A Simplex Architecture for Intelligent and Safe Unmanned Aerial Vehicles

By

Prasanth Vivekanandan

Submitted to the Department of Electrical Engineering and Computer Science and the
Graduate Faculty of the University of Kansas
in partial fulfillment of the requirements for the degree of
Master of Science

Prof.Yun,Heechul, Chairperson

Committee members

Prof.Kulkarni,Prasad

Prof.Luo,Bo

Date defended:

The Thesis Committee for Prasanth Vivekanandan certifies
that this is the approved version of the following thesis :

A Simplex Architecture for Intelligent and Safe Unmanned Aerial Vehicles

Prof. Yun,Heechul, Chairperson

Date approved: _____

Abstract

Unmanned Aerial Vehicles (UAVs) are increasingly demanded in civil, military and research purposes. However, they also possess serious threats to the society because faults in UAVs can lead to physical damage or even loss of life. While increasing their intelligence, for example, adding vision-based sense-and-avoid capability, has a potential to reduce the safety threats, increased software complexity and the need for higher computing performance create additional challenges software bugs and transient hardware faults that must be addressed to realize intelligent and safe UAV systems.

In this thesis, we present a fault tolerant system design for UAVs. Our proposal is to use two heterogeneous hardware and software platforms with distinct reliability and performance characteristics: High-Assurance (HA) and High-Performance (HP) platforms. The HA platform focuses on simplicity and verifiability in software and uses a simple and transient fault tolerant processor, while the HP platform focuses on intelligence and functionality in software and uses a complex and high performance processor. During the normal operation, the HP platform is responsible for controlling the UAV. However, if it fails due to transient hardware faults or software bugs, the HA platform will take over until the HP platform recovers.

We have implemented the proposed design on an actual UAV using a low-cost Arduino and a high-performance Tegra TK1 multi core platform. Our case-studies show that our design can improve safety without compromising performance and intelligence of the UAV.

Acknowledgements

First and foremost I would like to thank Dr.Heechul Yun who is my advisor. This work was possible mainly because of his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

Besides my advisor, I thank Dr.Shawn Keshmiri for his help and support. This is an inter disciplinary work involving Aerospace and Computer science and Dr.Keshmiri supported me by providing the required resource and advise for this work. I would like to thank Dr.Gonzalo Garcia for patiently answering all my questions on the autonomous controller design.

I would like to thank all the members of our research group Farzad Farshchi, Waqar Ali and Elise McEllhiney for being very friendly and helpful. A special thanks to Prathap Kumar Valsan for his support and valuable advice.

Most importantly, I would like to thank my family. I owe my parents a great debt of gratitude for teaching me valuable life lessons and always helped to keep my spirits up.

Publication

This work has led to the following publication:

P. Vivekanandan, G. Garcia, H. Yun and S. Keshmiri, "A Simplex Architecture for Intelligent and Safe Unmanned Aerial Vehicles," 2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Daegu, 2016, pp. 69-75. doi: 10.1109/RTCSA.2016.17 **nominated for best student paper award**

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use of Unmanned Aerial Vehicles (UAVs) is rapidly increasing in recent years due to diverse recreational, commercial, and military applications. Modern UAVs have begun incorporating a range of new technology, including electronic sensors that stabilise them. Some models can even be programmed to fly set paths or patterns. Another good example is a UAV with a small camera attached to it that gives the birds-eye view of the surroundings.

These advancements have not only seen UAVs becoming popular amongst even the most casual of hobbyists, but also in commercial endeavours. Large companies such as Facebook, Google and Amazon have already invested heavily in research related to drones, and even smaller companies such as florists, pizza restaurants (and even real-estate agencies) have started to investigate how drones can be used in their industry. More recently news agencies have begun to look at using drones when covering certain news stories, and Hollywood has already started using drones during the production of movies and TV shows.

Most of the UAVs work in close proximity with the humans and the physical environment. Hence, there is an increasing demand for intelligent UAV systems that are cognizant of the surrounding environment and perform sophisticated tasks including collision avoidance.

An intelligent UAV system requires integration of advanced sensor packages (e.g. vision) and high computational performance to process the enormous amount of real-time sensor data and to

execute complex algorithms in a timely manner. The rapidly increasing computing capacity of modern embedded computing platforms multiple CPU cores, GPU, and other accelerators makes it feasible to develop such a UAV system while satisfying size, weight, and power (SWaP) requirements of UAVs. In order to get the full benefits of this powerful hardware, a powerful software frame work like CUDA for GPU applications, opencv for vision based application and so on is required. The combination of powerful hardware together with the powerful software makes the computing platform in the UAV to look almost like a desktop computer.

1.1 Motivation

The push for higher intelligence in UAV systems also creates serious side effects in terms of safety and reliability. First, the complexity of software systems is rapidly increasing, which makes it difficult to weed out software bugs. For example, an intelligent flight control system with vision based collision avoidance capability, which in itself can be complex and difficult to verify, may also depend on complex middleware packages (e.g., Robot Operating System, ROS [32]) and the OS (e.g., Linux), each of which may be comprised of multi-million lines of code.

Second, to achieve high intelligence, the use of high performance computing platforms is necessary. However, high performance computing platforms are increasingly prone to transient hardware faults (soft errors) due to environmental effects such as single-event upsets (SEUs) [15]. SEUs are caused by high energy particle strikes from cosmic rays [39] which result in bit flips. The technology treads to develop efficient and high-performance processors shrinking dimensions and operating voltage, and increased frequency and density have dramatically increased the possibilities of SEUs [15], which could result in unexpected failures in the system [23, 36]

There has been a large body of research in the control systems community regarding the design of fault tolerant control of UAV systems [34]. While these fault tolerant controllers are designed to handle structural damage, actuator and sensor failures, they typically do not handle system-level failure such as on-board computing platform malfunction, which prevents execution of the control algorithms in the first place. System-level reliability can be generally improved by redundancy. A

well-known technique is triple modular redundancy (TMR) [26] in which three identical systems produce control outputs in order to survive from failures of any one of the system. However, the size, weight, and power considerations as well as the cost make the TMR solution undesirable, especially in small UAVs.

1.2 Contribution

To address the safety and reliability challenges of UAV systems, we present a UAV system design and implementation based on Simplex architecture [35]. The main idea of the Simplex architecture is that a simple verifiable controller provides safety of the system, while a complex, high-performance controller strives to achieve high system performance. The choice between the two controllers is determined by a decision module, which constantly assesses the safety status of the system and makes the decision.

Our contribution is a novel application of the Simplex architecture to develop both intelligent and reliable UAV systems in a cost effective manner. Specifically, we realize the Simplex architecture using two heterogeneous hardware platforms with distinct reliability and performance characteristics. The idea is that we use a reliable but less performance hardware platform, which we call a High-Assurance (HA) platform, to be responsible for safety while we use a more performance, but potentially less reliable platform, which we call a High-Performance (HP) platform, for performance and intelligence of the UAV. Only the HA platform both software and hardware forms the trusted computing base (TCB) of the UAV. The HP platform, on the other hand, can be affected by software bugs and SEUs, which could result in failures (e.g., crashes), but the whole system safety will still be insured by the safety platform.

1.3 Implementation

We applied the proposed design in implementing a custom fixed-wing UAV system, shown in Figure 1.1, using a low-cost Arduino as our HA platform and a Tegra TK1 as HP platform. We

Figure 1.1: DG 808S with our custom built avionics

demonstrate the ability to recover from crashes through a set of fault-injection experiments in a hardware-in-the-loop simulation setup. The results suggest that the proposed design can substantially increase safety of intelligent UAVs.

1.4 Thesis Overview

The rest of the thesis is organized as follows. Chapter 2 discusses various fault types in a UAV and related background. Chapter 3 presents the proposed UAV simplex architecture. Chapter 4 describes our implementation of a fixed-wing UAV based on the proposed architecture. Chapter 5 presents our evaluation results and we conclude in Chapter 6.

Chapter 2

Background and Related Work

In this section, we discuss various types of faults that can occur in a UAV and safety conditions that should be followed by the aircraft in order to be safe and reliable. We then discuss related work.

2.1 Faults

An UAV is a cyber-physical system, which includes cyber part (computing hardware and software) and physical part (sensors, actuators, UAV frame, and etc).

Faults in physical components of an UAV and their handling has been well studied in the control and aerospace communities. For example, structural damage or bias in sensor readings can be tolerated by advanced adaptive control algorithms that take such effects into account. A comprehensive review on the topic can be found in [34]. However, advanced control algorithms must be realized in software binaries running on computing platforms i.e., the cyber system.

Faults in the cyber system can arise from a number of different reasons. The most common type of faults are logical faults in software i.e., software bugs. As the demand for higher intelligence and functionality increase, the size and complexity of flight control software keeps increasing [20]. While model-based design and verification methodologies have made great progress [21, 20, 16], it is still difficult to completely weed out all bugs.

Another type of faults are *temporal faults*, which occur when the real-time requirements (i.e., deadlines) of various tasks in the system are not met. Temporal correctness is difficult to guarantee especially in multi core architecture because of uncontrolled sharing of many performance critical hardware resources such as cache and DRAM among concurrently executing tasks can cause highly variable timing [25, 30].

Third, often overlooked but increasingly important type of faults are *transient hardware faults* due to single event upsets (SEUs). SEUs are caused by cosmic radiation and alpha particles [39, 27] resulting bit flips. Because SEUs can occur anywhere in SRAM, registers, and combinational logic, it can cause, for example, an invalid instruction exception, a parity error, a memory access violation, a wrong conditional branch, and ultimately a system crash [36]. As technology scaling continues (i.e., more transistors in a chip), sensitivity to radiation has dramatically increased [15]. Traditional circuit-level solutions—special circuit design and manufacturing process—are not only expensive but also often lag several generations behind the state-of-the-art processors [18]. This is a serious impediment to develop intelligent UAVs that require cutting edge high performance computing capabilities.

2.2 Safety Conditions

In a UAV, the controller program acts in conjunction with the physical dynamics of the system. Based on the dynamics, the system has safety conditions that needs to be respected to maintain the safety of the UAV. Aircraft stall is one example for the safety condition [2]. When an aircraft flies at a certain altitude, it makes an angle called angle of attack with the wind. If the autonomous controller gives output such that the aircraft starts to climb then this angle of attack increases. Once the angle of attack crosses the threshold value (varies from one aircraft to another and it is determined using the aircraft's dynamics), the wings of the aircraft will not be able to lift the aircraft.

Another safety condition that needs to be respected by the UAV system is the regulations. The UAVs operate at close proximity with humans and physical environment. The government makes

the regulations that must be followed by commercial UAVs. One such regulation is the limit on the altitude that commercial UAVs should fly. In United States the commercial UAVs are allowed to fly less than 400 feet from ground level [3].

In summary, as the demand for higher intelligence in UAVs increases, the importance of reliable computing software and hardware platform the cyber system increases. However, the increased software complexity and the use of high performance multi core processors, while necessary, would increase all three types of faults logical, temporal, and transient hardware faults of the cyber system of an UAV. The UAV systems face safety threats not only from these faults but also from the autonomous controller that is not obeying the physical constraints.

2.3 Related Work

Software reliability have long been a major concern in safety-critical cyber-physical systems. For example, software in commercial airplanes must follow certain standards (e.g., DO-178C [33] and ARINC-653 [13]) and be certified by certification authorities. At the application-level, model-based designs and formal-methods [20, 16, 21] have made significant advance, it is still difficult to verify the correctness of the software at the code-level [28]. Furthermore, the applications also rely on many other system software components, including the OS and middleware, each of which may contain bugs.

System-level reliability and fault-tolerance techniques, especially in the context of intelligent cyber-physical systems, have been studied to mitigate possible software bugs. SAFER is a middleware framework that uses software-level redundancy to enhance system-level reliability. However, such a middleware solution may not survive from an OS failure (e.g., BSOD). The seL4 is a microkernel where its functional correctness was formally verified [24]. However, functional correctness does not means its temporal correctness is guaranteed. Nor it guarantees correct operations in the presence of transient hardware faults (SEUs), which are becoming more prevalent as technology scaling continues [18]. C'Mon focuses on detecting OS-level timing faults (e.g., deadline misses due to scheduling) that are caused by SEUs [36]. To guard the OS code and data against SEUs,

dOSEK proposed to use a special encoding in storing OS code and data, which enables detection and recovery from the SEUs [23]. However, its software-based approach comes at a significant performance penalty.

SEUs can be reduced at each hardware component-level by, for example, adopting ECC and other so called "hardening" techniques [15]. Also, hardware redundancy solutions, such as Triple-modular redundancy (TMR) [26] or dual-redundancy, can protect the system from transient hardware faults [15]. In today's commercial airplanes and satellites, the physical redundancy based techniques are used due to their high safety-critical requirements [19]. However, these hardware techniques comes at substantial space and performance penalties and high cost and development time—unsuitable for UAVs. Also, the processors used in these applications often several generations behind the stat-of-the-art COTS processors [25, 17]. This is a serious problem for intelligent UAVs where high computing performance is a key to achieve high intelligence.

The simplex architecture [35] is a special form of redundancy in which safety and performance are decoupled to safety and performance controllers, respectively. It was originally implemented at the application software-level in a single computing platform and used to upgrade/verify controllers without stopping the system [35]. More recently, Bak et al. implemented the safety-critical parts of the system—safety controller and decision logic—in FPGA so that it can tolerate system-level faults in the computer system running the performance controller for an inverted pendulum and a cardiac pacemaker [14]. [10] is an extension of simplex architecture and Abdi et al showed that a software fault in cyber-physical system can be recovered by restarting the system at run-time. In [12], Abdi et al proposed a novel approach to design a controller that enables the system to restart and remain safe during and after the restart. Their approach is also based on simplex architecture and it requires only one processing unit. Hence, [12] is almost similar to the original simplex architecture, but it also provides recovery from system level faults. Another version of simplex is shown in [38] and it is used to detect intrusion in cyber-physical systems. Abdi et al in [11] proposed a version of simplex architecture that adds security to the cyber-pyhsical systems by adding a monitoring module(checks for any intrusion) inside complex system. If monitoring

module fails to detect any security failures, the periodic restart protects the system.

Chapter 3

UAV Simplex Architecture

In this chapter, we review the Simplex architecture [35] and describe our two heterogeneous platforms based approach.

The Simplex architecture is composed of three components: a safety controller, a performance controller, and a decision logic [35]. Normally, the performance controller drives the plant (in our case, the UAV) as it offers higher control performance. However, if the safety conditions of the plant are to be violated, as determined by the decision logic, the safety controller will assume the control of the plant until the performance controller is recovered (for example, by restarting it). In this way, faults in the performance controller does not cause safety failure of the system.

In the original Simplex applications, however, all three components share the same computing hardware (processor) and software platform (OS, middleware) [35]. This means that system-level faults in the shared hardware and software platform—SEUs in the processor, bugs in the OS and middleware—could still compromise the safety of the system. as shown in Table 3.1. To overcome the limitations, we propose to realize the Simplex architecture by using two platforms with distinct reliability and performance characteristics, as shown in Table 3.1. The High-Assurance (HA) platform focuses on safety and verifiability over performance and functionality. For hardware, we assume that it uses chips that are more tolerant to SEUs. While a number of techniques can be used to reduce the SEU rate of a chip—ECC, manufacturing process, and etc.—using simple, low-

Table 3.1: Platform characteristics

Platform	High-Assurance (HA)	High-Performance (HP)
Hardware	SEU resistant	SEU susceptible
Software	Verifiable	Unverifiable

density chips running at low operating frequency could also help reduce the overall SEU rate [15]. For software, we assume that the platform uses a small RTOS with proven (verified) reliability. For example, the seL4 micro-kernel's functional correctness was formally verified [24]. Also, there are many other commercial/open-source RTOSs that have been used in critical applications.

