2.2.5 Hybrid Architectures

In a hybrid architecture, the SoC and FPGA are two separate chips connected to each other on the printed circuit board (PCB). This architecture gives the product designers flexibility to choose the SoC and FPGA independent of each other and depending on the application. The Intel Aero is a COTS drone that has the hybrid architecture. An Intel Atom application processor is paired with a small FPGA. Stamenkovich implemented an assurance system for the Intel Aero drone [36]. Hardware monitors synthesized from LTL formulas are

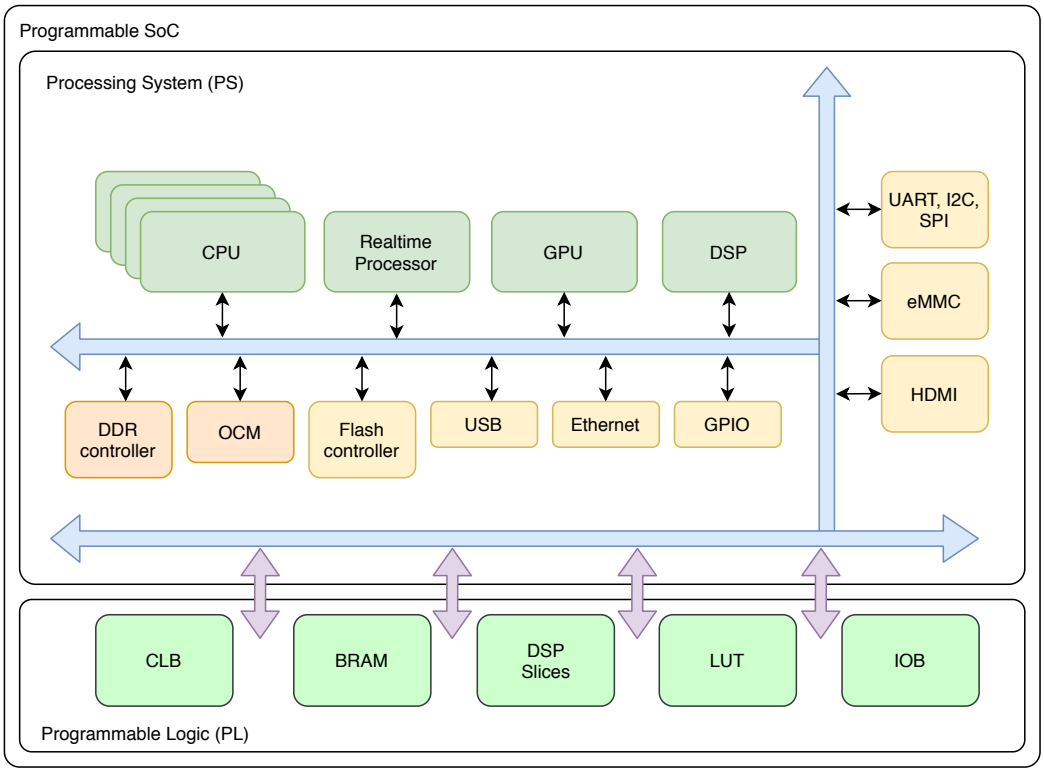


Figure 2.5: Block diagram of a Zynq PSoC

implemented on the resource-constrained FPGA onboard the drone. The monitors enforce a virtual cage to demonstrate the applicability of runtime verification on COTS drones. The architecture of the Aero's compute board is shown in Figure 2.6. When flying the drone in manual mode, the data from the radio control (RC) remote controller is sent to the application processor. The virtual cage monitors use the GPS data to determine if the drone would exit the virtual cage. On detection, a recovery control function (RCF) is invoked. The RCF instructs the drone to land. This work provides a proof-of-concept for runtime verification on drones by implementing hardware monitors on an FPGA.

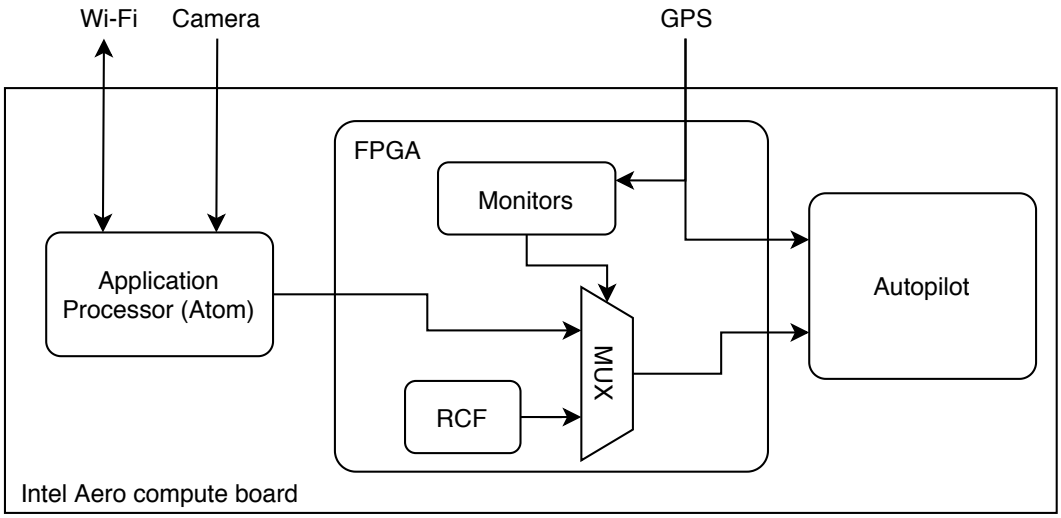


Figure 2.6: The monitors implemented on the Intel Aero drone

2.3 HITL Simulation

HITL simulation is a technique employed in the design and test of real-time embedded control systems. Some examples of real-time embedded control systems are the anti-lock braking system in automobiles and UAV autopilots. The control system is implemented in software and runs on a microcontroller or a real-time processor. While the control system can be tested on the fully assembled end product, this is undesirable as it can incur high costs and significant risk if the control system is defective or the system parameters require further tuning. HITL simulation is an alternative approach where the complexity of the *controlled* system is moved to a test platform, and the control system is tricked into believing that it is interacting with the real system. The mathematical model of the controlled system and all related dynamic systems are used in the HITL simulation test platform. In the case of a quadcopter drone, the controlled system is the quadcopter. The model includes: characteristics of the quadcopter such as its geometry, the aerodynamic properties, and the rotors; sensory systems such as gyroscope, barometer, accelerometer, and GPS; and environmental characteristics like buildings, trees, air turbulence, and terrain. Test platform

developers emulate the real world as accurately as possible by increasing the number of interacting systems modeled. However, this comes with a penalty of requiring increased processing power to run the HITL simulation. The HITL simulation differs from a software-in-the-loop (SITL) simulation wherein the controlled system and the control system are both simulated in the test platform.

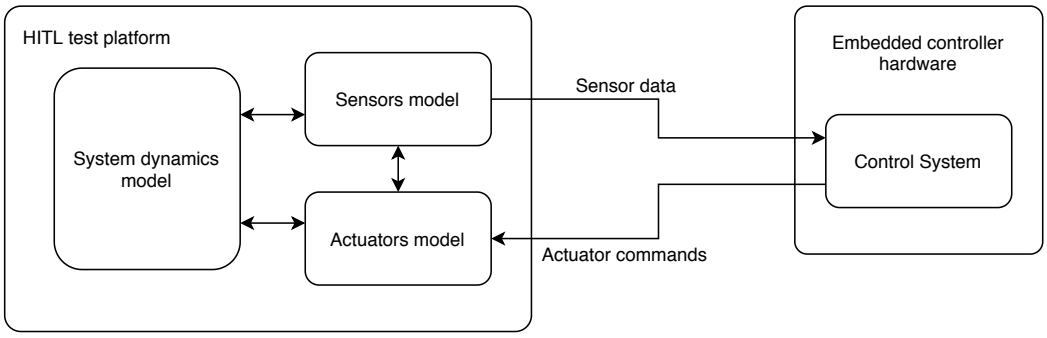


Figure 2.7: Block diagram of a HITL simulation system

During a HITL simulation, the test platform emulates the sensors and provides the data to the embedded controller, which in turn runs the control loop and generates actuator signals. The actuator signals are fed back to the test platform and are used by the system models to emulate the next set of sensor data. This is shown in Figure 2.7. Each iteration of the simulation is called a *step* in the HITL simulation. For the embedded controller to run the control system reliably, the simulator must provide the sensor data at no less than a minimum rate. Thus, HITL simulations with detailed models are run on a modern desktop machine equipped with a powerful graphics card. Communication between the test platform and the embedded controller can be over proprietary protocols or standard protocols like universal asynchronous receiver-transmitter (UART), universal serial bus (USB), or controller area network (CAN). Gazebo is a widely used simulation platform for developing robots and autonomous vehicles [16].

2.4 Open Source Drone Project

Dronecode is a nonprofit open source project that works towards building a comprehensive platform for UAV development [12]. The project includes the flight controller hardware, autopilot software, ground control station, and software toolkit for application-specific customization. A platform like Dronecode is typically chosen when designing a new UAV product. Most of the critical components of the hardware and software can be easily integrated, tested, and customized.

2.4.1 Pixhawk

Pixhawk is an open-hardware project under Dronecode. The main objective of Pixhawk is to provide standardized reference hardware designs for the UAV component manufacturers. Several versions of the flight controller hardware have released as the project evolved. The flight controller typically contains an Arm-based primary processor, an I/O processor, onboard sensors, pulse width modulated outputs for motors, inputs from radio controller, GPS, and multiple standard interfaces like SPI, UART, and I2C to connect external sensors [38]. A micro-USB port is provided for debugging, programming, and hardware-in-the-loop simulation. Pixhawk supports the PX4 and ArduPilot flight stacks. Figure 2.8 is an image of the first generation Pixhawk flight controller.

2.4.2 PX4

PX4 is the autopilot software stack for drones and other unmanned vehicles and has support for airframes like multicopters, fixed-wing, and vertical take-off and landing (VTOL) aircraft [25]. The software repository supports the development and testing of autopilot firmware.

Figure 2.8: Pixhawk with ports for external hardware

PX4 has support for both SITL and HITL simulation and can run on several simulators like Gazebo, AirSim [6] and jMAVSim [27].

2.4.3 MAVLink

MAVLink is a novel messaging protocol that is used across Dronecode projects [21]. The onboard components use MAVLink messages to communicate data in a publisher-subscriber pattern. MAVLink also transfers commands and acknowledgments between the UAV and QGC. In a HITL simulation, the emulated sensor data is packaged as MAVLink messages and sent to the flight controller, and the flight controller sends the actuator commands through MAVLink messages. The publisher-subscriber pattern allows the creation of a channel, and multiple components simultaneously listen to the channel to get the published message. The

CRC checksum field protects MAVLink messages from data corruption arising from high latency and high electrical noise. MAVLink also provides the means to detect packet drops and to authenticate messages. A single header file allows MAVLink to be integrated with a C project. Messages can be packed and unpacked using the provided APIs.

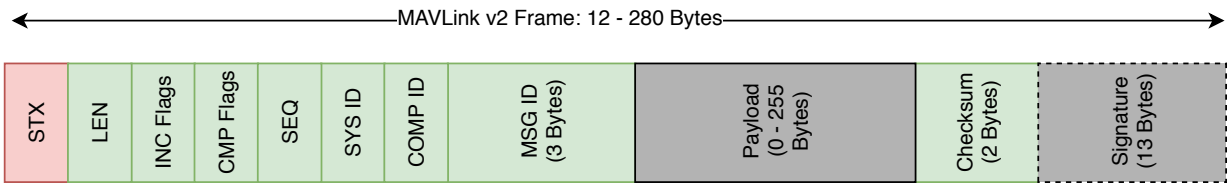


Figure 2.9: MAVLink v2 packet structure

Figure 2.9 shows the structure of a MAVLink v2 packet. The first byte `STX` is the packet start marker and has a fixed value 0xFD. The `LEN` field denotes the length of the payload. The maximum payload in a single MAVLink packet is 255 bytes. The `SEQ` field represents the sequence number of the packet, which is used to detect packet drops. `MSG ID` denotes the type of message and is 3 bytes long. The payload can also be empty for packets like acknowledgment. `Signature` is an optional field that can be used to authenticate MAVLink packets. The fields in `INC Flags` and `CMP Flags` set flags to indicate supported and unsupported features. `SYS ID` is the ID of the sending vehicle, whereas `COMP ID` is the ID of the component generating the MAVLink message.

2.4.4 QGroundControl

QGroundControl (QGC) is a ground control station(GCS) software. It is used for setting up, monitoring, and controlling a vehicle that runs PX4 or ArduPilot [29]. QGC is also used for creating flight plans that are uploaded to the Pixhawk, using either a USB or a wireless connection. Important flight parameters such as the takeoff and landing speed and cruising speed are configured through QGC. Commands such as arming the drone, selecting

the flight modes, and dynamically changing the waypoints during the flight are sent to the vehicle using the MAVLink messaging protocol. On receiving and processing the commands, the vehicle responds with an acknowledgment. The vehicle can also send specific messages that are decoded by QGC. MAVLink Inspector is a tool available within QGC that monitors, decodes, and displays the messages received by QGC. Figure 2.10 is a screenshot of the QGroundControl software. The floating MAVLink Inspector window shows the MAVLink packets that QGC has received.

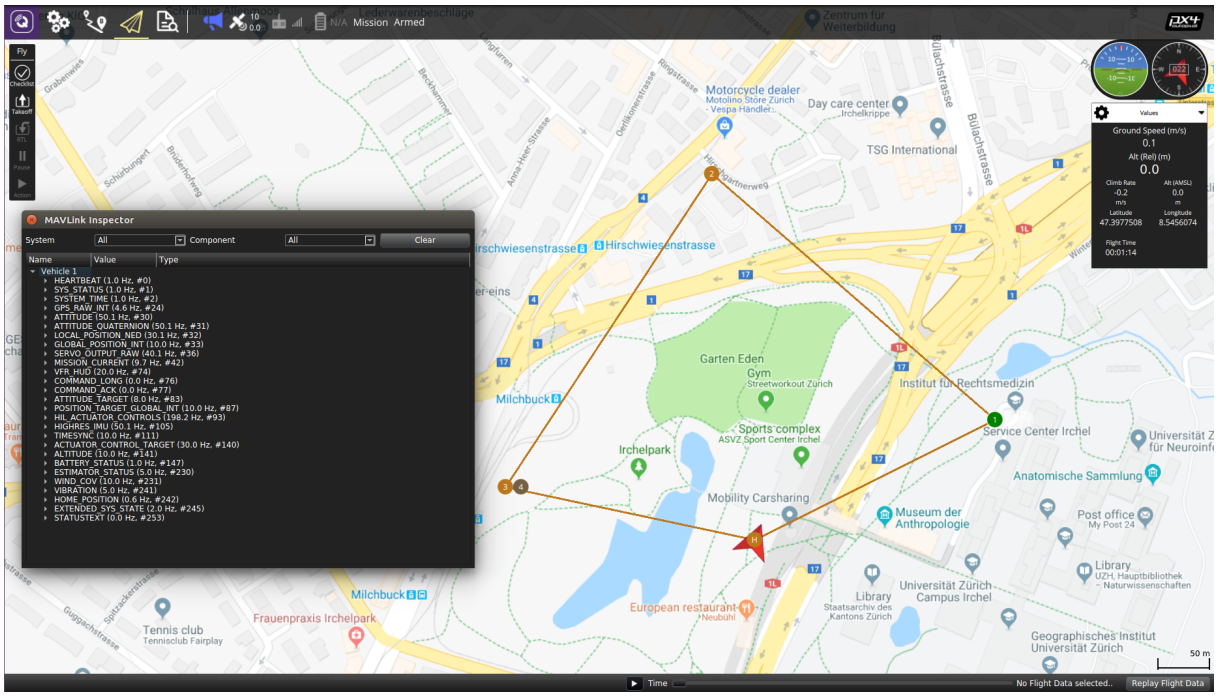


Figure 2.10: Screenshot of QGroundControl with MAVLink Inspector window

Chapter 3

High-level Design

The primary goal of this research is to create a platform optimized for the efficient runtime verfication of UAVs. This chapter discusses the considerations for choosing the elements used in the platform. The high-level design of the platform, its external interfaces, and the simulation process are also presented.

The HITL simulation technique is used as it offers the following advantages:

1. Realistic simulation: Using a high-fidelity model in HITL simulation is a close approximation to real-world scenarios. The actual flight controller runs the autopilot software in a simulated environment. Creating the high-fidelity model and characterizing the model is outside the scope of this work.

2. Low cost: Since HITL simulation does not require the actual UAV, the cost of procuring, maintaining, and repairing the UAV, in case of a crash, is reduced to zero.

3. Shorter turnaround time: When a newly implemented design is ready for validation, HITL simulation can be immediately run. Appropriate weather conditions and legally permitted air spaces would otherwise be required to fly a UAV and to validate any change made to the design [1].

4. Ease of instrumentation: Implementing runtime verification with hardware monitors requires instrumenting and observing electrical signals in the UAV. Signals from the

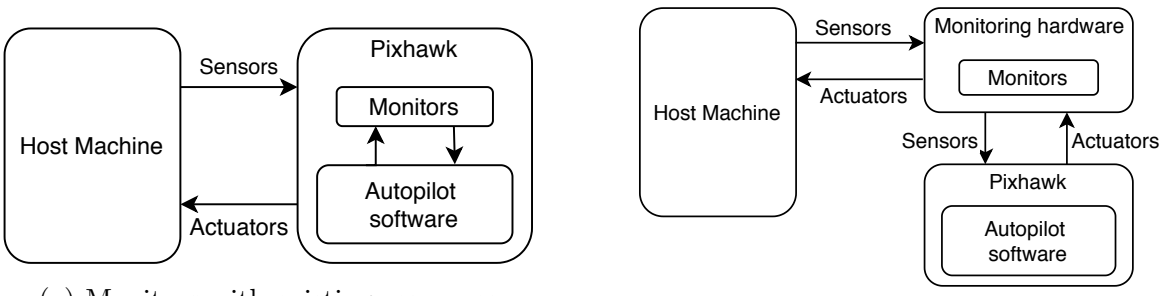
sensors embedded on the flight controller are not easily accessible without reengineering the flight controller. Similar problems may arise with GPS and RC remote controller signals. In an HITL simulation environment, a single physical interface exists between the simulator and the flight controller. Instrumenting this interface provides access to all the required signals.

