Trusted Unmanned Aerial System Operations

Lakshman Theyyar Maalolan

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 Patterson, Chair

Michael Hsiao

Chang Woo Min

May 7, 2020

Blacksburg, Virginia

Keywords: Runtime verification, Safety monitors, FPGA, UAS, Formal methods

Copyright 2020, Lakshman Theyyar Maalolan

Trusted Unmanned Aerial System Operations

Lakshman Theyyar Maalolan

(ABSTRACT)

Proving the correctness of autonomous systems is challenged by the use of non-deterministic artificial intelligence algorithms and ever-increasing lines of code. While correctness is conventionally determined through analysis and testing, it is impossible to train and test the system for all possible scenarios or formally analyze millions of lines of code. This thesis describes an alternative method that monitors system behavior during runtime and executes a recovery action if any formally specified property is violated. Multiple parallel safety monitors synthesized from linear temporal logic (LTL) formulas capturing the correctness and liveness properties are implemented in isolated configurable hardware to avoid negative impacts on the system performance. Model checking applied to the final implementation establishes the correctness of the last line of defense against malicious attacks and software bugs. The first part of this thesis illustrates the monitor synthesis flow with rules defining a three-dimensional cage for a commercial-off-the-shelf drone and demonstrates the effectiveness of the monitoring system in enforcing strict behaviors. The second part of this work defines safety monitors to provide assurances for a virtual autonomous flight beyond visual line of sight. Distinct sets of monitors are called into action during different flight phases to monitor flight plan conformance, stability, and airborne collision avoidance. A wireless interface supported by the proposed architecture enables the configuration of monitors, thereby eliminating the need to reprogram the FPGA for every flight. Overall, the goal is to increase trust in autonomous systems as demonstrated with two common drone operations.

Trusted Unmanned Aerial System Operations

Lakshman Theyyar Maalolan

(GENERAL AUDIENCE ABSTRACT)

Software code in autonomous systems, like cars, drones, and robots, keeps growing not just in length, but also in complexity. The use of machine learning and artificial intelligence algorithms to make decisions could result in unexpected behaviors when encountering completely new situations. Traditional methods of verifying software encounter difficulties while establishing the absolute correctness of autonomous systems. An alternative to proving correctness is to enforce correct behaviors during execution. The system's inputs and outputs are monitored to ensure adherence to formally stated rules. These monitors, automatically generated from rules specified as mathematical formulas, are isolated from the rest of the system and do not affect the system performance. The first part of this work demonstrates the feasibility of the approach by adding monitors to impose a virtual cage on a commercially available drone. The second phase of this work extends the idea to a simulated autonomous flight with a predefined set of points that the drone must pass through. These points along with the necessary parameters for the monitors can be uploaded over Bluetooth. The position, speed, and distance to nearby obstacles are independently monitored and a recovery action is executed if any rule is violated. Since the monitors do not assume anything about the source of the violations, they are effective against malicious attacks, software bugs, and sensor failures. Overall, the goal is to increase confidence in autonomous systems operations.

Acknowledgments

I express my sincere gratitude to Dr. Cameron Patterson for the opportunity to work on this research project, for letting me explore new ideas, and his mentorship, guidance, and support through my graduate career. I have grown to develop a work ethic that includes his pursuit of perfection, which I hope to carry with me forever.

I would like to thank Dr. Michael Hsiao and Dr. Chang Woo Min for graciously accepting to be a part of my advisory committee. The skills that I picked up early on in their courses were valuable not only for my research work but will remain so for the rest of my scientific career. I acknowledge my teammates, Joseph Stamenkovich and Akhil Rafeeq for their enthusiastic collaboration and peer support. I extend my appreciation to Jerry Lopez, Heber Herencia-zapana, and Paul Meng of GE Research and Glen Gallagher of GE Aviation for their involvement and contribution to this project.

I am eternally grateful to my parents and sister for accepting my decision to move far and pursue a graduate career. Their constant support has been instrumental in the successful completion of my Master's degree. I also thank my friends, Nandhitha Venkatesh, Shashank Raghuraman, and Subramanian Mahadevan for being my rock of moral support, particularly during difficult times. I also thank Madhava Krishnan and Sudharsan Sadagopan for making life in Blacksburg memorable. The intellectual inputs and emotional involvement from family and friends have been vital in my decision-making process.

This work has been funded 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 x

List of Tables xiii

1 Introduction 1

1.1 Contributions . . . 2

1.1.1 First Phase Implementation . . . 3

1.1.2 Second Phase Implementation . . . 4

1.2 Thesis Organization . . . 5

2 Background 6

2.1 Design-related Background . . . 6

2.1.1 Verification . . . 6

2.1.2 Target Platform . . . 8

2.2 Unmanned Aerial System . . . 10

2.2.1 Ground Control Station . . . 10

2.2.2 Pixhawk Flight Controller . . . 10

2.2.3 Flight Plan . . . 12

2.2.4 Micro Air Vehicle Link . . . 13

2.2.5 Gazebo Simulator 13

2.2.6 UAV Software Stack 13

3 High-level Design **15**

3.1 Abstract Monitor Architecture 15

3.2 Design Choices 18

3.2.1 Hardware Monitors 19

3.2.2 Monitor Complexity 20

3.3 Monitor Synthesis Flow 21

3.4 Related Work 22

3.4.1 Runtime Verification Techniques 22

3.4.2 Hardware Monitors 23

3.4.3 Other Related Work 23

4 First Phase—Targeting a Commercially Available UAS **25**

4.1 Virtual Cage 26

4.2 Monitor Synthesis 27

4.2.1 Informal Requirements 27

4.2.2 Requirements in LTL 27

4.2.3 LTL Specifications to Abstract Automata 29

4.2.4 Abstract Automata to C Code 30

4.2.5 C Code To Parallel Hardware 32
4.3 Monitor Implementation Analysis 34
4.4 Implementation in the Intel Aero 36
4.4.1 Pedigreed Component Architecture 38
4.4.2 Monitor Block Integration 41
4.4.3 Resource Utilization and Performance Analysis 45
4.4.4 Virtual Cage Flight Test Results 46

5 Monitoring an Autonomous Flight Beyond Visual Line of Sight 49

5.1 Motivation 49
5.2 Architecture 51
5.2.1 Design Choice 52
5.3 Hardware-in-the-loop Simulation 53
5.4 User Interface Processor 56
5.4.1 Monitors Configuration Flow 56
5.4.2 User Interface Processor as a Master 58
5.5 I/O Processor 60
5.6 Monitor Processor 61
5.6.1 Identifying Phases of Flight 62
5.6.2 Monitor Blocks Integration 65

5.7 Obstacle Detection and Collision Avoidance 66
5.7.1 Simulating the Obstacle 66
5.7.2 Simulating Collision Avoidance 67
5.8 Inter-processor Communication 70
5.8.1 User Interface Processor as a Server 70
5.8.2 Mailbox – the Bridge between the Processors 72

6 Safety Monitors 74

6.1 Takeoff Monitors 75
6.1.1 Takeoff Position Monitors 75
6.1.2 Takeoff Speed Monitors 77
6.2 Runtime Monitors 80
6.2.1 Runtime Position Monitor 81
6.2.2 Runtime Speed Monitors 83
6.2.3 Altitude Monitors 90
6.2.4 Collision Avoidance Monitor 96
6.3 Land Monitors 97
6.3.1 Land Position Monitors 99
6.3.2 Land Speed Monitors 99

7 Evaluation 100

7.1 Software Optimization 100

7.2 Virtual Flight Tests 102

7.2.1 Runtime Speed Violation 103

7.2.2 Runtime Position Violation 104

7.2.3 Collision Avoidance Violation 105

7.3 Timing Analysis 106

7.4 Scalability Analysis 109

7.5 Summary 113

8 Conclusion **114**

8.1 Future Work 115

Bibliography **116**

Appendices **125**

Appendix A Flight Plan in JSON format **126**

List of Figures

2.1 Verification techniques . . . 7

2.2 FPGA structure example . . . 9

2.3 QGroundControl interface . . . 11

2.4 Pixhawk 4 flight controller . . . 11

2.5 Example flight plan . . . 12

3.1 Adding safety monitors to a UAS . . . 16

3.2 Monitor event timing . . . 17

3.3 Monitor synthesis flow . . . 21

4.1 Virtual cage limits . . . 26

4.2 Atomic proposition values arising from different flight scenarios . . . 29

4.3 Büchi automaton for the *within_cage_max_alt* specification . . . 30

4.4 Monitor wrapper registers . . . 34

4.5 Monitor analysis process . . . 35

4.6 Intel Aero compute board before adding the monitors . . . 37

4.7 Intel Aero drone with GPS unit mounted on the top . . . 37

4.8 Aero compute board after adding monitors . . . 38

4.9 MAX 10 FPGA hardware resource types 39

4.10 I/O soft processor and monitor blocks 40

4.11 Monitor wrapper registers 43

4.12 Processor interface to the monitor wrapper 44

4.13 Triggering the land RCF 47

5.1 HITL simulation architecture 51

5.2 RTA design architecture 52

5.3 HITL simulation 54

5.4 HITL simulation with RTA system in place 55

5.5 Flow of steps involved in configuring monitors 56

5.6 UAV cruising through its last leg 62

5.7 Identifying cruise and land flight phases 64

5.8 Template of a monitor 65

5.9 Commands sent from QGC 68

5.10 Virtual cage imposed on the drone 69

5.11 Change in flight plan to avoid an obstacle 70

5.12 Client-server interaction between UIP and MPs 71

5.13 Mailbox between processors 73

6.1 Safe takeoff zone 76

6.2 Flying corridor 81
6.3 Different flight paths with same ground speed 85
6.4 Calculation of speed 85
6.5 Change in direction during a flight 88
6.6 Triangle formed by waypoints A, B, and C 89
6.7 No change in altitude during runtime 91
6.8 Increase in altitude during runtime 92
6.9 Calculation of safe altitude zone during climb 94
6.10 Decrease in altitude during flight 94
6.11 Calculation of safe altitude zone during descent 95
6.12 Different scenarios leading to different AP values 98

7.1 A successful flight 102
7.2 Runtime under speed violation 103
7.3 Runtime position violation 104
7.4 Collision avoidance violation 105
7.5 Single Monitor Processor System 110
7.6 Dual Monitor Processor System 110

List of Tables

4.1 SystemVerilog Assertions applied to the *within_cage_max_alt* monitor's hardware implementation 36

4.2 Resource utilization for the MAX 10M08 FPGA 45

4.3 Average execution time of software and hardware modules 46

6.1 Velocity fields in the GPS packet 84

7.1 Initialization time 106

7.2 Execution time of different sets of monitors 107

7.3 Time taken for MP to report monitor violations to IOP 108

7.4 Initialization time after doubling the number of monitors 111

7.5 Execution time of different sets of monitors after doubling the number of monitors 111

7.6 Comparison of time taken to report a violation 112

7.7 Inter-processor communication time 112

7.8 Resource utilization 113

Chapter 1

Introduction

Autonomous vehicles have gained immense popularity over the last decade with multi-billion dollar corporations and research groups across the globe. Amazon conducted its first successful delivery through Prime Air in December 2016 [34] and has been testing its Scout delivery robots in two cities in the US [4]. Google Wing, in collaboration with Mid Atlantic Aviation Partnership (MAAP), has been using drones to deliver goods in southwest Virginia [16].

But, where are these delivery drones when we really need them? [42]. In the middle of a global health pandemic, when humans are requested to maintain social distance, drones and last-mile delivery robots would have been the ideal choice for delivering food supplies and medicines. Though numerous trials have been conducted, these autonomous vehicles are far from being omnipresent in society since they have not gained the complete trust of humans. This is because there are several instances of these systems causing disruptions to humans and wildlife. Rogue drones affecting airport operations [14, 27, 39] and crashing into structures [8, 11] are common examples.

Software code that controls and makes decisions in these systems is prone to bugs, and vulnerabilities that can be exploited by malicious attackers. The failure of physical sensors results in incorrect information being passed to the control software, which will lead to undesirable actions. These insecurities create a need for the highest degree of functional verification. The components in aircraft go through different levels of verification process

before being certified for use. However, the software code in these components are developed by closely following the guidelines released by aviation authorities. This approach is not feasible for most autonomous systems due to the open source nature of the development of control software and the use of non-deterministic machine learning and artificial intelligence algorithms. No amount of training can satisfy the need to make them perfect. The use of complex algorithms and an evolving code base makes it difficult for formal methods to ensure the absence of undesirable behaviors. This task is made daunting by the fact that formal methods need to be applied to every version of the software. It is insufficient to prove one version of the code is functionally correct as future updates could invalidate that claim.

The alternative is using runtime verification techniques to actively monitor the system. Linear temporal logic (LTL) formulas capture the expected behaviors of the systems, which are rules that should not be violated. LTL is preferred over propositional logic because it incorporates the notion of time with operators that refer to future states. Generally, runtime verification is used during development, and the added software or hardware code gets removed prior to deployment because of concerns about overheads. This work proposes ways to retain the safety monitors during deployment to guard against malicious attacks and software bugs without affecting performance. Field Programmable Gate Arrays (FPGAs) are chosen as the target to implement monitors. The isolation provided by FPGAs prevents vulnerabilities in software from compromising the monitoring system. Parallel execution of independent monitors in the configurable hardware greatly reduces the latency introduced.

1.1 Contributions

An unmanned aerial system (UAS) is chosen as the target system for this research. This work aims to increase the trustworthiness of two common categories of drone operations

by deploying safety monitors to prevent unsafe actions. The first category consists of UAS flights within a confined space and the second category comprises autonomous flights beyond visual line of sight (BVLOS) with a predefined flight plan . The former includes activities like drone photography and recreational flights. Drone delivery and pipeline patrols are examples of the latter.

1.1.1 First Phase Implementation

The first phase of this work investigates the feasibility of the approach by imposing a three-dimensional cage on a commercially available Intel Aero drone. The Aero has a nearly unused FPGA through which sensor data and commands pass through. Monitors are implemented in this FPGA to have access to the system's inputs and outputs. The rules that define the cage are captured as LTL formulas. A tool flow is created that translates LTL formulas to automata to C code to a hardware description language (HDL). Model checking is applied to the HDL code to ensure the monitors satisfy formally stated behaviors and to verify whether the LTL specifications are correctly stated. The author's specific contributions are:

1. Capturing rules as LTL formulas and synthesizing hardware monitors from them.
2. Verifying the correctness of monitors by applying model checking to the final hardware implementation.
3. Hardware/software co-design consisting of integrating the memory-mapped monitors with a soft processor.

Integration with other components of the UAS, sensor data processing, and invoking a recovery control function were implemented by others.

1.1.2 Second Phase Implementation

The second phase of this work demonstrates the effectiveness of monitors in enforcing correct behaviors for an autonomous flight BVLOS in a simulated environment. A slight modification is made to a regular hardware-in-the-loop (HITL) simulation setup by the introduction of an FPGA. The commands and messages exchanged between the simulation environment and flight control computer are routed through the FPGA, which hosts the monitors. This addition does not require any change to the software stack. A highly scalable architecture is proposed to support a large number of monitors, wireless interface to a trusted device and HITL simulation. These tasks are handled by dedicated processors interacting with each other. Monitors can be configured wirelessly using a convenient process before every flight. Multiple independent monitors ensure that the UAS strictly follows the filed flight plan, the flight is stable and collision with any nearby air vehicles is avoided. The author's contributions are:

1. Design of the multiprocessor architecture that supports a large number of monitors and multiple interfaces to the external world.
2. Inclusion of a real-time processor to support the monitor configuration flow
   (a) A Bluetooth interface to support the serial transmission of a file that contains the flight plan and monitor parameters.
   (b) The ability to store the file in flash memory for future use.
   (c) A parser script to extract waypoints and parameters from the file.
3. Processors
   (a) A soft processor to invoke the correct set of monitors for the given flight phase. The current flight phase is identified without relying on commands from the

autopilot.

(b) A simulation of an obstacle detection sensor in an application processor.

(c) Synchronization among processors and inter-processor communication.

4. Safety monitors

   (a) Defined liveness and correctness properties for different flight phases and captured as separate LTL formulas.

   (b) Synthesized hardware monitors from LTL formulas and applied model checking.

   (c) Implemented deterministic algorithms utilized by the monitors.

   (d) Integration of the memory-mapped monitors with a soft processor.

Others implemented the interfaces for HITL simulation, encoding and decoding inter-processor packets, and recovery control function invocation. The Bluetooth interface implementation was done in collaboration with Akhil Ahmed Rafeeq.

1.2 Thesis Organization

The rest of the thesis follows the following structure. Chapter 2 introduces concepts related to UAS and various target platforms available. A high-level overview of the solution and design decisions are presented in Chapter 3. Chapter 4 discusses the synthesis flow, results of hardware model checking and the first phase implementation. The second phase is captured in the next three chapters: Chapter 5 delineates the improved runtime assurance (RTA) architecture for monitoring autonomous flight, Chapter 6 describes the safety monitors, and Chapter 7 presents the virtual flight test results and evaluates the design. Finally, Chapter 8 offers conclusions.

Chapter 2

Background

This chapter provides background on concepts and techniques used in this project. Section 2.1 discusses various choices available for the target platform and different forms of verification. Section 2.2 focuses on UAV related topics useful for understanding the rest of the thesis.

2.1 Design-related Background

2.1.1 Verification

Verification is the act of ensuring that the implementation is correct, and the system behaves as expected. There are different forms of verification, as shown in Figure 2.1. In a commercial setting, testing and static analysis are common methods of verification. Static analysis refers to examining the source code without executing it [70]. It is also known as inspection and is usually done as a preliminary step. CppDepend and HCL AppScan are examples of static analysis tools [10, 18]. Testing can take two forms—automated test procedure or demonstration. Demonstration involves operating/executing the system and ensuring that it functions as expected. Automated test procedure usually involves executing test scripts to simulate different scenarios and ensuring that the system outputs match the outputs specified in the requirements.

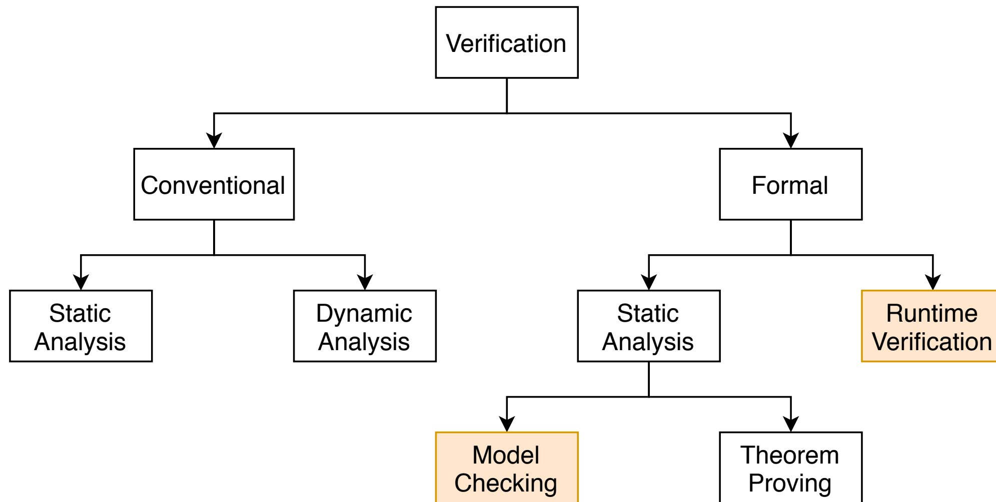


Figure 2.1: Verification techniques

Theorem proving involves providing mathematical proof of a particular system property or behavior, and it can be used for hardware verification [60]. Some consider it to be the highest form of assurance, but it is hard to achieve. Model checking, another form of verification, is a static formal analysis method which uses a model of the system and a set of formally defined behaviors, and methodically proves (or disproves) that the system adheres to the rules. Model checking [59] can ensure correct behavior across all possible executions, unlike conventional testing, in which only a handful of scenarios are tested. Model checking explores all possible states of the model. Therefore, if the number of states is too high, it can result in a state explosion [58]. Runtime verification techniques are used to ensure that the system adheres to formally stated correct behaviors when it is active. A piece of software code or hardware is added to monitor the state of the active system and capture any violation from a predefined set of rules. The monitor code is removed before deployment to reduce overheads. A simple example of runtime verification is adding *assert* statements in software or HDL code. The pioneers of runtime verification started a company under the same name [38].

The runtime verification and model checking techniques are used in this thesis. An RTA system is added to the UAV to detect any violation from expected behavior. Model checking is used to prove the correctness of the safety monitors.

2.1.2 Target Platform

There are several platforms available in the real-time embedded world for the implementation of any new system. A microcontroller is an integrated circuit (IC) capable of executing instructions to perform a specific task or application. It usually operates between 1 MHz and 200 MHz, comprises a central processing unit, random access memory (RAM), I/O ports, and non-volatile memory, and executes instructions in a sequential manner.

Field Programmable Gate Arrays (FPGA) are a type of ICs that can be configured or reprogrammed by the user [41]. Generally, ICs are designed for specific purposes and cannot be modified after the manufacturing process. In contrast, FPGAs are designed to be reprogrammable, and thus, suit a variety of applications. They are also known as the Programmable Logic.

An FPGA consists of blocks of programmable logic units, memory, digital signal processing units, and programmable I/O peripherals, as shown in Figure 2.2. The interconnection between these blocks can also be programmed. An assortment of memory block sizes is available. The I/O pins provide the interface to the external world. These pins are connected to the I/O ports on the board like HDMI, Ethernet, and peripheral module (Pmod) interface. In addition to creating hardware, FPGAs can be used to create soft processors or softcore processors. The logic resources are programmed to form an instruction set processor with dedicated instruction and data memories. The soft processors do not have the processing speed of the modern-day hard-silicon processors. However, they are closer to the hardware

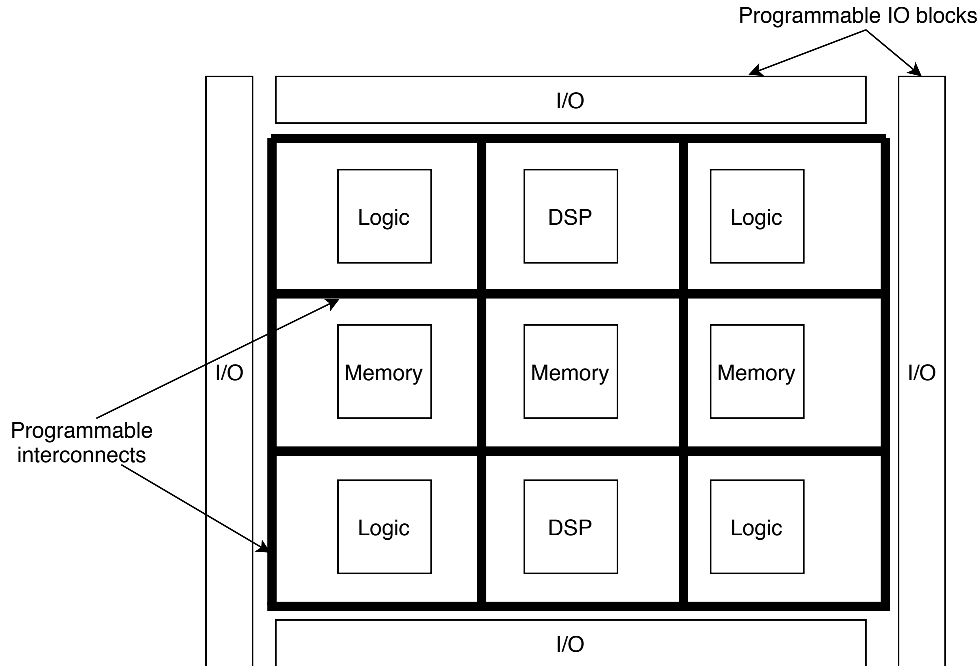


Figure 2.2: FPGA structure example

blocks, and thus, provide faster and easier interaction. Multiple soft processors can be created to perform dedicated tasks and interact with different hardware blocks. A set of task-specific soft processors provide a useful and simpler alternative to multiplexing these tasks on a single set of processor/ memory/ bus resources.

A system-on-chip (SOC) refers to a system that encompasses one or more microcontrollers (or microprocessors), an FPGA, memory, and other dedicated processing units like GPU, which are interconnected. The microcontrollers can be used for I/O operations with the outside world. FPGA and other processing units can be used for performing specific tasks. A Xilinx UltraScale+ MPSoC [26] was used in the second phase of the project.

2.2 Unmanned Aerial System

An unmanned aerial vehicle (UAV), commonly referred to as a drone, is an autonomous flying vehicle without a human pilot onboard. It is powered by a battery and equipped with multiple sensors, actuators, a flight controller, and a camera. It can be controlled by a remote pilot using a remote controller, or it can fly autonomously. The following subsections brief about the software and hardware components that form the UAS.

2.2.1 Ground Control Station

A ground control station (GCS) is a base station at the ground level used to control and manage UAVs. It is usually a portable computer that executes GCS software, which can be used to plan and file a mission. This software displays the flight information received through telemetry. Figure 2.3 shows a typical graphical user interface of QGroundControl (QGC) software which is used in this work [36]. A UAS encompasses the UAV, GCS, and the communication link between them (generally radio).

2.2.2 Pixhawk Flight Controller

A flight controller is the heart of the UAV. It receives sensor inputs, processes the information, and sends commands to the actuators (or motors controlling the propellers). It is also responsible for producing physical actions from control commands. Pixhawk is an open-source hardware community aimed at developing the standards for reliable and affordable autopilot hardware [30]. Pixhawk has multiple autopilot hardware which support both ArduPilot and PX4 flight stacks [5, 35]. Pixhawk 4, shown in Figure 2.4, is a flight controller optimized for PX4 version 1.7.

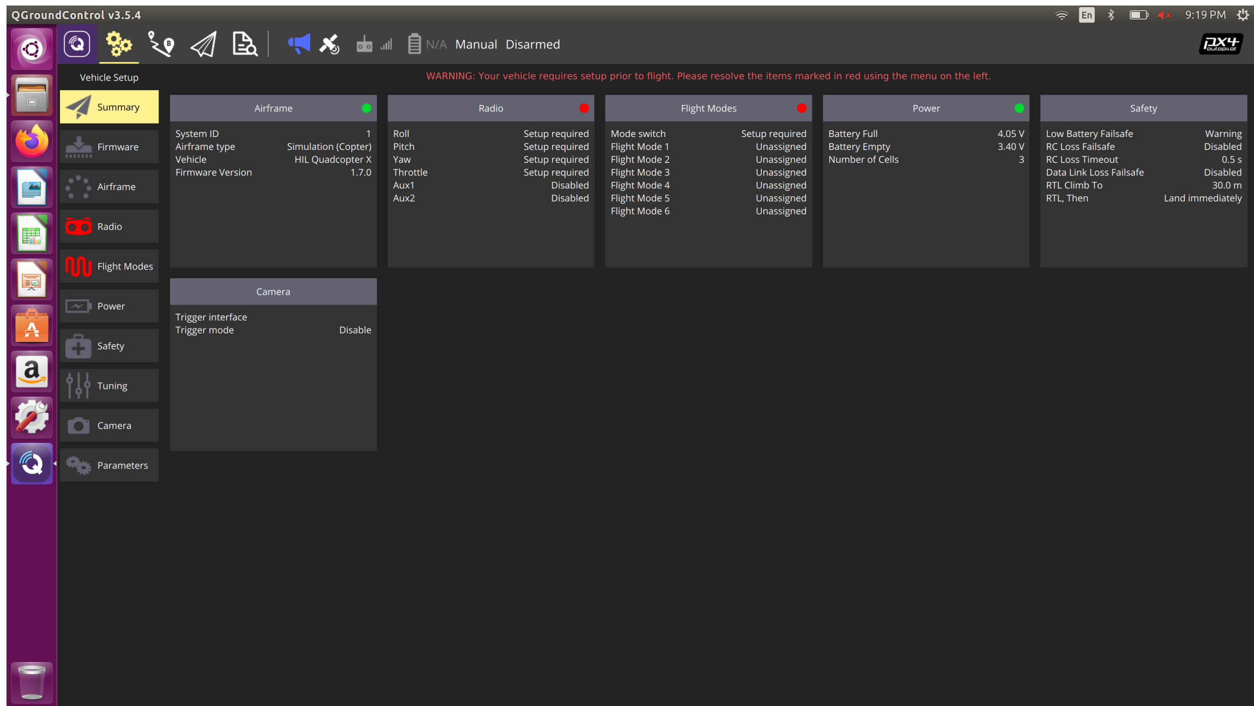


Figure 2.3: QGroundControl interface

Figure 2.4: Pixhawk 4 flight controller

2.2.3 Flight Plan

Flight plans are documents that indicate the planned path for a UAS. The path consists of one or more waypoints, which are points in the 3D space (latitude, longitude, and altitude). The UAS is expected to travel from one waypoint to the next in a straight line. QGC is one of the several applications that can be used to create flight plans.

