**Cassiopeia - House of Computer Science**
**Student Report**
Selma Lagerlöfs Vej 300
Telephone +45 9940 9940
Fax +45 9940 9798
http://www.cs.aau.dk

**Title:**
Applying Application-Level Correctness to the Paparazzi Autopilot

**Theme:**
Specialisation in Computer Science

**Project period:**
DAT10, spring semester 2014

**Project group:**
d103f14

**Members:**
Heine Gatten Larsen
Morten Turn Pedersen
Thomas Viesmose Birch

**Supervisors:**
René Rydhof Hansen
Mads Christian Olesen

**Page count:** 68

**Appendix count:** 4

**Finished:** June 3, 2014

**Abstract:**

Unmanned aerial vehicles (UAVs) are becoming increasingly available. As a result of this, software which is cheap and accessible to everybody is needed.
The open-source Paparazzi project aims to provide a cheap software and hardware solution for UAVs. However, the Paparazzi software is not secured against outside factors resulting in individual bits changing their values. In this report we introduce a way of comparing different flight paths to each other, and we present a method to make the Paparazzi autopilot application-level correct. Finally we test our method's performance when a single event upset is injected into the autopilot during a flight.

*This page intentionally left blank*

# Contents

# Chapter 1

# Introduction

Today, technology is getting more autonomous, making human interaction with technology less important. As the technology becomes more autonomous, the tolerance for errors decreases. If an error occurs in a system and the system cannot fix it itself, human interaction is required to correct the error.
This is however not possible in e.g. autonomous drones, which may be deployed thousands of kilometres away from the home base. In such cases, errors can have dire consequences, even resulting in the loss of an important piece of technology.
To prevent autonomous technology from being rendered useless, it is required for them to not propagate small faults into larger, application-breaking errors.

Unmanned aerial vehicles (UAVs) are becoming more popular as a hobby and commercial tool due to their increased availability and decreased price. However cheap, mass-market UAVs are relatively simple in software design and are prone to single event upsets. A single event upset is where a single bit changes its state caused by outside factors such as cosmic radiation [12]. Securing against single event upsets can increase the computational overhead and mass-market UAVs do not posses much processing power. However numerical correctness is not vital in a hobby UAV, therefore application-level correctness is a viable and cheaper solution, with regards to processing power. The approach for numerical correctness is to ensure that every single value in a system is correct. Application-level correctness does not distinguish whether the values are correct or not, as long as the user does not perceive the fault. The field of application-level correctness is not as well researched as numerical correctness, but several solutions and methods have been published [5, 10].

In the worst case scenario: if a UAV experiences a single event upset, communication with the UAV could be lost. This could result in the UAV crashing, harming people, or make it impossible to recover it. In both cases the UAV would be lost.

In this project we will be looking into improving the autopilot system Paparazzi [13]. Specifically we will look into Paparazzi's fixedwing autopilot and improve it to withstand single event upsets. Unlike most fault tolerance work, we will stray from numerical correctness and look into application-level correctness. The reasoning behind this choice is that as long as faults are masked, goes unnoticed by the user, the program's execution will be perceived as correct.
Because of this choice, going through all functions and variables and implementing numerical correctness would not be a solution. A numerical correct solution would be a more effective, but also more processing power demanding solution than we are aiming for with an application-level correct approach. This is due to the trade-off between correctness and computation overhead.

We will model the Paparazzi autopilot in the verification tool UPPAAL [2]. By analysing the UPPAAL model, we will discover the vulnerabilities in the Paparazzi autopilot. We will also define a fidelity metric to imply the similarity of two flight paths. A fidelity metric is a mathematical function returning a numerical value e.g. describing how closely two flight paths resemble each other. An application-level correct solution for the Paparazzi autopilot will be presented using selective triple modular redundancy. By using the UPPAAL model in corporation with our fidelity metric, we will argue about the correctness of the proposed solution.

In Chapter 2 we will look at basic aviation terminology which applies to UAVs, fault tolerance, and application-level correctness. Chapter 3 will go into detail about existing solutions to application-level correctness.
Chapter 4 will delve into details about the Paparazzi project and the autopilot source code. In Chapter 5 we will analyse the Paparazzi implementation and present our fidelity metric. Chapter 6 will showcase our solution to ensure application-level correctness in Paparazzi and the testing of our solution. Finally in Chapter 7 we will summarise the findings of this report.

# Chapter 2

# Terminology

In this chapter we will describe basic terminology from the aviation industry, fault tolerance, and application-level correctness. This terminology will be used throughout the report. The following section is heavily based upon our previous work in [8].

## 2.1 Aviation

The aeroplane model used in this section is a Boeing 777, but the same terminology also applies to UAVs. We have discarded all the terminology not applicable to UAVs.

**UAV:** UAV is an abbreviation of *Unmanned Aerial Vehicle* and is commonly referred to as *drone*. As the name suggests, a UAV is an aircraft which does not have a pilot aboard during flight.

**Pitch:** The pitch of an aeroplane describes the angle of the nose of the aeroplane contra the tail of the aeroplane. If the pitch is increased, the nose will go up and vice versa. In other words, the pitch describes whether the aeroplane is facing upwards or downwards. See Figure 2.1 for an illustration.

**Roll:** An aeroplane can roll in two directions; left and right. The roll is often used for changing the heading of the aeroplane. The change is achieved by rolling the aeroplane and changing the pitch to turn. In other words, roll means turning on the axis going through the cabin of the aeroplane. See Figure 2.1 for an illustration.

**Yaw:** An aeroplane can yaw in two directions; left and right. When yawing, the direction, the aeroplane is facing, is turned left or right. Yawing is

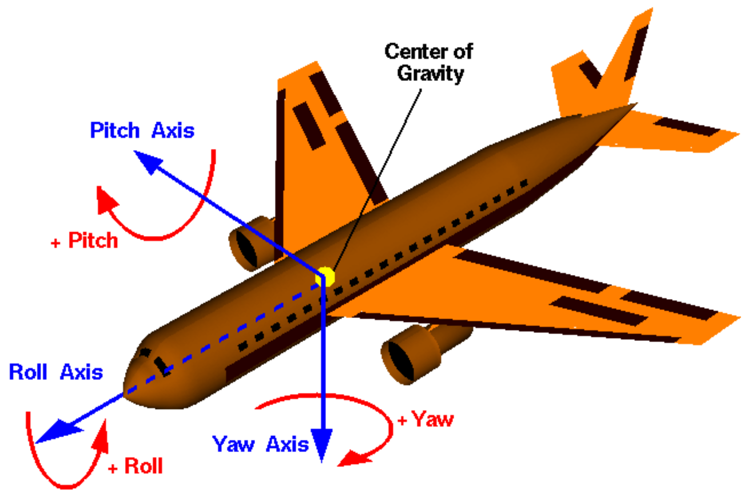


Figure 2.1: Aeroplane axes. Source: NASA [7]

not used for changing the heading of the aeroplane. Yawing is used mid-flight to counter crosswinds, and is also used for taxiing in the airport. See Figure 2.1 for an illustration.

**Angle of Attack:** The angle of attack of an aeroplane is the angle of the aeroplane's wings compared to the relative motion of the atmosphere. A higher angle of attack allows the aeroplane to ascend faster to a certain degree. The angle of attack is illustrated in Figure 2.2.

**Stall:** Stall is a condition where the angle of attack, or pitch, is so high that the wind travelling over the wing decreases to a point where the aeroplane becomes unable to support its own weight. In a stall condition, the pilots must decrease the pitch and increase travel speed in order to gain enough lift from the wings again.

**Slats:** Slats are also known as leading-edge slats. They are positioned on the front of the wing and are used to give the aeroplane a higher angle of

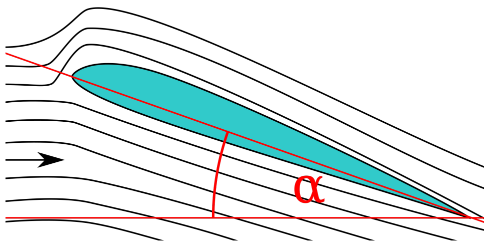


Figure 2.2: The angle of attack, $\alpha$, illustrated. Source: Wikimedia [9]

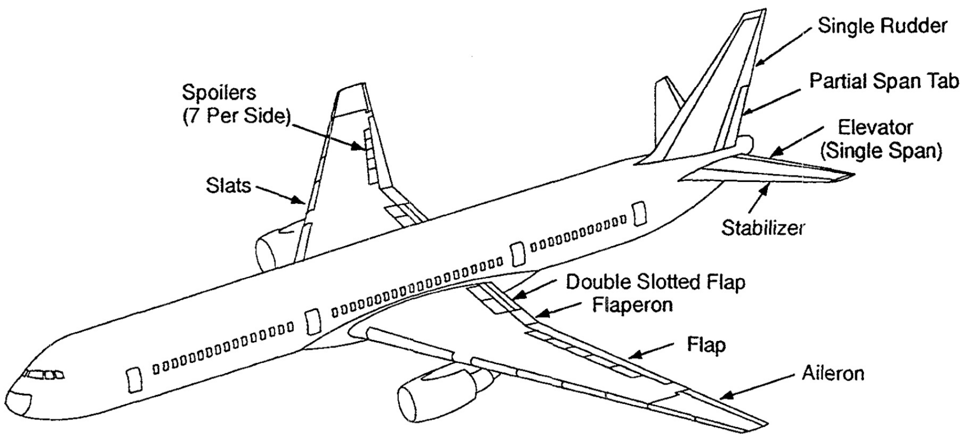


Figure 2.3: Aeroplane overview of a Boeing 777. From [19]

attack. They are primarily deployed and used during landing to allow the aeroplane to fly in at lower speeds.

**Spoilers:** A spoiler is used to reduce lift by corrupting the airflow over the wing, thus creating a controlled stall. They are placed on top of the wings and are normally used during descent to avoid picking up speed.

**Flaps:** There are two usages for flaps, during takeoff and during landing. During takeoff they can be used to shorten the distance required to get airborne. This effect is achieved by lowering the stall speed and by increasing the amount of drag across the top of the wing, giving more lift.
During landing the flaps can be used to achieve the same effect as during takeoff by allowing the aeroplane to have a sharper angle of descent and to reduce the speed the aeroplane needed to be flown safely.

**Aileron:** The aileron is a small control surface placed near the wing tips. They are used in pairs, one on each wing to allow the aeroplane to roll.

**Flaperon:** A combination of flaps and ailerons.

**Trim:** Trim is an extra tool for the pilots to ease their workload and give them better control over the aeroplane's behaviour. Trim locks one of the controls of the aeroplane into a specific setting, such that the pilot e.g. does not have to maintain backwards pressure on the yoke to gain altitude. There are usually three trims, one for each of the axes of the aeroplane.

**Rudder:** The rudder is a directional control surface. It allows the aeroplane to yaw. It is also used to stabilise the aeroplane against crosswinds.

**Elevator:** The elevator is a control surface which allows the pilot to change the pitch of the aeroplane.

**Stabilizer:** The horizontal part of the tail. It is used to trim the elevation of the aeroplane.

**Overspeed:** Overspeed is a condition where the aeroplane's speed is higher than the aeroplane was designed for. Overspeed is not a constant, it varies depending on air pressure and other conditions.

**Elevation:** The elevation is the altitude of the aeroplane above a fixed reference point. The reference point is usually the sea level, but is not limited to sea level.

**APS:** APS is an abbreviation of AutoPilot System and is the general term used to describe the autopilot on an aeroplane.

**FBW:** FBW is an abbreviation of Fly-By-Wire and is a system used by e.g. Boeing. The FBW system translates the analogue input from the pilots to hydraulic pressure used to affect the control surfaces of the aeroplane.
In UAVs hydraulic pressure is not necessarily used in the FBW systems. More often electric power is used to run servos, this is possible since the forces generated by the smaller plane is controllable using electric power.

**Flight Plan:** A flight plan is a series of checkpoints outlining a route through the air which an aeroplane is supposed to follow.

**Flight Path:** A flight path is a route through the air which an aeroplane has travelled.

## 2.2 Fault Tolerance

We differ between three types of threats towards the application: fault, error and failure. These definitions are as in [8, 1].

**Error:** An error is a system state that may cause a subsequent failure.

**Failure:** A failure occurs when an error reaches the service interface and alters the service. A system's service is its behaviour as it is perceived by its user.

**Fault:** A fault is the adjudged or hypothesised cause of an error. A fault can have two states: it may be active when it creates an error, otherwise it can be dormant.

### 2.2.1 Application-Level Correctness

For a UAV, only a limited amount of resources for computation is available, therefore a full fault tolerant implementation is too costly to implement. A solution for achieving fault tolerance without increasing the overhead too much is application-level correctness.
A mission for a UAV could be to travel its route without diverging too much from its flight plan. Therefore numerical correctness on the hardware level is not necessary since some divergence is acceptable. The important parts of the UAV's control systems are that it does not crash, deadlock, or in any other way prevent the UAV from completing its mission.
We define application-level correctness in regards to a UAV in Definition 2.1 which is based on the definition by Li and Yeung in [10].

**Definition 2.1.** A UAV is application-level correct if the UAV completes its flight without diverging significantly from its planned course.

Application-level correctness has been used in other areas such as multimedia decompression where e.g. a pixel error does not affect the user's multimedia experience [10].
With regards to UAV applications, application-level correctness (as defined in Definition 2.1 above) is when a UAV completes a preplanned flight plan without flying off course, thereby following the preplanned route within an acceptable margin. A bit of deviation from the route is to be expected even in a faultless run due to outside influences and the fact that a UAV cannot make a perfect 90 degree turn.

# Chapter 3

# Related Work

In this chapter we present some of the previous research done within fault tolerance and single event upsets detection focusing on application-level correctness.

## 3.1 Soft Error Detection Through Software Fault Tolerance

Soft Error Detection Through Software Fault Tolerance techniques [14] or SEDTSFT for short, is used to create fault-tolerance against soft errors in high-level code. They focus on detecting errors by adding redundancy and leave the recovery up to the user. Their targeted language is `C`, and they provide rules for transforming all basic `C` constructs. They claim their approach is general enough to also work for other high-level languages. Their approach does require the user to disable compiler optimisation. This is because the compiler optimisation will remove the fault tolerance operations because it will be identified as redundant operations.