On the other hand, the High-Performance (HP) platform focuses on performance and functionality over safety. For hardware, it may use a complex, high-performance (multicore) processor, which runs at multi-gigahertz frequencies and is composed of multi-billion transistors. Generally, such a processor suffers more SEUs than a simpler, low-performance one because higher density and operating frequency negatively affect SEUs [15]. For software, it may use a rich OS (e.g., Linux) and middleware solutions (e.g., Robot Operating System: ROS) to offer sophisticated capabilities needed to implement high intelligence. However, each of the software package may be comprised of many million lines of code, which makes it very difficult, if not impossible, to verify its correctness.

Among the three components in the Simplex architecture, we assign the safety controller and the decision logic on the HA platform, while assign the performance controller on the HP platform.

The performance controller may implement an advanced control algorithm and intelligent sensing capabilities, such as vision-based collision avoidance, that require high computing performance and rich OS and middleware support. The HP platform offers such computing power and functionality but it does not guarantee safety and reliability. On the other hand, the safety controller implements a simple and proven control algorithm that is always ready to take over the performance controller, if necessary, as determined by the decision logic. In our design, as in the original Simplex architecture, correct functioning of the safety controller and the decision logic is required. In addition, the HA platform provid es a safe and reliable execution environment although it may

Table 3.2: Fault detection

Observed behaviors	Faults in the HP platform
No output	OS/controller crash
Delayed output	Deadline miss
Unsafe output	Bugs, SEUs, bad controller design, etc.

not provide high computing performance or rich middleware support.

3.1 Fault Model

We assume that faults can occur only in the HP platform. In other words, we trust the correct functioning of the HA platform the safety controller and decision logic. To ensure this, hardware of the HA platform must be resistant to transient hardware faults, and the complexity of its software must be limited to a level that can be rigorously tested and possibly verified. On the other hand, we do not trust both hardware and software of the HP platform—i.e., the OS can crash, the performance controller may crash or produce invalid outputs (e.g., NaN output) or simply miss the deadline due to resource contention. Also, the HP platform can suffer transient hardware faults (SEUs) which lead to system crash or application failure.

Lastly, in this work, we do not consider physical faults, such as structural damage, bias in sensor readings, sensor and actuator malfunction, and so on. We assume that these physical issues are tolerated by adaptive control algorithms, which have been extensively studied in the aerospace engineering community [34].

3.2 Safety Region

Physical systems, especially UAVs, have limits and constraints that need to be respected. For example, a UAV should have angle of attack less than critical angle of attack [2] to avoid stall condition. Safety region S is a subset of State Space Model SSM where all this constraints are respected.

$$SSM = Ax + Bu \tag{3.1}$$

$$S \subseteq SSM \tag{3.2}$$

Where x denotes the states of the system and u represents the inputs to the system. A and B matrices are the dynamics of the system. The safety region can be defined by the following equation.

$$S = \{x | S_x \cdot x \leq 1\} \tag{3.3}$$

Equation 3.3 conveys that for every state x in the system state space model SSM there exists a safety matrix S_{x} such that dot product of safety matrix and their corresponding state should be less than or equal to one. If the system state is outside this safety region then the system is said to be unsafe.

Safety region also considers the limits on the actuators. The actuators have their range for operation. Hence this range should also be considered while defining the safety region. The safety region that defines this operating range for the actuators are formulated as follows.

$$S_u = \{u | S_u \cdot u \leq 1\} \tag{3.4}$$

Equation 3.4 conveys that, for every actuator value generated by the controller algorithm or the input to the system u in the system state space model, there exists a safety matrix S_{u} such that dot product of safety matrix and their corresponding system input should be less than or equal to one. If the controller generates the value to the actuators outside this region, then actuators will not be able to handle them.

Chapter 4

Prototype Avionics

We have designed an avionics based on the UAV simplex architecture. In this chapter, we discuss about the hardware and software that are used in our custom built avionics.

4.1 Hardware

4.1.1 Sensors

We categorize sensors into two groups: basic and advanced sensors. Basic sensors include GPS/IMU, airspeed, and pressure sensors and they are essential to flight. For GPS/IMU, we use a VectorNav VN-200 [8] module. For airspeed and pressure, we use an AMS 5812 pressure sensor and a pitot tube. The basic sensors are shared by both safety and performance controllers. On the other hand, advanced sensors are only used by the performance controller and considered not essential to flight although they may increase performance of the system. The advanced sensors include, for example, radars and cameras, which can be used to implement advanced sense-and-avoid capabilities. We are currently implementing a vision and radar based sense and avoid system.

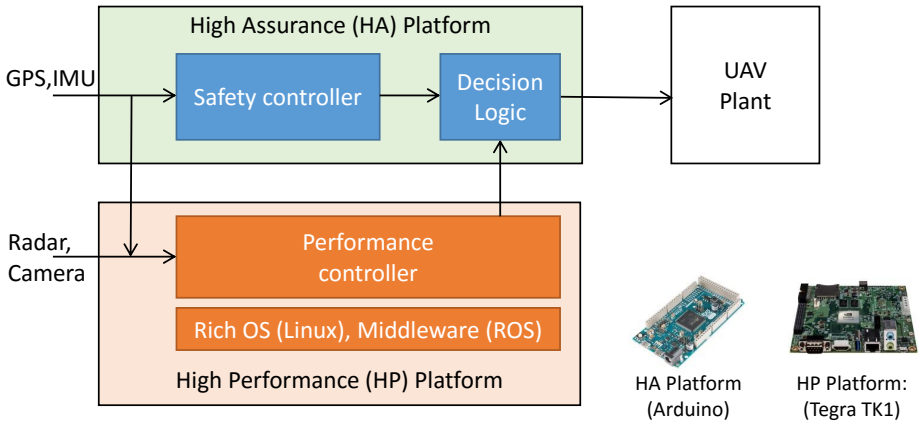

Figure 4.1: UAV simplex architecture

4.1.2 High Performance Platform

The HP platform is responsible for real-time processing of advanced sensors (e.g., vision and radar) and sophisticated control algorithms to achieve high control performance and sense-and-avoid capabilities. To satisfy the performance demand, we use Nvidia's Tegra K1 processor, which equips four ARM Cortex-A15 cores, running at 2.3 GHz, and 192 Kepler based GPU cores [31].

4.1.3 High Assurance Platform

The HA platform is ultimately responsible for safety of the system and therefore must be simple and highly resistant to transient hardware faults. For this, we use an Arduino Due platform. The platform equips a single-core Cortex-M3 processor with 80 MHz maximum operating frequency. It is based on simple in-order architecture and the number of transistors of the chip is much smaller than that of the Tegra K1. These characteristics make the HA platform less susceptible to SEUs. Also, the platform supports numerous I/O options GPIO, PWM, I2C, and etc that are needed to connect various basic sensors.

4.2 Software

4.2.1 High Performance Controller

The performance controller is implemented on the Tegra K1. We use Ubuntu 12.04 Linux as the OS and the Robot Operating System (ROS) as the middle ware framework. In ROS, the system, a robot, is composed of a set of nodes, each of which is a separate Linux process participating in the communication network for the robot. A node can publish messages to a communication channel, called a topic, which can be subscribed by other nodes to receive the published messages. Figure 4.2 shows the nodes and topics of the performance controller. In the figure, the controller DG808 node is the main control node. The node subscribes the `/vectornav/ins` topic, to receive GPS and IMU sensor values, and the `/arduino/pwm`, to receive commands from the remote controller (RC) and pressure sensor values. (The RC is used for manual control of the UAV). The controller node publishes control outputs to the `servo_op` topic, which is then subscribed by the arduino node and is forwarded to the HA platform via serial. The MICORHARD ground station node is responsible for communicating with the remote ground station on the laptop PC. We use the open-source Qgroundcontrol [6] as the ground station software. We use a microhard modem for long range communication [4] between the UAV and the ground station to send/receive statistics (UAV to groundstation) and waypoints (groundstation to UAV) Currently, the performance controller only uses the basic sensors, and it does not utilize radar and vision sensors; we are actively developing radar and vision-based sense-and-avoid capability in the performance controller

4.2.2 High Assurance Controller

The safety controller runs on the Arduino Due platform. The control algorithm is a simple PID based one [22] and it is modeled and validated using Matlab Simulink [7]. Figure 4.3 shows the simulink model. We then generate C code of the controller using Matlab Simulink Coder. In this way, we minimize the possibility of bugs in the safety controller, although it does not guarantee the absence of bugs in the model. In fact, our initial model contained a divide zero bug, which was

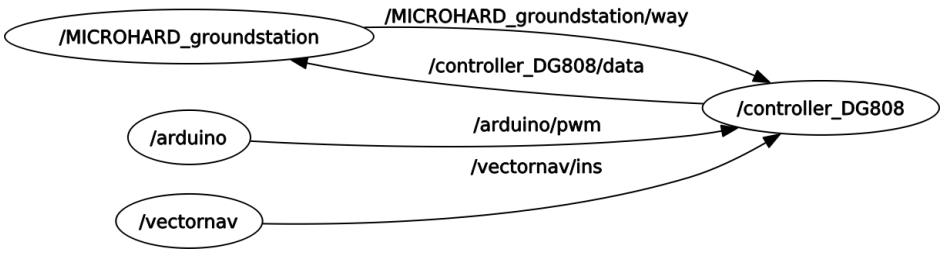

Figure 4.2: ROS based software architecture on the highperformance platform (Tegra K1)

manifested only in a certain sensor input value ranges. Nevertheless, model-based design is highly desired for a safety controller as it can significantly reduce the possibility of coding mistakes and other common software bugs [20, 16].

4.2.3 Decision Logic

The decision logic also runs on the Arduino Due, along with the safety controller. In fact, the generated C code of the safety controller is merged with the decision logic. The safety controller is directly called by the decision logic while the output of the performance controller is received asynchronously from the Tegra K1 via serial. Which output is used to actuate the UAV is then determined by the decision logic. Figure 4.4 shows the overall process of the decision logic. Note that each loop is executed periodically at a regular interval (20Hz in our current implementation). The implementation assumes that the safety controller always completes within the interval without errors—i.e., `out_hap` is always valid and computed within the interval (deadline). On the other hand, the output of the performance controller can be deemed unsafe or invalid due to a number of reasons.

4.3 Fault Detection and Recovery

The decision logic detects faults in the HP platform by observing its outputs (See Line 14 in Figure 4.4). Once a fault is identified, the decision logic switches its control to the safety controller and tries to recover the HP platform by restarting the system.

Table II shows the observations by the decision logic and the corresponding faults at the HP

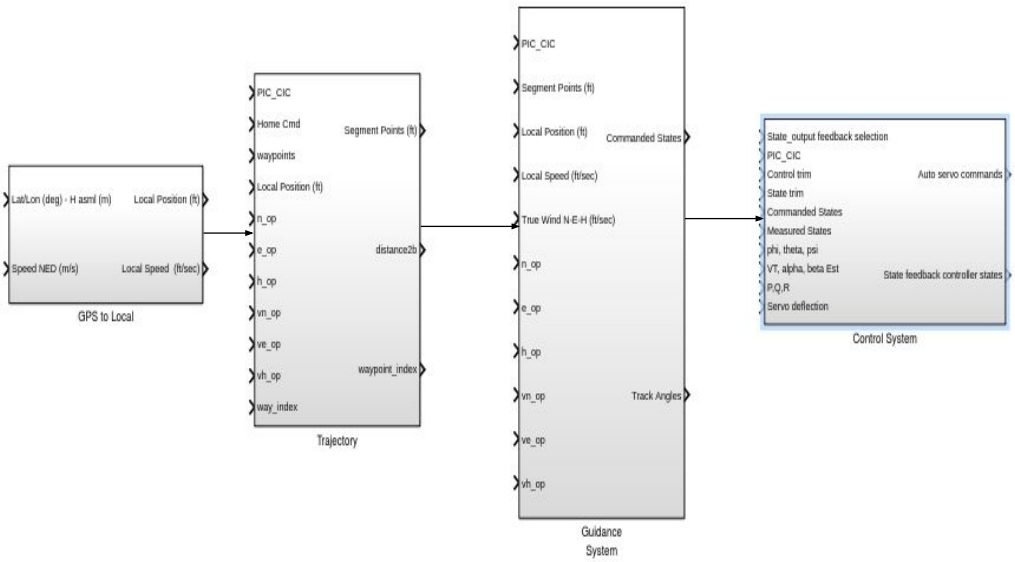

Figure 4.3: Autonomous controller block diagram

```
1  void loop()
2  {
3      // basic sensor input
4      sensor_data = read_sensors();
5      send_to_HPP(sensor_data);
6
7      // execute safety controller
8      out_hap = safety_controller(sensor_data);
9
10     // wait for the performance controller
11     out_hpp = receive_from_HPP(timeout);
12
13     // decision logic
14     if (decision_check(sensor_data,out_hpp));
15         run_servo(out_hpp);
16     else {
17         run_servo(out_hap);
18         // recover HPP
19         try_recover_hpp();
20     }
21
22     sleep_until_next_period();
23 }
24
25 int decision_check(states,hpp_output)
26 {
27     // condition for faults check
28     if(detect_faults())
29         return 0;
30
31     // conditions for safety check
32     // check if system states are inside safety region
33     if(S_states·states ≥ 1)
34         return 0;
35     // check if HPP outputs are inside actuator safety region
36     if(S_actuators·hpp_output ≥ 1)
37         return 0;
38 }
```

Figure 4.4: Decision logic on Arduino.

platform. First, 'No output' means that the decision logic does not receive performance controller's control outputs. This can be caused by software faults such as OS or controller crashes. In our current implementation, the decision logic waits one more control period before it makes a switch to the safety controller. If the outputs are received in the next period, the decision logic's observation is 'Delayed output' and the system continues to use the performance controller. The aircraft states for every cycle, must be inside a safety region in order to maintain the stability of the system.

4.3.1 Safety Region

The safety region depends on the aircraft dynamics and hence this region varies from one aircraft to another. The safety region for the UAV is defined based on Section 3.2. We added our custom built avionics whose design is based on our UAV simplex architecture to a glider named DG808 that is shown in Figure 1.1. The safety region for this UAV is defined based on its states and the inputs to the system.

The critical angle of attack for the DG808 glider is 30 deg [1]. Therefore the safety region containing the set of states where UAV will not have stall condition is given by,

$$S = \begin{pmatrix} 0 & 1/30 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/30 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} airspeed \\ angleofattack \\ sideslipangle \\ roll \\ pitch \\ P \\ Q \\ R \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{4.1}$$

The actuators used in this UAV have their operation angles. The servo motors used for aileron and rudder are capable of handling deflection angles from -15 to +15 deg where as the servo motor

[1] The critical angle of attack for DG808 was calculated using Advanced Aircraft Analysis software. [1].

assigned to the elevator operates with in -20 to +20 deg. Therefore the actuator safety region is given by,

$$S_u = \begin{pmatrix} 1/15 & 0 & 0 \\ -1/15 & 0 & 0 \\ 0 & 1/15 & 0 \\ 0 & -1/15 & 0 \\ 0 & 0 & 1/20 \\ 0 & 0 & -1/20 \end{pmatrix} \begin{pmatrix} aileron \\ rudder \\ elevator \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{4.2}$$

4.3.2 Decision Logic Conditions

In this section we define the decision logic conditions for DG808 glider. In every cycle, the outputs of high performance platform are sent to the decision logic and evaluated against all the conditions mentioned in this section. If the controller output is valid against all these conditions then the decision logic says that high performance platform is not safe and allows it to handle the system. The conditions are described using Linear Temporal Logic (LTL) which is a mathematical language for describing linear time properties [29].

4.3.2.1 Condition 1: Valid Controller Output

- Aim : To check whether the controller generates the outputs within the servo motor range. The controller may diverge and produce outputs that cannot be handled by the servo motors or the controller program might have crashed.

- LTL condition :

$$-15\,deg \;\leq\; \Box\, aileron \;\leq\; 15\,deg \tag{4.3}$$

$$-20\,deg \;\leq\; \Box\, elevator \;\leq\; 20\,deg \tag{4.4}$$

$$-15\ deg\ \leq\ \Box\ rudder \leq\ 15\ deg \tag{4.5}$$

- Explanation : The controller generally produces aileron, elevator and rudder surface outputs. These outputs are given to the servo motors. The servo motors have operation range. This condition is used to verify whether the controller produces outputs within the operation range or not.

