Formal Verification and Fault Mitigation for Small Avionics Platforms using Programmable Logic

A thesis submitted for the degree of Doctor of Philosophy of The Australian National University

Benjamin Paul Coughlan
Supervised by
Uwe Zimmer

The Australian National University
August 2020

This work was supported by the Defence Science and Technology Organization

Statement of Authorship

I hereby state that this thesis is my own original work and that I am the sole author of this thesis. When referencing the efforts of others I have clearly cited their work in the text and provided details among the references listed.

Benjamin Paul Coughlan

Acknowledgements

To the logic group within CECS, especially Bruno Woltzenlogel Paleo. Thank you for your help and guidance in dealing with first order logic and SAT solvers. That made things *way* easier than doing everything by hand.

To YiLiang Wu in the solar group. While my work with solar cells didn't make it into this thesis, I really appreciate the time you took to help me slice and dice all that silicon as well as testing the resulting cells.

To Tim & Dan, thanks for giving me the time I needed to complete this thesis.

To all my friends whom I've neglected while writing this thesis, I believe it's time for many beers... and the first few are probably on me.

To my supervisor, Uwe, thanks for putting up with me. I know this took longer than we both expected. The only person more stubborn than myself was you, and it was your faith in me that saw this project finally reaching a satisfying end. Hopefully someday we can publish the rest.

To my Mum, thanks for everything.

Abstract

As commercial and personal unmanned aircraft gain popularity and begin to account for more traffic in the sky, the reliability and integrity of their flight controllers becomes increasingly important. As these aircraft get larger and start operating over longer distances and at higher altitude they will start to interact with other controlled air traffic and the risk of a failure in the control system becomes much more severe.

As any engineer who has investigated any space bound technology will know, digital systems do not always behave exactly as they are supposed to. This can be attributed to the effects of high energy particles in the atmosphere that can deposit energy randomly throughout a digital circuit. These single event effects are capable of producing transient logic levels and altering the state of registers in a circuit, corrupting data and possibly leading to a failure of the flight controller. These effects become more common as altitude increases, as well as with the increase of registers in a digital system.

High integrity flight controllers also require more development effort to show that they meet the required standard. Formal methods can be used to verify digital systems and prove that they meet certain specifications. For traditional software systems that perform many tasks on shared computational resources, formal methods can be quite difficult if not impossible to implement. The use of discrete logic controllers in the form of FPGAs greatly simplifies multitasking by removing the need for shared resources. This simplicity allows formal methods to be applied during the development of the flight control algorithms & device drivers.

In this thesis we propose and demonstrate a flight controller implemented entirely within an FPGA to investigate the differences and difficulties when compared with traditional CPU software implementations. We go further to provide examples of formal verifications of specific parts of the flight control firmware to demonstrate the ease with which this can be achieved. We also make efforts to protect the flight controller from the effects of radiation at higher altitudes using both passive hardware design and active register transfer level algorithms.

Contents

List of Acronyms

ADC	Analogue to Digital Converter
AMSL	Above Mean Sea Level
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
BEC	Battery Eliminator Circuit
BOM	Bill of Materials
BRAM	Block Random Access Memory
CCC	Clock Conditioning Circuit
CMOS	Complementary Metal-oxide-semiconductor
CORDIC	Coordinate Rotation Digital Computer
COTS	Commercial Off the Shelf
CPU	Central Processing Unit
CS	Chip Select
DC	Direct Current
DDR	Double Data Rate
DSM	Digital Spectrum Modulation
DSP	Digital Signal Processing
DSSS	Direct Sequence Spread Spectrum
ECC	Error Correction Codes
EEPROM	Electrically Erasable Programmable Read-only Memory
EMI	Electromagnetic Interference
ESC	Electronic Speed Controller
FF	Flip-Flop
FIFO	First in, First out
FIT	Failures in Time
FPGA	Field Programmable Gate Array
FPU	Floating-point Unit
FSM	Finite State Machine
GCS	Ground Control Station
GPS	Global Positioning System
HALE	High Altitude, Long Endurance
HGM	High Gain Mode
HDL	Hardware Description Language
IIC, I2C	Inter-Integrated Circuit
IMU	Inertial Measurement Unit
IO	Input/Output
ISM	Industrial, Scientific and Medical
ISR	Interrupt Service Routine
LDO	Low Dropout
LPDDR	Low Power Double Data Rate
LUT	Lookup Table
MAC	Media Access Control

MAV	Micro Air Vehicle
MEMS	Microelectromechanical Systems
MMCM	Mixed-mode Clock Manager
MUX	Multiplexer
NVM	Non-Volatile Memory
OCM	On Chip Memory
OSI	Open Systems Interconnection
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PHY	Physical Layer
PID	Proportional, Integrator, Derivitive
PLD	Programmable Logic Device
PLL	Phase Locked Loop
PWM	Pulse-width Modulation
RAM	Random Access Memory
RC	Resistor/Capacitor
RF	Radio Frequency
RISC	Reduced Instruction Set Computer
ROM	Read-only Memory
RTL	Register Transfer Level
SD	Secure Digital
SDI	Serial Data In
SDO	Serial Data Out
SEE	Single Event Effect
SEFI	Single Event Functional Interrupt
SEL	Single Even Latch-up
SET	Single Event Transient
SEU	Single Even Upset
SMT	Satisfiability Modulo Theory
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SoC	System-on-Chip
SoM	System-on-Module
UART	Universal Asynchronous Receiver/Transmitter
UAV	Unmanned Aerial Vehicle
UHF	Ultra-high Frequency
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language

List of Figures

Introduction

Avionic systems are required to be exceptionally reliable as their failure can result in significant damage to or loss of an aircraft, loss of capabilities in a higher level system and even loss of human life. System reliability is generally defined with regard to a mean time to failure. The fly-by-wire control system of the Boeing 777 [1] requires the probability of failure for each component to be less than 10^{-10} per flight hour. This level of reliability is achieved through use of proper architectural design, redundancy involving dissimilar implementations as well as rigorous testing and verification.

With the advent of small, commercially operated or even personal unmanned aircraft, a new generation of flight controllers has emerged which are capable of piloting aircraft while being constructed of cheap COTS parts that are available to unaffiliated individuals. These devices are typically the result of academic research or open source community efforts and are not expected to reach the same level of reliability as commercial or military grade avionics deployed on full scale aircraft. In addition, the smaller airframes used for unmanned aircraft may not support the size, weight or power requirements for redundant avionics components, greatly increasing the reliability requirements of an individual device.

Despite the shortcomings of common commercial avionics, their responsibility for the safety of the aircraft, the operator, nearby property and people, sees no reduction. Given the potential number of these aircraft operating in civilised airspace in the near future, often by untrained operators, it can be argued that the required reliability of these devices is significantly increased to achieve the same safety outcomes.

There are a number of fault sources that can cause avionics components to fail including hardware faults, programming errors, and Single Event Effects (SEEs) cause by high energy particles corrupting digital storage. While an aircraft is a large and complex system consisting of many electrical, mechanical and software components, each of which contributes to the aircraft's mean time to failure, this thesis will focus on the software & firmware components.

The expected reliability of software within an airborne system is prescribed in standards such as DO-178C [2] which details the required software development life cycle stages and artefacts that can provide verification of the software with respect to the standard. The end

result of such a certification being confidence that the software artefacts implement specific requirements, only those requirements, and that the mean time to failure is satisfactorily large. Similarly, DO-254 [3] provides standard practices for the verification of hardware in airborne systems. Within this thesis the term firmware will be used to refer to computational logic implemented within a programmable logic device, while hardware will refer to the physical devices on which that logic operates, as well the peripheral devices and supporting components that make up hardware devices. Software will refer to implementations that are executed on a CPU or microcontroller.

Software systems have traditionally been preferred by developers for their flexibility and rich environment of tools and knowledge. While capable software compilers and processors to execute the results have been widely available for many decades, PLDs, FPGAs and ASICs have either been too simplistic to compete with the potential scale of a software system, or far too cumbersome for developers to work with. However, in the last 20 years vendors have brought commercially available FPGAs to market at a scale that can implement a complete autopilot as well as the tools to develop for them. I aim to show that an autopilot can be implemented without any software components, and that the firmware components of this device can be formally verified. This device should be capable of autonomously piloting a fixed wing aircraft to follow prescribed waypoints without human interaction.

I propose an hardware design that is centred around an FPGA, to provide a device that can be formally verified and provides mitigation against SEEs. All peripheral drivers, state estimation and navigation algorithms, actuator control and even communications will be implemented within the FPGA. I will cover the requirements of the devices and physical hardware design in detail, albeit similar to most other commercial autopilots. Once the hardware platform is established, a detailed design of the firmware components will be provided to demonstrate the complete system. From this firmware design, a few key constructs (such as counters, finite state machines etc.) will be used as examples for formal verification techniques. I will also provide techniques and algorithms for the radiation hardening of firmware components.

Firmware Verification

With the goal of verifying the behaviour of the software and firmware components, developers will typically resort to unit tests which exercise components in isolation with the intention of verifying them against their specification. This requires the unit tests themselves to prove confidence that they cover all significant fault paths identified in the implementation and that they themselves are correct. It is often very difficult, or even impossible to provide unit tests that exhaustively test a given component. Another approach is to apply some form of formal methods to the components in which formal proofs can be formed showing various assertions on a model representation of the component. This potentially provides exhaustive reasoning that a component behaves correctly for all combinations of inputs and their timing. Applying formal methods to a design requires skills are that are not typically found in an industrial software/firmware development team. The process of mapping an implementation to a formal specification manually can be quite a cumbersome and error prone process. There has been some effort to generate behavioural implementations automatically from formal specifications [4] but is yet to be widely adopted.

Certification against DO-178C and DO-254 requires that both the software & firmware components and the hardware devices they are executing on have been verified with respect to their specifications and that those specifications are valid with respect to the airborne system. While these two standards allow certification credit for the use of COTS parts, modern processors and operating system make formally verifying software components very difficult. While a COTS processor itself may have formal verification of its behaviour, features such as out of order execution, caching & virtual memory, simultaneous multithreading and single instruction multiple data cycles, all in the presence of asynchronous interrupts significantly increases the complexity of formally verifying even a simple software component.

In contrast, formal verification of firmware components is much simpler. The component implementation is typically translated into a formal language such as first order logic [5] or metric temporal logic [6] to provide a model. A number of assumptions providing the specification of the component's inputs are provided as well as the required properties of its outputs. These properties can then be proved or disproved by an automated theorem prover [7] [8] [9] [10], or even by hand in some cases, to provide a verification of the firmware component's behaviour. As firmware components within an FPGA exist on their own dedicated silicon with

no shared resources and well defined interfaces, these models remain as simple specifications of the component and only the component.

Of course the physical implementation is itself subject to verification with regard to logical correctness and physical timing. As these are typically COTS parts, vendor tools ensure a particular design behaves correctly on a specific device, providing the developer with timing and routing reports. It is left as the vendor's responsibility to ensure these tools are correct, and the resulting design performs as specified by the user/designer.

Single Event Effects

While formal verification provides confidence in the logical implementation of a digital system, there are phenomena that can affect the behaviour of such a system beyond the scope of verification. Assuming that a design is deployed within appropriate conditions such as temperature and clock frequency, it may still be affected by high enery particles originating externally which may cause transient signal levels, toggle register values, or even damage components [11] [12] [13] [14]. These effects are collectively known as Single Event Effects (SEEs) and can be described in three main categories: Single Event Transients (SETs), Single Event Upsets (SEUs) and Single Event Functional Interrupts (SEFIs) [15]. While SEEs are only a minor concern for a single device at sea level, their frequency significantly increases with altitude & latitude [16] such that designers of airborne system must provide mitigation to maintain system reliability. Increases in the density of SRAM elements within digital circuits, smaller fabrication processes and lowering operating voltages also contribute to the increased sensitivity of these devices to SEEs.

The fundamental cause of any SEE begins as an SET; a particle impacts a digital circuit with enough energy to cause a significant transient signal somewhere within the design [17]. Where this SET occurs, its amplitude and its duration all combine to determine the effect it will have on the system. For example, if an SET occurs on an asynchronous signal such as an interrupt or reset, these could be triggered erroneously; if an SET were to occur on a synchronous data signal within the required window of its clock edge, erroneous data could be latched into the register; and if a significant SET occurs within the transistors of a Flip-Flop (FF) it could actually toggle the state of the FF resulting in an SEU.

A Single Event Upset occurs when an SET affects part of a circuit involving a memory element (such as SRAM or a FF) resulting in the corruption of the data element. Single Event Upsets are arguably the most significant concern for designers given they are an erroneous state of a memory element by definition, and without mitigation they are likely to be persistent, feeding errors downstream far beyond the SET that originally caused it.

Single Event Functional Interrupts are a special case of SEU that affect the behaviour of the digital design rather than just corrupting data. These are most notable on SRAM FPGAs in which the behaviour of the design is stored in many SRAM elements including the routing switches, LUT implementations and configuration of more complex components within the device. An SEU occuring on any of these elements will persistently alter the behaviour of the design potentially even in circuits designed to otherwise detect errors.

Radiation Hardening

Many techniques have been developed to harden digital designs to mitigate SEEs and they all boil down to redundency; maintaining several copies of a design module with the same inputs and running them in parallel. By comparing the outputs of redundant copies errors can be detected and potentially corrected. Redundancy can be implemented spacially by simple duplication, or temporally by performing the same operation multiple times. Hardening techniques can be implemented at different layers within the design process, starting with direct hardware hardening, injecting hardening circuits into user designs during compilation/synthesis, or included by the designer explicitly.

The most common technique in radiation hardening, be it at the hardware or synthesis or design level, is Triple Modular Redundency (TMR) [18]. This simply involves maintaining three seperate copies of the module to be hardened and providing a voting algorithm to detirmine the correct output. While this concept seems quite simple, its practical implementation requires particular attention to the resulting FPGA configuration, particularly with regard to the position and routing of the voting circuits [19]. Some vendor tools allow for the automatic triplication of design elements at a significant cost to resource usage within the device.