### Code Modification

In Table 3.1 an example of transforming a variable assignment can be seen. Each variable is now stored in two separate memory locations, and a variable check is inserted on the variable each time it is used in an instruction. This ensures that every time a variable is used it is checked with the duplicated value, if they do not contain the same value, the special error function is called.
In Table 3.2 an example of a function transformation can be seen. Note that the return values has been removed, and two variables are instead passed by reference. This is done because the function now needs to return two values instead of one. Note again that they only check if variable contains the correct value right after they are used in an assignment.

| Original code | Modified code |
|---|---|
| $a = b;$ | $a_0 = b_0;$<br>$a_1 = b_1;$<br>$if(b_0! = b_1)$<br>    $error();$ |
| $a = b + c;$ | $a_0 = b_0 + c_0;$<br>$a_1 = b_1 + c_1;$<br>$if((b_0! = b_1)\|\|(c_0! = c_1))$<br>    $error();$ |

Table 3.1: Example of translation [14]

| Original code | Modified code |
|---|---|
| $res = search(a);$<br>$\dots$<br>$int\ search(int\ p)$<br>$\{$<br>    $int\ q;$<br>    $\dots$<br>    $p = q + 1;$<br>    $\dots$<br>    $return(1);$<br>$\}$ | $search(a_0, a_1, \&res_0, \&res_1);$<br>$\dots$<br>$void\ search(int\ p_0, int\ p_1, int\ *r_0, int\ *r_1)$<br>$\{$<br>    $int\ q_0, q_1;$<br>    $\dots$<br>    $q_0 = p_0 + 1;$<br>    $q_1 = p_1 + 1;$<br>    $if(p_0! = p_1)$<br>        $error();$<br>    $\dots$<br>    $*r_0 = 1;$<br>    $*r_1 = 1;$<br>    $return;$<br>$\}$ |

Table 3.2: Example of function translation [14]

```
                           majority(r4,r4',r4")
ld r3 = [r4]               ld r3 = [r4]
                           mov r3' = r3
                           mov r3"= r3
add r1 = r2, r3            add r1 = r2, r3
                           add r1'= r2', r3'
                           add r1"= r2",r3"
                           majority(r1,r1',r1")
                           majority(r2,r2',r2")
st [r1] = r2               st [r1] = r2
```

(a) Original code (b) SWIFT-R transformed code

Figure 3.1: Example of SWIFT-R transformation as seen in [4]

## 3.2 SWIFT-R

Unlike the approach in the previous section, the SWIFT-R approach [4] triplicates all assembly instructions and uses a majority vote function to not only detect errors, but also recover from these errors. An example can be seen in Figure 3.1. In Figure 3.1a the original code is shown. The purpose of the code is to add the values in register $r2$ and $r3$ and store the result in $r1$. In Figure 3.1b the modified code is shown. In the first step, the majority function is used to determine if any inconsistencies exist. It is worth noting that SWIFT-R does not require hardware changes, but requires that ECC[1] memory is used. The ECC memory automatically detects and corrects internal memory corruptions.

## 3.3 AALCASE

Cong and Guraraj present in [5] a way of securing application-level correctness using a pre-compile analysis and runtime monitoring. We will now look at the two different parts of the solution.

**Analysis** Cong and Guraraj have implemented their analysis techniques as a pass in the LLVM[2] compiler framework. We use the same example as Cong and Guraraj uses, which can be seen in Listing 3.1 with the resulting LLVM IR[3] code is shown in Listing 3.2.
The analysis of the code starts by calculating the weighted program dependence graph (PDG). The PDG shows which LLVM IR instructions are

[1]ECC is an abbreviation of Error-Correcting Code
[2]LLVM is an abbreviation of Low-Level Virtual Machine
[3]IR is an abbreviation of Intermediate Representation

```
X=sqrt(Y);
for(i=1; i<N; ++i)
{
  C[i] = C[i-1] + i;
  output[i] = C[i] + X;
}
```

Listing 3.1: Running example from [5]

```
entry:
X = call sqrt(Y);
bb:
i = phi [entry,1] [bb, i_inc]
c_i_1 = load &(C[i-1])
add_C = c_i_1 + i
store add_C, &(C[i])
c_i = load &(C[i])
out_i = add c_i, X
store out_i, &(output[i])
i_inc = add i, 1
cond = cmp i_lt, i_inc, N
br cond bb, exit
```

Listing 3.2: LLVM IR of example [5]

dependent on which. The weights in the PDG describes the maximum number of instances of the target nodes that are dependent on the source node. The resulting PDG from analysing Listing 3.2 can be seen in Figure 3.2. Using the weights of the edges, the analysis determines whether the instruction is critical or not. If an instruction is marked as critical, the instruction is duplicated and checked at runtime. If an error is detected, the execution is rolled back to the start of a basic block and instructions are re-executed. A basic block is a sequence of instructions where there is only one entry point and one exit point.

**Runtime Monitoring** The task of the runtime monitoring is to keep track of the edge weights in runtime. If an edge weight increases above a specified threshold, a signal is triggered for making the instructions critical in runtime.

## 3.4 Rely

Carbin, Misailovic, and Rinard present in [3] the programming language Rely, which enables the programmer to differentiate between reliable and unreliable computations. The model of the machine Rely's code is running

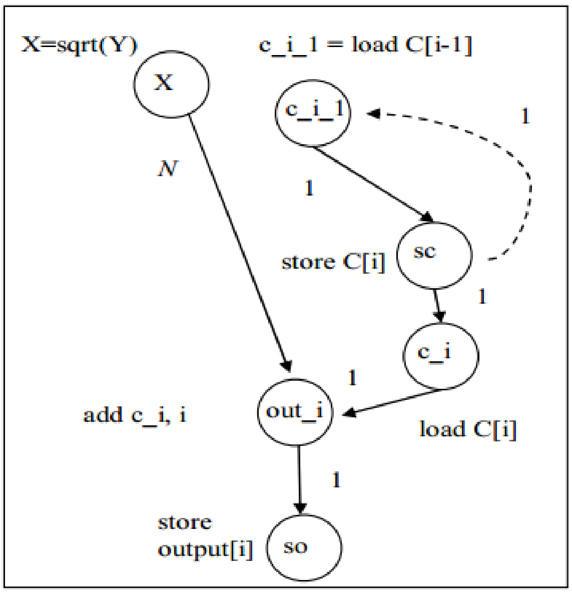


Figure 3.2: The resulting PDG by analysing the code in Listing 3.2. From [5]

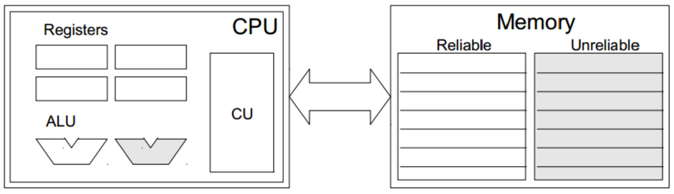


Figure 3.3: Rely's hardware illustration. The gray boxes are unreliable components, where the white boxes are reliable components. From [3]

on can be seen in Figure 3.3.

The intuition behind Rely is that unreliable hardware is cheaper than reliable hardware. Unreliable hardware allows Rely to achieve more processing power at a cheaper cost. However not all applications are able to function without numerical correctness. This is dealt with by having both reliable and unreliable hardware. Note that the registers on the CPU and the computational unit are reliable, only the arithmetic logic unit has an unreliable component as well as the memory.
Rely's syntax is similar to ordinary `C` code. Listing 3.3 shows how variables can be declared. Lines 1-3 shows how unreliable variables are declared, or in other words the variables are stored in the unreliable memory. Line 4 shows how reliable variables are declared.
Along with unreliable variable declaration, unreliable arithmetic and boolean functions can be utilised. Listing 3.4 shows the difference between how re-

```
int minssd = INT_MAX,
    minblock = −1 in urel;
int ssd, t, t1, t2 in urel;
int i = 0, j, k;
```

Listing 3.3: Rely variable declaration example [3]

```
int xr;
int xu in urel;
xr = 5 + 4; //reliable computation
xu = 5 +. 4; //unreliable computation
```

Listing 3.4: Rely arithmetic functions example

liable and unreliable arithmetic calls are made. The syntax for unreliable operations are almost the same as the reliable operations, a '.' is added after the operator, as seen in Listing 3.4.

## 3.5 Summary

In the previous sections we have presented several solutions for fault tolerance. In the SEDTSFT solution, double modular redundancy was proposed on a `C` code level. This solution will not suite our needs as it does detect single events upsets, but cannot correct them.
In the SWIFT-R solution a triple modular redundancy on assembly level is proposed. SWIFT-R has a suitable approach for our needs. However we will focus on the `C` code level and use SWIFT-R as an inspiration. The AALCASE proposed solution is, as with SWIFT-R, on a lower programming level than the focus of this report.
Rely is a programming language which focuses on the unreliable region of hardware, and tries to envelop it in the language. However Rely assumes that the reliable region is reliable, meaning that errors cannot occur here. This does not apply to modern computer architecture, as the correctness is decreasing due to smaller components. Due to these factors, Rely is not suitable for our needs.

# Chapter 4

# The Paparazzi Project

Paparazzi is an open-source autopilot system oriented towards inexpensive autonomous aircraft of all types. The Paparazzi Project was started in 2003 and has been under continuous development ever since.
Paparazzi aims at developing a full system, hardware and software, for controlling a fixedwing UAV [11, 13]. This includes the airborne processor with its required sensors, the airborne autopilot software, a ground control station, the communication protocols linking the different components, and a simulation environment.
An overview of a Paparazzi setup is illustrated in Figure 4.1. This setup will be explained further in the following sections.

## 4.1 Hardware

The autopilot hardware has been created using Atmel AVR and Philips ARM7 LPC microcontrollers. The hardware includes a single or dual microcontroller and the required connectors to handle servos, motor controllers,

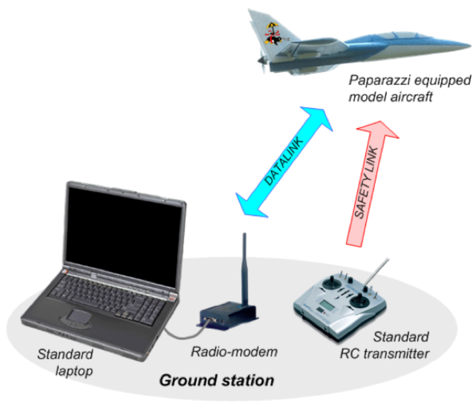


Figure 4.1: Paparazzi overview. From [13]

R/C receiver, and sensors.
The hardware includes a minimum set of sensors in an effort to keep costs low and reliability high. Sensors are used to calculate e.g. position and altitude using infrared sensors, GPS receiver, and an optional gyroscope.

### 4.1.1 Architecture

Paparazzi relies on cheap existing hardware for the UAV's control system, along with a laptop and an antenna on the ground. The aeroplane control system consists of a control cord, GPS, infrared sensors, radio-modem, and an optional gyroscope. The control system is interconnected with the existing components of the aeroplane such as battery, servos, and a data link. It is possible to add a range of extra equipment to the aeroplanes. In [13], Brisset et al. give examples of attaching a camera with a video transmitter or a paintball gun.

The infrared sensors are used to calculate the pitch, roll, and altitude by measuring the difference in temperature between the sky and the ground. They measure the temperature over the wings, along the length of the aeroplane, and on the top and bottom of the aeroplane.

The control system, which is the main hardware component, consists of one or two microprocessors, either one controlling the autopilot and one controlling the Fly-By-Wire process or both on one microprocessor. The Fly-By-Wire process contain all the critical code and controls all the critical parts of the aeroplane's operation, including servo control and the R/C override. The R/C override makes it possible for the operators to take manual control over the aeroplane to try and recover it in case of an error. The autopilot process controls the navigation, sensors, payload (if any), communication, and most importantly the autonomous processing.

If we look into the autopilot, it is composed of three steps. First step is the sensor acquisition where the data from the GPS, infrared sensor, and the gyroscope is collected. The data is then sent to the state estimation in the second step where it is used to smooth the input before it is passed to the last step. The last step consists of a stack of control loops that calculate what actions to take and send the commands to the actuators. For an overview see Figure 4.2.

## 4.2 Ground Control Station

From the Ground Control Station (GCS) the user is able to monitor the status of one or more UAVs and control them through a graphical user interface.

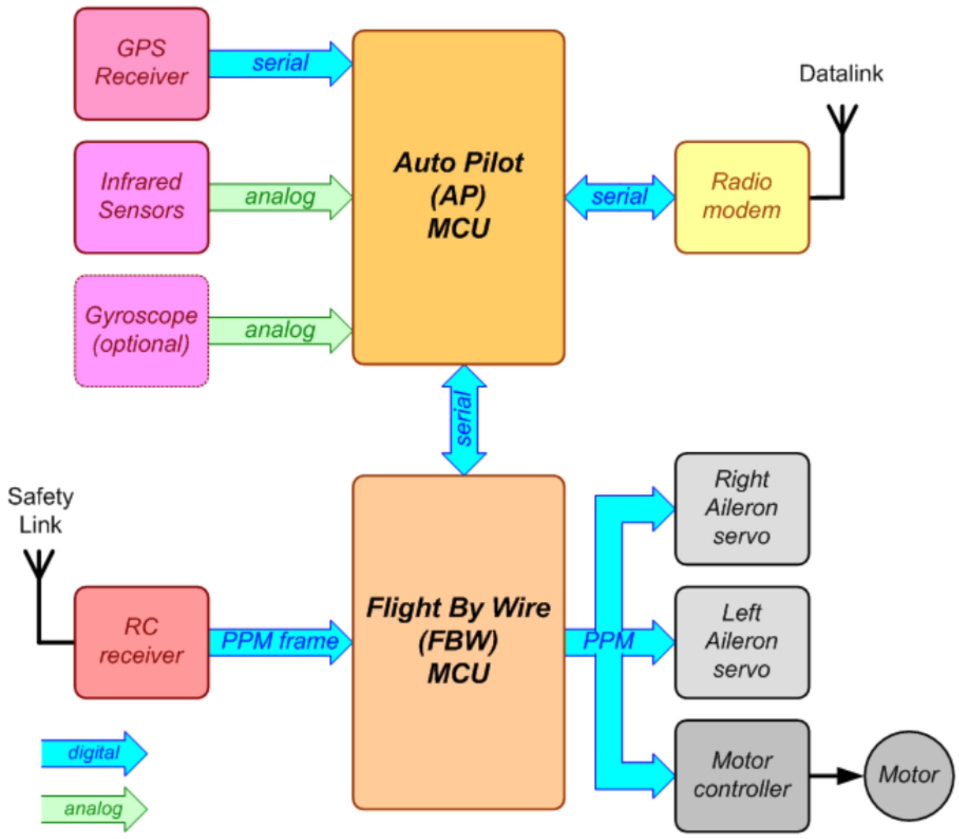


Figure 4.2: Paparazzi autopilot design. From [13]

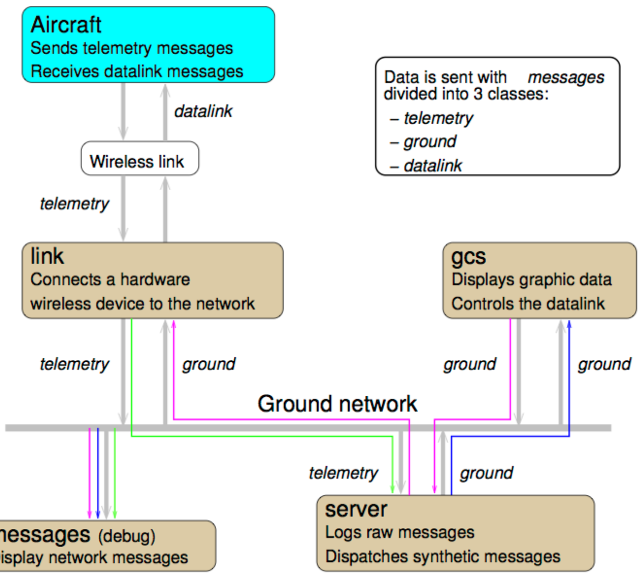
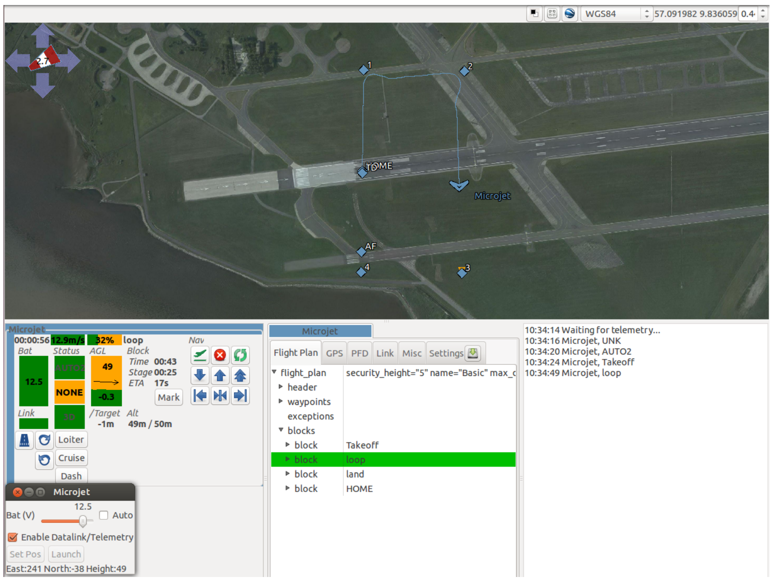


Figure 4.3: Ground control station architecture. From [13]

The distributed architecture for the ground control station is illustrated in Figure 4.3. The architecture is split into three main agents which are connected to a bus:

- The *link*-agent is responsible for the hardware peripherals. It handles message translation between the bus and the peripheral (usually a serial link).
- The *server*-agent records the messages coming from the *link*-agents in a log and dispatches synthetic information to other listening agents (e.g. the graphical user interfaces). Configuration (airframe description, flight plan etc.) of the flying aircraft and making it available on the bus is also handled by this agent. Finally the agent also computes some environmental information (wind, ground speed etc.) and broadcasts it on the bus.
- The *gcs*-agent is the graphical user interface. It displays all information coming from the server agent and sends back orders from the operator.