4.3.2.2 Condition 2: Kalman Filter Divergence

- Aim : Most of the autonomous system will use kalman filters in their logic to estimate some of the missing sensor values or to reduce the noise level in the sensor data. These kalman filter, if designed with improper gains and coefficients, are highly unstable and will diverge the controller.

- LTL condition : The controller program in our test bed estimates the side slip and angle of attack.

$$-30\ deg\ \leq\ \Box\ \alpha \leq\ 30\ deg\ ^{2} \tag{4.6}$$

  Where α is the angle of attack

$$-30\ deg\ \leq\ \Box\ \beta \leq\ 30\ deg\ ^{2} \tag{4.7}$$

  where β is the side slip angle

- Explanation : The side slip and angle of attack estimation should always be converged. If this condition fails then the kalman filter has diverged. The failure of this condition is due to improper value for the filter gains and co efficient. This failure will eventually lead to failure of condition in the section 4.3.2.1.

[2]This value is calculated using AAA software [1] based on the vehicle dynamics.

4.3.2.3 Condition 3: Sensor Communication with the Performance Controller

- Aim : In UAV simplex architecture, the performance controller is not directly connected to the sensors. All the sensors are connected to the assurance platform and then passed to the performance controller through a communication medium. In our glider UAV, we used the USB communication between the platforms. Unfortunately, the USB communication is not always reliable. The connection can be lost due to some vibrations to the system and it can restored by re-spawning the program. In order to restore the communication, we currently treat this as an safety issue and restart the performance platform.

- LTL condition : The high performance platform always monitors the USB health condition and sends this health status to the decision logic in high assurance platform.

$$\Box\, USB \models TRUE \tag{4.8}$$

  where USB denotes the USB health status

- Explanation : The communication between high performance platform and the sensors is always monitored by the high performance platform and this health condition should always be good for the high performance controller. This failure will eventually lead to failure of condition in the sections 4.3.2.1 and 4.3.2.2.

4.3.2.4 Condition 4: Stall Speed Condition

- Aim : An aircraft will experience a stall condition when its angle of attack is greater than threshold value.

- LTL condition :

$$\Box\, ((alt > alt_0) \rightarrow (v_{air} > v_{stall})) \tag{4.9}$$

  where alt is the altitude measured and alt_0 is the ground altitude, v_{air} is the air speed measured by the sensor and v_{stall} is the stall speed of the aircraft and its value depends on the

flight dynamics.

- Explanation : In most of the small commercial UAVs, there is no dedicated sensor for measuring this angle. But the stall condition can be checked using the altitude and air speed measured from GPS sensor and the pitot tube respectively. The condition given in Equation 4.9 shows that when the aircraft is air bound identified by monitoring the altitude value from GPS sensor, then the air speed of the vehicle should be always greater than the stall speed. If this condition fails, then it indicates that the aircraft is climbing up at a high angle such that the pitot tube is not able read the air speed.

4.3.2.5 Condition 5: FAA Regulation on Altitude

- Aim : The Federal Aviation Administration (FAA) has regulations like the commercial UAV should always fly within 400 feet [3] from ground. This condition checks whether the high performance controller is obeying this regulation.

- LTL condition :

$$\Box\,(alt \leq 400) \tag{4.10}$$

- Explanation : When the high performance controller produces output that makes the aircraft to climb altitude above 400 feet from ground level, then this condition fails and the decision logic switches to high assurance controller and restarts the high performance controller.

Chapter 5

Evaluation

In this chapter, we evaluate our UAV simplex architecture through a case study using Hardware-in-the-Loop (HiL) simulation. We also talk about the flight test using our custom built avionics. More detailed explanation regarding the flight test is described in Section 5.2.

5.1 HiL Test

Hardware-in-the-Loop (HiL) simulation is a technique that is used for testing control systems. In UAVs the physical parts (sensor and actuators) are connected with the cyber part(controller). In HiL testing the physical parts are replaced by simulation(flight simulator) that can mimic the sensors and actuators of the UAV.

5.1.1 Setup

Figure 5.1 shows the Hardware-in-the-Loop (HIL) experiment setup that we used to evaluate our Simplex based UAV system. The experiment setup consists of an Arduino Due, a Tegra TK1, and a laptop. The Arduino Due executes the safety controller and decision logic, while the Tegra TK1 platform executes the performance controller. Together, they form the cyber system of the UAV. On the other hand, the laptop runs the ground station and a custom built flight simulator. The flight simulator, which is implemented in Matlab, provides simulated sensor values to both

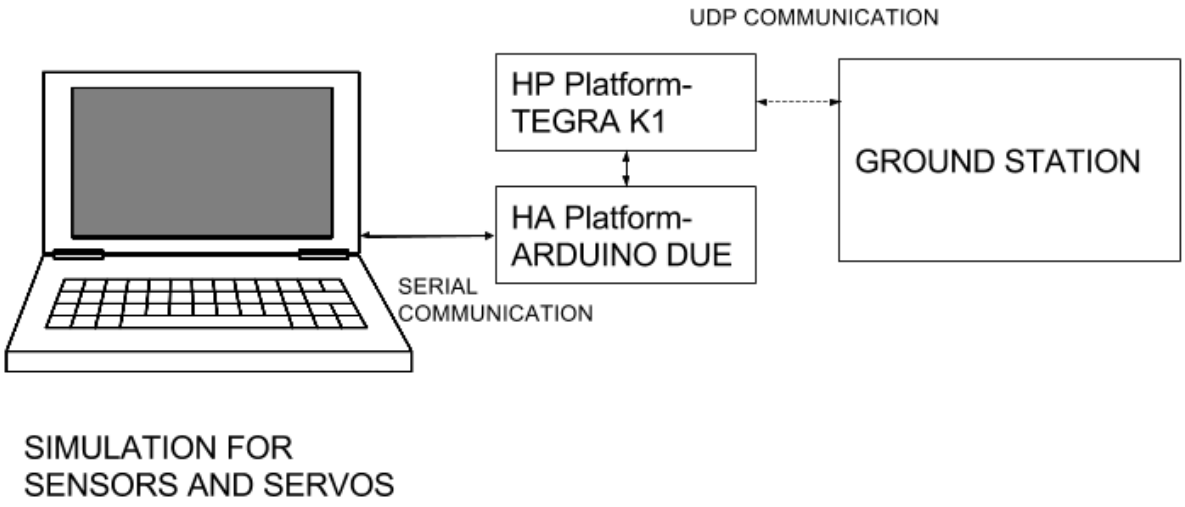

Figure 5.1: Experiment setup for UAV simplex

the controllers and in turn receives the control command surface outputs from the controllers and calculate the next inputs.

5.1.2 Case Study: Performance Controller Crash

In this case-study, we intentionally crash the performance controller in the HP platform to evaluate the system's ability to detect and recover from the fault.

Figure 5.3 shows the outputs of the performance controller (top), the safety controller (middle), and the decision logic (bottom), collected over 200 seconds duration (Note that the figure shows only one of the four control outputs.) For the first 100 seconds, both safety and performance controllers are working in parallel on HA and HP platforms, respectively. In this normal operation mode, the decision logic chooses to use the outputs of the performance controller. At 100 seconds, however, we inject a fault by manually terminating the performance controller in the HP platform. Because the decision logic observes no outputs from the performance controller, it switches to use the outputs from the safety controller. At about 130 seconds, the performance controller is restarted and produces control outputs. However, the decision logic does not immediately switch back to the performance controller because the outputs of the performance controller is not stabilized yet. The

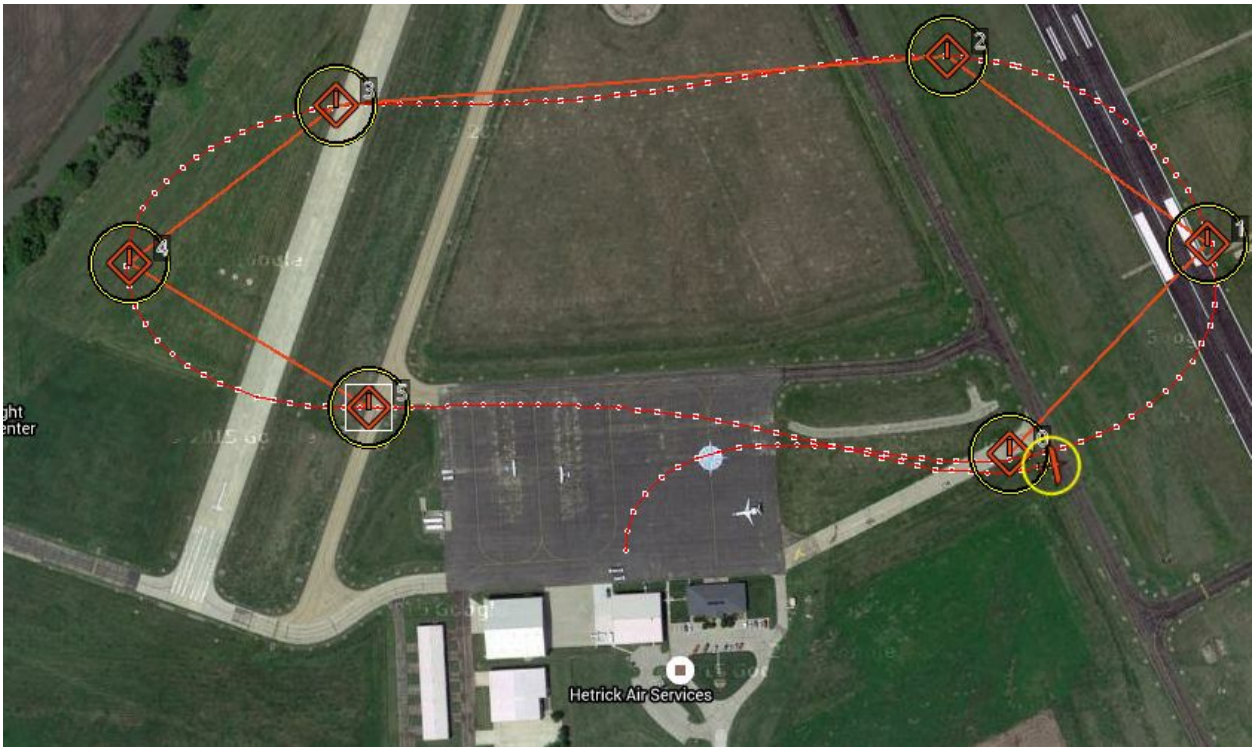

Figure 5.2: The flight path

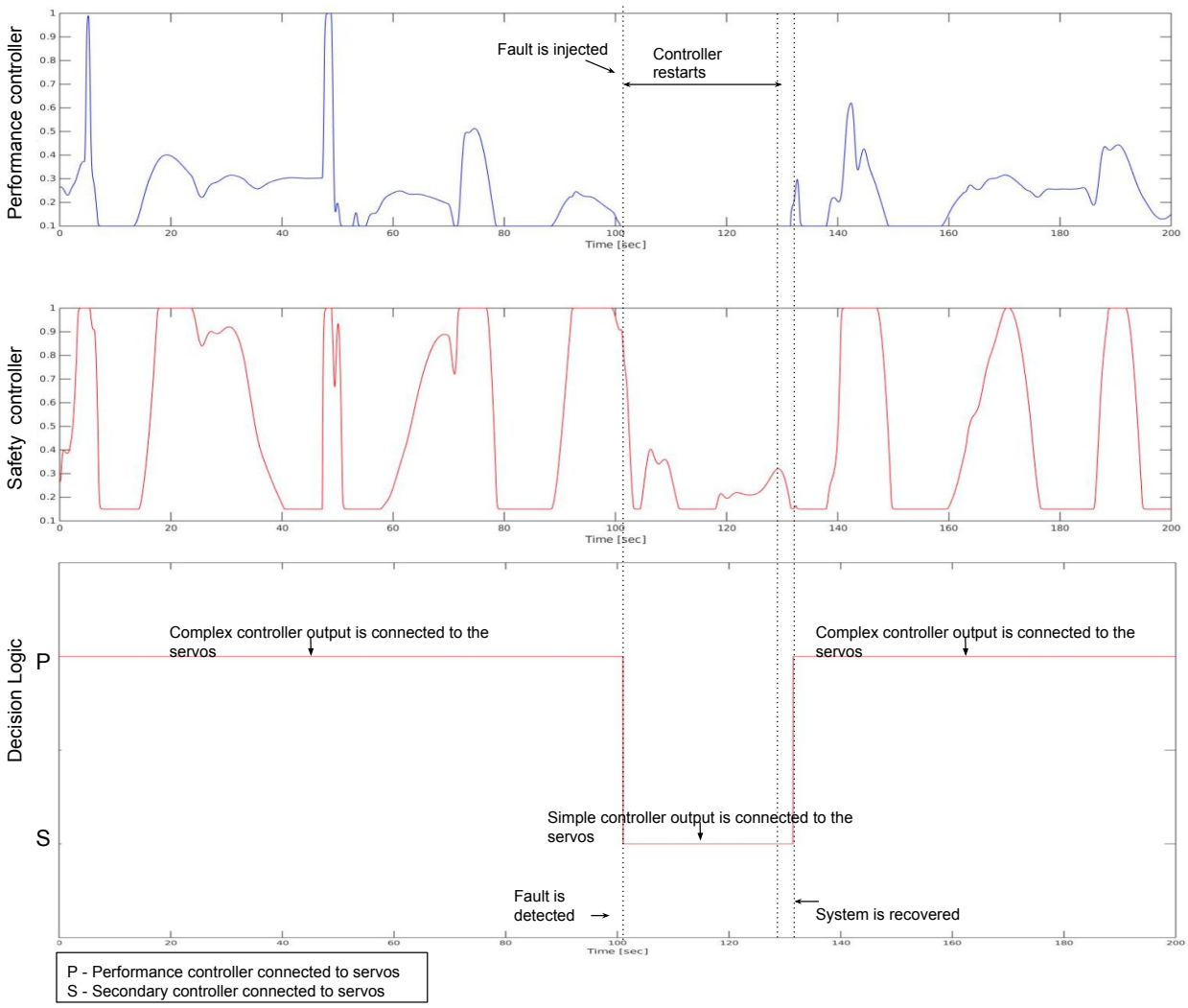

Figure 5.3: Outputs of safety controller, performance controller, and decision logic. A fault is injected at time 100 second.

stability factor depends on the convergence of the filters and the guidance logic of the controller. Once the outputs are stabilized, the decision logic switches back to use performance controller's outputs. Figure 5.2 shows the ground station tracking of the UAV system during the test. The solid orange lines denote the waypoint trajectory uploaded to the UAV system from the ground station and the red lines show the trajectory followed by the flight and the white dots represent the position update from the UAV system on the ground station. Our UAV system is designed to update the ground station at 10Hz.

5.2 Flight Test

In this section, we discuss about the test that we conducted using our avionics (mentioned in Chapter 4) in a real aircraft shown in Figure 1.1. In this test, we disabled the safety controller and decision logic in high assurance platform. Instead, we used two performance controllers in high performance platform and we manually switched the control from one performance controller to another. The aim of this test was to ensure that the aircraft is stable, while switching the control from one controller to another.

The test procedure was to takeoff the aircraft using pilot commanded RC values. Once the aircraft reached a steady state, the autopilot was engaged. The way points required by the autopilot was uploaded from the ground through ground station application. We integrated two autonomous controller programs (“autopilot1 and autopilot2") in the high performance platform of our avionics. Both the controllers mission was to follow the waypoints uploaded from ground station. Instead of using decision logic to switch control from one controller program to another controller, the ground station application was used for switching. We modified our avionics program to accept the switching command from ground station to switch between the controllers. It can be seen from the third sub plot in Figures 5.4, 5.5, 5.6 and 5.7, the control was switched from one controller to another. The sub plot shows us that at 270^{th} second of the test, value in this plot has changed from ‘0’ to ‘1’ indicating that the avionics has switched the control from “autopilot1" controller to “autopilot2" controller. The control was again switched back to “autopilot1" controller at 350^{th} second of the flight test. The switching between controllers was repeated in the shorter and longer leg of the mission trajectory to see any sort of instability or rough transition.