Some FPGA vendors offer radiation tolerant or radiation hardened products, particularly for space or military applications. These radiation hardened devices attempt to mitigate

SEEs at the hardware level with specialised fabrication technologies [20] [21] which helps reduce the SEE cross section for elements within the device; however behavioural design SEE mitigation is still required for design data elements. There are two significant approaches to FPGA hardware implementations which have significant consequences for the required SEE mitigations strategies. The more common SRAM based FPGAs must consider SEEs in the configuration memory of the device [22] [23], while FLASH or anti-fuse based FPGAs [24] typically have a configuration memory which is immune to SEEs. Given FLASH based FPGAs have a persistent configuration memory, the energy required to toggle any bit is significantly higher, thus greatly reduces the SEE cross section of those elements. However this does not extend to the data storage elements within a design such as flip-flops and block-RAMs. SRAM based FPGAs must actively monitor their configuration memory for SEUs and reconfigure when required in a process known as scrubbing [25] [26]. Radiation tolerant SRAM based FPGAs typically provide hardware support for monitoring and scrubbing.

More selective approaches to building redundency into user designs exist [28] by providing libraries that take advantage of the strong typing of VHDL to allow designers to explicitely insert TMR logic in the sensistive parts of their design. A more extreme approach [29] takes advantage of detailed application specific knowledge and formal methods to validate data based on relationships implicit in the design. These fine-grained approaches save significant resources when compared with a global TMR regime implemented by the synthesizer. However it does require much more explicit consideration of the effects of SEE in particular regions of a design. There is also the opportunity to replace specific implementations of redundancy with other algorithms such as Hamming Codes [30] [31].

For our purposes, it is highly desirable that any SEE induced faults are entirely masked and that we do not incur any downtime to recover from faults. Consider a hardened Inertial Measurement Unit (IMU) which is maintaining the current attitude of an aircraft. The aircraft does not stop manoeuvring to allow for the recovery of a fault. This implies that the techniques required to harden SRAM based FPGAs are not feasible in our design, and thus neither are SRAM based FPGAs themselves. Our design will instead use a FLASH based FPGA of which the configuration memory is immune to SEEs and we need only consider these effects on the data elements in the design.

Validating SEE Mitigation Strategies

Significant effort has been put towards validating the hardening strategies described previously in different context and operating environments. Testers implement fault campaigns against a design under test by injecting faults at various times and places while monitoring and validating the output to assess the rate at which the design detects and corrects faults. I would categorise these efforts into three distinct groups, based on the use of simulation versus actual hardware implementations of the hardened designs under test.

The first group focuses on simulation of the circuit under test [32], relying on the design to be executed and interacted with within a simulation environment. RTL simulation is a standard part of any HDL designer's toolbox and is often the only way to observe faults in a design to aid in debugging. These simulation tools are widely available, including many open source, non-vendor specific options. However, RTL only simulation is only valuable when the design is well behaved (probably synchronous) and the designer is disciplined with their implementation. More advanced simulations use the post-synthesis and post-routing implementations which include models of the actual vendor specific components that would be used to implement the design on an actual FPGA. These models can also be annotated with realistic propagation delays through the elements themselves as well as routing resources used. As will be discussed in this thesis, SEEs are not "well behaved" and significant differences were found with different simulation techniques.

The second group uses actual FPGA hardware to implement the design under test and provide it with real time inputs while using techniques such as bitstream modification [33] or partial reconfiguration to "inject" faults into the design [25] [26]. Partial reconfiguration is a feature of many modern FPGAs which allows part of the configuration memory of the FPGA to be rewritten while the device is running, without interrupting other parts of the design. Faults can therefore be injected anywhere within the design, including the configuration memory to test against SEFIs. This process allows control over where faults occur and can perform much faster than simulation based campaigns. However it is very limiting with regard to "when" the faults happen, typically requiring the design under test to be restarted after each reconfiguration. Asynchronous SEEs cannot be effectively observed with this technique.

Finally, actual hardware devices can be subjected to elevated radiation environments through the use of particle accelerators and ionizing radiation emitting masses such as at the LANSCE facility in Los Alamos [34] [35]. The advantage of such realistic testing is that it covers all aspects of a design, whereas simulated fault campaigns may neglect areas not considered by the designer. The disadvantage is the lack of control on when and where faults are injected within the design.

As we have opted to use a FLASH based FPGA in our design, we expect the configuration memory to be immune to SEEs. Therefore we will not consider testing these regions of the design. However we must still consider validating our radiation hardening techniques with the data memory of the design including Flip-Flops and Block RAMS among other potential locations. The most effective method was to use simulation software to inject faults in specific locations. While our RTL simulation demonstrated effective mitigation of faults, the effect of propagation delays with asynchronous faults often subverted our attempts to completely mask errors. This effect is discussed further in "Validating the ECC Implementation" on page 48.

Existing Autopilots

At the time of writing, there are several commercially available, popular avionics platforms for small, privately operated unmanned aircraft. There are also numerous bespoke platforms in various research institutions and commercial products. Many of the most popular platforms are summarised in [29]. There are two distinct styles of flight controller that we are interested in: Those based on a microprocessor or CPU; and those based on FPGA or SoC processors including programmable logic.

Beginning its life as a research platform for small scale Micro Airborne Vehicles (MAVs) the Pixhawk [36] has become very popular among amateur and commercial UAV developers. Originally developed as a part of the Pixhawk project from ETH [37] this device provides a powerful microprocessor, the ARM Cortex-M4 running at 168MHz with an included FPU. This hardware platform is most commonly found running the Ardupilot [38] suite of software and has proven successful in many academic and hobbyist scenarios. The hardware platform is a small, efficient collection of peripherals including accelerometer, gyroscope, compass and barometer. All memory is included within the CPU package and many external connectors are provided for attaching other sensors and actuators.

The OcPoC [39] and Phenix Pro [40] both use a Xilinx Zynq SoC which includes a dual-core Cortex-A9 Processor and FPGA. These devices are at the very high end in terms of processing power and the CPU included is surely intended for much more than just controlling the aircraft. This is apparent from power requirements of each of these devices at 4W and 2.6W respectively [29]. While the processing architecture is the most similar to the design presented in this thesis (at least in shape if not processing power) the software and firmware implementations are significantly different. The OcPoC ships with the Ardupilot software, while the Phenix Pro advertises a bespoke real-time operating system PhenOS. Both of these platforms advertise the use of the FPGA for payload operations such as video processing and computer vision with the occasional mention of the FPGA as an accelerator for CPU operations. This is in contrast with this work in which the FPGA is the primary flight controller without assistance from the CPU.

Some other, somewhat older, publications [41] [42] have described autopilot designs that include FPGAs beside a main CPU. These describe the advantages of the FPGA as allowing flexible IO and parallel processing to off-load tasks from the main processor. These things are true, but do not address the requirements for developing a platform towards certifiable safety critical systems that are discussed in this thesis. Some also touch on redundancy by allowing multiple redundant sensors to be attached to the device, but do not make any mention of the processing system itself. As I will discuss in this thesis, the true value of discrete logic processors is their determinism and the ability of the designer to build in redundant

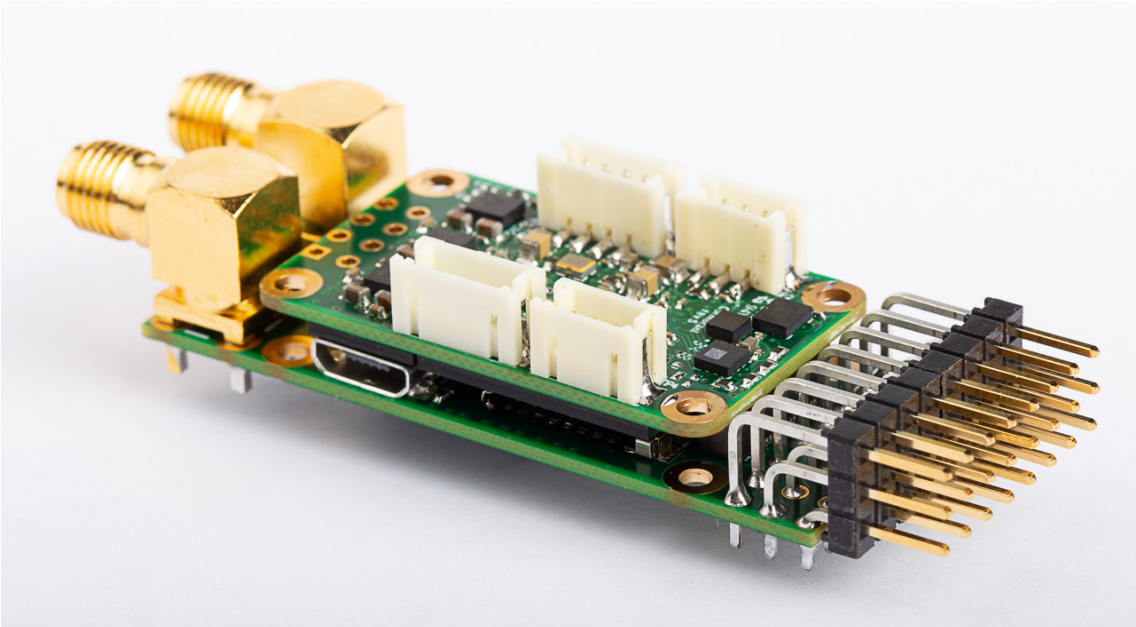

Fig. 1 The complete, assembled flight controller.

processing elements. These things lead to a hardware platform that can be formally verified and radiation hardened for operations in safety critical and high neutron flux environments.

The work presented in this thesis is intended to provide a platform on which users can implement flight controllers of similar or improved performance, potentially leveraging exisiting IP from other platforms. I have intended to provide processing capability very similar to the Pixhawk while still providing for the verification and fault mitigation strategies described. While the Cortex-M3 is typically clocked slower than the Cortex-M4 and is missing an FPU, I would argue that much of this functionality could be recovered by moving software components to the accompanying FPGA.

While other designs presented include FPGAs, I did not encounter an example in the literature that included one with any hardware radiation hardening. Most autopilot designs I found were based on the Zynq SoC, which is a very powerful processor and FPGA combination, but is still SRAM based and so SEFIs still pose a significant problem, not to mention the power consumption of these devices.

The autpoilot presented here is based on the M2S025 SmartFusion SoC which includes a Cortex-M3 and FPGA. It is expected that this platform would be capable of running current ArduPilot implementations (with some modifications) while allowing designers to consider and experiment with the verification and hardening strategies described in this thesis.

Fig. 2 The Pulsar 2.5E assembled before takeoff.

Experimental Platform

The flight controller presented in this thesis was developed to support an experimental, solar powered, fixed wing glider based on the Pulsar 2.5E airframe. This aircraft is an aerodynamically efficient, powered glider which has been endowed with photo-voltaic cells to generate power and recharge the battery while the aircraft is in flight. The aircraft has a wingspan of 2.5 m and a takeoff weight of 1126 g. This aircraft is intended as a platform to develop and test behavioural optimization for a solar powered aircraft working towards larger scale High Altitude, Long Endurance (HALE) aircraft [44] [45].

The use of this airframe constrains the flight controller used in project. The fuselage is very narrow, with its larger sections dedicated to housing the battery. As it is intended to be solar powered for long durations, power consumption is incredibly important. Given its long endurance and potentially high altitude operations, a highly reliable flight controller with radiation tolerance is required. This flight controller design presented in this thesis provides for all of these requirements.

Fig. 3 Ben Coughlan hand launching the Pulsar 2.5E on its maiden flight.

Over Thinking Counters

Formal verification of processing logic, whether it be implemented in hardware or software, is fundamental to the behavioural verification of modern complex systems. Modern avionics, especially of the kind found in unmanned aerial vehicles, derive their behaviour completely from algorithms implemented on some form of processor. The correct functioning of this processor and the algorithms it is executing is critical to the aircraft over which it has control. As these aircraft grow in popularity and begin to operate in airspace shared with manned aircraft, the standards for design and manufacturing of the avionics involved must improve to match or exceed devices already operating in the aviation industry.

Here I investigate the formal verification of algorithms implemented on a Field Programmable Gate Array (FPGA) using increasingly complex implementations of a counter for illustration. The specifications of the counter are defined using first order logic and the implementations are provided in VHDL. Additionally the specification is implemented in SMT2 for automated theorem proving. The practical implications of each implementation are discussed before a formal verification against each specification. A functionally similar algorithm using a software implementation is also presented and the difficulties of its formal verification are discussed.

A Simple Counter

The humble counter is a staple in any hardware designer's toolbox. It is a simple register and an adder that can be used to count events within a system. When those events are periodic (such as the system clock) the counter can be used to generate events at regular intervals to manage timing within a real-time system. Counters are commonly used to divide clock rates down for use as baud rate generators for serial communications, sample rate timers and pulse width generators, often directly controlling actuators, etc. Clock management devices within FPGAs that allow for rate changes such as PLLs and MMCMs typically include a counter as an output divider. As time and event sequencing is a fundamental aspect of any real-time processing system, the counter plays a very important role and warrants close attention to ensure that it performs as expected. The counter's relative simplicity also makes it an excellent example to explore in detail.

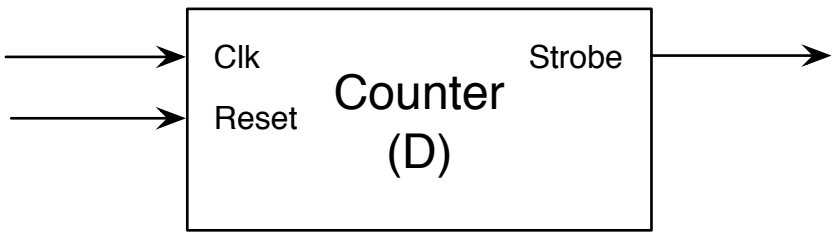

Fig. 4 Simple counter entity.