3.1 Architectural Choices

Two fundamentally distinct architectural choices exist to implement runtime verification with HITL simulation for UAVs. As represented in Figure 3.1, the choices are:

1. To use the existing resources on the flight controller to implement monitors and perform corrective actions.

2. To add additional resources to implement the monitors and perform corrective actions.



(a) Monitors with existing resources

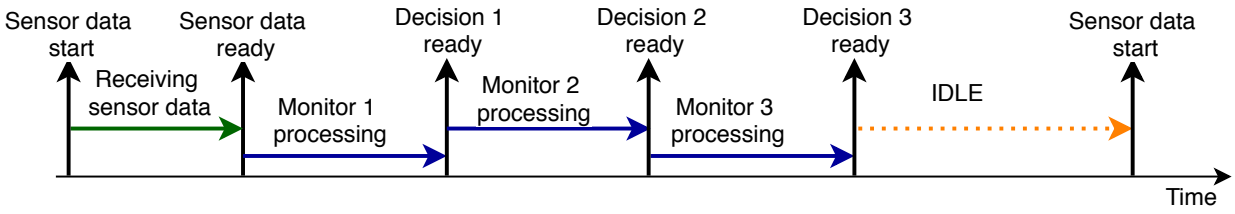
(b) Monitors with additional resources

Figure 3.1: Monitors implemented with and without additional resources

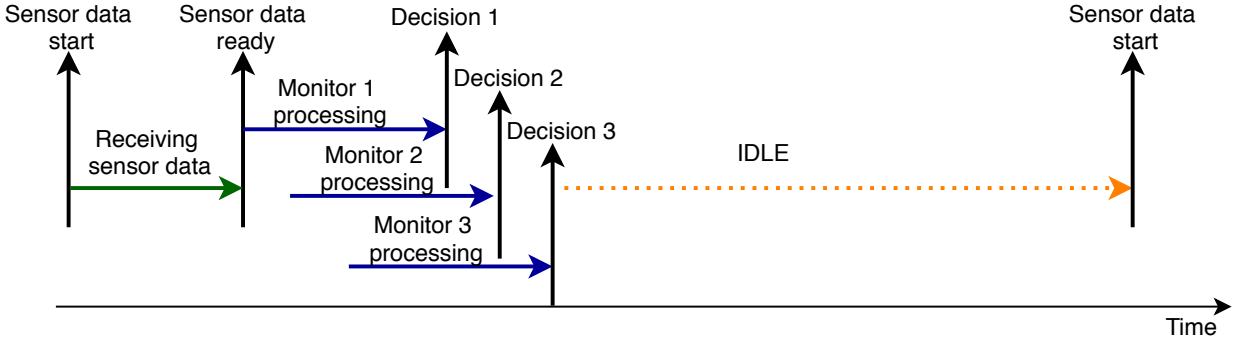
While using the existing resources offers advantages in terms of cost, area, and power, the following factors need investigation while designing a system to monitor and perform corrective actions:

1. *Hardware and software implementation of the monitoring system*: Monitors implemented in software run on a processor and thus execute sequentially. Monitors implemented as hardware can be designed to execute in parallel. Sensor inputs to the monitors arrive periodically. The idle time available between two sensor inputs constrain the maximum number of implementable monitors. Figure 3.2a shows the timeline for sequential execution of 3 monitors, and Figure 3.2b shows the parallel execution of the same. The parallel execution scheme has a larger idle time and hence can accommodate the implementation of additional monitors in the future.

   The flight controller can only implement software monitors. Implementing hardware monitors would require additional hardware resources like an FPGA.



(a) Serial execution of monitors

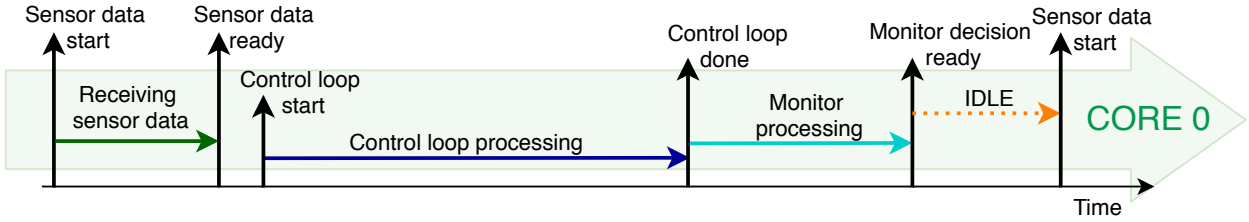


(b) Parallel execution of monitors

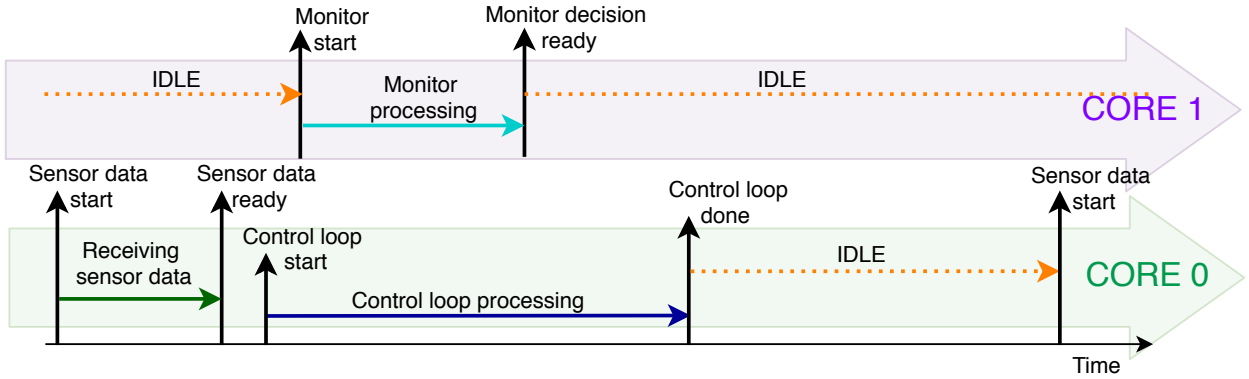
Figure 3.2: Timeline of sequential and parallel execution of monitors

2. *Shared CPU cycles*: Most flight controllers, like the Pixhawk, use a single-core CPU to run the autopilot software [38]. The control algorithms and other miscellaneous processes of the autopilot software run periodically on the CPU. When a software

monitor is added to run on the same CPU, the CPU cycles (processing time) have to be shared between the autopilot processes and the monitor process, as shown in Figure 3.3a. Sharing the CPU cycles could potentially lead to resource starvation and cause the autopilot system to fail. If the flight controller has a multi-core CPU, the autopilot processes can run on one core, and the monitoring system can run on another core, as shown in Figure 3.3b. Multi-core CPUs are generally not present in low-cost, low-power, and open source flight controllers.



(a) Single-core system: CPU cycles shared between autopilot and monitor processes



(b) Multi-core system: Core 0 running autopilot processess and core 1 running monitor process

Figure 3.3: Timeline of a single-core and a multi-core system running autopilot and monitors

3. *Security*: For safety-critical applications like UAVs, a robust runtime monitoring system should increase resilience to various security threats. A malicious agent can gain access to the UAV during a flight and can either disable the monitors or obstruct the corrective action. Implementing the monitoring system in discrete, isolated hardware reduces its exposure to the outside world and thus the chances of compromise.

4. *Ease of implementation*: For the software monitors to run on the flight controller, the monitors should be integrated with the autopilot software stack. Any changes made to the monitor would require recompiling the whole software stack, which is time-consuming. PX4 is in continuous development, and software updates are released frequently [26]. Thus the monitors would also require frequent integration with the updated software stack. The hardware or software monitors that run on isolated hardware can be developed independently of the autopilot software.

5. *Isolation from component failures*: Implementing the monitoring system in isolated hardware reduces the chance of total system failure that could occur due to software or hardware malfunctions of a component in the system.

6. *Application-specific hardware*: Flight controllers are designed only to run the autopilot software and navigate the UAV. Monitors, on the other hand, are application-specific and may require complex rules that cannot run on the flight controller's CPU. Adding separate hardware to run the monitoring system creates processing islands that are dedicated to performing specific tasks.

Due to the above factors, the monitors are implemented in external hardware, as shown in Figure 3.4. A Xilinx Zynq UltraScale+ PSoC is integrated with the HITL simulation setup consisting of the host machine and the Pixhawk [50]. This PSoC has a rich set of fixed-function hardware in the PS and abundant programmable resources in the PL. The soft processors and the monitors are implemented in the PL. The Xilinx ZCU104 is the evaluation kit that hosts the PSoC and other peripheral connectors used in this work [46]. Chapter 4 presents a detailed description of the hardware implementation. In addition to the previously mentioned benefits of using external hardware to implement the monitors, the Xilinx PSoC-based platform provides the following advantages:

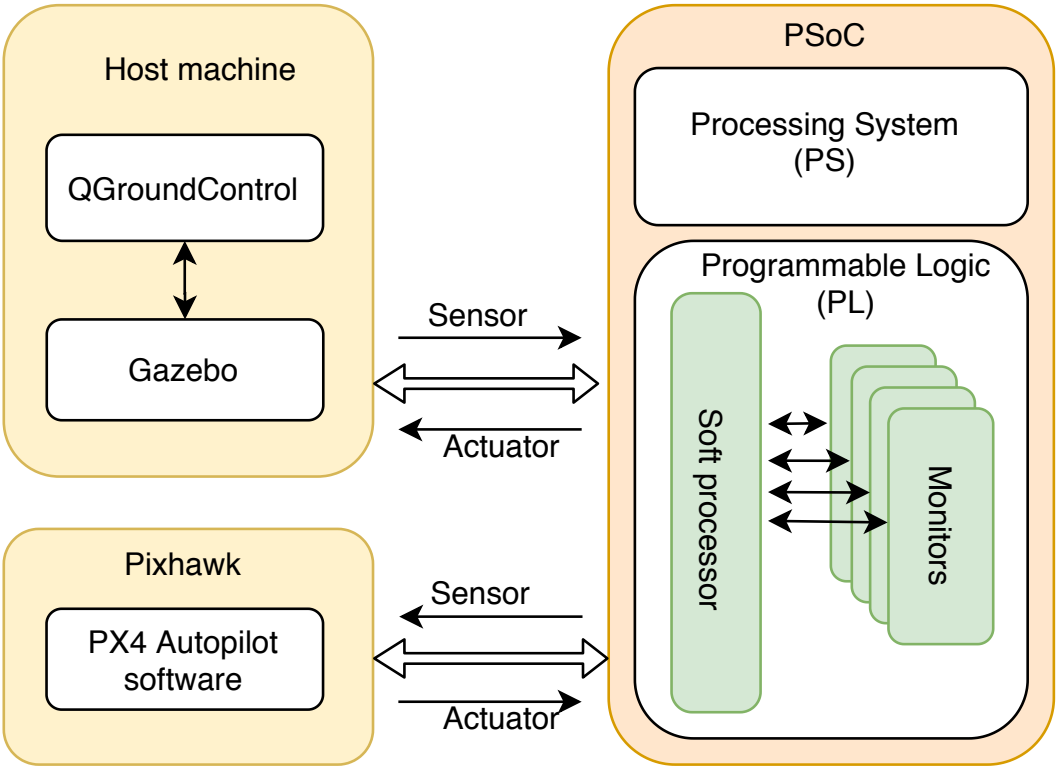


Figure 3.4: High level architecture: Xilinx Zynq UltraScale+ PSoC integrated with HITL simulation setup

1. Modularity: The Pixhawk flight controller, PSoC, and the host machine are all modular, and each of them can be upgraded independently. The only dependency is between the version of the PX4 autopilot software running on the Pixhawk and the application running on the simulator on the host machine.

2. A Xilinx FPGA can implement a soft processor. During development, software monitors can be implemented first and run on the soft processor and later turned into hardware monitors. Developing and testing software monitors is less time-consuming than going through the process of high-level synthesis (HLS) to generate a hardware monitor.

3. The application processors, GPU, and DSP, which are the fixed-function hardware

available on the PSoC, can be used for high-level applications like image recognition, machine learning, obstacle avoidance. These applications can generate outputs that control the flight of the UAV. Designing and validating a system where the application processor on the PSoC controls the UAV is outside the scope of this thesis. A brief analysis of the architecture for such a design is presented in Chapter 6.1.

3.1.1 Implementation Specifications and Requirements

The implementation of the monitoring and recovery system has to meet the following specifications and requirements:

1. *Stability*: The HITL simulation with the implemented monitoring system needs to be stable throughout the testing duration of the flight. When running a HITL simulation without the noise parameters, the simulation is stable if the UAV follows the planned mission closely, maintains a steady trajectory with a leveled flight, and does not crash the UAV. The simulation results should also be consistent and repeatable.

2. *Minimal delay*: The instrumentation logic, monitoring system, and recovery system inject a processing delay. Real-time control systems like the autopilot software are sensitive to delays introduced in the control loop. The total delay introduced should be small enough such that the control loop operates without failure.

3. *Scalability of monitors*: The implementation must be capable of supporting the addition of numerous monitors, either in hardware or software.

4. *FPGA resource usage*: The implementation of all the systems other than the monitors should optimize FPGA resource usage and power. The operating frequency of the soft processors should be optimized for power, performance, and area (PPA).

5. *Programmable RCF*: The implementation should support invoking different RCFs depending on the use case.

3.2 Simulation Process

The high-level steps for running the HITL simulation with the monitoring system are shown in Figure 3.5. Each simulation is the execution of a flight mission, in which the UAV starts at the home coordinates, flys through specific coordinates called waypoints, and lands either at the home coordinates or different coordinates.

1. The first step in the simulation process is to create a flight plan. The flight plan contains crucial information like the GPS coordinates and altitude of all the waypoints in the mission and other mission-related parameters. The autopilot software needs this information to navigate the UAV.

2. The steps highlighted in gray are not mandatory for a HITL simulation. NASA is developing a system called UAS Traffic Management (UTM) to enable UAS operations for low-altitude, civilian use [22]. For a UTM-compliant autonomous flight, the flight plan must be filed with UTM to obtain approval for the flight mission. The approval process may require modifications to the flight plan.

3. The approved flight plan gets uploaded to the Pixhawk through QGC.

4. The monitoring system requires the flight plan, along with parameters that configure the monitors. Hence preprocessing is done to a copy of the approved flight plan and uploaded to the monitoring system.

5. The monitoring system is instructed to start the monitors.

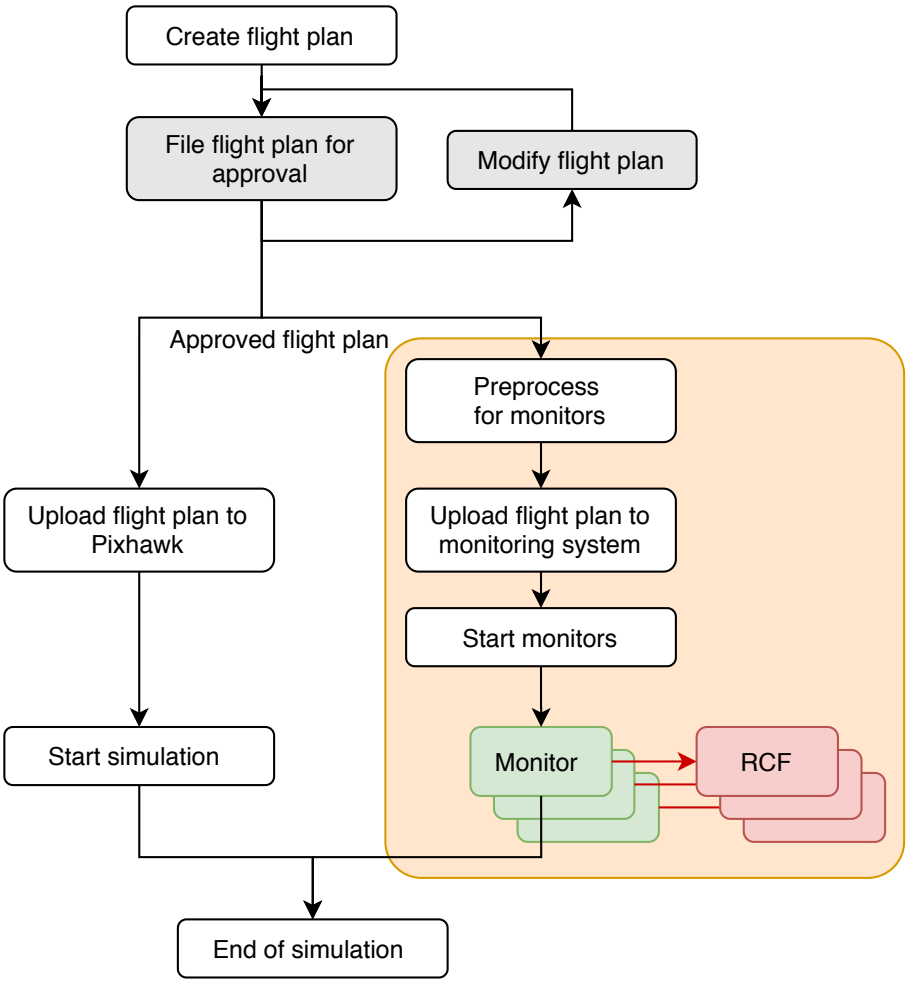


Figure 3.5: High-level steps for the HITL simulation

6. The HITL simulation is then run on the host machine. QGC sends the commands to arm the UAV and to start the mission.

7. When the mission is in progress, the monitoring system continues to run the monitors and, if necessary, invokes the RCF. The blocks highlighted in green and red are run automatically and do not need manual invocation.