We have recorded the video of ground station during the test. The details about this video is mentioned in Apendix A.

Figures 5.4, 5.5, 5.6 and 5.7 shows the states of the system while it was in autopilot. The x-axis of the figures shows the time in seconds and the y-axis shows the angles in deg and rates in deg/s respectively. It can be seen from the figures that there was no sudden spike in any states during the switch between controllers (at 270^{th}, 350^{th} and 380^{th} second of the flight test) and there was no

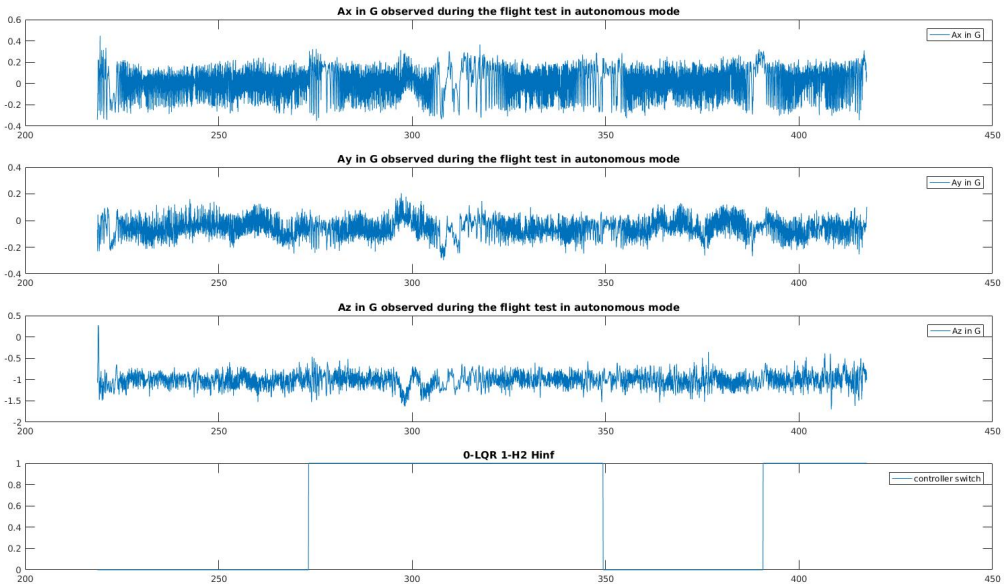

Figure 5.4: Accelerator values measured from flight test, X-axis: Time in seconds, Y-axis: Acceleration in G

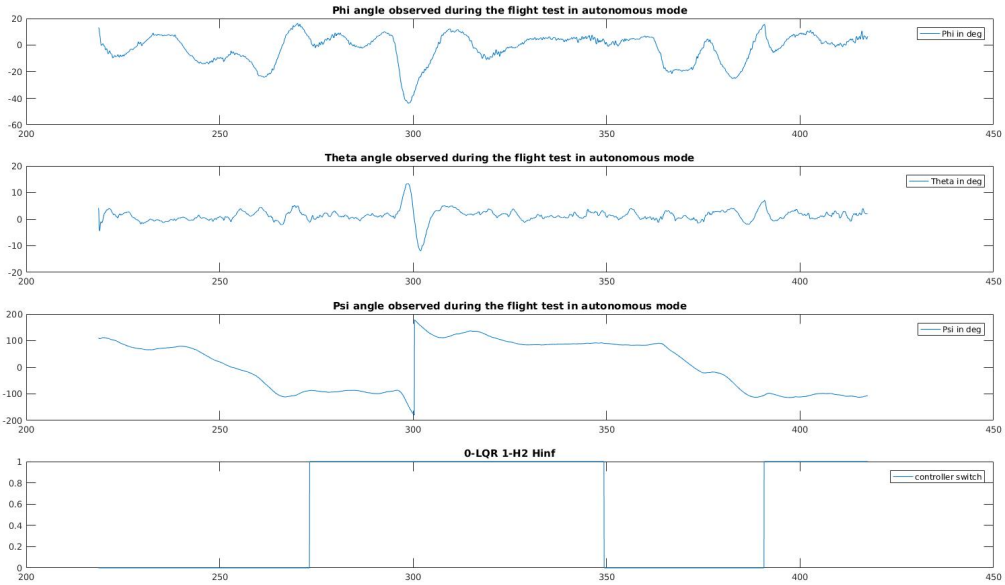

Figure 5.5: Aircraft orientation in Euler angles measured from flight test, X-axis: Time in seconds, Y-axis: Euler angles in degrees

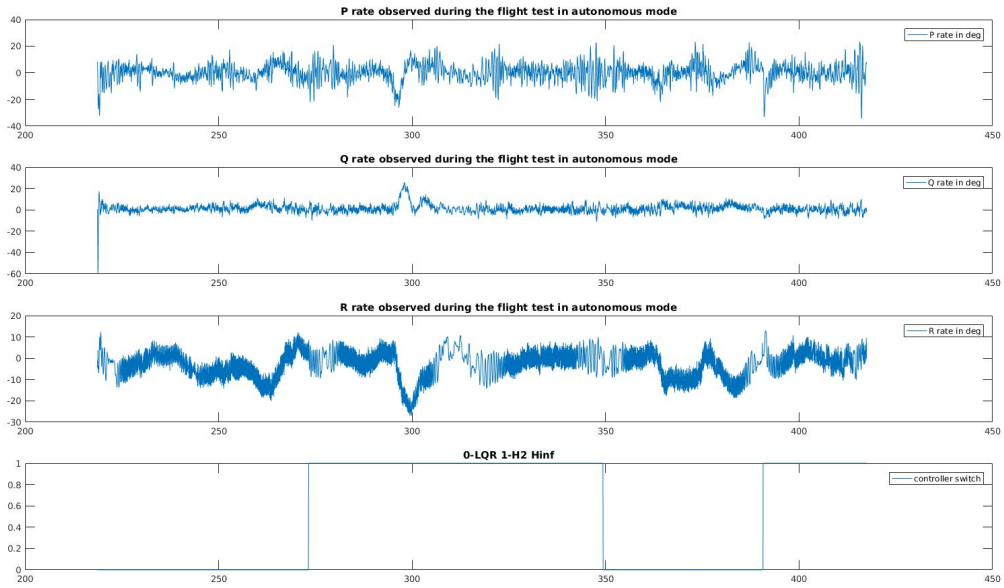

Figure 5.6: Body rates measured from flight test, X-axis: Time in seconds, Y-axis: Body rates in deg/s

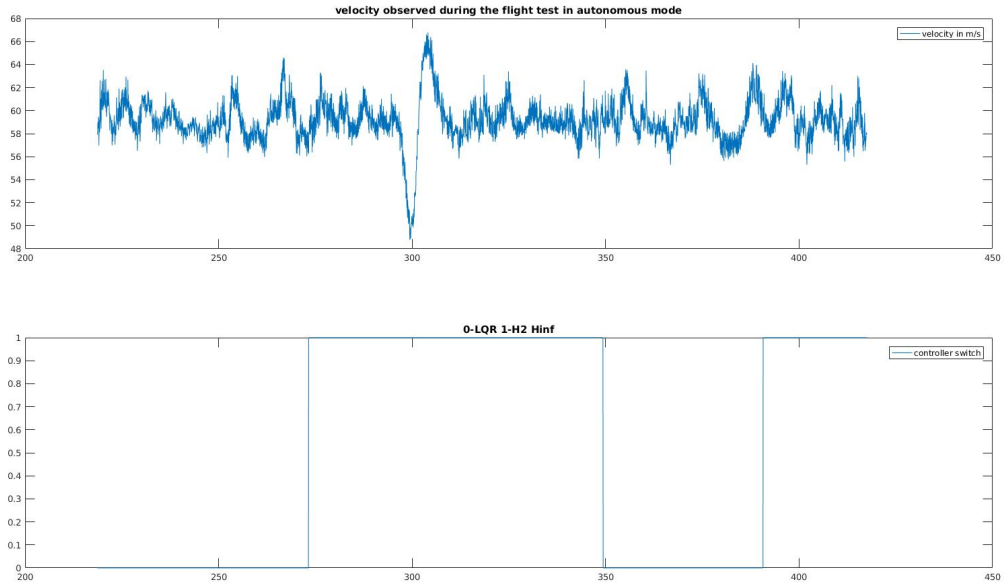

Figure 5.7: Aircraft velocity measured from flight test, X-axis: Time in seconds, Y-axis: velocity in ft/s

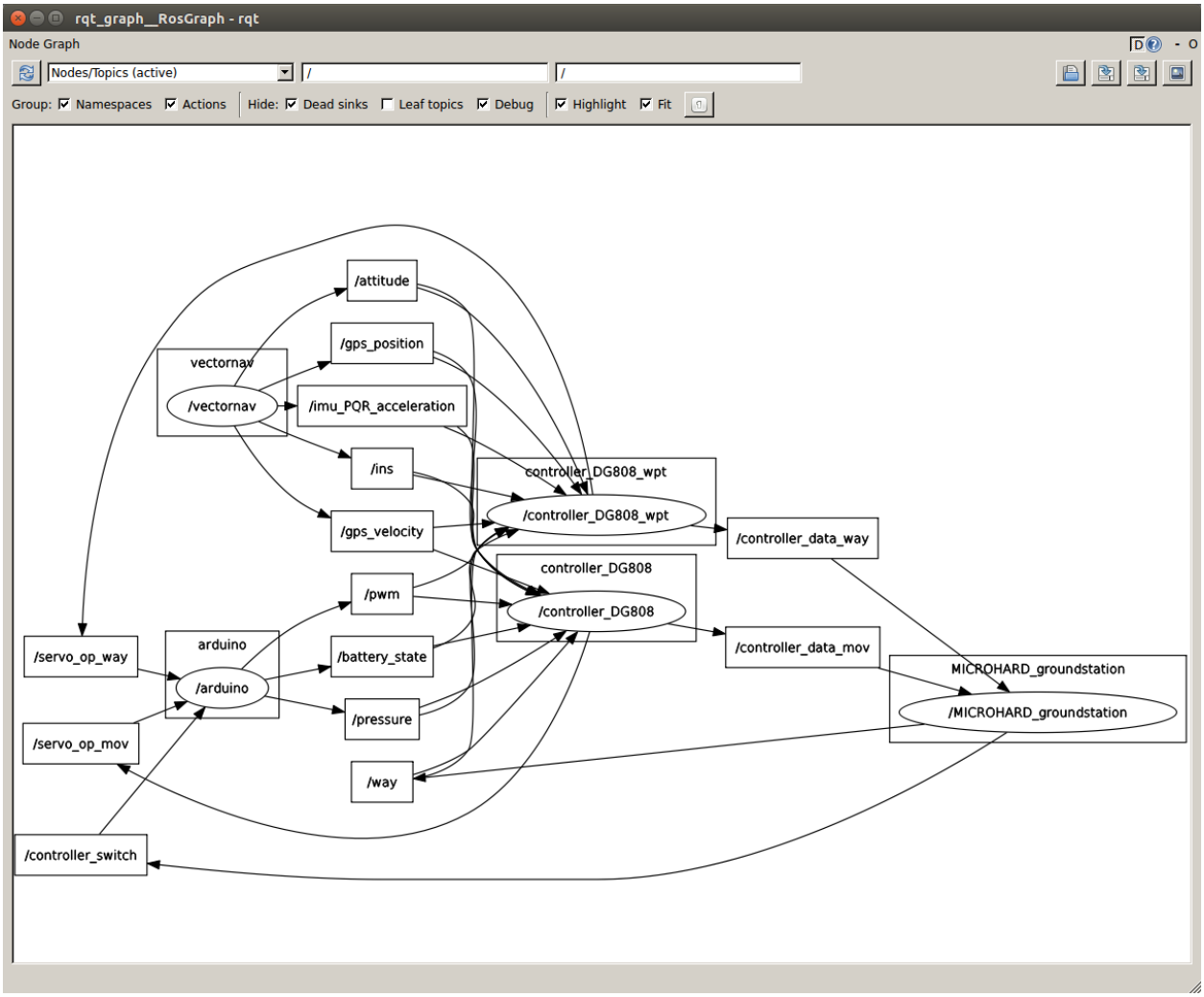

Figure 5.8: Rosgraph plot showing the nodes and topics and for the switch between controllers case

instability in the system behavior, thus the switching between the controllers in the middle of the mission was smooth while following the mission.

However, the above test did not include the decision module for fault detection because the aim of the test was to see the stability of the aircraft, after switching the controller. In this test, the switch between the controllers was initiated by the ground station operator. Figure 5.8 shows the ROS software architecture of the test. As a future work, we plan to integrate the decision logic and do a flight test with fault injection.

Chapter 6

COTS Autopilot with UAV Simplex Architecture

In UAV simplex architecture, two different hardware platforms were used. They were classified as High Performance platform and High Assurance platform. So far, we have used an ARM Cortex-M3 based custom built computing board as the HA platform, which runs the decision logic and the safety controller.

The realization of UAV simplex architecture was hindered by this custom built board since it takes more time and cost to develop and verify. Hence in this chapter, we replaced the custom built board with an available COTS autopilot that also provides sensor acquisition and a simple mission controller. We used Pixhawk [5] as our COTS autopilot along with our onboard computer.

Pixhawk, as shown in Figure 6.1, is a famous COTS autpilot that also provides easy interface to connect with an on board computer and runs a simple PID controller that follows the uploaded mission from ground station. Pixhawk uses a cortex-M4F processor that operates at 168 MHz. It also provides 14 PWM pins, that are required for the servo motors and supports peripherals like I2C and CAN, that are essential for communicating with the sensor. However, the main advantage of using Pixhawk over our custom built avionics is the that, the Pixhawk software framework provides integration support to the sensors required for an UAV. Whereas, in our custom built avionics, we

Figure 6.1: Pixhawk outer view

need to take care of sensor integration.

6.1 Implementation

The following steps are involved in replacing the custom built avionics board with the Pixhawk COTS board.

The pixhawk board comes with two telemetry ports. These ports output all the sensor data encrypted using mavlink protocol. The telemetry port 1 will be always used for communication with ground station. Hence the communication between pixhawk and the onboard computer is established using a usb2ttl converter between telemetry port 2 and the onboard computer.

The mavlink encrypted data is available to the onboard computer. These data should be decrypted before sending them to the complex controller algorithm. The ROS provides a mavlink encryption and decryption package called MAVROS. MAVROS decrypts these data and publishes sensor data. To support our complex controller, an extra ROS node is implemented that converts the sensor data from MAVROS topic to the topics supported by the complex controller node.

The Figure 6.2 shows the rosgraph containing all the nodes and topics in this setup. Once the inputs are given to the on board controller algorithm, the output will be generated. The pixhawk also generates its output value. The values generated by the pixhawk should be replaced with the

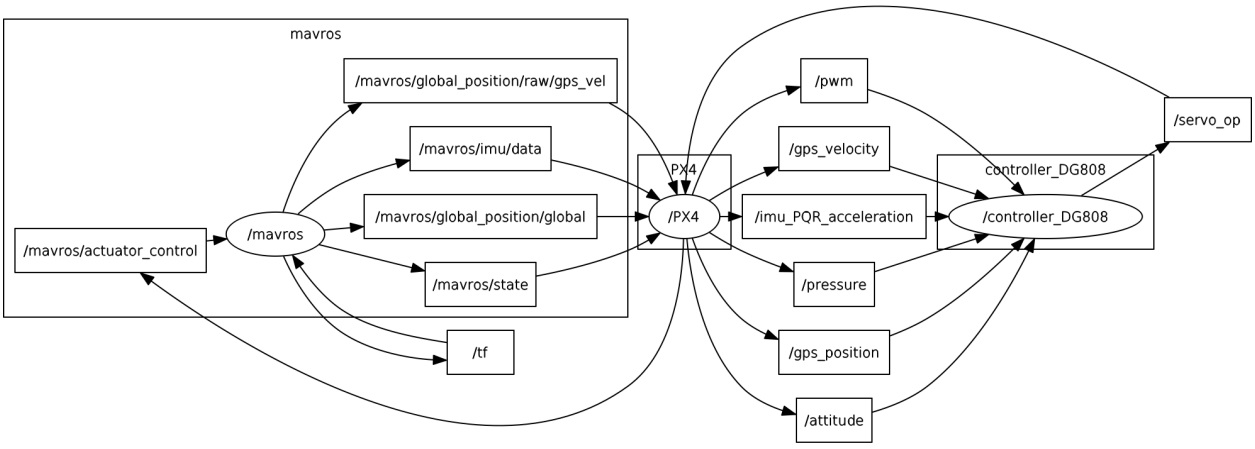

Figure 6.2: ROS nodes outline for PX4 and the controller program

values from the on board computer. For this, Pixhawk supports the offboard mode that rewrites its output with the on board computer's output.