In this chapter I will be demonstrating the requirements, design & implementation of a counter and discussing the verification of such a design to provide an example of the use of formal methods in a hardware system.

In order to formally verify a counter, we must first define its requirements. Out first example is a very simple counter that divides the input clock frequency by a constant value.

1. The counter shall periodically assert strobe when reset is not asserted.
2. The period of the strobe output is the period of the clock input × D.
3. The counter shall not assert strobe while reset is asserted.

To formalise these requirements we translate them into first order logic [66] to become conjecture for which we verify a model of the implementation to come.

$$\forall t \left(\begin{array}{c} \text{strobe}(t) \wedge \forall t' \, (t \leq t' < t + D \Rightarrow \neg \text{reset}(t')) \\ \Rightarrow \text{strobe}(t + D) \end{array} \right) \qquad [3.1]$$

This equation provides that a **strobe** at time **t** implies that another **strobe** will occur at time t+D so long as **reset** is not asserted during that time.

This formalisation assumes a discrete time governed by the system clock in which a variable $t \in Z$ represents the sequence of rising edges of that clock. A discrete time is appropriate for synchronous systems in which every signal is driven by a register sensitive to a global clock. This allows us to only consider the values of those signals on the rising edge of the system clock. This of course assumes that any propagation delays in the physical implementation of the system are shorter than the clock period; this is usually enforced by vendor tools during synthesis. Asynchronous logic, such as input from external devices is required to be synchronised to the system clock.

While [3.1] provides that **strobe** is periodic while **reset** is not asserted, it does not say anything about the state of **strobe** between the periodic assertions, or while **reset** is asserted. These require further conjecture.

$$\forall t \begin{pmatrix} \text{strobe}(t) \Rightarrow \\ \forall t' (t < t' < t + D \Rightarrow \neg \text{strobe}(t')) \end{pmatrix} \qquad [3.2]$$

$$\forall t (\text{reset}(t) \Rightarrow \neg \text{strobe}(t + 1)) \qquad [3.3]$$

We now have constraints on **strobe** in between periodic assertions that it is not asserted. We also have that asserting **reset** will prevent **strobe** from asserting at the next time **t**. Note that there is a single time-step delay from asserting **reset** to expecting **strobe** to be de-asserted. This is because we require a **reset** synchronous with our system clock, as an asynchronous **reset** will violate the discrete time assumption.

We now have a periodic **strobe** that is defined for both the periodic events, the time in between those events and whenever **reset** is asserted. Our conjecture so far relies on **strobe** being asserted at some **t** before any periodicity will hold, so finally we must specify that strobe eventually (for some practical interval) asserts after **reset** is de-asserted.

$$\forall t \begin{pmatrix} \forall t' (t \leq t' < t + D \Rightarrow \neg \text{reset}(t')) \\ \Rightarrow \exists \tau (t < \tau \leq t + D \wedge \text{strobe}(\tau)) \end{pmatrix} \qquad [3.4]$$

This conjecture requires that if **reset** is de-asserted for long enough, then **strobe** will be asserted within some interval. This doesn't specify exactly when **strobe** will be asserted as we don't care that much in this example, as long as it happens within one period of the counter.

We now have a complete set of constraints for the behaviour of the **strobe** output of our counter.

These axioms provide a formal specification for the behaviour of the system described in our original requirements that cannot be misinterpreted, providing a solid place to start with the design and implementation of the counter. In addition, once the implementation is complete, these axioms can be used, along with a model of the implementation, to automatically prove the implementation is consistent with its requirements.

Implementing the Counter

The counter is to be implemented on an FPGA which will contain a set of flip-flops, lookup tables and routing to provide a layout matching our detailed design. This design is supplied in a hardware description language. For this example, as well as the other work in this thesis we will be using VHDL for detailed designs.

```
entity Counter is
    generic (
        N : integer range 2 to 100
    );
    port (
        Clk     :   in std_logic;
        Reset   :   in std_logic;
        Strobe  :   out std_logic
    );
end entity;

architecture Behavioural of Counter is
    signal Count : unsigned(N downto 0);
begin
    process(Clk) is
    begin
        if rising_edge(Clk) then
            if Reset = '1' then
                Count <= (others => '0');
            else
                Count <= ('0' & Count(N-1 downto 0)) + 1;
            end if;
        end if;
    end process;

    Strobe <= Count(N);
end architecture;
```

Fig. 5 VHDL Implementation of a simple counter.

This implements a simple modulo 2^N counter which increments by 1 on each clock edge and provides a single cycle **strobe** every time the counter wraps to zero. This simple counter can be used to divide the system clock frequency by 2^N and trigger events at the lower rate. Using a modulo-2 counter is a common optimization over the more obvious counter that compares to some maximum value. By specifying **count** as an N-bit unsigned type (actually N+1 bits

as mentioned later), we avoid leaving this implementation detail to the synthesizer. While using an integer type matches the level of abstraction of our specification, eventually this register will be implemented in a fixed number of bits in the FPGA fabric which has implications for when its value will wrap. We also avoid the need for a comparator against a MAX value, which reduces the resources needed for this counter. This does now constrain the selection of D to some 2^N value which can be propagated back to our higher level specification.

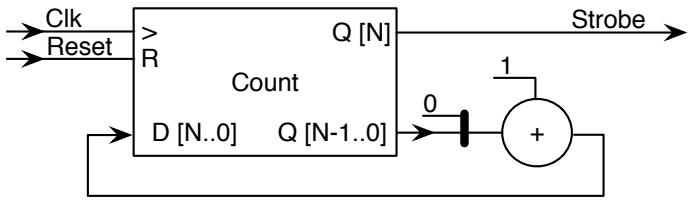

Fig. 6 RTL Schematic of a simple counter.

This simple counter consists of two elements, a register and an adder. On the first rising edge of **clock**, with **reset** asserted, the value presented on the Q port of **count** will be zero. This will propagate through the combinatorial adder to present a 1 on the D port of **count**. On the next rising edge of clock, when **reset** is de-asserted, the value on port D of **count** will be latched into the register and presented on port Q. Again this propagates through the adder and provides our increment each clock cycle. When the value of **count** is at its maximum 2^N-1, the adder will overflow and wrap to zero. The count register is one bit longer than needed to store the overflow of the adder which will be used to drive the strobe output. This bit is excluded from the adder input so that the increment remains modulo 2^N.

From this design we will extract the model of the entity as a set of axioms in first order logic. While there have been attempts to automate this process [63] entities at this level of complexity are easily modelled by hand.

$$\forall t\,(\mathrm{reset}\,(t) \Rightarrow \mathrm{count}\,(t+1) = 0) \qquad [3.5]$$

$$\forall t\,(\neg\mathrm{reset}\,(t) \Rightarrow \mathrm{count}\,(t+1) = \mathrm{count}\,(t)\,\mathrm{mod}2^N + 1) \qquad [3.6]$$

These axioms are simple translations of the decision tree within the VHDL process. As the process is synchronous (everything is registered on the rising clock edge) all decisions are based on the state at time t, with assignments taking affect at time t+1. Equation [3.5] resets

the counter to zero whenever reset is asserted, while equation [3.6] provides the increment and wrap around logic of the N-bit unsigned vector. It should also be noted that some theorem provers support non integer numeric types such as bit vectors, which imply modulo arithmetic as included here; we're limiting the complexity of the logic for illustrative purposes.

Finally the combinatorial strobe output must also be specified.

$$\forall t\,(\mathrm{count}(t) \geq 2^N \Leftrightarrow \mathrm{strobe}(t)) \qquad [3.7]$$

As strobe is driven by the MSB of count (the overflow bit), strobe will be asserted when, and only when, count is greater than or equal to 2^N.

We now have a model of our counter in equations [3.5], [3.6] and [3.7] as well as conjecture we wish to prove in equations [3.1], [3.2], [3.3] and [3.4]. To prove this conjecture holds over our model, we can write them in SMT and hand them to a theorem prover [7] [8] [9]. First we prove that the model is satisfiable with no conjecture to show that it is at least consistent.

```
(declare-fun Count (Int) Int)
(declare-fun Reset (Int) Bool)
(declare-fun Strobe (Int) Bool)
(declare-const MAX Int)

;;3.5 - Count Resets to zero on reset
(assert (forall ((t Int))
    (=> (Reset t)
        (= (Count (+ t 1)) 0))))

;;3.6 - Count increments every t without Reset
(assert (forall ((t Int))
    (=> (not (Reset t))
        (= (Count (+ t 1)) (+ (mod (Count t) MAX) 1)))))

;;3.7 - Strobe asserts whenever Count = MAX
(assert (forall ((t Int))
    (and
        (=> (>= (Count t) MAX)
            (Strobe t))
        (=> (not (>= (Count t) MAX))
            (not (Strobe t))))))

;;Strobe happens exactly every MAX cycles
(define-fun periodic () Bool
    (forall ((t Int))
        (=> (and
                (Strobe t)
                (forall ((e Int))
                    (=> (and
                            (>= e t)
                            (<= e (+ t MAX)))
                        (not (Reset e)))))
            (and
                (Strobe (+ t MAX 1))
                (forall ((e Int))
                    (=> (and
                            (> e t)
                            (<= e (+ t MAX)))
                        (not (Strobe e))))))))
;;(assert (not periodic))

(check-sat)
```

Fig. 7 SMT2 Specification of the simple counter.

Then we introduce the negation of each conjecture and prove that the model is unsatisfiable. Doing this for each conjecture in isolation, we can prove the model meets the safety requirements specified in our formal requirements.

Extending the Simple Counter

The counter above is a very simple example only capable of dividing an input clock by some 2^N. In many situations a designer will be required to divide the input by a more arbitrary ratio and it would be desirable to use inputs other than the system clock, such as another cascaded counter. Our second look at the counter introduces an enable input to allow arbitrary input events, Fig. 8. The model must now be updated to match the improved implementation. Equations [3.5] and [3.7] can remain as they are, but [3.6] must be replaced with

$$\forall t\left(\begin{array}{r}\neg reset(t) \wedge enable(t) \Rightarrow \\ count(t+1) = count(t) \bmod 2^N + 1\end{array}\right) \qquad [3.8]$$

$$\forall t\,(\neg reset(t) \wedge \neg enable(t) \Rightarrow count(t+1) = count(t)) \qquad [3.9]$$

Equation [3.8] replaces [3.6] as the model of the implementation to include the enable input while [3.9] constrains the no-op case. While the VHDL implementation and synthesized hardware imply that count won't change in the case enable is not asserted, we must specify it here for completeness.

```
entity Counter is
    generic (
        N : integer range 2 to 100;
    );
    port (
        Clk     :   in std_logic;
        Reset   :   in std_logic;
        Enable  :   in std_logic;
        Strobe  :   out std_logic
    );
end entity;

architecture Behavioural of Counter is
    signal Count : unsigned(N downto 0);
begin
    process(Clk) is
    begin
        if rising_edge(Clk) then
            if Reset = '1' then
                Count <= (others => '0');
            else
                if Enable = '1' then
                    Count <= ('0' & Count(N-1 downto 0)) + 1;
                end if;
            end if;
        end if;
    end process;

    Strobe <= Count(N);
end architecture;
```

Fig. 8 VHDL Implementation of the extended counter.

The addition of the enable input and the change in behaviour calls for a new specification of the counter. The biggest difference in the behaviour of the counter is the definition of input events to which the counter reacts. Originally these events were simply system clock edges. Now, input events are defined by input clock edges on which enable is asserted. This removes the assumptions we had in our original specification that input events were periodic and on every time step. Without these assumptions we are now forced to include some model of the input in our specification. That is, we must count and be aware of the number of input events to the counter before we can predict its behaviour.

$$\forall t\left(\begin{array}{r}\text{reset}(t) \vee \text{strobe}(t+1) \Rightarrow \\ \text{incount}(t+1) = 0\end{array}\right) \qquad [3.10]$$

$$\forall t\left(\begin{array}{c}\neg\text{reset}(t) \wedge \text{enable}(t) \wedge \neg\text{strobe}(t+1) \Rightarrow \\ \text{incount}(t+1) = \text{incount}(t) + 1\end{array}\right) \qquad [3.11]$$

$$\forall t\left(\begin{array}{r}\neg\text{reset}(t) \wedge \neg\text{enable}(t) \Rightarrow \\ \text{incount}(t+1) = \text{incount}(t)\end{array}\right) \qquad [3.12]$$

These three axioms are not a model of the counter's behaviour, nor are they conjecture we wish to prove about the counter; they are instead a model of the input events to the counter that provides required context for the conjecture we wish to prove.

Finally the safety properties of the counter must be re-specified:

4. The counter shall assert strobe once after every 2^Nth input event.

This is the only constraint we are left with, as the increased ambiguity of the counter's behaviour given the non-deterministic input removes any assumptions we can make regarding periodicity or eventuality of the strobe output.

$$\forall t\left(\begin{array}{l}\neg\text{reset}(t) \wedge \text{enable}(t) \wedge \text{incount}(t) = 2^N \Rightarrow \\ \text{strobe}(t+1)\end{array}\right) \qquad [3.13]$$

Note that this conjecture relies on *incount* from the input model to determine how many input events have occurred as we can no longer infer this directly from our time variable.