8. After the mission completes, the simulation is terminated.

The blocks highlighted in orange are the additional steps required by the monitoring and

recovery system.

3.3 External Interfaces

The monitoring and recovery system implemented on the Zynq UltraScale+ PSoC requires two external interfaces. One interface is for communicating the HITL simulation data, and the other is for configuring the monitors. Figure 3.6 shows the high-level block diagram of the HITL simulation platform with the two external interfaces. The following sections describe the details of each of the interfaces.

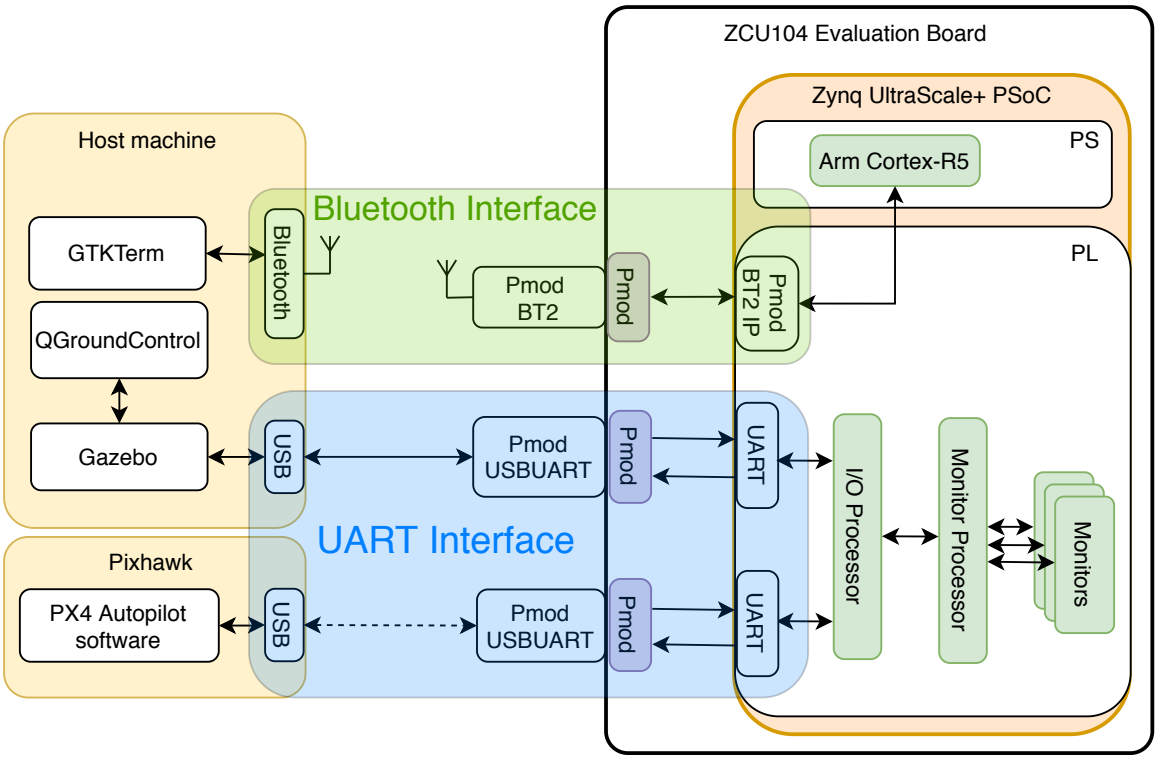


Figure 3.6: Block diagram highlighting the two external interfaces

3.3.1 UART Interface

The basic HITL simulation setup of PX4 with a Pixhawk uses serial communication to transfer data between the host machine and the Pixhawk. Serial communication takes place over a USB interface, where the host machine acts as the USB host and the Pixhawk as the USB device. On connecting the Pixhawk to the host machine, the Pixhawk gets detected as a serial device. On a Linux machine, the Pixhawk handle gets created as `/dev/ttyACM0`. The simulator, like Gazebo or AirSim, uses this handle to write and read data serially to/from the Pixhawk.

The monitoring system has to intercept the serial communication link between the host machine and the Pixhawk to obtain the data it requires. A UART, which is also a serial communication device, is used to connect the monitoring system to the host machine and the Pixhawk. The UART communicates data using two separate physical channels, the transmit data channel (TxD) and the receive data channel (RxD). Older UART implementations use two additional channels for dataflow control: the request-to-send (RTS) channel and the clear-to-send (CTS) channel. Each of the channels is a physical pin in the UART device, and electrical wires are used to connect two UART devices.

Modern computers do not have a dedicated UART interface. A USB-to-UART converter, like the Pmod USBUART manufactured by Digilent, can be used to communicate with a UART device [9]. Figure 3.7 shows the USBUART. One end of the USBUART connects to a USB port on the host machine and the other end to a Pmod connector on the ZCU104 evaluation board. UART signals are generated on the Pmod end of this device. The Pmod interface (Peripheral module interface) is an open standard defined by Digilent and is available on many development boards and evaluation boards [11]. UART hardware implemented on the Zynq PSoC's PL converts the UART signals to data bytes and vice-versa. An input/output

(I/O) soft processor can read and write data bytes from/to the implemented UART hardware.

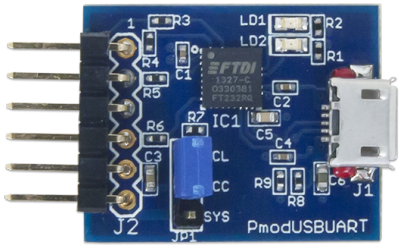


Figure 3.7: Digilent USBUART Pmod

In Figure 3.6, the connections highlighted in blue show the two UART connections: one connects the PSoC to the host machine and the other to the Pixhawk. The connection between the PSoC and Pixhawk, shown with a dashed line, requires extra hardware, and a detailed explanation is provided in Chapter 4.4.2.

3.3.2 Bluetooth Interface

Relevant information needed to configure the monitoring system is stored in the flight plan. For example, a monitor implemented to keep track of the UAV's position needs to know the waypoints through which the UAV should fly. The GPS coordinates and the altitude of the waypoints, the total number of waypoints, and the flight speed are some of the parameters available in the flight plan upon creation. A preprocessing step adds additional static information, such as the takeoff and landing speed, to the flight plan. The processed flight plan is transferred to the monitoring system through a Bluetooth interface, along with the command to start all the monitors. Since mobile devices like iPads support creating the

flight plan, a Bluetooth interface, which is available on most mobile and laptop devices, is used to transfer the flight plan and to start the monitors.

The Xilinx ZCU104 evaluation board does not come with a built-in Bluetooth interface. Figure 3.8 is an image of Pmod BT2, a Pmod-compatible peripheral module that can be connected to the ZCU104 board to add Bluetooth connectivity [10]. After power-up, the Pmod BT2 gets detected as a Bluetooth device, and the host computer can pair with it. An open source application, like GTKTerm [32] on Linux or PuTTY [37] on Windows, is used to send the flight plan and other commands to the monitoring system.

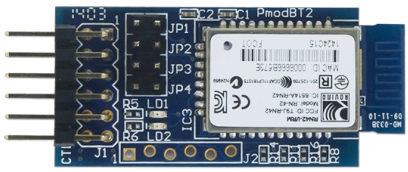


Figure 3.8: Digilent BT2 Pmod

In Figure 3.6, the Bluetooth interface is highlighted in green. Digilent provides the Pmod BT2 IP (intellectual property), which is the configurable hardware core required to send and receive data over the Bluetooth interface. This IP is implemented on the PSoC's PL. The Arm Cortex-R5, present in the PSoC's PS, writes and reads data bytes from this IP, which gets communicated through the Bluetooth interface.

Chapter 4

Implementation

This chapter covers the implementation of the runtime verification platform. An overview of the PSoC evaluation board used in this research is presented, followed by a detailed description of the hardware design implemented on the PSoC. Details of a novel inter-process communication protocol are presented. The procedure for setting up the two simulators for runtime verification is also explained.

4.1 ZCU104 Evaluation Board

The UAV runtime verification platform developed in this thesis uses a Xilinx Zynq UltraScale+ PSoC on a Xilinx ZCU104 evaluation board. The evaluation board contains the Xilinx Zynq UltraScale+ PSoC chip along with various other components that make development on the PSoC possible [47]. Figure 4.1 is the image of the evaluation board. The board contains the circuitry that generates a regulated power supply for the PSoC. The PSoC can be programmed, and applications can be run in debug mode using a JTAG interface. The evaluation board contains an FTDI FT4232 chip, which is a USB to quad-channel (Channel A to D) serial interface IC. The FT4232 configuration is such that Channel A gets used as a JTAG interface, and Channels B, C, and D get used as UART interfaces. A UART interface from Channel B and C connect to two UARTs present in the PS, and the UART interface from Channel D connects to I/O pins of the PL. Connections are made between

the pins of FTDI FT4232 chip and the pins of the PSoC through the PCB and thus are fixed connections.

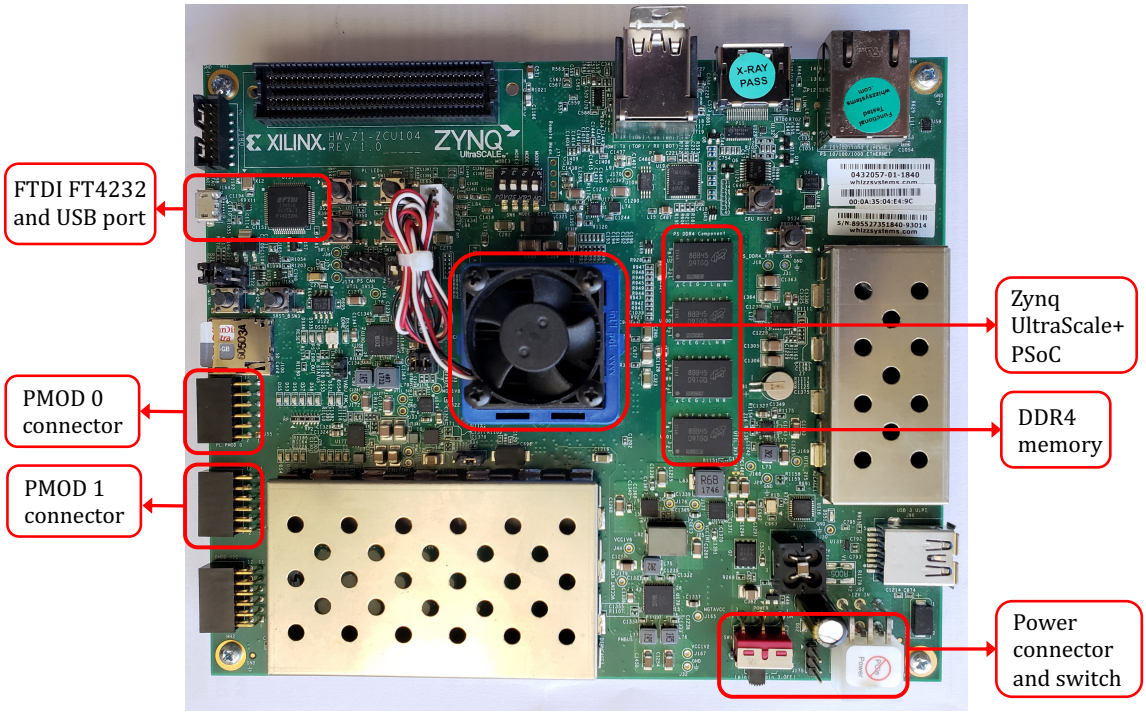


Figure 4.1: Xilinx ZCU104 Evaluation Board

The evaluation board contains a Micron MT25QU512 chip, which is a 512 Mb Quad SPI NOR flash memory. It is mounted on the underside of the board and hence is not visible in the image. Since flash memory is a non-volatile memory, the evaluation board can be configured to use a boot image stored in the flash memory for booting the PSoC. In this thesis, the non-volatile flash memory stores the flight plan used during the HITL simulation. A 2GB DDR4 memory is available to the processing cores in the PS and PL. The application running on the real-time Cortex-R5 processor uses the DDR4 memory. There are two general-purpose I/O (GPIO) Pmod connectors, PMOD0 and PMOD1, and one I2C Pmod connector available on the evaluation board. The BT2 and USBUART Pmods connect to the GPIO Pmod connector.

4.2 Hardware Design

Figure 4.2 is the detailed block diagram of the verification platform. It shows the blocks implemented in the PL, the blocks utilized from the PS, and their interactions with the components present on the evaluation board. The modules used from the PS are the Arm Cortex-R5 real-time processor, Quad SPI (QSPI) controller, DDR memory controller, and the AXI interconnect that connects all the components [51]. The DDR4 memory, NOR flash memory, and the Pmod connectors on the evaluation board also get utilized. The FT4232 USB to serial chip is used for programming the device and for displaying debug messages from the runtime verification platform. A UART controller in the PS, connected to the FT4232 chip, is used during debugging and is not shown in the block diagram. The configurable hardware cores to access the BT2 and USBUART Pmods, soft-core processors to process sensor data and run monitors, and hardware blocks to interconnect the components are implemented in the PL.

The Xilinx Vivado tool suite is used for creating the hardware design [44]. Vivado supports generating configurable hardware for Xilinx PSoCs using a graphical interface [49]. IPs are inserted, configured, and connected graphically to form a block design that gets synthesized and implemented on the PSoC.

All components of the runtime verification platform can be organized into three processing domains, each containing a processor core that runs a dedicated task and logic blocks that connect the three domains. The processing domains are highlighted with different colors in Figure 4.2. The following sections describe the details of the processing domains.

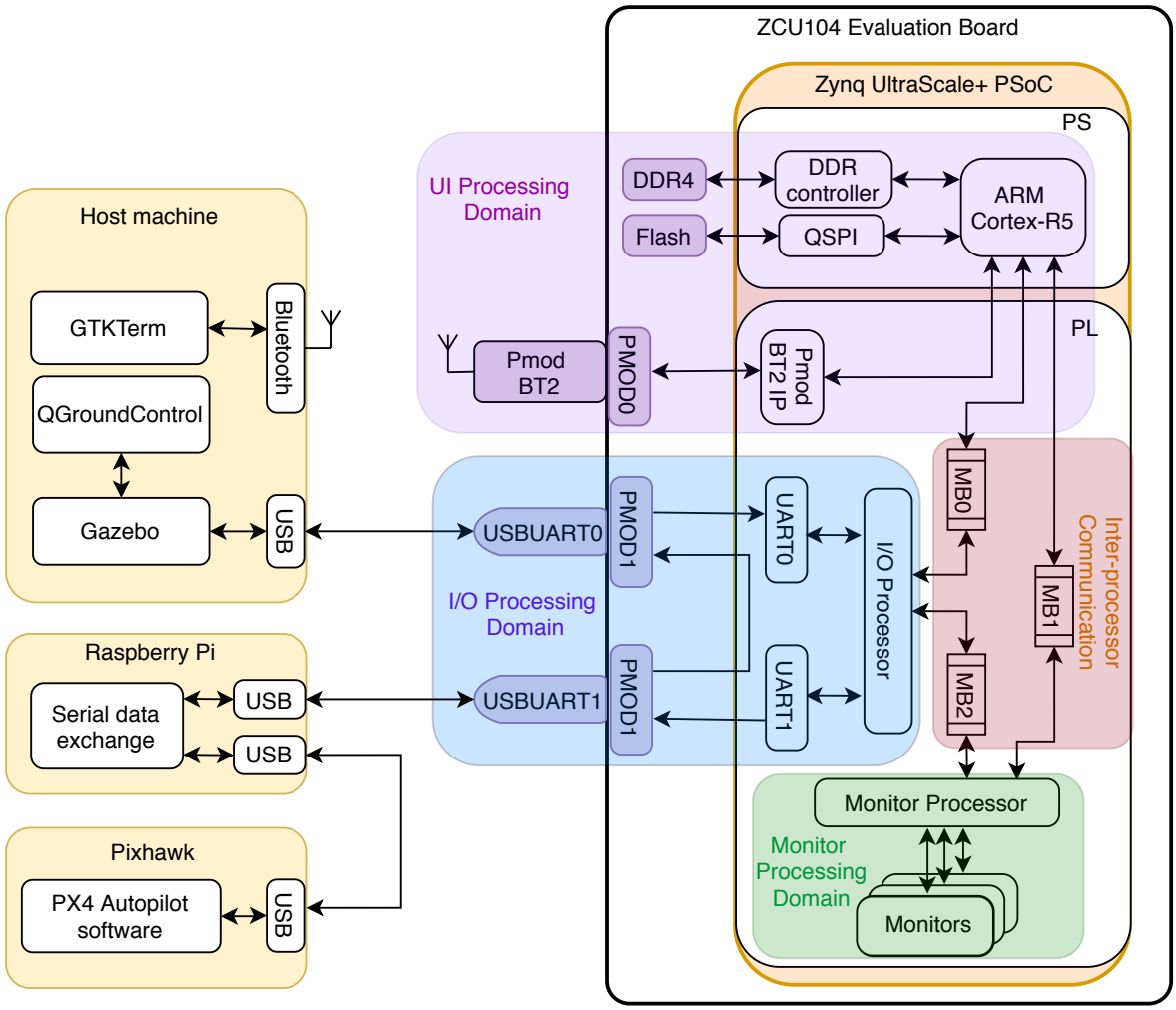


Figure 4.2: Detailed block diagram of the verification platform

4.3 User Interface Processing Domain

As described in Chapter 3.2, the monitoring system requires the flight plan for its monitors to function. A simple user interface (UI) application is developed to run on the verification platform through which the user can upload a flight plan and start the monitors. The user interface processing domain includes all the modules that enable the UI application.