6.2 HiL Test

The UAV simplex architecture using the Pixhawk is in Hardware In the Loop simulation. The setup consists of the following applications.

1. Xplane Flight gear simulator: Xplane [9] simulator runs the model of the aircraft and it takes the controller output as its input and generates the states. These states are sent to the controller program.

2. QGround station: The states generated by the Xplane are delivered to the controller program running in the pixhawk through Qground station. The Qground station is connected with the Pixhawk and communicates with the Xplane using UDP communication.

The Qground station receives all the state information from Xplane. Then, Qground station encrypts the values using mavlink protocol. The mavlink encrypted values are given to the on board computer. The on board computer generates the control output based on the state information and sent them to Xplane through Qground station.

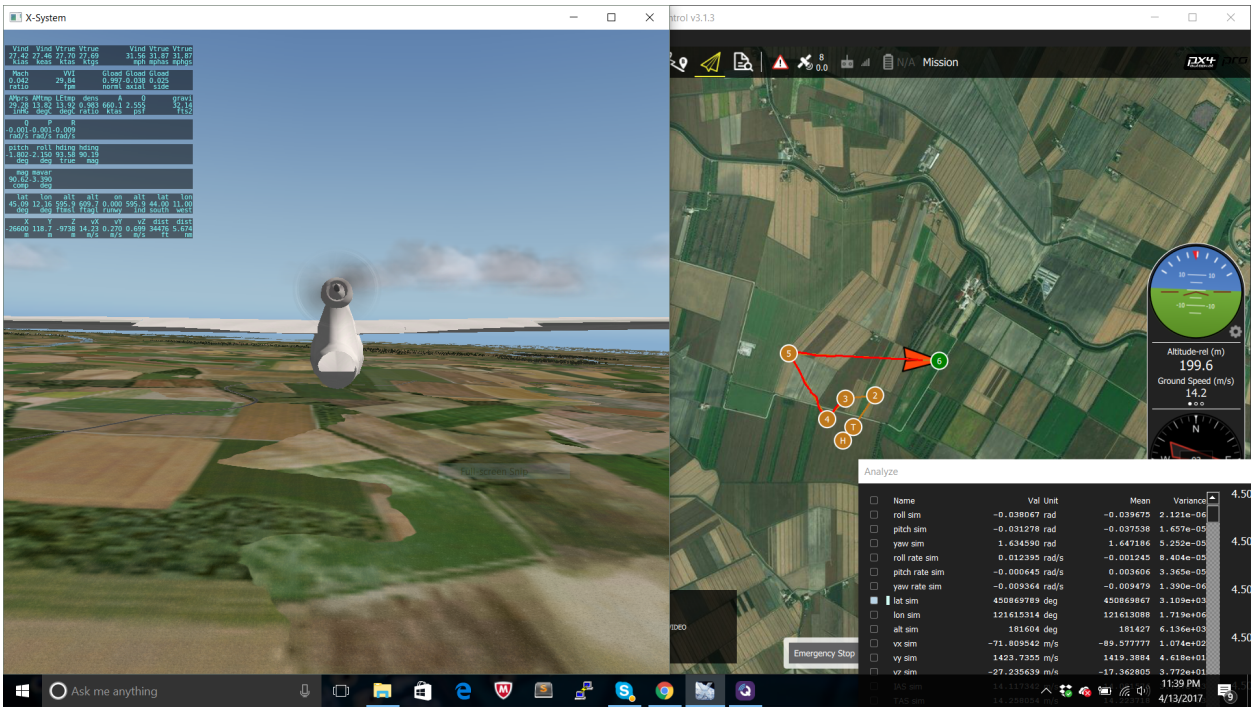

Figure 6.3: Hardware in the Loop for PX4 as the safety platform

In this work, we have not added the decision logic into the Pixhawk. The Pixhawk will automatically switch its mode from "offboard" to "return to land" when it stops receiving the values from on board computer. This feature provides safety against the fail stop case. But it cannot detect and recover from the unsafe output conditions. Adding the decision logic into the Pixhawk is our future work.

6.3 Aircraft Setup using Pixhawk and On Board Computer

We have installed the avionics containing Pixhawk (as a HA platform) and an Odroid-XU4 board (as a HP platform). As shown in the Figure 6.4, this is the avionics setup installed in the aircraft named Skyhunter.

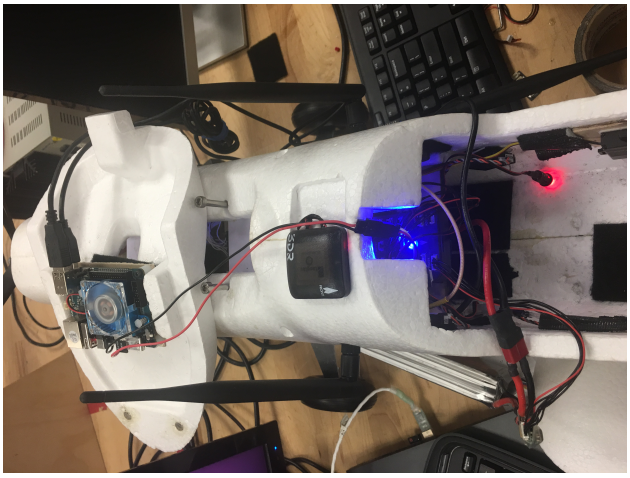

Figure 6.4: Skyhunter using Pixhawk and odroid avionics

Chapter 7

Conclusion

We have presented a fault-tolerant UAV design based on the Simplex [35] architecture. The proposed design uses two heterogeneous platforms with distinct reliability and performance characteristics.

Our main idea is that we use a reliable but less performance hardware platform, we call a High-Assurance (HA) platform, to be responsible for safety while we use a more performance, but potentially less reliable platform, which we call a High- Performance (HP) platform, for performance and intelligence of the UAV. During the normal operation, the HP platform is responsible for controlling the UAV. However, if it fails due to transient hardware faults or software bugs, the HA platform will immediately take over until the HP platform recovers. As such, our design provides a much needed fail-operational property in the UAV.

We have implemented the proposed design on an actual UAV using a low-cost Arduino and a high-performance Tegra TK1 platform. We have demonstrated the system's ability to detect and recover from failures through a set of experiments in a hardware-in-the-loop simulation setting. We did a flight test to show that the system is stable after switching between the controllers. We have also showed using hardware in the loop simulation that UAV simplex architecture can be implemented using COTS autopilot. In the future, we plan to implement more sophisticated vision sense-and-avoid capability.

References

[1] Advanced Aircraft Analysis. http://www.darcorp.com/Software/AAA/.

[2] Aircraft stall. http://www.controlchat.com/physics-of-flight-the-stall/.

[3] FAA Increases Altitude Limit for Commercial UAVs. https://www.aiaa.org/Detail.aspx?id=33070/.

[4] Microhard. http://www.microhardcorp.com/n920.php.

[5] Pixhawk COTS Autopilot. https://pixhawk.org/.

[6] Qground station. http://qgroundcontrol.org/.

[7] Simulink. http://www.mathworks.com/products/simulink/.

[8] Vn-220 rugged gps/ins. http://www.vectornav.com/products/vn200-rugged/.

[9] Xpalne flight simulator. http://www.x-plane.com/.

[10] F. A. T. Abad, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo. Reset-based recovery for real-time cyber-physical systems with temporal safety constraints. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, Sept 2016.

[11] F. Abdi, M. Hasan, S. Mohan, D. Agarwal, and M. Caccamo. Resecure: a restart-based security protocol for tightly actuated hard real-time systems. In *1st Workshop on Security and Dependability of Critical Embedded Real-Time Systems (2016)*.

[12] F. Abdi, R. Tabish, M. Rungger, M. Zamani, and M. Caccamo. Application and system-level software fault tolerance through full system restarts. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ICCPS '17, pages 197–206, New York, NY, USA, 2017. ACM.

[13] Aeronautical Radio Inc. *Avionics Application Standard Software Interface (ARINC) 653*, 2013.

[14] S. Bak, D. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. The system-level simplex architecture for improved real-time embedded system safety. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 99–107. IEEE, 2009.

[15] R. Baumann. Soft errors in advanced computer systems. *Design & Test of Computers, IEEE*, 22(3):258–266, 2005.

[16] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. *UPPAAL: a tool suite for automatic verification of real-time systems*. Springer, 1996.

[17] P. Bieber, F. Boniol, M. Boyer, E. Noulard, and C. Pagetti. New Challenges for Future Avionic Architectures. *AerospaceLab Journal*, (4), 2012.

[18] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, 2005.

[19] D. Brière and P. Traverse. AIRBUS A320/A330/A340 electrical flight controls-A family of fault-tolerant systems. In *Fault-Tolerant Computing*, pages 616–623. IEEE, 1993.

[20] J. B. Dabney and T. L. Harman. *Mastering simulink*. Pearson/Prentice Hall, 2004.

[21] P. Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. John Wiley & Sons, 2010.

[22] G. A. Garcia, S. Keshmiri, and R. Colgren. Advanced h-infinity trainer autopilot. *AIAA Modeling and Simulation Technologies Conference*, August 2010.

[23] M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann. dosek: the design and implementation of a dependability-oriented static embedded kernel. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 259–270. IEEE, 2015.

[24] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.

[25] O. Kotaba, J. Nowotsch, M. Paulitsch, S. Petters, and H. Theilingx. Multicore in real-time systems temporal isolation challenges due to shared resources. In *Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems*, 2013.

[26] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[27] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, Jan 1979.

[28] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate. Testing or formal verification: Do-178c alternatives and industrial experience. *Software, IEEE*, 30(3):50–57, 2013.

[29] R. Murray. Lecture on linear temporal logic. `http://www.cds.caltech.edu/~murray/courses/afrl-sp12/L3_ltl-24Apr12.pdf`, 2012.

[30] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Dependable Computing Conference (EDCC)*, pages 132–143. IEEE, 2012.

[31] NVIDIA. *NVIDIA Tegra K1 Mobile Processor, Technical Reference Manual Rev-01p*, 2014.

[32] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[33] RTCA. *DO-178C Software Considerations in Airborne Systems and Equipment Certification*, 2011.

[34] I. Sadeghzadeh and Y. Zhang. A review on fault-tolerant control for unmanned aerial vehicles (uavs). *Infotech@ Aerospace*, 2011.

[35] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

[36] J. Song and G. Parmer. C'mon: a predictable monitoring infrastructure for system-level latent fault detection and recovery. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 247–258. IEEE, 2015.

[37] P. Vivekanandan. Source code for our avionics. `https://github.com/tharunprasanth/drone_ros`.

[38] M. K. Yoon, S. Mohan, J. Choi, J. E. Kim, and L. Sha. Securecore: A multicore-based intrusion detection architecture for real-time embedded systems. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 21–32, April 2013.

[39] J. F. Ziegler. Terrestrial cosmic rays. *IBM journal of research and development*, 40(1):19–39, 1996.

Appendix A

Videos

We have made few videos showing our HiL test

1) The Hardware in the Loop simulation using our custom built avionics, showing fault tolerance against the crash fault video can be found in this link `https://youtu.be/p6BT3UN8qJE`

2) The video playback of the ground station during the real flight test with switch between controllers can be found in this link `https://www.dropbox.com/s/qrctgrqe5k1dz2i/CTRL_Switch_LQR2H2-2017-03-26_18-09-41.avi?dl=0`

Appendix B

Issues Faced and Solved

This appendix includes information about some of the issues that we faced and solved while integrating and using our avionics, matlab code generation and in bench test.

B.1 Divide by Zero

We are using kalman filters in our controller program for prediction and estimation of the system states. Divide by zero occured at the estimation of 'VT', the true velocity which is calculated using the pressure value from the 'pitot tube'. Pitot tube is connected to a differential pressure sensor. When the UAV system is static in ground, it reads zero pressure that results in zero velocity value. This zero velocity is given to the kalman filter block as seen in Figure B.1. This block outputs 'NaN' and will cause the controller to diverge. We fixed this issue by giving a small constant value to the velocity, when it reads zero in ground.

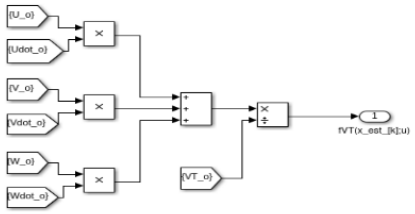

Figure B.1: Estimation of True velocity

```
function [Height,North,East]=lla_to_fe(inertial_local_data,alt,lat,lon)

% Convert the given LLA coordinates to Flat Earth coordinates.
%
% ref_alt - meters ASML
% ref_lat - deg.
% ref_lon - deg.
%
% Height, North, East are in feet

Height = (alt - inertial_local_data.ref_alt)*(1/0.3048);
North  = (lat - inertial_local_data.ref_lat)*(inertial_local_data.fact_lat);
East   = (lon - inertial_local_data.ref_lon)*(inertial_local_data.fact_lon);
```

Figure B.2: Matlab code to convert GPS position from global to local

B.2 Structure Ordering

The autonomous controller in initial stages, is designed using Matlab. For the guidance block(guides in following the mission) in autonomous controller, the GPS position in local frame is required. Usually, all GPS sensors output GPS values in global frame. Hence we need to convert them from global to local frame. The Figure B.2 shows the Matlab code to convert the GPS position from global to local frame. The conversion consist of current GPS position in global frame, reference position in global frame and conversion factor based on the reference position. We use Matlab Embedded code generator to generate C code from the autonomous controller Matlab files. The Figure B.3 shows the generated C code. In generated code, the values of the conversion factor based on the reference position and the reference position in global frame are interchanged. This interchanged values lead to wrong guidance values. Hence the aircraft in hardware in the loop testing did not follow the mission. The reason for this value interchange in the generated code was, these variables are created as structures. The order in which the elements inside this structure was different from the order in which the elements inside the structure was defined. This declaration and definition mismatch showed no impact on Matlab simulation, but in code generation, this mismatch was reflected. This issue was resolved by following uniform structure order during its declaration and definition.