Experienced designers will likely have noticed a bug in this implementation, as will surely be pointed out by any theorem prover examining this model. For illustration purposes, we provide a proof by contradiction in Fig. 9.

$$\forall t\left(\begin{array}{l}\text{strobe}(t) \Rightarrow \\ \neg\text{reset}(t-1) \wedge \text{enable}(t-1)\end{array}\right) \qquad \text{RTP}$$

$$\neg\text{reset}(\tau) \wedge \text{strobe}(\tau) \wedge \neg\text{enable}(\tau) \qquad \text{assum.}$$

$$\text{strobe}(\tau) \Rightarrow \text{count}(\tau) \geq 2^N \qquad 3.7$$

$$\neg\text{reset}(\tau) \wedge \neg\text{enable}(\tau) \Rightarrow \text{count}(\tau+1) = \text{count}(\tau) \qquad 3.9$$

$$\therefore \text{count}(\tau+1) \geq 2^N$$

$$\text{count}(\tau+1) \geq 2^N \Rightarrow \text{strobe}(\tau+1) \qquad 3.7$$

$$\text{strobe}(\tau+1) \Rightarrow \text{enable}(\tau) \Rightarrow \perp$$

Fig. 9 Proof of safety violation when enable is not asserted at the same time as strobe.

The assumption at the beginning of this proof is a common input state where the output strobe was asserted but no input event occurred on the same clock edge. This will happen frequently in applications where enable is periodically asserted, or in fact most applications where enable is not constantly asserted. This condition results in strobe remaining asserted as it is driven by a registered overflow bit from the counter. As this bit is not updated unless enable is asserted, this bit remains high until the next input event, leaving strobe asserted and thus violating our safety property in [3.2]. The fix is simple of course; by clearing this bit when enable is de-asserted (the equivalent of count = count mod 2^N).

Adding Jitter

While it is rarely the designer's intention to add jitter, it is sometimes unavoidable. The counter so far has only been capable of counting modulo 2^N events, which is likely not the number of events a designer cares about. If we were to remove the modulo 2^N optimization we made at the beginning, we would still be limited to integer number of events. While this is more general than modulo 2^N it is still limiting. This limitation can be overcome by incrementing the counter with a value other than one. By selecting appropriate exponent and increment values we can approximate any ratio less than one, between input and output events.

The approximation is where the jitter comes in. As the output is still quantized to a discrete time step it is not possible to divide by a non integer ratio. The best that can be achieved is that subsequent divisions 'average' to the desired ratio over many iterations. In practice, this involves asserting the strobe output at slightly different sized intervals, the difference being considered jitter. While this is a deterministic process for any output event with reference to a specific reset event, this is not how most designers of downstream entities are likely to consider it. For the remainder of this discussion, this jitter will be considered non-deterministic.

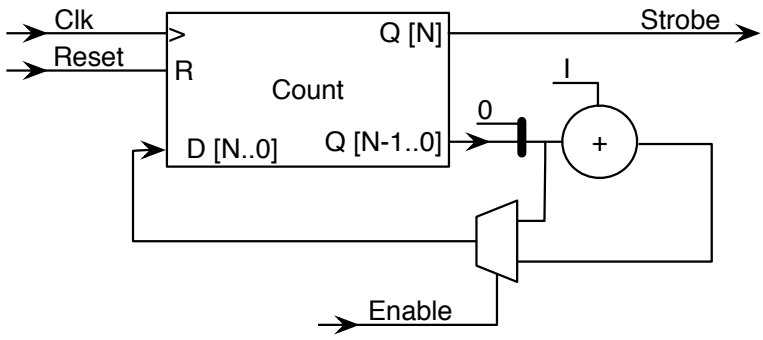

Fig. 10 Counter extended with enable and integer increment.

Our previous counter is now given another generic parameter 'increment' which is used as the second input to the adder (previously just 1). With the introduction of the increment parameter, the ratio of the counter is now given as $r = 2^N/I$. This can result in non-integer ratios which need to be considered in the verification. While the average ratio is given as above, the actual distance between strobe assertions will jitter between two distinct input events, the floor and ceiling of **r**. To accommodate this, we must relax our specifications in [3.2] and [3.4].

$$\forall t\left(\begin{array}{l}\neg reset(t) \wedge enable(t) \wedge incount(t) = H \Rightarrow \\ \quad strobe(t+1)\end{array}\right) \qquad [3.14]$$

$$\forall t\left(\begin{array}{l}strobe(t) \Rightarrow \\ \quad \neg reset(t) \wedge enable(t) \wedge incount(t-1) \geq L\end{array}\right) \qquad [3.15]$$

As *strobe* will now be asserted on one of two values of *incount*, with one being non-deterministic, we must try to specify what we can. We know that if an input event occurs on the high value of *incount*, H, *strobe* must be asserted; but we can't say the same about the low side L. Similarly, when an output event occurs we know that *incount* was either the high or low value but we cannot determine which without much more context. These weaker properties will hold for the counter and it is up to the designers of downstream entities to determine if this jitter is acceptable.

Upsetting the Counter

Verifying that the implementation of a counter is only the first step of ensuring that it operates correctly in real world operation. The anticipated environments and required reliability of an autopilot means we must also consider Single Event Upsets (SEUs). SEUs are memory corruption within a digital device that occurs as the result of high energy particles impacting

a physical bit of memory and delivering enough charge to change the state of that bit. These events are rare, but increase in frequency with altitude [16]. In this section we will consider the impact of SEUs on our counter example as well as ways to mitigate these effects. SEUs and other effects are covered in more detail in "Upsetting Logic".

Our counter has several registers involved in its implementation, N bits to hold the count value and one extra for the overflow that drives the strobe output. These registers are typically implemented as SRAM in the physical device and are susceptible to SEUs. One might also consider the configuration memory of the FPGA, that is the RAM that holds the logical design of the counter and defines its behaviour. It is true that configuration memory is often implemented in SRAM and susceptible to SEUs, however the remainder of this thesis works exclusively with FLASH based FPGAs which do not suffer from SEUs in the same way, so we will not consider configuration memory upsets as a source of faults.

An N-bit counter has N+1 bits that are at risk of SEUs, the result of such an event being that any bit randomly changes its value. In the counter, this results in the count register representing a different value which will alter the duration of the counter period and violate the safety properties defined above. The effect this has on the system overall of course depends on the context in which the counter is instantiated, but this would be considered a complete failure of the counter. There are a few ways to mitigate this risk of failure in our example counter all of which require a redundant representation of the count register.

Triple Redundant Counter

The most generic and robust method of guarding against SEUs is to simply duplicate the registers and monitor them for any differences. These duplicates should be identical and driven by identical, or possibly the same, logic signals. We can then expect the output of these entities to match on every rising system clock edge. A mismatch between any two entity outputs indicates a fault that must be dealt with. Using two redundant copies allows us to detect a fault but we have no way of knowing in which copy the fault occurred. Without knowing which copy is correct it is impossible to gracefully recover from the fault resulting in erroneous output from the entity, either by leaving the fault in place or forcing a synchronisation between the two. In order to gracefully recover from an SEU, we must maintain at least three copies of the registers. Having three copies allows us to implement a majority voting scheme which can be used to identify an individual failed entity and synchronise it with the known

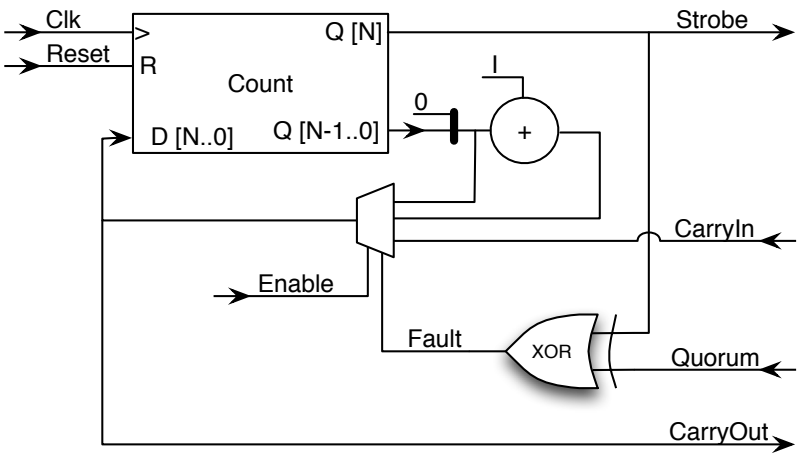

Fig. 11 A counter with facilities to daisy chain redundant counters.

good copies. This can be done such that no erroneous output is produced and downstream entities need never know about the fault that occurred. While triple redundancy will guard completely against a single fault, the occurrence of multiple faults simultaneously must be considered and will be discussed below.

We will now extend our previous counter implementation to provide for redundant representations of the state registers and gracefully recover from faults in any single copy. Fig. 11 illustrates a single counter entity that is intended to be daisy chained with identical copies to provide redundancy. The main idea here is that each counter can compare its own strobe output with the resulting quorum, provided by an external *majority votes* algorithm. If the strobe ever differs from the quorum, then a fault has occurred in this counter's registers. The carryin and carryout ports are provided so that each counter can see its upstream neighbour's value, and provide its own value to its downstream neighbour. In the event of a fault, the faulty counter will copy the next value of its upstream neighbour into its own value register. As we are assuming that only one fault occurs at any time, it doesn't matter which neighbour's value is copied, as all counter values external to the faulty entity should be correct. The topology of a triple redundant counter implementation is illustrated in Fig. 12.

In addition to the counter entities, we must also develop the quorum entity that is responsible for determining what the majority of counter entities think the output should be. In this example, this is almost trivial as the strobe outputs are only a single bit. The quorum entity simply needs to implement

$$\text{quorum} = (A \wedge B) \vee (B \wedge C) \vee (A \wedge C) \qquad [3.16]$$

This logically combines the inputs to a single output that represents what any two counters agree on. As this output is fed back to the counters, any counter that disagrees will recognise the fault and correct itself to match the others on the next clock cycle.

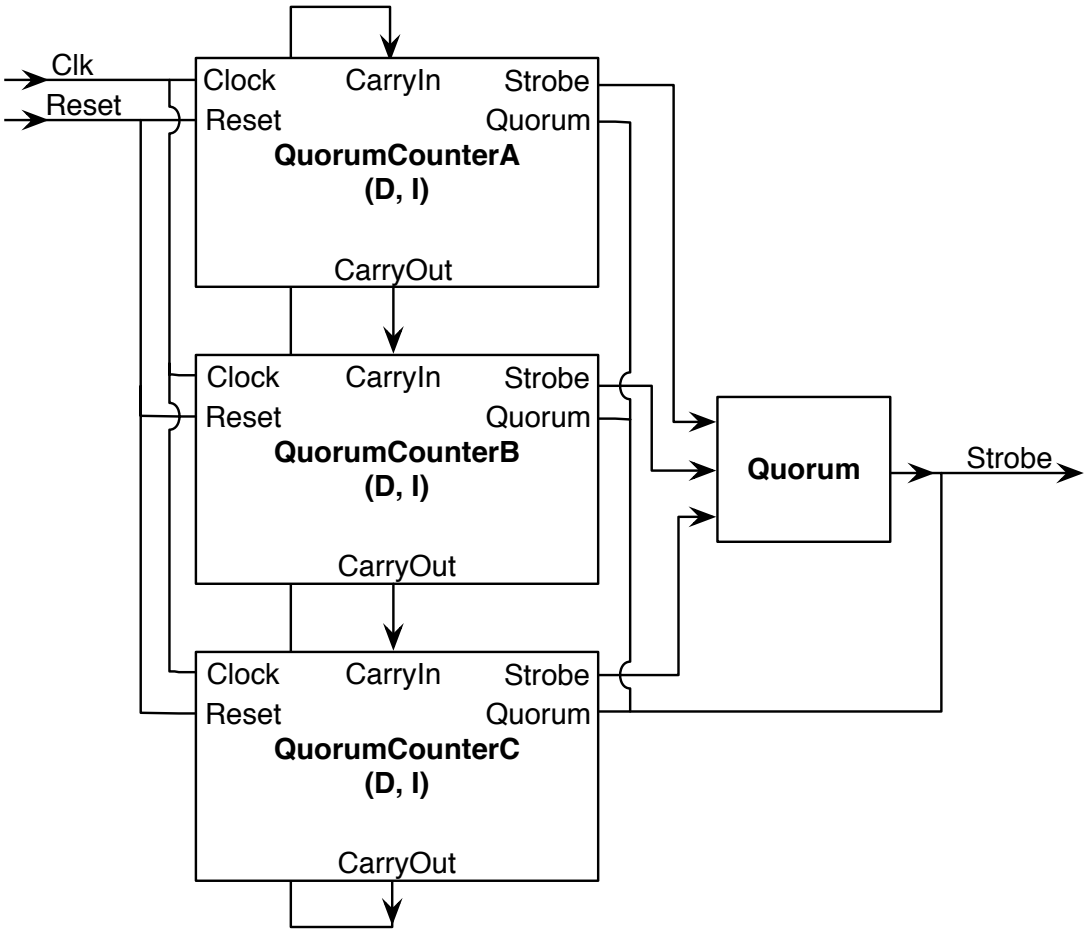

Fig. 12 An example of daisy chained redundant counters.

It is important to note that many synthesizers will detect identical logic, that is entities with the same inputs and the same behaviour, and attempt to optimize them into a single instance. This of course removes the redundant registers and the mitigation of SEUs. This can be avoided with vendor specific methods, but must be checked in the implemented design. A complete implementation of the triple redundant counter is available in Appendix A.

While this implementation completely guards against a single fault, it does take some time to correct it as the fault condition is only apparent when at least one counter asserts its strobe at the end of its period. While this may be the faulty counter, the worst case duration for a fault is up to an entire counter period. This time window increases the chances of a second fault occurring, and given the arbitrary length of the counter period, can do so quite substantially.

If two faults occur in the same counter then nothing exceptional happens, the single counter will still disagree with the quorum and correct itself at the end of the next period. If two faults occur in different counters then we no longer have a quorum and an erroneous output will be produced. The counters will however recover to create a new quorum, albeit incorrect. This is the same result as expected in a single, non-redundant counter. This scenario can be mitigated by extending the daisy chain of redundant counters to include more entities. By simply adding more counters and extending the quorum to combine five inputs, the counter can completely guard against two simultaneous faults. This design can guard against two faults by only using four counters if the designer can accept the risk that the two simultaneous faults do not produce the same output on two counters, i.e. a false quorum. This chain can be extended arbitrarily to guard against however many faults the designer feels necessary. The only limit is the timing on the combinatorial path through the MUX from each counter's carryin/carryout ports.