4.3.1 UI Application

When the verification platform is powered up, any device capable of transferring the flight plan serially over Bluetooth can pair with the platform. The user interacts with the platform using a serial terminal. On running the UI application, the platform checks its internal memory and displays if a previously uploaded flight plan is available. The user can choose to reuse the flight plan or upload a new flight plan.

4.3.2 Non-volatile Memory

To support flight plan reuse, the flight plan needs to be stored in non-volatile memory. There are two non-volatile memories available on the evaluation board: the secure digital I/O (SDIO) memory card and the NOR flash memory. The Arm Cortex-A53 quad-core application processor is capable of running a Linux operating system, for which the boot image is stored on the SD card, and its file system uses the SD card. Hence, the runtime verification platform uses the other non-volatile memory, the NOR flash memory, for storing the flight plan.

The NOR flash chip is connected to a set of multiplexed I/O (MIO) pins, MIO0 to MIO5, of the PSoC through the PCB. The PS contains two QSPI controllers, one of which connects to pins MIO0 to MIO5 and hence to the flash chip. The QSPI controller is connected to the AXI interconnect in the PS that connects the application processor, the real-time processor, DDR controller, and all other I/O peripherals of the PS. All processor cores having access to this AXI interconnect can, therefore, access the flash memory.

Circuits implemented in the PL can only access a subset of the MIO pins through the extended MIO (EMIO) interface. This subset does not include the pins MIO0 to MIO5. Hence a QSPI controller implemented in the PL cannot use these MIO pins and, therefore,

cannot access the flash memory. A processor core from the PS has to be used for accessing the flash memory to read and write the flight plan.

4.3.3 Bluetooth Hardware

The BT2 Pmod shown in Figure 3.8 provides the Bluetooth interface to the runtime verification platform and is attached to the PMOD0 connector on the evaluation board. The complete Bluetooth protocol is implemented within the Pmod, and the Pmod provides a simple UART interface to send and receive data. The Digilent IP PmodBT2 v1.0 is added to the block design in the Xilinx Vivado IP integrator to use the BT2 Pmod. This IP implements the required UART and provides an AXI slave interface through which an AXI master (processor core) can configure the Pmod and transfer data. Connections are made from this IP to the PMOD0 connector in the block design, as shown in Figure 4.3. Figure 4.4 is the image showing the PmodBT2 connected to the PMOD0 connector on the evaluation board.

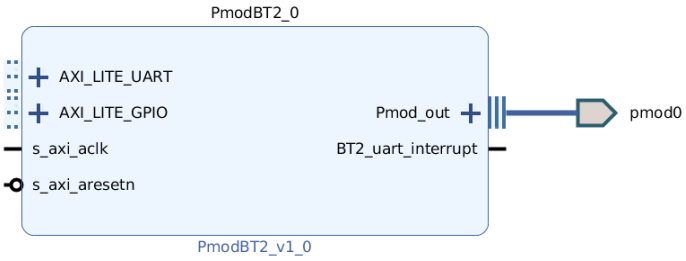


Figure 4.3: Block design connecting PmodBT2 IP to PMOD0 connector

Xilinx does not define the interface definition of Pmod connectors for any of its development boards. Without the Pmod connector definitions, each pin of the IP must be individually routed to the PSoC's pin that corresponds to the Pmod connector, and constraints must be written for all these pins. Pmod connector definitions are added to the *board.xml* and

Figure 4.4: Image showing the PmodBT2 connected to PMOD0 connector

part0_pins.xml files of the ZCU104 board to use the Pmod connectors as a port in the block design.

4.3.4 Real-time Processor

To make the runtime verification platform secure, we would ideally like to have all the hardware modules isolated and implemented in the PL. However, for reasons mentioned in Chapter 4.3.2, a processing core from the PS is needed to read and write the flight plan to flash memory. One of the Arm Cortex-R5 processor cores from the PS is used to run the UI application. The R5 processor is connected to the AXI interconnect in the PS as an AXI master and can, therefore, access the flash memory through the QSPI controller. Dedicated IPs are not required to be added in the block design to access the flash memory. The software code running on the R5 processor uses the DDR4 for instruction and data memory. For the R5 processor to receive the flight plan from the Bluetooth hardware, the PmodBT2 IP's AXI slave should be connected to the AXI interconnect in the PS. Figure 4.5 is the complete block design showing connections between the PS and the PmodBT2 IP.

With this hardware design, the R5 processor can read the flight plan through Bluetooth and write/read the flash memory. Figure 4.6 shows the address space of the PmodBT2 IP.

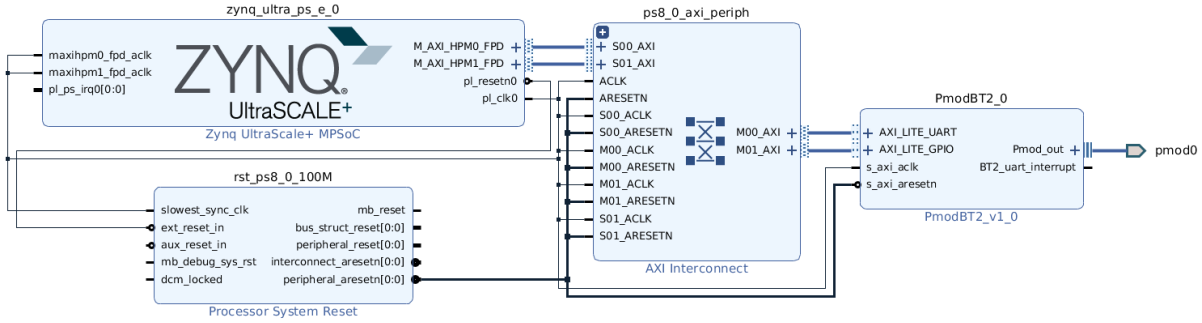


Figure 4.5: Block design connecting PS to PmodBT2 IP

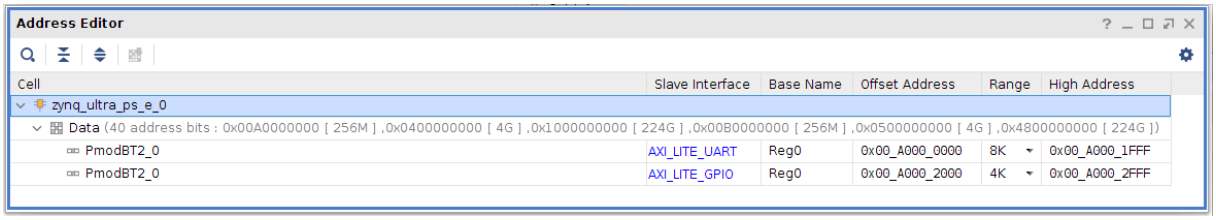


Figure 4.6: Address space of the PmodBT2 IP

The R5 processor parses the flight plan and extracts useful parameters to configure the monitors. A client-server model is used for communicating the parameters, where the R5 processor functions as a server and the monitor processor as a client. The client requests specific parameters, and the server responds with the requested data.

4.4 I/O Processing Domain

In this processing domain, the data communicated between the simulator and the flight controller during the HITL simulation is intercepted, processed, and passed on to the monitor processing domain. Figure 4.7 is the block diagram of the I/O processing domain extracted

from the complete block diagram.

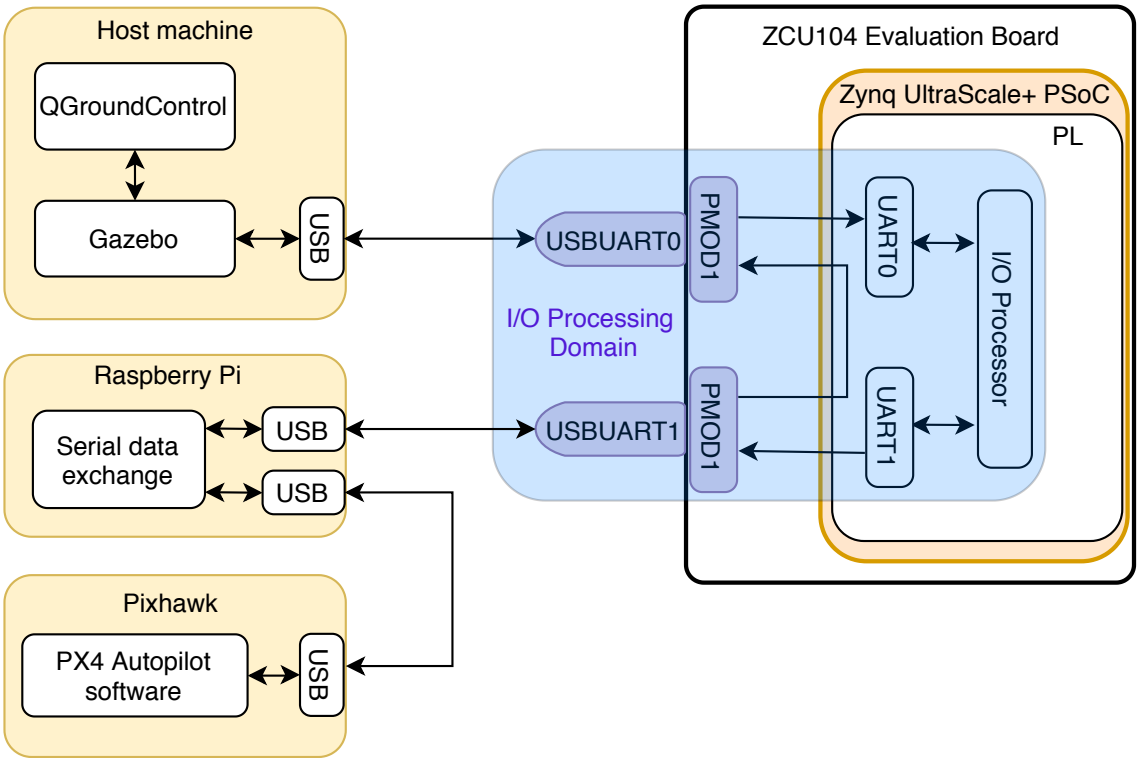


Figure 4.7: Block diagram of the I/O processing domain

4.4.1 UART Hardware

The simulator creates a communication link with the USBUART Pmod connected to the host machine and communicates the simulation data with it. The USBUART Pmod translates the data between the USB interface and the UART interface. The USBUART Pmod has a 6-pin interface that can connect to the upper or lower half of the Pmod connector on the evaluation board. The Xilinx UARTLite v2.0 IP implements a configurable UART hardware in the PL [41]. The UARTLite IP creates an AXI slave interface, and the UART signals `tx` and `rx`. Data is written and read through the AXI interface in bytes, and the UART IP converts the bytes of data into serial data and vice-versa. For routing the `tx` and `rx` signals

to the I/O pins of the PSoC wired to the Pmod connectors, these signals are specified as ports in the block design. Assigning the ports to the physical pins of the PSoC is done through the constraints file.

The verification platform uses two hardware UARTs (UART0 and UART1) to communicate with the two USBUART Pmods (USBUART0 and USBUART1). The two USBUART Pmods connect to the upper half and lower half of the Pmod connector, PMOD1, on the evaluation board. As shown in Figure 4.7, the Pmod USBUART0 communicates with the host machine and is accessible in the I/O processing domain through UART0. The Pmod USBUART1 communicates with the flight controller and is accessible through UART1.

The sensor data is unidirectional and flows from the simulator to the flight controller. The present set of monitors planned for implementation only requires the sensor data and not the actuator data. Hence the actuator data does not need decoding in the I/O processing domain. The incoming UART signal from the flight controller's Pmod, USBUART1, does not connect to UART1. Instead, it is bypassed to USBUART0 through the PL, as depicted in the block diagram in Figure 4.7. Figure 4.8 is the block design showing the two UARTLite IPs UART0 and UART1 and their `tx` and `rx` signals connected to ports. The signal bypassed from USBUART1 to USBUART0 is also shown.

Figure 4.9 show the constraints corresponding to the connections shown in the previous block design. Lines 2 through 11 assign the port names used in the block design to physical pins of the PSoC. Lines 14 through 19 set the I/O standards of the pins to 3.3V LVCMOS, and lines 20 through 23 set the slew rate of the output pins to `FAST`.

The UARTLite IP allows configuring a fixed baud rate before implementing the hardware on PL, whereas the UART 16550 IP supports a software programmable baud rate. Since the resource utilization for UART 16550 IP is higher than UARTLite IP, and the verifica-

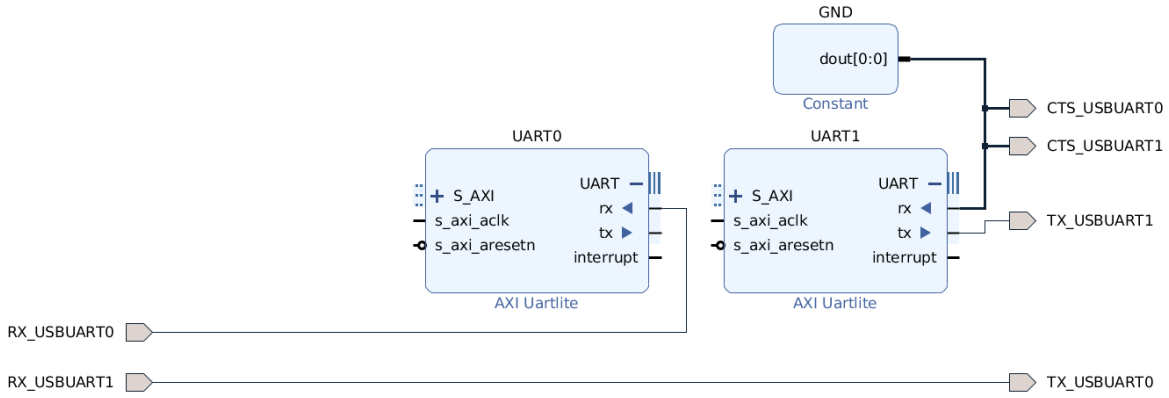


Figure 4.8: Block design showing the UARTs and bypassed signal

```
1  #PMOD1.bottom connected to USBUART0
2  set_property PACKAGE_PIN M8  [get_ports RX_USBUART0]
3  set_property PACKAGE_PIN M10 [get_ports TX_USBUART0]
4
5  #PMOD1.top connected to USBUART1
6  set_property PACKAGE_PIN K8 [get_ports RX_USBUART1]
7  set_property PACKAGE_PIN K9 [get_ports TX_USBUART0]
8
9  #Connections for CTS pins
10 set_property PACKAGE_PIN M9 [get_ports CTS_USBUART0]
11 set_property PACKAGE_PIN L8 [get_ports CTS_USBUART1]
12
13 #Drive strength constraints for RX and TX pins
14 set_property IOSTANDARD LVCMOS33 [get_ports RX_USBUART0]
15 set_property IOSTANDARD LVCMOS33 [get_ports RX_USBUART1]
16 set_property IOSTANDARD LVCMOS33 [get_ports TX_USBUART0]
17 set_property IOSTANDARD LVCMOS33 [get_ports TX_USBUART1]
18 set_property IOSTANDARD LVCMOS33 [get_ports CTS_USBUART0]
19 set_property IOSTANDARD LVCMOS33 [get_ports CTS_USBUART1]
20 set_property SLEW FAST [get_ports TX_USBUART0]
21 set_property SLEW FAST [get_ports TX_USBUART1]
22 set_property SLEW FAST [get_ports CTS_USBUART0]
23 set_property SLEW FAST [get_ports CTS_USBUART1]
24
```

Figure 4.9: Constraints assigning port names in the block design to PSoC pins

tion platform does not require programmable baud rates, the UARTLite IP is selected for implementing the UART with a baud rate of 921.6 Kbaud. To operate at 921.6Kbaud, the frequency of the clock signal driving the UARTLite IP must be 300MHz or higher. Figure 4.10 shows the configuration of the UARTLite IPs.

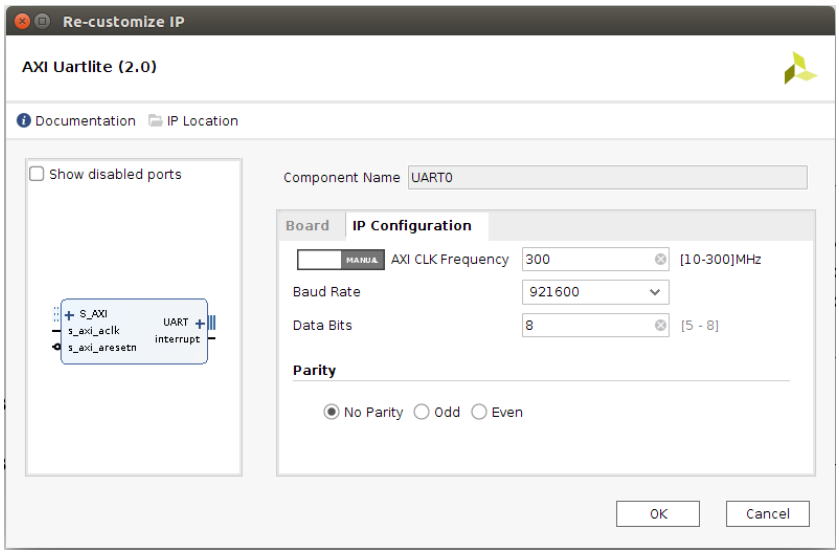


Figure 4.10: Configuration of UARTLite IPs

Figure 3.7 shows the 6 pins of the USBUART Pmod that connect to the Pmod connector. Table 4.1 shows the description of the pins. For the Pmod to function as intended, none of its inputs pins can be left floating. The RXD and CTS are inputs pins, and CTS is a signal that is not generated by the UARTLite IP. Thus, a dummy pin is created in the block design that drives the CTS pin to its active value of 0. In the early stages of this work, the CTS pins were not driven resulting in unstable HITL simulations.