B.3 USB Hub Power Issue

In our avionics, we used usb connection to connect with our custom built data acquisition board, with GPS and IMU sensor and with the telemetry module to communicate with the ground station.

```
void EKF_IFS_1_ConvertLatLonAlttoFlatEarth(real_T rtu_alt, real_T rtu_lat,
 real_T rtu_lon, rtB_ConvertLatLonAlttoFlatEarth_EKF_IFS_1_T *localB)

 /* MATLAB Function 'Intelligent Flight System (subsystem)/GPS to Local/Conv
 /*  Convert the given LLA coordinates to Flat Earth coordinates. */
 /*  */
 /*  ref_alt - meters ASML */
 /*  ref_lat - deg. */
 /*  ref_lon - deg. */
 /*  */
 /*  Height, North, East are in feet */
 /* '<S23>:1:11' */
 localB->Height = (rtu_alt - 255.4224) * 3.280839895013123;

 /* '<S23>:1:12' */
 localB->North = (rtu_lat - 283665.7711) * 38.915422;

 /* '<S23>:1:13' */
 localB->East = (rtu_lon - 364574.1647) * -95.317747;


*
```

Figure B.3: Matlab generated C code to convert GPS position from global to local

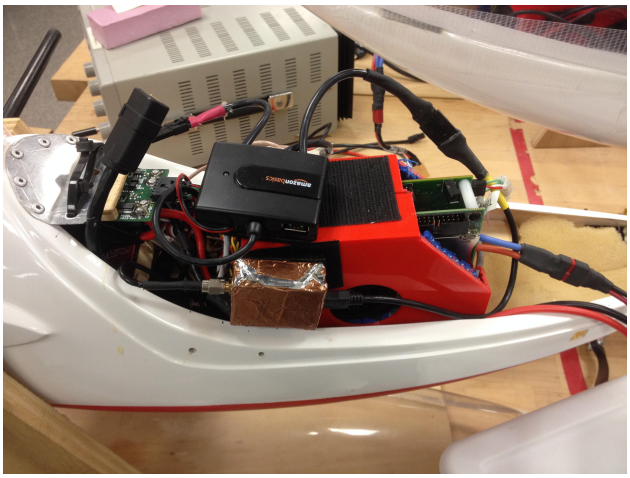

Figure B.4: Picture of our avionics containing a usb hub

In order to support all the usb connections, we use an non powered usb hub as shown in Figure B.4. The Vector Nav VN-200 sensor [8](GPS and IMU) draws current from the usb hub to operate. Since we were using a non powered hub, the current drawn by the hub from the processing board was not sufficient, the VN-200 sensor values had a huge variation for successive reading. This huge variation caused our autonomous controller to diverge (NaN values). this issue was resolved by using a powered usb hub. Though the powered usb hub resolved the VN-200 sensor data issue, it produced another issue during flight tests. The aircraft that we used had small area to accommodate the avionics. Hence the avionics and the usb hub were placed very close to each other. The avionics and the usb hub inside the aircraft are covered using the aircraft's canopy. Somehow, this canopy put more pressure on the power cables to the usb hub and the power line to hub was broken. Since the hub was not powered the controller program did not receive any values from the sensors, that

resulted in landing the aircraft. This issue was fixed by soldering the power cables.

We have encountered other hardware related issues, like battery used in the aircraft did not provide enough power to motor. This added difficulty in aircraft take off. Another issue that we faced is, the motor mount got broken while the aircraft was flying. The pitot tube, required for measuring the air speed, was not reading proper values. This happened because the tube was covered with frozen ice during the winter.

Appendix C

Avionics Installation

We have included the instructions to install our avionics in this appendix. In this thesis, we used Tegra K1. We have included the bash commands detailed instructions on setting up the Tegra K1 processor.

C.1 Software Dependencies

1. OS - Ubuntu
2. Middleware - ROS

C.2 Installing our Avionics Code

All our code are maintained in our github repository [37]. Clone our repository inside the ROS workspace. After building the code, the avionics is ready to install into the aircraft. The description about each folder in our repository is as follows:

1. aircraft: This folder contains all the Matlab and generated C codes of our test aircrafts.
2. autopilot_releases: This folder contains all the Matlab files that are required for the simulation tests.

3. launch: This folder contains all the ROS launch files that are created for each aircarft. This launch file calls all the required programs like sensor data reading, controller and ground station communication.

4. mavlink: We use mavlink protocol to communicate with the ground station. his folder contains library files for supporting mavlink.

5. msg: ROS using topics to publish and subscribe from other nodes. This folder contains the structure for all custom messages.

6. src: This folder contains all the source files. This includes the main controller program, reading sensor data, writing to servo and ground station communication.

Appendix D

Source Code Listing

This appendix lists the source code for important programs we developed for the avionics.