If a single N-bit counter has an expected SEU rate of r, then the expected rate of a triple redundant counter experiencing two simultaneous SEUs can be expressed as the rate of three individual counters times the probability that a second SEU occurs in one of the other counters within the period of the counter before the fault is corrected.

$$\begin{aligned} r_3 &= 3r \times 2r \times \tfrac{p}{3600} \\ &= 6r^2 p \times 3600^{-1} \end{aligned} \qquad [3.17]$$

Where p is the period of the counter. As an example, a triple redundant counter dividing a 12.5MHz system clock down to 50Hz using 24 bits has an approximate reduction in error rates of 2×10^{12} when compared to an individual counter. This shows that the triple redundant counter provides a substantial decrease in the rate of erroneous outputs as a result of SEUs.

A Counter Example

This chapter so far has laid the groundwork to demonstrate the value of designing with FPGAs in systems where formal verification and SEU tolerance are required. Traditionally, CPUs are employed to implement behavioural algorithms in software, given the ease of design and abundance of resources. It's only fair to provide an example implementation and discussion of the verification of a functionally similar entity implemented in a software context.

While a typical microprocessor includes timer peripherals which provide the same benefits as the hardware implementation above (because it is a hardware implementation) we will be focusing on the characteristics of a software implementation. This is not an example of effective design, but does aim to illustrate the difference between the verification of a simple entity in hardware versus a comparable implementation in software, and that this contrast will scale to less contrived examples.

Consider the simple program in assembly in Fig. 13 that generates a pulse at regular, deterministic intervals just as the VHDL counters above.

```
loop:       MOV [D], 0x01         ;output high
            MOV [D], 0x00         ;output low
            MOV A, 10             ;set timer duration
timer:      DEC A                 ;loop till 0
            JNZ timer
            JMP loop
```

Fig. 13 A simple software timer.

The interval is timed by a simple loop that decrements register A until it reaches zero. The CPU then jumps to the output instructions which outputs a high signal for one CPU cycle and then restarts the timer loop. The value loaded into register A controls the duration of the timer in addition to the overhead of pulsing the output and restarting the loop. In this simple example, we can derive the output interval as 2xN+4 CPU clock cycles where N is the value loaded in to A. Of course it's possible to rearrange various instructions to coerce this loop to provide any interval required.

This simple example can indeed be verified, and proven to always meet a deterministic interval much the same as the hardware counters before. However, this does require some significant assumptions. We assume that each instruction only takes one CPU cycle to execute, which is not typically the case in real microprocessors. There are many considerations to make regarding how this program is written as any modification to the output stage or trying to modify the duration of the timer will likely require another iteration of counting instructions. If this program is written in a higher level language that requires compiling, then we must also consider the compiler and likely inspect its output to ensure we meet the required interval.

Let's now consider a CPU which has a single timer that can be used to trigger an interrupt at regular intervals, but can't directly drive an output. In this case, we can configure the timer to divide the CPU clock by any integer. Each time the timer loops it will trigger an interrupt

that will break execution of whatever the CPU was doing, and then execute the interrupt handler which will provide the output before returning and waiting for the next interrupt. This leaves the CPU free to execute other tasks in between output events and leaves the hard part of the accurate interval timing to the dedicated timer hardware.

This seems like an almost ideal solution with only a small amount of consideration of the ISR required to provide output pulses at a deterministic and regular interval. Unfortunately, most interrupt circuits are not as deterministic as we would hope, often taking several clock cycles between the interrupt event and the execution of the ISR. This is often dependant on what the CPU was doing at the time and the specific timing of the event with respect to the system clock. This of course adds some jitter to our output, but should otherwise maintain a regular interval.

Things become much more complex if we consider a CPU that is running many tasks involving separate interrupts. Each ISR can delay future interrupts, or interrupts can interrupt ISRs depending on each implementation. This means that in the presence of other interrupt events, our timer can be arbitrarily delayed by another ISR. This can lead to an increase in output jitter, errors in the period of the timer or errors in the width of the output pulse. We need to be confident that the implementation of the interrupt handler being invoked does not modify memory related to our timer, such as the A variable; and that it returns the stack and program counters to where they need to be. If we were to try and verify such a system we would need to know the detailed implementation of every other task running on the CPU even if they are completely unrelated to our timer. This is where software verification complexity begins to explode. Adding other modern technologies such as out of order execution, caching, frequency scaling and multi-core architectures; the practice of verifying a program on a particular CPU can become an exceedingly complex endeavour.

Conclusion

Using an example of a simple counter, we have illustrated its specification, implementation and verification on a FPGA, including considerations for SEUs. The verification of algorithms implemented on FPGAs is simpler than similar implementations in software. As an FPGA implementation is literally a logical design mapped to silicon, the translation to first order logic which can be automatically analysed with a theorem prover is almost trivial. The verification of software on the other hand can be quite complex as the machinery of the CPU

must also be considered. The main advantage that FPGA implementations have is that they do not need to consider any external factors beyond their own interface specification, where as software algorithms must be aware of any other task with which they share a CPU.

While the examples in this chapter may seem trivial in the context of an unmanned aircraft, these concepts easily scale and map to new algorithms as will be demonstrated in the remainder of this thesis.

Upsetting Logic

Our previous discussion on formal verification of logic designs should provide complete confidence that a design will function as expected when appropriately implemented on a suitable FPGA. However, most designers using those techniques will be upset to learn of phenomena occurring in the physical world that can cause faults outside of the rational domain of their formal methods.

Single Event Effects (SEEs) are a collection of faults that can occur in a digital system as the result of ionizing radiation such as high energy neutrons or alpha particles. This radiation is typically the results of cosmic rays interacting with matter in the atmosphere. These faults range from transient levels in logic circuits, charge accumulation, latch-up and possibly physical damage to the silicon itself. These effects increase with logic density, altitude & latitude to a point where all modern avionics should take measures to mitigate their effects.

While mitigation of SEEs starts with the physical hardware and silicon design, some of these effects cannot be mitigated in hardware alone and require the firmware to tolerate faults caused by SEEs. Here I will discuss the effects of SEEs on FPGA designs in the context of avionics before exploring methods of mitigating these faults in the firmware.

Introduction

In recent decades, avionics in commercial & military aviation as well as space craft have increased their dependence on small scale digital systems in the form of microprocessors, memory arrays and programmable logic devices. These devices contain many transistors on silicon dies arranged to perform computations and store information. As these silicon gates shrink in size, the power required to alter their state also shrinks, much to the approval of engineers wishing to lower the power requirements and heat generated by these devices. However this has brought the energy required into the scale in which various ionizing radiation can trigger state changes in a digital system [47] [48]. This shrinking scale combined with an increasing density of digital systems means mitigation of Single Event Effects (SEEs) is required for any modern avionics system.

In the early 1990s a comprehensive study was performed [11] which quantified the expected number of SEEs in select SRAM components as a result of the natural radiation environment of the Earth's atmosphere from ground level up to space shuttle flights. They found neutron radiation and its secondary reaction products interacting with the silicon die to be the primary cause of SEEs. They recommended that designers of all avionics systems consider the impact of SEEs and outlined some mitigation and hardening techniques.

Atmospheric Neutron Environment

Cosmic rays are the main source of radiation in the atmosphere . Cosmic rays, and their reaction products with oxygen and nitrogen atoms in the atmosphere result in a radiation environment consisting of neutrons, protons, electrons, muons, pions, heavy ions and others [11]. Studies have shown that it is neutrons that are most significant at producing SEEs at aircraft altitudes [12].

Neutron flux in the atmosphere varies with both altitude and latitude [16]. The Earth's magnetic field shields lower energy cosmic rays close to the equator, thus the neutron flux at the poles is significantly greater. As cosmic rays pass through the atmosphere they react with oxygen and nitrogen atoms dispersing other particles, which in turn react with other atoms. This process leads to a peak neutron flux around 60,000 ft, with the neutron flux at sea level

being hundreds of times smaller. Taber & Normand [11] provide a model of the neutron flux in the atmosphere with the goal of estimating SEE rates in digital avionics.

$$\text{Upset Rate} = \int_{E_n} dN/dE\,\sigma_{nseu}(E_n)\,dE \qquad [4.1]$$

Where

dN/dE is the differential neutron flux spectrum.

$\sigma_{nseu}(E_n)$ is the SEU cross-section at neutron energy E_n.

Single Event Effects

Single Event Effects are categorised based on the effect they have on the circuit they are affecting. While they are all caused by the same high energy particles, their effects vary significantly depending on where exactly they impact. I will briefly describe the three main categories of SEE here and note that only SEUs are of concern in the resulting platform design.

Single Event Transients (SET)

Energy can be deposited on any part of a circuit and does not need an n-p junction to have an effect. When energy is deposited on to a wire or some combinatorial logic path it can erroneously change the level of that path, temporarily propagating the error through the circuit [50]. In a completely synchronous design this is unlikely to have an effect, as the error condition would need to be maintained for long enough around the receiving register's next clock edge to corrupt the data being latched, resulting in an SEU. In asynchronous designs this can be more of a problem if the erroneous level is maintained long enough to trigger an action by downstream logic, such as an asynchronous reset or interrupt.

Single Event Upsets (SEU)

When a particle delivers energy to just the right location in a flip-flop it can bias one junction just enough to cause the flip-flop to change its state [11] [12]. This is called a Single Event Upset and is the main concern for developing any system which includes a large amount of SRAM and is operating at significant altitude. SEUs can occur in any SRAM cell either in large banks of bulk RAM, or in trickier places like CPU registers or external device configuration registers. In the case of FPGAs, SEUs can occur in any flip-flop scattered around the fabric, any block RAM device, or even in the configuration of the FPGA itself. In the

case of SEUs affecting the configuration of the FPGA it is difficult to correct the errors as the behaviour of the design is itself corrupted, requiring an external processor & memory element to correct the error. Configuration errors require mitigation in the hardware by either implementing redundancy on the silicon or by using a FLASH medium to store and recover the configuration.

Single Event Latch-up (SEL)

The most dangerous category of SEE is the Single Event Latch-up as it cannot be corrected without removing power from the affected circuit, and it can cause permanent damage [14]. SELs occur when a high energy particle inadvertently triggers the formation of a parasitic structure in a flip-flop that provides a low-impedance path between the power rails. This effect must be mitigated by the design of the silicon fabric itself.

Expected Upset Rates

In later chapters I will discuss the design of an avionics platform built around the M2S025 from Microsemi. The M2S025 System-on-Chip (SoC) which provides a CPU and FPGA fabric on the same silicon die. The CPU, FPGA and RAM are all susceptible to SEEs and mitigation strategies will be described. Here I estimate the likely upset rates due to SEEs in the M2S025 during my experimental application.

The model from Taber & Normand [11] above requires the SEU neutron cross section to be known for the device in question, which typically requires the device to undergo neutron beam testing, which is not common practice for hardware vendors. Luckily for us, Microsemi has done just that and provides detailed results [52] including neutron cross sections for various SEEs and silicon elements in the M2S device family as well as expected failures in time (FIT) for devices operating in New York City.

The M2S025 includes both a CPU and FPGA, however only the FPGA will be considered as the CPU is not going to be used for any safety critical tasks in our design. The FPGA contains a few different types of functional and storage elements, each type with its own SEU cross section. The main elements we are concerned about are the storage elements: the flip-flops on each logic cell and in each Math block, and the bulk RAM elements. As the FPGA fabric is implemented in FLASH we don't need to consider upsets to configuration memory [53].

Table 1 summarises the elements in the FPGA of an M2S025 that we must consider with respect to SEUs. Microsemi had a number of their SmartFusion2 devices tested at the LANSCE facility to determine the Failures in Time (FIT) of their various elements. As the M2S025 was not explicitly tested, I assume the same characteristics as the M2S050. The testing also misses details on the FFs within the Math blocks. From the data sheet [51] I assume each Math block contains 130 FFs. I also assume that they have the same SEU cross section as FFs in the main FPGA fabric as they are not explicitly listed in the test report. The test report does not detail the spectrum of neutron energy applied during the testing, but the LANSCE facility is capable of neutron energy levels from 0.1 MeV up to 600 MeV which easily covers the range I expect in normal atmospheric conditions. I assume the spectra to be similar to real world conditions. Finally, as mentioned above, the neutron flux varies significantly with altitude and latitude. Taber & Normand normalised their models around operations at 40,000 feet and 45° latitude, at which point they estimate a neutron flux of 0.85 $n/cm^2 \cdot sec$.

Element Type	**# of Elements**	**SEU Cross-section**	**SEU per Hour**
Flip-Flop	27696	1.82E-14	1.54E-6
Flip-Flop (Math Block)	34×130	1.82E-14[1]	2.46E-7
LSRAM	31×18432	2.53E-14	4.42E-5
uSRAM	34×1152	1.31E-14	1.57E-6

Table 1 Summary of SEU susceptible elements in an M2S025 FPGA at 40,000ft.
1. Assumed the same as Slice Flip-Flop.

The aggregate estimate of SEU per hour in a fully utilized M2S025 FPGA is 4.76E-5 upsets/hour or 21,028 hours between failures. Table 1 also shows that the most significant portion of these failures is likely to occur within the bulk storage arrays, LSRAM.

We can calculate this same upset rate for the processors used in other, similar flight controller designs. The OcPoc [39] and Phenix Pro [40] both use devices from the Zynq family, though the OcPoc include a Z7010, while the PhenixPro uses the larger Z7020. The Zynq family proved difficult to find detailed SEU cross sections divided by element type. Using the SEU cross section of 6E-15 as measured by [54] we can estimate the expected upsets per hour of the FPGA fabric. We only consider the data elements and not the configuration memory of the FPGA in the Zynq to try to compare the equivalent value of the M2S025. We also include the Z7007S (the smallest of the Zynq family) as it is the most similar in size to the M2S025.