Pin	Signal	Description
1	RTS	Request-to-send
2	RXD	Receive data; the data moving from USB to UART
3	TXD	Transmit data; the data moving from UART to USB
4	CTS	Clear-to-send
5	GND	Power supply ground
6	VCC	Power supply

Table 4.1: USBUART Pmod pin description

4.4.2 Raspberry Pi

The USB standard classifies all hardware supporting the USB protocol as either a USB host or a USB device. A USB host initiates all USB transactions. The USBUART Pmod and the flight controller are both USB devices. Hence data cannot move between USBUART1 and the flight controller without a USB host in between. A USB controller is present in the PSoC and can, therefore, function as a USB host. Using this USB controller would require running an operating system in the PS. An alternative solution is to use a modular single-board computer like the Raspberry Pi to serve as a USB host [15]. The Raspberry Pi has multiple USB ports, to which the USBUART1 and flight controller are connected. Various operating systems are supported on the Raspberry Pi, of which the Raspbian Linux distro is the most popular [14]. The application running on the Raspberry Pi obtains handles to the USBUART1 and flight controller, attempts to read data from each handle, and when there is data, writes the data to the other handle.

4.4.3 I/O Processor

During a HITL simulation, the simulator packs the sensor data in MAVLink packets and sends the MAVLink packets serially. Sending the data in a packetized format ensures protection against missing bytes and missing packets that may lead to corrupted simulation results. For extracting the necessary sensor data from the MAVLink packets, the runtime verification platform has to assemble and decode the serially received data bytes. MAVLink developers release the MAVLink C library containing several convenience functions that help in encoding and decoding MAVLink packets.

The MicroBlaze IP is a soft-core processor that adds software execution capability in the PSoC's PL [48]. The instruction and data memory for the soft-core processor are imple-

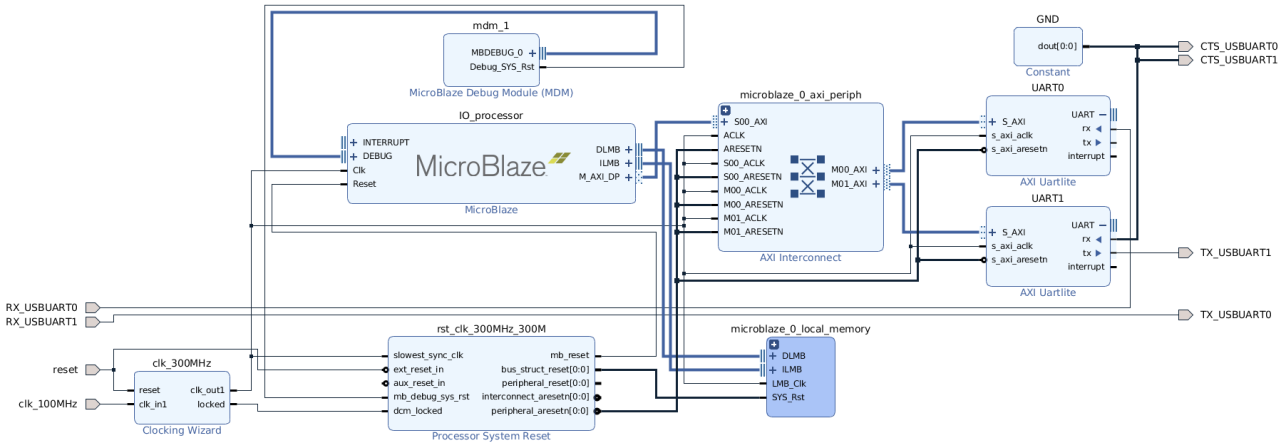


Figure 4.11: Block design of a standalone I/O processing domain

mented in the PL, and the size of the memory is configurable. In the Figure 4.7, the I/O processor is a MicroBlaze processor and is connected to the two UARTs through the AXI interface. Figure 4.11 is the block design of a standalone I/O processing domain. The I/O processor reads the serial data received by UART0, assembles the MAVLink packet, and decodes it using the MAVLink C library functions. The bytes read from UART0 are immediately written to UART1 so they get transferred to the flight controller with the least latency.

Decoding the MAVLink packet is done with the helper function `mavlink_parse_chan`, which takes one input byte at a time and returns the boolean `true` if a valid packet is decoded. Listing 4.1 shows the lines of code that are required to decode the MAVLink packet. `mavlink_status_t` is a struct that stores information regarding the status of the MAVLink packet. `mavlink_message_t` is a struct whose fields store the decoded MAVLink packet. The helper functions can decode MAVLink packets coming from multiple sources referred to as channels. The `chan` argument distinguishes the different channels. The function `mavlink_parse_chan` is called repeatedly for every byte received from UART0, where the data byte is passed as the `byte` argument. The required data fields from the decoded

MAVLink packet are extracted and sent to the monitor processing domain. The MAVLink C library provides handy functions to extract individual fields within the payload.

Listing 4.1: Code to decode MAVLink packets

```
#include <common/mavlink.h>
mavlink_status_t    status;
mavlink_message_t   msg;
int chan = MAVLINK_COMM_0;  // 0 for single channel

...
while(serial.bytesAvailable > 0)
{
   uint8_t byte = serial.getNextByte();
   if (mavlink_parse_char(chan, byte, &msg, &status))
   {
      switch(msg.msgid)
      {
         case MAVLINK_MSG_ID_HIL_GPS:
         // ... Extract the required data fields here ...
      }

   }
}
...
```

If any monitor detects a violation, the monitor processing domain requests the I/O processor to initiate an RCF. The RCF is a predetermined MAVLink packet that instructs the flight controller to perform a specific task, like landing the UAV. Multiple RCFs can be supported in the I/O processor, and the monitor processing domain can choose to initiate one or more RCF. When requested, the I/O processor sends the RCF MAVLink packet to the

flight controller after the completion of any ongoing MAVLink packet transmission that was initiated by the simulator.

4.5 Monitor Processing Domain

The monitor processing domain consists of one or more MicroBlaze soft-core processors that run the monitors. The complete block diagram in Figure 4.2 shows a soft-core processor labeled as the monitor processor. The user interface processing domain provides parameters required for configuring the monitors, and the I/O processing domain provides the sensor data used for evaluating the monitors. Software and hardware monitors are supported in this processing domain. Software monitors are implemented as software functions that take one or more sensor data values as its input arguments and return a boolean value indicating the result of the monitor. Hardware monitors are generated as memory-mapped IPs using Vivado HLS [42]. The IPs have a slave interface through which the monitor processor writes the sensor data and reads the results of the evaluated monitors. The hardware monitor implementation decides if the monitor processor has to block on the monitor results or can execute in a non-blocking fashion.

Implementation and evaluation of different hardware and software monitors for UAVs using the runtime verification platform are outside the scope of this thesis. Lakshman Maalolan, a fellow graduate student working on the same project, is responsible for designing and evaluating the monitors.

4.6 Inter-Processor Communication

In a hardware design with multiple processor cores, various techniques are employed for communicating data between the cores. Shared memory is a common technique used in the implementation of multi-core processors. The Xilinx Mailbox v2.1 IP is a more secure alternative for communicating data between processor cores within a Xilinx PSoC [43]. Three Xilinx Mailbox IPs are used to interconnect the three processing domains. The I/O processor and the monitor processor communicate with the R5 processor using the client-server model. In contrast, communication between the I/O processor and monitor processor uses a FIFO model.

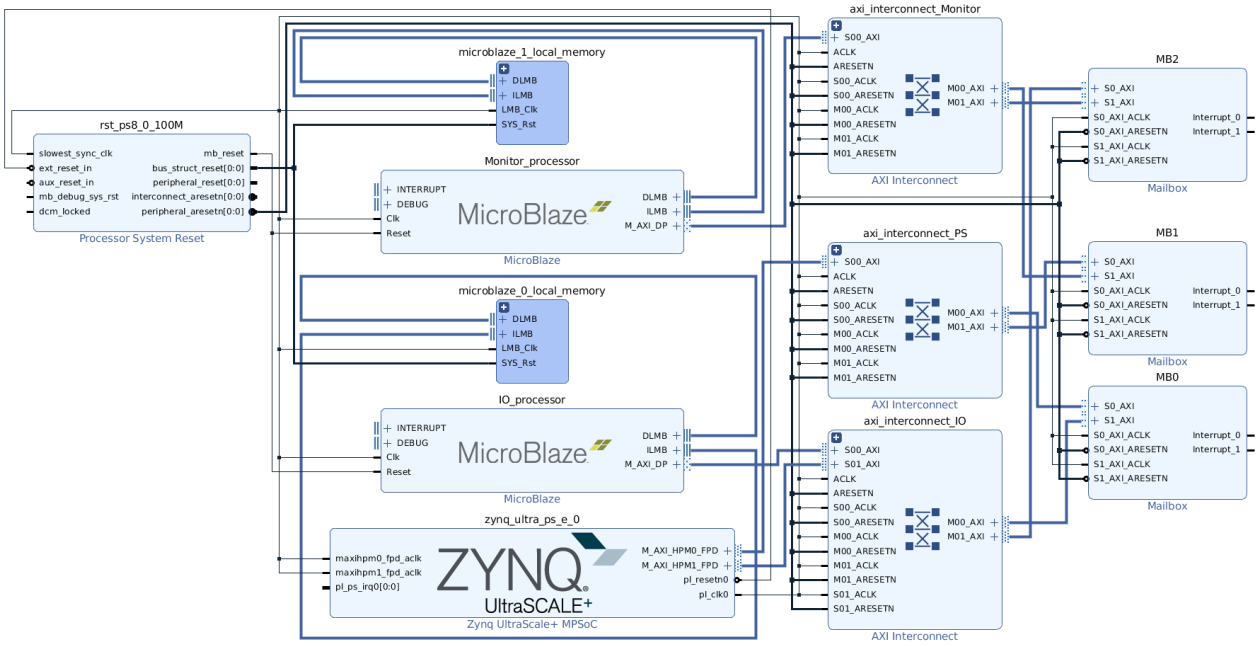


Figure 4.12: Block design of Mailbox IPs interconnecting the three processor cores

4.6.1 Mailbox

Communication through the Xilinx Mailbox IP is bidirectional, and the IP can be connected through various configurable interfaces. All components in the processing domains are connected through the AXI interconnect. Hence the Mailbox IP is also configured to use the AXI interface. When the IP is instantiated in the block design, it provides two AXI slave interfaces. The AXI masters that connect to this interface can send and receive data in a FIFO manner. Bidirectional communication is achieved using two independent FIFOs, where the depth of the FIFO is independently configurable. The depth of the FIFO is configured to the maximum value of 8192 bytes to minimize the chances of overflow that may occur due to the asynchronous execution of the processing domains. The width of the FIFO is 32 bits and cannot be changed. Hence data is written to or read from the FIFO in chunks of 4 bytes. The Mailbox IP also allows the two communicating domains to have asynchronous clocks. With this feature, the power and performance of the processing domains can be optimized independently. Figure 4.12 shows the standalone block design of the Mailbox IPs interconnecting the three processor cores. Each processor core connects to an AXI interconnect IP having one slave and two master interfaces. In this design, each processor core can communicate with all other cores.

4.6.2 TLV Protocol

The Mailbox IP does not define a structure for the communication link. A client-server communication model requires a well-defined structure for the communicated data. A simple protocol, called Tag-Length-Value (TLV) protocol, is developed to add structure, robustness, and flexibility to the communication link between the three asynchronously operating processing domains. TLV is a packet-based protocol where a header field is prefixed to the

payload. The header field is 32 bits long and contains a 16-bit marker, an 8-bit tag field, and an 8-bit length field. The structure of a TLV packet is shown in Figure 4.13. The marker field indicates the start of a new TLV packet. The type of the payload and its length are encoded in the tag and length fields, respectively. The receiver uses these fields to decode the payload.

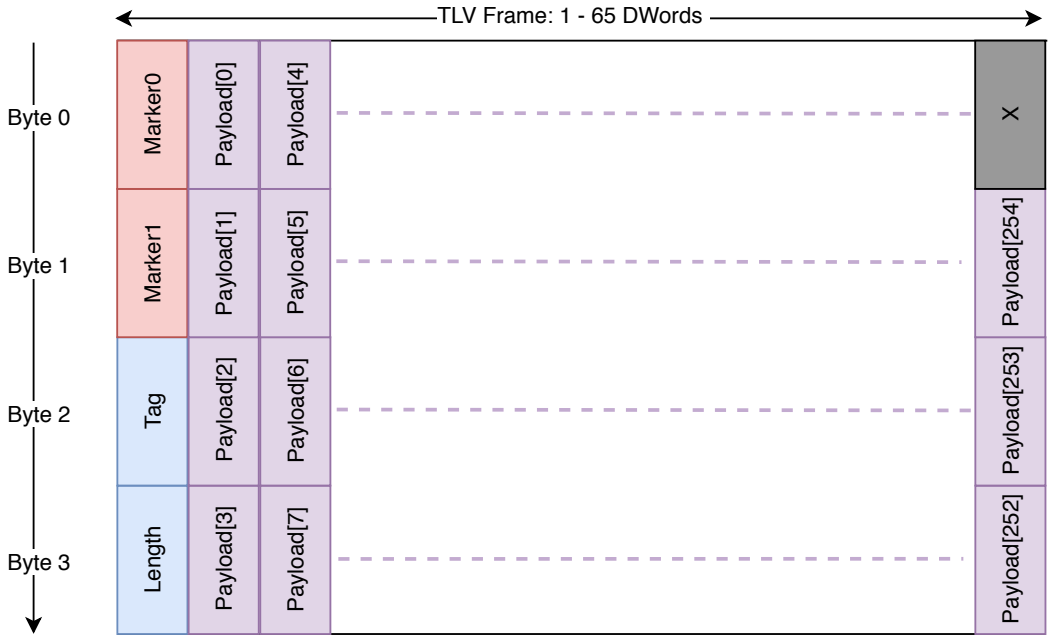


Figure 4.13: Structure of TLV packet

The TLV protocol adds minimal overhead to the Mailbox based communication link. As mentioned in Section 4.6.1, data is written and read in 32-bit blocks. The overhead introduced by TLV is equivalent to an additional write and read operation. For the TLV packets with no payloads, like a request-for-acknowledgment packet or acknowledgment packet, there is no additional overhead introduced by TLV protocol. Communicating the acknowledgment would require at least one write and read operation of the Mailbox, which is the same number of operations with the TLV protocol. Hence, the overhead of the TLV protocol is lesser than one write and read of the Mailbox.

Several helper functions are developed to encode, decode, and unpack data in the TLV format. The implementation of these helper functions is contained within a single header C file. The software code running on the three processing domains can include the header file in the SDK application to use the TLV protocol.

4.7 Simulators

HITL simulation with the runtime verification platform is tested on two popular simulators, Gazebo and AirSim. The two simulators are open source but use different physics engines. The PX4 autopilot software stack supports running simulations on Gazebo natively, whereas AirSim requires small changes in the source code.

The GCS software, like QGroundControl, is used to create a flight mission, control the UAV, and track the UAV's mission. Actions like arming the UAV, starting a mission, and changing the mission midflight require sending MAVLink commands to the flight controller. The GCS connects to the simulator through UDP, and the simulator bridges the two-way communication of MAVLink packets between the GCS and the flight controller. Figure 4.14 shows the connections and interface between the simulator, GCS, and the flight controller.

4.7.1 Gazebo Simulator Setup

Gazebo is a 3D robotics simulator that uses the Open Dynamics Engine (ODE) physics engine to simulate sensors and actuators. The PX4 autopilot software stack supports software-in-the-loop (SITL) and HITL simulation with Gazebo on the Linux platform. The PX4 development guide provides detailed instructions for setting up the Gazebo simulator and for running SITL and HITL simulation [28]. On building the software stack for running a

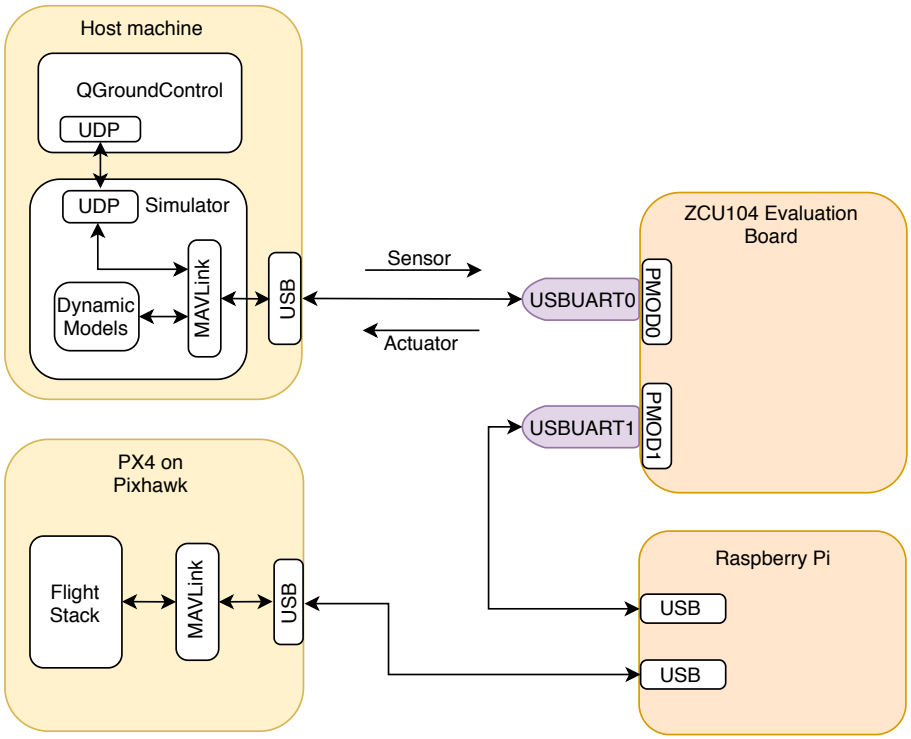


Figure 4.14: Interaction between simulator, GCS and flight controller

simulation, files with a `.sdf` extension are created for each of the supported UAV models. Iris is the quadcopter model used in this work. The `iris.sdf` file contains all the parameters required to configure the simulation. Setting the variable `serialEnabled` to `true` runs a HITL simulation. `serialDevice` is the variable that points to the flight controller hardware. The host machine detects a serial device when the Pixhawk flight controller is connected through USB. On a Linux machine, the handle to the Pixhawk is created as `/dev/ttyACM0`.