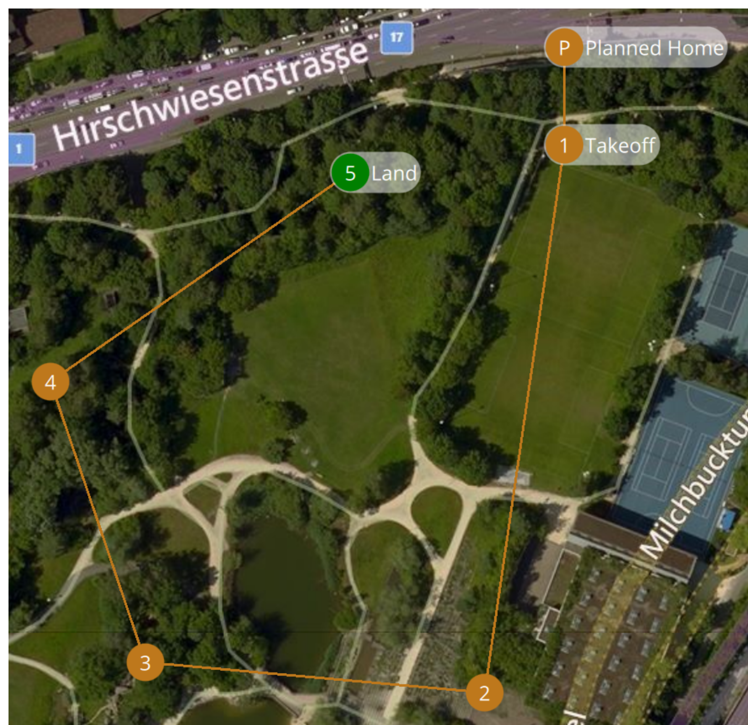


Figure 2.5: Example flight plan

Figure 2.5 shows a pictorial representation of a flight plan created in QGC. The waypoints are numbered 1 to 5. Note that `1` is the takeoff point, and `5` is the land point. The flight plan can also be downloaded in JSON format [21]. Appendix A gives an example of a JSON-formatted flight plan and the descriptions of the fields can be found in [31].

For our work, flight plans downloaded from the QGC application were directly used. In the future, we expect a UAS operator to be able to submit the plan to an authority for ap-

proval. NASA is working on one such air traffic management authority known as UAS traffic management (UTM), which focuses on enabling UAS operations in low altitude airspace [68].

2.2.4 Micro Air Vehicle Link

The Micro Air Vehicle Link (MAVLink) [23] protocol is used for communication between the GCS and UAV, as well as between different components in the UAV. Messages and commands are packed in the packet format specified in the MAVLink developer guide [29]. The receiver sends back an acknowledgment to the sender for every packet.

2.2.5 Gazebo Simulator

Gazebo [19] is a simulation environment for autonomous robots. It can be used for software-in-the-loop (SITL) and HITL simulation of drones [15]. The SITL is used for developing and testing software algorithms. In contrast, the HITL is used during the development of a hardware component. In HITL, MAVLink packets are exchanged between the simulator and the hardware unit under test.

2.2.6 UAV Software Stack

Autonomous systems consist of several applications that collectively guide them to their next state. In UAVs, these applications include machine learning-based navigation algorithms, autopilot software, real-time environment mapping, and obstacle avoidance. These are termed as complex applications or functions because it is hard to predict their outcome as well as prove their correctness. Present-day UAVs have a multicore processing system onboard to execute several applications. Artificial intelligence and machine learning algorithms have

found increasing use in autonomous systems. It is desirable for the UAV to learn from its previous flights and make better decisions when it encounters a similar situation next time. However, this leads to unpredictability and non-determinism; that is, the UAV can produce different outputs for the same set of inputs. This makes it harder to verify the correctness of the system. Another concerning question is, how will the system react to a situation it has not been trained for?

The runtime monitoring system described in this thesis acts as a complement to the complex applications. It is deterministic and verified using model checking. It overrides the commands from the complex functions when there is a violation of expected behavior.

Chapter 3

High-level Design

This chapter captures the high-level design for the safety assurance system. Specific implementation details are discussed in later chapters. The design is common to both the project phases described in this thesis.

3.1 Abstract Monitor Architecture

ASTM F3269-17 defines best practices to *Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions* [53]. The primary goal of the document is to recommend a runtime assurance (RTA) architecture to verify complex functions implemented on a UAS that cannot be assured through traditional methods. There is a distinction between *non-pedigreed* and *pedigreed* components. A non-pedigreed component is a software or hardware that has not been or cannot be proven to always exhibit a defined behavior with an acceptable level of certainty. On the other hand, a pedigreed component can be proven to always follow a prescribed behavior to the required level of certainty. Using isolated and pedigreed safety monitors to confirm the integrity of non-pedigreed functions has the advantage of not constraining how the non-pedigreed components are implemented.

Figure 3.1 shows how the pedigreed monitors provide an external layer of oversight for the complex functions directing the flight controller. Non-pedigreed components transmit flight functions through the isolated component to the flight controller. If sensor data indicates

a deviation from correct and safe behavior, the monitors activate the switch that routes the recovery control function (RCF) to the flight controller rather than the normal flight function source. The sensor data source and flight controller are required to be pedigreed through the imposition of other standards and practices. RCFs may vary from situation to situation, and several may exist on a particular system. A priority system is required if multiple RCFs may be triggered simultaneously,

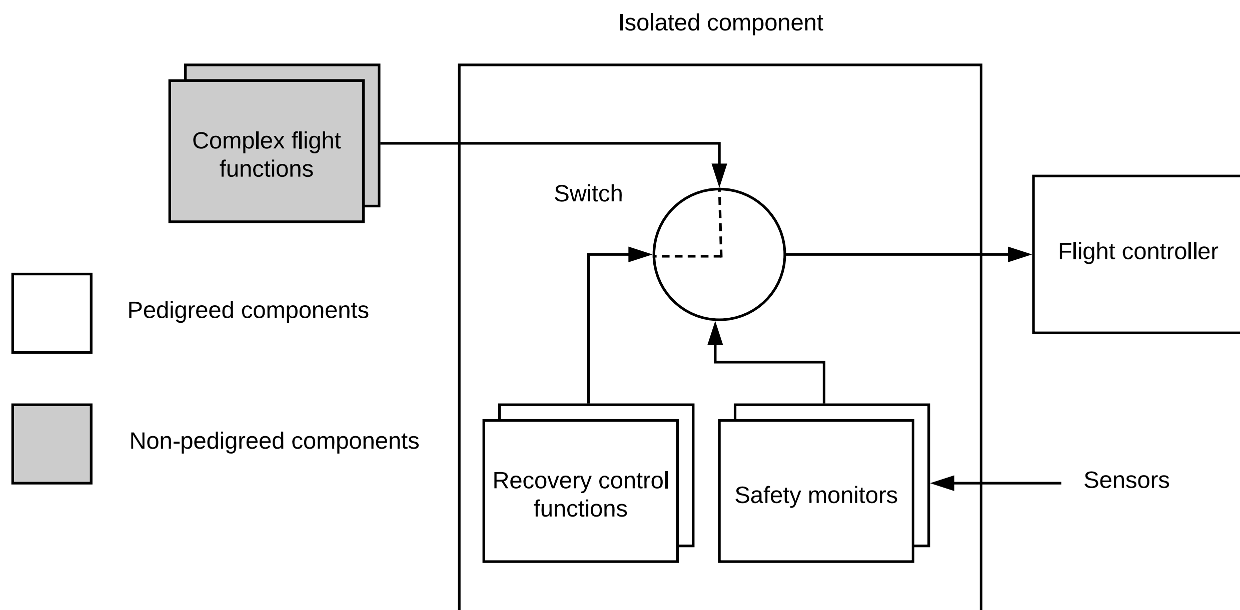


Figure 3.1: Adding safety monitors to a UAS

Figure 3.2 illustrates the monitor and RCF timing events. The RTA safety monitor cycle begins with a new set of UAS sensor data. Some steps are required regardless of monitor outputs, with additional time required if an RCF is invoked. Concurrent execution can reduce the execution time as the number of monitors is increased so that all processing is completed before new sensor data are available. For data required by both the monitors and other UAS components, the sensor input communication channel can be shared instead of passing through the monitor system in order to reduce control loop latency.

The RTA must fulfill a variety of requirements:

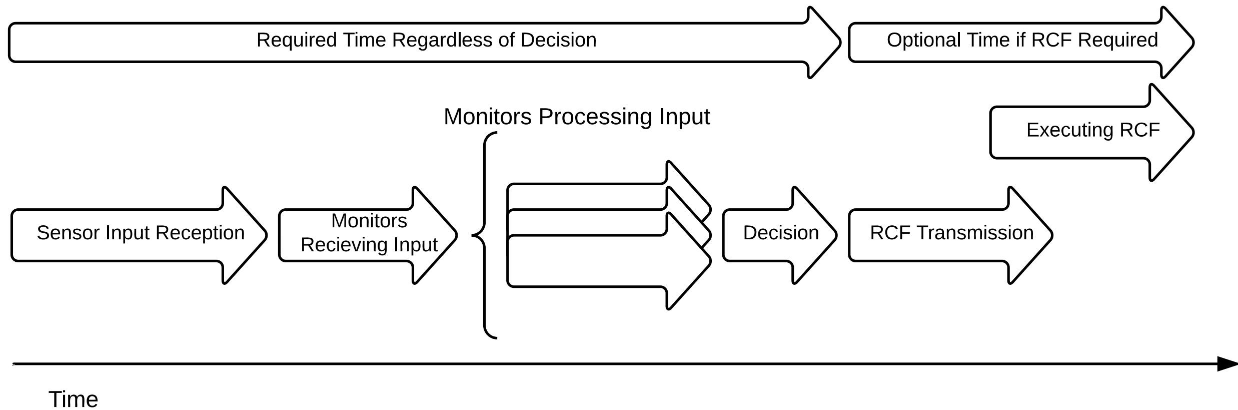


Figure 3.2: Monitor event timing

1. A priority-based RCF hierarchy needs to be established.

2. Complex functions must be separate from the safety monitors, RCFs, and the switch.

3. An operational risk assessment must determine the likelihood and severity of the failures leading to an RCF.

4. An initialization protocol is needed to verify that individual components of the RTA safety monitor architecture are functioning properly.

5. The sensor types, performance, accuracy, precision, and frequency required by the safety monitors need to be determined.

6. Safety monitors must be able to cope with the failures of these inputs.

7. Complex functions must send the vehicle control commands to the RTA switch at the frequency needed for proper vehicle management.

8. RCFs must be designed to constrain the vehicle within predetermined limits.

9. The RTA switch must allow a continuous flow of commands to the vehicle management system even while switching between complex functions and RCFs.

10. The safety monitor must invoke the RCF in time to keep the vehicle operating within predetermined limits.

11. Each RCF may be enforced temporarily or until the vehicle completes a mission or mission phase.

12. Safety monitors must consider the confidence and quality of the sensor data when making a decision.

In summary, monitors and RCFs need isolation from the non-pedigreed components, very low latency, robustness, and correctness assurance.

3.2 Design Choices

Several choices had to be made throughout the design and implementation of the RTA system. Two major design choices common to the overall design are discussed in the following two subsections. The choices specific to the implementation are discussed in later chapters. All the design choices meet the following constraints:

1. **Correct:** The monitoring system should be correct. It should detect any violation and interfere only when there is a violation of expected behavior.

2. **Strictly additive:** The RTA system should be a separate component that can be added to an existing system without requiring changes to other components.

3. **Isolated:** The RTA system should be isolated from other components.

4. **Performance:** The RTA system should add minimal latency so that it does not affect the performance of the UAV.

3.2.1 Hardware Monitors

Runtime verification, as discussed earlier, is conventionally performed during development where the monitors are additional software code that gets removed during deployment. However, in our approach, the runtime monitors (or the RTA system) remain deployed during production. Monitors implemented in hardware meet the constraints as follows:

1. *Correct*: Hardware is more suitable for formal verification. Several commercial and free (educational) hardware model checking tools can verify the correctness of the hardware monitors. Also, hardware model checking is usually more tractable than software model checking.

2. *Strictly additive*: Implementing the RTA system as separate hardware eliminates the need to make modifications to exiting components and application software. The only change required is to route sensor data through the added component.

3. *Isolated*: Software bugs and malicious attacks on the application software should not compromise the monitoring system. Configurable hardware implemented in an FPGA provides this required isolation from application software and the operating system.

4. *Performance*: It is easier to optimize the performance in hardware. Good knowledge of the OS and intricacies of the language is required to optimize software code, whereas the hardware development tools handle the optimization while implementing in an FPGA, which provides more opportunities for parallelism. Xilinx high-level synthesis (HLS) tools can be used to generate hardware blocks from software code. This tool produces different area/speed tradeoffs for the same software code to meet different timing requirements.

5. *Latency*: A key performance metric in any real-time system is latency, which is the time taken for a piece of software code or a hardware block to complete its execution.

It is critical for the latency of the runtime verification system to be minimized. Having multiple parallel monitors in hardware adds less latency than sequentially invoked software monitors. Hardware is inherently parallel, and thus it is straightforward to implement blocks of hardware in FPGA that execute in parallel.

3.2.2 Monitor Complexity

The critical step in the synthesis of monitors lies in the very beginning—capturing the requirements for the monitors as LTL formulas. The output of subsequent steps and the complexity and correctness of the final implementation in hardware depends on the LTL formula. Thus, capturing the requirements needs the utmost attention to detail. Multiple small monitors are better than a single large monitor that tries to enforce multiple behaviors. Several problems arise when developing a single complex monitor:

1. Capturing the requirements for multiple monitors as a single LTL formula is difficult. It is easy to make mistakes even while coming up with a simple formula, let alone complex ones.

2. Readability and self-documentation are important while capturing requirements. Even if all the requirements were captured correctly as a single formula, the formula would end up being complex and challenging to understand.

3. The automaton becomes overly complex, and it is hard to track all the transitions.

4. Monitors are necessarily state machines. Applying model checking to a hardware model that has a large number of states is time-consuming and risks state explosion.

Therefore, multiple small, simple, and independent monitors were implemented. This facilitates concurrent execution and incurs less latency than a complex monitor.

3.3 Monitor Synthesis Flow

Generally, the logic for the hardware is handwritten in an HDL – like VHDL or Verilog. This HDL gets converted into the bitstream by the FPGA tools, which gets programmed into the configurable hardware. The manual step in this approach is capturing the logic in an HDL. This thesis describes a flow to remove this manual step and entirely automates the synthesis of hardware monitors. The two significant advantages are:

1. **Speed:** The described flow is faster than manually capturing the logic in HDL.

2. **Fewer errors:** Minimal hand-coding of HDL reduces the potential for oversights.

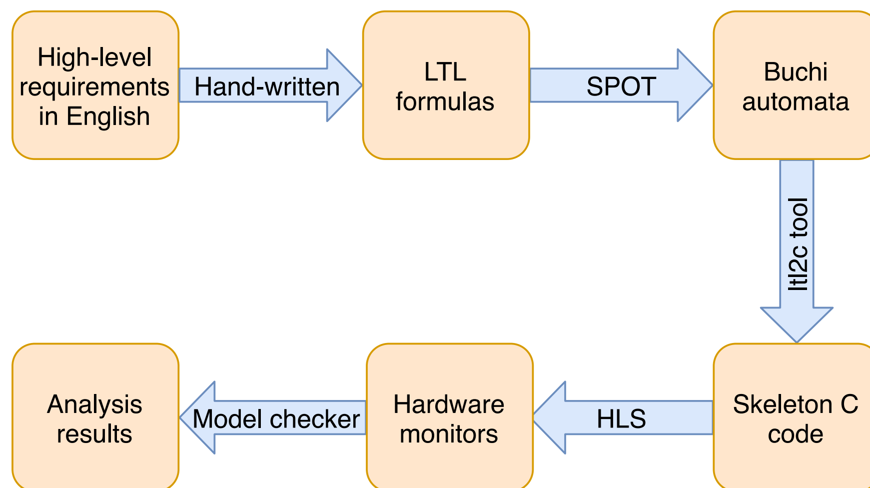


Figure 3.3: Monitor synthesis flow

The flow of steps involved in synthesizing hardware monitors, as illustrated in Figure 3.3, are:

1. Capture the high-level requirements for the monitors informally in a language like English.

2. Capture the requirements formally in LTL.

3. Convert the LTL formulas to Büchi automata [67] using the EPITA SPOT tool [62].
4. Convert the textual representation of the automata into C code using a script.
5. Convert the C code to HDL using the HLS tool.
6. Analyze the hardware monitors using a model checking tool.

With the help of an example, Section 4.2 explains the steps and concepts involved in the synthesis of hardware monitors.

3.4 Related Work

This section is based on a published, co-authored paper [78].

3.4.1 Runtime Verification Techniques

The use of runtime verification to enhance the correctness and safety of real-time embedded systems has been investigated by many researchers. Bartocci et al. provide a comprehensive overview of runtime verification covering different specification languages, types of monitors, instrumentation techniques, and monitorability [54]. Multiple runtime enforcement and healing techniques and ways to use them to detect, prevent and react to failures are discussed in [63]. The generation of runtime verification monitors from LTL and timed LTL (TLTL) specifications is extensively studied by Bauer et al. where they also demonstrate the feasibility of their methodology [56]. Instrumentation conveys necessary information about the monitored system to the monitors. Cassar et al. discuss instrumentation techniques—broadly classified as online and offline instrumentation—in detail and analyze the existing monitoring tools [57].

3.4.2 Hardware Monitors

Implementing monitors in FPGA hardware is a technique previously adopted by several others [65, 72, 73, 75, 77]. Pellizzoni et al. proposed a framework, BusMOP, used to synthesize hardware monitors. Monitors are synthesized into an FPGA and the device is plugged into a PCI bus to monitor transactions between different COTS peripherals [73]. In [77], formal specifications expressed in past-time LTL (pLTL) formulas are synthesized into hardware monitors used for runtime verification of embedded software in a system-on-programmable-chip which is instrumented to send information to the hardware monitor. A procedure for the synthesis of hardware monitors from signal temporal logic (STL) assertions is presented in [65], and the viability of the approach for mixed and digital signal systems is demonstrated with examples. Nguyen et al. put forth a framework for synthesizing hardware monitors from STL assertions for assessing correctness and robustness of automotive systems-of-systems emulated on hardware. Similar to our approach, the monitors are synthesized using HLS [72]. In [66], the authors discuss a monitoring algorithm, EgMon, used to monitor an autonomous research vehicle (ARV) comprised of COTS components by passively observing the system's broadcast buses. The monitor is implemented on an ARM development board and evaluated against the logs obtained from the robustness test of their ARV. The utility of runtime monitoring at different phases of electronic product development in the automotive industry is well-depicted in [76].

3.4.3 Other Related Work

UAS security and safety are active and critical areas of research. R2U2 is a monitoring framework which can detect and diagnose security threats to UASes but cannot prevent or mitigate attacks [75]. The hardware monitor runs its check on the GPS, ground control

inputs, and critical data obtained from the flight software (sensor data, actuator data and flight software status). The trustworthiness of the approach is demonstrated by evaluating R2U2 on a NASA DragonEye UAS in a lab environment. Afman et al. propose a mechanical-based safety system to design a safe-to-crash UAS, in addition to discussing existing hardware and software safety measures [51]. Coupled with runtime verification, such designs will improve UAS trust.

Model checking helps in verifying the correctness of runtime monitors. A model checking framework for runtime monitors, Copilot-Kind, is presented by Laurent et al. [69]. The tool augments the Copilot framework which targets runtime verification of ultra-critical systems [74].

Chapter 4

First Phase—Targeting a Commercially Available UAS

The first phase of this project was aimed at demonstrating the viability of the high-level design and monitor synthesis flow described in Chapter 3. The work focused on proving that an RTA system can be seamlessly added to an existing COTS UAS without affecting its performance. The monitors enforced a virtual cage and ensured that the UAS did not escape the cage. The complexity and number of monitors can be increased if the design is shown to be feasible.

Section 4.1 explains the virtual cage. Section 4.2 uses virtual cage's maximum altitude monitor to illustrate translation from LTL to a corresponding monitor implemented in hardware. Hardware model checking results for the maximum altitude monitor are described in Section 4.3. Section 4.4 highlights the virtual cage monitor's implementation, integration, and testing on a COTS quadrotor UAS.

The work described in this chapter, along with the high-level requirements captured in Chapter 3, is based on a published, co-authored paper [78]. The first phase work was a collaboration with Joseph Stamenkovich [79].

4.1 Virtual Cage

A virtual cage was chosen because of its simplicity and practicability. Virginia Tech has a drone cage that allows flying drones within a confined space. Figure 4.1 illustrates a simple virtual cage. The term "virtual" is used because it is not a physical cage; instead, the cage limits are programmed into the FPGA. The shape and complexity of the cage were limited by resource availability. Thus, a cuboidal cage was chosen because it allows for easier comparisons. As shown in the figure, the ground is the lower limit for the altitude. Five other sides—maximum altitude, minimum latitude, maximum latitude, minimum longitude, and maximum longitude—along with the ground define the cuboidal virtual cage. The UAV can stay anywhere inside the cage, but not outside it. The monitors trigger an RCF when the UAS is about the escape the fence. A virtual cage is a good approximation for drone operations such as surveying where the UAS can fly anywhere inside the designated area but not go outside it. Each fence is guarded by an independent monitor executing in parallel. The UAS location is decoded from the NMEA GPS packet [61].

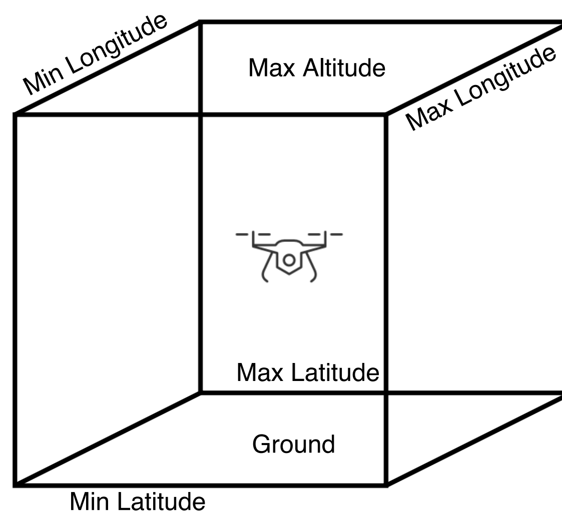


Figure 4.1: Virtual cage limits

4.2 Monitor Synthesis

Section 3.3 gave a high-level introduction of the monitor synthesis flow. This section explains the flow in detail using the virtual cage's maximum altitude monitor as an example. The architecture described in Section 3.1 supports an arbitrary number of monitor "plug-ins" synthesized from correctness specifications captured in LTL. To avoid potential errors arising from manual implementation, the monitors are generated using the steps described in following subsections.

4.2.1 Informal Requirements

The requirements for a monitor should first be captured in a language like English. This step is not directly involved in the flow. The objective for the $within_max_alt$ monitor is to ensure that the UAS does not rise above the set maximum altitude (MAX_ALT). This can be captured as a requirement, such as:

"The UAS shall remain below the MAX_ALT altitude."

The same requirement can be rewritten as:

"The output of the $within_max_alt$ monitor shall be false
if the UAS is about to rise above MAX_ALT."

However, informally captured requirements can be ambiguous and lead to different interpretations. Therefore, it is important to capture the requirements in a formal language.

4.2.2 Requirements in LTL

Formal specifications need to be captured in a language with a formal syntax and semantics. While propositional logic allows formulas to be defined from time-invariant, boolean-valued,

atomic propositions (APs) and logical operators, event sequencing and timing are essential aspects of reactive systems and need to be included in descriptions of correct behavior. LTL extends propositional logic with operators referencing the future. Time is modeled as a discrete sequence of states, and the temporal operators include:

$X\,a$	proposition a is true in the *neXt* state,
$F\,a$	proposition a will be true at some *Future* time,
$G\,a$	proposition a will be *Globally* true in the future,
$a\,U\,b$	proposition a will hold true *Until* b becomes true.

As an example, correct behavior for a UAS flying inside a virtual cage can be partially captured with the LTL formula

$$(nearing_fence \wedge in_slow_zone) \Rightarrow (can_slowdown\, U\, stopped) \tag{4.1}$$

In other words if the UAS is approaching a particular fence and enters the fence's slowdown zone then it should decelerate in proportion to the remaining distance from the fence until it stops, as illustrated in Figure 4.2. This expresses the desired approximate actions when a UAS moves towards a fence. The APs have a different binding for each fence. For example, the AP values for the maximum altitude monitor are determined from the UAS vertical speed (ver_speed) and altitude (alt):

$$nearing_fence = (ver_speed > 0) \wedge (alt > MID_ALT), \tag{4.2}$$

$$in_slow_zone = (alt > UP_SLOWDOWN_START), \tag{4.3}$$

$$can_slowdown = (ver_speed \propto (MAX_ALT - alt)), \tag{4.4}$$

$$stopped = (ver_speed = 0). \tag{4.5}$$

Momentum must be taken into account since a UAS cannot stop instantaneously. On the other hand, a UAS should be allowed to move parallel to a fence without decelerating even if it is within that fence's slowdown zone. The distinctions between nearing the fence and being in the slowdown zone, as well as between slowing down and stopping, provide more detailed scrutiny of the UAS behavior. One could impose additional behavioral constraints with extra formulas rather than complicate a single formula. For example, a catch-all assertion could require the UAS to always be inside the cage in case there is a literal loophole in the other formulas.

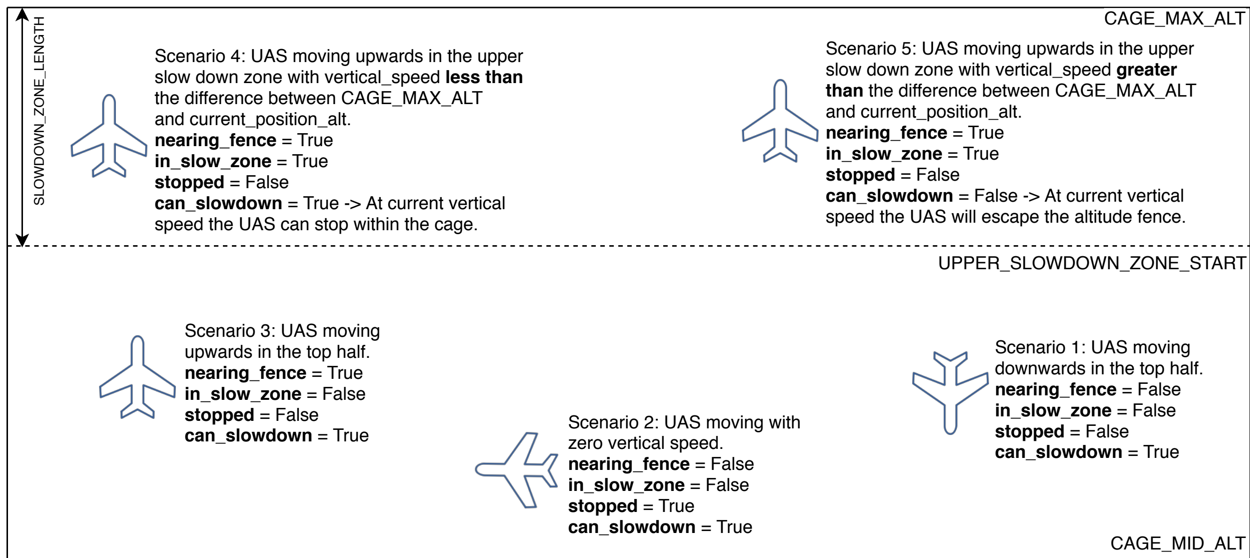


Figure 4.2: Atomic proposition values arising from different flight scenarios

4.2.3 LTL Specifications to Abstract Automata

LTL formulas can be converted to Büchi automata, which are an infinite input extension to finite automata used in model checking. EPITA Spot tool can be used to achieve this translation [12]. The Spot tool's open-source code was downloaded and added to the synthesis flow [40]. A C++ script uses the APIs provided by the tool to convert LTL formula

to a textual representation of the automaton. The Büchi automaton corresponding to Equation 4.1 generated by the EPITA Spot tool is shown in Figure 4.3. Valid event sequences are displayed; any other event sequences are invalid. A Büchi automaton accepts its input if a final state (the figure's rightmost state) is visited infinitely often. Rather than remaining in the accepting state, the automaton may be reset by returning to the initial state.

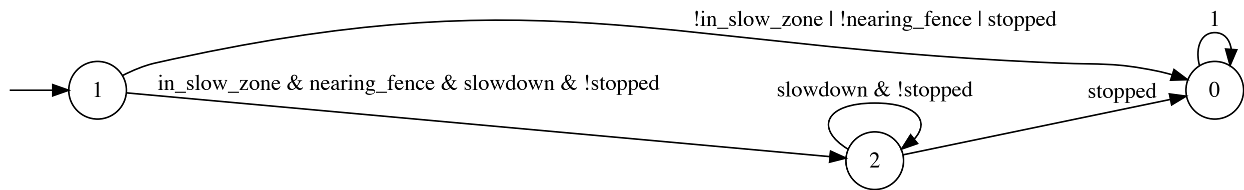


Figure 4.3: Büchi automaton for the *within_cage_max_alt* specification

4.2.4 Abstract Automata to C Code

An intermediate translation to C is easier than a direct translation to an HDL such as VHDL or Verilog, especially for defining override calculations and algorithms. Unlike HDLs, there is no need for explicit clock or reset signals. The function in Listing 4.1 is invoked once per control cycle, and is generated semi-automatically. First, a C++ script generates the function skeleton and state machine code from a textual representation of the *within_cage_max_alt* automaton shown in Figure 4.3. State information is retained in a local, static `state` variable. Function arguments provide the sensor values needed to define the APs. The C++ script adds declarations of the APs appearing in Equation 4.1, although initialization code must be manually entered (lines 1,2, 7-10). Any undefined automata transitions cause the `state` variable to have the `ALARM_STATE` value. If this occurs then an RCF (such as forcing the UAS to hover or land) may be invoked.