Device	# of Elements	SEU Cross-section	SEU per Hour
M2S025	6.43E5	2.42E-14[2]	4.76E-5
Z7007S	1.88E6	6E-15	3.45E-5
Z7010	2.71E6	6E-15	4.98E-5
Z7020	5.30E6	6E-15	9.73E-5

Table 2 Comparison of SEU rates between SoC FPGAs.
2. Weighted average for all element types.

Here we use the total number of elements in the FPGA of the SoC that are susceptible to SEUs, not including configuration memory which would significantly increase the expected rate.

The Pixhawk [36] is built around an STM32F427 which does not include programmable logic like the other processor families discussed so far. However the Pixhawk is the most similar in processor capability to our design, so for the sake of comparison we can compare the expected upset rates within the CPU. As an entire flight controller can be implemented within the On Chip Memory (OCM) of the STM32 we will not include external memory devices such a DDR RAM. We will also not include the many SRAM elements within the CPU itself such as system control registers as these are overly complicated for this comparison. Using the same SEU cross section values as above as well as the value for the STM32 as measured by [55] we can compare the expected SEU rates in the OCM of the three processor families.

Device	OCM Size (KB)	SEU Cross-section	SEU per Hour
M2S025	80	2.42E-14[2]	4.9E-5
Zynq	256	6E-15	3.9E-5
STM32	256	2E-14	1.3E-4

Table 3 Comparison of SEU rates between CPU OCMs.

It can be seen from these expected SEU rates that the Zynq family benefits from a reduced SEU cross section but it does present a much larger surface area given the device sizes. The Zynq also requires protection against faults in its configuration memory. The M2S025 and STM32 are similar with their expected upset rates, but it should be noted the that the M2S025 implements ECC protection on its OCM (at the cost of some space).

SEEs in Peripheral devices

It should be noted that the FPGA does not operate in isolation, but in a system that includes several other devices for sensing, actuation and external storage/communications. Even

switch-mode power supplies and oscillators contain elements susceptible to SEEs [17] [56] [57] [61].

Oscillators

Oscillators provide timing references and drive the synchronous process in computation. Erroneous oscillators may produce transient glitches which may corrupt data in synchronous processes by violating setup & hold times of registers, or produce an incorrect frequency reference which will violate assumptions made by the rest of the system. It is obvious that the frequency references provided must be robust against environmental effects including vibration, temperature and SEEs.

There are three common technologies available to provide frequency references in small avionics systems. These are crystal oscillators, RC oscillators and MEMS oscillators. Each of these have their own advantages and disadvantages. The basic concept is similar for each technology, a filter combines with an amplifier such that a positive feedback results and the circuit resonates. The filter (also called the resonator) is the main difference in these three technologies and is responsible for each of their characteristics.

RC oscillators use a network of capacitors and resistors to form a filter with the correct phase shift at the desired frequency. These tend to be the least accurate and stable references as they have significant temperature coefficients in their filter elements.

Crystal (or ceramic) oscillators use a piece of piezoelectric material as their resonating element, most commonly quartz crystal. These are much more stable over temperature and are orders of magnitude more accurate than RC oscillators. Many studies have investigated the transient [17] and aging [56] effects of radiation of quartz resonators with regard to space bound systems. The transient effects of radiation on crystal resonators tends to be limited by the thermal effects of the energy deposition [56], while the ageing effects of radiation are insignificant within the atmosphere.

MEMS oscillators are similar to crystal oscillators but they use a MEMS element as their resonator. MEMS oscillators are more resistant to shock and vibration and their more controlled manufacturing process can make them more accurate than crystal oscillators. As the entire oscillator circuit can be etched onto a single silicon die, manufactures have opted to make the reference frequency programmable at the factory to allow mass production of many different frequency references using the same silicon design. As demonstrated in [57] the

most vulnerable part of the MEMS oscillator to SEEs is the configuration registers used to set the output frequency. Any corruption to the registers in this region results in a persistent error to the output frequency and requires a power cycle to restore the device to its expected behaviour.

All three of these oscillator technologies still require an amplifier that is typically implemented in silicon either as transistors or an op-amp. These components are susceptible to SEEs as shown in [59]. SEEs may produce short pulses in amplifier current with an exponential settling time dependant on the design bandwidth of the amplifier. As the resonator acts as a very narrow band-pass filter, these pulses are only likely to produce small errors in amplitude which would not be noticeable to a digital circuit. Any small pulses on the output of the oscillator are unlikely to overcome the trace impedance to provide a wide enough voltage swing to produce a false clock edge.

Switch-mode power supplies

Modern processing systems often require several different voltages among their components which is usually achieved with DC-DC power converters. The selection of topology is important regarding SEEs as some are susceptible to SELs [58]. Further studies [61] have investigated SEEs on PWM controllers which are typically used in switch-mode power supplies and found some effects including phase shifts and amplitude errors that would increase voltage ripple at the output of the converter; additionally that converters with a soft-start feature could be upset into restarting their soft-start timer, leading to a brown out of the voltage rail for significant time.

Low dropout regulators (LDO) are alternatives to switch mode power supplies and their operation is much simpler. LDOs require fewer external components and they don't produce the same EMI that switch mode regulators do. The disadvantage of LDOs is their overall efficiency, as the power difference between the input and the output is essentially 'burnt' in the pass transistor. LDOs are less susceptible to SEE as they don't involve any digital logic so any transients are absorbed by the analogue circuity.

Sensors and Actuators

Gone are the days of cumbersome mechanical gyroscopes used to sense an aircraft's orientation. These have been replaced by MEMS sensors which implement the gyro sensor on a silicon die. A typical aircraft will include sensors for 3D acceleration and rotation as well as

a compass, airspeed indicator and altimeter. Many sensors for each of these categories are available commercially and have been developed and proven effective over many years. However these sensors are not immune from environmental effects including temperature and radiation. While most of these devices can compensate for temperature variations, SEEs are not usually explored as these devices are typically used in less-than-critical applications, and not far above sea level. These sensors often contain a number of RAM based elements including configuration registers, data buffers and in some cases, entire microprocessors. While we do not have access to the internal workings of these devices in order to improve their radiation tolerance we must consider ways of mitigating this risk if we are to include them in our system.

Mitigation Strategies

Mitigating SEEs in an FPGA design is a combination of design choices and active coding strategies. Some components are inherently immune to radiation effects, while others must be constantly monitored for the correct behaviour. This section will focus on the coding strategies involved in monitoring and correcting components within the FPGA as well as external components that the FPGA has enough control over. With these strategies in place and effective design decisions regarding the remaining hardware components, it is hoped that the negative effects of SEEs will be significantly reduced during the operation of the resulting flight controller.

The theory behind SEE mitigation in a digital system is simple enough, but these techniques do come at a cost of resource usage or system throughput. We will compare methods and discuss the trade-offs in terms of circuit timing and logic resource usage to inform the design using realistic and cost-effective parts that are neither infinitely large nor fast.

Error Correcting Codes

Error correcting codes provide a method of encoding data with redundancy that allows errors to be detected and corrected. They have a long history in computer science and are employed on most communications networks and in reliable data storage [30]. In the context of FPGA avionics, there are two cases that can benefit from error correcting codes to make the system tolerant of SEEs. Data stored in the device, either in RAM or in fabric registers is susceptible to corruption from high energy particles, more so than transient effects as the

duration that data is stored in these elements could be indefinite. As described above, SEUs result in a memory element altering its value. Single bit errors are rare, but common enough to trouble any aircraft with an endurance measured in hours or more, and especially for any safety critical system. Two or more bit errors are much less common. Multi-bit errors caused by a single event are exceptionally rare but have been detected in high density memory elements such as RAM [11]. Multi-bit errors are more likely to occur the longer pre-existing errors are allowed to persist in memory as the result of multiple SEEs. Physical separation and high frequency "scrubbing" of memory elements mitigates most of the risk of multi-bit errors. Single bit errors are still possible regardless of data longevity and can occur in even the most rapidly accessed register. To tolerate single bit errors we must apply error correcting codes to every register in the design. This requires considerable resources in the FPGA that could otherwise be used for the application, so it is desirable to keep this implementation as efficient as possible.

Here we consider three circuits that implement single bit error correction and potentially multi-bit error detection. These three circuits vary in how many data bits they encode as well as how many Flip-Flops (FF) and Look-Up-Tables (LUT) are required to implement them. The ratio of these two resources to the number of data bits will be the main criteria to decide which is most appropriate in which design. Flip-Flop usage is analogous to the 'rate' of an error correcting code, that is how many bits are added to the data bits to form the codeword. LUT usage is an indication of the complexity of the algorithm required to implement the error correction. We must also consider the performance of the algorithm, which can be measured in the additional propagation delay through LUTs required for error correction, which combined with application logic will limit the speed of the resulting design.

FPGAs from different vendors or different product lines have varying architectures on which to implement designs, which does affect how an algorithm can be optimised. The main fabric of the FPGA is divided into logic slices which typically include a small LUT and a FF or two. The number of inputs to a LUT limits the complexity of logic that can be performed in a single LUT. For the avionics platform described later in this thesis we use the Igloo2 fabric [60] from Microsemi which uses 4 input LUTs (4LUT) with one Flip-Flop per slice. The actual resource usage in a given design also varies as the synthesiser optimizes and combines logic patterns that may be spread over several combinatorial LUTs.

Simple Repetition

The first and simplest circuit to consider is simply repeating the data bit and storing it in multiple FFs, commonly referred to as Triple Modular Redundancy (TMR) [15]. A very simple 'majority wins' circuit can correct any single bit error in the FFs.

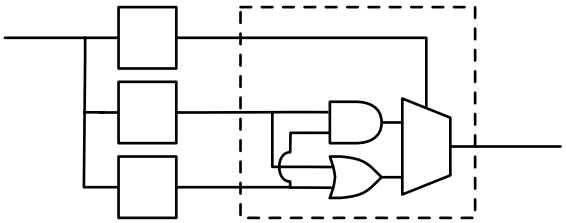

Fig. 14 Simple repetition of critical register with majority voting circuit.

This circuit requires 1 3LUT and 3 FFs for each data bit encoded and the delay increase is a constant 1 LUT extra. This simple circuit provides a (3, 1, 3) codeword. Care must be taken with the synthesizer as most will "optimize" this circuit by removing the redundant logic.

Semi-repetition

Designed in an attempt to optimally utilize the 4LUT required in the simple repetition circuit, this design repeats each data bit twice and then stores a parity bit of two neighbouring data bits. This allows us to use 5 FFs and 3 LUTs for every 2 data bits; essentially trading a FF for a LUT when compared with the simple repetition circuit. The delay increase is still constant but increased to 2 LUTs.

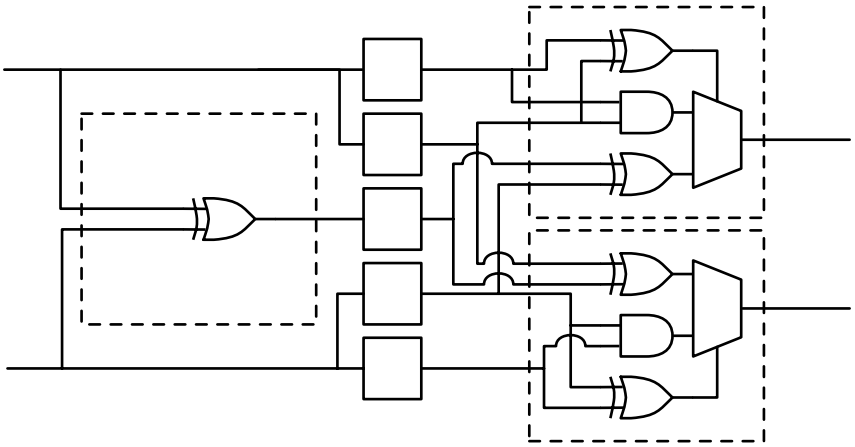

Fig. 15 Protecting two critical registers with a shared parity bit.

This circuit works by comparing the repeated data bits, if they are the same then that value is output. If they differ, then assume the parity bit and one data bit from the neighbour are

correct (we only expect a single error out of the 5 bits) and then generate the correct data bit. This circuit gives us a (5, 2, 3) codeword.

In low frequency designs such as the avionics being developed for this thesis, it is typical to utilise more LUTs than FFs in the FPGA as longer arithmetic paths can still meet timing requirements while using less FFs. This circuit allows us to shift the balance from FFs to LUTs if required.

Hamming Codes

Hamming codes [30] [31] provide a simple and scalable encoding scheme for arbitrary word lengths that provides single bit error correction, and with extension, double error detection. The number of FFs required scales logarithmically with data length, however the LUTs required scales exponentially. Similarly the delay increases with word length so this algorithm is likely impractical for large word lengths. Linear combinations of (7, 4, 3) and (6, 3, 3) hamming code implementations can be used to encode larger data words and balance the requirements of FF, LUTs and path delay.

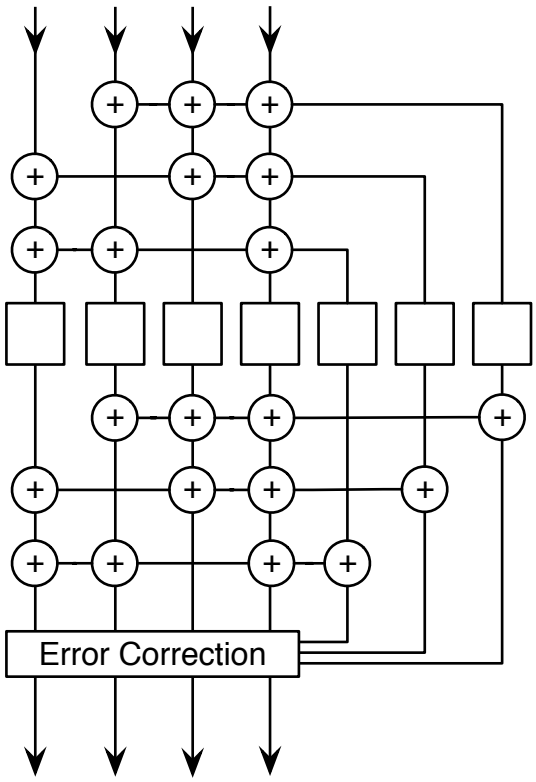

Fig. 16 Illustration of ECC implementation before and after critical registers.