A screenshot of the Paparazzi ground station user interface is displayed in Figure 4.4.



```
flight_plan     security_height="5" name="Basic" max_c
  header
  waypoints
  exceptions
  blocks
    block       Takeoff
    block       loop
    block       land
    block       HOME
```

```
10:34:14 Waiting for telemetry...
10:34:16 Microjet, UNK
10:34:20 Microjet, AUTO2
10:34:24 Microjet, Takeoff
10:34:49 Microjet, loop
```

Figure 4.4: Ground control station GUI

### 4.2.1 Flight Plans

A flight plan in Paparazzi is an XML document which specifies how the aircraft should travel in autonomous mode. The flight plan is composed of waypoints and block elements. The Paparazzi system includes a flight plan editor which can be used to create or edit flight plans.
Along with the flight plan, a definition of `HOME` is provided. `HOME` is the base point of the flight plan and it is also the place that the UAV returns to if something goes wrong.
The root `flight_plan` element is specified with several attributes:

```
<flight_plan name lat0 lon0 ground_alt security_height
home_mode_height qfu alt max_dist_from_home>
```

- `name` - The name of the flight plan
- `lat0, lon0` - Defines the latitude/longitude coordinates of the reference point {0,0}.
- `ground_alt` - The ground altitude.
- `security_height` - The altitude of which flying around is safe.
- `home_mode_height` - The altitude of which flying around the home waypoint is safe.
- `qfu` - The magnetic heading in degrees.

- `alt` - Default altitude of waypoints.
- `max_dist_from_home` - The maximum allowed distance from `HOME`.

A waypoint element is defined by:

```
<waypoint name wpx wpy [alt] [height]/>
```

- `wpx, wpy` - Real positional coordinates from your reference point {0,0}.

A waypoint with the name `HOME` is required for any flight plan, as the autopilot will use this waypoint in case of failure.

A block element is used group several navigation modes of where the aircraft should travel to. Exceptions are also supported, where a giving condition would let the autopilot go to another block.

The navigation modes includes:

- attitude - hold a fixed attitude,
- heading - keep a given course,
- go - fly to a given waypoint,
- path - fly to a given list of waypoints
- circle - circle around a waypoint,
- oval - fly around the two waypoints in an oval where the long sides are straight,
- eight - fly a in figure of an eight through a waypoint and around another,
- stay - hold the position,
- follow - follow another aircraft,
- xyz - circle around a moveable point giving with the R/C transmitter stick.

The vertical control of the aircraft is achieved with the following properties:

- alt (default) - hold the given altitude,
- climb - hold the given vertical speed (m/s)
- throttle - set the given throttle (between 0 and 1),
- glide - hold the given slope between two waypoints.

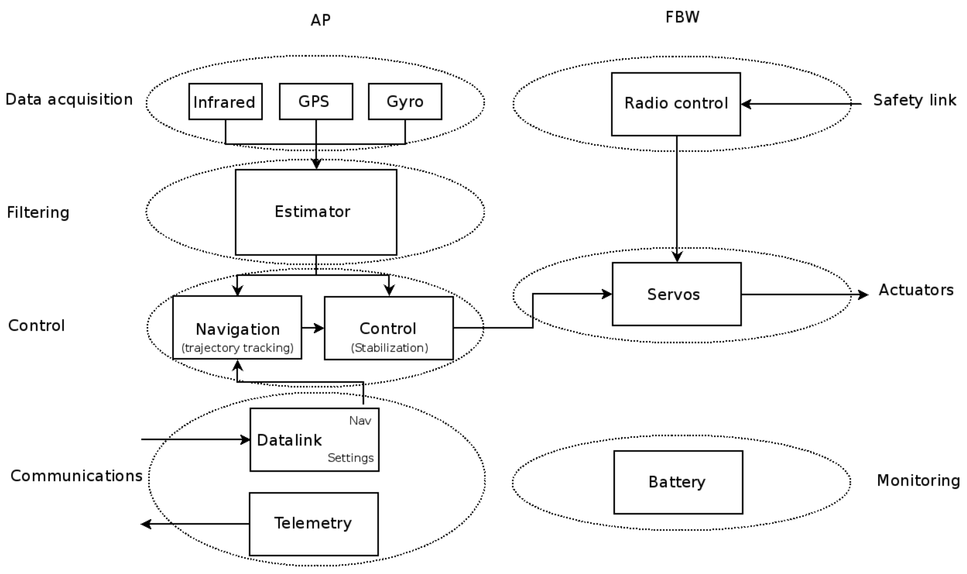


Figure 4.5: Paparazzi functional diagram. Source: Paparazzi Wiki [16]

### 4.2.2 Simulation

The Paparazzi Project includes a simulator which is able to run the airborne code on the host computer. Paparazzi includes three different simulator "backends" for a Flight Dynamic Model (FDM). The FDM is used in the simulator for calculating the physical forces acting upon the aircraft e.g. lift, thrust, and drag. The differences between the three simulators are varying degrees of realism and intended purpose. The three are:

1. **sim:** The basic fixedwing simulator written in OCaml
2. **jsbsim:** A more advanced fixedwing simulator. JSBSim is an open source flight dynamics library.
3. **nps:** A more advanced rotorcraft simulator with sensor models and currently also uses JSBSim as FDM.

## 4.3 Fixed Wing Autopilot in Paparazzi

In this section we will take a closer look at the Paparazzi source code. We will mainly focus on the fixedwing autopilot source in detail. This is the default UAV used in Paparazzi and also the UAV used in the Paparazzi examples. In Figure 4.5 a functional diagram of the APS and FBW is illustrated.

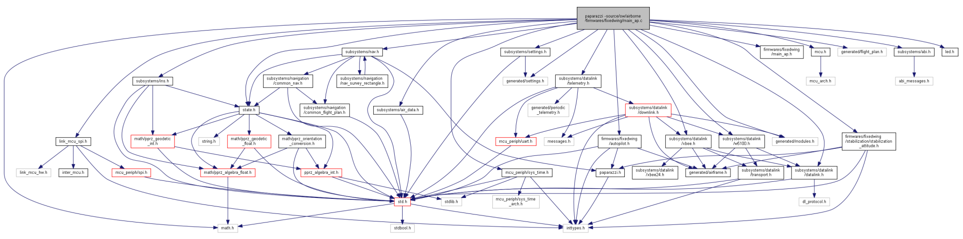

paparazzi -source/sw/airborne /firmwares/fixedwing/main_ap.c
subsystems/nav.h
subsystems/ins.h
subsystems/navigation /common_nav.h
subsystems/navigation /nav_survey_rectangle.h
state.h
subsystems/navigation /common_flight_plan.h
subsystems/air_data.h
link_mcu_spi.h
math/pprz_geodetic _int.h
string.h
math/pprz_geodetic _float.h
math/pprz_orientation _conversion.h
link_mcu_hw.h
inter_mcu.h
mcu_periph/spi.h
math/pprz_algebra_float.h
pprz_algebra_int.h
std.h
stdlib.h
math.h
stdbool.h
subsystems/settings.h
generated/settings.h
subsystems/datalink /telemetry.h
generated/periodic _telemetry.h
subsystems/datalink /downlink.h
mcu_periph/uart.h
messages.h
firmwares/fixedwing /autopilot.h
subsystems/datalink /xbee.h
subsystems/datalink /w5100.h
generated/modules.h
firmwares/fixedwing /stabilization/stabilization _attitude.h
mcu_periph/sys_time.h
paparazzi.h
subsystems/datalink /xbee24.h
generated/airframe.h
subsystems/datalink /transport.h
subsystems/datalink /datalink.h
mcu_periph/sys_time _arch.h
dl_protocol.h
inttypes.h
firmwares/fixedwing /main_ap.h
mcu.h
generated/flight_plan.h
subsystems/abi.h
led.h
mcu_arch.h
abi_messages.h

Figure 4.6: The dependencies of main_ap.c. Enlarged version in Appendix A

### 4.3.1 Math Library

Paparazzi includes their own math library which is used by all supported autopilots. The math library is located in `(paparazzi)/sw/airborne/math` and provides functions to the autopilot. These includes:

- Manipulate euler angles, quaternion and rotation matrix
- Basic trigonometry using fixed-point algebra
- Perform geodetic transformations (e.g rotation of vectors into ENU[1] frame which are used to describe the aircraft's attitude)

### 4.3.2 Subsystems

Every autopilot in Paparazzi uses one or more subsystems. These are located in `(paparazzi)/sw/airborne/subsystems`. Subsystems is a convention used to provide several implementations of a peripheral (microcontroller peripherals, an external IMU (Inertial Measurement Unit) board etc.), protocols (GPS, communications etc.) and algorithms (control, estimation etc.). An IMU is a sensor which is only used to measure the accelerations and rotation rates. An IMU is not used on the fixedwing autopilot which uses the infrared light sensor instead.

#### Main Autopilot Loop

This section explains the main files which are used in the autopilot. These main files utilise the math library and several subsystems, explained in the previous sections, for controlling the aircraft.
The main autopilot loop's file dependencies can be seen in Figure 4.6. An enlarged version of the dependency graph can be found in Appendix A.
Due to the complexity of the dependencies, we will now take a high level look at the `handle_periodic_tasks_ap` loop. The loop which handles the periodic tasks in the autopilot can be seen in Listing 4.1. The code for

[1]ENU is an abbreviation of East, North, Up

```c
void handle_periodic_tasks_ap(void) {
  if (sys_time_check_and_ack_timer(sensors_tid))
    sensors_task();

  if (sys_time_check_and_ack_timer(navigation_tid))
    navigation_task();

#ifndef AHRS_TRIGGERED_ATTITUDE_LOOP
  if (sys_time_check_and_ack_timer(attitude_tid))
    attitude_loop();
#endif

  if (sys_time_check_and_ack_timer(modules_tid))
    modules_periodic_task();

  if (sys_time_check_and_ack_timer(monitor_tid))
    monitor_task();

  if (sys_time_check_and_ack_timer(telemetry_tid)) {
    reporting_task();
    LED_PERIODIC();
  }
}
```

Listing 4.1: Main loop for the autopilot from main_ap.c

each of the functions in `handle_periodic_tasks_ap(void)` can be seen in Appendix B.
Each of the tasks are timed and should only be run in a certain frequency. Therefore before each task, there is a check whether enough time has elapsed since last execution. This is achieved via the function `sys_time_check_and_ack_timer(tid_t id)`. If enough time has passed since the last time the task has executed, the task will execute.
Note that each task has its own specification of how often the function is executed. This is saved in a struct which contains information of how much time has to pass until the next execution, and how much time has passed since last execution.

**sensors_task** The sensors task collects the data from the sensors on the fixedwing aircraft. It starts by collecting data from the data link (`ahrs_propogate`) which may contain navigational data for the aircraft. The sensor task also collects data from the AHRS, which is the infrared units calculating heading, attitude and yaw. Afterwards, it checks whether the GPS unit is available or not. Finally the `sensors_task` calculates the state of the aircraft using the function `ins_periodic`.

**navigation_task** The navigation task keeps track of which navigational mode the aircraft is in and acts accordingly. It starts by checking whether the GPS is available or not (using timeout), and afterwards calculates the route according to its autopilot mode. The Paparazzi project uses the following modes:

- `PPRZ_MODE_MANUAL` - Manual control from the GCS.
- `PPRZ_MODE_AUTO1` - Automatic stabilisation of the aircraft.
- `PPRZ_MODE_AUTO2` - Automatic stabilisation and navigation of the aircraft.
- `PPRZ_MODE_HOME` - The aircraft flies back to "home".
- `PPRZ_MODE_GPS_OUT_OF_ORDER` - The GPS receiver fails or temporarily loses signal, the aircraft navigates without access to the GPS.
- `PPRZ_MODE_NB` - Not used in the source code.

**attitude_loop** The attitude loop handles the aircraft's attitude meaning that its position and heading are correct. This is done by manipulating the pitch and the throttle accordingly.

**modules_periodic_task** This loop handles the control of the aircraft. This task is generated specifically for each type of aircraft, due to them having unique configurations.

**monitor_task** The monitor's task is to keep track of the flight time. After updating the flight time it checks whether there is enough battery power left to complete the mission or not. The monitor then checks whether the aircraft has passed too far away from `HOME`. If this is the case, it kills the throttle. Finally it checks whether the launch procedure has occurred or has been completed correctly, if either of these are false, it resets the aircraft to the beginning of a flight with regards to flight time and sending a signal to the GCS that takeoff has begun.

**reporting_task** The reporting task's task is to send periodic telemetry to the GCS on the ground. At the first iteration, it sends an additional signal to the GCS that the aircraft has booted.

### 4.3.3 Simulator

When starting the simulator in Paparazzi the program compiles the autopilot and uses the output files for running the autopilot and simulating the flight.

### Failsafe

Paparazzi includes several built-in failsafe features on different kind of levels. At the high-level the flight plans, described in Section 4.2.1, includes exceptions which makes it possible to make the flight plan failsafe. On the lower level, the FBW system and autopilot include different failsafe features e.g. losing RC connection, losing GPS connection, and a low battery voltage.

Should the aircraft reboot in midair, the aircraft would start over its flight plan and climb to the altitude of which the takeoff altitude is defined. Afterwards the rest of the flight plan would be executed. This means that waypoints already visited before the reboot will be visited again.

### Scheduling

No real scheduling takes place in the normal version of Paparazzi, however a real time version is in development [6]. In the normal version the main loop runs as fast as possible, and each task executes at the specific programmed interval. The tasks try to acquire the timer when they need to execute, and then releases the timer once they finish. There is currently nothing to stop a subroutine if it uses to much computation time or is stuck in an endless loop.

# Chapter 5

# Analysis of the Paparazzi Autopilot

In this chapter we will present an analysis of the Paparazzi autopilot. This analysis will be used for applying application-level correctness to the Paparazzi autopilot.

## 5.1 Overview

In Figure 5.1 a diagram of our view of the Paparazzi autopilot microcontroller architecture is illustrated. The *program* in our case is the Paparazzi autopilot system. As the figure shows, the program does not have direct access to the memory. All memory accesses go through the CPU. In our view of the architecture, only the main memory can experience single event upsets. Everything inside the CPU is assumed to execute numerically correct and the program is stored on a non-volatile numerical correct disk.