When the runtime verification platform is introduced, the simulator sends and receives data from the USBUART Pmod. The handle created for the Pmod is `/dev/ttyUSB0`. For using the Pmod, the variable `serialDevice` should point to the Pmod's handle. The variable `baudRate` configures the serial communication baud rate and is set to 916200. On launching a HITL simulation with these two variables changed, simulation data goes through the verification platform. Figure 4.15 shows a screenshot of a HITL simulation on Gazebo with

the runtime verification platform. Latitude and longitude monitors, implemented as software monitors, are tested on the verification platform. Monitors trigger the RCF as expected when the flight mission is changed mid-flight.

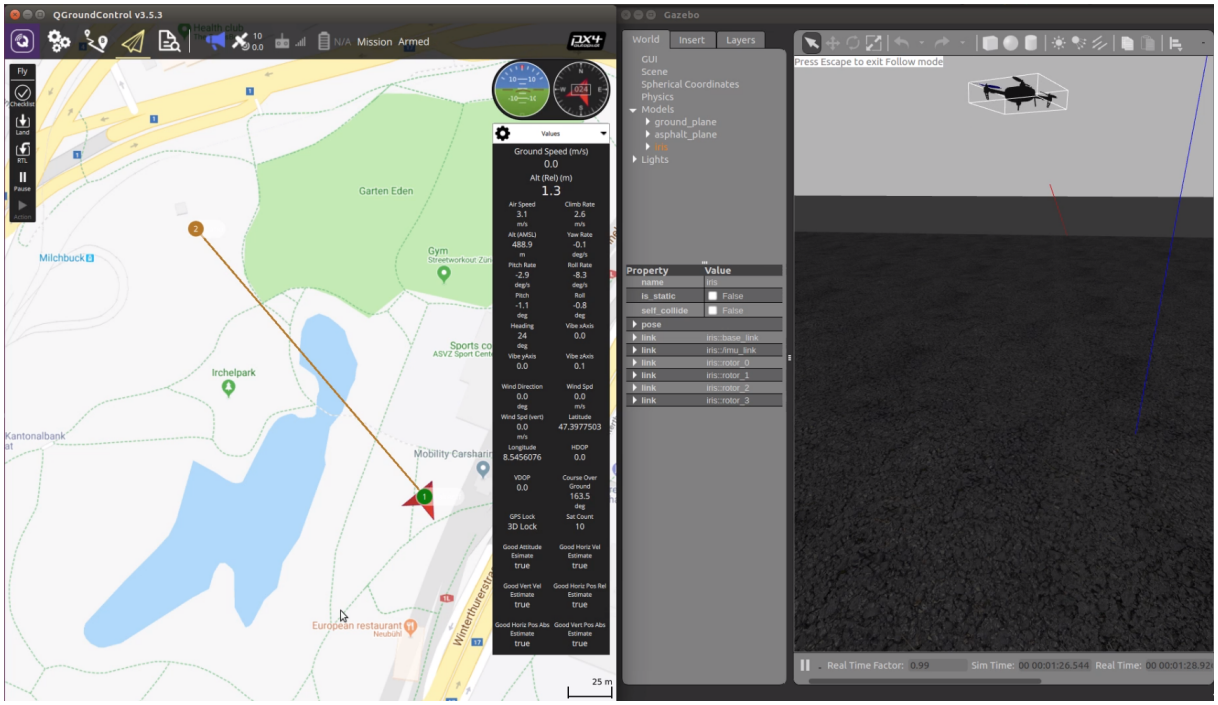


Figure 4.15: Screenshot of HITL simulation with Gazebo

Depending on the number of serial devices connected to the host machine, the handle to the Pmod may change to a different number. This variability is prevented by adding a simple udev rule that runs every time a USB device is connected to the host machine. The rule appended to the file `/etc/udev/rules.d/99-com.rules` is:

```
[SUBSYSTEM=="tty", ATTRS(idVendor)=="0403",
ATTRS(idProduct)=="6001", SYMLINK+="ttyPMOD"]
```

If the USB device's vendor ID is 0403 and the product ID is 6001, this rule creates a symbolic link `/dev/ttyPMOD` which points to the attached USB device. This combination of vendor ID and product ID is specific to Digilent USBUART Pmods. Hence, the variable `serialDevice` can now always point to `/dev/ttyPMOD`.

4.7.2 AirSim Simulator Setup

Microsoft developed the AirSim simulator for AI research. It is used in the development and testing of machine learning algorithms for autonomous ground and aerial vehicles. The simulator is built on Epic Games' Unreal Engine 4 physics engine. Using the Unreal Engine requires a powerful GPU. The Unity Engine is an alternative that can run simulations without a GPU [8]. PX4 does not natively support the AirSim simulator. However, AirSim developers have created guides to help users run HITL simulations with PX4 based flight controllers on AirSim [7].

AirSim uses a `Settings.json` file to modify the simulation parameters, similar to the `iris.sdf` file in the Gazebo environment. The `Settings.json` file is created in the `~/Documents/AirSim` directory. The variable `SimMode` determines the type of vehicle. Setting this variable to `Car` starts the simulation with a ground vehicle model, while `Multirotor` starts with a quadcopter model. The variable `SerialPort` is set to `/dev/ttyPMOD` when running HITL simulation with the verification platform.

For using QGC with AirSim, QGC must be launched before starting the HITL simulation. In contrast, QGC must be launched after starting the HITL simulation when using the Gazebo simulator. Figure 4.16 shows a screenshot of a HITL simulation on AirSim using the Unity Engine with the runtime verification platform.

Since AirSim was originally developed with Unreal Engine and to run on Windows, the source code requires a fix to enable HITL simulation with Unity Engine on Linux. The function `GetVehicleCompanion` in the file `VehicleCompanion.cs` within the directory `UnityDemo/Assets/AirSimAssets/Scripts/Vehicles` requires the fix shown in Listing 4.2.

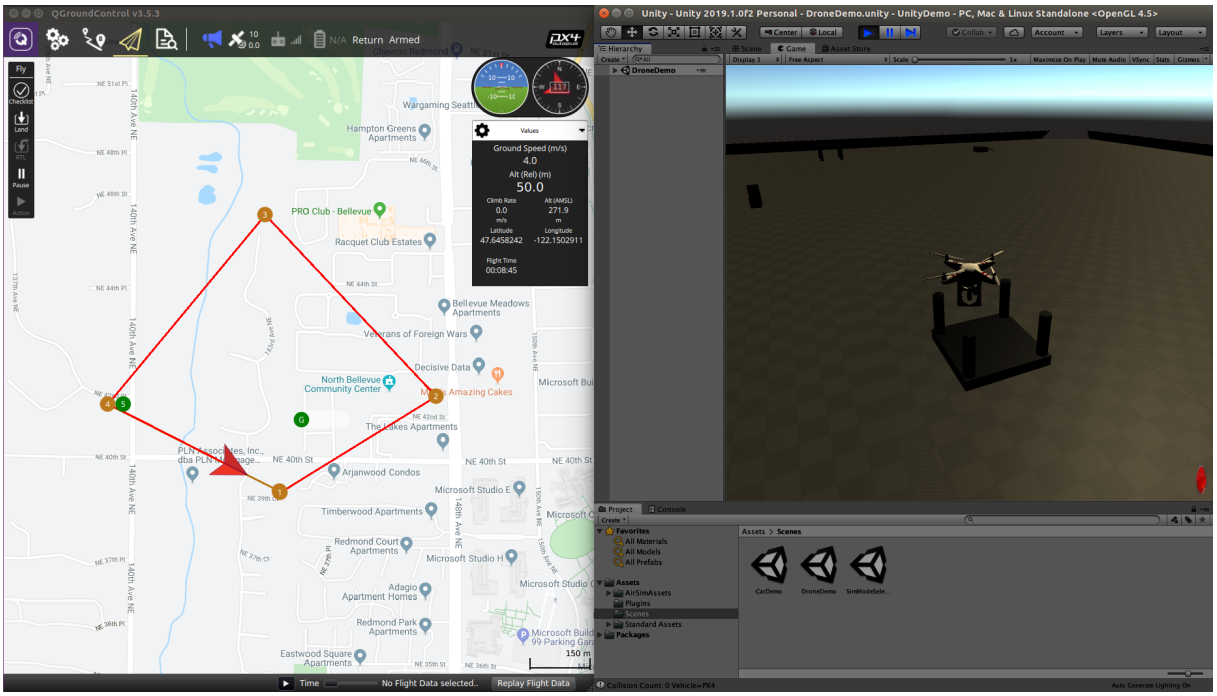


Figure 4.16: Screenshot of HITL simulation with Gazebo

Listing 4.2: Fix for using AirSim with Unity on Linux

```
1 else if (AirSimSettings.GetSettings().SimMode == "Multirotor")
2     companion.vehicleName = "SimpleFlight";
3     companion.vehicleName = AirSimSettings.GetSettings().DefaultVehicleConfig;
```

4.7.3 Simulator Memory Utilization

Gazebo spawns three memory-intensive processes when running a simulation. Table 4.2 shows these processes with the minimum and maximum memory utilization measured between the start and end of the simulation.

AirSim has one memory-intensive process, and its memory utilization is shown in Table 4.3. The memory footprint values are obtained by adding the RSS values of all the memory mappings displayed in `/proc/PID/smaps`, where RSS is how much of the mapping is currently

Process name	Min (KB)	Max (KB)
gazebo	31160	31160
gzserver	118404	118712
gzclient	222572	222572

Table 4.2: Memory utilization of Gazebo simulator

resident in RAM. From the two tables, on average, we see that AirSim has a larger memory footprint compared to Gazebo. Gazebo is the preferred simulator for developing control system algorithms like autopilot firmware, whereas AirSim is preferred for developing computer vision-based machine learning algorithms.

Process name	Min (KB)	Max (KB)
Unity	697516	881108

Table 4.3: Memory utilization of AirSim simulator

Chapter 5

Evaluation and Results

In this chapter, we evaluate the simulation platform in terms of latency and resource utilization in the FPGA. The methodology used for determining the latency is described. Techniques for optimizing the latency and resource utilization are presented, and the improvements are measured. The processing time for inter-processor communication is also measured.

5.1 Latency in the Runtime Verification Platform

A real-time control system like the autopilot software is sensitive to any delays introduced in the control loop. The implementation of runtime verification requires observing various signals used by the autopilot software. The runtime verification platform using HITL simulation presented in this thesis gains visibility of the required signals by intercepting the data link between the simulator running on the host machine and the autopilot software running on the Pixhawk. The data interception mechanism using UART, I/O processor, and Raspberry PI introduces a delay in the control loop.

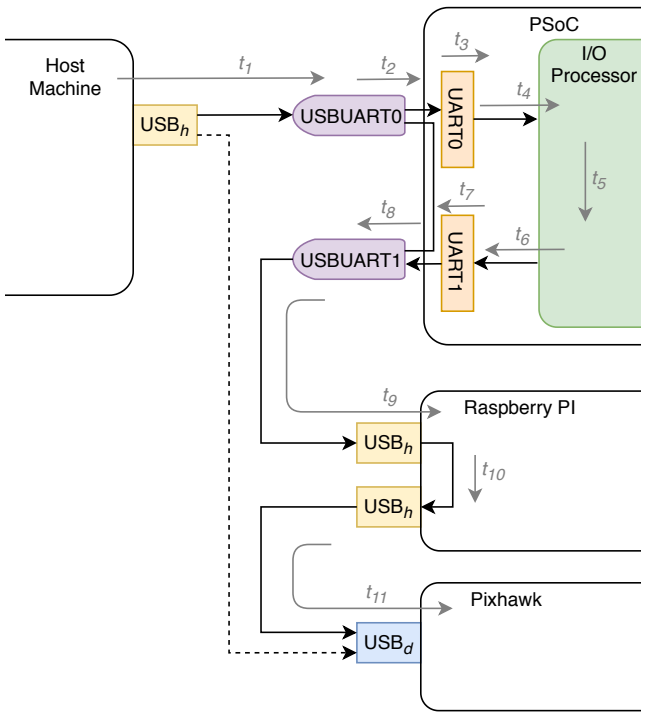


Figure 5.1: Delays in forward dataflow from the simulator to the Pixhawk

5.1.1 Latency Evaluation Methodology

In Figure 5.1, the dashed arrow shows the forward path of dataflow from the simulator to the Pixhawk without the monitoring platform. The solid arrows show the dataflow through the various components in the monitoring platform. The dataflow from each component to the next can be visualized as a hop, and each hop injects a delay that contributes to the total delay added by the monitoring platform. The gray-colored arrows in Figure 5.1 represent these individual delays.

- t_1 : The time it takes to send data through the USB interface. It includes the execution time of the USB drivers at the host-end and the low-level USB protocol implemented in hardware at the host-end and the device-end.
- t_2 : The time the USBUART Pmod (USBUART0) takes to encode a byte of data into

a physical UART signal. It depends on the baud rate configured at the USB host.

- t_3 : The time the hardware IP UART0 takes to decode the physical UART signal and store it in its read-buffer.
- t_4 : The I/O processor's execution time to read data from the read-buffer through the AXI interface. The operating frequency of the I/O processor and the AXI interface determines this time delay.
- t_5 : Execution time for any software code between reading and writing the UART data. The software code can include decoding the MAVLink message, constructing a TLV message, and sending the TLV message to the monitor processor.
- t_6 : The I/O processor's execution time to write data to the UART hardware IP's write-buffer. It includes the AXI interface delay.
- t_7 : The time the hardware IP UART1 takes to encode the data available in the write-buffer into a physical UART signal.
- t_8 : The time the USBUART Pmod (USBUART1) takes to decode the physical UART signal into data bytes and store it in a buffer that can be read by a USB host.
- t_9 : The time Raspberry Pi takes to read data from USBUART1 through the USB interface. It includes the execution time of the USB drivers and the low-level USB protocol implemented in hardware at the host-end (Raspberry Pi) and the device-end (USBUART1).
- t_{10} : The software execution time between reading the data from the Pmod to writing the data on Pixhawk's interface. The software code checks if the number of bytes read from the Pmod is non-zero and writes the same number of bytes on Pixhawk's serial handle.
- t_{11} : The time required to send the data to Pixhawk through the USB interface. The

process is similar to t_1.

In the system without the monitoring platform, shown with the dashed arrows, the communication delay is due to the USB driver and the low-level hardware implementation of the USB protocol, which is equivalent to t_1. The monitoring platform adds delays t_2 through t_{11}. A forward loopback path is created to evaluate the aggregated delay of t_2 through t_{11}. Figure 5.2 shows the forward loopback path where the connections in red are removed, and the connection in green is added. A hardware timer implemented in the FPGA measures the time taken to send data through UART1 that gets looped back and received on UART0.

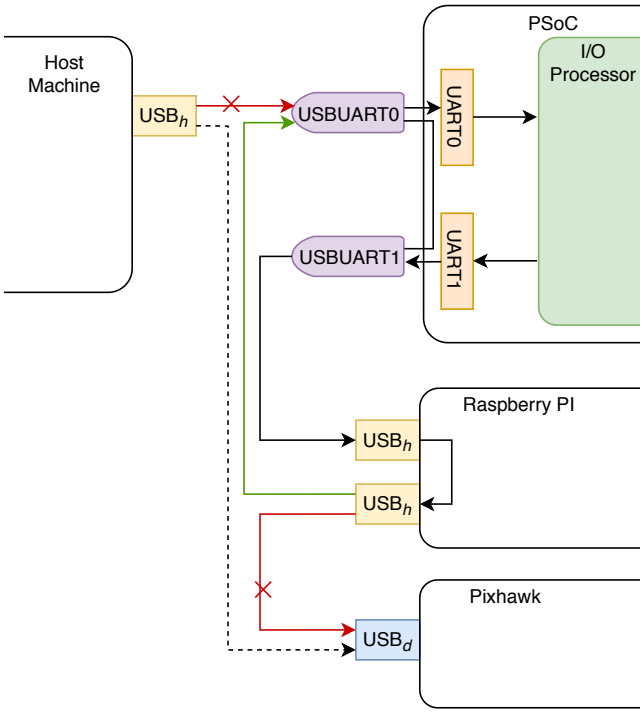


Figure 5.2: The forward loopback path

Dataflow from the Pixhawk to the simulator without the monitoring platform is shown in Figure 5.3 with the dashed arrow. The monitoring system adds a backward path similar to the forward path with one difference. The monitoring system does not need to observe the

actuator signals going back from the Pixhawk to the simulator. Hence, the UART signals do not need to be decoded by the I/O processor. The physical UART signal going from USBUART1 to USBUART0 is bypassed through the FPGA. Consequently, the backward path does not have delays t_3 through t_7. Figure 5.4 shows the loopback path used for measuring the aggregated delay in the backward path. The connections in red are removed, and the connection in green is added. Since the UART signal in the backward path is not decoded in the FPGA, the hardware timer cannot be used to measure the loopback delay. A software timer in the Raspberry Pi is used to measure the loopback delay in the backward path. Data is written to USBUART1 that gets looped back to the Raspberry Pi through USBUART0. Since the data takes a different path in the forward and backward direction, the delay introduced by the monitoring platform is asymmetrical and hence needs to be measured independently.