Auto Scrubbing

The assumption that multi-bit errors are rare enough not to require correcting relies heavily on the idea that any single bit errors are removed in a short enough period that it is unlikely another single bit error occurs within the same codeword. In order to achieve this, each codeword must be refreshed at a regular interval either by loading a new codeword into it, or by loading the error corrected version of the previous codeword. The latter process is known as 'scrubbing'. If a register is updated with new data at a high enough frequency, scrubbing is unnecessary. However many registers in a design will not have new data regularly or will be updated too slowly. In these cases, the error correcting circuit can be configured to automatically scrub the codeword in the registers by looping the error corrected data back into the input. This requires more LUTs to implement and will probably increase power consumption as the FFs are being enabled more frequently than they otherwise would be.

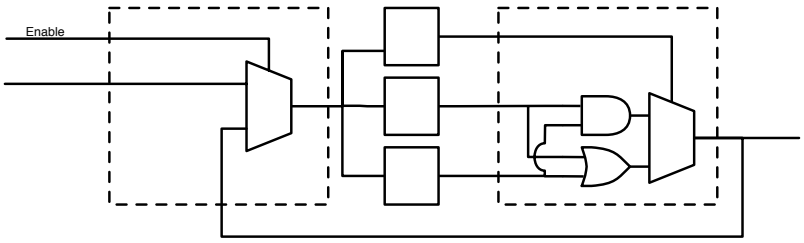

Fig. 17 Triple redundant register with voting circuit and auto-scrubbing.

Encoding State Machines

All structures utilising flip-flops in the FPGA fabric are susceptible to single event effects and must include some form of mitigation. While the vast majority of flip flops will be used in a data path, control structures such as state machines must also be considered.

A typical state machine implementation consists of two main components; the state value register which holds the current state, and the combinatorial 'next state' logic which transforms the current state and a collection of inputs into the value of the state on the next clock cycle.

Designers will usually use an enumerated type to define the state variable rather than trying to directly implement it as a bit vector, as all data eventually becomes. The synthesizer maps these state values to a specific encoding. Modern synthesizers are capable of detecting the state machine and optimizing it before selecting a state encoding method that best suits a particular scenario. These encoding methods include:

Binary which simply represents the state value as a collection of binary encoded integers. This provides the simplest and most compact encoding scheme.

Grey-code which tries to minimize output glitches from the state machine by only toggling one 'bit' of the encoded value on each transition. This only works if the state transitions through consecutive integer values.

One-hot/One-cold encodes the state variable in a vector of as many bits as there are state values, with the current state being whichever bit is set/not set. This provides the highest performance encoding as no logic is required to resolve a number of bits to a particular state value.

With respect to SEE mitigation, the encoding scheme doesn't actually have much effect if the entire bit vector is protected with ECC. The main concern here is that the state variable must be explicitly encoded as protect-able data type such as an integer or bit vector.

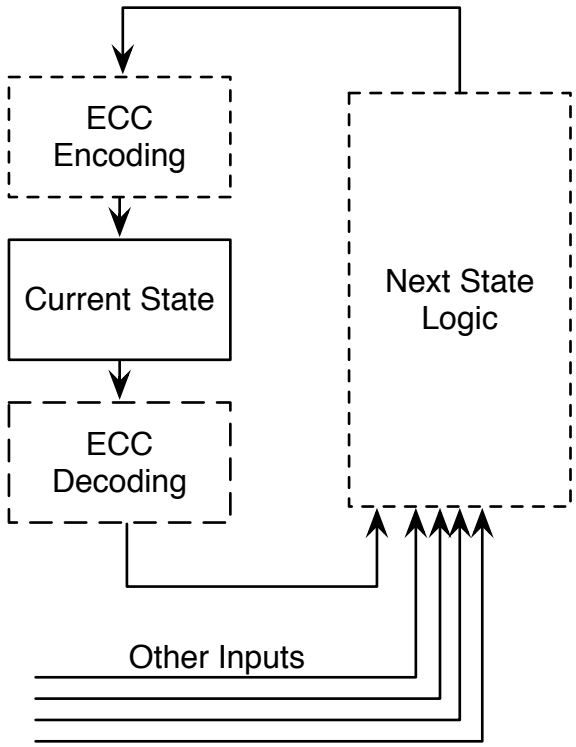

Fig. 18 ECC encoding of state machine. Combinatorial components in dashed lines.

Block Level Mitigation

The discussion of SEE mitigation has so far focused on algorithms implemented in the logic elements of the FPGA (LUTs and FFs). However, modern FPGAs include a number of other resources for data storage or high performance signal processing which we must also address.

While these resources vary from vendor to vendor, three distinct types of resources are usually present in the fabric. These include Clock Conditioning Circuits, RAM and Math blocks. While each vendor gives them a different name, and the design & capabilities differ, these are found scattered around the FPGA fabric amongst the logic resource and are available for designers to use in their algorithms. They are also susceptible to single event effects.

Math Blocks

Math blocks are hardware implementations of some arithmetic operations that can operate at much higher frequencies than their slice logic equivalents. While each FPGA vendor has there own design for the math block they provide, they typically include a multiplier, accumulator and a number of optional pipeline registers to improve performance. These blocks are very useful in digital signal processing as just a simple multiplication operation can consume a significant number of LUTs and be difficult to meet performance requirements.

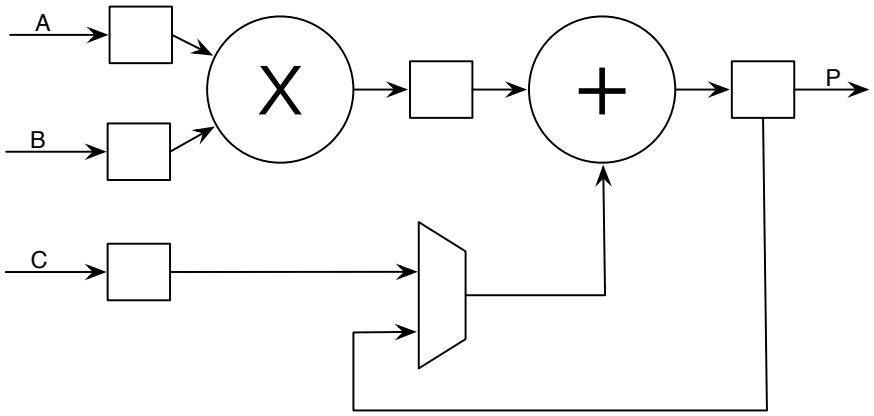

Fig. 19 A simple MATH/DSP block implementation.

While the maths blocks are less susceptible to configuration corruption, their internal registers are still a cause for concern in radiation hardening a design. These registers are likely identical to the FFs in the main FPGA fabric and would be just as susceptible to corruption from SEEs. However, the FFs in the main fabric could be protected by adding error correcting bits, this is not possible with the math block's internal registers as their inputs are not available to route through LUTs that can encode ECC bits. Unfortunately, short of hardware hardening implemented by the vendor (which some vendors do) there is no way to mitigate SEE effects in these internal registers. Instead, we must either limit the functionality of the math block to not use these registers, which may be possible with relatively slow clock speeds, or provide redundancy either spatially or temporally.

Redundancy can be provided for Math blocks (as well as other block level entities) in two ways. If speed is of concern, then replicating the block 3 times will allow maximum throughput and protection against any errors within the block's internal registers. A voting circuit on the output of the replicated blocks combines the outputs. Alternatively, if resource consumption is more of a concern than speed, a single block can be used and simply perform the same operation over three consecutive clock cycles. The output of the block can feed a shift register within a voting circuit that can combine the results of the three operations.

Block RAMS

Small blocks of RAM are also found scattered around the FPGA fabric that allow storage of bulk data, readily accessible to fabric logic for various purposes. These can be used to store data such as buffering packet data in a FIFO, or as part of the system algorithms by storing a lookup table that can resolve a complex function in a single clock cycle. However they are used, BRAMs are susceptible to the effects of SEEs; more so than FFs as they typically consist of much higher density memory that holds its value for longer periods of time.

Typical BRAMs provide two asynchronous read/write ports that allow the memory to be accessed simultaneously by two separate processes, which is convenient for memory scrubbing applications. The two use cases for BRAMs have very different integrity checking requirements. In the case of static lookup tables, the RAM contents is not altered by the application and is likely loaded from persistent, SEE immune storage at system start. In this instance a checksum for the RAM contents can be provided along with the data and frequently recalculated. When a fault is detected, the entire RAM contents can be reloaded from persistent storage. As most of the data being rewritten is not actually changing, this can usually be done without interrupting application processes accessing the other port.

The processes of recalculating the checksum requires a loop over every data word in the RAM which takes a significantly longer period than single clock cycle ECC calculations previously discussed. This means that there is a much greater opportunity for SEEs to occur between scrubbing periods, increasing the possibility of multi-bit errors.

In the case where a BRAM is used to store dynamic application data, there is no persistent storage to compare with. In this case, a proportion of the RAM can be dedicated to ECC bits for each word (or possibly another BRAM entirely). RAM scrubbing can then be achieved by simply reading out each word and correcting it according to its ECC bits, then rewriting it to the RAM. This can also be done transparently to application processes on the second

port. Again, we must consider the rate at which a scrubbing process can loop over the RAM, as we can only read and correct one data word every two clock cycles.

Expected Upset Rates with Mitigation

Now that we have detailed the mitigation strategies for SEEs affecting various elements within the M2S025 we will now quantify the improved update rates. We no longer consider single bit upsets, as the mitigation strategies detailed above effectively correct any SEU affecting only a single bit so no single bit errors will propagate into the system. Now we consider two bit errors that cannot be corrected by the techniques above. As the fabric elements are divided up into many code words, only two bit errors affecting a single code word will result in errors propagating forward. Multiple errors in different codewords will each be corrected. We don't consider single SEUs that result in multiple errors within a codeword as this was not observed during neutron testing of the M2S050 and is not considered likely. This leaves only the effect of multiple SEUs affecting the same code word before corrections can be applied; that is within the scrubbing period of that codeword.

We divide the 27696 flip-flops in the M2S025 into 11 bit codewords, each including 7 bits of data and 4 bits of ECC. This ratio varies with specific data word lengths as does the resulting codeword 2SEU cross section. 7 bit data words are expected to be a reasonable mean. We assume these codewords are scrubbed on every clock cycle, for which we assume a 20MHz clock. The resulting codeword 2SEU cross section is calculated by

$$\delta_{2SEU} = \left(\frac{\delta_{bit} \times L}{f_{scrub}} \right)^2 \qquad [4.2]$$

Where δ_{bit} is the SEU cross section of the bits within a codeword, L is the length of the codeword in bits including data and ECC bits and f_{scrub} is the scrubbing frequency of the codeword. It should be mentioned that SEUs affecting the same bit within a codeword do not result in a two bit error, in fact the second is correcting the first. This reduces the cross section of the codeword to the second SEU by one bit, but we don't include that here for simplicity.

We divide the other fabric elements into codewords in a similar way. The Math Blocks are considered an entire codeword of 130 bits, and each operation is repeated 3 times on consecutive clock cycles to provide redundant data. The BRAMS are each considered an entire

codeword that is scrubbed at 20MHz/depth with a checksum that has a hamming distance of at least 2.

Element Type	# Elements	δ_{bit}	L	f_{scrub}	δ_{2SEU}	SEU/Hour
Flip-flop	27696	1.82E-14	11	20Mhz	1.00E-40	9.08E-34
Flip-Flop (Math Block)	4420	1.82E-14	130	6.67MHz	1.26E-37	1.54E-32
LSRAM	571392	2.53E-14	18432	9.77 kHz	2.28E-27	2.54E-22
uSRAM	39168	1.31E-14	1152	156.25 kHz	9.33E-33	1.14E-27

Table 4 Codeword SEU cross sections with mitigation

The results of mitigation on various fabric elements within an M2S025 are presented in Table 4. When compared with the single element cross sections in Table 1 a significant decrease in the effects of SEUs can be observed. The total aggregate expected SEU/hour for a fully utilized M2S025 is estimated at 2.54E-22, again this is dominated by the LSRAM cross section.

In implementing ECC on all fabric registers, many of those registers are devoted to that task as opposed to application tasks. In the example above, with a mean codeword length of 7 bits, about 43% of FFs in the FPGA are used for ECC. This ratio improves as the codeword length increases however this also increases the SEU cross section as each codeword contains more bits. This represents significant resource savings when compared with more common TMR techniques [26]. However the increased complexity of the error correcting circuits add new challenges as discussed below.

Validating the ECC Implementation

The ECC library implementation, as discussed in Firmware Design, can be validated at the RTL layer simply using a standard test bench and simulation tools. For a given bit vector length, all values of the vector are generated, encoded with ECC, decoded and compared with the original value. In order to confirm that it correctly mask all errors, the encoded vector is copied once for each bit in the vector, and faults are injected by inverting the respective bit in each copy of the encoded vector. Each corrupted copy is then decoded individually and compared with the original value. This demonstrates that the implementation will correct and mask any single bit error within its protected vector. For a bounded vector length this exhaustive testing it not difficult to perform on a modern PC. The longest encoded vector encountered in our design was only 24bits wide.