D.1 Controller Function Code

This is the most important code. It subscribes all the sensor data and assigns it to Matlab generated variables. It calls the controller function and then it publishes the controller output.

```
1  /**
2   * @file     controller_DG808.cpp
3   * @author   Prasanth Vivekanandan
4   * @date     8/18/2016
5  **/
6
7
8  #define DEBUG 1
9  #include <ros/ros.h>
10 #include <stddef.h>
11 #include <stdio.h>
12 #include <time.h>
13 #include <unistd.h>
14 #include <inttypes.h>
15 #include <unistd.h>
16 #include <getopt.h>
17 #include <stdlib.h>
18 #include <math.h>
19
20 #include "DG808_GNC_mov.c"
21 #include "rtGetNaN.c"
22 #include "rt_nonfinite.c"
23 #include "rtGetInf.c"
24 #include "rtwtypes.h"
25 #include "helpers.h"
```

```
26
27 #include <drone_ros/ins.h>
28 #include <drone_ros/pwm.h>
29 #include <drone_ros/data.h>
30 #include <drone_ros/way.h>
31 #include <drone_ros/hil_data.h>
32
33 //messgae library for GPS position
34 #include "sensor_msgs/NavSatFix.h"
35 //message library for GPS velocity
36 #include "geometry_msgs/Twist.h"
37 //message library for PQR and acceleration
38 #include "sensor_msgs/Imu.h"
39 //message library for Phi,Theta and Psi
40 #include "geometry_msgs/Point.h"
41 #include "sensor_msgs/Temperature.h"
42 #include "sensor_msgs/FluidPressure.h"
43 #include "sensor_msgs/BatteryState.h"
44 /*************************************************************************
45  * Public Definitions
46  ************************************************************************/
47 #define PDEBUG(fmt, ...)                                                \
48         do { if (DEBUG) fprintf(stderr, fmt, __VA_ARGS__); } while (0)
49
50
51 /*************************************************************************
52  * Public Types
53  ************************************************************************/
54 #define SAVEFILE 1
55 #define DEG2RAD 0.0174532925
56 #define GRAVITY 9.8
57 #define DENSITY 0.002295
58 /*************************************************************************
59  * Global Variables
60  ************************************************************************/
61 ros::Publisher g_pub_data_arduino;
62 ros::Publisher g_pub_data_gs;
63
64
65 ros::Publisher g_pub_data_hil; //used to publish data to the Hardware In the Loop node
66 void call_controller();
67
68
69 std::string g_controller_frame_id;
70
71 struct usb_status_struct usb_status; // Stores the health condition of USB devices ; Used by the decision logic block in the
       auto generated code
72 struct pwm_conversion_struct pwm_conversion_factors; //Structure to store the conversion factors for each surfaces
73 struct helpers_struct helpers; // To be used in ROS subscription in log node and controller node
74
75 ExtU_DG808_GNC_mov_T *controller_in  = &DG808_GNC_mov_U;
76 ExtY_DG808_GNC_mov_T *controller_out = &DG808_GNC_mov_Y;
77 /*************************************************************************
78  * Public Function Prototypes
79  ************************************************************************/
80 void send_pwm();
```

```
81
82  uint64_t get_elapsed(struct timespec *start, struct timespec *end)
83  {
84          uint64_t dur = (end->tv_sec * 1000000000 + end->tv_nsec) -
85              (start->tv_sec * 1000000000 + start->tv_nsec);
86          return dur;
87  }
88
89  double get_time(struct timespec *calc,struct timespec *calc1)
90  {
91          double dur = ((double)calc1->tv_sec+ ((double)calc1->tv_nsec/1000000000)) -
                ((double)calc->tv_sec+((double)calc->tv_nsec/1000000000));
92
93          return dur;
94  }
95
96  void pwm_callback(const drone_ros::pwm::ConstPtr & msg)
97  {
98
99          controller_in->RC.throttle_cmd = msg->Throttle;
100         controller_in->RC.elevator_cmd = msg->Elevator;
101         controller_in->RC.aileron_cmd = msg->Aileron;
102         controller_in->RC.rudder_cmd = msg->Rudder;
103         controller_in->VTalphabetameas.VT = msg->velocity;
104
105         if(msg->PICCIC == 1){
106                 if(helpers.SAS_flag == 2)
107                         controller_in->PICCIC = 0;
108                 else
109                         controller_in->PICCIC = 1;
110         }
111         if(msg->PICCIC == 0){
112                 controller_in->PICCIC = 0;
113                 //trim update
114                 controller_in->throttletrim = controller_in->RC.throttle_cmd;
115                 controller_in->elevatortrim = controller_in->RC.elevator_cmd;
116                 controller_in->ailerontrim = controller_in->RC.aileron_cmd;
117                 controller_in->ruddertrim = controller_in->RC.rudder_cmd;
118
119         }
120         usb_status.usb_arduino = msg->usbstatus;
121 }
122
123
124 void DAQ_pressure_callback(const sensor_msgs::FluidPressure::ConstPtr & msg_pressure)
125 {
126         helpers.pressure = msg_pressure->fluid_pressure;
127 }
128
129 void DAQ_voltage_callback(const sensor_msgs::BatteryState::ConstPtr & msg_battery_state)
130 {
131         helpers.voltage = msg_battery_state->voltage;
132 }
133
134
135 void way_callback(const drone_ros::way::ConstPtr & msg_waypoints)
```

```
136 {
137         ROS_INFO_STREAM("finished␣reading␣waypoints");
138         controller_in->WaypointsIN.rp = 1;
139         for(int w=0;w<32;w++)
140         {
141          controller_in->WaypointsIN.n[w] = msg_waypoints->north[w];
142          controller_in->WaypointsIN.e[w] = msg_waypoints->east[w];
143          controller_in->WaypointsIN.h[w] = msg_waypoints->height[w];
144          controller_in->WaypointsIN.v[w] = msg_waypoints->velocity[w];
145          controller_in->WaypointsIN.fp   = msg_waypoints->fp;
146          }
147         printf("the␣way␣point␣are␣%f␣%f␣%f␣%f",controller_in->WaypointsIN.n[0],
148         controller_in->WaypointsIN.e[0],controller_in->WaypointsIN.h[0],controller_in->WaypointsIN.v[0]);
149 }
150
151
152 void gps_callback(const drone_ros::ins::ConstPtr & msg_gps_imu)
153 {
154         ROS_INFO_STREAM("finished␣reading␣gps␣signals");
155         usb_status.usb_vectornav = msg_gps_imu->usbstatus;
156         helpers.gps_time = msg_gps_imu->time;
157         helpers.gps_week = msg_gps_imu->week;
158
159 }
160
161 void gps_position_callback(const sensor_msgs::NavSatFix::ConstPtr & msg_gps_position)
162 {
163         ROS_INFO_STREAM("finished␣reading␣gps_position␣");
164         controller_in->GPSPositionmeas.Latitude = msg_gps_position->latitude;
165         controller_in->GPSPositionmeas.Longitude = msg_gps_position->longitude;
166         controller_in->GPSPositionmeas.Altitude = msg_gps_position->altitude;
167 #if HIL_TEST
168                 call_controller();
169 #endif
170 }
171
172 void gps_velocity_callback(const geometry_msgs::Twist::ConstPtr & msg_gps_velocity)
173 {
174         ROS_INFO_STREAM("finished␣reading␣gps_velocity␣");
175         controller_in->GPSVelocitymeas.V_north = msg_gps_velocity->linear.x;
176         controller_in->GPSVelocitymeas.V_east = msg_gps_velocity->linear.y;
177         controller_in->GPSVelocitymeas.V_down = msg_gps_velocity->linear.z;
178 }
179
180 void imu_PQR_acceleration_callback(const sensor_msgs::Imu::ConstPtr & msg_imu_PQR_acceleration)
181 {
182         ROS_INFO_STREAM("finished␣reading␣imu_PQR_acceleration␣");
183         controller_in->BodyRatesmeas.P = msg_imu_PQR_acceleration->angular_velocity.x;
184         controller_in->BodyRatesmeas.Q = msg_imu_PQR_acceleration->angular_velocity.y;
185         controller_in->BodyRatesmeas.R = msg_imu_PQR_acceleration->angular_velocity.z;
186         controller_in->Accelerometermeas.Axb = msg_imu_PQR_acceleration->linear_acceleration.x ;
187         controller_in->Accelerometermeas.Ayb = msg_imu_PQR_acceleration->linear_acceleration.y ;
188         controller_in->Accelerometermeas.Azb = msg_imu_PQR_acceleration->linear_acceleration.z ;
189 }
190
191 void imu_attitude_callback(const geometry_msgs::Point::ConstPtr & msg_imu_attitude)
```

```
192 {
193         ROS_INFO_STREAM("finished reading imu_attitude ");
194         controller_in->EulerAnglesmeas.phi = msg_imu_attitude->x;
195         controller_in->EulerAnglesmeas.theta = msg_imu_attitude->y;
196         controller_in->EulerAnglesmeas.psi = msg_imu_attitude->z;
197 }
198
199 void send_hil_output()
200 {
201         drone_ros::hil_data msg_to_hil;
202         msg_to_hil.controller_throttle_output = controller_out->ControlSurfaceCommands.throttle_cmd;
203         msg_to_hil.controller_elevator_output = controller_out->ControlSurfaceCommands.elevator_cmd;
204         msg_to_hil.controller_aileron_output = controller_out->ControlSurfaceCommands.aileron_cmd;
205         msg_to_hil.controller_rudder_output = controller_out->ControlSurfaceCommands.rudder_cmd;
206         msg_to_hil.dist_2_waypoint = controller_out->DistancetoWaypoint;
207         msg_to_hil.waypoint_index = controller_out->WaypointIndex;
208         msg_to_hil.guidance_VT = controller_out->CommandedStates.VT_cmd;
209         msg_to_hil.guidance_phi  = controller_out->CommandedStates.phi_cmd;
210         msg_to_hil.guidance_theta = controller_out->CommandedStates.theta_cmd;
211         msg_to_hil.eta_Lat = controller_out->TrackAngles.etaLat;
212         msg_to_hil.eta_Lon = controller_out->TrackAngles.etaLon;
213         msg_to_hil.n_op = controller_out->n_mov_pt[0];
214         msg_to_hil.e_op = controller_out->e_mov_pt[0];
215         msg_to_hil.h_op = controller_out->h_mov_pt[0];
216         g_pub_data_hil.publish(msg_to_hil);
217 }
218 /**
219  * update a message to be sent to the micro hd ROS node
220  */
221 void set_ground_data(drone_ros::data * msg_data)
222 {
223         msg_data->header.frame_id = g_controller_frame_id;
224         msg_data->angle.x = controller_in->EulerAnglesmeas.phi;
225         msg_data->angle.y = controller_in->EulerAnglesmeas.theta;
226         msg_data->angle.z = controller_in->EulerAnglesmeas.psi;
227         msg_data->lla.x = controller_in->GPSPositionmeas.Latitude;
228         msg_data->lla.y = controller_in->GPSPositionmeas.Longitude;
229         msg_data->lla.z = controller_in->GPSPositionmeas.Altitude;
230         msg_data->velocity.x = controller_in->GPSVelocitymeas.V_north;
231         msg_data->velocity.y = controller_in->GPSVelocitymeas.V_east;
232         msg_data->velocity.z = controller_in->GPSVelocitymeas.V_down;
233         msg_data->angrate.x = controller_in->BodyRatesmeas.P;
234         msg_data->angrate.y = controller_in->BodyRatesmeas.Q;
235         msg_data->angrate.z = controller_in->BodyRatesmeas.R;
236         msg_data->PICCIC = controller_in->PICCIC;
237         msg_data->HomeCmd = controller_in->HomeCmd;
238         msg_data->Throttle = controller_in->RC.throttle_cmd;
239         msg_data->Elevator = controller_in->RC.elevator_cmd/DEG2RAD;
240         msg_data->Rudder = controller_in->RC.rudder_cmd/DEG2RAD;
241         msg_data->Aileron = controller_in->RC.aileron_cmd/DEG2RAD;
242         msg_data->acceleration.x = controller_in->Accelerometermeas.Axb;
243         msg_data->acceleration.y = controller_in->Accelerometermeas.Ayb;
244         msg_data->acceleration.z = controller_in->Accelerometermeas.Azb;
245         msg_data->pressure = helpers.pressure;
246         msg_data->voltage = helpers.voltage;
247         msg_data->air_velocity = controller_in->VTalphabetameas.VT;
```

```
248         msg_data->alphaest = controller_out->vtalphabeta_est.alpha;
249         msg_data->betaest = controller_out->vtalphabeta_est.beta;
250
251         msg_data->time = helpers.gps_time;
252         msg_data->week = helpers.gps_week;
253         //for(int y=0;y<4;y++)
254         msg_data->autothrottle =
255             controller_out->ControlSurfaceCommands.throttle_cmd;
256         msg_data->autoelevator =
257             controller_out->ControlSurfaceCommands.elevator_cmd /
258             DEG2RAD;
259         msg_data->autoaileron =
260             controller_out->ControlSurfaceCommands.aileron_cmd /
261             DEG2RAD;
262         msg_data->autorudder =
263             controller_out->ControlSurfaceCommands.rudder_cmd / DEG2RAD;
264         msg_data->d2b = controller_out->DistancetoWaypoint;
265         msg_data->a_usbstatus = usb_status.usb_arduino;
266         msg_data->v_usbstatus = usb_status.usb_vectornav;
267         msg_data->guidance_VT = controller_out->CommandedStates.VT_cmd;
268
269         msg_data->guidance_theta = controller_out->CommandedStates.theta_cmd;
270         msg_data->guidance_phi = controller_out->CommandedStates.phi_cmd;
271         msg_data->guidance_beta = controller_out->CommandedStates.beta_cmd;
272         msg_data->trackangles_etaLat = controller_out->TrackAngles.etaLat;
273         msg_data->trackangles_etaLon = controller_out->TrackAngles.etaLon;
274         msg_data->auto_switch = controller_in->PICCIC;
275         msg_data->angles_est[0] = controller_out->angles_est.phi;
276         msg_data->angles_est[1] = controller_out->angles_est.theta;
277         msg_data->angles_est[2] = controller_out->angles_est.psi;
278
279         msg_data->brates_est[0] = controller_out->brates_est.P;
280         msg_data->brates_est[1] = controller_out->brates_est.Q;
281         msg_data->brates_est[2] = controller_out->brates_est.R;
282
283         msg_data->accel_est[0] =  controller_out->accelerometer_est.Axb;
284         msg_data->accel_est[1] =  controller_out->accelerometer_est.Ayb;
285         msg_data->accel_est[2] =  controller_out->accelerometer_est.Azb;
286
287         msg_data->pos_est[0] =  controller_out->gpspos_est.Latitude;
288         msg_data->pos_est[1] =  controller_out->gpspos_est.Longitude;
289         msg_data->pos_est[2] =  controller_out->gpspos_est.Altitude;
290
291         msg_data->vel_est[0] =  controller_out->gpsvel_est.V_north;
292         msg_data->vel_est[1] =  controller_out->gpsvel_est.V_east;
293         msg_data->vel_est[2] =  controller_out->gpsvel_est.V_down;
294
295         msg_data->wind_est[0] =  controller_out->inertialwind_est.VwN;
296         msg_data->wind_est[1] =  controller_out->inertialwind_est.VwE;
297         msg_data->wind_est[2] =  controller_out->inertialwind_est.VwH;
298
299         msg_data->throttletrim = controller_in->throttletrim;
300         msg_data->elevatortrim = controller_in->elevatortrim;
301         msg_data->ailerontrim  = controller_in->ailerontrim;
302         msg_data->ruddertrim = controller_in->ruddertrim;
303
```

```
304         msg_data->SAS_switch = controller_in->SASswitch;
305         msg_data->delta_psi = controller_out->delta_psi;
306
307         for( int i=0 ; i<6;i++){
308                 msg_data->mw_height[i] = controller_out->Modified_wpt.h[i];
309                 msg_data->mw_north[i] = controller_out->Modified_wpt.n[i];
310                 msg_data->mw_east[i] = controller_out->Modified_wpt.e[i];
311                 msg_data->mw_velocity[i] = controller_out->Modified_wpt.v[i];
312         }
313         msg_data->n_op = controller_out->n_mov_pt[0];
314         msg_data->e_op = controller_out->e_mov_pt[0];
315         msg_data->h_op = controller_out->h_mov_pt[0];
316         msg_data->waypoint_index = controller_out->WaypointIndex;
317 }
318
319 //call the controllers for HiL testing
320 void call_controller()
321 {
322 #if HIL_TEST
323         DG808_GNC_mov_step();
324         send_pwm();
325         send_hil_output();
326         drone_ros::data msg_data;
327         set_ground_data(&msg_data);
328         g_pub_data_gs.publish(msg_data);
329 #endif
330 }
331
332  void assign_servo_commands()
333 {
334         controller_in->ServoCommands.throttle_cmd = controller_out->ControlSurfaceCommands.throttle_cmd;
335         controller_in->ServoCommands.elevator_cmd = controller_out->ControlSurfaceCommands.elevator_cmd;
336         controller_in->ServoCommands.aileron_cmd  = controller_out->ControlSurfaceCommands.aileron_cmd;
337         controller_in->ServoCommands.rudder_cmd   = controller_out->ControlSurfaceCommands.rudder_cmd;
338 }
339
340 void assign_servo_dfeflections()
341 {
342         controller_in->Servodeflection[0] = controller_out->ControlSurfaceCommands.throttle_cmd;
343         controller_in->Servodeflection[1] = controller_out->ControlSurfaceCommands.elevator_cmd;
344         controller_in->Servodeflection[2] = controller_out->ControlSurfaceCommands.aileron_cmd;
345         controller_in->Servodeflection[3] = controller_out->ControlSurfaceCommands.rudder_cmd;
346 }
347
348
349
350 void send_pwm()
351 {
352         drone_ros::pwm msg_servo;
353         msg_servo.Throttle = controller_out->ControlSurfaceCommands.throttle_cmd;
354         msg_servo.Elevator = controller_out->ControlSurfaceCommands.elevator_cmd;
355         msg_servo.Aileron = controller_out->ControlSurfaceCommands.aileron_cmd;
356         msg_servo.Rudder = controller_out->ControlSurfaceCommands.rudder_cmd;
357         g_pub_data_arduino.publish(msg_servo);
358 }
359
```

```
360 void selftest(float throttle, float elevator, float aileron, float rudder)
361 {
362         controller_out->ControlSurfaceCommands.throttle_cmd = throttle;
363         controller_out->ControlSurfaceCommands.elevator_cmd = elevator;
364         controller_out->ControlSurfaceCommands.aileron_cmd  = aileron;
365         controller_out->ControlSurfaceCommands.rudder_cmd   = rudder;
366         send_pwm();
367 }
368
369 int auto_test(int timecheck)
370 {
371         PDEBUG("\n The time in autopilot routine is %d",timecheck);
372         float up_limit = 10 * DEG2RAD;
373         float down_limit = -10 * DEG2RAD;
374         if ((timecheck >= 0) && (timecheck < 1))//timer > 0 and  < 1 sec
375         {
376                 selftest(0, 0, 0, 0);
377         }
378         if ((timecheck > 1) && (timecheck < 4)) //timer > 1 and < 4 sec
379                 selftest(0, 0, up_limit, 0);    //positive aileron
380         if ((timecheck > 4) && (timecheck < 5)) //timer > 4 and < 5 sec
381                 selftest(0, 0, 0, 0);
382         if ((timecheck > 5) && (timecheck < 8)) //timer > 5 and < 8 sec
383                 selftest(0, 0, down_limit, 0); //negative aileron
384         if ((timecheck > 8) && (timecheck < 9)) //timer > 8 and < 9 sec
385                 selftest(0, 0, 0, 0);
386         if ((timecheck > 9) && (timecheck < 12))        //timer > 9 and < 12 sec
387                 selftest(0, up_limit, 0, 0);    //positive elevator
388         if ((timecheck > 12) && (timecheck < 13))       //timer > 12 and < 13 sec
389                 selftest(0, 0, 0, 0);
390         if ((timecheck > 13) && (timecheck < 16))       //timer > 13 and < 16 sec
391                 selftest(0, down_limit, 0, 0); //negative elevator
392         if ((timecheck > 16) && (timecheck < 17))       //timer > 16 and < 17 sec
393                 selftest(0, 0, 0, 0);
394         if ((timecheck > 17) && (timecheck < 20))       //timer > 17 and < 20 sec
395                 selftest(0, 0, 0, up_limit);    //positive rudder
396         if ((timecheck > 20) && (timecheck < 21))       //timer > 20 and < 21 sec
397                 selftest(0, 0, 0, 0);
398         if ((timecheck > 21) && (timecheck < 24))       //timer > 21 and < 24 sec
399                 selftest(0, 0, 0, down_limit); //negative rudder
400         if ((timecheck > 24) && (timecheck < 26))       //timer > 24 and < 25 sec
401                 selftest(0.1, 0, 0, 0);
402         if ((timecheck > 26) && (timecheck < 28))       //timer > 25 and < 26 sec
403                 selftest(0.1, 0, 0, 0); // 2% of throttle
404         if ((timecheck > 28) && (timecheck < 30))       //timer > 26 and < 27 sec
405                 selftest(0.2, 0, 0, 0); // 4% of throttle
406         if ((timecheck > 30) && (timecheck < 32))       //timer > 27 and < 28 sec
407                 selftest(0.2, 0, 0, 0); // 6% of throttle
408         if ((timecheck > 32) && (timecheck < 34))       //timer > 28 and < 29 sec
409                 selftest(0.3, 0, 0, 0); // 8% of throttle
410         if ((timecheck > 34) && (timecheck < 36))       //timer > 29 and < 30 sec
411                 selftest(0.3, 0, 0, 0); // 10% of throttle
412         if ((timecheck > 37) && (timecheck < 38))
413                 selftest(0, 0, 0, 0);
414         if (timecheck > 38) {
415                 selftest(0, 0, 0, 0);
```

```
416                     return 0;
417             }
418             return 1;
419     }
420
421
422     int autopilot_testmode()
423     {
424             ros::Rate loop_rate(20);
425             int auto_commit=0;
426             while (ros::ok()) {
427                     // In auto start up routine
428                     int timecheck;
429                     struct timespec tests,teste;
430                     ros::spinOnce();
431                     if (auto_commit == 0)
432                             clock_gettime(CLOCK_REALTIME, &tests);
433                     // In autopilot startup
434                     auto_commit = 1;
435                     clock_gettime(CLOCK_REALTIME, &teste);
436                     timecheck = teste.tv_sec - tests.tv_sec;
437
438                     if (!(auto_test(timecheck))){
439                             // Autpilot startup is completed
440                             PDEBUG("end of startup\n",0);
441                             return 1;
442                     }
443                     if (controller_in->PICCIC  == 0){
444                             selftest(0, 0, 0, 0);
445                             auto_commit = 0;
446                             return 1;
447                     }
448                     loop_rate.sleep();
449             }
450     }
451
452
453
454     void read_parameter()
455     {
456
457             ros::NodeHandle nh;
458
459             /* ------  alpha and beta  ------ */
460             nh.getParam("/controller_param/alpha",controller_in->VTalphabetameas.alpha);
461             nh.getParam("/controller_param/beta",controller_in->VTalphabetameas.beta);
462
463             /* ------  enable  ------ */
464             nh.getParam("/controller_param/enable",(int&)controller_in->EnableDisable);
465
466             /* ------  home command  ------ */
467             nh.getParam("/controller_param/hmcmd",(int&)controller_in->HomeCmd);
468
469             /* ------  controller selection  ------ */
470             nh.getParam("/controller_param/controller_selection",controller_in->Controllerselection);
471
```

```
472         /* ------ gamma flag ------ */
473         nh.getParam("/controller_param/gamma_flag",controller_in->gamma_flag);
474
475         /* ------ LQR selection ------ */
476         nh.getParam("/controller_param/LQRselection",controller_in->LQRselection);
477
478         /* ------ smoothing switch guidance ------ */
479         nh.getParam("/controller_param/smoothing_switching_guidance",controller_in->Smoothingswitchguidance);
480
481         /* ------ smoothing switch servo ------ */
482         nh.getParam("/controller_param/smoothing_switching_servo",controller_in->Smoothingswitchservo);
483
484         /* ------ time_window_servo ------ */
485         nh.getParam("/controller_param/time_window_servo",controller_in->time_window_servo);
486
487         /* ------ time_window_guidance ------ */
488         nh.getParam("/controller_param/time_window_guidance",controller_in->time_window_guidance);
489
490         /* ------ KpLon gain in guidance ------ */
491         nh.getParam("/controller_param/KpLon",controller_in->KpLon);
492
493
494
495         /* ------ KiLon gain in guidance ------ */
496         nh.getParam("/controller_param/KiLon",controller_in->KiLon);
497
498
499         /* ------ L Lat ------ */
500         nh.getParam("/controller_param/L_Lat",controller_in->L_Lat);
501
502         /* ------ L Lon ------ */
503         nh.getParam("/controller_param/L_Lon",controller_in->L_Lon);
504
505         /* ------ KaLon gain in guidance ------ */
506         nh.getParam("/controller_param/Ka_Lon",controller_in->Ka_Lon);
507
508         /* ------ d2b (distance to b) for guidance ------ */
509         nh.getParam("/controller_param/dist2b",controller_in->dist2b);
510
511         /* ------ KpLat gain for guidance ------ */
512         nh.getParam("/controller_param/KpLat",controller_in->KpLat);
513
514         /* ------ SAS switch 0- off 1-on for only autopilot 2-on for RC ------ */
515         nh.getParam("/controller_param/SAS_switch",helpers.SAS_flag);
516         if (helpers.SAS_flag == 2)
517                         controller_in->SASswitch = 1;
518         else
519                         controller_in->SASswitch = helpers.SAS_flag;
520
521         /* ------ SAS gains ------ */
522         nh.getParam("/controller_param/k_dtheta_delevator",controller_in->K_SAS_theta_de);
523         nh.getParam("/controller_param/k_dQ_delevator",controller_in->K_SAS_q_de);
524         nh.getParam("/controller_param/k_dP_daileron",controller_in->K_SAS_P_da);
525         nh.getParam("/controller_param/k_dR_drudder",controller_in->K_SAS_R_dr);
526         nh.getParam("/controller_param/K_phi_de",controller_in->K_phi_de);
527
```

```
528         /*------ correction switch ------*/
529         nh.getParam("/controller_param/correction_switch",(int&)controller_in->correction_switch);
530
531         /*------ Saturation switch ------*/
532         nh.getParam("/controller_param/Saturation_switch",(int&)controller_in->Saturation_switch);
533
534         /*------ phi_cmd_switch ------*/
535         nh.getParam("/controller_param/phi_cmd_switch",(int&)controller_in->phi_cmd_switch);
536
537         /*------ Vcorrection_switch ------*/
538         nh.getParam("/controller_param/Vcorrection_switch",(int&)controller_in->Vcorrection_switch);
539
540         /*------  K_speed ------*/
541         nh.getParam("/controller_param/K_speed",controller_in->K_speed);
542
543         /*------  Vstall ------*/
544         nh.getParam("/controller_param/Vstall",controller_in->Vstall);
545
546         /*------ K_acc_guidance ------*/
547         nh.getParam("/controller_param/K_acc_guidance",controller_in->K_acc_guidance);
548
549         /*------Acc_feedback_on------*/
550         nh.getParam("/controller_param/Acc_feedback_on",(int&)controller_in->Acc_feedback_on);
551
552         /*------throttle trim------*/
553         nh.getParam("/controller_param/throttle_trim",controller_in->throttletrim);
554
555         /*------elevator trim------*/
556         nh.getParam("/controller_param/elevator_trim",controller_in->elevatortrim);
557
558         /*------aileron trim------*/
559         nh.getParam("/controller_param/aileron_trim",controller_in->ailerontrim);
560
561         /*------rudder trim------*/
562         nh.getParam("/controller_param/rudder_trim",controller_in->ruddertrim);
563
564         /*------mpf radius------*/
565         nh.getParam("/controller_param/mpf_radius",controller_in->mpfradius);
566
567         /*------mpf switch------*/
568         nh.getParam("/controller_param/mpf_switch",(int&)controller_in->mpfswitch);
569
570         /*------path select------*/
571         nh.getParam("/controller_param/path_select",controller_in->path_select);
572
573         /*------Q weight------*/
574         nh.getParam("/controller_param/Q_weight",controller_in->Q_weight);
575
576         /*------max allow error------*/
577         nh.getParam("/controller_param/max_allow_error",controller_in->max_allow_err_d);
578
579         /*------VT trim------*/
580         nh.getParam("/controller_param/VTtrim",controller_in->VTtrim);
581
582         /*------guidance selection------*/
583         nh.getParam("/controller_param/guidance_selection",controller_in->Guidanceselection);
```

```
584
585         /*------dt------*/
586         nh.getParam("/controller_param/dt",controller_in->dt);
587 #if HIL_TEST
588         nh.getParam("/controller_param/gps_vn", controller_in->GPSVelocitymeas.V_north);
589         nh.getParam("/controller_param/gps_ve", controller_in->GPSVelocitymeas.V_east);
590         nh.getParam("/controller_param/gps_vd", controller_in->GPSVelocitymeas.V_down);
591 #endif
592 }
593
594
595 int main(int argc, char *argv[])
596 {
597         long iter, repeat = 0;
598         double interval_sec = (double)1 / 20;
599         bool auto_commit;  // 0 - not in initial autopilot
600                            // 1 - in intial autopilot
601         int switch_count = 1;  // to count number of times pic_cic is turned on
602         struct timespec start, end;
603         ros::init(argc, argv, "controller_DG808");
604         ros::NodeHandle n1;
605
606         ros::Time start_ros_timer, end_ros_timer;
607
608         ros::Subscriber sub_hil = n1.subscribe("/pwm", 1000, pwm_callback);
609         ros::Subscriber sub_gps_imu_hil=
610             n1.subscribe("/ins", 1000, gps_callback);
611         ros::Subscriber sub_waypoints = n1.subscribe("/way", 1000, way_callback);
612         ros::Subscriber sub_gps_pos = n1.subscribe("/gps_position", 1000, gps_position_callback);
613         ros::Subscriber sub_gps_velocity = n1.subscribe("/gps_velocity", 1000, gps_velocity_callback);
614         ros::Subscriber sub_imu_PQR_acceleration = n1.subscribe("/imu_PQR_acceleration", 1000, imu_PQR_acceleration_callback);
615         ros::Subscriber sub_imu_attitude = n1.subscribe("/attitude", 1000, imu_attitude_callback);
616
617         ros::Subscriber sub_DAQ_pressure = n1.subscribe("/pressure", 1000, DAQ_pressure_callback);
618
619         ros::Subscriber sub_DAQ_voltage = n1.subscribe("/battery_state", 1000, DAQ_voltage_callback);
620
621
622         g_pub_data_gs = n1.advertise < drone_ros::data > ("data", 1000);
623         g_pub_data_arduino = n1.advertise < drone_ros::pwm > ("servo_op", 1000);
624         g_pub_data_hil = n1.advertise < drone_ros::hil_data > ("to_hil", 1000);
625
626         ros::Rate loop_rate(20);
627         static int seq = 0;
628         seq++;
629         ros::Time timestamp = ros::Time::now();
630
631         // ------ To initialize the matlab generated structures   -----//
632         DG808_GNC_mov_initialize();
633
634         // ------ Assigning ROS parameters from launch file  ------ //
635         read_parameter();
636
637         clock_gettime(CLOCK_REALTIME, &start);
638         iter = 0;
639         start_ros_timer = ros::Time::now();
```

```
640         while (ros::ok()) {
641                 double remain_us;
642                 uint64_t tmpdiff;
643                 drone_ros::data msg_data;
644
645                 ros::spinOnce();
646
647 #if FLIGHT_TEST
648         /*if((controller_in->GPSPositionmeas.Latitude != 0) &&
649                 (controller_in->GPSPositionmeas.Longitude != 0) &&
650                 (controller_in->GPSPositionmeas.Altitude != 0))*/
651                         DG808_GNC_mov_step();
652                 PDEBUG("In␣Flight␣test␣mode",0);
653                 if(controller_in->PICCIC == 1)
654                 {
655                         switch(switch_count){
656                                 case 1: switch_count = switch_count + autopilot_testmode();  // in initial autopilot check
                                    don't need to send the pwm
657                                     PDEBUG("\n␣switch␣count␣is␣%d",switch_count);
658                                     while(controller_in->PICCIC){
659                                             ros::spinOnce();
660                                             selftest(0, 0, 0, 0);
661                                             if(!controller_in->PICCIC)
662                                                     continue;
663                                             loop_rate.sleep();
664                                     }
665                                     break;
666                                 default:
667                                         send_pwm();
668                         }
669                 }
670
671                 /* Output to the motor controller */
672                 PDEBUG
673                     ("Out:␣throttle=%f␣elevator=%f␣aileron=%f␣rudder=%f\n␣\n",
674                      controller_out->ControlSurfaceCommands.throttle_cmd,
675                      controller_out->ControlSurfaceCommands.elevator_cmd,
676                      controller_out->ControlSurfaceCommands.aileron_cmd,
677                      controller_out->ControlSurfaceCommands.rudder_cmd);
678                 clock_gettime(CLOCK_REALTIME, &end);
679                 iter++;
680                 tmpdiff = get_elapsed(&start, &end);
681                 PDEBUG("iter␣%ld␣took␣%" PRIu64 "us\n", iter, tmpdiff / 1000);
682                 set_ground_data(&msg_data);
683                 end_ros_timer = ros::Time::now();
684                 msg_data.duration = (end_ros_timer - start_ros_timer).toNSec() * 1e-9;
685                 g_pub_data_gs.publish(msg_data);
686                 printf("\n\n␣The␣time␣diff␣is␣%f", (end_ros_timer - start_ros_timer).toNSec() * 1e-9);
687                 // to run only at 20Hz
688                 loop_rate.sleep();
689 #endif
690                 //assign servo commands and servo deflection values
691                 assign_servo_commands();
692                 assign_servo_dfeflections();
693                 clock_gettime(CLOCK_REALTIME, &start);
694         }
```

```
695         DG808_GNC_mov_terminate();
696         return 0;
697 }
```