Listing 4.1: C function for the *within_cage_max_alt* automaton

```c
#define MID_ALT ((CAGE_MIN_ALT + CAGE_MAX_ALT) / 2)
#define UP_SLOWDOWN_START (CAGE_MAX_ALT − SLOW_ZONE_LENGTH)

bool within_cage_max_alt(int alt, int ver_speed) {
  enum States {STATE0, STATE1, STATE2, ALARM_STATE};
  static enum States state = STATE1;
  bool nearing_fence = ((ver_speed > 0) && (alt > MID_ALT));
  bool in_slow_zone  = (alt > UP_SLOWDOWN_START);
  bool can_slowdown  = (ver_speed <= (MAX_ALT − alt));
  bool stopped       = (ver_speed == 0);

  switch (state) {
      case STATE0: //accepting state
      case STATE1:
        if (!in_slow_zone || !nearing_fence || stopped)
          state = STATE0;
        else if (in_slow_zone && nearing_fence && can_slowdown && !stopped)
          state = STATE2;
        else
          state = ALARM_STATE;
        break;
      case STATE2:
        if (stopped)
          state = STATE0;
        else if (can_slowdown && !stopped)
          state = STATE2;
        else
          state = ALARM_STATE;
        break;
      default:
        state = ALARM_STATE;
        break;
  }
  return (state != ALARM_STATE);
}
```

Although the function code requires manual additions, all monitor automata code is auto-

matically synthesized from the original LTL. Hence a state machine implementation detecting discrepancies between correct and observed behavior is generated directly from formal specifications. This is preferable to manually generating an abstract automata or its C implementation because of the added complexity and potential for oversights. For additional assurance, Section 4.3 applies model checking to the automaton's final hardware implementation without concern for state explosion.

Translating Büchi automata to C allows several extensions that cannot be expressed in LTL. As shown, APs may be defined using any C language statements and function calls applied to sensor variables. Although not applicable to this example, liveness can be checked by imposing a time limit to reach an accepting state. This provides much of what is offered by timed automata [52] without complicating the LTL. If a control cycle counter reaches zero before a particular state is reached, a corrective action can be requested by the monitor. Hence time bounds can be set on certain state transitions without adding extra states and transitions. These extensions still permit the use of existing tools to translate basic LTL formulas into Büchi automata.

4.2.5 C Code To Parallel Hardware

High-level synthesis (HLS) transforms C code into digital circuits rather than processor instructions [71], and is used to generate the automata and switching logic within the monitor. Direct hardware implementation prevents common errors or attacks such as buffer overflows and stack corruption since monolithic memory and stacks are not used. Rather, physically distinct and private memories are allocated to arrays and buffers, and dedicated registers store function arguments and return values. This can improve performance and enable parallelism since a single, shared memory is often a computing bottleneck. The full

C language may be used except for library functions providing system calls and dynamic memory management.

To workaround C's sequential semantics, HLS normally adds parallelism by unrolling and/or pipelining inner `for` loops containing the calculations used in vector, matrix, and reduction operations. However, we require task parallelism (concurrent execution of monitor functions such as the virtual cage's five monitors) rather than data parallelism. Effective use of HLS requires the software structure and interfaces to mirror the desired hardware structure. This was the primary reason to: (1) isolate each monitor in a function such as Listing 4.1's `within_cage_max_alt()`; (2) explicitly pass sensor values as arguments so that the monitors may be consulted every control cycle; and (3) minimize the function's code complexity in order to reduce the input/output latency.

Listing 4.2: C function for the *within_cage* monitor wrapper

```
1 bool within_cage(int alt, int ver_speed, int lat, int lat_speed,
2                  int lon, int lon_speed) {
3   return (within_cage_max_alt(alt, ver_speed) &
4           within_cage_min_lat(lat, lat_speed) &
5           within_cage_max_lat(lat, lat_speed) &
6           within_cage_min_lon(lon, lon_speed) &
7           within_cage_max_lon(lon, lon_speed));
8 }
```

HLS semantics permit a sequence of functions to be executed concurrently if there are no dependencies. Therefore, HLS is also applied to a wrapper C function that invokes all monitors in order to execute them in parallel. The wrapper C function shown in Listing 4.2 accepts the six arguments (altitude, vertical speed, latitude, latitude speed, longitude, and longitude speed), invokes the five fence monitors with these arguments, and returns whether all monitors return `true`.

As will be described in more detail in Section 4.4, the Intel HLS compiler [48] generates a

hardware block for the wrapper function with the interface shown in Figure 4.4. The wrapper hardware module appears to a processor as a bus slave component with the function's arguments, return value, and handshake registers mapped to consecutive memory addresses [49]. Unlike HDL, the clock, reset, and handshake signals are not explicit in the C code. If `busy` is not asserted, arguments can be written to the input registers prior to asserting `start`. Return values can be read from the output registers when `done` is asserted.

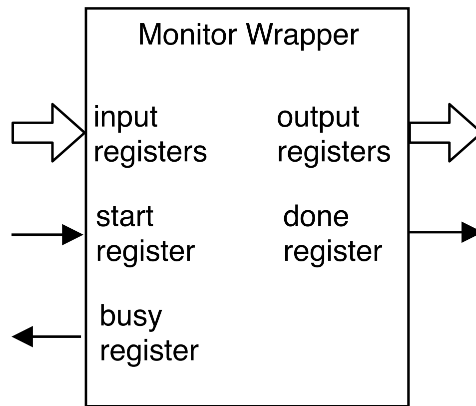


Figure 4.4: Monitor wrapper registers

4.3 Monitor Implementation Analysis

Synthesized monitors are analyzed using a model checking tool to ensure they satisfy formally captured behaviors. Model checking is facilitated by the explicit state machine structure of the monitor code. While conventional testing can only check for correct behavior during particular executions, model checking can ensure these behaviors across all possible executions. We opt to perform model checking on the HDL code generated by the HLS rather than the C code because it is closer to the final implementation. Since most HDL model checking tools support Verilog or SystemVerilog, the HLS generated VHDL code is converted to Verilog code using the `vhd2vl` tool [50]. This Verilog code is converted to SystemVerilog

code with the addition of SystemVerilog Assertions (SVAs). The translation from VHDL to Verilog is not required if the HLS tool's output is in Verilog (as is the case in the Xilinx Vivado HLS tool).

The monitor analysis steps are depicted in Figure 4.5. SystemVerilog code is input to the EBMC tool which can perform both bounded and unbounded model checking [47]. We are interested in two different analyses: (1) verify whether the synthesized HDL code satisfies Equation 4.1; and (2) verify whether the LTL specifications are correctly stated. The second analysis is important since behaviors may be under-specified or over-specified. Table 4.1 describes the assertions both informally and formally, and whether the assertion is satisfied.

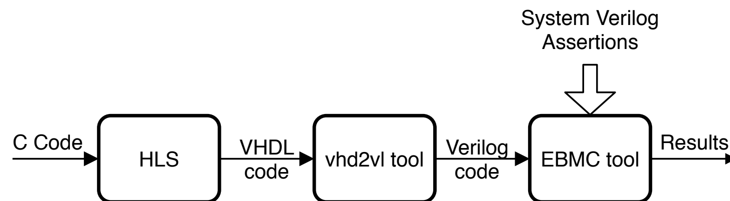


Figure 4.5: Monitor analysis process

The first case is analyzed by translating Equation 4.1 to SVA *S1*. The EBMC tool generates an expected counterexample for this assertion: the UAS is in the slowdown zone, heading towards the fence at a speed with which it cannot slow down and stop within the fence. This scenario causes a transition to the monitor's `ALARM` state which triggers a "land" RCF. The second case is analyzed by submitting different logical queries to the EBMC tool. SVAs *S2*–*S6* are not derived from the LTL formulae and independently capture the expected behaviors of the monitor. For example, *S2* checks whether (1) the monitor always detects if the UAS cannot slow down and stop within the cage, and (2) the monitor enters the `ALARM` state when this situation occurs. Validation of this assertion confirms there is no scenario in which the monitor does not detect if the UAS is about to escape the cage. Monitor trust is enhanced by confirmation of all assertions.

ID	Description	SystemVerilog Assertion	Result
S1	LTL formula (Equation 4.1)	$(nearing_fence$ & $in_slow_zone) \mid\rightarrow (can_slowdown$ until $stopped)$	Failure
S2	Monitor always detects when the UAS is about to leave the cage	$(current_state$!= ALARM & $!can_slowdown) \mid\rightarrow (!within_cage_max_alt$ & $next_state$ == ALARM)	Success
S3	Monitor does not trigger a false alarm	$(current_state$!=ALARM & $(can_slowdown \parallel stopped)) \mid\rightarrow within_cage_max_alt$	Success
S4	Monitor remains in alarm state once it enters that state	$(current_state$==ALARM) $\mid\rightarrow (!within_cage_max_alt$ & $next_state$ == ALARM)	Success
S5	UAS cannot escape the cage if it is not nearing the fence	$(current_state$!= ALARM & $!nearing_fence) \mid\rightarrow within_cage_max_alt$	Success
S6	UAS cannot escape the cage if it is not in the slow zone	$(current_state$!= ALARM & $!in_slow_zone) \mid\rightarrow within_cage_max_alt$	Success

Table 4.1: SystemVerilog Assertions applied to the *within_cage_max_alt* monitor's hardware implementation

4.4 Implementation in the Intel Aero

The virtual cage was imposed on an Intel Aero drone, shown in Figure 4.7. The Aero is a commercial-off-the-shelf (COTS) quadrotor with airframe parts supplied by Yuneec and a computing board supplied by Intel. It was selected because of a nearly unused MAX 10 FPGA between the compute board's Atom quad-core application processor and Arm-based flight controller 4.6. The Aero normally only uses the FPGA to implement an analog-to-digital conversion block to monitor the battery voltage, SPI interface, and I2C bridge for the compass [64]. The Atom processor's operational frequency is 2.4 GHz, and the drone can fly

for up to 20 minutes with a maximum operating distance of 984 feet and a ceiling of 4500 meters. It can reach horizontal speeds up to 49 feet per second and has a maximum weight of about 4 pounds [1]. The constraints mentioned in Section 3.2 were satisfied by implementing the RTA system in the FPGA. If any of the monitors are triggered, the commands from the Atom processor can be bypassed, and an RCF can be executed.

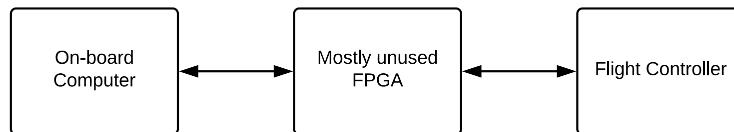


Figure 4.6: Intel Aero compute board before adding the monitors

Figure 4.7: Intel Aero drone with GPS unit mounted on the top

With all the relevant data and sensor information passing through the FPGA, monitors were seamlessly added to the system despite the small size of the FPGA. Figure 4.8 shows the added FPGA functionality. On platforms without the necessary components already

present on the system, hardware would have to be added. Flight functions can be preloaded on the Atom or transferred in real time from a ground station to the Atom over Wi-Fi. Common choices are QGC and Mission Planner [24]. The Aero supports running either PX4 or ArduPilot firmware in the flight controller. ArduPilot was selected for its ability to accept human-readable GPS data.

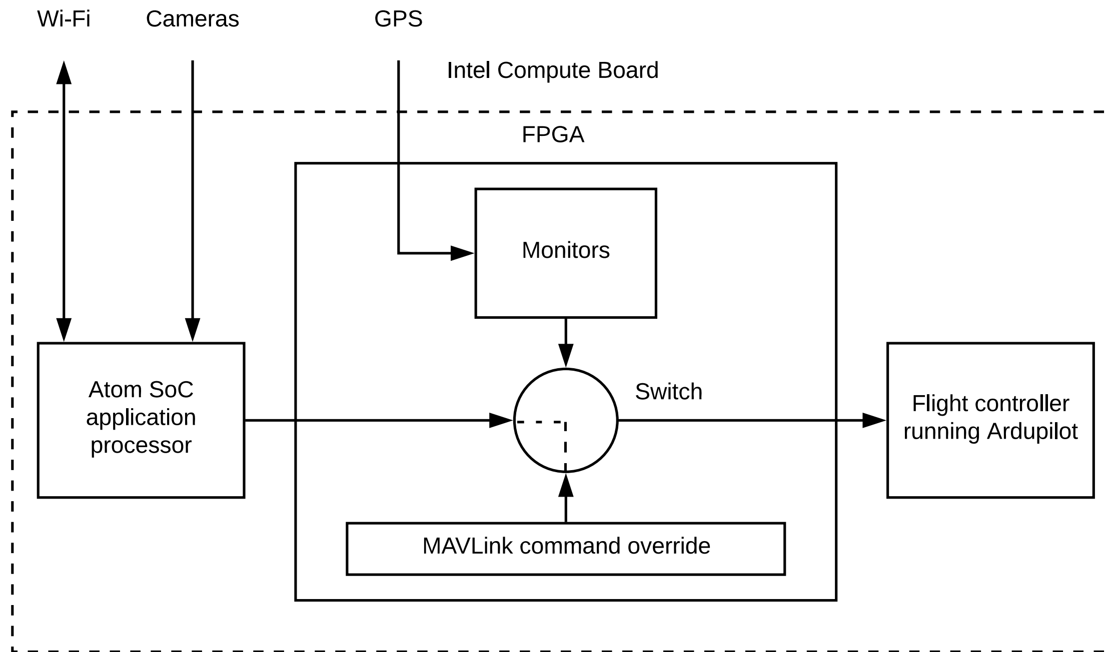


Figure 4.8: Aero compute board after adding monitors

4.4.1 Pedigreed Component Architecture

The Nios II soft processor architecture is available on Intel FPGAs [3]. Figure 4.9 shows the resource types available on the MAX 10 FPGA [45, 46]. A phase-locked loop (PLL) takes the 25 MHz external clock signal as an input and produces a 10 MHz clock for the ADC, a 50 MHz clock for the SPI interface, and a 100 MHz clock for the Nios II processor. The large number of input and output pins available allow for communication with numerous other subsystems, including the GPS and MAVLink interfaces used by the safety monitors.

Even though the FPGA functionality may be altered at design time, it cannot be changed at runtime. The programming interface can even be physically disabled, effectively turning the FPGA into a fixed-function ASIC. The soft processor has a simple 32-bit RISC organization with performance options such as a five-stage pipeline.

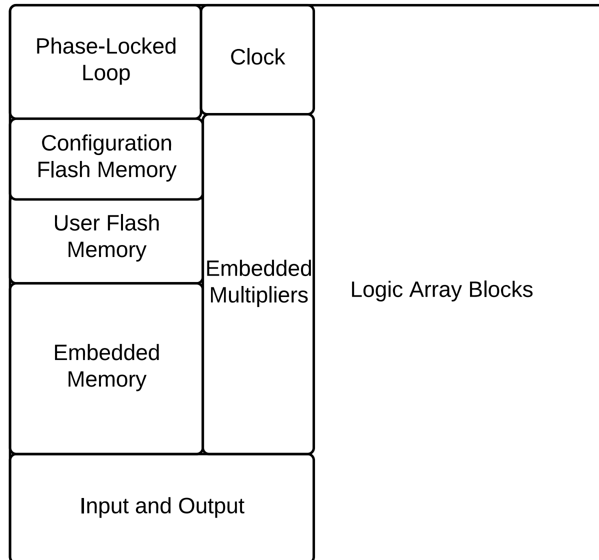


Figure 4.9: MAX 10 FPGA hardware resource types

Figure 4.10 shows the individual components and their connections. The Nios II processor handles the I/O interrupts, invokes the monitors, and initiates the RCF. There are separate UART communication channels for the GPS data, the only sensor information necessary for the demonstration, and the MAVLink protocol between the Atom application processor and the flight controller. The soft processor does not interact with any of the existing components shown at the bottom of Figure 4.10. Instantiating a pipelined processor uses 58% more lookup tables (LUTs) and 27% more flipflops than the monitors.

Performance-oriented hardware and software optimizations include:

- Large hardware FIFO buffers added to the UARTs enable much faster baud rates by buffering entire packets in hardware before being processed by software interrupt

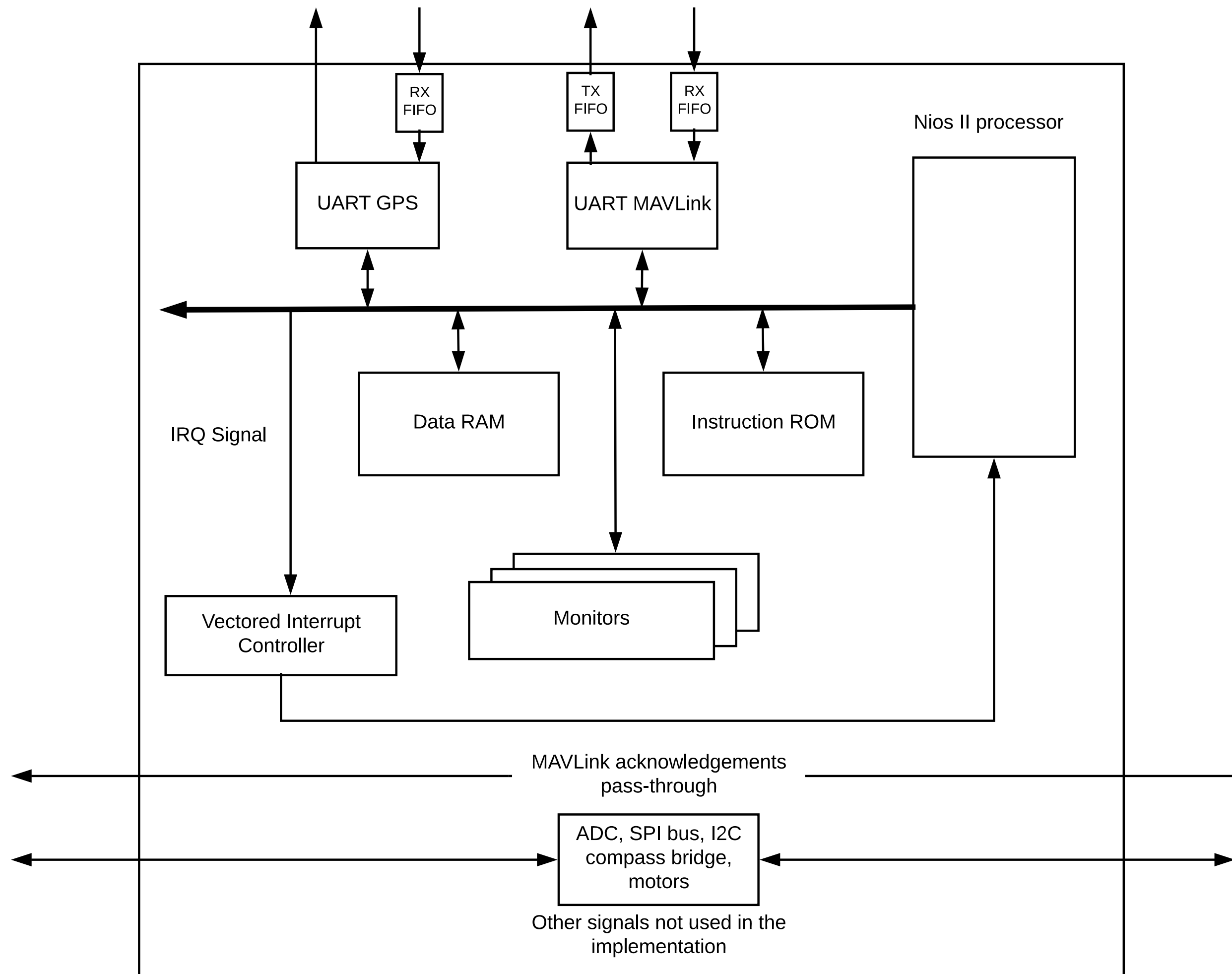


Figure 4.10: I/O soft processor and monitor blocks

handlers. These require 265% more LUTS, 246% more flipflops, and 8k memory bits compared to standard, memory-less UARTs with hardware buffering for just a few bytes.

- Interrupt context switching overhead is drastically reduced by an external vectored interrupt controller that causes the processor to switch to a shadow register set [2].

- Tightly coupled data and instruction memories enable concurrent instruction and data access with cache-level performance. By better partitioning and organization of mem-

ory, performance is increased without additional logic elements.

- A double-buffer scheme allows concurrent processing of consecutive packets by the `main()` and interrupt tasks. Twice as much memory is required compared to a default ring buffer scheme, but the memory is present and available on the FPGA.

- Vectoring all UART interrupts to the same interrupt routine allows all pending interrupts to be checked before returning from the interrupt handler. This optimization reduces instruction memory requirements by having fewer independent functions.

- To avoid floating point arithmetic, the degree, whole minute, and first three fractional minute digits of the GPS position string are treated as integers to generate a distance precision of minutes/10^3 with

$$(\text{deg} \times 3600000) + (\text{min} \times 60000) + (\text{frac. min}) \tag{4.6}$$

  The precision is still well within GPS resolution. For indoor applications where the degree is not relevant, a precision of minutes/10^6 is possible using six fractional minute digits in

$$(\text{min} \times 60000000) + (\text{frac. min}) \tag{4.7}$$

4.4.2 Monitor Block Integration

Algorithm 1 summarizes the Nios II processor's cyclic executive code structure. Initialization includes enabling UART communication channels, connecting them to an interrupt service routine, and ensuring that a GPS lock has been obtained in accordance with F3269-17 requirement (4) discussed in Section 3.1. The Nios II code is part of a pedigreed component since it initiates RCFs in response to monitor anomaly detections. RCF precomputations

help to ensure compliance with requirements (10) and (11). The main loop verifies the integrity of the data prior to sending it to the monitors, which satisfies requirements (6) and (12). Relevant sensor data are written to the monitor wrapper registers shown in Figure 4.11, followed by a write to the `start` register. These registers can only be accessed by the Nios processor and are therefore isolated from non-pedigreed components, in accordance with requirement (2).

Algorithm 1 main() task

```
1: initialize communication
2: make recovery commands
3: verify sensor data
4: while forever do
5:     if new sensor data then
6:         load sensor data into the monitors
7:         start monitors
8:         wait for monitors to finish
9:         if any monitor indicates violation then
10:            stop the producer from receiving new data
11:            overwrite the producer with the RCF
12:            if full packet received and transmitted then
13:                if buffers are ready to swap then swap buffers
14:                end if
15:            end if
16:        end if
17:    end if
18: end while
```

As shown in Figure 4.12, the soft processor treats the monitor wrapper as an Avalon bus peripheral with a conventional software/hardware interface consisting of memory mapped data, control and status registers. The wrapper signals completion via a `done` bit in a status register, after which the overall monitor outcome may be read from a data register. Using a hardware wrapper for hardware monitors utilizing the same sensor data and RCF achieves the lowest latency by permitting a hardware-implemented, logical AND reduction to the five

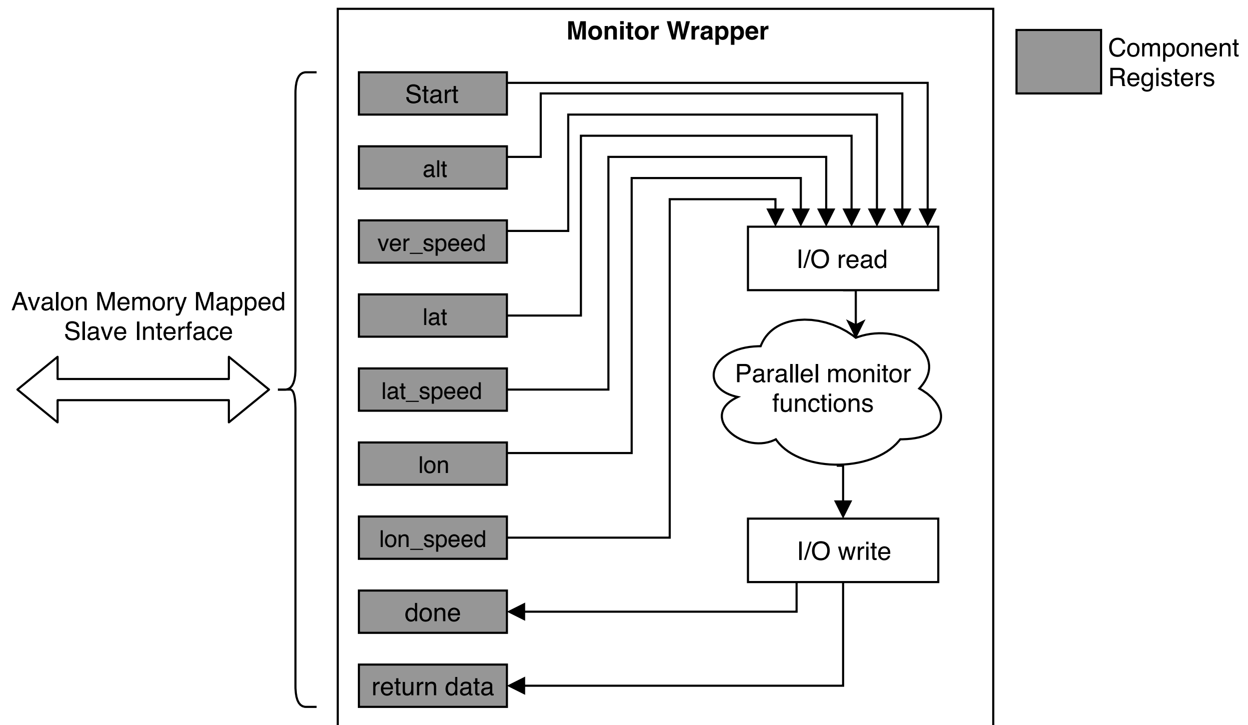


Figure 4.11: Monitor wrapper registers

monitor outputs that are computed at approximately the same time. If the system is within the predefined limits as determined by the monitors analyzing the current sensor data, the main loop repeats the overall cycle with new sensor data. If the system is not within the predefined limits, the appropriate RCF is selected and triggered in a manner that satisfies F3269-17 requirements (1), (8) and (9).

Processing sensor and complex function data is a critical role for the Nios II processor, with latency of particular concern. Optimization focused on minimizing interrupt latency and maximizing UART baud rates to reduce timing disturbances on communication between the complex functions and flight controller. Algorithm 2 shows the interrupt task actions. A double buffer scheme is utilized with transmit and receive buffers for each UART channel. All packets are either a fixed size or have a length field in the packet header. Once this size has been reached by both the `main()` and interrupt tasks, their buffer pointers are

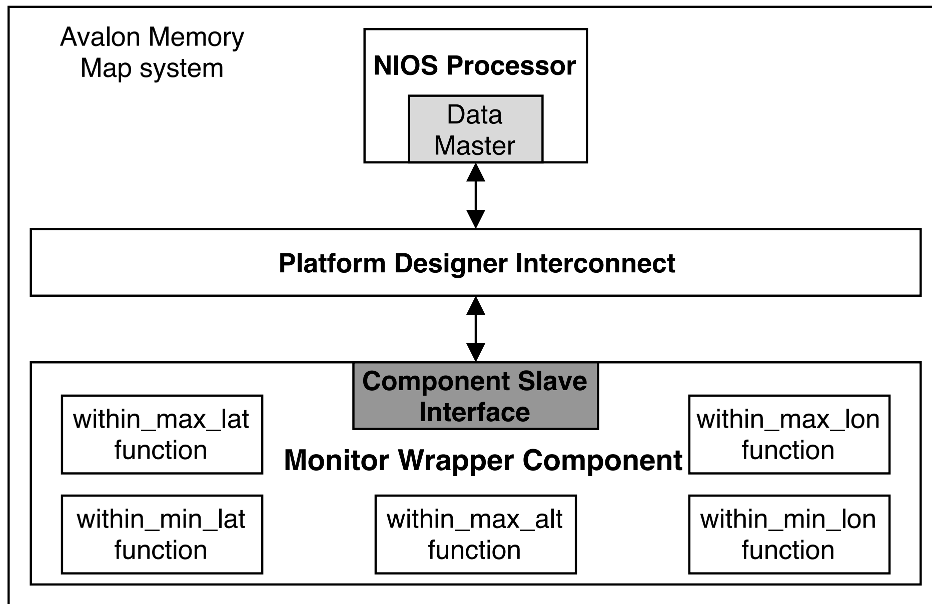


Figure 4.12: Processor interface to the monitor wrapper

simply swapped to avoid buffer copy and clearing overheads. While a single circular buffer is sufficient for byte transfers, a buffer pair is more appropriate for MAVLink's variable-sized packets subject to transmission errors and flushing.

Both tasks must be finished receiving or transmitting their respective packets for the swap to occur. When an interjection is triggered, data reception is stopped and replaced by the RCF. To curtail commands triggering the override and fulfill F3269-17 requirement (7), the RCF overwrites the transmit and not the receive buffer while still allowing for a continuous flow of commands to the flight controller. Loading a complete RCF packet into the transmit buffer makes it ready to swap with the receive buffer. If the consumer buffer was overwritten instead, a partially completed packet would be sent leaving the system in an undefined state while also wasting time receiving data that is dropped. By checking to swap after loading the receive buffer with the RCF, the last packet transmission is allowed to complete.