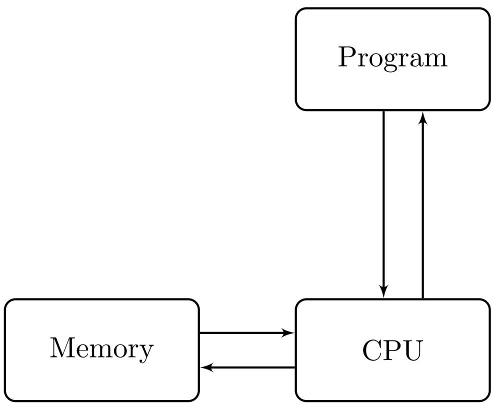


Figure 5.1: A figure of our assumptions

## 5.2 Assumptions

In our solution we will be making the following assumptions:

1. We assume that all computations performed in the GCS execute with numerical correctness. The reasoning behind this assumption is that the GCS runs on a normal laptop and the increase in computational cost for running a numerical correct version of the software is achievable.

2. We assume that no errors exist in the Paparazzi code and if any errors exists, we assume it is intentional. We assume the source code is intended to behave exactly the way it currently does and any errors are supposed to exist. We will not try to improve on how Paparazzi operates the aircraft.

3. We assume that no single event upsets occurs in the initialisation of the aircraft and focus only on single event upsets which happen after initialisation. The reasoning behind this and the previous assumption is that the scope of this report is to secure the Paparazzi autopilot system against single event upsets and not to improve the efficiency of the Paparazzi autopilot implementation.

4. We assume that any errors in the radio waves are handled by the underlying systems. This assumption is an extension to assumption 2.

5. No real CPU requirements are presented for Paparazzi, but we assume the extra processing power and memory needed to make the changes necessary for application-level correctness is available. This assumption is based on the hardware specifications of Paparazzi [17]. Paparazzi is able to run on a system with 60 MHz clock speed and 32 kB RAM, but other available chips have 168 MHz clock speed and 1024 kB RAM. So even if we more than double the requirements on the processing power, we only limit the number of microcontrollers capable of running Paparazzi.

6. We assume the single event upsets only occur in the main memory. This will exclude errors affecting the program counter and values saved in registers as seen in Figure 5.1. Since we are looking at idealised hardware, we assume that registers, program counter etc. are already secured against single event upsets. The single event upsets will be modelled in `C` by flipping a single bit in a variable. All of the code will be compiled with the GCC flag `-O0` to disable compiler optimisations and code rearrangements to ensure that the code behaves as expected when translated into assembly.

In short we look at single event upsets and the cascading errors they can cause in the Paparazzi autopilot software.

## 5.3 Modelling the System in UPPAAL

Before implementing application-level correctness into the Paparazzi system, we must know which variables are critical for the system. To find the critical functions and variables, we have created a model in the verification tool UPPAAL [2] for two functions in the main autopilot loop (Listing 4.1 on page 26). The model can be compared to a control flow graph due to them having the same structure and giving the same information of the system.
An observation about the autopilot is that it consists of one loop, which is the main loop controlling the order of operations. Other loops are smaller predetermined repeats e.g. a for loop running three times, calling the same function, but with different input.
The two functions `sensors_task()` and `navigation_task()` have been chosen because they are the most important functions to control the aircraft. Another reason is that the functions are being executed at different frequencies; `sensors_task()` runs at 60 Hz while `navigation_task()` runs at 4 Hz. If every function had been modelled, the UPPAAL model would have been more precise and given better insight on which variables were read the most before getting rewritten. For simplicity and due to time constraints, only these two functions will be modelled.

We suggest that if a variable is read often without getting rewritten, the variable has a larger vulnerability to single event upsets:

**Hypothesis 1.** The number of times a variable is read before getting rewritten, indicates the impact of the variable on the system.

We believe this hypothesis holds, because if an error occurs right after a variable has been written, the error will affect other parts of the system each time this variable is being read.

The model has been manually created in UPPAAL which makes it possible for us to count the frequency of variables being used. The final model consists of 214 templates and 904 locations. We modelled a total of one second of a flight resulting in `sensors_task()` being run 60 times and `navigation_task()` being run 4 times. This could be increased, but was sufficient for our needs.
Every function called by going through the two chosen functions is implemented in its own template. The `main_ap` function template can be seen in Figure 5.2. As it can be seen in the template, each function is called by the main function and after calling, it waits for the called function to signal

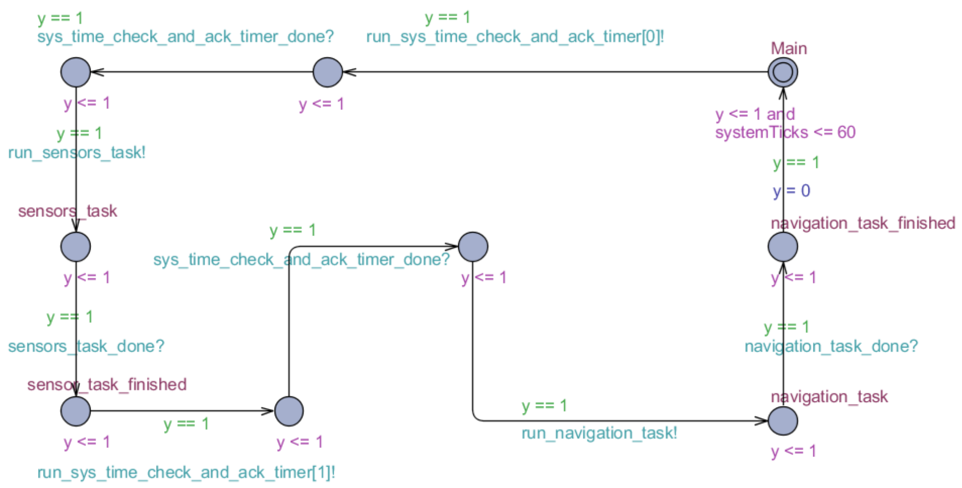


Figure 5.2: UPPAAL template of the main loop in Paparazzi

that it has finished execution. This ensures that only a single function is executing at a time, thus keeping the strict order from the original code. Note that each function keeps track of whether they are allowed to run or not.
Each global variable read or written during execution of these functions has been modelled in their own template, using the same basic structure for each variable template. An example of the variable template construct can be seen in Figure 5.3. Along with the template, UPPAAL code was made to keep track of how many times each variable has been written and read during a run. Most notably, the code keeps track of the maximum number of times each variable has been read before getting rewritten. This information is saved in the variable `maxreads`. This code can be seen in Listing 5.1.

Our approach aims to identify variables which are read often before getting rewritten, meaning that a single event upset could affect the system several times before the variable is corrected.
By having the usage frequency for all variables and functions, we can determine how important they are for implementing application-level correctness in Paparazzi.

## Limitations

Certain limitations exist in the model. First and foremost; the model contains some inaccuracy. Only some of the subsystems have been modelled. Consequently some variables can seem less important due to them not being used representatively in the modelled subsystems compared to the subsystems which are not modelled. This can result in variables seemingly being

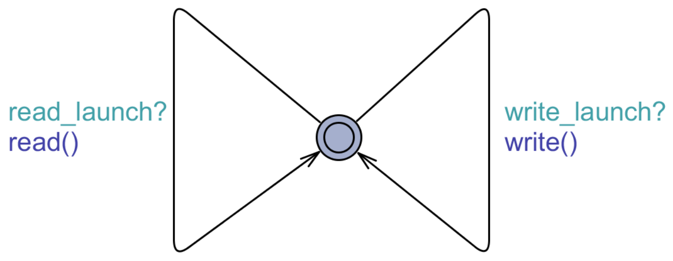


Figure 5.3: Template for the variable launch

```
int reads = 0;
int writes = 0;
int maxreads = 0;

void read() {
  ++reads;
  if(reads > maxreads) {
    maxreads = reads;
  }
}

void write() {
  ++writes;
  reads = 0;
}
```

Listing 5.1: UPPAAL variable code

|  |  |
|---|---|
| # of variables | 153 |
| # of variables unread | 41 |
| # of variables unwritten | 78 |
| # of variables with a `maxreads` value above 1 on average | 42 |
| # of variables with a `maxreads` value above 2 on average | 24 |
| # of variables with a `maxreads` value above 5 on average | 6 |

Table 5.1: Key numbers from running the model

read without ever being written to or vice versa.
Another limitation is that the model is an over-approximation of the real system. This is because there exists some execution paths through the UPPAAL model which does not exist in the real Paparazzi implementation.

### 5.3.1 Results

We have computed 30 random traces through our model and observed each variable, calculating the average `maxreads` of the different variables. We compute the average to give an indication of the variables' importance during an average flight.
We are interested in the number of times a variable is read before it is overwritten. This can be used to signify the importance of the variable's value being correct. A variable with a `maxreads` value of 1, will only be read once before overwritten, therefore an error will only affect the system once. However if a variable with a `maxreads` value of 60 will potentially affect the system for the duration of the flight.
In theory the best option would be to use the build-in query language to calculate the average over all possible paths through the model, but this is not feasible due to the size of the model.
Table 5.1 shows key numbers from the traces. The complete list of results can be seen in Appendix C. Due to the limitations stated in previous section, variables with a `maxreads` value of 0 or 60 should be disregarded, because they are most likely used in parts of the system which have not been modelled in this report.

## 5.4 Fidelity Metric

When the Paparazzi autopilot has been modified according to the results from the UPPAAL model, we need to compare the new autopilot with the original. To accomplish this we must define a fidelity metric. The fidelity metric is used to compare flight paths and return a numeric value representing the degree of deviation between the expected and the given path.

### 5.4.1 Comparing Paths

Each flight is recorded as a finite set of points in a three dimensional Cartesian coordinate system. A finite set of points corresponds to a flight path. We want to find a function that, given two finite sets of points, will return a value signifying how similar the two paths are. To achieve this we use the *Hausdorff distance* also known as the *Hausdorff metric*. A definition of the Hausdorff distance can be seen in Equation (5.1).

The Hausdorff distance, given two finite sets of points $A$ and $B$, calculates the shortest distance from any point in $A$ to any point in $B$, resulting $r_1$. Then the calculations are done from $B$ to $A$, resulting $r_2$. In the end the largest of the results $r_1$ and $r_2$ is returned. In Figure 5.4 an example of the two candidates is shown.

$$h(A,B) = \max\{\max_{a\in A}\min_{b\in B} d(a,b), \max_{b\in B}\min_{a\in A} d(a,b)\} \tag{5.1}$$

$$d(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2} \tag{5.2}$$

**Algorithm 1:** Hausdorff distance from [18]

```
Data: Two set of points A and B
Result: The longest minimum distance between the two sets of
        points
1  h = 0;
2  for every point $a_i$ in A do
3      shortest = $\infty$;
4      for every point $b_i$ in B do
5          $d_{ij} = d(a_i, b_j)$;
6          if $d_{ij}$ < shortest then
7              shortest = $d_{ij}$;
8          end
9      end
10     if shortest > h then
11         h = shortest;
12     end
13 end
14 return h;
```

We use the Hausdorff distance as our fidelity metric, because if two flights paths are similar, we consider them identical. This implies that several smaller deviations would be treated as insignificant, were as one large deviation will have a large impact on the perceived flight.

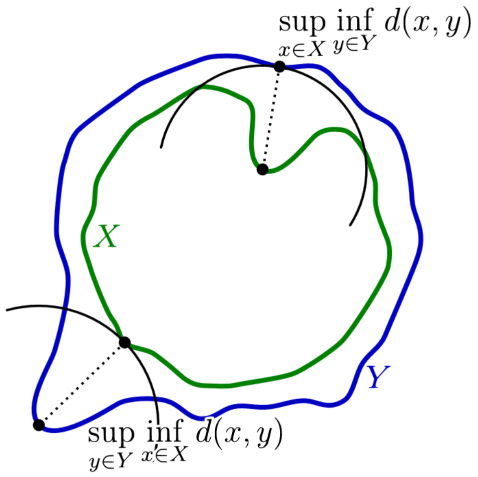

$\sup_{x \in X} \inf_{y \in Y} d(x,y)$
$X$
$Y$
$\sup_{y \in Y} \inf_{x \in X} d(x,y)$

Figure 5.4: Example of Hausdorff distance. Source: Wikipedia [15]

Hausdorff's algorithm can be seen in Algorithm 1. Note that the algorithm only shows the computations of the first pass and must be called with both the arguments $A, B$ and $B, A$ and select the maximum.

# Chapter 6

# Implementation & Test

In this chapter, we will present the implementation for applying application-level correctness into the Paparazzi autopilot. Also our test setup and the results from testing the original and modified Paparazzi code will be presented.

## 6.1 Securing the System

We split the variables from the Paparazzi autopilot code into two types, based on statistics from the UPPAAL model:

1. Non-secured variables - When maxread is $[0, 2]$
2. Secured variables - When maxread is $]2, \infty]$

The exact bound set between the secured and non-secured variables can be modified to affect the correctness of the execution. A lower bound will result in increased execution time, but also an increase in correctness of execution. A higher bound will result in decreased execution time, but also a decrease in correctness of execution. If bound is set to 0, the execution would in theory provide numerical correctness. Note that there is a possibility that some variables are rarely read, but have a significant impact on the system. These variables would most likely not be secured.

**Non-secured variables** Variables that are determined by the UPPAAL model to have little effect on the system. This would be variables only read in rare cases or variables often overwritten.
The code for these variables and their uses are unchanged compared to the original implementation of the Paparazzi autopilot.

**Secured variables** Variables that are determined by the UPPAAL model to have a large effect on the system. This would be variables that affect several other variables or are read several times before getting overwritten. The code for these variables are modified in a similar fashion to SWIFT-R [4]: The variable is triplicated when written. Before a secured variable is read, a majority vote is used to determine the correct value of the variable. When a secured variable is written, it is written to all three instances.

### 6.1.1 Alternatives

A third type of security could be introduced to the code. However this approach requires extensive knowledge about the system to guarantee correct execution. The code for variables with a medium impact on the system are duplicated. Before a variable is read, the system checks whether both variables are identical or not. If they are different, the function, or part of the function, is skipped until the point where the value would be corrected. This approach is similar to Rebaudengo et al. [14].
Another alternative would also to change the method used to analyse the importance of variables, e.g. utilising a code dependency graph.

## 6.2 Implementing Application-Level Correctness

As mentioned in the previous section, we limit our implementation of fault tolerance to only consider variables that on average were, at some point during execution, read more than two times before getting overwritten throughout simulating the model. Securing the variables is done with a majority vote function, which can be seen in Listing 6.1. A majority vote function was made for each data type, however these functions are identical in implementation. Note that the majority functions receives pointers securing that the variables are corrected in memory.
In Listing 6.2 a segment of the secured code can be seen, specifically the `stateCalcPositionUtm_f` function which is used during `navigation_task`. In this example, only the variable `state.pos_status` has a `maxreads` value above 2. This variable has been triplicated by introducing two new variables; `status_pos_status2` and `status_pos_status3`. In line 2 and 7 in Listing 6.2, a majority vote is cast for `state.pos_status`, securing that the variables agree on the value at this point of execution.
In line 15, 16, and 17 an example of the other part of the application-level correctness implementation can be seen. When the value of `state.pos_status` is set, the same value is saved in the copies of the variable. Note that `state_pos_status2` and `state_pos_status3` are not explicitly set to `state.pos_status`, but actually requires a recalculation for each variable, securing that if `state.pos_status` is corrupted right after it has been given its value, it will not corrupt the other two variables.

```c
void majority_vote_f(float *v1, float *v2, float *v3) {
  if(*v1 == *v2 && *v2 == *v3)
    return;

  if(*v1 == *v2){
    *v3 = *v1;
    return;
  }

  if(*v1 == *v3){
    *v2 = *v3;
    return;
  }

  if(*v2 == *v3){
    *v1 = *v2;
    return;
  }
}
```

Listing 6.1: Majority vote float function

An important note is that the `sensors_task` has not been modified since all variables read and written in the function, has a `maxreads` of less than or equal to 2.