D.2 Sensor Data Acquisition Code

This section lists the code to read all sensor data and output the servo commands to the motors.

D.2.1 RC, Air Velocity and Battery Voltage

This sub section contains the code to read RC, air velocity and battery voltage from our custom built avionics board. This also contains code to send values to the actuator.

```
1  int main(int argc, char *argv[])
2  {
3          ros::init(argc, argv, "arduino");
4          ros::NodeHandle n;
5
6          // ------ Assigning ROS parameters from launch file ------ //
7          read_parameter();
8
9  #if FLIGHT_TEST
10         ros::Publisher pub_pwm;
11         pub_pwm = n.advertise<drone_ros::pwm>("pwm", 1000);
12
13         ros::Publisher pub_temperature;
14         pub_temperature = n.advertise<sensor_msgs::Temperature>("temperature", 1000);
15
16         ros::Publisher pub_pressure;
17         pub_pressure = n.advertise<sensor_msgs::FluidPressure>("pressure", 1000);
18
19         ros::Publisher pub_battery_state;
20         pub_battery_state = n.advertise<sensor_msgs::BatteryState>("battery_state", 1000);
21 #endif
22
23         ros::Subscriber sub = n.subscribe("/servo_op", 1000, servoreception);
24
25         ros::Rate loop_rate(20);
26
27         ros::Time timestamp = ros::Time::now();
28         int tty_fd = establish_serial();
29         char buf[1024];
30
31         usbstatus.usb_arduino = 1;
32         while (ros::ok()) {
33
34                 ros::spinOnce();
35
36 #if FLIGHT_TEST
```

```
37
38                  drone_ros::pwm msg_pwm;
39                  sensor_msgs::Temperature msg_temperatrure;
40                  sensor_msgs::FluidPressure msg_pressure;
41                  sensor_msgs::BatteryState msg_battery_state;
42
43                  getrc(tty_fd);          //gets the rc inputs
44                  getpressure(tty_fd);    //gets the pressure inputs
45                  gettach(tty_fd);        //gets the tach
46                  getcurrent(tty_fd);     //gets the current
47
48
49                  msg_pwm.header.stamp = timestamp;
50
51                  msg_pwm.PICCIC = helpers.auto_switch;
52                  msg_pwm.Throttle = helpers.th_percent;
53                  msg_pwm.Elevator = helpers.el_deg;
54                  msg_pwm.Aileron = helpers.ai_deg;
55                  msg_pwm.Rudder = helpers.ru_deg;
56
57
58                  msg_pressure.fluid_pressure = helpers.pressure;
59
60                  msg_pwm.velocity = helpers.air_velocity;
61
62                  msg_temperatrure.temperature = helpers.temperature;
63
64                  msg_battery_state.voltage = helpers.voltage;
65
66                  msg_pwm.tach = helpers.rpm;
67
68                  msg_battery_state.current = helpers.current[0];
69
70                  msg_pwm.usbstatus = usbstatus.usb_arduino;
71
72                  pub_pwm.publish(msg_pwm);
73                  pub_temperature.publish(msg_temperatrure);
74                  pub_pressure.publish(msg_pressure);
75                  pub_battery_state.publish(msg_battery_state);
76
77  #endif
78
79                  //send output
80                  sprintf(buf, "%d␣%d␣%d␣%d\n", (int)throttle, (int)elevator,
81                          (int)aileron, (int)rudder);
82
83                  sendoutput(tty_fd, buf);
84
85                  loop_rate.sleep();
86          }
87  }
```

D.2.2 GPS and IMU

We have used Vector Nav VN 200 [8] for measuring position and attitude. The following code is to read from VN200 sensor.

```
int main(int argc, char *argv[])
{
        ros::init(argc, argv, "vectornav");
        ros::NodeHandle n;


        std::string port;
        int baud;
        int check_time = 0;
        char fname[128];
        static int seq = 0;
#if FLIGHT_TEST

        ros::Publisher pub_ins;
        pub_ins = n.advertise < drone_ros::ins > ("ins", 1000);

        //add publisher handler for sensor_msgs/NavSat that contains GPS position
        ros::Publisher pub_nav_sat_fix;
        pub_nav_sat_fix = n.advertise < sensor_msgs::NavSatFix > ("gps_position", 1000);

        //add publisher handler for geometry_msgs/Twist that contains GPS velocity
        ros::Publisher pub_Twist_GPS_velocity;
        pub_Twist_GPS_velocity = n.advertise < geometry_msgs::Twist > ("gps_velocity", 1000);

        //add publisher handler for sensor_msgs/Imu that contains P,Q,R and linear acceleration
        ros::Publisher pub_Imu_PQR_Acceleration;
        pub_Imu_PQR_Acceleration = n.advertise < sensor_msgs::Imu > ("imu_PQR_acceleration", 1000);

        //add publisher handler for geometry_msgs/Point that contains Phi, Theta and Psi
        ros::Publisher pub_attitude;
        pub_attitude = n.advertise < geometry_msgs::Point > ("attitude", 1000);


        ros::Time timestamp = ros::Time::now();
        n.param < std::string > ("serial_port", port, "/dev/gps");
        n.param < int >("serial_baud", baud, 115200);
        n.param < std::string > ("imu/frame_id", imu_frame_id, "LLA");
        ros::Rate loop_rate(20);

        VN_ERROR_CODE vn_retval;

        ROS_INFO("Initializing␣vn200.␣Port:%s␣Baud:%d\n", port.c_str(), baud);

        vn_retval = vn200_connect(&vn200, port.c_str(), baud);
        if (vn_retval != VNERR_NO_ERROR) {
                char vn_error_msg[100];
                ROS_FATAL
                    ("Could␣not␣conenct␣to␣vn200␣on␣port:%s␣@␣Baud:%d;␣Error␣%d␣\n"
                     "Did␣you␣add␣your␣user␣to␣the␣'dialout'␣group␣in␣/etc/group?",
```

```
50                  port.c_str(), baud, vn_retval);
51          exit(EXIT_FAILURE);
52      }
53
54      unsigned short status;
55      int check = 0;
56      VnVector3 ypr, latitudeLognitudeAltitude, Velocity, accel, angularRate,
57          positionAccuracy;
58      double gpstime;
59      unsigned short week;
60      unsigned char fix, nos;
61      float speed_acc, time_acc;
62      clock_gettime(CLOCK_REALTIME, &start);
63      clock_gettime(CLOCK_REALTIME, &calc);
64
65      while (ros::ok()) {
66
67          double remain_us;
68          uint64_t tmpdiff;
69          //creating message handler for toics
70          drone_ros::ins msg_ins;
71          sensor_msgs::NavSatFix msg_gps_position;
72          geometry_msgs::Twist msg_gps_velocity;
73          sensor_msgs::Imu msg_imu_PQR_acceleration;
74          geometry_msgs::Point msg_attitude;
75
76          if (check <= 0)
77              vn200_getGpsSolution(&vn200,
78                                   &gpstime,
79                                   &week,
80                                   &fix,
81                                   &nos,
82                                   &latitudeLognitudeAltitude,
83                                   &Velocity,
84                                   &positionAccuracy,
85                                   &speed_acc, &time_acc);
86
87          if ((gpstime > 0) && (week > 0)) {
88              check++;
89              printf("\n\n the time is %lf", gpstime);
90          }
91          vn_retval = vn200_getInsStateLla(&vn200,
92                                           &ypr,
93                                           &latitudeLognitudeAltitude,
94                                           &Velocity, &accel,
95                                           &angularRate);
96          if (vn_retval != VNERR_NO_ERROR) {
97              char vn_error_msg[100];
98              msg_ins.header.seq = seq;
99              msg_ins.header.stamp = timestamp;
100             msg_ins.header.frame_id = imu_frame_id;
101             msg_ins.usbstatus = 0;
102         } else {
103             msg_ins.header.seq = seq;
104             msg_ins.header.stamp = timestamp;
105             msg_ins.header.frame_id = imu_frame_id;
```

```
106                         msg_attitude.x = ypr.c2 * 0.0174532925;//phi     // Intentional re-ordering
107                         msg_attitude.y = ypr.c1 * 0.0174532925;//theta
108                         msg_attitude.z = ypr.c0 * 0.0174532925;//psi
109
110                         msg_ins.time = gpstime;
111                         msg_ins.week = week;
112                         //msg_nos       = nos;
113
114                         msg_gps_position.latitude = latitudeLognitudeAltitude.c0;
115                         msg_gps_position.longitude = latitudeLognitudeAltitude.c1;
116                         msg_gps_position.altitude = latitudeLognitudeAltitude.c2;
117
118                         msg_gps_velocity.linear.x = Velocity.c0;
119                         msg_gps_velocity.linear.y = Velocity.c1;
120                         msg_gps_velocity.linear.z = Velocity.c2;
121
122                         msg_imu_PQR_acceleration.linear_acceleration.x = accel.c0 / GRAVITY;
123                         msg_imu_PQR_acceleration.linear_acceleration.y = accel.c1 / GRAVITY;
124                         msg_imu_PQR_acceleration.linear_acceleration.z = accel.c2 / GRAVITY;
125
126                         msg_imu_PQR_acceleration.angular_velocity.x = angularRate.c0;
127                         msg_imu_PQR_acceleration.angular_velocity.y = angularRate.c1;
128                         msg_imu_PQR_acceleration.angular_velocity.z = angularRate.c2;
129                         msg_ins.usbstatus = 1;
130                 }
131                 // Publishing the topics
132                 pub_ins.publish(msg_ins);
133                 pub_nav_sat_fix.publish(msg_gps_position);
134                 pub_Twist_GPS_velocity.publish(msg_gps_velocity);
135                 pub_Imu_PQR_Acceleration.publish(msg_imu_PQR_acceleration);
136                 pub_attitude.publish(msg_attitude);
137
138                 ROS_INFO("\nData:\n"
139                          "  YPR.Yaw:                %f\n"
140                          "  YPR.Pitch:              %f\n"
141                          "  YPR.Roll:               %f",
142                          ypr.c2, ypr.c1, ypr.c0);
143                 loop_rate.sleep();
144                 clock_gettime(CLOCK_REALTIME, &end);
145                 tmpdiff = get_elapsed(&start, &end);
146                 remain_us = (interval_sec * 1000000 - tmpdiff / 1000);
147                 printf("\n\ntook %llu", tmpdiff / 1000);
148                 clock_gettime(CLOCK_REALTIME, &start);
149                 clock_gettime(CLOCK_REALTIME, &calc1);
150                 double time_ms = get_time(&calc, &calc1);
151
152         }
153         vn200_disconnect(&vn200);
154 #endif
155         return 0;
156 }
```