Algorithm 2 Interrupt handler task

```
1:  if any interrupt triggered then
2:      for each communication channel do
3:          if pending data to receive then add byte to receive buffer
4:          end if
5:          if pending data to transmit then remove byte from transmit buffer
6:          end if
7:          if full packet received and transmitted then
8:              if buffers are ready to swap then swap buffers
9:              end if
10:         end if
11:     end for
12: end if
```

4.4.3 Resource Utilization and Performance Analysis

Table 4.2 shows the FPGA resources available and those used to implement preexisting interfaces and the RTA consisting of the Nios II soft processor and monitor hardware blocks. The processor and monitors use the same 100 MHz clock. No additional I/O pins were required by the RTA. The Nios II processor consumed roughly 33% of the logic elements, the monitors consumed 23%, and the UARTs consumed 18%.

Resource	Available	Used	Usage
Lookup table	8064	5799	72%
Flipflop	8064	3436	43%
Embedded SRAM (Kb)	49	38	78%
User flash memory (Kb)	1376	0	0%
General purpose I/O pin	112	34	30%
Embedded multiplier	48	6	13%
Clock phased-locked loop	1	1	100%

Table 4.2: Resource utilization for the MAX 10M08 FPGA

Packet-sized hardware FIFOs allow 921k baud over the MAVLink channel, the standard baud rate used by the PX4 flight controller. Prior to this optimization, communication

was limited to 57.6k baud, which is supported by ArduPilot. The interrupt service routine originally consumed 3% of the processor cycles, which was decreased to 1.5%. Table 4.3 details the average execution time required by various software and hardware modules. The last four rows compare the different means of implementing the monitors. The sequential, overlapped, and parallel schemes differ primarily in the software/hardware interface, with sequential corresponding to a blocking interface to each monitor, overlapped corresponding to a non-blocking protocol, and parallel using hardware control circuitry to enable true concurrent monitor execution. Wrapping the five hardware monitors in a hardware block resulted in the fastest execution. Because the virtual cage monitors are computationally simple, software/hardware interfacing overheads reduce the speedup arising from hardware implementation.

Module	**Implementation**	**Time (ns)**
Initialize communication	software	6640
RCF precomputations	software	17220
Parse sensor data	software	70883
RCF interjection	software	6096
Double buffer swap	software	571
Interrupt service routine	software	2064
Timing measurement overhead	software	20
5 sequential monitor functions	software	898
5 sequential monitors	hardware	1512
5 overlapped monitors	hardware	1111
5 parallel monitors + wrapper	hardware	779

Table 4.3: Average execution time of software and hardware modules

4.4.4 Virtual Cage Flight Test Results

The monitors invoke a `land` RCF when attempting to manually or autonomously fly the UAS outside a 100×100 foot lateral by 50 foot vertical virtual cage defined at the Kentland

experimental aerial systems outdoor test facility. After placing the drone at the center of the cage, it is necessary to wait for a GPS lock although this is normally done before any flight. An additional monitor could be defined to prevent takeoff before the GPS lock is achieved. GPS data accuracy and frequency limit the precision of the cage boundaries. To prevent the drone from leaving the cage between sensor packets, and to provide enough time to land before leaving the cage, a trigger zone is defined within the cage borders. If the monitors determine that the drone's current speed could result in a cage breach before the next GPS sensor packet, a land is triggered. Expensive calculations such as division are avoided by calculating speed in terms of the difference between consecutive GPS locations.

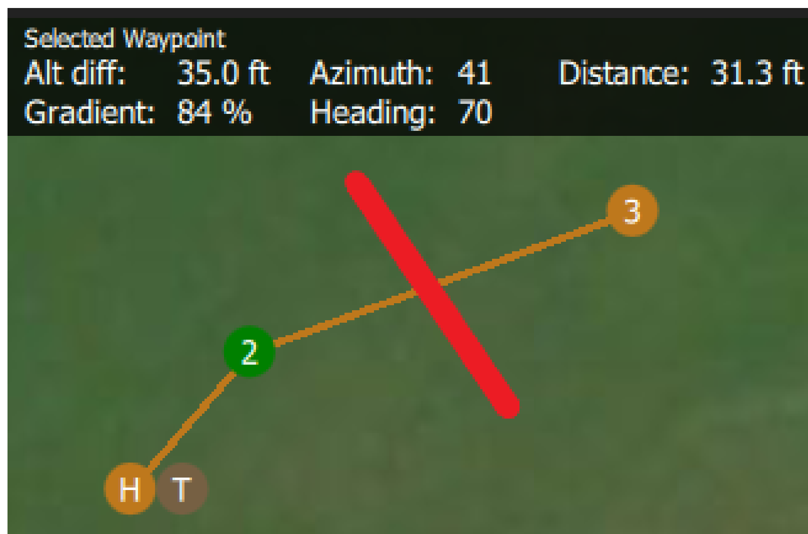


Figure 4.13: Triggering the land RCF

Figure 4.13 shows a flight track with the RCF initiated at the cage boundary shown by the red line. The UAS first climbs to 35 feet, remaining under the altitude limit of 50 feet. It then proceeds to waypoint 2 that is 31.3 feet from the takeoff position. When the speed defined above is greater than the distance remaining to the edge of the cage, the drone is forced to land. The trigger zone is sufficiently wide so that the drone may not breach the cage before the next GPS packet even at its maximum horizontal speed. The drone tries to

reach waypoint 3 (70 feet from the start), but is forced to land near the 50 foot fence. A faster drone may need a greater sensor sampling frequency or a larger trigger zone. Inertia and drone flight mechanics may occasionally result in the drone landing slightly outside the cage.

To summarize, the first phase of this work demonstrated that it is possible to add an RTA system to an existing UAS consistent with the ASTM standard F3269-17 to enforce strict behaviors. Though the monitors were simple, the idea can be extended to a larger number of sophisticated monitors, as discussed in the next chapter.

Chapter 5

Monitoring an Autonomous Flight Beyond Visual Line of Sight

The second phase of this project, covered in the next three chapters, was focused on creating complex monitors that provide assurances for autonomous flights BVLOS. Section 5.1 discusses the motivation to extend the work to monitor an autonomous flight with a predefined flight plan. Section 5.2 introduces the improved architecture that supports a large number of monitors, wireless interface to a trusted device, and HITL simulation. The HITL simulation with Pixhawk hardware, Gazebo simulator, and QGC is discussed in Section 5.3. Sections 5.4 to 5.8 explain different components in the updated architecture, which includes the User interface processor (UIP), Monitor processor (MP) and I/O processor (IOP).

5.1 Motivation

The simple virtual cage imposed on the Intel Aero helped to demonstrate the feasibility of our approach. However, real-life situations are more complicated than that.

1. **Create a geofence with a complex geometry**

   Complex geometry is typically an intersection of several simple shapes. These days drones are used to survey oil fields and construction sites (water bodies and hilly areas in partic-

ular). It is important to ensure that the drones remain in the permitted airspace, which may have complex geometry, and do not enter the restricted areas.

2. **Create the geofences dynamically**

   In the first phase, the geofence was static, which meant that the FPGA had to be reprogrammed every time the UAS was moved to a new location. This is not desirable in a commercial setting. The design should be capable of creating the geofence dynamically by digesting a flight plan to remove the time overhead incurred from programming the FPGA.

3. **Monitor other sensor data**

   Apart from the global position of the UAV, it is desirable to monitor other sensor data like ground speed, vertical speed, pitch rate, yaw rate, and roll rate. Monitoring these sensor data provide information about the stability of the flight and the ability to guard against other unexpected behaviors.

4. **Expanding the number of monitors**

   How many monitors can be added without affecting the performance of the system? Ideally, a runtime safety assurance system should be able to monitor all the important sensor data with minimal latency.

The next challenge was to design a monitoring system that allowed UAV to fly BVLOS. This involved dynamically creating sophisticated monitors, each guarding against a different violation. In addition to GPS location, other sensor data were monitored, which subsequently increased the number of monitors. Having this system in place will increase the trust in the UAS and allow for drone operations like delivery or pipeline patrols.

Intel discontinued the Aero drone, and it became difficult to get support. Thus, to demonstrate this design, there was a transition from real flights with an Intel Aero to virtual flights

in an HITL simulation environment. This change allowed transitioning from a small FPGA to a more powerful programmable SOC platform, the Xilinx Zynq UltraScale+ ZCU104 evaluation board [44]. This board has at least 60x the logic element resources of the Intel Aero's MAX 10M08. The ZCU104 also includes a quad-core, 64-bit, >1 GHz Arm Cortex-A53 application processor, a dual-core, 32-bit Arm Cortex-R5 real-time processor, and two independent DDR4 memory channels. The abundant resources enabled implementing several monitors without concern about exhausting the resources.

5.2 Architecture

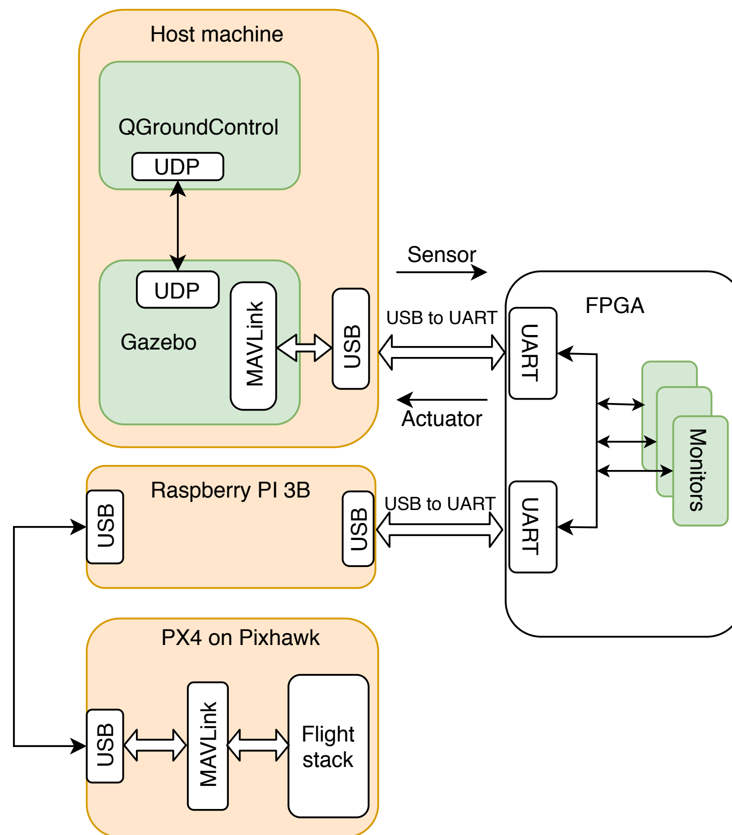


Figure 5.1: HITL simulation architecture

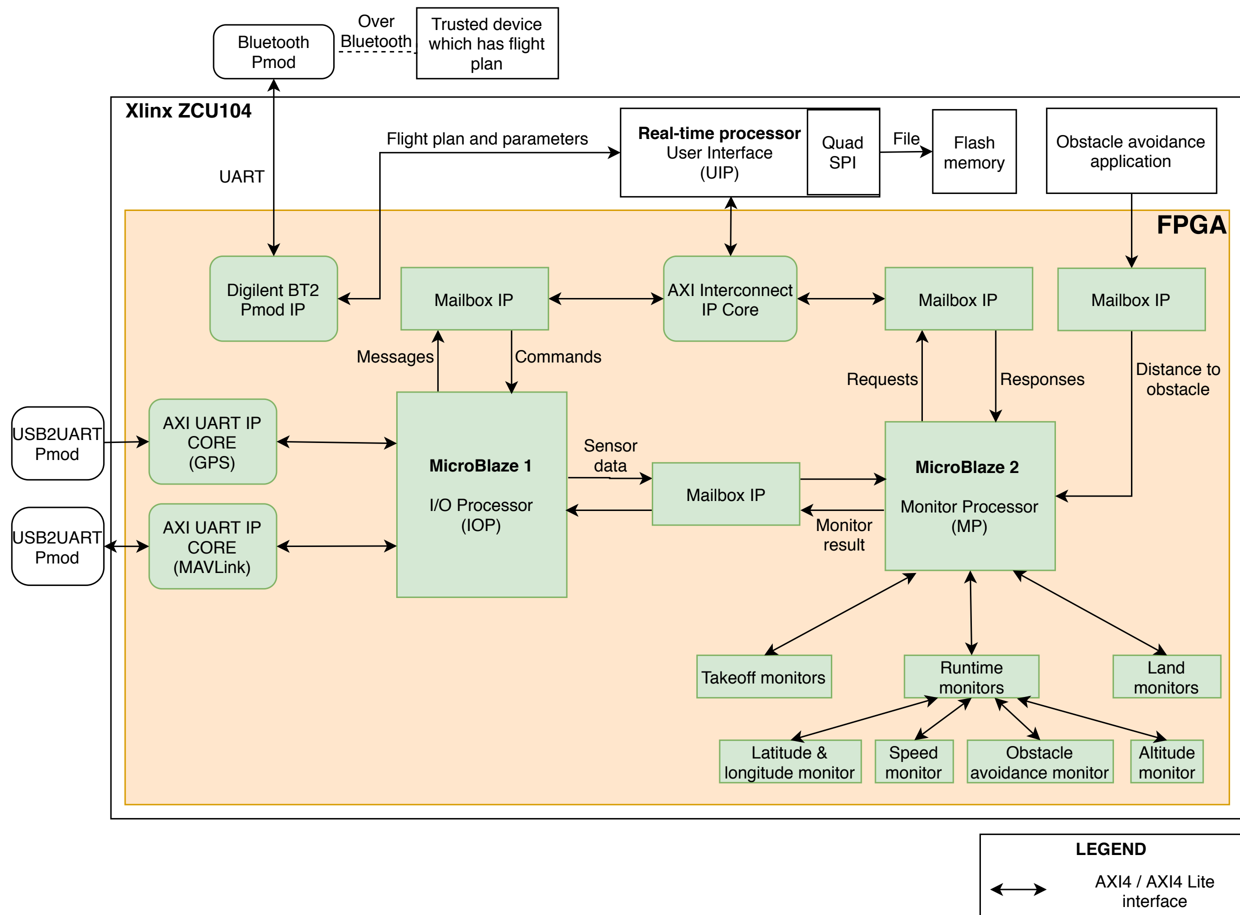


Figure 5.2: RTA design architecture

Figure 5.1 captures the overall architecture for HITL simulation and Figure 5.2 gives a closer look at the FPGA. The architecture provides assurances for an autonomous flight by monitoring data from multiple sensors.

5.2.1 Design Choice

The design consists of three processors, each performing a distinct task and interacting with each other. The monitors are implemented as hardware blocks in the FPGA. Processors help the monitors to perform their task. The reason for implementing monitors in hardware was discussed in detail in Section 3.2.1.

5.2.1.1 Why Multiple Processors?

In the initial demonstration, the small FPGA in the Intel Aero restricted the number of soft processors to one. An interrupt service routine (ISR) was implemented to read sensor data from the UART buffer. However, the resource-rich Xilinx Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC enabled the implementation of a multiprocessor system [25].

It is desirable to have multiple processors, each performing a distinct task, rather than a single software image trying to handle everything. Simple, separate functions are easier to understand, debug, and maintain. In a system with multiple interfaces to the outside world, it is desirable to handle these interfaces individually. Typically, ISRs are used to handle external interfaces. However, they make the system non-deterministic as a constant latency cannot be guaranteed. The context switching complications make it difficult to establish the correctness of the system. In order to facilitate applying model checking to the software code in the processors, ISRs were avoided. Once the relationship between the processors is established in a multiprocessor system, they become more independent. Thus, it is easier to establish the correctness of each function. As a result, multiple processors, each performing different I/O operations and interacting with each other, were chosen.

5.3 Hardware-in-the-loop Simulation

HITL simulation refers to the simulation of every component of a system except one. There is one hardware component that interfaces with all other components simulated in software. For this demonstration, HITL simulation with a Pixhawk flight controller is used [30]. The Pixhawk runs the PX4 autopilot software and the rest of the UAV components are simulated in the Gazebo simulation environment [35].

A typical HITL simulation with Gazebo, QGC, and Pixhawk is illustrated in Figure 5.3. Gazebo environment simulates the UAV components like actuators and is responsible for updating the sensor values. The frequency of updates varies between different sensors. These sensor values are communicated to the autopilot through MAVLink packets. The flight controller is responsible for taking control decisions and commanding the actuators. Commands from the Pixhawk are sent back to Gazebo to simulate the actuators.

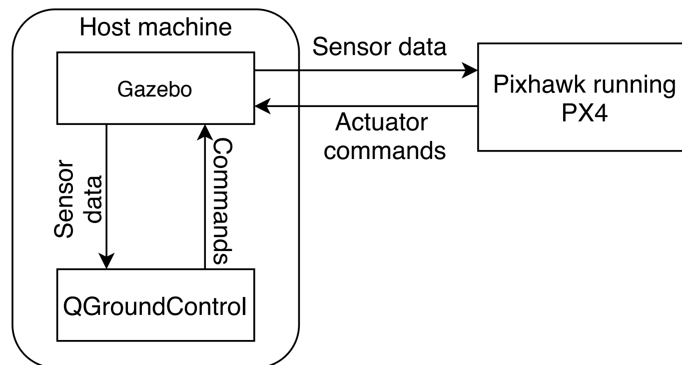


Figure 5.3: HITL simulation

QGC provides a better graphical user interface (GUI) than Gazebo to view the simulated flight. It also allows users to send commands to the autopilot. The sensor values are communicated to the QGC from Gazebo. Commands from QGC (like `Start`) are sent to Gazebo, which in turn routes it to the autopilot.

The RTA system is added between the Gazebo simulation environment and the Pixhawk autopilot, as shown in Figure 5.4. This position allows the RTA systems to access the sensor data for monitoring purposes. Besides, if a guard is triggered, it can cutoff commands from application software (or QGC) while executing the RCF.

The Xilinx ZCU104 board did not have the required number of USB connections to establish communication between the host PC and Pixhawk. However, there were unused Pmod ports

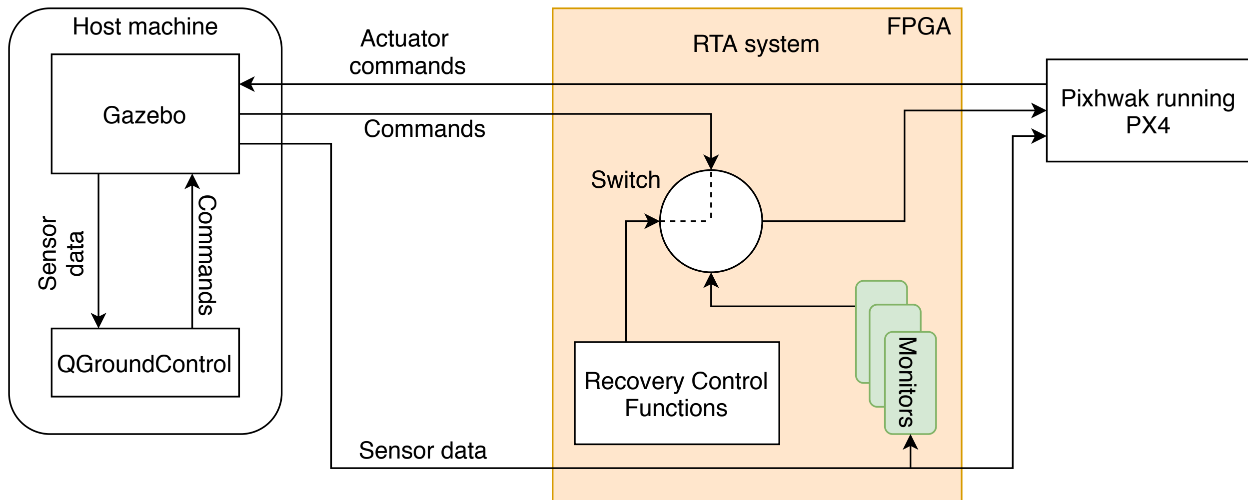


Figure 5.4: HITL simulation with RTA system in place

on the board. As a result, USB-to-UART Pmod devices were used as an intermediary. These devices convert serial USB data to UART and vice versa. To transfer data serially over USB, one of the connected devices must be a host device. Both the Pmod USB-to-UART device and the Pixhawk are incapable of functioning as a host. Therefore, a Raspberry Pi was added in between the Pmod and Pixhawk to serve as a host on both sides.

The flow of data in the HITL simulation (Figure 5.1) is captured below:

1. Gazebo transmits sensor data and commands serially over USB. The other end of the USB is connected to the Pmod device, which stores the data in the UART buffer.

2. The IOP reads the data from UART and transmits to the Raspberry Pi through the Pmod pins. The sensor data utilized by the monitors are forwarded to the MP.

3. The Raspberry Pi sends the received data to the Pixhawk.

4. The actuator commands sent by the Pixhawk bypass the RTA system and are directly routed to Gazebo.

5.4 User Interface Processor

The 64-bit ARM Cortex-A53 quad-core processor is generally used for running software applications and is not suited to perform I/O operations. In addition, I/O operations are generally handled by the real-time processors. Thus, one core of the 32-bit dual-core ARM Cortex-R5 real-time processor is used to interact with the user. A trusted device (smartphone, tablet, or laptop) can communicate with the FPGA over Bluetooth.

5.4.1 Monitors Configuration Flow

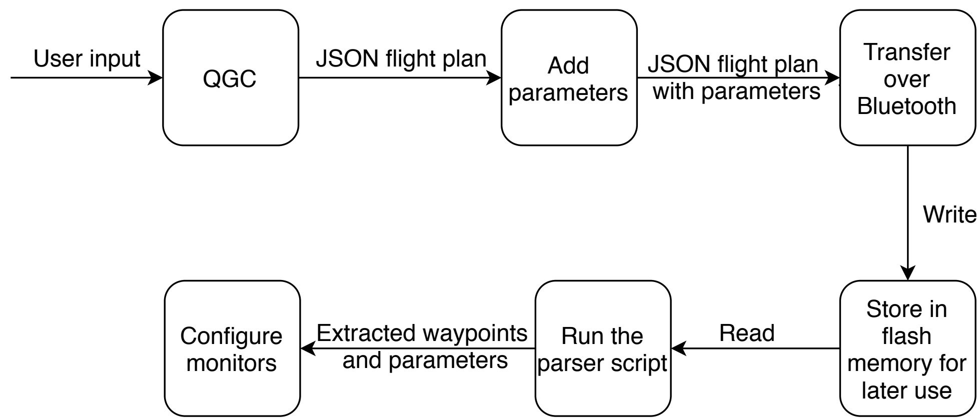


Figure 5.5: Flow of steps involved in configuring monitors

An autonomous flight requires a different set of rules for different legs of the flight. Different tolerance limits are required for different flights. Therefore, unlike phase one where the virtual cage and threshold values were constants, there should be a provision to alter the tolerance limits of the monitors, that is, the hardware monitors need to be configurable. Figure 5.5 captures the steps involved in configuring the monitors.

5.4.1.1 Adding Parameters in Flight Plan

Reconfigurability is achieved by adding the tolerance for the monitors as parameters in the JSON formatted flight plan. This enables setting different tolerance limits for different flights based on the weather and capabilities of the UAV. The usage of all the parameters is discussed in detail in Chapter 6. Listing 5.1 enumerates some of the parameters.

Listing 5.1: First few parameters in the flight plan

```
1       "corridorWidth": 2500,
2       "takeoffWidth": 800,
3       "takeoffSpeed": 3000,
4       "takeoffSpeedTolerance": 500,
5       "takeoffSpeedThreshold": 250,
6       "runtimeLowerSpeedTolerance": 1500,
7       "runtimeUpperSpeedTolerance": 500,
8       "hardThresholdTime": 60,
9       "softThresholdTime": 40,
10      . . .
```

5.4.1.2 Flight Plan Transfer and Storage

The flight plan for an airplane can be wirelessly uploaded through Wi-Fi or Bluetooth. Wireless transfer of files removes the need for external connectors. Thus, Bluetooth was chosen as the interface between the FPGA and an external device.

A Digilent BT2 Bluetooth Pmod is used to provide the Bluetooth interface for the FPGA [7]. This device receives serial data and transmits the data through a UART. Digilent provides Pmod Bluetooth IP cores in Xilinx Vivado, which is connected to the Pmod pins on one side and a real-time processor on the other [32]. The flight plan is transferred serially over Bluetooth. There are existing applications that can be used for this purpose: GTKTerm in Linux or RealTerm in Windows [17, 37]. While transferring data serially, it is important to

indicate the end of the file to notify the UI application that the entire flight plan has been transferred. This is achieved by adding the sequence of characters `__END__` at the tail end of the file.

Storing a received file in flash memory allows uploading flight plans ahead of the scheduled flight departure as well as reusing flight plans. The AXI_QUAD_SPI IP core available in the Xilinx Vivado tool can be used to write and read data from the flash memory [6]. The file size is stored in the first four bytes, and file contents are stored from the fifth byte. The size of the file is required while reading back the file. The file is run through a parser after it is received. A parser script was developed using the JSMN JSON parser package to extract the waypoints and parameters [20]. All the other information in the flight plan is ignored. If there are no errors found while parsing the file, a message is sent through Bluetooth indicating that the file is parsed successfully. If not, an error message is sent.

The monitoring system needs to be active only during the flight. The start of the flight is indicated by a `__START__` message. This allows the system to prepare itself for monitoring the flight. The UAV is not allowed to take off without receiving the start message. When a start message is received, the file is read from the flash device and parsed. The user is notified if the file does not exist or if the parsing is unsuccessful. If the parsing is successful, then a `Ready to Start` message is sent.

5.4.2 User Interface Processor as a Master

In a multiprocessor system, where each processor interacts with the other, it is essential to have all the processors synchronized. For example, it is undesirable to have one processor transmitting data when the other is going through its initialization. Therefore, a synchronization mechanism was implemented to ensure that the simulation begins after all

the processors have completed their initialization phase. The UIP acts as a master by sending start commands to other processors. The IOP and MP wait until they receive the start command. The UIP is the ideal choice for acting as the master because it receives the start command from the user. It can also prevent the flight from starting if there are issues with the flight plan. Algorithms 3, 4, and 5 capture the synchronization mechanism.

Algorithm 3 UIP - steps for synchronization

```
 1: Receive flight path through Pmod Bluetooth
 2: if __START__ command is received from user then
 3:     Send START_MON_PROC signal to MP and wait for a response
 4:     if MP responds with READY signal then
 5:         Write the number of parameters followed by the encoded parameters in Mailbox
 6:         if PARAM_ACK is received then
 7:             Write the first two waypoints
 8:             if WAYPT_ACK is received then
 9:                 Send START_IO_PROC signal to IOP
10:             end if
11:         end if
12:     end if
13: end if
14: while forever do
15:     if NEXT_WAYPOINT request received from MP then
16:         if Last leg then
17:             Send LAST_LEG message
18:         else
19:             Send next waypoint
20:         end if
21:     end if
22: end while
```

Algorithm 4 IOP - steps for synchronization

```
1: if START_IO_PROC signal received then
2:     while forever do
3:         Transmit MAVLink packets to Pixhawk
4:         Write the GPS data in Mailbox FIFO
5:     end while
6: end if
```

Algorithm 5 MP - steps for synchronization

```
1: if START_MON_PROC received from the UIP then
2:     Send READY signal
3:     if Parameters received then
4:         Read the number of parameters from the Mailbox FIFO
5:         Read the parameters and store them in local memory
6:         Send PARAM_ACK
7:         if Waypoints received then
8:             Send WAYPT_ACK
9:         end if
10:    end if
11: end if
12: while forever do
13:    if New flight leg then
14:        Send waypoint request (WAYPT_RQST) to the UIP
15:    end if
16:    if Next waypoint (NEXT_WAYPT) received from UIP then
17:        Store the next waypoint in local memory
18:    end if
19:    if New GPS data received from IOP then
20:        Invoke monitors
21:        if Violation detected then
22:            Send VIOLATION message to IOP
23:        end if
24:    end if
25: end while
```

5.5 I/O Processor

The IOP is responsible for tapping the sensor data. One control cycle of the IOP is captured in Algorithm 6. The latency added by the IOP leads to unstable HITL simulation. To reduce the latency, the MP communicates to the IOP only when there is a violation. This avoids unnecessary overhead involved in decoding a `NO VIOLATION` message every cycle.

The recovery control used in this demonstration is a `LAND` command which forces the UAV to stop its flight and simply land. The land recovery function is effective in scenarios like surveying and drone photography. However, it may not be the best option in operations like

Algorithm 6 IOP task

1: Read MAVLink packet from the UART buffer
2: Forward the packet to the Pixhawk through Pmod pins
3: Check the message or command ID in the MAVLink packet
4: **if** the packet corresponds to a sensor data **then**
5: Decode the sensor data values from the packet
6: Write the data to Mailbox FIFOs in an encoded format
7: **end if**
8: Check for a reply from the MP
9: **if** the MP indicates a violation **then**
10: Execute the RCF
11: **end if**

drone delivery or pipeline patrol. Other recovery actions are beyond the scope of this work and hence were not implemented. This project focuses on detecting a violation from expected behavior in the future (5 cycles ahead) and demonstrating that it is possible to recover before something unexpected happens. The recovery action can be customized depending on the operation.