### 6.2.1 Limitations

Our implementation of this application-level correctness has its advantages and drawbacks. As mentioned earlier we wanted to secure the Paparazzi system with little or no performance overhead. By only using the majority vote function on variables read more than 2 times, we are decreasing the amount of overhead compared to full numerical correctness.
A drawback of using this majority vote function before variables are being read, is that a single event upset can happen in the majority vote function or just after, which will result in Paparazzi reading an affected variable. This could be fixed by having hardware support for the majority vote in the CPU.

We have chosen to use the majority vote function before, and not after, a variable is read. This is because if the majority vote was used right after, the variable would already have affected the system if a single event upset had been present.

```c
void stateCalcPositionUtm_f(void) {
  majority_vote_uint16_t(&state.pos_status, &state_pos_status2, &↩
      state_pos_status3);

  if (bit_is_set(state.pos_status, POS_UTM_F))
    return;

  majority_vote_uint16_t(&state.pos_status, &state_pos_status2, &↩
      state_pos_status3);

  if (bit_is_set(state.pos_status, POS_LLA_F)) {
    utm_of_lla_f(&state.utm_pos_f, &state.lla_pos_f);
  }
  else if (bit_is_set(state.pos_status, POS_LLA_I)) {
    /* transform lla_i -> lla_f -> utm_f, set status bits */
    LLA_FLOAT_OF_BFP(state.lla_pos_f, state.lla_pos_i);
    SetBit(state.pos_status, POS_LLA_F);
    SetBit(state_pos_status2, POS_LLA_F);
    SetBit(state_pos_status3, POS_LLA_F);
    utm_of_lla_f(&state.utm_pos_f, &state.lla_pos_f);
  }
...
```

Listing 6.2: Code snippet of a secured function in Paparazzi

## 6.3 Testing

In our tests we run the modified fixedwing aircraft in the Paparazzi simulator. We use FlightGear as a visualiser and as the aircraft's black box. Each test is run 10 times unless otherwise mentioned and the results are then used to calculate the divergence from the expected path. The flight plan is plotted in Figure 4.4 on page 22. Note that Paparazzi is not designed to fly to and touch each corner point, but to fly within a fixed distance of the point.

First a baseline is presented and afterwards, the results from injecting a single event upset into Paparazzi will be presented.

The variables chosen for testing are:

- `pprz_mode` - This variable keeps track of which flight mode the autopilot is in, ranging from manual to complete autopilot. See Section 4.3.2 for a complete list of modes.
  This variable has been secured. It has been chosen due to its high influence on the aircraft's behaviour. Which could be observed by the high `maxreads` average value of 5.2.
- `state.utm_pos_f->alt` - This variable keeps track of the altitude of its current position.
  This variable is not secured due to its `maxreads` average value being

exactly 2.0. This variable has been chosen due to it being the variable with the highest `maxreads` value without being secured.

For each variable, a single event upset will be injected into each bit of the variable. This is to make sure that the test is not run on a bit which has no or abnormally large significance compared to the other bits. The injection occurs at the $2500^{th}$ iteration of the main autopilot loop. Due to simulated effects, such as wind, the injection does not occur at the exact same spot on the flight path.

### 6.3.1 Base Flight Path

For using the Hausdorff distance, a flight path is needed for comparison. The flight path which is defined in the GCS is not the ideal path to compare against for our purposes, since it does not represent the real flight behaviour. An artificial flight path was created based on the average of 1000 runs of the original unmodified Paparazzi code. This artificial flight path is used for comparison throughout the rest of this chapter.

### 6.3.2 Tests Without Single Event Upsets

In each version of the Paparazzi code, the tests were run 1000 times. The bars in Figure 6.1 show the average divergence during all flights calculated using the Hausdorff distance. The error bars show the maximum and minimum divergence from the flight path. The difference between the two versions of the code are 0.5 metres in favour of the modified version. This small distance is believed to be due to outside factors such as wind strength and direction in the simulator.

The Hausdorff distances gathered from testing can be seen in Appendix D.

### 6.3.3 The `pprz_mode` Variable

In Figure 6.2 the difference in Hausdorff distance between the original and modified Paparazzi code with a single event upset present in the `pprz_mode` variable is shown. For this test, 10 runs were made for each bit in the unsigned 8-bit integer `pprz_mode` in each version of the code.

In the original Paparazzi autopilot code we observed that the aircraft's behaviour changed drastically when injecting a single event upset. When bit number 0 or 1 was flipped, the aircraft flew straight back to `HOME` and circled around. When bit number 2 to 7 was flipped, the aircraft continued to fly in its current direction. This mostly occurred during a turn, causing the aircraft to circle around. In some cases the aircraft was already finished turning and just flew straight, ignoring the flight plan. This behaviour is

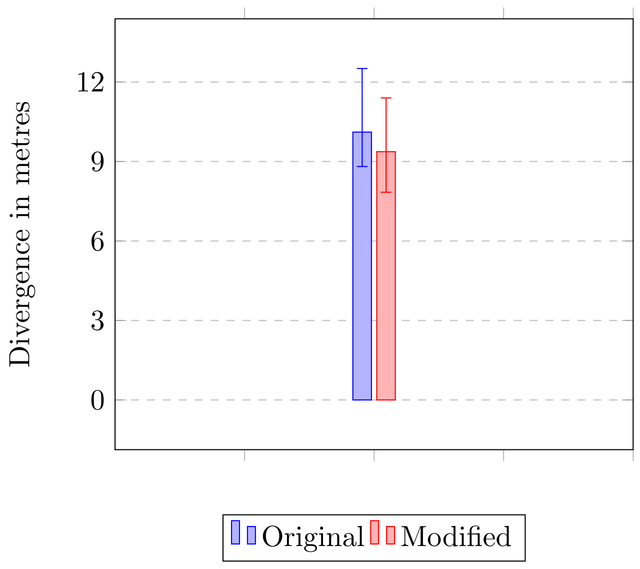


Figure 6.1: Hausdorff distances without a single event upset present

what is causing the big outliers compared to the average. Note that in bit number 2 to 6, only a single data point is far away from the average, where in bit number 7, there are two which are a significant distance from the average. Figure 6.3 shows each individual data point for bit number 7.

The modified code behaved as the user would expect. All average Hausdorff distances are in between the two tests without single event upsets. From the user's perspective, an error did not occur during the flight and the aircraft behaved within the boundaries of what would be considered normal.
For a more in depth view of the modified code's Hausdorff distances, see Figure 6.4.

The Hausdorff distances gathered from testing can be seen in Appendix D.

### 6.3.4 The `state.utm_pos_f->alt` Variable

The Hausdorff distances from testing the `state.utm_pos_f->alt` variable can be seen in Figure 6.5 and Figure 6.6. For this test, single event upsets where injected in each bit of the 32-bit float.

From the results, one can see that the single event upset does not affect the aircraft significantly. This is due to it being overwritten often by the sensors, meaning the single event upset does not affect the aircraft's flight path for more than a fraction of a second. This also implies that the variable

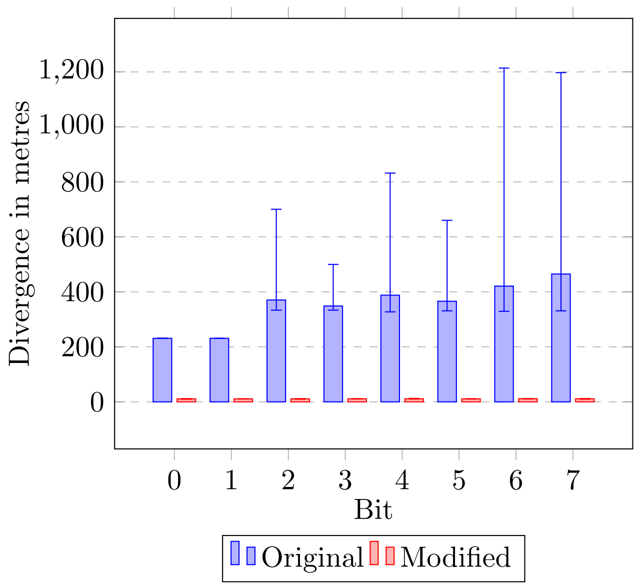


Figure 6.2: Hausdorff distances with a single event upset on the `pprz_mode` variable

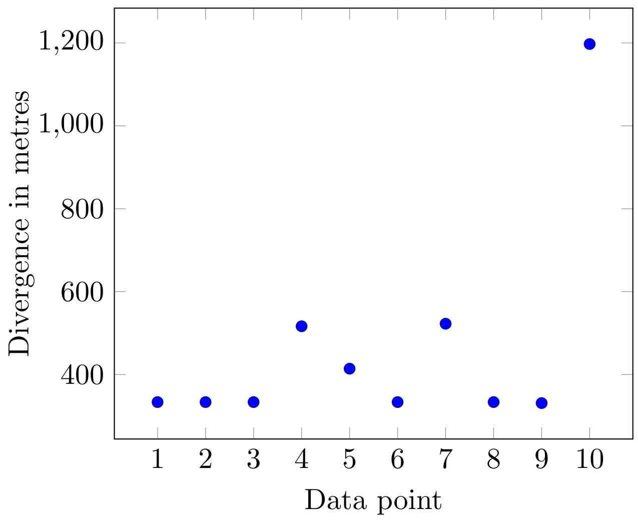


Figure 6.3: Hausdorff distances for the original code with a single event upset in the `pprz_mode` variable on bit number 7

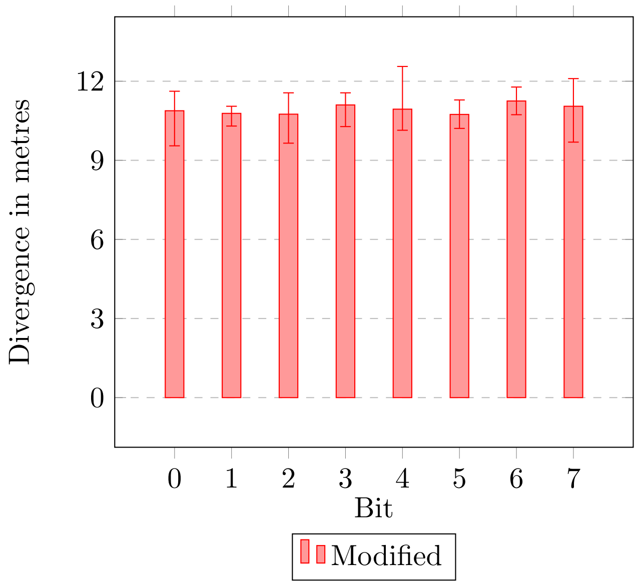


Figure 6.4: Hausdorff distances on the modified code with a single event upset in the `pprz_mode` variable

is less significant with respect to application-level correctness.

The Hausdorff distances gathered from testing can be seen in Appendix D.

### 6.3.5 Other Variables

To test Hypothesis 1, additional tests were performed. Each test was run five times and an average of all tests on each variable was calculated.
First we ran tests on seven variables, which have an average `maxreads` in between the averages of the variables `state.utm_pos_f->alt` and `pprz_mode`. The results are shown in Figure D.1 on page 67. The tests showed that two out of the seven had a major impact on the behaviour of the aircraft, while one other had a small impact on the behaviour.

Secondly we ran tests on seven variables, which have an average `maxreads` below 1.0. The first five variables have the highest average `maxreads` just below 1.0. The last two have an average `maxreads` closer to 0. The reason these variables have been chosen is to give a broad perspective on the impact of variables with different average `maxreads`. The results are shown in Figure D.2 on page 67. Only a single variable, `gps_lost`, had an impact on the perceived flight path.

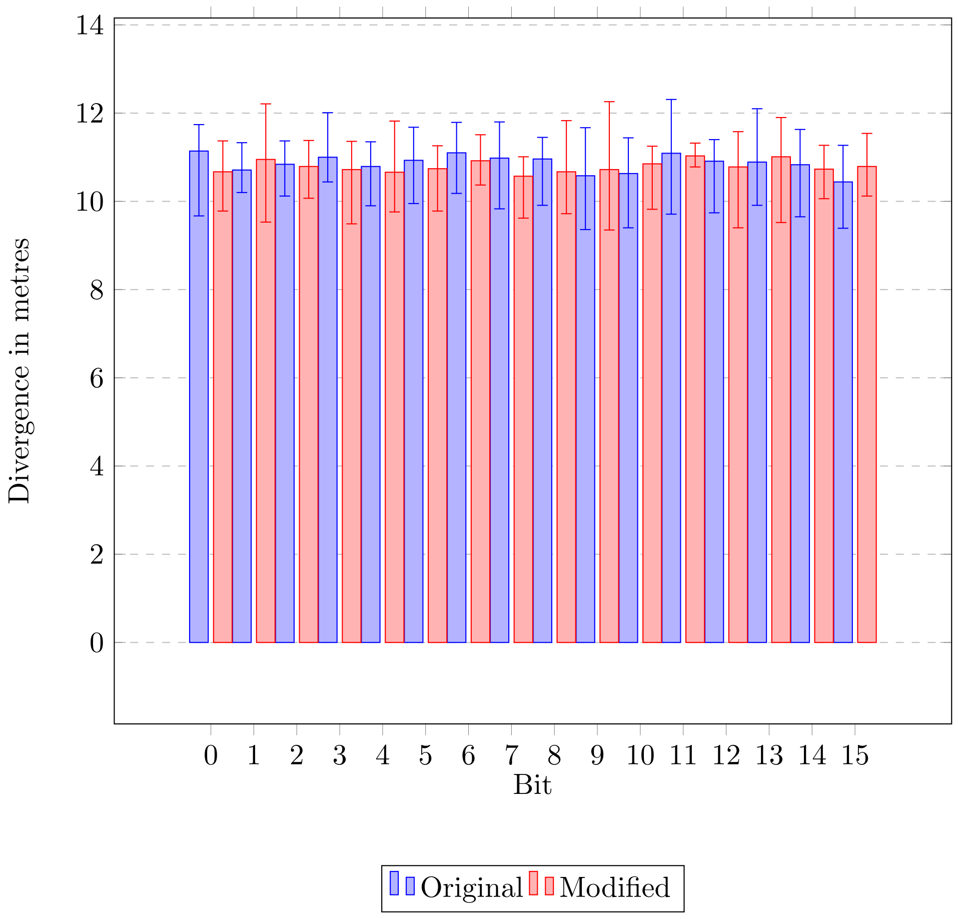


Figure 6.5: Hausdorff distances with a single event upset in the `state.utm_pos_f->alt` variable part 1