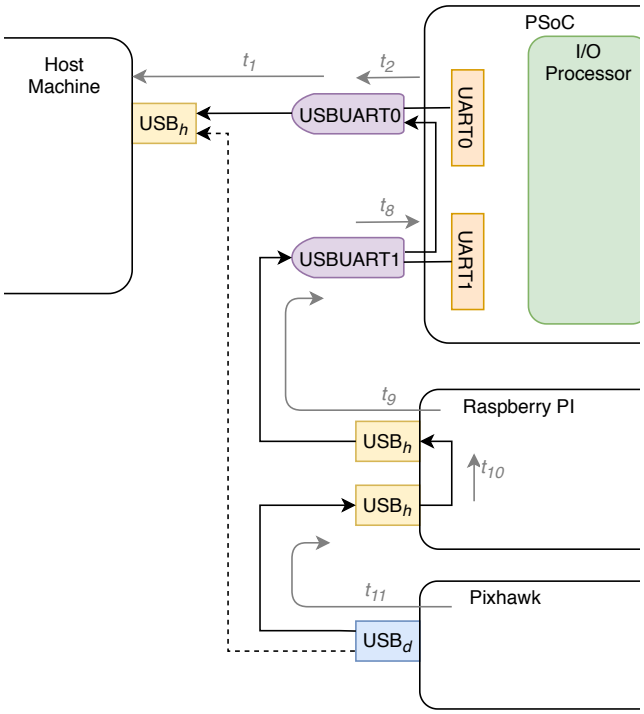


Figure 5.3: Delays in backward dataflow from the Pixhawk to the simulator

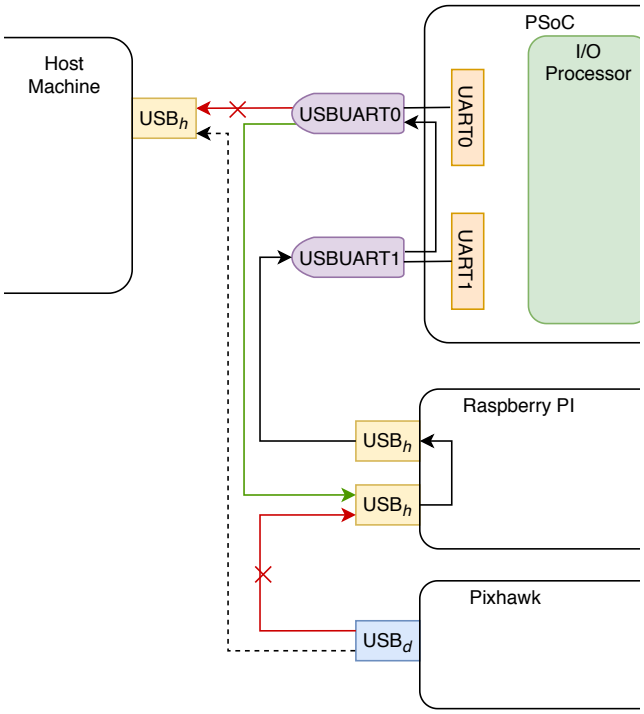


Figure 5.4: The backward loopback path

5.1.2 Bulk Transfer in USB

Achieving a higher throughput is prioritized over latency in the USB protocol by default. Hence a USB interface is well suited for applications that transfer data in chunks than for applications that transfer data one byte at a time.

Transferring data through a USB interface makes use of the underlying USB drivers in the operating system (OS). Invoking the OS and executing the driver leads to higher latency of the USB interface as compared to a simpler interface like a UART. However, the USB interface provides much higher throughput. Thus, it is suitable for applications such as HITL simulation that require transferring large chunks of data at sparse intervals as opposed to applications that frequently transfer a small number of bytes. The sensor data in a HITL simulation is approximately sent once every 200ms and is packetized in MAVLink messages

that can range from 12 bytes to over 256 bytes. Due to the bulk transfer properties of USB, the total latency and the latency normalized over the payload size are measured for single and multi-byte payloads.

The FTDI FT232R chip present on the USBUART Pmod performs the translation between the USB and UART. The FT232R has a 128-byte deep RX FIFO to buffer the data going from the USB interface to the UART interface and a 256-byte deep TX FIFO to buffer the data going from the UART interface to the USB interface. The depth of the TX and RX FIFO in the hardware UART IPs UART0 and UART1 is increased from 16 bytes to 512 bytes to support the bulk transfers with an adequate margin. Hence the payload sizes used for latency measurement are between 1 and 512 bytes.

5.1.3 Latency Optimization

The FT232R has an internal timer called `latency_timer` that determines the longest time for which data received from UART could get buffered in the device before getting sent to the USB host. It is a programmable timer that can be modified to achieve the right balance between latency and processing load on the USB host. Reducing this timer value causes data to be sent to the USB host more frequently, thereby reducing the latency.

The FT232R receives data from UART and stores it in the 256-byte deep TX FIFO. When requested, the FT232R sends the buffered data to the USB host if the buffer has the requested number of bytes or if the `latency_timer` has counted down to zero. The `latency_timer` is reinitialized and started when the last byte is transferred from the buffer to the USB host. The default value of the `latency_timer` is 16ms. With the default value, a single byte sent to the Pmod through UART can potentially get read by the host 16ms later. 1ms is the smallest value that can be set to `latency_timer`, and this setting should minimize the

overall latency introduced by the monitoring platform.

5.1.4 Latency Measurement Software Code

Algorithm 1 shows the high-level algorithm for measuring the latency of the forward loopback path described in Figure 5.2. The I/O processor runs the software code and has access to a hardware timer, referred to as `timer0`, that is implemented on the FPGA. Sending the actual payload after executing operations 4 and 5 in Algorithm 1 causes the FT232R to introduce maximum latency and, therefore, to maximize the overall latency. The software code running on Raspberry Pi reads data from USBUART1 and writes it to USBUART0 in an infinite loop.

Algorithm 1 Latency measurement of forward loopback path

1: initialize UART0, UART1, `timer0`
2: initialize the `payload` array
3: **for** *iteration* $= 1 \ldots 10$ **do**
4: send `payload[0]` on UART1
5: wait until `payload[0]` is received on UART0
6: start `timer0`
7: send entire `payload` on UART1
8: wait until entire `payload` is received on UART0
9: stop `timer0`
10: read and print the value of `timer0`
11: reset `timer0`
12: sleep for 100ms
13: **end for**

Measuring the latency of the backward loopback path, described in Figure 5.4, uses the same algorithm with the following changes:

- The software code runs on the Raspberry Pi and not on the I/O processor.
- The payload is read and written to USBUART0/1 instead of UART0/1.

- A software timer is used in the Raspberry Pi in the place of the hardware timer.

5.1.5 Latency Results

Five different payload sizes are chosen between 1 byte and 512 bytes, and for each of the payload sizes, the latency is measured ten times and the average latency is reported. The latencies for the forward and backward paths, with and without the `latency_timer` optimization, are plotted in Figure 5.5 and tabulated in Table 5.1.

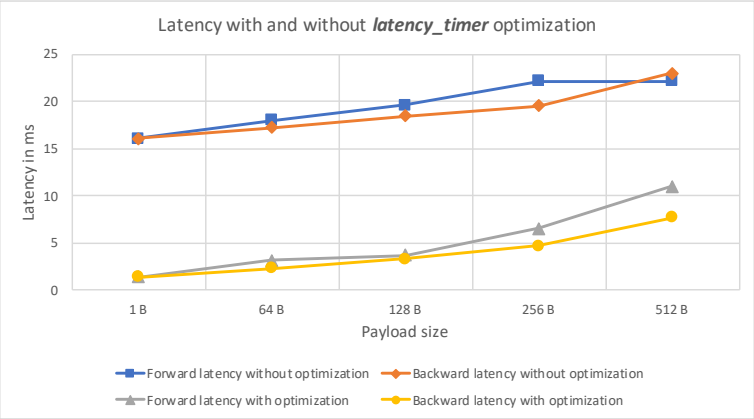


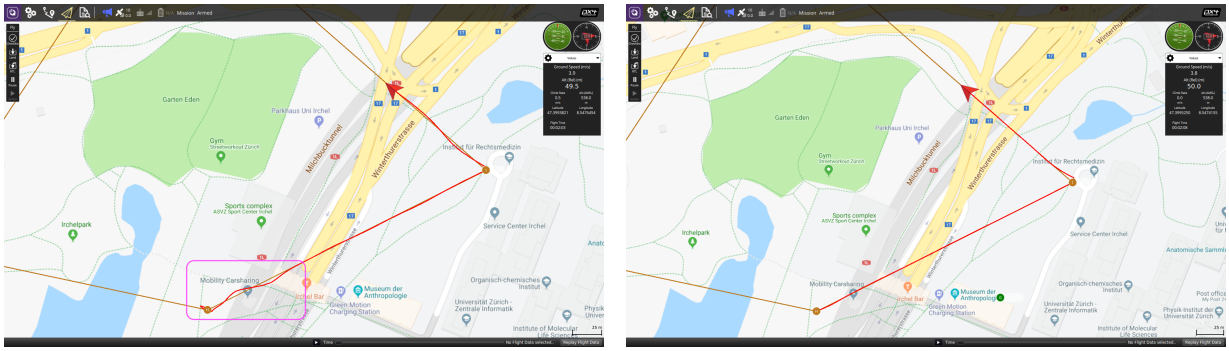
Figure 5.5: Forward and backward path latency

Payload size (bytes)	Latency (ms)			
	Without optimization		With optimization	
	Forward path	Backward path	Forward path	Backward path
1	16.09	16.09	1.38	1.39
64	17.96	17.24	3.13	2.25
128	19.64	18.49	3.66	3.27
256	22.13	19.50	6.50	4.65
512	22.13	22.97	11.02	7.69

Table 5.1: Forward and backward path latency

Due to the bulk transfer properties of USB, we see that the overall latency increases slowly even when the payload size is doubled. Compared to the backward path, the forward path

has the I/O processor and an additional UART that translates data between bytes and serial UART signals. The latencies of the forward and backward paths are comparable. Thus, the I/O processor and the UART have a small contribution to the overall latency. The upward trend in the graph is due to the linear increase in time to transfer the payload with the UART serially. At a baud rate of 921.6 Kbaud, it takes 5.56ms to transmit 512 bytes of data. The effect of the `latency_timer` settings is evident, with approximately 15ms of delay separating the pair of curves. Figures 5.6a and 5.6b are screenshots of the HITL simulation with the `latency_timer` set to 16ms and 1ms, respectively. Simulation with a 16ms setting creates slight oscillations, which cause the UAV to sway side to side along the flight path. This is highlighted in Figure 5.6a. This effect is not present in a simulation with a 1ms setting.



(a) With 16ms setting (b) With 1ms setting

Figure 5.6: HITL simulation without and with latency_timer optimization

Figure 5.7 shows the normalized latency on a log scale, and the corresponding data is present in Table 5.2. The main observation from the graph is that the normalized latency or the latency per byte reduces significantly with increasing packet sizes. The size of the transmit and receive buffers present in the hardware UART IP and the FT232R chip become the bottleneck for increasing the packet size indefinitely.

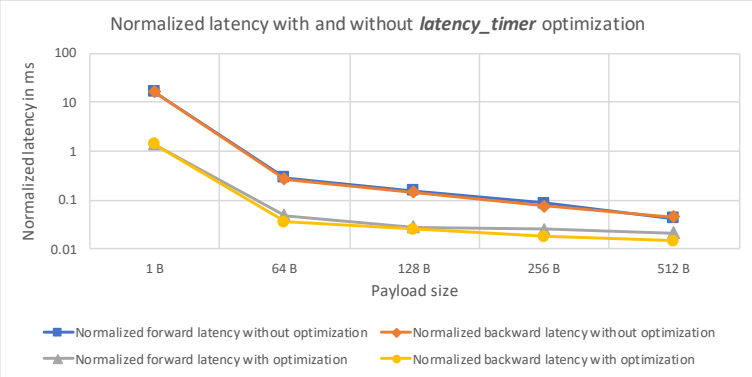


Figure 5.7: Forward and backward path normalized latency

Payload size (bytes)	Latency (ms)			
	Without optimization		With optimization	
	Forward path	Backward path	Forward path	Backward path
1	16.09	16.09	1.38	1.39
64	0.28	0.27	0.05	0.03
128	0.15	0.14	0.03	0.03
256	0.09	0.08	0.02	0.02
512	0.04	0.04	0.02	0.01

Table 5.2: Forward and backward path normalized latency

5.2 Implementation Analysis

When a hardware design is implemented in ASIC, the power, performance, and area (PPA) metrics, determine the quality of implementation. For the designs implemented in an FPGA, resource utilization is used instead of the area metric. A good design is one that minimizes power and area and simultaneously maximizes the performance. The maximum operating frequency is a measure of performance. For this research work, minimizing resource utilization is the primary focus. Maximizing the operating frequency is not necessary as the hardware implemented on the FPGA introduces a minimal delay in the HITL simulation. Instead, reducing the operating frequency is desirable as lower frequency results in reduced power dissipation.

5.2.1 Configurable IPs

The monitoring platform uses the FPGA present in the Xilinx Zynq chip to implement the I/O processor, monitor processor, UART hardware IPs, Bluetooth IP, and the interconnects, all of which have configurable settings. The Xilinx Mailbox v2.1 IP that is used to create the interface between the processors has a configurable FIFO depth. Increasing the FIFO depth leads to higher resource utilization. A bigger FIFO logic can get mapped to Ultra-RAM instead of Block RAM. Similarly, different configurations of the MicroBlaze IP utilize different FPGA resources to implement the soft processor. The I/O processor configuration does not have to be the same as the monitor processor configuration. The I/O processor does not perform arithmetic calculations, whereas software monitors running on the monitor processor would.

The UART hardware IP, Xilinx UARTLite v2.0, supports configuring the baud rate, but the FIFO depth is not configurable. Changes are made in the VHDL source code of the IP to increase the FIFO depth to 512 bytes.

5.2.2 Resource Optimization with Hardware Monitors

The runtime verification platform supports monitors implemented in hardware and software. For the software monitors to run optimally, the monitor processor configuration instantiates a floating-point unit, a hardware multiplier, and a hardware divider. These specialized hardware accelerator blocks speed up the arithmetic operations that are frequently used in the software monitors and are essential for supporting the timely execution of large numbers of software monitors. However, when all the software monitors get implemented as hardware monitors, there is no further need for the hardware accelerators and can be excluded from the monitor processor.

5.2.3 Multi-Clock Domain Design

The baud rate of all the UART interfaces is configured to 921.6 Kbaud. For the Xilinx UARTLite v2.0 hardware IP to operate at 921.6 Kbaud, the clock input to this IP should be at a minimum of 300MHz. Xilinx provides an IP called Clocking Wizard v6.0 which generates the clocking circuitry [52]. This IP uses the mixed-mode clock manager (MMCM) resource present in the FPGA. The IP can be configured to generate multiple clock signals, and each of the generated clocks can be of different frequency. Using the Clocking Wizard IP, a multi-clock domain scheme is designed. The UART IPs and its interface with the AXI Interconnect v2.0 IP operate at 300MHz and the rest of the blocks at 150MHz.

Multi-clock domain designs require careful handling of signals that cross the clock domain boundaries. Special circuits called clock domain crossing (CDC) cells should be inserted on the clock domain boundaries, and CDC checks should be performed. The synthesis tool automatically takes care of placing CDC cells and running the CDC checks. Implementing this multi-clock domain scheme reduces the overall resource utilization and makes timing closure of the lower frequency blocks easy.

5.2.4 Implementation Reports

The resource utilization, timing, and power consumption estimates for three configurations of the monitoring platform are presented. The details of the three configurations are captured in Table 5.3. The resource utilization for the three configurations of the monitoring platform is shown in Table 5.4. Figure 5.8a shows the percentage reduction of resources utilized in configuration 2 and 3 compared to configuration 1.

In a design with failing timing paths, the worst negative slack (WNS), measured in ns, is the amount by which the worst path fails to meet the timing requirement. WNS is a

		Configuration 1	**Configuration 2**	**Configuration 3**
Multi-clock design		No	Yes	Yes
I/O Processor	Operating frequency Preset configuration Hardware accelerators	300 MHz Application FPU, multiplier, divider	150 MHz Application FPU, multiplier, divider	150 MHz Microcontroller -
Monitor Processor	Operating frequency Preset configuration Hardware accelerators	300 MHz Application FPU, multiplier, divider	150 MHz Application FPU, multiplier, divider	150 MHz Microcontroller -
UART	Clock frequency Baud rate FIFO depth	300 MHz 921600 baud 512 Bytes		

Table 5.3: The three configurations used for PPA analysis

	Configuration 1	**Configuration 2**	**Configuration 3**	**Available**
LUT as logic	11040	10932	7989	230400
LUT as memory	1273	1273	1166	101760
CLB registers	11029	10517	8215	460800
BRAM	96	94	94	312
URAM	4	4	4	96
DSP	8	8	0	1728
BUFG	4	5	4	544
I/O	19			360
MMCM	1			8

Table 5.4: Resource utilization summary

positive number for designs that meet timing, and it is a measure of the ease with which the implementation tool can perform place and route. The implementation of all three configurations has no failing timing paths. Table 5.5 reports the min-time (setup) and max-time (hold) WNS for the three configurations.