5.6 Monitor Processor

A MicroBlaze soft processor is used to implement the MP. It sits in the center of the system, interacting with the UIP, IOP, obstacle detection application, and hardware monitors. At a high level, the MP performs the following tasks:

1. Receive parameters and waypoints from the UIP.

2. Set the parameter values in the input registers of the monitors.

3. Receive the sensor data from the IOP and write it to different monitors.

4. Receive distance to an obstacle from the obstacle detection application.

5. Invoke the right monitors depending on the phase of the flight.

6. Read the results of the monitors.

7. Communicate to the IOP if there is a violation.

5.6.1 Identifying Phases of Flight

Different sets of monitors must be invoked during different phases of the flight. For example, it is critical to closely monitor the ground speed during the cruise and the vertical speed during takeoff and landing. The MP is responsible for invoking the correct set of monitors. In order to do that, the MP needs to know the current phase of the flight. One way to identify the phase of the flight is to observe the commands sent between the Pixhawk autopilot and the Gazebo simulation environment. However, this is not the correct approach because the monitoring system should be independent of the application software as well as the autopilot software. The MP utilizes the GPS location data to determine the current phase of the flight.

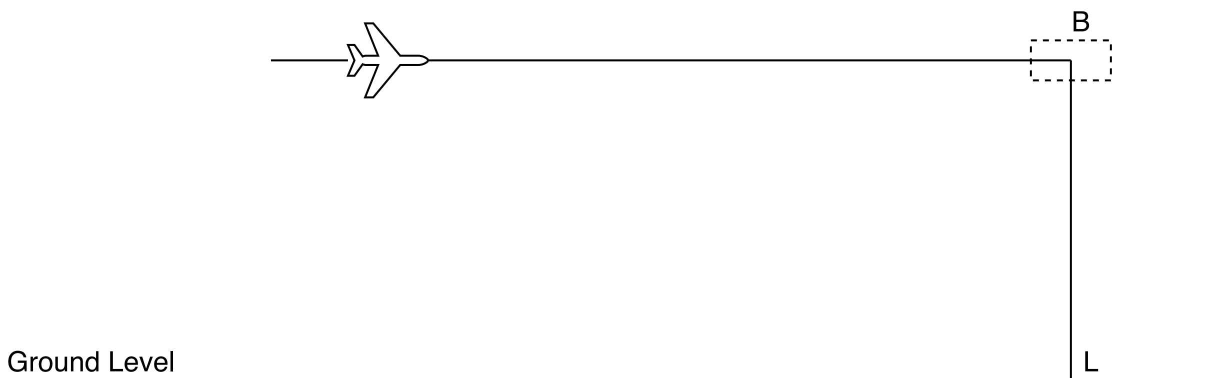


Figure 5.6: UAV cruising through its last leg

The example illustrated in Figure 5.6 helps to understand this. Consider a UAV approaching waypoint `B`, the point at which it must begin its landing phase. Ideally, a land command is executed when the UAV reaches waypoint `B`, and the descent begins. Assume that the

UAV continues flying without landing due to a bug. If the monitoring system waited for the land command, which never arrived, it would have continued to enforce the set of behaviors expected during the cruise. However, if the MP senses the distance to point `B`, it can switch to the land monitors when the UAV enters the safe landing zone (indicated by the dotted rectangle). This ensures that the set of behaviors expected during land are enforced. If the UAV continued to cruise without landing, an RCF would be executed.

The methods used to identify takeoff, cruise, and landing phases are given below.

5.6.1.1 Identifying Takeoff

After the initialization sequence is complete, the MP continuously checks for the altitude value and vertical speed in the GPS packets. If the altitude value keeps strictly increasing for N frames (configurable) and the speed is positive, then it concludes that the UAV is taking off. The takeoff monitors are then called into action every frame.

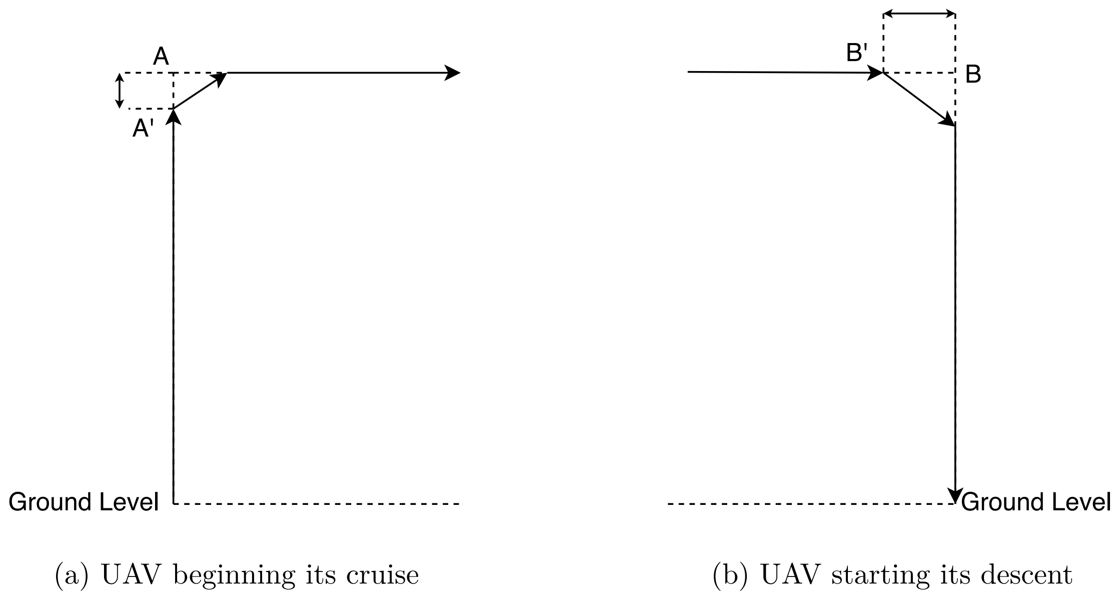
It was observed that even in a simulation environment, GPS sensor data has some fluctuations. This is especially true for altitude values. For example, when the UAV is stationary, two consecutive packets can report slight variations in altitude. Thus, the model should take into account such fluctuations in sensor data.

5.6.1.2 Identifying Cruise

As the UAV takes off, the MP continuously checks if it has reached the *cruise_begin_altitude*. If the UAV has risen to that altitude, then the runtime monitors are called into action every frame. The *cruise_begin_altitude* is given by:

$$cruise_begin_altitude = takeoff_altitude - takeoff_tolerance$$

It is observed that the UAV climbs straight up to a certain altitude slightly lower than the altitude of the first waypoint, then begins its cruise while still climbing up to the required altitude. This is illustrated in Figure 5.7a. Though the altitude of the first waypoint is `A`, the UAV begins cruise at altitude `A`′. The difference `A - A`′ should be less than *takeoff_tolerance*, which is configurable.



(a) UAV beginning its cruise

(b) UAV starting its descent

Figure 5.7: Identifying cruise and land flight phases

5.6.1.3 Identifying Landing

The necessary step required for identifying a switch to the landing phase is determining whether the UAV is in its last leg of the flight. The UIP supplies waypoint locations to the MP in a client-server relationship (discussed in detail in Section 5.8.1). When the MP requests for the next waypoint (`NXT_WAYPOINT` request) after beginning the final leg, the UIP replies with a `LAST_LEG` message. When this message is received, the MP continually calculates the distance between the UAV and the last waypoint (the point where the descent begins). If the distance is less than the parameter *safe_land_zone_radius*, then landing

monitors are called into action. In Figure 5.7b, though `B` is the last waypoint, the UAV begins its descent at point `B′`. The difference `B - B′` should be less than *safe_land_zone_radius*.

5.6.2 Monitor Blocks Integration

Monitors are connected to the MP as memory-mapped slaves. The interface between the monitor blocks and the processor is similar to the one used in the first phase (Section 4.4.2). Xilinx's AXI buses replace the Intel's Avalon buses, and AXI Interconnect is used in place of the Platform Designer interconnect. Figure 5.8 depicts the template of a monitor.

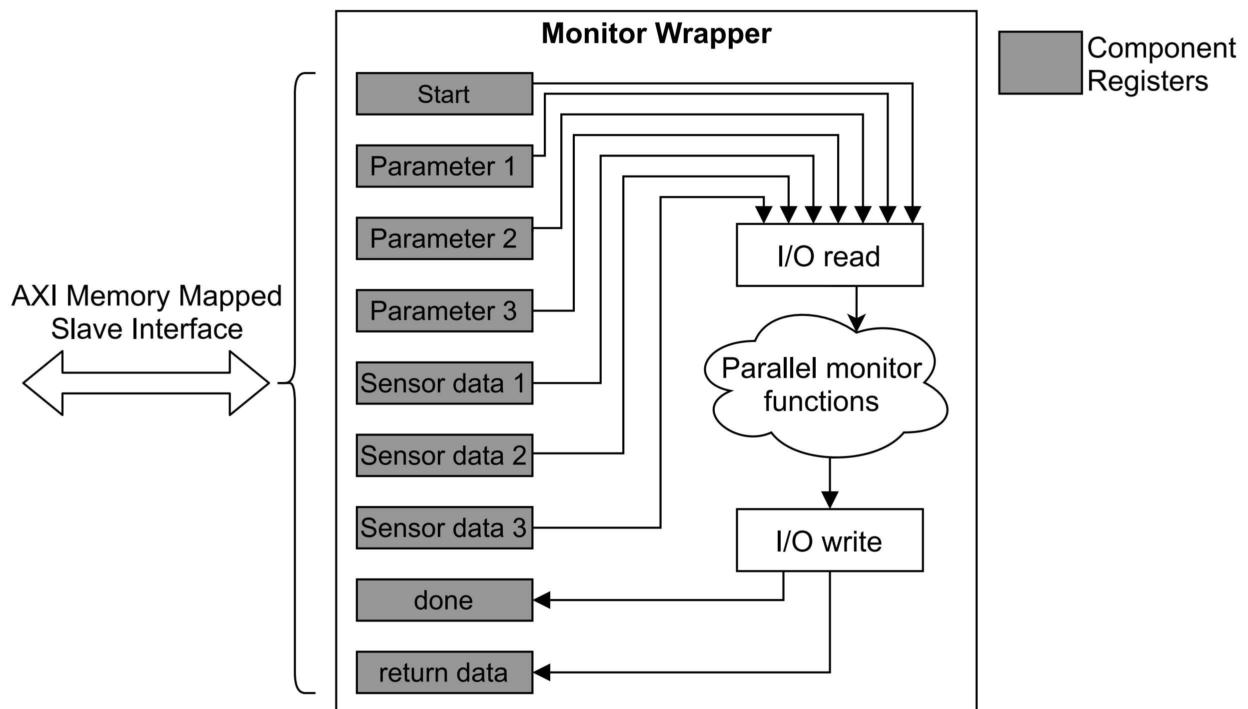


Figure 5.8: Template of a monitor

In addition to the sensor data, the parameters must be written to the registers. The parameter values do not change during runtime. Therefore, they are written only during the initialization stage. Algorithm 7 captures the interaction between the MP and monitors.

Algorithm 7 Invoking the monitors

1: Initialize: Write the tolerance values to the parameter registers
2: **if** new sensor data received **then**
3:     Wait for ready signal
4:     Load sensor data into monitors
5:     Start the monitors
6:     Wait for monitor completion (indicated by setting the done register)
7:     Read the results
8: **end if**

5.7 Obstacle Detection and Collision Avoidance

It is reported that "Drones with obstacle detection and collision avoidance sensors are becoming more prevalent in both the consumer and professional sectors" [28]. The directions in which obstacles can be detected varies from one UAV to the other and it depends on the number and position of sensors. Different obstacle detection sensors are available in the market: stereo vision using two cameras, monocular vision [55], ultrasonic, LIDAR, and others.

5.7.1 Simulating the Obstacle

Obstacle detection is a black box for the RTA system, which implies that runtime verification relies only on the output of the sensor and not on the method used for sensing the obstacle. The general methodology during the development phase of any component in the aircraft is to simulate other components that interact with the unit under test. This is done either by running automated scripts or setting inputs manually. Inspired by this approach, the obstacle detection sensor is simulated.

The following steps are used to simulate the obstacle detection sensor:

1. Create a test flight plan. This will be the flight plan followed by the UAV in the HITL

simulation with runtime monitors in action.

2. Create other flight plans that interfere or intersect the test flight plan. These are the flight plans for the obstacles.

3. Execute HITL simulation with the flight plans created in step 2 without the safety monitors.

4. Record the flight data (GPS positions and speed) for the obstacle flights.

5. Store the obstacle flight data in a table in the obstacle detection application (executed in one of the ARM Cortex-A53 processors).

6. Choose an obstacle and replay the flight data while simulating the test flight plan.

7. Calculate the distance to the obstacle and the rate of convergence in the obstacle detection application using the current GPS position received from the IOP.

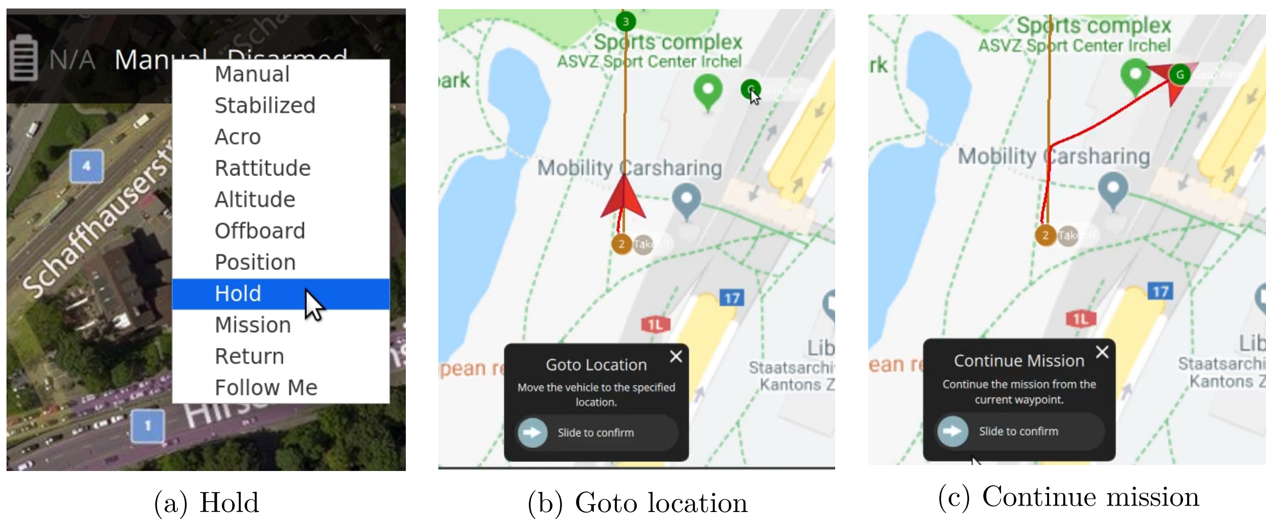
8. Send the distance and rate of convergence to the MP.

The MP invokes the collision avoidance monitor when obstacle details are received from the obstacle detection application. In contrast, other monitors are invoked when GPS data is received.

5.7.2 Simulating Collision Avoidance

Typically, a collision avoidance application receives data from the obstacle detection sensors and executes a complex function to determine a new path for the UAV to avoid the obstacle. Sometimes a human is also in the loop to help the system make a better decision. However, for the demonstration in this work, such an application is not used. Instead, the operator

takes the decision and sends commands to the autopilot from QGC. Two commands are used to evade obstacles: `HOLD` and `REPOSITION`. The former commands the UAV to hold its current position, whereas the latter directs the UAV to a new waypoint. Once the obstacle is safely avoided, the `CONTINUE_MISSION` command can be used to get the UAV back on its way. Figure 5.9 shows screenshots of commands sent from the QGC application.



(a) Hold

(b) Goto location

(c) Continue mission

Figure 5.9: Commands sent from QGC

The IOP listens to commands and messages exchanged between the Gazebo or QGC and Pixhawk autopilot. When any of the `HOLD`, `REPOSITION`, or `CONTINUE_MISSION` commands are detected, the IOP communicates it to the MP. This replicates a system where the collision avoidance application sends its decision to the RTA system. Thus, the MP assumes that the commands were sent to avoid an obstacle. The old set of monitors guarding against the old flight plan is no longer valid and the monitors must be adjusted based on the command.

5.7.2.1 Adjusting the Monitors to a HOLD Command

When a HOLD command is sent, the monitors should ensure that the UAV holds its position. This is achieved by enforcing a small 3D virtual cage centered at the UAV's current position. In Figure 5.10, UAV traveling from A to B is held at point H after receiving a HOLD command. Due to momentum, the UAV cannot stop immediately. Thus, the hold monitors are engaged after a certain configurable tolerance time. When the CONTINUE_MISSION command is received, the MP reverts to the old set of monitors.

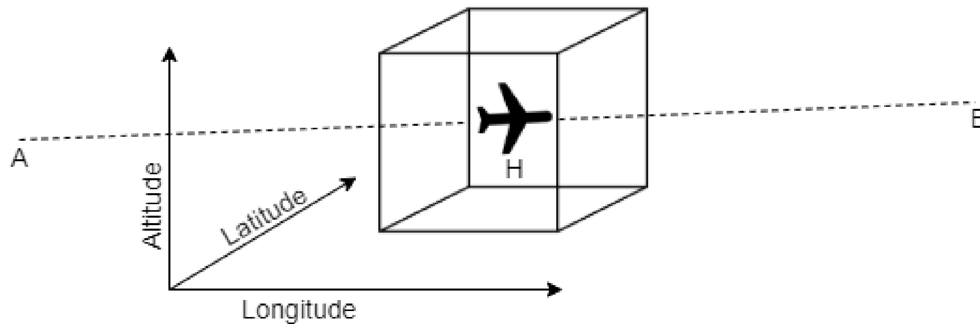
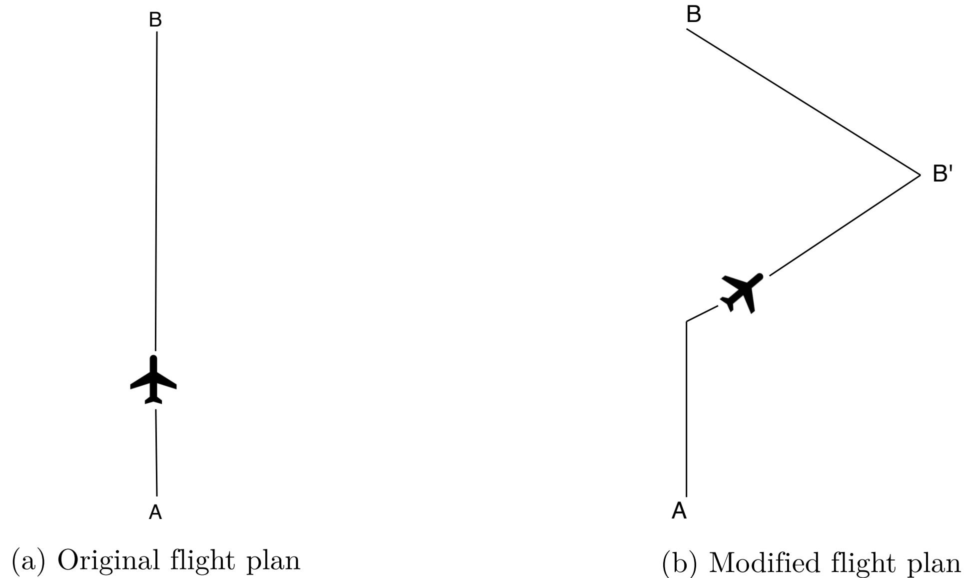


Figure 5.10: Virtual cage imposed on the drone

5.7.2.2 Adjusting the Monitors to a REPOSITION Command

There is a deviation from the original flight plan when a command to reposition is received, as illustrated in Figure 5.11. The monitors must adjust to this change. The command to reposition, MAV_CMD_DO_REPOSITION, includes the GPS coordinates for the new waypoint [9]. This information is passed on to the MP from the IOP, and new runtime monitors are constructed. When the CONTINUE_MISSION command is sent, the UAV flies towards the next waypoint in the original flight plan (Figure 5.11b).



(a) Original flight plan

(b) Modified flight plan

Figure 5.11: Change in flight plan to avoid an obstacle

5.8 Inter-processor Communication

All four processors in the system need to communicate with each other continuously. The MP needs to receive GPS data from the IOP, waypoints from the UIP, and the distance to obstacle from the obstacle detection application, which in turn requires the current position information from the IOP. The MP needs to instruct the IOP to execute an RCF if any guard is triggered. Thus, robust inter-processor communication is critical to the design.

5.8.1 User Interface Processor as a Server

The UIP needs to share the waypoints and parameters essential for instancing the monitors with the MP. The initial attempt to share data involved allocating a Block RAM (BRAM) for this purpose. The BRAM was shared directly between the processors. The UIP wrote the data to BRAM and sent a signal to the MP indicating that the data was available. However, this approach is not scalable. There are two possible ways to share data when multiple soft

processors are added: one BRAM shared among all the processors or one BRAM for each soft processor. The correctness will be harder to establish when more than one MicroBlaze has read/write access to shared data memory. This difficulty is because of the introduction of a design similar to the thread programming model. Having multiple BRAMs increases the area of the hardware unnecessarily. Thus, both these solutions are undesirable, and this approach was dropped.

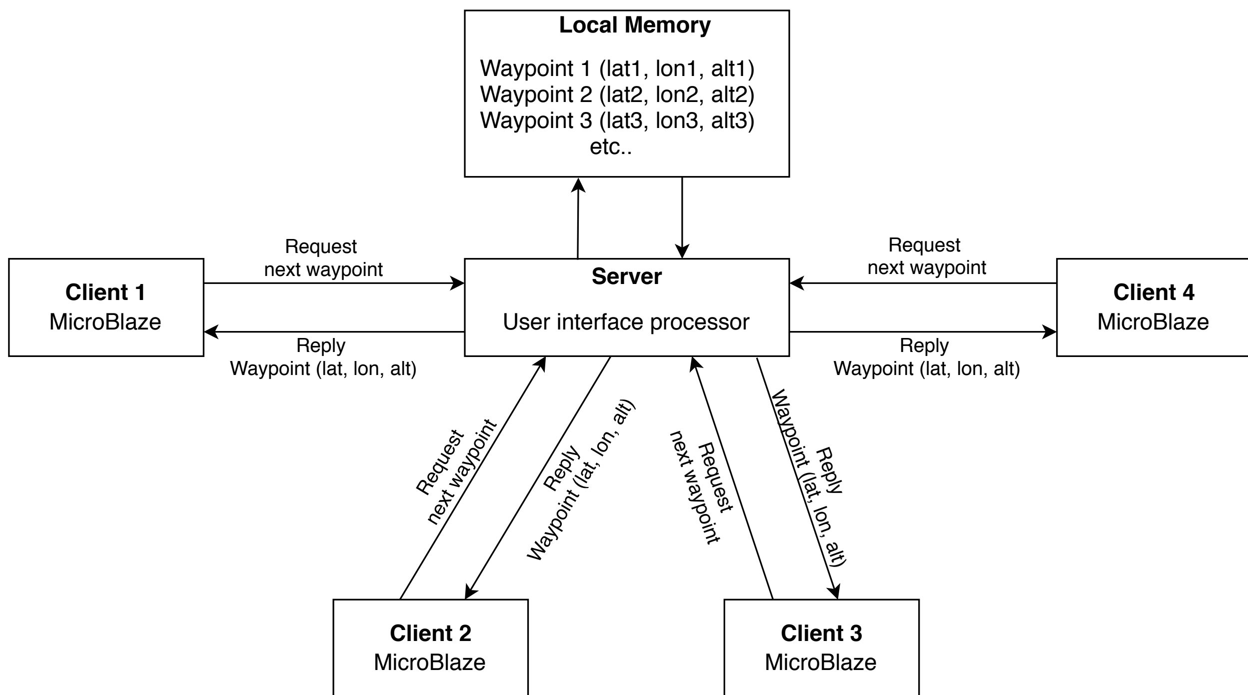


Figure 5.12: Client-server interaction between UIP and MPs

It was decided that a client-server interaction, shown in Figure 5.12, with message-based APIs would be the best approach. The UIP will act as the server, and the soft processors will act as clients, requesting waypoint and parameters from the server. The soft processors request the next waypoint at the beginning of each segment. Once a segment is completed, the previous waypoint information is replaced with the next one. This ensures that only the required information is stored in the memory, and the old data is overwritten. This is particularly useful when the flight plan is lengthy (many waypoints). This approach is

scalable as the server can handle multiple clients. The frequency of requests depends upon the length of the flight segments. If the flight segments are long, the requests are less frequent and vice versa. It is safe to assume that the time interval between two requests from a client will be at least in the order of tens of seconds since flight segments are generally not shorter than that. The latency in serving the requests will be in the order of microseconds. Thus, one server can handle many clients easily. In the event that the number of clients grows beyond the capacity of a server, multiple servers can be used.

5.8.2 Mailbox – the Bridge between the Processors

A Mailbox IP core is utilized to support the client-server relationship discussed above and to enable effective communication among all the processors. Xilinx states that: "In a multiprocessor environment, the processors need to communicate data with each other. The easiest method is to set up inter-processor communication through a mailbox" [22]. The reasons to select a Mailbox are:

1. Mailbox supports full-duplex communication with bidirectional FIFO between processors, as shown in Figure 5.13.
2. Depth of FIFO and the type of memory (Distributed RAM, Block RAM, or Ultra RAM) are configurable.
3. Multiple Mailboxes can be connected to a single processor, as shown in Figure 5.2.
4. Interrupts are not necessary. Mailbox works in polling mode.
5. It supports both AXI4-Lite and AXI4-Stream bus interfaces.
6. The Mailbox driver supports non-blocking read and writes through the AXI4-Lite interface. AXI4-Stream has its own non-blocking read and write commands.

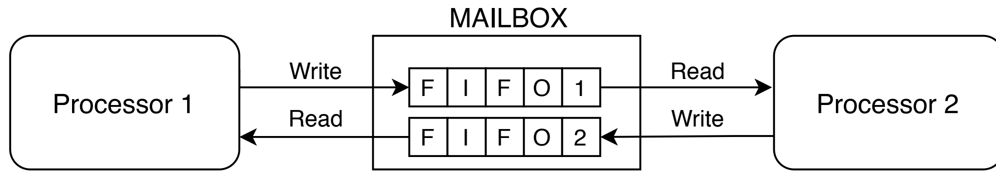


Figure 5.13: Mailbox between processors

Looking at Table 2.1 of the Mailbox IP documentation [22], the typical latencies and throughput with the AXI4-Stream interface are only slightly less than with AXI4-Lite. Consecutive `get` or `put` instructions using different registers, without stalling, help achieve the minimum latencies and maximum throughput. The Arm cores do not support the AXI4-Stream directly as it is not a bus. A direct memory access (DMA) block would be needed as an intermediary leading to additional software overheads. AXI4-Stream becomes efficient when both the sender and receiver use it. The MicroBlazes already need an AXI4-Lite bus in order to communicate with monitors, so using the AXI4-Stream interface does not avoid allocating bus logic resources. Therefore, the AXI4-Lite interface was preferred over the AXI4-Stream interface.

Chapter 6

Safety Monitors

Multiple independent parallel monitors synthesized from LTL formulas are implemented in the FPGA hardware. As mentioned earlier, a different set of monitors are active during different phases of flight. Sections 6.1, 6.2 and 6.3 explain the LTL formulas and algorithms used to calculate the AP values for takeoff monitors, runtime monitors and land monitors respectively. The monitors fall into three different categories:

1. **Flight plan conformance monitors**

   These monitors ensure that the UAV strictly follows the filed flight plan. The autopilot software in drones does an excellent job of closely following these points. However, if the software is corrupted or if there is a malicious attack to hijack the drone, it is essential to identify the unexpected deviation from the flight plan and trigger an RCF.

2. **Stabilized flight monitors**

   Following the flight plan strictly is not enough. The flight needs to be stable. For example, if there is a heavy opposing wind, the UAV could be strictly following the plan but progressing very slow. Onboard batteries power most UAVs, and if its flight time exceeds the expected duration, it could deplete the battery. Alternatively, the UAV could run into other UAVs if it flies too fast. Thus, it is vital to monitor variations in speed.

3. **Airborne collision avoidance monitor**

   This monitor guards against the failure of the collision avoidance system to avoid other

air vehicles. Incorrect actions could be either because of a software bug or the presence of many nearby obstacles. In the former case, the collision avoidance system miscalculates. In the latter case, it may not be able to find the right path to evade all obstacles. In these situations, the best solution may be to trigger an RCF.

6.1 Takeoff Monitors

Monitoring begins as soon as the UAV is powered on, and it is ready to takeoff. Six monitors are called into action during takeoff. They fall into two different categories:

6.1.1 Takeoff Position Monitors

As the UAV takes off, it should ideally climb straight up from the starting point on the ground to reach the target altitude. However, in practice, the UAV sways a little as it ascends. The takeoff position monitors ensure that the UAV does not significantly deviate from the ideal line. Visualizing a virtual cage helps in understanding these monitors. This is similar to the virtual cage in the initial demonstration but different in magnitude. In the first phase, the cage was a large 3D cuboidal space. However, here the virtual cage, depicted in Figure 6.1, resembles a corridor inside which the drone can fly.