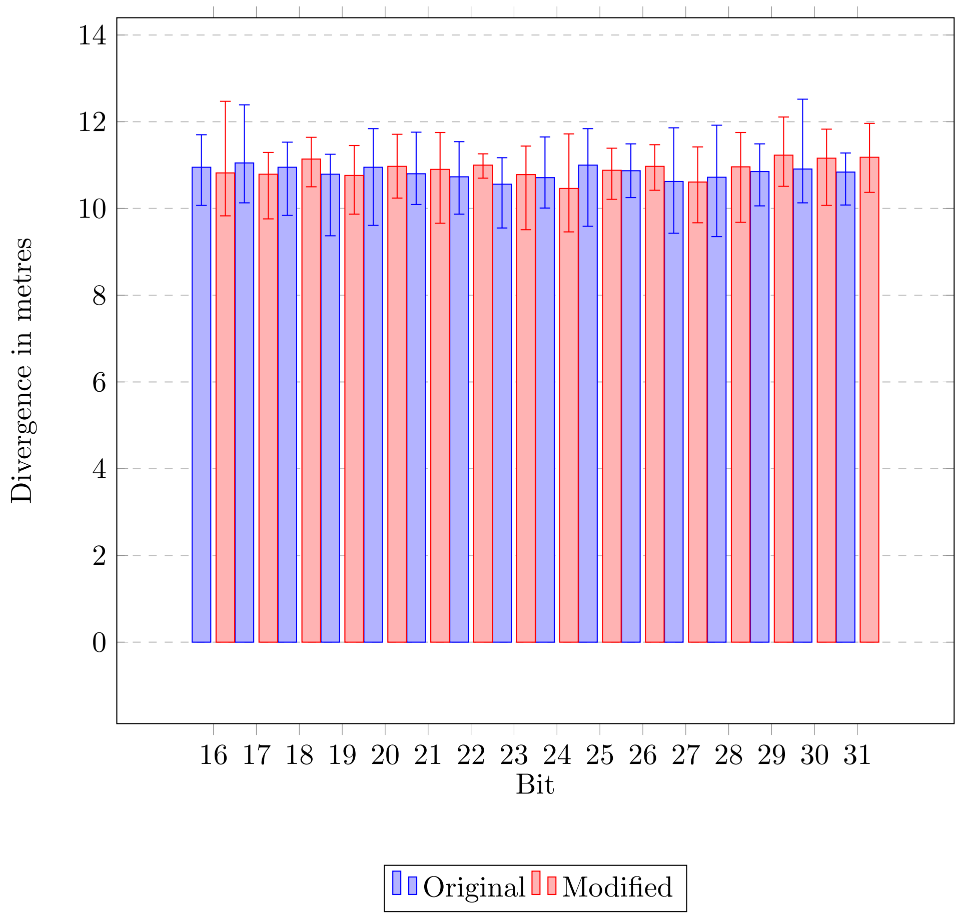


Figure 6.6: Hausdorff distances with a single event upset in the `state.utm_pos_f->alt` variable part 2

|  | Average |  |  | Overhead |  |  |
|---|---|---|---|---|---|---|
|  | `st` | `nt` | `total` | `st` | `nt` | `total` |
| Original code | $1\mu s$ | $4\mu s$ | $19\mu s$ | 0% | 0% | 0% |
| Modified code | $1\mu s$ | $5\mu s$ | $20\mu s$ | 0% | 25% | 5% |

Table 6.1: Results of performance test. The negative overhead of `st` has been disregarded for the total overhead calculations due to the code of the `st`-function being identical in both versions

### 6.3.6 Overhead

In this section we present the amount of overhead of our application-level correctness implementation into Paparazzi. These results have been conducted through a test of 100 runs on both the original and the modified Paparazzi code.
As shown in Table 6.1, the results of our performance tests show that the modified `navigation_task` (`nt`) function has a 25% overhead compared to the original function. The `sensors_task` (`st`) has an overhead of 0% compared to the original function, which is due to the two versions of `sensors_task` being identical. When accounting for `sensors_task` running at 60 Hz and `navigation_task` running at 4 Hz, the total amount of overhead is 5%.

## 6.4 Summary

The results of the tests show that some variables are not to be considered critical with regards to application-level correctness. This can be seen from the results regarding `state.utm_pos_f->alt` where all tests showed, that a single event upset did not impact the overall flight noticeably.
The tests also showed that the opposite case is true. Explicitly this was true with regards to `pprz_mode`, which showed that when a single event upset was injected into the variable, it had a large impact on the perceived flight path.
All tests regarding the modified code, showed that the aircraft behaved within the boundaries a user would expect. When a single event upset was injected, the aircraft identified the error and recovered successfully.
Our testing of other variables in Section 6.3.5 indicated that Hypothesis 1 is true. Results showed that variables with a higher average `maxreads` value are more likely to have a larger impact on the perceived flight path. Note that the only variable with a low average `maxreads` value, which impacted the system, is a boolean and therefore consists of only one bit.

The overall overhead calculations showed that the modified code has a 7.08% overhead. However, this number might be artificially low, because

`navigation_task`, which was mainly responsible for the overhead, only runs at 4 Hz, while `sensors_task` runs at 60 Hz. If the entire system had been modelled, it is likely that the overhead would be significantly larger, since most of the other tasks runs at 60 Hz or 15 Hz.
Even if the entire system was modelled and if `navigation_task` is representative, the total overhead might be close to the overhead of `navigation_task` which was 27.14%. This amount of overhead is acceptable. Note however that changing the bound of which variables to secure, the overhead might change accordingly.

# Chapter 7

# Conclusion

In this report, we introduced a new approach to application-level correctness, inspired by previously published methods, designed for the Paparazzi autopilot software. We have also introduced a method to compare the similarity of flight paths.

We have studied the Paparazzi autopilot source code and discovered that the Paparazzi source code is prone to single event upsets. By using application-level correctness, it is possible to secure the system enough for it to behave correctly from the user's perspective, without a full n-modular replication. This approach has shown that a solution securing part of the code was possible without a large increase in overhead.

We were unable to locate previous work regarding the comparison of flight paths. We discovered that the Hausdorff distance algorithm satisfies our needs for finding the largest divergence between two flight paths and served its purpose as a fidelity metric. However the Hausdorff distance algorithm requires the user to create an artificial base flight path.

We performed several tests on both our modified version of the Paparazzi autopilot software and the original version. The tests showed, by using the Hausdorff distance, the original Paparazzi autopilot software is prone to single event upsets in some variables. The tests also showed our approach was able to correct the single event upsets and complete each flight according to the flight plan.

According to Hypothesis 1, we suggested that a variable which is read often before getting rewritten had a larger impact on the system. The results of the tests show that the amount of times a variable is read does give an indication of the significance of the variable. However, some variables are read often, but does not affect the system from a user's perspective. Likewise it is

possible that variables that seem insignificant, actually have a large impact on the system if altered.
The UPPAAL model served well as a way to analyse the significance of the variables. However due to the model being an overapproximation of the Paparazzi autopilot, the importance of some variables may be artificially high. A more precise indication could have been achieved by utilising a code dependency graph from analysing the `C` code of the Paparazzi autopilot.

With respect to our definition of application-level correctness (Definition 2.1 on page 11), application-level correctness has been achieved, using the approach introduced in this report.

# Bibliography

[1] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats - a taxonomy. In *IFIP Congress Topical Sessions*, pages 91–120, 2004.

[2] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *DIMACS/SYCON Workshop on Hybrid Systems III*, pages 232–243, 1995.

[3] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Object Oriented Programming Systems Languages & applications*, OOPSLA '13, pages 33–52, 2013.

[4] J. Chang, G.A. Reis, and D.I. August. Automatic instruction-level software-only recovery. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference*, pages 83–92, 2006.

[5] Jason Cong and Karthik Gururaj. Assuring application-level correctness against soft errors. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference*, pages 150–157, Nov 2011.

[6] David Conger. RT Paparazzi. `http://wiki.paparazziuav.org/wiki/RT_Paparazzi`, January 2014. [Online; accessed 16-April-2014].

[7] NASA Glenn Research Center. Aircraft rotation. `http://www.grc.nasa.gov/WWW/k-12/airplane/Images/rotations.gif`, 2013. [Online; accessed 30-September-2013].

[8] Morten Turn Pedersen Heine Gatten Larsen and Thomas Birch Mogensen. FauToPilot. `www.turn-pedersen.dk/projects/d902e13.pdf`, January 2014. [Online; accessed 20-March-2014].

[9] Theresa Knott. Angle of attack.svg. `http://upload.wikimedia.org/wikipedia/commons/6/6d/Angle_of_attack.svg`, 2003. [Online; accessed 16-December-2013].

[10] Xuanhua Li and Donald Yeung. Application-level correctness and its impact on fault tolerance. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium*, pages 181–192, Feb 2007.

[11] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. *Worst-Case Execution Time*, 4, 2006.

[12] E. Normand. Single-event effects in avionics. *Nuclear Science, IEEE Transactions*, 43(2):461–474, Apr 1996.

[13] Michel Gorraz-Pierre-Selim Huard Pascal Brisset, Antoine Drouin and Jeremy Tyler. The Paparazzi solution. *Micro Aerial Vehicle*, 2006.

[14] Maurizio Rebaudengo, Matteo Sonza Reorda, Marco Torchiano, and Massimo Violante. Soft-error detection through software fault-tolerance techniques. In *Defect and Fault Tolerance in VLSI Systems, 1999. DFT'99. International Symposium*, pages 210–218. IEEE, 1999.

[15] Claudio Rocchini. Example of Hausdorff distance, October 2007. `http://en.wikipedia.org/wiki/File:Hausdorff_distance_sample.svg`.

[16] Felix Ruess. Devguide/designoverview, January 2012. `http://wiki.paparazziuav.org/wiki/DevGuide/DesignOverview`.

[17] Piotr Esden-Tempski Stephen Dwyer, Felix Ruess and Sergey Krukowski. Category:autopilots. `http://wiki.paparazziuav.org/wiki/Autopilots`, August 2013. [Online; accessed 25-March-2014].

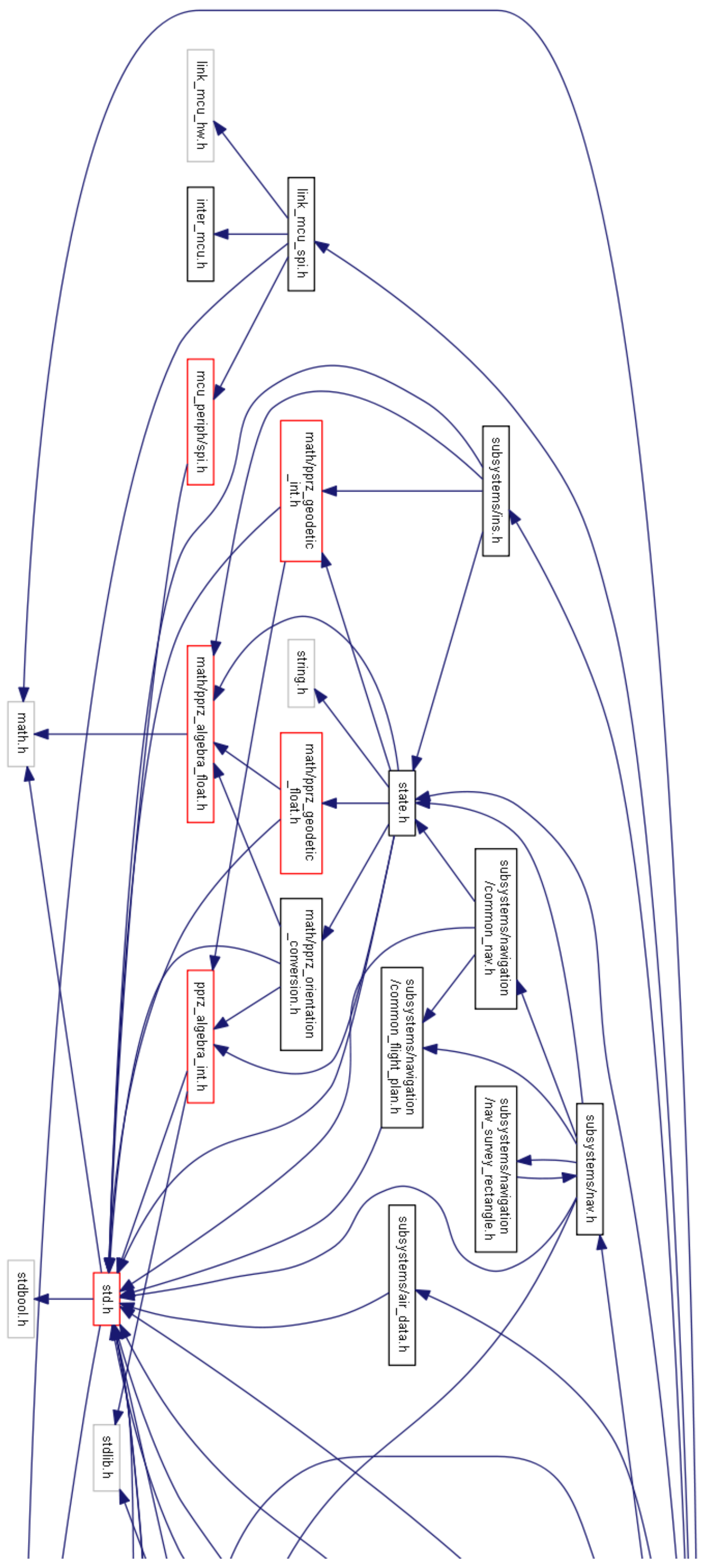
[18] Godfried T. Toussaint. Hausdorff distance between convex polygons. `http://cgm.cs.mcgill.ca/~godfried/teaching/cg-projects/98/normand/main.html`, 1998. [Online; accessed 8-May-2014].

[19] YC Yeh. Triple-triple redundant 777 primary flight computer. In *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, volume 1, pages 293–307. IEEE, 1996.