The total power dissipation of a chip is the sum of the static power and dynamic power. Static power is mainly due to leakage current. The supply voltage determines the static power. Techniques like power gating can reduce static power dissipation. Dynamic power

	Configuration 1	Configuration 2	Configuration 3
Setup WNS (ns)	0.017	0.345	0.056
Hold WNS (ns)	0.017	0.016	0.013

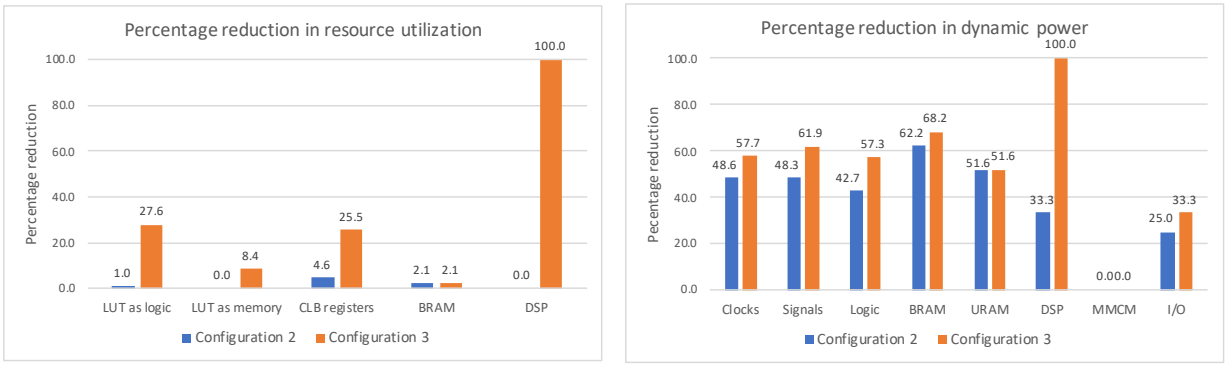
Table 5.5: WNS for setup and hold time checks

	Dynamic power estimate (W)		
	Configuration 1	Configuration 2	Configuration 3
Clocks	0.142	0.073	0.06
Signals	0.118	0.061	0.045
Logic	0.082	0.047	0.035
BRAM	0.201	0.076	0.064
URAM	0.031	0.015	0.015
DSP	0.003	0.002	0
MMCM	0.099	0.099	0.099
I/O	0.024	0.018	0.016
PS	2.684	2.678	2.678
Total	3.386	3.069	3.012

Table 5.6: Dynamic power estimate

is the power dissipation that can be attributed to toggling signals, and is proportional to the frequency of toggling and square of the supply voltage. Xilinx Vivado performs power estimation for the whole Zynq PSoC and reports static and dynamic power at a more granular level. Table 5.6 shows the main components of dynamic power for the three configurations of the monitoring platform. PS has the most significant contribution to dynamic power and remains unchanged in the three configurations. Figure 5.8b shows the percentage reduction of dynamic power dissipation of configuration 2 and 3 compared to configuration 1.

From Figures 5.8a and 5.8b showing the relative reduction in resources and power, we see that reducing the operating clock frequency in configuration 2 results in a significant reduction in dynamic power. The optimization in resource utilization is not as significant as the optimization in power. Configuration 3, where the hardware accelerators are removed and operating frequency is reduced, results in significant optimization in resource utilization and dynamic power. Configuration 3, paired with `latency_timer` optimization, is the preferred



(a) Resource optimization

(b) Power optimization

Figure 5.8: Percentage reduction in resource utilization and power dissipation

choice for implementing hardware monitors.

5.3 Inter-Processor Communication Evaluation

Data communication between the I/O processor and the monitor processor happens over the Mailbox interface. The I/O processor intercepts the GPS MAVLink message, picks the relevant fields, packages it into a GPS TLV message, and writes the TLV message to the mailbox. Packaging the GPS data into a TLV message takes a fixed number of I/O processor cycles. Thus, the time taken to create a GPS TLV message depends on the operating frequency of the I/O processor. The number of CPU cycles and the corresponding time taken when operating at 150MHz and 300MHz is tabulated in Table 5.7.

CPU cycles	**Execution time at 150MHz**	**Execution time at 300MHz**
145	966us	483us

Table 5.7: Execution time for packing a GPS TLV message

Similar to creating the GPS TLV message, writing the TLV message to the mailbox takes a fixed number of CPU cycles. The GPS TLV message is 24 bytes or 6 Dwords in size. Table 5.8 captures the number of CPU cycles and the corresponding time taken when operating

at 150MHz and 300MHz for writing the 6 Dwords to the mailbox.

CPU cycles	Execution time at 150MHz	Execution time at 300MHz
483	3.22ms	1.61ms

Table 5.8: Execution time for writing a GPS TLV message to mailbox

The execution time analysis is only performed for GPS data, as this is the only data communicated from the I/O processor to the monitor processor for the current set of monitors implemented. When new monitors get added in the future, more simulation data may be required by the monitor processor, and similar execution time analysis can be performed.

GPS packets are sent every 200ms from the simulator. The buffer depth in the mailbox IP is configured to 4096 bytes. Each GPS TLV message is 24 bytes long. Hence, 170 GPS messages, equivalent to 34 seconds of simulation, can get buffered in the mailbox before the I/O processor stalls. The monitor processor should consume the GPS TLV messages at a steady rate of 5 GPS TLVs per second to avoid stalling.

Chapter 6

Conclusions

The adoption of autonomous vehicles in everyday life is approaching as technological advances continue to happen at an unprecedented rate. Ensuring the safe operation of these autonomous vehicles will be essential. Trust can be enhanced by actively monitoring the physical state of the system using runtime verification techniques.

Using UAVs as an example, this thesis presented a platform for developing and validating software and hardware monitors using programmable hardware. HITL simulations are employed in this platform as it eliminates the complications associated with testing the monitors on a real UAV. Bluetooth connectivity was added to the platform to create an interface for transferring the flight plan and initiating the monitoring system. The TLV protocol was developed to facilitate inter-processor communication. Simulations were run to validate the proper functioning of monitors and the execution of recovery functions. The platform is designed to support the invocation of multiple recovery functions.

The latency introduced by the platform was measured and further optimized. Three different configurations of the platform were analyzed for resource utilization, timing closure, and power consumption. The platform is validated with two simulators, each catering the needs of different classes of developers: Gazebo for the autopilot software developers, and AirSim for machine learning application developers.

6.1 Future Work

The Xilinx UltraScale+ PSoC used in this thesis has a rich set of processor cores and peripherals. The Arm Cortex-A53 quad-core processor is left unused so that applications like surveillance using a camera module can run on it. Further, machine learning algorithms that control the UAV can be integrated with the monitors using this platform. The AirSim and Gazebo simulators can generate video streams from one or more virtual cameras mounted on the simulated UAV. These video streams capture the environment from the UAV's perspective, and thus, can be used with machine learning algorithms to control the UAV.

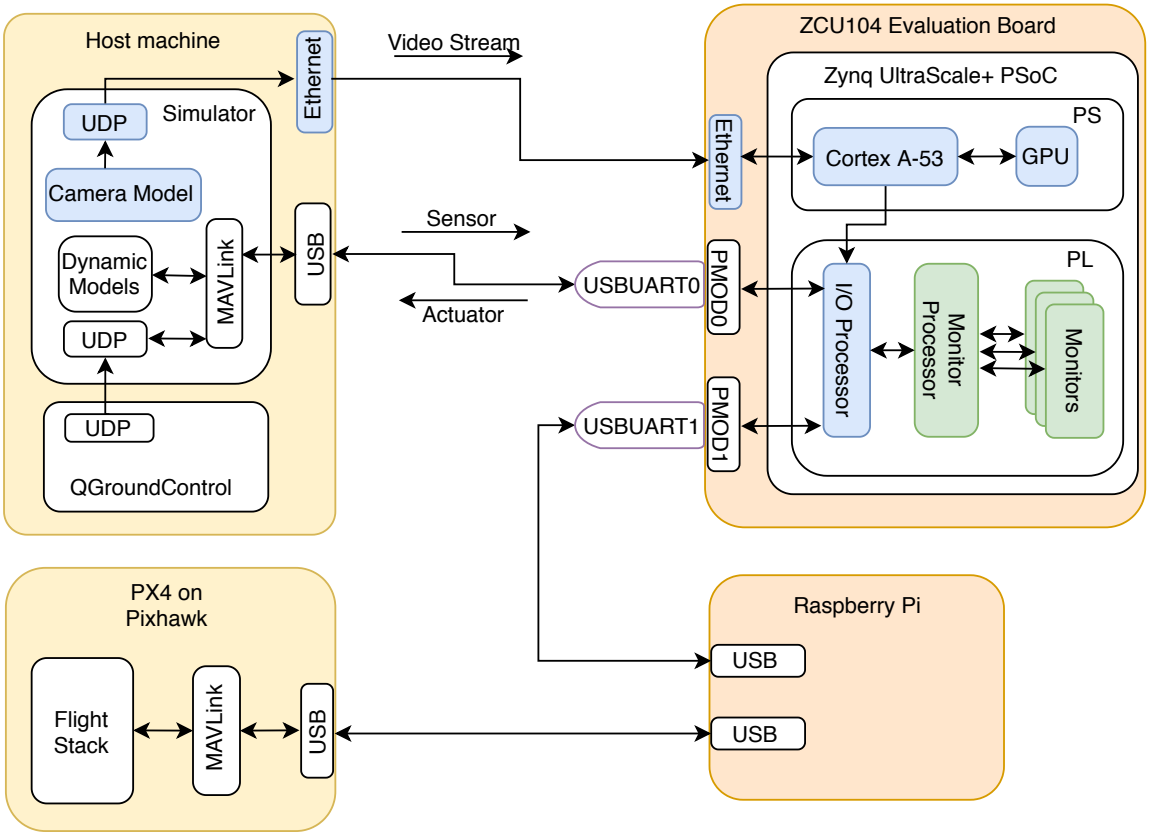


Figure 6.1: High-level block diagram for integrating ML algorithms

Figure 6.1 shows the high-level block diagram of one possible solution, where the components

highlighted in blue show the changes needed to the current platform. The simulator sends the video streams through a UDP port over an Ethernet connection to the application processor. Machine learning algorithms using the video streams can run on the application processor and can also use the GPU. The control commands generated by the algorithm are sent to the Pixhawk through the I/O processor. In the current implementation, the I/O processor forwards sensor data and QGC commands from the simulator to the flight controller, and occasionally injects RCF commands to the flight controller. In the proposed architecture, commands from the application processor could also be sent to the flight controller.

AirSim has the support to simultaneously spawn two or more UAVs in a HITL simulation. This feature could be explored to test sense-and-avoid algorithms and eventually build safety monitors for obstacle avoidance.

Bibliography

[1] Federal Aviation Administration. Airspace Restrictions. URL https://www.faa.gov/uas/recreational_fliers/where_can_i_fly/airspace_restrictions/. [Accessed: 2019-07-25].

[2] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. *Introduction to Runtime Verification*, pages 1–33. Springer, Cham, 2018. ISBN 978-3-319-75632-5. doi: 10.1007/978-3-319-75632-5_1. URL https://doi.org/10.1007/978-3-319-75632-5_1.

[3] Bloomberg L.P. UAV Market worth USD 45.8 Billion by 2025 - MarketsandMarkets, 2019. URL https://www.bloomberg.com/press-releases/2019-10-29/unmanned-aerial-vehicle-uav-market-worth-45-8-billion-by-2025-exclusive-report-by-marketsandmarkets.

[4] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. *Electronic Proceedings in Theoretical Computer Science*, 254:15–28, 09 2017. doi: 10.4204/EPTCS.254.2.

[5] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[6] Microsoft Corporation. AirSim Simulator, 2017. URL https://github.com/microsoft/AirSim. [Accessed: 2019-08-19].

[7] Microsoft Corporation. PX4 Setup for AirSim, 2017. URL https://microsoft.github.io/AirSim/px4_setup/. [Accessed: 2019-08-19].

[8] Microsoft Corporation. AirSim on Unity - Linux, 2019. URL https://microsoft.github.io/AirSim/Unity/#linux. [Accessed: 2019-08-19].

[9] Digilent, Inc. PmodUSBUART Reference Manual rev. A, 2016. URL https://reference.digilentinc.com/_media/reference/pmod/pmodusbuart/pmodusbuart_rm.pdf. [Accessed: 2019-04-25].

[10] Digilent, Inc. PmodBT2 Reference Manual rev. A, 2019. URL https://reference.digilentinc.com/_media/reference/pmod/pmodbt2/pmodbt2_rm.pdf. [Accessed: 2019-04-25].

[11] Digilent, Inc. Digilent Pmod Interface Specification 1.2.0, 2017. URL https://reference.digilentinc.com/_media/reference/pmod/pmod-interface-specification-1_2_0.pdf. [Accessed: 2019-01-25].

[12] Dronecode. Dronecode website. URL https://www.dronecode.org/about/.

[13] Melvin Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[14] Raspberry Pi Foundation. Raspbian OS. URL https://www.raspbian.org.

[15] Raspberry Pi Foundation. Raspberry Pi 4 Tech Specs, 2019. URL https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/.

[16] Gazebo. Gazebo Simulator Webpage. URL http://gazebosim.org/.

[17] Christian Jacobi, Kai Weber, Viresh Paruthi, and Jason Baumgartner. Automatic formal verification of fused-multiply-add FPUs. In *Design, Automation and Test in Europe*, pages 1298–1303. IEEE, 2005.

[18] Aaron Kane, Omar Chowdhury, Anupam Datta, and Philip Koopman. A case study on runtime monitoring of an autonomous research vehicle (ARV) system. In *Runtime Verification*, pages 102–117. Springer, 2015.

[19] N. Kitchen and A. Kuehlmann. Stimulus generation for constrained random simulation. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 258–265, 2007.

[20] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009. URL https://doi.org/10.1016/j.jlap.2008.08.004.

[21] MAVLink. MAVLink Developer Guide. URL https://mavlink.io/en/. [Accessed: 2019-04-01].

[22] NASA. UAS Traffic Management, 2015. URL https://utm.arc.nasa.gov/index.shtml. [Accessed: 2019-07-25].

[23] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In *2008 Real-Time Systems Symposium*, pages 481–491, Nov 2008. doi: 10.1109/RTSS.2008.43.

[24] Andrew Putnam. Fpgas in the datacenter: Combining the worlds of hardware and software development. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI '17, page 5, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349727. doi: 10.1145/3060403.3066860. URL https://doi.org/10.1145/3060403.3066860.

[25] PX4. PX4 Firmware GitHub Repository, 2012. URL https://github.com/PX4/Firmware. [Accessed: 2019-01-25].

[26] PX4. PX4 Firmware GitHub Releases, 2012. URL https://github.com/PX4/Firmware/releases. [Accessed: 2019-01-25].

[27] PX4. jMAVSim Simulator, 2013. URL https://github.com/PX4/jMAVSim.

[28] PX4. Gazebo HITL simulation, 2017. URL https://dev.px4.io/v1.9.0/en/simulation/hitl.html#jmavsimgazebo-hitl-environment. [Accessed: 2019-01-25].

[29] QGroundControl. QGroundControl Website. URL http://qgroundcontrol.com/. [Accessed: 2019-01-25].

[30] Prakash Rashinkar, Peter Paterson, and Leena Singh. *System-on-a-chip verification: methodology and techniques*. Springer Science & Business Media, 2007.

[31] Thomas Reinbacher, Matthias Függer, and Jörg Brauer. Runtime verification of embedded real-time systems. *Formal methods in system design*, 44(3):203–239, 2014. URL https://link.springer.com/article/10.1007/s10703-013-0199-z.

[32] Julien Schmitt. GTKTerm GitHub Repository. URL github.com/zdavkeos/gtkterm.

[33] Johann Schumann, Patrick Moosbrugger, and Kristin Y Rozier. R2U2: Monitoring and diagnosis of security threats for unmanned aerial systems. In *Runtime Verification*, pages 233–249. Springer, 2015.

[34] Amazon Web Services. Amazon EC2 F1 Instances. URL https://aws.amazon.com/ec2/instance-types/f1. [Accessed: 2020-01-25].

[35] Dimitry Solet, Jean-Luc Béchennec, Mikaöl Briday, Sébastien Faucou, and Sébastien Pillement. Hardware runtime verification of embedded software in SoPC. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–6. IEEE, 2016.

[36] Joseph Allan Stamenkovich. Enhancing trust in autonomous systems without verifying software. Master's thesis, Virginia Tech, 2019.

[37] Simon Tatham. PuTTY Website. URL https://www.chiark.greenend.org.uk/~sgtatham/putty/.

[38] PX4 Dev Team. Pixhawk 4 Specifications. URL https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk4.html. [Accessed: 2019-07-15].

[39] Antti Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, 1996.

[40] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *IET Software*, 1:172–179, October 2007. ISSN 1751-8806. doi: 10.1049/iet-sen:20060076. URL https://digital-library.theiet.org/content/journals/10.1049/iet-sen_20060076.

[41] Xilinx, Inc. AXI UARTLite v2.0 Product Guide, 2017. URL https://www.xilinx.com/products/intellectual-property/axi_uartlite.html.

[42] Xilinx, Inc. Vivado Design Suite User Guide: High-Level Synthesis v2017.1, 2017. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-synthesis.pdf.

[43] Xilinx, Inc. Mailbox v2.1 Product Guide, 2018. URL https://www.xilinx.com/support/documentation/ip_documentation/mailbox/v2_1/pg114-mailbox.pdf.

[44] Xilinx, Inc. Vivado Design Suite, 2018. URL https://www.xilinx.com/products/design-tools/vivado.html.

[45] Xilinx, Inc. Zynq-7000 SoC Data Sheet v1.11.1, 2018. URL https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. [Accessed: 2019-01-25].

[46] Xilinx, Inc. ZCU104 Evaluation Kit, 2018. URL https://www.xilinx.com/products/boards-and-kits/zcu104.html#hardware.

[47] Xilinx, Inc. ZCU104 Evaluation Board User Guide v1.1, 2018. URL https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf.

[48] Xilinx, Inc. MicroBlaze Processor Reference Guide v2019.2, 2019. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug984-vivado-microblaze-ref.pdf. [Accessed: 2019-04-25].

[49] Xilinx, Inc. Vivado IP Integrator, 2019. URL https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0009-vivado-using-ip-integrator-hub.html.

[50] Xilinx, Inc. Zynq UltraScale+ MPSoC Data Sheet v1.8, 2019. URL https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[51] Xilinx, Inc. Zynq UltraScale+ Technical Reference Manual v2.1, 2019. URL https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf.

[52] Xilinx, Inc. Clocking Wizard v6.0 Product Guide, 2020. URL https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v6_0/pg065-clk-wiz.pdf.