`X` and `Y` represent the latitude and longitude coordinates of the takeoff point, respectively. `G` is the altitude above mean sea level at the ground and `A` is the target altitude. The UAV should ideally follow the dark center line while taking off. The four dotted lines form the four vertical edges of the virtual cage. The position monitors ensure that the UAV stays within the four vertical virtual fences while taking off. Given takeoff point (*takeoff_lat*, *takeoff_lon*), and the parameter takeoff position threshold (*takeoff_pos_threshold*), the *safe_takeoff_region*

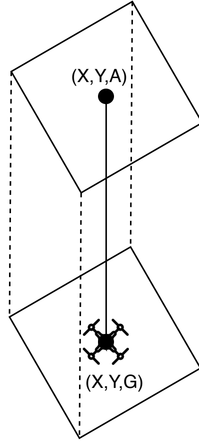


Figure 6.1: Safe takeoff zone

is the polygon formed by the four lines:

$$\begin{aligned} lower_lat &= takeoff_lat - takeoff_pos_threshold \\ upper_lat &= takeoff_lat + takeoff_pos_threshold \\ lower_lon &= takeoff_lon - takeoff_pos_threshold \\ upper_lon &= takeoff_lon + takeoff_pos_threshold \end{aligned}$$

There are four position monitors, each one entrusted to guard against breaching one of the four fences: minimum latitude, maximum latitude, minimum longitude and maximum longitude. All four monitors are similar in structure, and thus it is enough to understand one. The `within_maximum_latitude` monitor, which guards the maximum latitude fence, will be used as an example. The LTL formula used to capture the expected behavior is given below.

$$nearing_fence \Rightarrow (can_slowdown\, U\ \neg nearing_fence) \tag{6.1}$$

The AP values for the `within_maximum_latitude` are determined from the current latitude position (lat) and velocity in latitude axis (lat_speed):

$$\begin{aligned} nearing_fence &= (lat_speed > 0) \\ can_slowdown &= (lat_speed < UP_LAT_LIMIT - lat) \end{aligned}$$

The latitude and longitude values received in the GPS packets have the unit ($degrees \cdot 10^7$). The velocity must be in ($degrees \cdot 10^7$)/$second$. However, the velocity values received in the GPS packet are in centimeters per second. To avoid mismatch of units, the velocity is calculated on the fly as the difference between the latitude coordinate values in two consecutive packets.

$$velocity = current_lat_position - previous_lat_position$$

The UAV can sway a little, but it should not be moving too fast in the latitude or longitude planes while taking off. Thus, whenever the velocity is greater than 0, the monitor checks whether the UAV will be able to slow down within the boundary at the current speed. In the simulation environment, the speed value can change from positive to negative and vice versa in two consecutive packets without receiving a zero velocity. This is because the GPS packets are received every 200ms. In order to determine if the drone is not moving towards the maximum latitude fence, it is necessary to check if the drone has zero or negative velocity along the latitude axis.

6.1.2 Takeoff Speed Monitors

The climb speed or the vertical speed for the UAV can be programmed from the QGC. The UAV's capability and the prevailing weather conditions are considered while setting a value for the climb speed. Though the UAV is expected to takeoff at this velocity, small deviations should be tolerated. Given the climb speed (set_climb_speed) and the parameters

vertical speed upper tolerance ($high_ver_speed_tolerance$) and vertical speed lower tolerance ($low_ver_speed_tolerance$), the safe vertical speed range ($safe_ver_speed_range$) is defined as:

$$low_takeoff_speed_limit \leq safe_ver_speed_range \leq high_takeoff_speed_limit$$

where

$$\begin{aligned} low_takeoff_speed_limit &= set_climb_speed - low_ver_speed_tolerance \\ high_takeoff_speed_limit &= set_climb_speed + high_ver_speed_tolerance \end{aligned}$$

The takeoff speed monitors are entrusted to ensure that the UAV's vertical speed remains within the safe vertical speed range.

6.1.2.1 Takeoff Under Speed Monitor

Strictly enforcing the velocity range is not a practical approach since not all UAV flights are smooth. There will be some factors that slow down the takeoff. Thus, it is important to allow the vertical velocity to drop below the $safe_ver_speed_range$ but not for too long. This monitor guards against the vertical speed dropping below the $safe_ver_speed_range$ and not recovering back to the $safe_ver_speed_range$ for a sufficiently long period. The LTL formula is given as:

$$\begin{aligned} &below_safe_ver_speed_range \Rightarrow (ver_speed_increasing \\ &\qquad U \; in_safe_ver_speed_range) \end{aligned} \tag{6.2}$$

The AP values are determined using current vertical speed (ver_speed):

$$
\begin{aligned}
in_safe_ver_speed_range = (ver_speed \geq low_takeoff_speed_limit) \\
\wedge\ (ver_speed \leq high_takeoff_speed_limit)
\end{aligned}
$$

$$
below_safe_ver_speed_range = (ver_speed < low_takeoff_speed_limit)
$$

$$
ver_speed_increasing = is_ver_speed_increasing()
$$

The *is_ver_speed_increasing*() function keeps track of how long the velocity has remained under the *safe_ver_speed_range*, as summarized in Algorithm 8. The counter value is compared against two parameters, soft and hard takeoff speed threshold times, to determine if the speed has stayed below the safe range for too long. Until the counter reaches *soft_takeoff_speed_threshold*, the velocity can drop. However, the velocity cannot be negative. In other words, the UAV cannot move downwards. When the counter reaches the *soft_takeoff_speed_threshold* the UAV should start making efforts to bring the vertical speed to the safe range. A flag is raised if speed continues to drop or remain the same, or the vertical speed increases but does not reach the safe zone before the counter value reaches the *hard_takeoff_speed_threshold*.

Algorithm 8 *is_vert_speed_increasing*() function

```
1:  Initialize static variables counter and previous_speed
2:  if counter value greater than hard_takeoff_speed_threshold then
3:      return false
4:  else
5:      if counter value greater than soft_takeoff_speed_threshold then
6:          if current_speed less than previous_speed then
7:              return false
8:          end if
9:      end if
10: end if
11: Assign current_speed to previous_speed
12: Increment counter
13: return true
```

6.1.2.2 Takeoff Overspeed Monitor

This monitor guards against the vertical speed increasing above the $safe_ver_speed_range$ and not recovering for a sufficiently long period. The LTL formula is:

$$above_safe_ver_speed_range \Rightarrow (ver_speed_decreasing \\ U\ in_safe_ver_speed_range) \quad (6.3)$$

The AP values are determined using current vertical speed (ver_speed):

$$in_safe_ver_speed_range = (ver_speed \geq low_takeoff_speed_limit) \\ \wedge (ver_speed \leq high_takeoff_speed_limit)$$

$$above_safe_ver_speed_range = (ver_speed > high_takeoff_speed_limit)$$

$$ver_speed_decreasing = is_ver_speed_decreasing()$$

The $is_ver_speed_decreasing$ function is similar to the $is_ver_speed_increasing$. It keeps track of how long the speed has been above the safe range and reports if the UAV is not making efforts to reduce its speed.

6.2 Runtime Monitors

When the takeoff is completed, a different set of monitors is called into action. These monitors, termed runtime monitors, observe different behaviors as the UAV cruises from the takeoff point to its destination. Most threats faced by the UAV occur during cruise flight. Thus, it is necessary to have multiple parallel monitors that guard against different potential failures.

6.2.1 Runtime Position Monitor

This monitor falls into the flight plan conformance category. It ensures that the drone does not significantly deviate from its expected latitude and longitude coordinate positions.

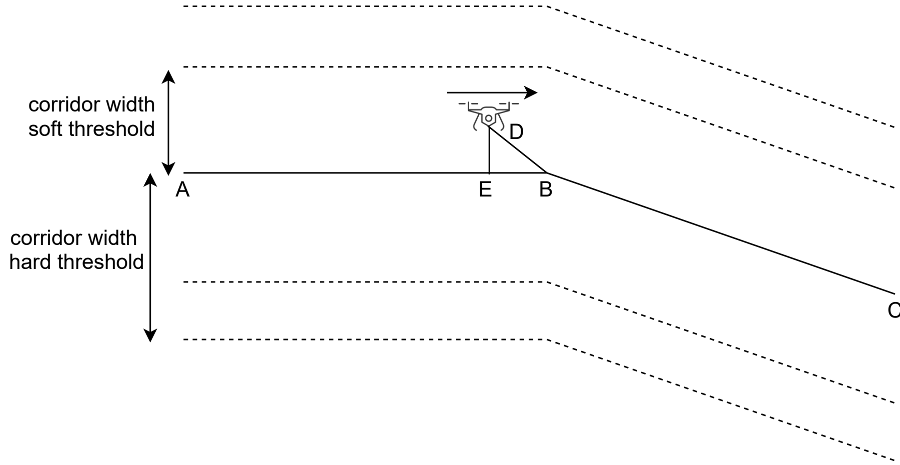


Figure 6.2: Flying corridor

Figure 6.2 explains the operation of the runtime position monitor. `A`, `B`, and `C` are the waypoints. `ABC` is the expected flight path where `AB` is the first segment, and `BC` is the second segment. Let us assume that the drone is currently at position `D`. The arrow indicates the direction of flight. `DE`, the perpendicular distance from the drone to the segment `AB`, is the deviation from the expected flight path. If the distance is less than the parameter *corridor_width_soft_threshold* then the UAV is well within the safe limits. If not, the UAV needs to be closely monitored. The LTL formula that captures this requirement is given below:

$$outside_soft_boundary \Rightarrow (can_slowdown\, U\, \neg outside_soft_boundary) \tag{6.4}$$

The APs are calculated using current distance (*current_dist*) and predicted future distance (*future_dist*):

$$outside_soft_boundary = (current_dist > corridor_width_soft_threshold)$$
$$can_slowdown = (future_dist < corridor_width_hard_threshold)$$

The latitude and longitude values from the GPS packets are used to calculate *current_dist*. The velocities in the latitude (*lat_speed*) and longitude axes (*lon_speed*) are calculated as the difference between latitude and longitude values in consecutive GPS packets. Then, the future position can be predicted using the current position and velocity:

$$future_lat = current_lat + lat_speed$$
$$future_lon = current_lon + lon_speed$$

A guard is triggered if the predicted future distance (*future_dist*) calculated using the above values is greater than the parameter *corridor_width_hard_threshold*.

6.2.1.1 Identifying Segment Switch

The runtime position monitor is tasked in identifying when the UAV completes one segment and moves to the next. As mentioned earlier, the MP requests the next waypoint from the UIP. When the waypoint is received, the MP passes the waypoint location to the runtime position monitor. If the next waypoint is available, the monitor needs to identify the segment closest to the UAV. In Figure 6.2, `C` is the next waypoint. The monitor calculates the shortest distance of the UAV from the segments AB and BC. The shortest distance can either be the perpendicular distance (like `DE`) or the distance to the start of the next segment (like `DB`). The UAV has moved on to the next segment if the distance to the next segment is less than the distance to the current segment. The *current_dist* and *future_dist* values used to calculate the APs are given by:

$$current_dist = (distance_from_BC < distance_from_AB) \ ? \ distance_from_BC : distance_from_AB \quad (6.5)$$

$$future_dist = (future_dist_from_BC < future_dist_from_AB) \ ? \ future_dist_from_BC : future_dist_from_AB \quad (6.6)$$

6.2.2 Runtime Speed Monitors

The runtime speed monitors fall into the "stabilized flight" category of monitors. The cruise speed for a fixed-wing UAV and hover speed for a rotary-wing UAV can be programmed through the QGC application and can be found in the flight plan, as shown in Listing 6.1.

Listing 6.1: Cruise speed and hover speed in flight plan

```
1  {
2      "fileType": "Plan",
3      "groundStation": "QGroundControl",
4      "mission": {
5          "cruiseSpeed": 15,
6          "firmwareType": 12,
7          "hoverSpeed": 5,
8          "items":
9     . . .
10 }
```

The cruise/hover speed is the expected ground speed of the UAV (speed of the UAV relative to the ground). Though the UAV normally maintains this speed, it is important to have some tolerance while monitoring. Given the ground speed (*set_ground_speed*) and the parameters runtime upper speed tolerance (*high_speed_tolerance*) and runtime lower speed tolerance (*low_speed_tolerance*), the *safe_ground_speed_range* is defined as:

$$rt_low_speed_thresh \leq safe_ground_speed_range \leq rt_high_speed_thresh$$

where

$$rt_low_speed_thresh = set_ground_speed - low_speed_tolerance$$

$$rt_high_speed_thresh = set_ground_speed + high_speed_tolerance$$

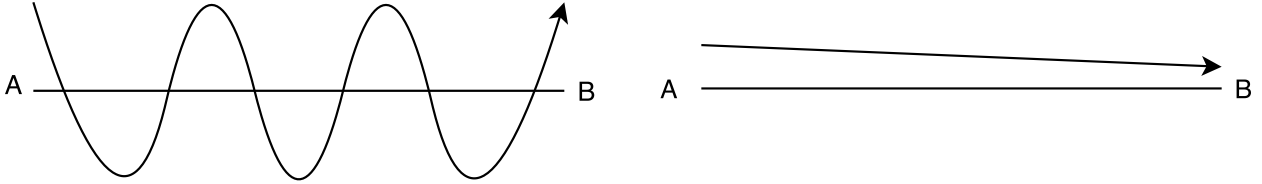
The runtime speed monitors ensure that the UAV's ground speed stays within the safe ground speed range.

During runtime, the current speed values are received in the GPS packet. Four different velocities are received in the GPS packets, as described in Table 6.1.

Field Name	**Type**	**Units**	**Description**
`vel`	uint16	cm/s	Ground speed of the UAV. It is always positive.
`vn`	int16	cm/s	Velocity in the north direction. It is positive when the UAV is traveling north and negative when the UAV is traveling south.
`ve`	int16	cm/s	Velocity in the east direction. It is positive when the UAV is traveling east and negative when the UAV is traveling west.
`vd`	int16	cm/s	Velocity in the down direction. It is positive when the UAV is descending and negative when the UAV is climbing.

Table 6.1: Velocity fields in the GPS packet

At first glance, it appears that the `vel` field in the GPS packet, which gives the ground speed, needs to be monitored. However, this field gives speed and not velocity. That is, it merely gives how fast the UAV is moving with respect to the ground, but it does not convey the direction in which the UAV is heading. To understand this better, consider the two cases shown in Figure 6.3. In both cases, the UAV is cruising at the set ground speed. However, the actual progress made between the two waypoints is more in case 2 than case 1. The UAV takes more time to reach the waypoint `B` in case 1 compared to case 2. Thus, simply monitoring the `vel` field is incorrect.



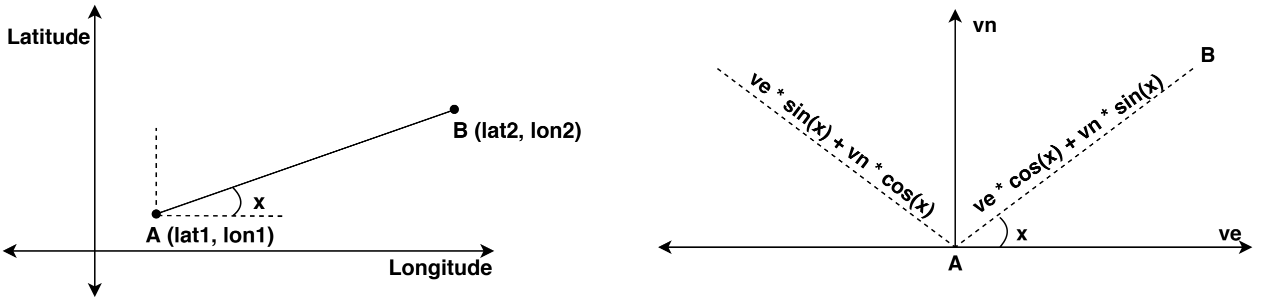
(a) Case 1: UAV traveling in a sinusoidal path from A to B. Even though the ground speed is equal to *set_ground_speed*, the overall progress made by UAV is less.

(b) Case 2: UAV traveling in a straight line path from A to B. If the UAV cruises at *set_ground_speed* then it will make sufficient progress along the expected path.

Figure 6.3: Different flight paths with same ground speed

The component of velocity along the straight line between the two waypoints (*vel_wp*) should be monitored. In order to obtain this velocity, the `vn` and `ve` fields are utilized. For small variations, the latitude and longitude axes form a Cartesian coordinate system. Thus, basic coordinate geometry can be used to calculate *vel_wp*.

Calculation of Speed



(a) Calculation of slope

(b) Calculation of velocity component

Figure 6.4: Calculation of speed

Let `A` (*lat1*, *lon1*) and `B` (*lat2*, *lon2*) be two consecutive waypoints along the flight path, as shown in Figure 6.4. The slope of the line `AB` is given by:

$$slope = \frac{(lat2 \text{ - } lat1)}{(lon2 \text{ - } lon1)}$$

The angle made by the line `AB` (indicated by `x` in Figure 6.4a) is given by:

$$angle = \tan^{-1}\ (slope)$$

The components of `vn` and `ve` along the line between waypoints are given by:

$$\begin{aligned} vn_comp &= vn * sin(angle) \\ ve_comp &= ve * cos(angle) \end{aligned}$$

The ground speed along the line between the waypoints is given by:

$$v_ground = vn_comp + ve_comp$$

Trigonometric and inverse trigonometric functions are expensive operations. Thus, it is necessary to avoid calculating slope, angle, cosine, and sine for every frame. This recalculation is avoided by passing a Boolean variable, which indicates a change in the flight segment, as an argument to the runtime speed monitor. This variable is `true` in the first frame after the drone completes a segment and switches to the next. Its value is `false` in the rest of the frames. Thus, the expensive operations are performed only once per flight leg.

6.2.2.1 Runtime Under Speed Monitor

The runtime under speed monitor ensures that the ground speed along the expected flight path does not stay below the *safe_ground_speed_range* for too long. Similar to the takeoff speed monitors, some grace period is allowed for the velocity to recover back to the safe ground speed range. The following LTL formula captures the requirement for the runtime under speed monitor.

$$below_safe_ground_speed_range \Rightarrow (ground_speed_increasing\, U \\ in_safe_ground_speed_range) \quad (6.7)$$

The APs are calculated from current ground speed ($ground_speed$):

$$in_safe_ground_speed_range = (ground_speed \geq rt_low_speed_thresh) \\ \wedge \ (ground_speed \leq rt_high_speed_thresh)$$

$$below_safe_ground_speed_range = (ground_speed < rt_low_speed_thresh)$$

$$ground_speed_increasing = is_ground_speed_increasing()$$

6.2.2.2 Runtime Overspeed Monitor

The runtime overspeed monitor ensures that the ground speed along the expected flight path does not stay above the $safe_ground_speed_range$ for too long. Some grace period is allowed for the velocity to recover back to the safe ground speed range. The following LTL formula captures the requirement for the runtime overspeed monitor.

$$above_safe_ground_speed_range \Rightarrow (ground_speed_decreasing\, U \\ in_safe_ground_speed_range) \quad (6.8)$$

The APs are calculated from current ground speed ($ground_speed$):

$$in_safe_ground_speed_range = (ground_speed \geq rt_low_speed_thresh) \\ \wedge (ground_speed \leq rt_high_speed_thresh)$$

$$above_safe_ground_speed_range = (ground_speed > rt_high_speed_thresh)$$

$$ground_speed_decreasing = is_ground_speed_decreasing()$$

6.2.2.3 Runtime Segment Switch Speed Monitor

Usually, the UAV has to change its course when it reaches a waypoint. If the change in direction is significant, the UAV slows down to alter its course. The velocity of the UAV could drop below the safe ground speed range while it switches from one flight segment to the next. The velocity picks up when the UAV completes the switch.

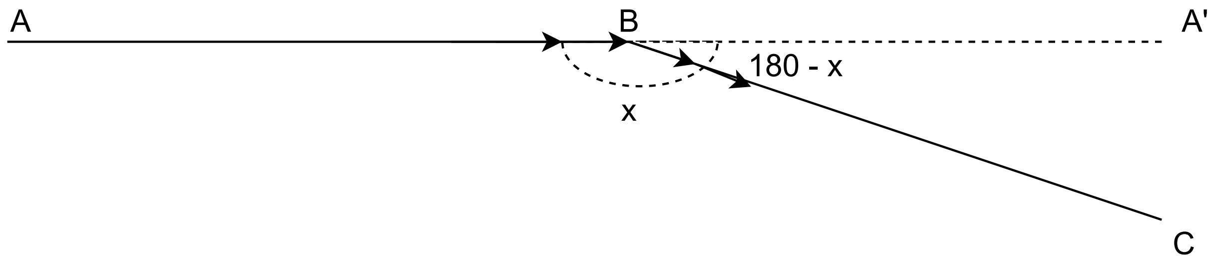


Figure 6.5: Change in direction during a flight

In Figure 6.5, `A`, `B` and, `C` are three consecutive waypoints and the arrowheads indicate the direction of flight. It can be observed that the course of the flight changes by a certain angle when the UAV reaches the waypoint `B` (indicated by `180-x` in the figure).

The segment switch speed monitor is called into action when the UAV begins a new segment, and is responsible for ensuring that the drone's velocity increases back to the safe ground speed range before a threshold time period. Though this monitor might look similar to the runtime under speed monitor, there is a distinction between the two. The soft and hard threshold time parameters for the under speed monitors (Section 6.1.2.1) are constant throughout the flight, whereas the thresholds for the segment switch monitor are calculated at the start of every leg. The drop in velocity depends on the angle of deviation from the current direction. For a small change in direction, the UAV can quickly alter its course. However, if the angle is large, the UAV takes some time to rotate itself to align with the new

course (`BC` in Figure 6.5) and then gradually increases its velocity. To model this behavior while monitoring, the soft and hard threshold parameters are proportional to the angle of deviation.

Calculating the angle of deviation:

Calculating the angle of deviation is essentially the problem of calculating the angle between three given points in a Cartesian plane. The angle between three waypoints can be calculated by treating them as the vertices of a triangle.

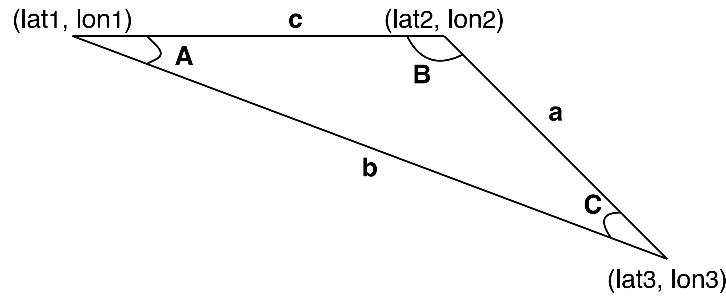


Figure 6.6: Triangle formed by waypoints A, B, and C

From the property of triangle:

$$b^2 = a^2 + c^2 - 2ac * \cos(B) \tag{6.9}$$

The angle `B` is given by,

$$B = \cos^{-1} \frac{(a^2 + c^2 - b^2)}{2ac} \tag{6.10}$$

Let `A (lat1, lon1)`, `B (lat2, lon2)` and `C (lat3, lon3)` be three consecutive waypoints as shown in Figure 6.6. The length of the edges (`a`, `b`, and `c`) in Equation 6.10 are given by:

$$a = \sqrt{(lat3 - lat2)^2 + (lon3 - lon2)^2}$$
$$b = \sqrt{(lat3 - lat1)^2 + (lon3 - lon1)^2}$$
$$c = \sqrt{(lat2 - lat1)^2 + (lon2 - lon1)^2}$$

The angle made by the three waypoints can be calculated by substituting the values of a, b, and c in Equation 6.10. This angle is marked as `x` in Figure 6.6. The angle of deviation from the current path is given by the angle formed by `A'`, `B` and `C`, which is `180 - x`.

Since this monitor is invoked immediately after a segment switch, the LTL formula is written under the assumption that the speed is already below the safe speed range. The LTL formula is given as:

$$ground_speed_increasing \, U \ in_safe_ground_speed_range \quad (6.11)$$

The AP values are calculated using the current ground speed ($ground_speed$):

$$ground_speed_increasing = is_ground_speed_increasing_seg()$$

$$in_safe_ground_speed_range = (ground_speed \geq rt_low_speed_thresh) \\ \wedge (ground_speed \leq rt_high_speed_thresh)$$

When the speed is back in the safe range, the other two runtime speed monitors are called into action.

6.2.3 Altitude Monitors

The UAV needs to fly at the specified altitude as it travels from one point in 3D space to another. The altitude monitors guard against the UAV flying too high or too low from the expected altitude. Given the expected altitude at a point ($set_altitude$) and the tol-

erance parameters ($ALT_UPPER_TOLERANCE$ and $ALT_LOW_TOLERANCE$), the $safe_altitude_zone$ is given by:

$$
\begin{aligned}
(set_altitude - ALT_LOW_TOLERANCE) \leq safe_altitude_zone \\
\leq (set_altitude + ALT_UPPER_TOLERANCE)
\end{aligned}
$$

Three different scenarios were considered while modeling altitude monitors and the following altitude profile plots explain the possible variations of altitude throughout the flight.

6.2.3.1 Scenario 1: Constant Altitude

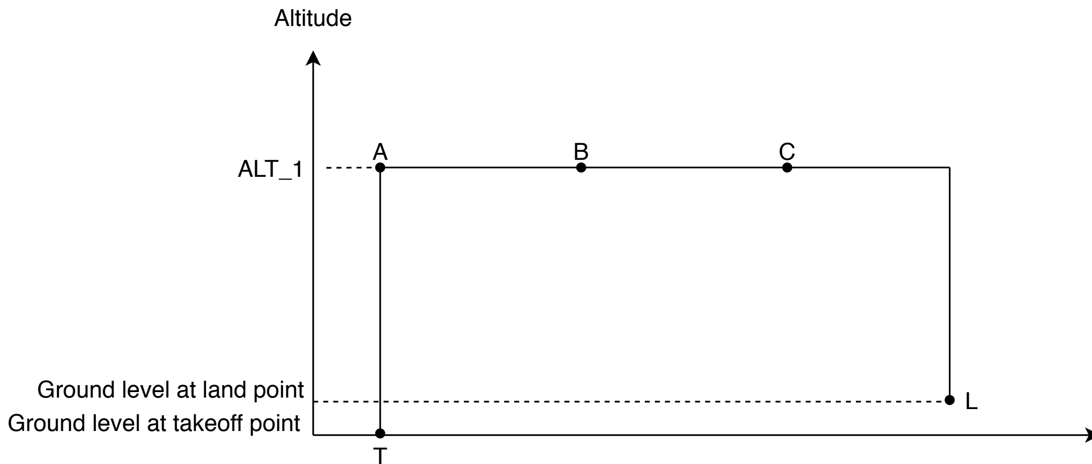


Figure 6.7: No change in altitude during runtime

In the plot in Figure 6.7, `T` represents the takeoff point. The UAV takes off from the ground level at this point until altitude `ALT_1`. The waypoint `A` has the same latitude and longitude coordinates as the takeoff point but is higher in altitude. `B` and `C` are two waypoints along the flight path, which are at the same altitude, `ALT_1`. `L` is the landing point. Note that the ground level at the landing point is not the same as that at the takeoff point. In the last leg of the flight, the UAV cruises at the altitude of the penultimate waypoint (`C` in the above example). The UAV begins to land when it is close to the latitude and longitude coordinates

of the landing point. In this example, the altitude of the UAV remains constant throughout the flight. This scenario is relatively easy to monitor compared to the following scenarios 2 and 3 as it is sufficient to monitor if the UAV remains in the *safe_altitude_zone* throughout the flight, thereby conforming to the altitude in the flight plan.

6.2.3.2 Scenario 2: Climb

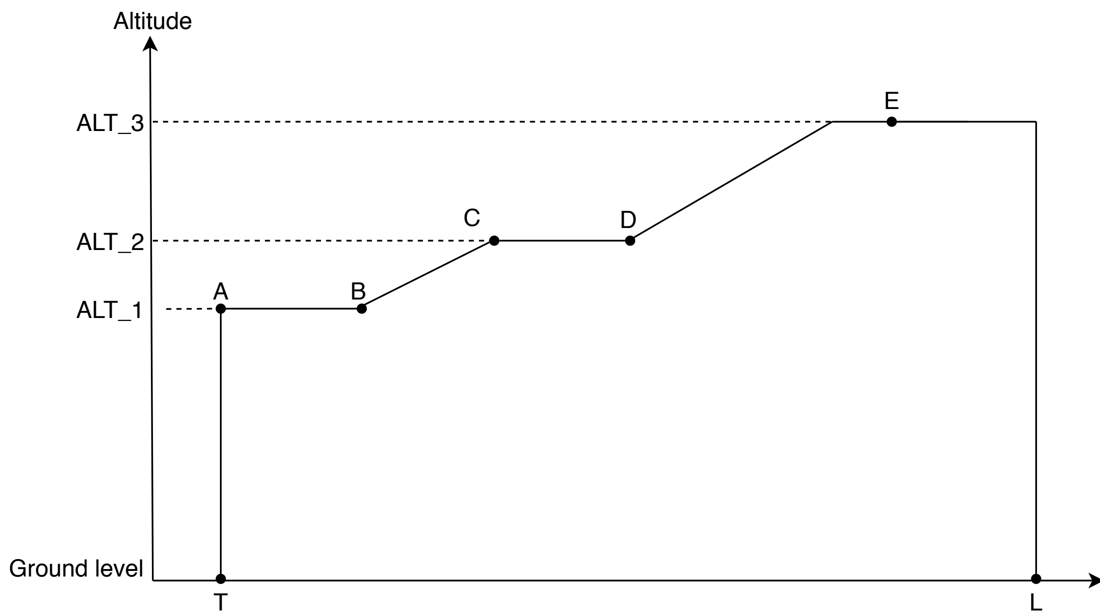


Figure 6.8: Increase in altitude during runtime

In the plot in Figure 6.8, the UAV takes off to point `A`, which is at an altitude `ALT_1` and cruises to the waypoint `B` at the same altitude. In the next segment of the flight, `BC`, the UAV climbs from `ALT_1` at waypoint `B` to `ALT_2` at waypoint `C`. The UAV climbs at the *climb_speed*, as discussed in Section 6.1.2, and the set *ground_speed* is maintained while climbing or descending. It can be observed that an altitude of `ALT_2` was reached exactly at waypoint `C`. This indicates that the time taken to climb (`ALT_2` – `ALT_1`) and the time taken

to cover the distance between `B` and `C` on a horizontal plane were the same. That is:

$$\frac{ALT_2 - ALT_1}{climb_speed} = \frac{Dist(B, C)}{ground_speed}$$

This is not the case always, as observed in the segment `DE`. The UAV climbed to altitude `ALT_3` before reaching waypoint `E` and continued cruising at the same altitude until waypoint `E`. Here:

$$\frac{ALT_3 - ALT_2}{climb_speed} < \frac{Dist(D, E)}{ground_speed}$$

When the flight plan involves a climb (or descent), it is essential to monitor the climb rate apart from enforcing flight plan conformance. Figure 6.9 illustrates how this is performed. Consider two waypoints `A` and `B` at altitudes `ALT_1` and `ALT_2` respectively. Given the *climb_speed* and the number of GPS packets received per second (*gps_packets_per_sec*), the *climb_speed_per_frame* is given by:

$$climb_speed_per_frame = \frac{climb_speed}{gps_packets_per_sec}$$

The expected altitude at the Nth frame during the climb is given by:

$$ALT_EXPECTED = ALT_1 + (N * climb_speed_per_frame)$$

The new *ALT_EXPECTED* value is computed every frame until it is equal to or greater than *ALT_2*. If it is greater than *ALT_2*, the *ALT_EXPECTED* is set as *ALT_2*. The safe altitude zone (*safe_altitude_zone*), which is computed using *ALT_EXPECTED*, is a moving target that the UAV must meet. Thus, flight plan conformance and rate of climb can be monitored by continuously moving the limits.