# Appendix A

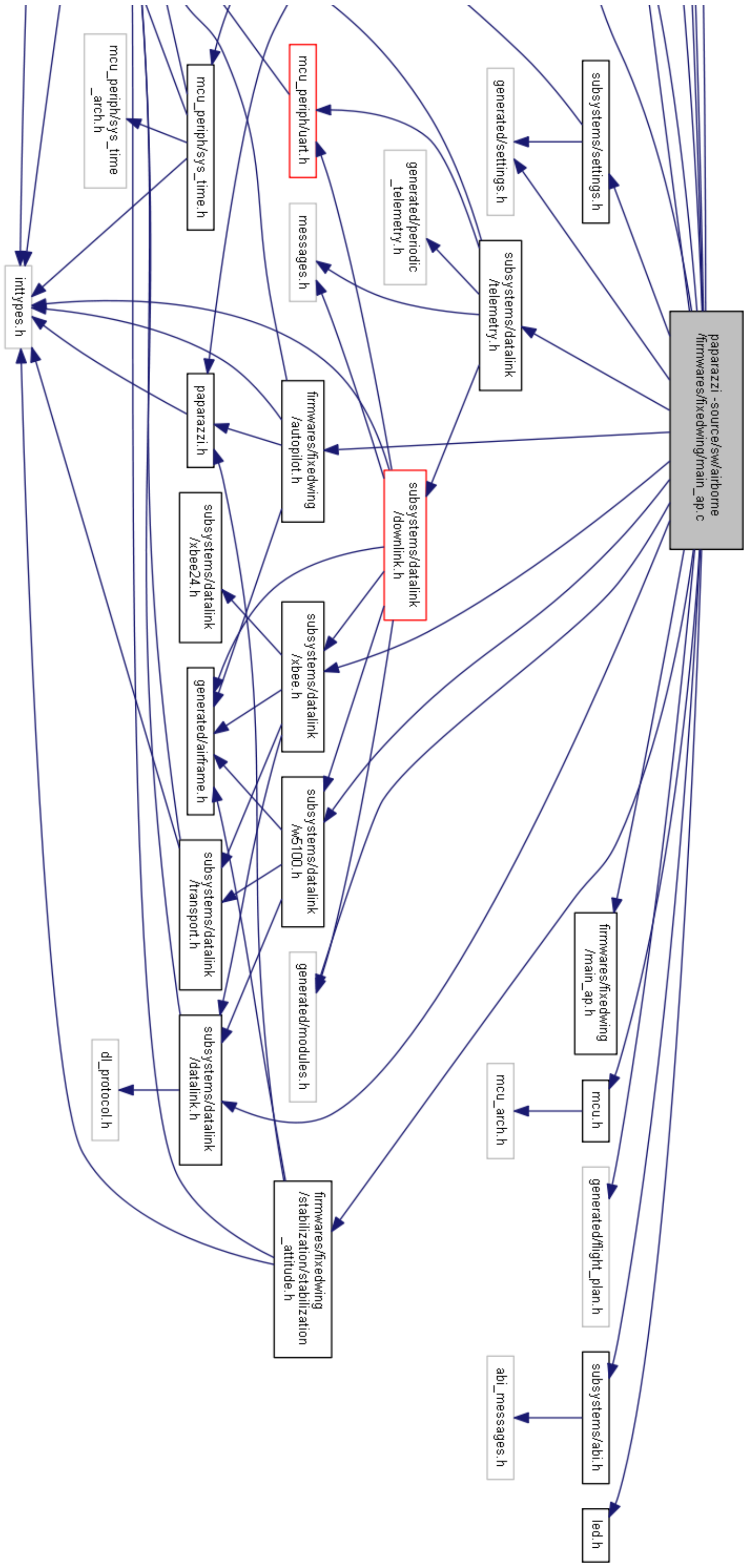
# Dependencies

The next two pages contain the dependency graph of the main_ap.c source file for the Paparazzi Project's fixedwing autopilot.



link_mcu_hw.h
inter_mcu.h
link_mcu_spi.h
mcu_periph/spi.h
math/pprz_geodetic_int.h
subsystems/ins.h
string.h
math.h
math/pprz_algebra_float.h
math/pprz_geodetic_float.h
state.h
math/pprz_orientation_conversion.h
pprz_algebra_int.h
subsystems/navigation/common_nav.h
subsystems/navigation/common_flight_plan.h
subsystems/navigation/nav_survey_rectangle.h
subsystems/nav.h
subsystems/air_data.h
stdbool.h
std.h
stdlib.h

Figure A.1: First half of main_ap.c's dependency graph



paparazzi -source/sw/airborne /firmwares/fixedwing/main_ap.c
subsystems/settings.h
generated/settings.h
subsystems/datalink /telemetry.h
generated/periodic _telemetry.h
messages.h
mcu_periph/uart.h
subsystems/datalink /downlink.h
firmwares/fixedwing /autopilot.h
paparazzi.h
mcu_periph/sys_time.h
mcu_periph/sys_time _arch.h
inttypes.h
subsystems/datalink /xbee24.h
subsystems/datalink /xbee.h
generated/airframe.h
subsystems/datalink /w5100.h
subsystems/datalink /transport.h
subsystems/datalink /datalink.h
dl_protocol.h
generated/modules.h
firmwares/fixedwing /stabilization/stabilization _attitude.h
firmwares/fixedwing /main_ap.h
mcu.h
mcu_arch.h
generated/flight_plan.h
subsystems/abi.h
abi_messages.h
led.h

Figure A.2: Second half of main_ap.c’s dependency graph

# Appendix B

# Paparazzi Source Code

In the following appendix, the code of the main loop and its child functions can be seen.

```
void handle_periodic_tasks_ap(void) {

  if (sys_time_check_and_ack_timer(sensors_tid))
    sensors_task();

  if (sys_time_check_and_ack_timer(navigation_tid))
    navigation_task();

#ifndef AHRS_TRIGGERED_ATTITUDE_LOOP
  if (sys_time_check_and_ack_timer(attitude_tid))
    attitude_loop();
#endif

  if (sys_time_check_and_ack_timer(modules_tid))
    modules_periodic_task();

  if (sys_time_check_and_ack_timer(monitor_tid))
    monitor_task();

  if (sys_time_check_and_ack_timer(telemetry_tid)) {
    reporting_task();
    LED_PERIODIC();
  }
}
```

Listing B.1: Main loop

```c
void sensors_task( void ) {
#if USE_IMU
  imu_periodic();

#if USE_AHRS
  if (ahrs_timeout_counter < 255)
    ahrs_timeout_counter ++;
#endif // USE_AHRS
#endif // USE_IMU

  //FIXME: this is just a kludge
#if USE_AHRS && defined SITL && !USE_NPS
  ahrs_propagate();
#endif

#if USE_BARO_BOARD
  baro_periodic();
#endif

#if USE_GPS
  gps_periodic_check();
#endif

  ins_periodic();
}
```

Listing B.2: Sensors task

```c
void navigation_task( void ) {
#if defined FAILSAFE_DELAY_WITHOUT_GPS
  /** This section is used for the failsafe of GPS */
  static uint8_t last_pprz_mode;
  /** If aircraft is launched and is in autonomus mode, go into
      PPRZ_MODE_GPS_OUT_OF_ORDER mode (Failsafe) if we lost the GPS */
  if (launch) {
    if (GpsTimeoutError) {
      if (pprz_mode == PPRZ_MODE_AUTO2 || pprz_mode == PPRZ_MODE_HOME) {
        last_pprz_mode = pprz_mode;
        pprz_mode = PPRZ_MODE_GPS_OUT_OF_ORDER;
        autopilot_send_mode();
        gps_lost = TRUE;
      }
    } else if (gps_lost) { /* GPS is ok */
      /** If aircraft was in failsafe mode, come back in previous mode */
      pprz_mode = last_pprz_mode;
      gps_lost = FALSE;
      autopilot_send_mode();
    }
  }
#endif /* GPS && FAILSAFE_DELAY_WITHOUT_GPS */
  common_nav_periodic_task_4Hz();
  if (pprz_mode == PPRZ_MODE_HOME)
    nav_home();
  else if (pprz_mode == PPRZ_MODE_GPS_OUT_OF_ORDER)
    nav_without_gps();
  else
    nav_periodic_task();
#ifdef TCAS
  CallTCAS();
#endif
#ifndef PERIOD_NAVIGATION_Ap_0 // If not sent periodically (in default 0 mode)
  SEND_NAVIGATION(DefaultChannel, DefaultDevice);
#endif
  /* The nav task computes only nav_altitude. However, we are interested
     by desired_altitude (= nav_alt+alt_shift) in any case.
     So we always run the altitude control loop */
  if (v_ctl_mode == V_CTL_MODE_AUTO_ALT)
    v_ctl_altitude_loop();

  if (pprz_mode == PPRZ_MODE_AUTO2 || pprz_mode == PPRZ_MODE_HOME
           || pprz_mode == PPRZ_MODE_GPS_OUT_OF_ORDER) {
#ifdef H_CTL_RATE_LOOP
    /* Be sure to be in attitude mode, not roll */
    h_ctl_auto1_rate = FALSE;
#endif
    if (lateral_mode >=LATERAL_MODE_COURSE)
      h_ctl_course_loop(); /* aka compute nav_desired_roll */

    // climb_loop(); //4Hz
  }
}
```

Listing B.3: Navigation task

```c
void attitude_loop( void ) {

#if USE_INFRARED
  ahrs_update_infrared();
#endif /* USE_INFRARED */

  if (pprz_mode >= PPRZ_MODE_AUTO2)
  {
    if (v_ctl_mode == V_CTL_MODE_AUTO_THROTTLE) {
      v_ctl_throttle_setpoint = nav_throttle_setpoint;
      v_ctl_pitch_setpoint = nav_pitch;
    }
    else if (v_ctl_mode >= V_CTL_MODE_AUTO_CLIMB)
    {
      v_ctl_climb_loop();
    }

#if defined V_CTL_THROTTLE_IDLE
    Bound(v_ctl_throttle_setpoint, TRIM_PPRZ(V_CTL_THROTTLE_IDLE*MAX_PPRZ), MAX_PPRZ);
#endif

#ifdef V_CTL_POWER_CTL_BAT_NOMINAL
    if (vsupply > 0.) {
      v_ctl_throttle_setpoint *= 10. * V_CTL_POWER_CTL_BAT_NOMINAL / (float)vsupply;
      v_ctl_throttle_setpoint = TRIM_UPPRZ(v_ctl_throttle_setpoint);
    }
#endif

    h_ctl_pitch_setpoint = v_ctl_pitch_setpoint; // Copy the pitch setpoint from the guidance to the stabilization control
    Bound(h_ctl_pitch_setpoint, H_CTL_PITCH_MIN_SETPOINT, H_CTL_PITCH_MAX_SETPOINT);
    if (kill_throttle || (!autopilot_flight_time && !launch))
      v_ctl_throttle_setpoint = 0;
  }

  h_ctl_attitude_loop(); /* Set  h_ctl_aileron_setpoint & h_ctl_elevator_setpoint */
  v_ctl_throttle_slew();
  ap_state->commands[COMMAND_THROTTLE] = v_ctl_throttle_slewed;
  ap_state->commands[COMMAND_ROLL] = -h_ctl_aileron_setpoint;

  ap_state->commands[COMMAND_PITCH] = h_ctl_elevator_setpoint;

#if defined MCU_SPI_LINK || defined MCU_UART_LINK
  link_mcu_send();
#elif defined INTER_MCU && defined SINGLE_MCU
  /**Directly set the flag indicating to FBW that shared buffer is available*/
  inter_mcu_received_ap = TRUE;
#endif
}
```

Listing B.4: Attitude loop

```c
void monitor_task( void ) {
  if (autopilot_flight_time)
    autopilot_flight_time++;
#if defined DATALINK || defined SITL
  datalink_time++;
#endif

  static uint8_t t = 0;
  if (vsupply < CATASTROPHIC_BAT_LEVEL*10)
    t++;
  else
    t = 0;
  kill_throttle |= (t >= CATASTROPHIC_BAT_KILL_DELAY);
  kill_throttle |= launch && (dist2_to_home > Square(↩
      KILL_MODE_DISTANCE));

  if (!autopilot_flight_time &&
      *stateGetHorizontalSpeedNorm_f() > MIN_SPEED_FOR_TAKEOFF) {
    autopilot_flight_time = 1;
    launch = TRUE; /* Not set in non auto launch */
    uint16_t time_sec = sys_time.nb_sec;
    DOWNLINK_SEND_TAKEOFF(DefaultChannel, DefaultDevice, &time_sec);
  }
}
```

Listing B.5: Monitor task

```c
void reporting_task( void ) {
  static uint8_t boot = TRUE;

  /** initialisation phase during boot */
  if (boot) {
    DOWNLINK_SEND_BOOT(DefaultChannel, DefaultDevice, &version);
    boot = FALSE;
  }
  /** then report periodicly */
  else {
    //PeriodicSendAp(DefaultChannel, DefaultDevice);
    periodic_telemetry_send_Ap();
  }
}
```

Listing B.6: Reporting task

# Appendix C

# UPPAAL Trace Results

This appendix lists the `maxreads` average values in Listing C.1 by running the UPPAAL model 30 times.

```
var_block_time.maxreads = 0.0
var_desired_x.maxreads = 0.0
var_desired_y.maxreads = 0.0
var_dist2_to_wp.maxreads = 0.0
var_errno.maxreads = 0.0
var_gps_fix.maxreads = 0.0
var_horizontal_mode.maxreads = 0.0
var_in_d_alt.maxreads = 0.0
var_in_d_lat.maxreads = 0.0
var_in_d_lon.maxreads = 0.0
var_nav_circle_radians.maxreads = 0.0
var_nav_circle_radians_no_rewind.maxreads = 0.0
var_nav_circle_radius.maxreads = 0.0
var_nav_circle_x.maxreads = 0.0
var_nav_circle_y.maxreads = 0.0
var_nav_flight_altitude.maxreads = 0.0
var_nav_pitch.maxreads = 0.0
var_nav_survey_active.maxreads = 0.0
var_nav_throttle_setpoint.maxreads = 0.0
var_stage_time.maxreads = 0.0
var_state_body_rates_f_p.maxreads = 0.0
var_state_body_rates_f_q.maxreads = 0.0
var_state_body_rates_f_r.maxreads = 0.0
var_state_enu_pos_f_z.maxreads = 0.0
var_state_enu_pos_i_x.maxreads = 0.0
var_state_enu_pos_i_y.maxreads = 0.0
var_state_enu_pos_i_z.maxreads = 0.0
var_state_enu_speed_f.maxreads = 0.0
var_state_enu_speed_f_z.maxreads = 0.0
var_state_h_speed_dir_f.maxreads = 0.0
var_state_h_speed_norm_f.maxreads = 0.0
var_state_ned_pos_f_alt.maxreads = 0.0
var_state_ned_pos_f_east.maxreads = 0.0
var_state_ned_pos_f_north.maxreads = 0.0
var_state_ned_pos_i_x.maxreads = 0.0
var_state_ned_pos_i_y.maxreads = 0.0
var_state_ned_pos_i_z.maxreads = 0.0
var_state_ned_speed_f.maxreads = 0.0
```

```
var_state_ned_speed_f_z.maxreads = 0.0
var_state_rate_status.maxreads = 0.0
var_too_far_from_home.maxreads = 0.0
var_state_ecef_pos_i_x.maxreads = 0.03
var_state_ecef_pos_i_y.maxreads = 0.03
var_state_ecef_pos_i_z.maxreads = 0.03
var_state_ned_origin_i_x.maxreads = 0.03
var_state_ned_origin_i_y.maxreads = 0.03
var_state_ned_origin_i_z.maxreads = 0.03
var_delta2_z.maxreads = 0.07
var_state_enu_pos_f_alt.maxreads = 0.07
var_state_enu_pos_f_east.maxreads = 0.07
var_state_enu_pos_f_north.maxreads = 0.07
var_state_enu_speed_i_x.maxreads = 0.07
var_state_enu_speed_i_y.maxreads = 0.07
var_state_enu_speed_i_z.maxreads = 0.07
var_state_ned_speed_i_x.maxreads = 0.07
var_state_ned_speed_i_y.maxreads = 0.07
var_state_ned_speed_i_z.maxreads = 0.07
var_state_utm_origin_f_x.maxreads = 0.07
var_state_utm_origin_f_y.maxreads = 0.07
var_state_utm_origin_f_z.maxreads = 0.07
var_delta2_x.maxreads = 0.1
var_delta2_y.maxreads = 0.1
var_state_enu_speed_f_x.maxreads = 0.1
var_state_enu_speed_f_y.maxreads = 0.1
var_state_utm_origin_f_alt.maxreads = 0.1
var_state_utm_origin_f_east.maxreads = 0.1
var_state_utm_origin_f_north.maxreads = 0.1
var_state_utm_pos_f_east.maxreads = 0.1
var_state_utm_pos_f_north.maxreads = 0.1
var_ltp_of_ecef_m02.maxreads = 0.17
var_state_ecef_pos_f_x.maxreads = 0.17
var_state_ecef_pos_f_y.maxreads = 0.17
var_state_ecef_pos_f_z.maxreads = 0.17
var_state_lla_pos_i_alt.maxreads = 0.17
var_state_lla_pos_i_lat.maxreads = 0.17
var_state_lla_pos_i_lon.maxreads = 0.17
var_state_ned_origin_f_x.maxreads = 0.17
var_state_ned_origin_f_y.maxreads = 0.17
var_state_ned_origin_f_z.maxreads = 0.17
var_ltp_of_ecef_m00.maxreads = 0.2
var_ltp_of_ecef_m01.maxreads = 0.2
var_ltp_of_ecef_m10.maxreads = 0.2
var_ltp_of_ecef_m11.maxreads = 0.2
var_ltp_of_ecef_m12.maxreads = 0.2
var_ltp_of_ecef_m20.maxreads = 0.2
var_ltp_of_ecef_m21.maxreads = 0.2
var_ltp_of_ecef_m22.maxreads = 0.2
var_ecef_x.maxreads = 0.2
var_ecef_y.maxreads = 0.2
var_ecef_z.maxreads = 0.2
var_state_lla_pos_f_alt.maxreads = 0.2
var_delta_x.maxreads = 0.3
var_delta_y.maxreads = 0.3
var_delta_z.maxreads = 0.3
var_state_ned_speed_f_x.maxreads = 0.33
var_state_ned_speed_f_y.maxreads = 0.33
var_circle_bank.maxreads = 0.37
var_state_h_speed_dir_i.maxreads = 0.43
var_state_utm_pos_f_zone.maxreads = 0.53
var_state_lla_pos_f_lat.maxreads = 0.6
```

```
var_state_lla_pos_f_lon.maxreads = 0.6
var_v_ctl_auto_throttle_submode.maxreads = 0.6
var_nav_ratio.maxreads = 0.73
var_h_ctl_roll_setpoint.maxreads = 0.83
var_gps_lost.maxreads = 0.93
var_v_ctl_climb_setpoint.maxreads = 0.93
var_body_rate_p.maxreads = 1.0
var_body_rate_q.maxreads = 1.0
var_body_rate_r.maxreads = 1.0
var_lateral_mode.maxreads = 1.0
var_prescaler.maxreads = 1.0
var_state_utm_initialized_f.maxreads = 1.03
var_h_ctl_course_pre_bank.maxreads = 1.17
var_h_ctl_course_dgain.maxreads = 1.33
var_h_ctl_course_pre_bank_correction.maxreads = 1.33
var_h_ctl_roll_slew.maxreads = 1.33
var_speed_depend_nav.maxreads = 1.33
var_state_ned_pos_f_x.maxreads = 1.4
var_state_ned_pos_f_y.maxreads = 1.4
var_state_ned_pos_f_z.maxreads = 1.4
var_nav_in_circle.maxreads = 1.57
var_v_ctl_mode.maxreads = 1.6
var_v_ctl_altitude_pre_climb.maxreads = 1.63
var_v_ctl_climb_mode.maxreads = 1.63
var_h_ctl_course_setpoint.maxreads = 1.7
var_last_pprz_mode.maxreads = 1.83
var_dist2_to_home.maxreads = 1.97
var_v_ctl_altitude_error.maxreads = 1.97
var_state_utm_pos_f_alt.maxreads = 2.0
var_state_ned_initialized.maxreads = 2.1
var_altitude_pgain_boost.maxreads = 2.17
var_v_ctl_altitude_max_climb.maxreads = 2.17
var_v_ctl_altitude_pgain.maxreads = 2.17
var_v_ctl_altitude_pre_climb_correction.maxreads = 2.17
var_v_ctl_altitude_setpoint.maxreads = 2.17
var_last_nav_alt.maxreads = 2.17
var_ground_alt.maxreads = 2.2
var_h_ctl_course_pgain.maxreads = 2.87
var_nav_altitude.maxreads = 2.87
var_h_ctl_roll_max_setpoint.maxreads = 3.1
var_nav_mode.maxreads = 3.1
var_waypoints_WP_HOME_x.maxreads = 3.1
var_waypoints_WP_HOME_y.maxreads = 3.1
var_nav_circle_trigo_qdr.maxreads = 3.3
var_GpsTimeoutError.maxreads = 4.0
var_launch.maxreads = 4.0
var_state_enu_pos_f_y.maxreads = 4.5
var_pprz_mode.maxreads = 5.2
var_state_enu_pos_f_x.maxreads = 8.37
var_state_speed_status.maxreads = 9.33
var_state_pos_status.maxreads = 10.67
var_gps_last_msg_time.maxreads = 60.0
var_sys_time_nb_sec.maxreads = 60.0
```