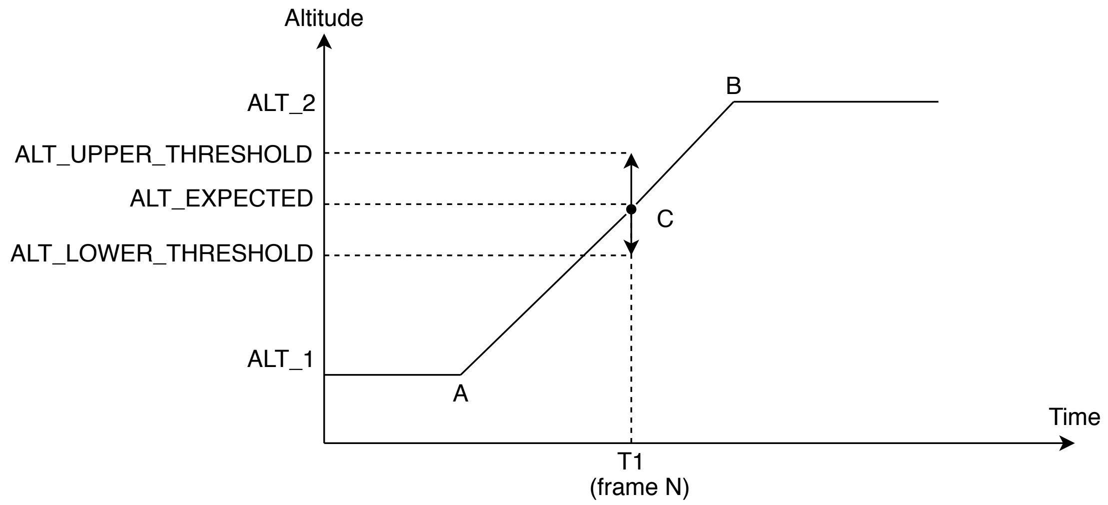


Figure 6.9: Calculation of safe altitude zone during climb

6.2.3.3 Scenario 3: Descent

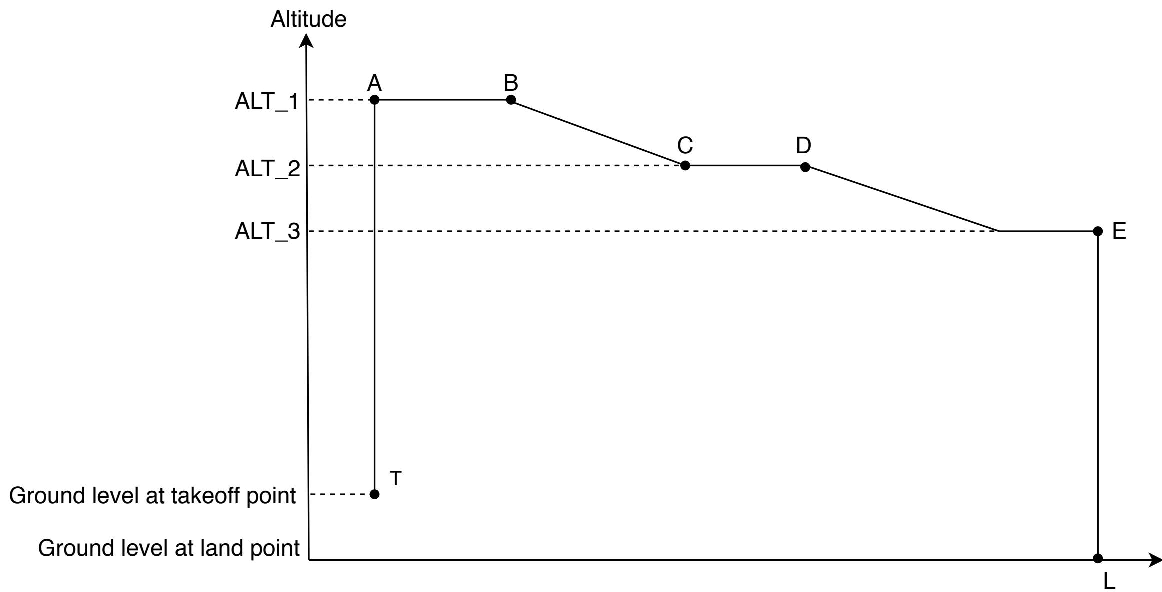


Figure 6.10: Decrease in altitude during flight

Replacing the *climb_speed* with *descent_speed* in the previous case and continuously lowering the *safe_altitude_zone* limits enable monitoring the rate of descent and flight plan conformance. The altitude profile plot in Figure 6.10 shows an example flight with de-

scent actions during runtime. The plot in Figure 6.11 illustrates the continuous update of *safe_altitude_zone*. Consider two waypoints `A` and `B` at altitudes `ALT_1` and `ALT_2` respectively as shown in Figure 6.11. Given the *descent_speed* and the number of GPS packets received per second (*gps_packets_per_sec*), the *descent_speed_per_frame* is given by:

$$descent_speed_per_frame = \frac{descent_speed}{gps_packets_per_sec}$$

The expected altitude at the Nth frame during the descent is given by:

$$ALT_EXPECTED = ALT_1 - (N * descent_speed_per_frame)$$

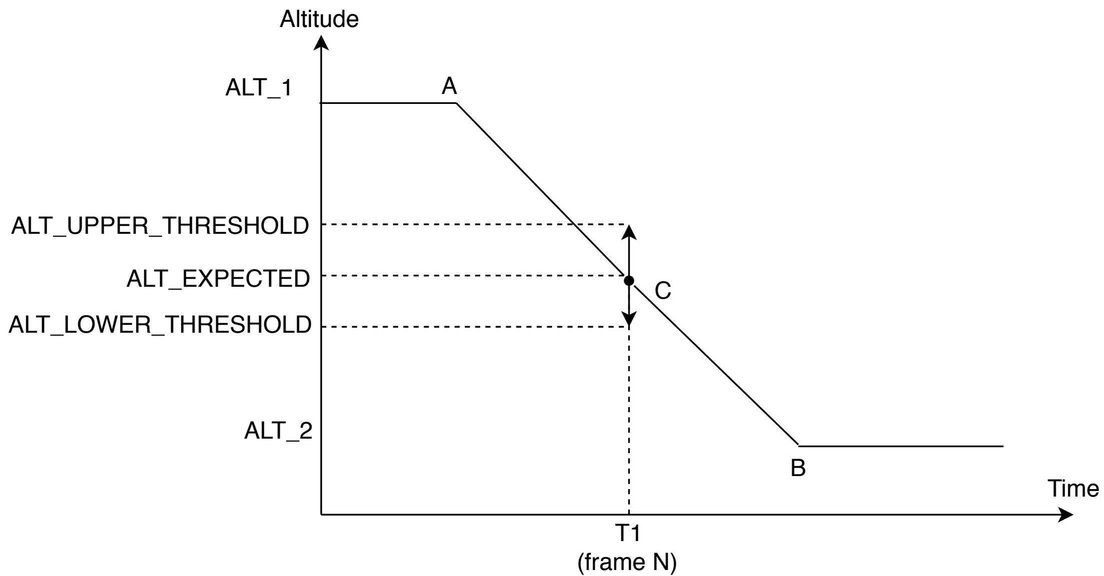


Figure 6.11: Calculation of safe altitude zone during descent

In all three scenarios, the monitors essentially ensure that the UAV remains within the *safe_altitude_zone*. There are two monitors, one guarding against breaching the upper altitude threshold (*within_max_alt*) and the other guarding against breaching the lower altitude threshold (*within_min_alt*). The requirement for *within_max_altitude* is captured

in the following LTL formula:

$$(nearing_fence \wedge in_slow_zone) \Rightarrow (can_slowdown\, U\ not_nearing_fence) \tag{6.12}$$

The APs are calculated using the current altitude (alt), and vertical speed (ver_speed):

$$\begin{aligned} nearing_fence &= (ver_speed > 0) \wedge (alt > MID_ALT) \\ in_slow_zone &= (alt > UP_SLOWDOWN_START) \\ can_slowdown &= (ver_speed \leq (ALT_ZONE_MAX - alt)) \\ not_nearing_fence &= (ver_speed = 0)\ \vee\ (ver_speed < 0) \end{aligned}$$

6.2.4 Collision Avoidance Monitor

The collision avoidance monitor guards against the UAV colliding with any flying or stationary obstacle. The distance and rate of convergence of the obstacle are received from the obstacle detection application (Section 5.7.1). The monitor should not act too soon or too late, and the collision avoidance application should be given sufficient time to take evasive action. However, the UAV should not be allowed to get so close to the obstacle that it is difficult to take any evasive action. The following LTL formula captures the requirement for the collision avoidance monitor:

$$(obstacle_in_vicinity \wedge approaching) \Rightarrow (can_avoid\, U\ obstacle_avoided) \tag{6.13}$$

The APs are calculated using the distance between the UAV and obstacle ($dist_to_obstacle$), and their rate of convergence ($convg_rate$):

$$\begin{aligned} obstacle_in_vicinity &= (dist_to_obstacle < OBSTACLE_VICINITY_RADIUS) \\ approaching &= (convg_rate > 0) \end{aligned}$$

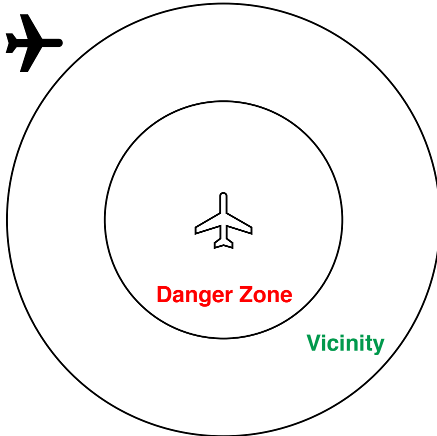
$$can_avoid = convg_rate < (dist_to_obstacle - OBSTACLE_DANGER_ZONE_RADIUS)$$

$$obstacle_avoided = (convg_rate < 0) \vee (dist_to_obstacle > OBSTACLE_VICINITY_RADIUS)$$

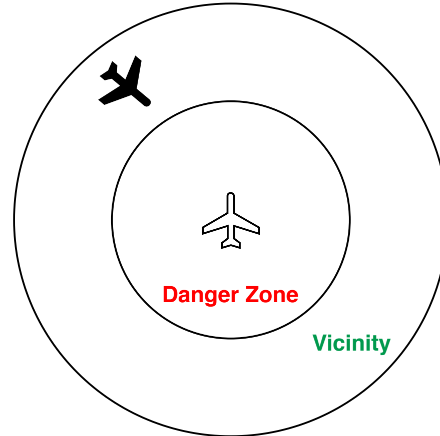
In words, if an obstacle in the vicinity of the UAV is moving towards it (or the UAV is moving towards a stationary object), then the obstacle must remain at a sufficiently long distance until any evasive action is taken. If the UAV gets too close to the object, then a guard is triggered. The rate at which the objects are converging must be taken into account. On the other hand, two objects must be allowed to fly parallel to each other as long as they remain outside each other's danger zone. Atomic propositional values for different scenarios are illustrated in Figure 6.12.

6.3 Land Monitors

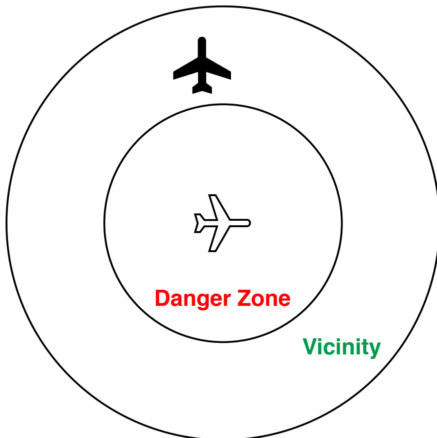
The land monitors are called into action when the UAV completes cruise flight and begins to land. There are two categories of monitors: land position monitors and land speed monitors. These monitors are similar to those discussed in the takeoff section and their LTL formulas are the same. The critical difference is in setting the tolerance parameters. Landing is a more complex action compared to takeoff, and the utmost care is required while enforcing the correct behaviors. The following two subsections discuss the reasons for the choice of parameter values and how the monitors differ from their takeoff counterparts.



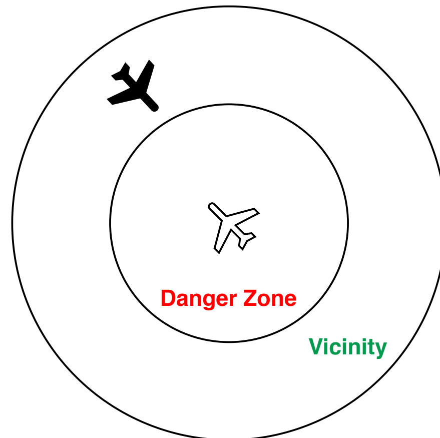
Scenario 1: Obstacle outside the vicinity but the distance between them is reducing.
obstacle_in_vicinity = false
approaching = true
can_avoid = true
obstacle_avoided = true



Scenario 2: Obstacle within the vicinity and the distance between them is reducing.
But the rate of convergence is less than (*distance_to_obstacle* - *Danger_Zone_Radius*).
obstacle_in_vicinity = true
approaching = true
can_avoid = true
obstacle_avoided = false



Scenario 3: Obstacle within the vicinity and the distance between them is increasing.
obstacle_in_vicinity = true
approaching = false
can_avoid = true
obstacle_avoided = false



Scenario 4: Obstacle within the vicinity and the distance between them is reducing.
But the rate of convergence is greater than (*distance_to_obstacle* - *Danger_Zone_Radius*).
obstacle_in_vicinity = true
approaching = true
can_avoid = false
obstacle_avoided = false

Figure 6.12: Different scenarios leading to different AP values

6.3.1 Land Position Monitors

In both virtual flights (Gazebo simulation), and actual flights (Intel Aero), the UAV sways much more during the land phase than the takeoff phase. Thus, the *safe_landing_zone* is set to be much wider than the *safe_takeoff_zone*.

6.3.2 Land Speed Monitors

The land under speed and overspeed monitors guard the rate at which the UAV descends. Landing typically takes more time than takeoff. The descent speed is set much lower than the climb speed because smoothness is more important than quickness while landing. If the UAV descends too fast, it will smash into the ground rather than smoothly touch the ground. Also, unlike the *climb_speed*, which is the expected rate of climb, the *descent_speed* is the maximum rate of descent. The UAV should not exceed this speed while descending but can be slower. Thus, the overspeed monitor has a stringent tolerance limit to avoid landing too fast. While landing, sometimes, the UAV can move upwards to position itself to land at the correct spot. Thus, the under speed monitor should allow the descent speed to drop below zero but ensure that the speed does not remain below zero for a long time. The *safe_descent_speed_range* is given as:

$$0 \leq safe_descent_speed_range \leq (descent_speed + high_descent_speed_tolerance)$$

where *high_descent_speed_tolerance* is very small.

Chapter 7

Evaluation

This chapter discusses the different ways in which the RTA system was evaluated. Section 7.1 lists the software optimizations performed to reduce the latency of the system. Virtual flight tests in the Gazebo simulation environment are discussed in Section 7.2, followed by timing analysis in Section 7.3. Finally, the scalability of the architecture is analyzed in Section 7.4.

7.1 Software Optimization

The software code executed in different processors are optimized to reduce the latency of the RTA system. This section discusses the ideas that were considered/implemented to improve the system's latency.

1. **Cache:** Initially, the MicroBlaze processors used in the design, namely IOP and MP, did not use instruction or data caches. Caches generally reduce the time taken to fetch data and therefore lead to faster system performance. However, caches are not useful because both the dedicated memory for a MicroBlaze and the cache are built using BRAM, giving rise to similar performances.

2. **Optimization Flags:** The Xilinx Software Development Kit (SDK) provides the option to choose between five GNU compiler optimization flags: none (O0), optimize

(O1), optimize more (O2), optimize most (O3), and optimize for size (Os). The O3 flag was set for all the processors resulting in a highly optimized code. Debugging optimized code is tricky as the sequence of instructions executed may not match the order in the original unoptimized code.

3. **Removal of assert statements:** Software optimization was required to reduce the overhead incurred while accessing registers in the hardware monitors. The HLS tool generates a driver for each of the monitors that provides APIs to access the registers. Listing 7.1 captures the function to set the current latitude register in the takeoff monitor. Note that the function asserts the instance pointer is not null and the component is ready before writing the data. These two `assert` statements add non-negligible overhead. While useful during development, these statements can be removed before deployment. Thus, the software code was modified to use the `_WriteReg` (line 5) and `_ReadReg` functions with correct offsets to directly access the registers instead of using the APIs provided by the driver. For example, the initialization of takeoff monitor parameters took 2767 ns with the `assert` statements and 1533 ns without them.

Listing 7.1: C function to set current latitude register in the takeoff monitor

```
1 void XTakeoff_monitor_Set_location_lat(XTakeoff_monitor *InstancePtr, u32 Data) {
2     Xil_AssertVoid(InstancePtr != NULL);
3     Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
4
5     XTakeoff_monitor_WriteReg(InstancePtr->Axilites_BaseAddress,
          XTAKEOFF_MONITOR_AXILITES_ADDR_LOCATION_LAT_DATA, Data);
6 }
```

7.2 Virtual Flight Tests

Section 4.3 explained the use of model checking to prove the correctness of monitors, which constitute one part of the RTA system. To test the entire system, virtual flight tests are performed in the Gazebo simulation environment. The functionality of the monitors, interaction between processors, interaction between the MP and hardware monitors, and timely execution of RCF are thoroughly tested by simulating different flight scenarios. Figure 7.1 gives an example of a simple flight plan. In this figure, the UAV, flying at a relative altitude of 49.4 m, is about to begin its landing phase after completing the cruise portion of the flight.

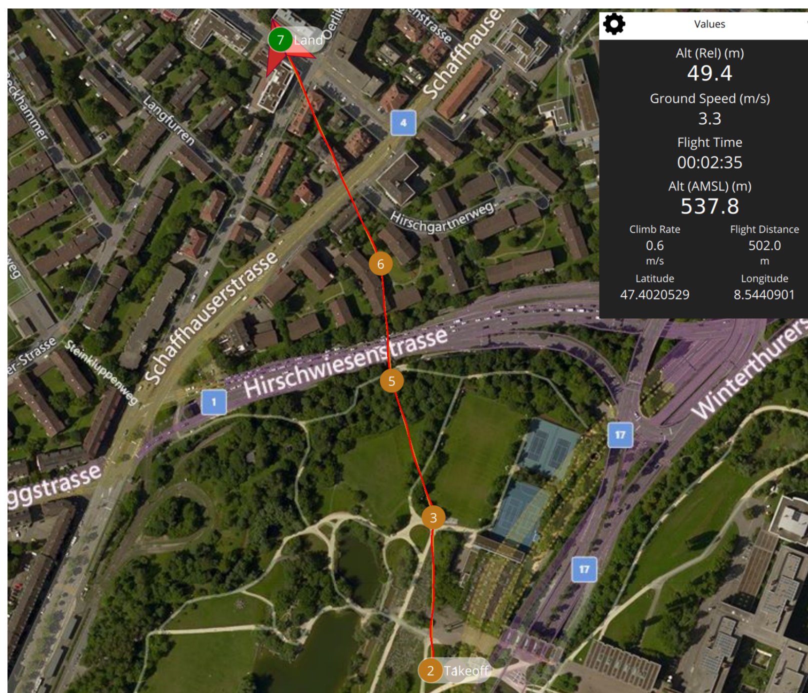


Figure 7.1: A successful flight

One method to test the monitors in action involves reducing the tolerance limits, causing even a slight deviation from expected behavior to be flagged. This approach is used in timing analysis as described in the next section. However, for the flight tests in the simulation

environment, it is desirable to replicate a real-life situation, and therefore refrain from altering tolerance limits. Thus, external stimuli are required to force different violations. These stimuli come from the QGC in the form of commands that alter the course of the flight, hold the UAS in its position, slow the UAS's progress, or set the climb or cruise speeds to be different from those in the flight plan. These situations mimic unexpected behaviors in real flights. Three flight test scenarios are discussed below.

7.2.1 Runtime Speed Violation

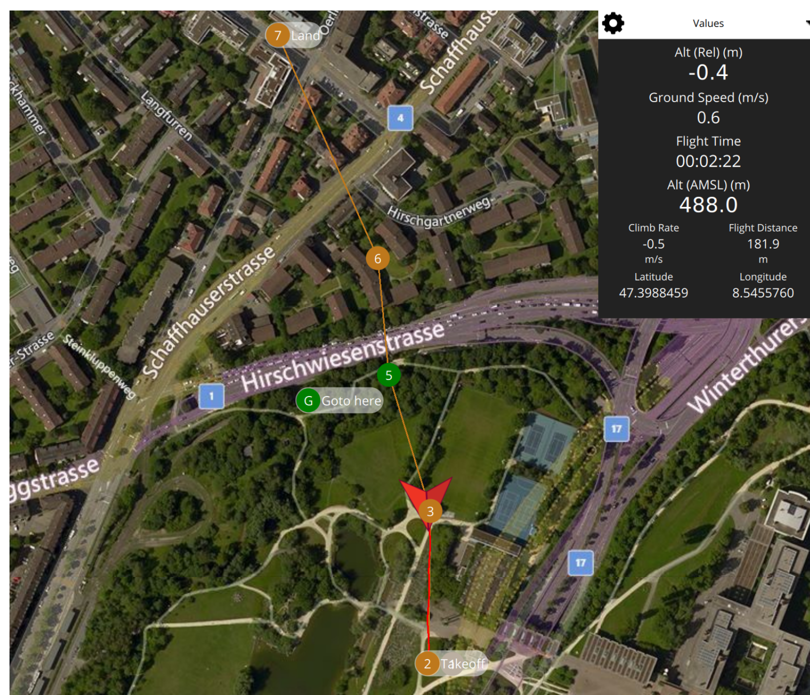


Figure 7.2: Runtime under speed violation

Figure 7.2 shows a test scenario in which the runtime under speed monitor detects a violation. A `HOLD` command is sent from QGC when the UAV crosses the waypoint marked `3`. The UAV holds on to its position long enough (more than the soft threshold time) for the under speed monitor to trigger the RCF causing the UAV to land. A relative altitude (first item in

the `Values` box in the figure) of -0.4 m (close to 0) indicates that the UAV is on the ground.

7.2.2 Runtime Position Violation

A test scenario that leads to a runtime position violation is illustrated in Figure 7.3. A `REPOSITION` command is sent from the QGC to alter the course of the UAV. The UAV begins to move away from the intended flight path and the runtime position monitor triggers the land RCF if the UAV could breach the corridor (*corridor_width_hard_threshold* distance). `Land`, highlighted by the red box, indicates the command being executed. The UAV is still going through its land phase as indicated by the relative altitude (38 m).

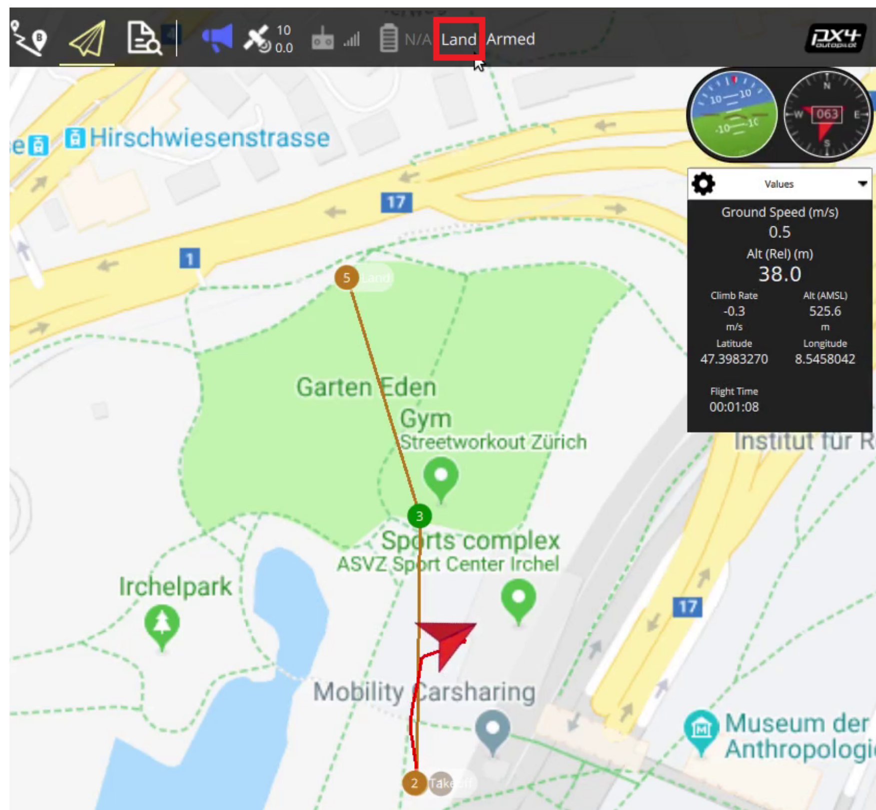


Figure 7.3: Runtime position violation

7.2.3 Collision Avoidance Violation

Section 5.7 explained the method used to simulate an obstacle and different evasive actions that can be taken to avoid the obstacle. The collision avoidance monitor, described in Section 6.2.4, triggers the RCF if the obstacle could breach the danger zone in the next 5 cycles (1 second). Figure 7.4 depicts a flight test scenario that leads to collision avoidance violation. The simulation on the left, traveling top to down, represents the obstacle. The UAV on the right is the UAV under test and it is equipped with an RTA system. The two UAVs are moving towards each other. No evasive action is taken in this scenario leading to a violation. The collision avoidance monitor triggers the land RCF to prevent a collision. The UAV on the right, which is landing, is at an altitude of 29 m when the obstacle is at an altitude of 50 m.

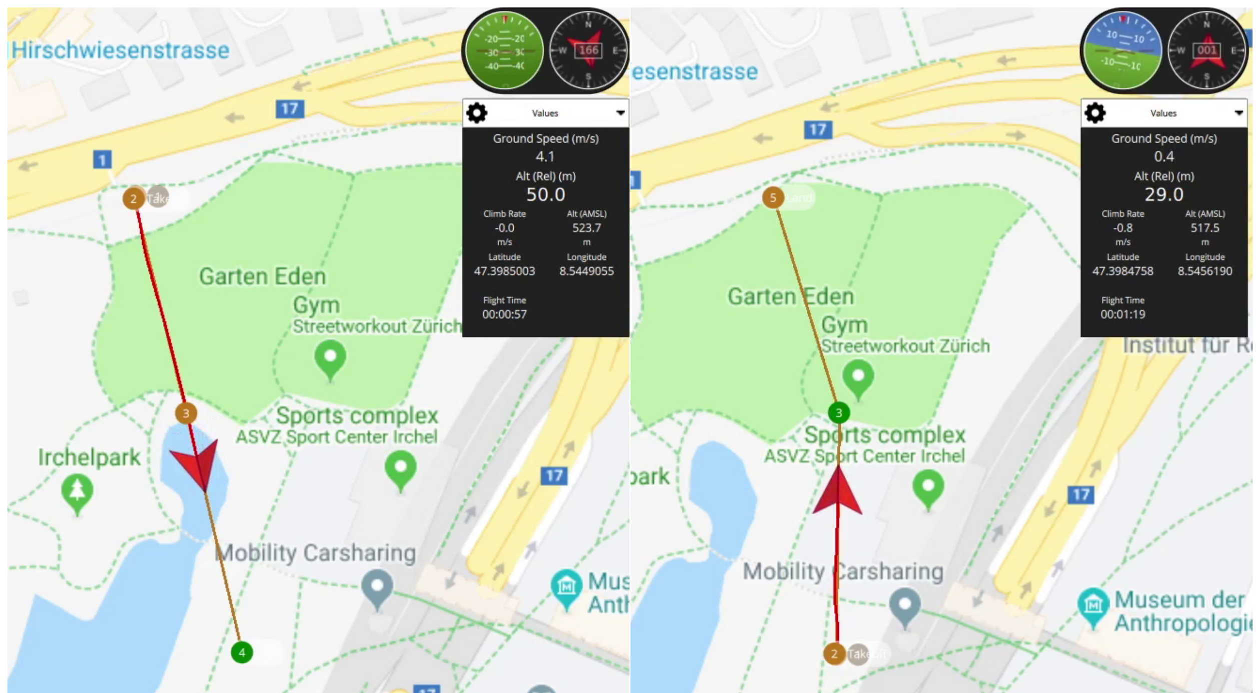


Figure 7.4: Collision avoidance violation

7.3 Timing Analysis

An AXI timer block is connected to each of the processors in the system [43]. The clock frequency of the timers is 150 MHz, which implies the timers tick every 6.666 ns. An automated test procedure, with test scripts triggering different violations, is used to measure the execution time of monitors and the overall latency introduced by the MP in case of violations. The latter is the time difference between the IOP writing the sensor data in the Mailbox and reading the violation message. This includes the time taken for the MP to read data from the Mailbox, invoke the monitors and write the violation message in the Mailbox. It also includes the time taken by the IOP to read data from the Mailbox.

In the tables in this section, "Software" refers to monitors executed sequentially in the MicroBlaze soft processor. "Hardware" refers to parallel monitors wrapped into one single hardware block. From the results of the first phase (Section 4.4.3), it is evident that wrapped, fully parallel hardware monitors perform better than sequential and overlapped hardware monitors. Thus, the latter two means of implementing the monitors are not considered here.

Phase	**Number of Parameters**	**Execution time (ns)**	
		Software	**Hardware**
Takeoff	8	340	1533
Runtime	11	586	2647
Land	8	320	1533

Table 7.1: Initialization time

Table 7.1 captures the time taken to set the parameters for different sets of monitors. As observed, initialization in software is approximately 4.5 times faster than in hardware. This is because initializing parameters in software involves a function call that sets a group of variables, whereas in hardware it involves writing to registers. However, initialization is a one-time process that is executed before takeoff. The execution time of the monitors is more important because they are periodically invoked during the flight.

Phase	#Monitors	Section	#Sensor Data	Execution time (ns)	
				Software	Hardware
Takeoff	6	Entire phase	3	1314	1027
Runtime	6	First frame in first leg	12	929354	7167
		First frame in 2 to N legs	12	1252234	9180
		1 to N-1 legs	6	268187	3780
		Last leg	6	208320	3247
	1	Collision avoidance monitor	2	200	940
Land	6	Entire phase	3	1327	1047

Table 7.2: Execution time of different sets of monitors

Table 7.2 captures the execution time of different sets of monitors. In the case of software monitors, the timer is started before the function call and stopped immediately after. In the case of hardware monitors, the timer is started before waiting for the ready signal and stopped after reading the result(s) from the register(s). In the takeoff and land phases, the hardware monitors are faster than the software counterparts. The difference in execution time is around 280 ns.

The runtime phase can be divided into five sections as shown in the table. The waypoints information is updated in the first frame of every leg and the monitor calculates the slope and angle of the upcoming flight leg. The change in direction is also calculated in the first frames of legs 2 to N. As a result, the execution time of the first frame in every leg is greater than the rest of the frames. In legs 1 to N-1, the distance to the next flight leg is calculated to detect a switch to the next segment. This calculation is not performed in the last leg. Thus, the execution is faster in the last leg. In the first four sections, the hardware monitors are faster by 130, 136, 71 and 64 times respectively.

The anomaly in this table is the collision avoidance monitor. The software monitor outperforms the hardware because there is only one monitor and there is no scope for concurrent execution. This is an example of a monitor that can exist in a soft processor rather than in hardware. The soft processor is also isolated from the application software. Thus, a

combination of software and hardware monitors can yield better performance.

Table 7.3 captures the time taken by the monitoring system (MP and monitors) to report a violation. The difference between the values in the table is a multiple of 1380 ns, which is the execution time of one IOP cycle. The IOP tries to read data from the Mailbox, and if it is empty, moves on to the next instruction and comes back in the next cycle.

Phase	**Monitor**	**Time (ns)**	
		Software	**Hardware**
Takeoff	Under-speed	13780	12400
	Overspeed	12400	12400
	Position	12400	12400
Runtime Without collision avoidance monitor. Legs 1 to N-1	Altitude	284260	16540
	Position	282880	16540
	Overspeed	280120	16540
	Under-speed	281500	16540
Runtime Without collision avoidance monitor. Last leg	Altitude	216640	16540
	Position	220780	16540
	Overspeed	220780	16540
	Underspeed	222160	16540
	Segment switch	222160	16540
Runtime With collision avoidance monitor. Legs 1 to N-1	Altitude	285640	19300
	Position	284260	19300
	Overspeed	281500	19300
	Underspeed	282880	19300
	Collision avoidance	9640	9640
Runtime With collision avoidance monitor. Last leg	Altitude	219400	19300
	Position	220780	19300
	Overspeed	223540	19300
	Under-speed	224920	19300
	Segment switch	224920	19300
Land	Under-speed	13780	12400
	Overspeed	12400	12400
	Position	12400	12400

Table 7.3: Time taken for MP to report monitor violations to IOP

The latency introduced with hardware monitors is a constant for a given phase irrespective

of the rule being violated. However, with software monitors, the latency varies for different violations because of the left to right execution property of the `AND` (`&&`) operator. If the expression or function on the left evaluates to `false`, then the right-hand side is not executed.

Table 7.3 also gives a comparison between the latency introduced in the absence and presence of the collision avoidance monitor. Since obstacle details come from a different source, the MP has to read data from two different Mailboxes and call different sets of monitors. As expected, the latency is greater when the collision avoidance monitor is included.

Looking at the same table, it can be observed that a violation detected by the collision avoidance monitor is signaled much faster. The reason for this is the order of execution of instructions in the IOP and MP. The IOP sends GPS sensor data to the obstacle detection application before sending it to the MP. In every execution cycle, the MP first tries to read the sensor data from the obstacle detection application. As a result, the collision avoidance monitor is executed first and the other set of monitors are executed later.

7.4 Scalability Analysis

The RTA system discussed so far has three types of monitors, 19 in total, which are invoked during different phases of flight. This section examines the scalability of the RTA system architecture, using hardware monitors, by analyzing the system performance when the number of monitors grows. For this analysis, the number of monitors is increased to 37, almost double the initial number. This is done by replicating the monitors and using a different set of parameters. There are two ways to support this increase: (1) Single Monitor Processor System (SMPS), which have one MP invoke all the 37 monitors, or (2) Dual Monitor Processor System (DMPS), which have 2 MPs invoke half the number of monitors each. In SMPS, depicted in Figure 7.5, the number of parallel monitors wrapped into one block is doubled.

The increase in the number of monitors is accompanied by an increase in the number of registers, which increases the number of write operations. In DMPS, depicted in Figure 7.6 the two MPs handle 19 and 18 monitors respectively. Hence, the number of write operations executed every cycle does not increase. As discussed in Section 5.4.2, the UIP interacts with the MPs in a client-server relationship, and it can support multiple MPs.

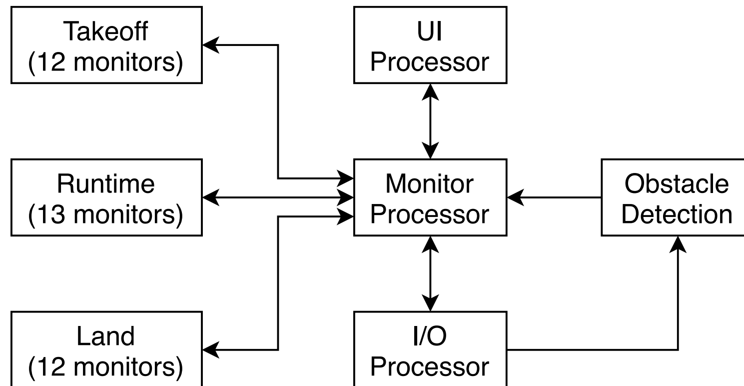


Figure 7.5: Single Monitor Processor System

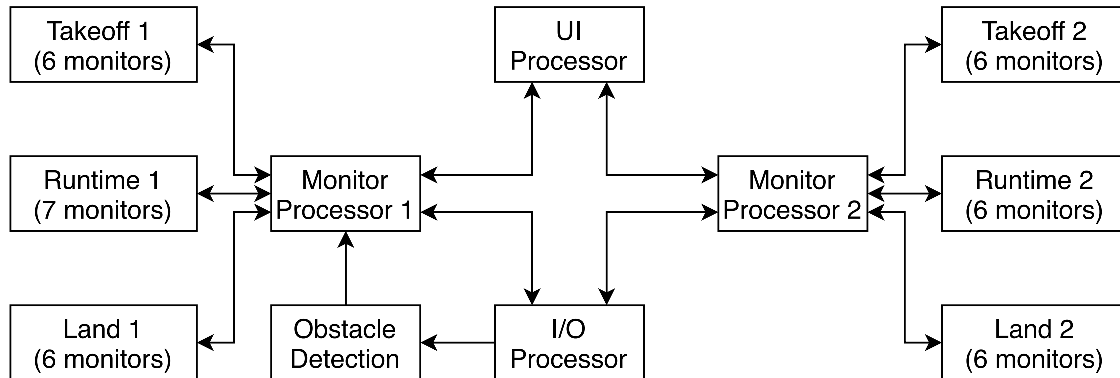


Figure 7.6: Dual Monitor Processor System

The parameter initialization time in the DMPS is the same as in Table 7.1. Table 7.4 gives the time taken to initialize parameters in the SMPS with 37 monitors. Though the number of parameters has doubled, the execution time has not. However, it has still increased by a considerable amount.

Phase	#Parameters	Time (ns)
Takeoff	16	2633
Runtime	21	3820
Land	16	2593

Table 7.4: Initialization time after doubling the number of monitors

Table 7.5 captures the execution time of different sets of monitors in the SMPS with 37 monitors. Since the monitors execute in parallel, increasing the number of monitors should not affect the actual execution time of the monitors. However, the overhead involved in writing sensor data to the registers must be included, which leads to an increase in the perceived execution time. Though the number of monitors has doubled, the execution time has not. Table 7.2, discussed earlier, also holds true for the DMPS.

<table>
<tr><th>Phase</th><th>#Monitors</th><th>Section</th><th>#Sensor Data</th><th>Time (ns)</th></tr>
<tr><td>Takeoff</td><td>12</td><td>Entire phase</td><td>6</td><td>1420</td></tr>
<tr><td rowspan="5">Runtime</td><td rowspan="4">12</td><td>First frame in first leg</td><td>24</td><td>9287</td></tr>
<tr><td>First frame in 2 to N legs</td><td>24</td><td>10780</td></tr>
<tr><td>1 to N-1 legs</td><td>12</td><td>5907</td></tr>
<tr><td>Last leg</td><td>12</td><td>4847</td></tr>
<tr><td>1</td><td>Collision avoidance monitor</td><td>2</td><td>940</td></tr>
<tr><td>Land</td><td>12</td><td>Entire phase</td><td>6</td><td>1453</td></tr>
</table>

Table 7.5: Execution time of different sets of monitors after doubling the number of monitors

Table 7.6 gives a comparison between the execution times of the monitoring systems (MP and monitors) in the SMPS and DMPS. As expected, the system with one MP handling all the monitors introduces higher latency. Reporting a violation in the DMPS takes a different amount of time for each of the two monitors. This difference arises because the IOP sequentially accesses the Mailboxes connected to the MPs resulting in less latency for the processor whose Mailbox is accessed first.

Phase	Time (ns)		
	SMPS	DMPS	
		Monitor 1	Monitor 2
Takeoff	14020	10700	12094
Runtime	19540	14194	15580
Land	14020	10700	12094

Table 7.6: Comparison of time taken to report a violation

An interesting observation can be made by comparing Tables 7.3 and 7.6. The latency introduced by each of the two monitors in the DMPS is less than the latency introduced by one monitor handling the same number of monitors (19). This is because the IOP uses different cyclic executives in each case. In the DMPS, two tasks are added to the IOP, writing to and reading from another Mailbox, resulting in different execution times. This observation is specific to this scenario and is not general.

Table 7.7 gives a comparison of inter-processor communication times in the two systems. In the SMPS, the IOP writes data to one Mailbox, which takes 1320 ns. In the DMPS, the IOP writes data to two Mailboxes, which takes twice the time. The UIP, upon receiving a start command from the user, sends start commands to the MPs and shares the parameters. The initial synchronization time increases by approximately 10% in the DMPS, which is acceptable. The time taken to receive the next waypoint and the last leg message in the DMPS remain the same as in the SMPS.

Processors	Communication	Time (ns)	
		SMPS	DMPS
IOP	Transmit one GPS packet	1320	2640
UIP and MP	Initial synchronization	78237	85293
	Receive next waypoint	12626	12380/9360
	Receive last leg message	3743	3640/2947

Table 7.7: Inter-processor communication time

Table 7.8 shows the FPGA resources available and those utilized by the different architectures discussed so far. The MicroBlaze processors and monitors are connected to a 150 MHz clock source, and the Zynq processing system is connected to a 500 MHz clock. It can be observed from the table that the hardware monitors consume substantially more resources than software monitors. The difference between the hardware and software columns gives the additional resources used by the hardware monitors. 626 out of the 634 digital signal processing units (DSPs) are used by the runtime monitors. Though the latency of DMPS is less compared to the SMPS, its resource utilization is approximately twice that of the SMPS.

Resource	**Available**	**Utilization**			
		Software	**Hardware**	**SMPS**	**DMPS**
Lookup table as logic	230400	9446	64677	66593	122964
Lookup table as memory	101760	1273	2003	2003	2922
Block RAM	312	58	119	119	186
Digital Signal Processors	1728	8	634	634	1266
Configurable logic blocks	28800	2286	12537	12827	22468

Table 7.8: Resource utilization

7.5 Summary

The worst-case response time, in the presence of hardware monitors, is around 20 μs. This response time is acceptable considering the fact that the GPS data is received every 200 ms. A combination of hardware and software monitors can be faster in certain scenarios. It can be concluded from these observations that the RTA architecture supports the addition of multiple MPs. Since inter-processor communication is not a bottleneck, the only constraint in scaling the architecture is resource availability.

Chapter 8

Conclusion

UAS complexity challenges traditional methods of software verification and analysis. The vast code base and unpredictability of the algorithms used in UAS make it difficult for formal methods to establish correctness. This thesis presents an alternative method to provide formal assurances for a UAS without verifying its application software. A runtime safety assurance system consistent with the ASTM standard F3269-17 is incorporated into the UAS to actively monitor the system's inputs and outputs. This change is strictly additive and does not require modifications to the software stack. The latency introduced by the RTA system is minimized by implementing the monitors, RCFs, and switching logic in an FPGA, which also provides isolation from other components of the UAS. A synthesis flow translating LTL formulas to hardware monitors increases productivity and reduces the potential for oversights. LTL specifications and the translation process are validated by applying model checking to the HDL implementation.

An initial demonstration with the Intel Aero establishes the feasibility of adding an RTA system to a COTS UAS and its effectiveness in enforcing correct behaviors. The second phase extends the idea to an autonomous flight BVLOS with an improved architecture and distinct sets of monitors for different flight phases. Configuring the monitors with waypoints and tolerance limits extracted from a flight plan transferred over Bluetooth removes the need to reprogram the FPGA for every flight. The monitoring system introduces minimal latency and the architecture supports the use of additional monitors without affecting performance.

Successful virtual flight tests are a promising step towards trusted UAS operations. The RTA system is not limited to UAS and can be applied to other autonomous systems where safety is paramount.

8.1 Future Work

Formally capturing requirements in LTL requires practice and can go awry. To avoid incorrect, incomplete or inconsistent requirements, NASA's Formal Requirements Elicitation Tool (FRET) can be used to convert requirements specified in English to LTL formulae, thereby increasing the level of automation in the monitor synthesis flow [13].

Adding cryptographic safeguards to the flight plan before uploading is one way to improve the system's security. The signature of the received file should be verified by the RTA system to ensure that the waypoints and parameters are unmodified. The system should support the ability to upload multiple flight plans and store them in the flash device. UAS operators could choose the file used for monitoring. Once the flight is complete, the operator should be able to delete the file.

At the present time, drone operations are still limited to surveying terrain and infrastructure. The regions surveyed usually have a complex polygonal shape with include and exclude areas. A geofence monitor that utilizes an established point-in-polygon algorithm, such as the winding number algorithm, can ensure that the UAV remains within the include areas [33]. A set of envelope protection monitors that observe the roll, pitch and yaw rates can ensure the UAV does not exceed its operating limits. Monitors have a useful role during system development, and not just after deployment. They can complement machine learning algorithms since they are deterministic and have mathematical assurances. Therefore, the monitors can also assist in training the algorithms.

Bibliography

[1] Intel - Aero Ready to Fly Drone Overview - CNET. URL https://www.cnet.com/products/intel-aero-ready-to-fly-drone/.

[2] Improving Nios II ISR Performance. URL https://www.intel.com/content/www/us/en/programmable/documentation/mwh1416947091611/mwh1416946760505/mwh1416946900343.html.

[3] Nios II Processor Reference Guide. URL https://www.intel.com/content/www/us/en/programmable/documentation/iga1420498949526.html.

[4] Amazon Scout Robots. URL https://www.theverge.com/2019/8/9/20798604/amazon-scout-robot-delivery-irving-southern-california-expansion-prime.

[5] Ardupilot home. URL https://ardupilot.org/.

[6] Xilinx AXI Quad SPI. URL https://www.xilinx.com/products/intellectual-property/axi_quadspi.html.

[7] Digilent PMOD BT2 Reference Manual. URL https://reference.digilentinc.com/reference/pmod/pmodbt2/reference-manual.

[8] Drones crash into high rise building. URL https://www.chicagotribune.com/news/breaking/ct-met-wacker-drive-drone-crash-20190214-story.html.

[9] MAVLink common messages. URL https://mavlink.io/en/messages/common.html#MAV_CMD_DO_REPOSITION.

[10] CppDepend. URL https://www.cppdepend.com/.

[11] FAA investigating after drone hits Devon Tower. URL https://kfor.com/news/faa-investigating-after-drone-hits-devon-tower/.

[12] EPITA Spot tool web application. URL https://spot.lrde.epita.fr/app/.

[13] NASA FRET Tool. URL https://software.nasa.gov/software/ARC-18066-1.

[14] Gatwick Airport: Drones ground flights. URL https://www.bbc.com/news/uk-england-sussex-46623754.

[15] Gazebo Drone Simulation. URL https://dev.px4.io/v1.9.0/en/simulation/gazebo.html.

[16] Google Wing Drone Delivery. URL https://vtnews.vt.edu/articles/2019/10/ictas-wingdronedeliverylaunch.html.

[17] GTKterm Serial Terminal. URL https://linux.die.net/man/1/gtkterm.

[18] HCL AppScan home. URL https://www.hcltechsw.com/wps/portal/products/appscan/offerings/standard.

[19] Gazebo Home. URL http://gazebosim.org/.

[20] JSMN JSON parser. URL https://github.com/zserge/jsmn.

[21] JSON introduction. URL https://www.json.org/json-en.html.

[22] Xilinx Mailbox IP. URL https://www.xilinx.com/support/documentation/ip_documentation/mailbox/v2_1/pg114-mailbox.pdf.

[23] MAVLink home. URL https://mavlink.io/en/.

[24] Mission Planner home. URL http://ardupilot.org/planner/.

[25] Zynq UltraScale+ MPSoC Data Sheet. URL https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[26] Xilinx Zynq UltraScale+ MPSoC. URL https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[27] Drone activity halts air traffic at Newark Liberty International Airport. URL https://www.washingtonpost.com/transportation/2019/01/22/drone-activity-halts-air-traffic-newark-liberty-international-airport/.

[28] Dronezon Top Collision Avoidance Drones. URL https://www.dronezon.com/learn-about-drones-quadcopters/top-drones-with-obstacle-detection-collision-avoidance-sensors-explained/.

[29] MAVLink packet format. URL https://mavlink.io/en/guide/serialization.html.

[30] Pixhawk home page. URL http://pixhawk.org/.

[31] QGroundControl - Plan File Format. URL https://dev.qgroundcontrol.com/en/file_formats/plan.html.

[32] Digilent Getting Started with Digilent Pmod IPs. URL https://reference.digilentinc.com/learn/programmable-logic/tutorials/pmod-ips/start.

[33] Inclusion of a Point In a Polygon. URL http://geomalgorithms.com/a03-_inclusion.html.

[34] Amazon Prime Air. URL https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011.

[35] PX4 home page. URL https://px4.io/.

[36] QGroundControl home. URL http://qgroundcontrol.com/.

[37] Realterm Serial Terminal. URL https://realterm.sourceforge.io/.

[38] Runtime Verification about. URL https://runtimeverification.com/about/.

[39] Singapore Changi Airport shuts runway over drone sighting. URL https://www.zdnet.com/article/singapore-changi-airport-shuts-runway-over-drone-sighting/.

[40] EPITA Spot tool installation. URL https://spot.lrde.epita.fr/install.html.

[41] Xilinx What is an FPGA? URL https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[42] Delivery drones: Where are they when we really need them? URL https://www.theregister.co.uk/2020/03/30/why_no_delivery_drones_during_coronavirus_lockdown/.

[43] Xilinx AXI Timer. URL https://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.pdf.

[44] Xilinx ZCU104 Evaluation Board User Guide. URL https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf.

[45] MAX 10 FPGA Device Architecture. Technical report, Intel, 2017. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/max-10/m10_architecture.pdf.

[46] Intel MAX 10 FPGA Device Overview, 2017. URL https://www.intel.com/content/www/us/en/programmable/documentation/myt1396938463674.html#myt1396939274982.

[47] Enhanced Bounded Model Checker - A model checker for hardware designs, 2019. URL http://www.cprover.org/ebmc/.

[48] Intel High Level Synthesis Compiler, 2019. URL https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[49] Avalon Interface Specifications, 2019. URL https://www.intel.com/content/www/us/en/programmable/documentation/nik1412467993397.html.

[50] Translate synthesizable VHDL into Verilog 2001, 2019. URL http://doolittle.icarus.com/~larry/vhd2vl/.

[51] Juan-Pablo Afman, Laurent Ciarletta, Eric Feron, John Franklin, Thomas Gurriet, and Eric N. Johnson. Towards a new paradigm of UAV safety. *CoRR*, abs/1803.09026, 2018. URL http://arxiv.org/abs/1803.09026.

[52] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[53] ASTM International. Standard practice for methods to safely bound flight behavior of unmanned aircraft systems containing complex functions. pages 1–9, Sept 2018. doi: 10.1520/F3269-17.Copyright.

[54] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. *Introduction to Runtime Verification*, pages 1–33. Springer International Publishing, 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.

[55] H. Bassmann and P. W. Besslich. Monocular computer vision. In *Third International Conference on Image Processing and its Applications, 1989.*, pages 107–111, July 1989.

[56] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011. ISSN 1049-331X. doi: 10.1145/2000799.2000800. URL http://doi.acm.org/10.1145/2000799.2000800.

[57] 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, 2017. doi: 10.4204/eptcs.254.2.

[58] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *Lecture Notes in Computer Science*, pages 1–30. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-35746-6_1. URL https://doi.org/10.1007/978-3-642-35746-6_1.

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

[60] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design*, pages 203–222, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-49177-4.

[61] Dale DePriest. NMEA GPS. URL https://www.gpsinformation.org/dale/nmea.htm.

[62] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, October 2016. doi: 10.1007/978-3-319-46520-3_8.

[63] Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. *Runtime Failure Prevention and Reaction*, pages 103–134. Springer International Publishing, Cham, 2018. ISBN 978-3-319-75632-5. doi: 10.1007/978-3-319-75632-5_4. URL https://doi.org/10.1007/978-3-319-75632-5_4.

[64] Paul Guermonprez. Intel Aero Wiki, 2018. URL https://github.com/intel-aero/meta-intel-aero/wiki.

[65] S. Jakšić, E. Bartocci, R. Grosu, R. Kloibhofer, T. Nguyen, and D. Ničkovié. From signal temporal logic to FPGA monitors. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 218–227, Sep. 2015. doi: 10.1109/MEMCOD.2015.7340489.

[66] Aaron Kane, Omar Chowdhury, Anupam Datta, and Philip Koopman. A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification*, pages 102–117, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23820-3.

[67] B. Khoussainov and A. Nerode. *Automata Theory and its Applications*. Progress in Computer Science and Applied Logic. Birkhäuser Boston, 2012. ISBN 9781461201717.

[68] Parimal Kopardekar, Joseph Rios, Thomas Prevot, Marcus Johnson, Jaewoo Jung, and John E Robinson. Unmanned aircraft system traffic management (UTM) concept of operations. 2016.

[69] Jonathan Laurent, Alwyn Goodloe, and Lee Pike. Assuring the guardians. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification*, pages 87–101, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23820-3.

[70] P. Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, July 2006. ISSN 1937-4194. doi: 10.1109/MS.2006.114.

[71] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016. ISSN 0278-0070. doi: 10.1109/TCAD.2015.2513673.

[72] Thang Nguyen, Ezio Bartocci, Dejan Ničković, Radu Grosu, Stefan Jaksic, and Konstantin Selyunin. The HARMONIA Project: Hardware Monitoring for Automotive Systems-of-Systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 371–379, Cham, 2016. Springer International Publishing. ISBN 978-3-319-47169-3.

[73] 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.

[74] Lee Pike, Sebastian Niller, and Nis Wegmann. Runtime verification for ultra-critical systems. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 310–324, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-29860-8.

[75] Johann Schumann, Patrick Moosbrugger, and Kristin Y. Rozier. R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification*, pages 233–249, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23820-3.

[76] Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, and Radu Grosu. Applying runtime monitoring for automotive electronic development. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 462–469, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46982-9.

[77] D. Solet, J. Béchennec, M. Briday, S. Faucou, and S. Pillement. Hardware runtime verification of embedded software in SoPC. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–6, May 2016. doi: 10.1109/SIES.2016.7509425.

[78] © 2019 IEEE. Reprinted, with permission, from J. Stamenkovich, L. Maalolan, and C. Patterson. Formal assurances for autonomous systems without verifying application software. In *2019 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED UAS)*, pages 60–69, Nov 2019. doi: 10.1109/REDUAS47371.2019.8999690.

[79] Joseph Allan Stamenkovich. Master's thesis, Virginia Tech, 2019. URL http://hdl.handle.net/10919/89950.

Appendices

Appendix A

Flight Plan in JSON format


```
{
    "fileType": "Plan",
    "groundStation": "QGroundControl",
    "mission": {
        "cruiseSpeed": 15,
        "firmwareType": 12,
        "hoverSpeed": 5,
        "items": [
            {
                "AMSLAltAboveTerrain": null,
                "Altitude": 50,
                "AltitudeMode": 1,
                "autoContinue": true,
                "command": 22,
                "doJumpId": 1,
                "frame": 3,
                "params": [
                    15,
                    0,
                    0,
                    null,
                    47.3996986,
                    8.54600979,
                    50
                ],
                "type": "SimpleItem"
            },
            {
                "AMSLAltAboveTerrain": null,
                "Altitude": 50,
```

```
31              "AltitudeMode": 1,
32              "autoContinue": true,
33              "command": 16,
34              "doJumpId": 2,
35              "frame": 3,
36              "params": [
37                  0,
38                  0,
39                  0,
40                  null,
41                  47.39818343845968,
42                  8.545683635609919,
43                  50
44              ],
45              "type": "SimpleItem"
46          },
47          {
48              "AMSLAltAboveTerrain": null,
49              "Altitude": 50,
50              "AltitudeMode": 1,
51              "autoContinue": true,
52              "command": 16,
53              "doJumpId": 3,
54              "frame": 3,
55              "params": [
56                  0,
57                  0,
58                  0,
59                  null,
60                  47.398267218710025,
61                  8.544297634907053,
62                  50
63              ],
64              "type": "SimpleItem"
65          },
66          {
67              "AMSLAltAboveTerrain": null,
68              "Altitude": 50,
69              "AltitudeMode": 1,
```

```
70                  "autoContinue": true,
71                  "command": 16,
72                  "doJumpId": 4,
73                  "frame": 3,
74                  "params": [
75                      0,
76                      0,
77                      0,
78                      null,
79                      47.39904339,
80                      8.54390969,
81                      50
82                  ],
83                  "type": "SimpleItem"
84              },
85              {
86                  "AMSLAltAboveTerrain": null,
87                  "Altitude": 0,
88                  "AltitudeMode": 1,
89                  "autoContinue": true,
90                  "command": 21,
91                  "doJumpId": 5,
92                  "frame": 3,
93                  "params": [
94                      0,
95                      0,
96                      0,
97                      null,
98                      47.39962203,
99                      8.54513533,
100                     0
101                 ],
102                 "type": "SimpleItem"
103             }
104         ],
105         "plannedHomePosition": [
106             47.39996839617687,
107             8.54600979,
108             479
```

```
109             ],
110             "vehicleType": 2,
111             "version": 2
112         },
113         "version": 1
114 }
```