Listing C.1: UPPAAL trace maxreads

# Appendix D

# Test Results

This appendix lists the results from testing Paparazzi's autopilot code, original and modified, against single event upsets. All distances are measured in metres. Table D.1 shows the Hausdorff distances without the occurrence of a single event upset. Table D.2 shows the Hausdorff distances with a single event upset present in `state.utm_pos_f->alt`. Table D.3 shows the Hausdorff distances with a single event upset present in `pprz_mode`.
Figures D.1 and D.2 illustrates the divergence in metres for variables tested on the original code. These variables was tested to confirm Hypothesis 1 on page 31.

| Type | Number of runs | Smallest divergence (m) | Largest divergence (m) | Average divergence (m) |
|---|---|---|---|---|
| Original without errors | 1000 | 10.11 | 13.81 | 11.41 |
| Modified without errors | 1000 | 9.37 | 12.93 | 10.90 |

Table D.1: Hausdorff distances gathered from testing without single event upset

| Type | Smallest divergence (m) | Largest divergence (m) | Average divergence (m) |
|---|---|---|---|
| Original with error in bit 0 | 9.68 | 11.75 | 11.14 |
| Modified with error in bit 0 | 9.78 | 11.37 | 10.67 |
| Original with error in bit 1 | 10.20 | 11.33 | 10.71 |
| Modified with error in bit 1 | 9.53 | 12.22 | 10.95 |
| Original with error in bit 2 | 10.12 | 11.37 | 10.84 |
| Modified with error in bit 2 | 10.08 | 11.38 | 10.79 |
| Original with error in bit 3 | 10.45 | 12.02 | 11.00 |

| Type | Smallest divergence (m) | Largest divergence (m) | Average divergence (m) |
|---|---|---|---|
| Modified with error in bit 3 | 9.50 | 11.36 | 10.72 |
| Original with error in bit 4 | 9.89 | 11.34 | 10.79 |
| Modified with error in bit 4 | 9.76 | 11.82 | 10.66 |
| Original with error in bit 5 | 9.95 | 11.68 | 10.93 |
| Modified with error in bit 5 | 9.78 | 11.26 | 10.74 |
| Original with error in bit 6 | 10.19 | 11.79 | 11.10 |
| Modified with error in bit 6 | 10.37 | 11.50 | 10.92 |
| Original with error in bit 7 | 9.83 | 11.80 | 10.98 |
| Modified with error in bit 7 | 9.62 | 11.01 | 10.57 |
| Original with error in bit 8 | 9.91 | 11.45 | 10.96 |
| Modified with error in bit 8 | 9.73 | 11.83 | 10.67 |
| Original with error in bit 9 | 9.36 | 11.68 | 10.58 |
| Modified with error in bit 9 | 9.35 | 12.26 | 10.72 |
| Original with error in bit 10 | 9.40 | 11.44 | 10.63 |
| Modified with error in bit 10 | 9.82 | 11.25 | 10.85 |
| Original with error in bit 11 | 9.72 | 12.31 | 11.09 |
| Modified with error in bit 11 | 10.78 | 11.32 | 11.03 |
| Original with error in bit 12 | 9.73 | 11.40 | 10.91 |
| Modified with error in bit 12 | 9.39 | 11.58 | 10.78 |
| Original with error in bit 13 | 9.91 | 12.10 | 10.89 |
| Modified with error in bit 13 | 9.52 | 11.90 | 11.01 |
| Original with error in bit 14 | 9.65 | 11.63 | 10.83 |
| Modified with error in bit 14 | 10.07 | 11.27 | 10.73 |
| Original with error in bit 15 | 9.39 | 11.27 | 10.44 |
| Modified with error in bit 15 | 10.12 | 11.54 | 10.79 |
| Original with error in bit 16 | 10.08 | 11.71 | 10.95 |
| Modified with error in bit 16 | 9.83 | 12.47 | 10.82 |
| Original with error in bit 17 | 10.13 | 12.39 | 11.05 |
| Modified with error in bit 17 | 9.75 | 11.29 | 10.79 |
| Original with error in bit 18 | 9.84 | 11.53 | 10.95 |
| Modified with error in bit 18 | 10.49 | 11.64 | 11.14 |
| Original with error in bit 19 | 9.37 | 11.25 | 10.79 |
| Modified with error in bit 19 | 9.88 | 11.45 | 10.76 |
| Original with error in bit 20 | 9.61 | 11.83 | 10.95 |
| Modified with error in bit 20 | 10.24 | 11.72 | 10.97 |
| Original with error in bit 21 | 10.09 | 11.76 | 10.80 |
| Modified with error in bit 21 | 9.66 | 11.74 | 10.90 |
| Original with error in bit 22 | 9.87 | 11.54 | 10.73 |
| Modified with error in bit 22 | 10.70 | 11.26 | 11.00 |

| Type | Smallest divergence (m) | Largest divergence (m) | Average divergence (m) |
|---|---|---|---|
| Original with error in bit 23 | 9.55 | 11.17 | 10.56 |
| Modified with error in bit 23 | 9.51 | 11.45 | 10.78 |
| Original with error in bit 24 | 10.01 | 11.64 | 10.71 |
| Modified with error in bit 24 | 9.46 | 11.72 | 10.46 |
| Original with error in bit 25 | 9.60 | 11.85 | 11.00 |
| Modified with error in bit 25 | 10.21 | 11.39 | 10.88 |
| Original with error in bit 26 | 10.25 | 11.49 | 10.87 |
| Modified with error in bit 26 | 10.42 | 11.46 | 10.97 |
| Original with error in bit 27 | 9.44 | 11.86 | 10.62 |
| Modified with error in bit 27 | 9.67 | 11.42 | 10.61 |
| Original with error in bit 28 | 9.35 | 11.92 | 10.72 |
| Modified with error in bit 28 | 9.67 | 11.74 | 10.96 |
| Original with error in bit 29 | 10.06 | 11.49 | 10.85 |
| Modified with error in bit 29 | 10.51 | 12.11 | 11.23 |
| Original with error in bit 30 | 10.12 | 12.51 | 10.91 |
| Modified with error in bit 30 | 10.07 | 11.84 | 11.16 |
| Original with error in bit 31 | 10.08 | 11.27 | 10.84 |
| Modified with error in bit 31 | 10.37 | 11.96 | 11.18 |

Table D.2: Hausdorff distances gathered from testing with error in `state.utm_pos_f->alt`

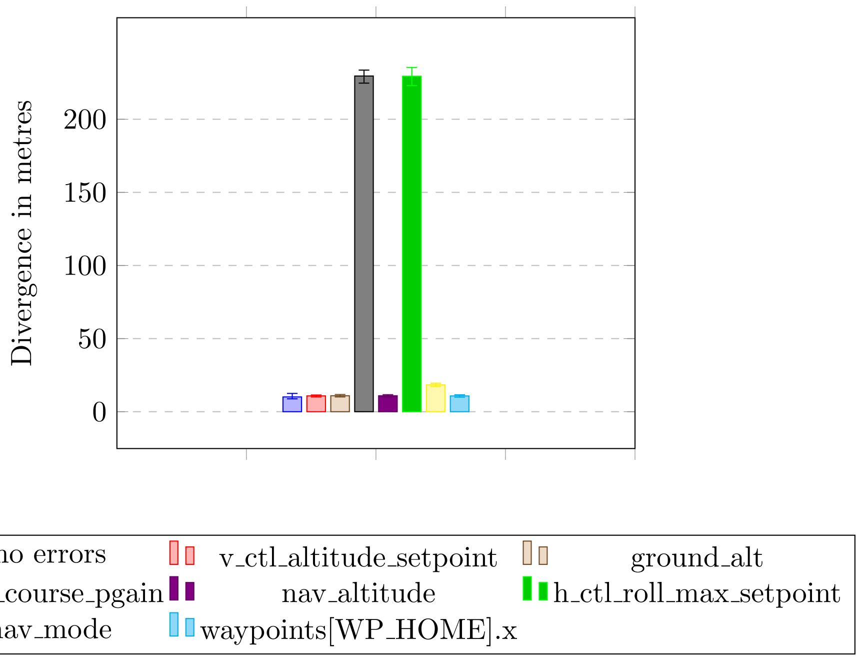


Figure D.1: Average error of variables with an average **maxreads** above 2.0. These variables where chosen because their **maxreads** are in between **state_utm_pos_f->alt** and **pprz_mode**

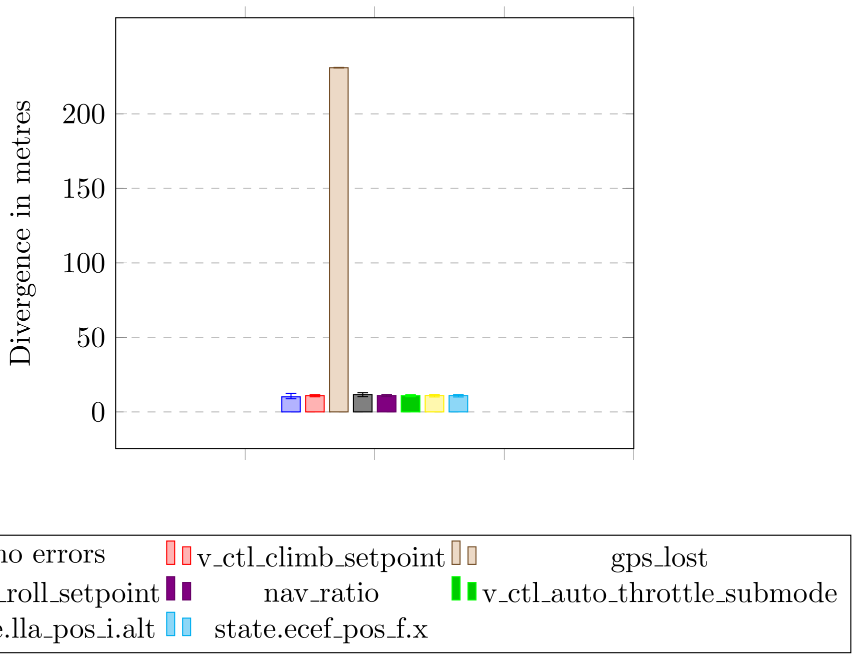


Figure D.2: Average error of variables with an average **maxreads** below 1.0

| Type | Smallest divergence (m) | Largest divergence (m) | Average divergence (m) |
|---|---|---|---|
| Original with error in bit 0 | 230.78 | 231.49 | 231.04 |
| Modified with error in bit 0 | 9.54 | 11.61 | 10.88 |
| Original with error in bit 1 | 230.77 | 231.22 | 230.99 |
| Modified with error in bit 1 | 10.30 | 11.06 | 10.78 |
| Original with error in bit 2 | 333.45 | 700.24 | 370.41 |
| Modified with error in bit 2 | 9.64 | 11.55 | 10.75 |
| Original with error in bit 3 | 333.45 | 484.68 | 348.57 |
| Modified with error in bit 3 | 10.27 | 11.55 | 11.09 |
| Original with error in bit 4 | 327.31 | 831.90 | 387.84 |
| Modified with error in bit 4 | 10.14 | 12.56 | 10.94 |
| Original with error in bit 5 | 331.00 | 660.07 | 365.87 |
| Modified with error in bit 5 | 10.21 | 11.29 | 10.74 |
| Original with error in bit 6 | 329.18 | 1213.77 | 420.81 |
| Modified with error in bit 6 | 10.74 | 11.79 | 11.25 |
| Original with error in bit 7 | 331.00 | 1197.33 | 464.85 |
| Modified with error in bit 7 | 9.69 | 12.11 | 11.05 |

Table D.3: Hausdorff distances gathered from testing with error in `pprz_mode`