However things get significantly more complicated with more realistic propagation delays included in the simulation. While the RTL simulation only considers the ideal logic, real world implementations have delays between circuit elements which add significant complications to their correct behaviour. Until this point we have only considered ECC encoding on ideal synchronous circuits, which is fine as long as guarantees can be made that all signal propagation stabilizes before the next rising edge of the clock. This is usually provided by the vendor's synthesis tools after placement and routing has been performed, based on the actual delay values of the selected circuit elements involved in the implementation. Unfortunately SEEs are asynchronous events and thus level changes as a result of SEEs may violate the setup and hold requirements of synchronous elements, or more significantly induce a skew between related bits through a complex combinatorial circuit.

Consider the circuit in Fig. 16 which illustrates a typically ECC encoding and decoding circuit. Any change in one of the codeword registers must propagate through the decoding circuit, error correction, and then any other user logic before arriving at the next synchronous element in the design. In order to effectively mask all SEEs in the codeword registers, the ECC circuit must ensure that the downstream logic is always presented with the correctly decoded output.

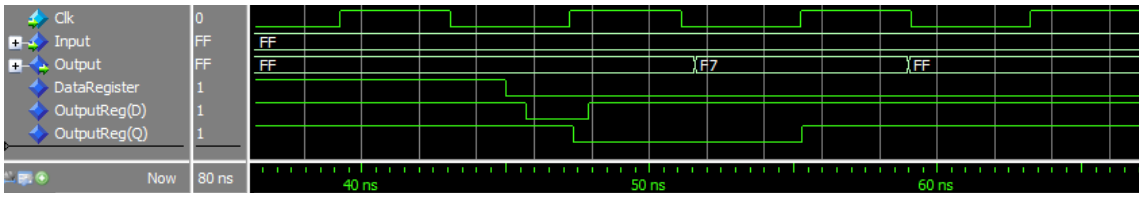

Fig. 20 Glitches in ECC decoding circuit

By simulating the post-route net-list, annotated with the propagation delays of the elements actually used in the design we can see this effect. Fig. 20 illustrates a few select data paths through the ECC decoding circuit of an 8bit input vector. Beginning with a steady state input of 0xFF and a correctly encoded/decoded output, a fault is injected at 45ns into one of the data registers storing the encoded vector. Shortly after, a transition can be observed on the input (D) of the output register. While this glitch is temporary and stabilises to the correct value, the intermediate value is latched into the output register on the rising edge of Clk at 47ns. This erroneous value is then propagated to downstream logic.

This demonstrates that SEEs can effectively subvert the protection of the decoding circuit if they occur within a period before the rising edge of the clock which is too short to allow

propagation and stabilisation of the correct value. This implies that the protection offered by ECC in this context only applies the 2 bit error rate for part of the clock period, while only providing a minor improvement on the single bit error rate for the remainder of the clock period.

An upper bound on this limitation can be derived by considering the fraction of the clock period in which the design timing is violated. The maximum clock frequency of the design is reported by the vendor tools after placement & routing is completed. This effect is the dominant factor limiting the effectiveness of our implementation of ECC to harden against and completely mask SEEs.

Discussion

We had intended to provide a radiation hardening strategy that would completely mask any single bit error throughout our entire design while minimising the resources required to do so. Our implementation of ECC applied to all memory elements in our design is expected to consume only 43% extra resources, compare with 220% when using standard TMR [26]. While these resource consumption rates appear accurate, and the RTL simulation of ECC shows it to be effective, the introduction of realistic propagation delays show significant flaws in this approach. The reduction in the rate of unmasked errors propagating to downstream logic when using ECC is bounded by the ratio of the system clock frequency (f_{clk}) to the maximum clock frequency of the design (f_{max}), though it is likely to be higher when specific designs are analysed.

While the implementation of ECC involves complicated combinatorial circuits which exacerbate this effect, I believe more traditional TMR approaches might still warrant consideration. The propagation delays through voting circuits and downstream combinatorial logic must be closely considered with regard to asynchronous SEEs. A simple experiment as above, but replacing the 8bit ECC vector with a single bit TMR vector did not show this effect. However this is likely due to the limits of the simulator, as a single bit TMR voting circuit can be implemented in exactly one 3LUT slice element. With propagation delays only being accurately modelled at the slice level this is potentially masking any skew from the slice's individual inputs.

While our ECC implementation is an improvement on no mitigation, by at least the ratio of f_{max} to f_{clk} (about x10 in our design), TMR appears to be a more effective, albeit expensive strategy to completely mask single bit errors.

Hardware Design

Here I define the requirements for a hardware device with which to implement our flight controller. This device is intended to control a small fixed wing aircraft autonomously, such that it can manage the aircraft's attitude, navigate between waypoints and maintain communication with a ground control station.

This device will include a main processor/FPGA in a monolithic System-on-Chip (SoC), the Microsemi M2S025. I will include memory elements such as DDR RAM and FLASH to support the operation of the processor. I also include some sensors such as accelerometers, gyroscopes, barometer & a compass on the board as well as provide interfaces to external GPS and airspeed sensors.

As this device is intended to be radiation tolerant, I briefly discuss the effects that radiation may have on each component in the design as well as ways in which these effects can be mitigated. The main focus is on the FPGA itself which is explicitly FLASH based to avoid corruption of configuration memory, but many of the components include digital systems and are susceptible to SEEs.

Introduction

Unmanned aircraft are becoming increasingly prevalent in society and the need to ensure these craft operate safely as they fly overhead should be the key concern of researchers in the industry. As the scale of these aircraft reduces so does their cost while their capability increases. While large scale aircraft are currently deployed by defence forces; civilian applications call for much smaller and more cost effective systems. However, as the size of the aircraft reduces so does the capability of the avionics on-board, leaving less computing power to maintain the aircraft's operating capability.

In order to maintain deterministic behaviour whilst reducing system size and power consumption I have developed a prototype flight controller based entirely around discrete logic in the form of Field Programmable Gate Arrays (FPGAs). This system trades development effort and flexibility for robustness and reduced size & power consumption. The prototype was constructed from COTS parts similar to those found on popular amateur systems. Future versions produced in volume could also benefit by replacing the FPGA with an Application Specific Integrated Circuit (ASIC), further lowering the power consumption and cost.

Several commercial flight controllers are available with various capabilities and prices. Each of these aim to autonomously pilot various unmanned aircraft. They are capable of tracking the attitude of the aircraft, locating it on the earth and manoeuvring it through a series of waypoints. Some devices are beginning to include FPGAs as accelerators, available to the main CPU processor [39] or as the bridge between arbitrary sensors and the CPU .

The advantage of a discrete logic system is inherent in the design of the firmware. Where a traditional CPU or micro-controller would perform all data collection, processing and control tasks within a single core, discrete logic allows for each task to be handled by dedicated silicon in the device. This vastly simplifies the firmware design, removes the need for real-time operating systems and simplifies component verification. It is also arguable that certain functions or algorithms benefit greatly from an implementation in logic rather than byte code and that system stability can be improved with faster update rates due to the parallelisation of operations.

Single Event Effects (SEEs) should be considered by all avionics engineers, as radiation in the Earth's atmosphere can interact with digital systems. High energy particles can deposit charge in any location within a digital system, causing transient logic levels, upsetting flip-

flops, corrupting data and potentially causing single event latch-up. While I am limited to COTS parts, I consider the effects radiation may have on the components I have include in my design and try to outline ways in which these effects can be mitigated. This is not always possible, and I concentrate my effort on the main processor/FPGA.

This is actually the third incarnation of such a device. All previous designs have put an emphasis on the use of a FLASH based FPGA and a completely autonomous platform. The main changes between each version have been in the size and capability of the FPGA, progressing through the ProASIC3 [67], igloo [68] and now SmartFusion2 (igloo2) [51] FPGA families. The original design did not include embedded DSP blocks, which required a significant portion of the FPGA fabric to be consumed by simple multipliers. The igloo fabric provided DSP blocks and the step to the SmartFusion2 added a small CPU.

This chapter will describe the requirements and design of a prototype flight controller that has been developed as a platform for further experimentation. While it is nice to design a generic device that can be used for any application it is important to consider a specific use case. Our future experimentation is planned around a solar powered, fixed wing glider. Specifically the airframe selected is the Pulsar 2.5E which significantly constrains the physical size of the avionics. Being solar powered, there is also considerable emphasis on low power consumption.

Fig. 21 The Pulsar 2.5E.

Platform Requirements

There are two primary categories of requirements for a simple avionics platform: navigation; the ability of the platform to determine where it is with respect to its goal, and control; the ability of the platform to control actuators towards achieving a goal. In addition to these two, I also have requirements relating to communication including the diagnostic and telemetry information available from the aircraft, and the physical constraints of the device.

Navigation

Knowing where you are and where you want to go is a basic requirement for any autonomous flight controller. Additionally, on a solar powered aircraft it is important to know where the sun is positioned with respect to photo voltaic arrays. This requires knowledge of the position of the aircraft on the earth, its attitude with respect to the horizon, as well as the date and time of day.

1. The platform must be able to determine its position on the earth within 10m at ≥1Hz.
2. The platform must be able to determine its attitude (roll, pitch, & yaw) with respect to gravity & magnetic north, within 3° at ≥20Hz.
3. The platform must be aware of the day of the year, and the time of day within one minute.
4. The platform must be able to navigate in order to traverse a list of waypoints.

which implies:

5. The platform must be able to determine the heading and distance from its location to that of a waypoint anywhere else on earth.

A platform with these capabilities would be able to follow paths and/or loiter at specific locations. It would also be capable of collecting telemetry data relating the sun's location with respect to photovoltaic arrays fixed to the aircraft, which would allow for detailed modelling and verification of solar power collection.

Control

This device is responsible for controlling the aircraft's actuators to maintain a sensible attitude and achieve navigation goals. While there is potential for many novel control system approaches, for the sake of specifying this platform I will be sticking to the basics while al-

lowing for other methods in the future. This platform is also intended to control a specific airframe which includes the following actuators: 2× flaps, 2× ailerons, elevator, rudder & throttle; however I hope to leave it capable of controlling other styles of aircraft.

6. The platform must be capable of supporting up to 8 servo style actuators.

This is typical of many avionics platform as most commercial actuators are either servos or make use of the same interface in the case of Electronic Speed Controllers (ESC). This interface being a standard 50Hz PWM. Having 8 channels allows us one extra channel to control a payload if needed.

7. The platform must allow for manual pilot control of actuators.
8. The platform must be capable of autonomously controlling actuators.

Allowing the pilot manual control is important during development and as a failsafe during experimentation. However the end goal is to have the flight controller pilot the aircraft autonomously and as such, should support both methods of control. The autonomous control has been left vaguely defined as the author cannot anticipate the control methods or even their goals at this time and is expecting to provide some generic processing capability to achieve various control schemes.

In the specific case of a fixed wing aircraft, the indicated airspeed is usually an important parameter for any control scheme. Additionally the aircraft's altitude is also something that is required to be controlled:

9. The platform must measure its altitude above sea level within 10m at ≥1Hz.
10. The platform must measure its indicated airspeed within 1m/s at ≥ 10Hz.

The actual control schemes used for this project will be detailed in Firmware Design.

Telemetry & Diagnostics

To provide a useful research tool, the avionics platform must be able to gather and either store or communicate data regarding the state of the aircraft and the environment in which it is operating. While there are an infinite number of metrics that can be gathered, this project is focused on the power consumption and generation of the aircraft. I also require data regarding the behaviour of the aircraft and its navigational state in order to develop and control the system.

11. The platform must measure the voltage across the main battery within 100mV at ≥10Hz.
12. The platform must measure the current draw from the main battery within 20mA at ≥10Hz.

As one of the main energy stores of the aircraft (the other being altitude) it is critical to know the charge state and power draw of the battery. This allows for detailed analysis of the power consumption of the aircraft through various manoeuvres as well as diagnostic information whilst operating.

13. The platform must measure the temperature of the motor, the battery, the ESC, itself & ambient air within 1°C at ≥1Hz.

This serves both diagnostics, as the symptom of many faults is excess heat and many tuning parameters are controlled within temperature limits, as well as telemetry which will often require temperature compensation or have temperature dependant variables.

14. The platform must provide navigation and data as telemetry, this includes:
    - Raw inertial sensor data: 3-axis gyroscope, 3-axis acceleration.
    - Raw Compass Data: 3-axis magnetic vector.
    - GPS Location.
    - IMU Calculated State: roll, pitch & yaw (heading).
    - Altitude.
    - Airspeed.

The acquisition of these parameters will have stricter requirements for the calculation of inertial state which is also likely to be at data rates greater than can be supported through telemetry. These should all be reported ≥10Hz as indications and debugging.

15. The platform must provide control data as telemetry, this includes:
    - Actuator values.
    - Flight state.
    - Current objective & calculated trajectory towards this objective.

Physical Requirements

In addition to the functional requirements above, the device must fit within the airframe and draw a reasonable amount of power. Finally there are requirements imposed by the expected operating environment.

16. The platform must fit within a 20 x 45 x 80mm bounding box.
17. The platform must consume <1W @ 5.5V.

It's difficult to assess the power consumption of similar platforms without physically testing them, as most manufacturers don't publish this information, or the features of each platform are not comparable. This is intended as an upper bound, but actually consumption should be minimized.

18. The platform must operate between altitude from Mean Sea Level (0m) up to 15,000m AMSL.
19. The platform must perform at temperatures from -40°C up to 50°C.

The actual expected temperature range throughout the operating altitude actually extends further below zero to -55°C, however COTS parts are limited to a standard industrial temperature range which limits us to -40°C.

Hardware Design

Experience has shown us that building a single monolithic device is not a practical way to integrate a flight controller with an airframe. This is because some "cabling" is RF coaxial cable or pneumatic tubing, both of which can be awkward to route to a centrally located flight controller. It is also the case that specific applications will have different requirements around communications, actuation and other peripherals. For these reasons I propose a design that is centred around a single processor board that contains only the critical components of a functioning flight controller with interfaces provided for optional and remote peripherals.

The main processor board includes the SoC, RAM and FLASH which are the basic components of the processing system. While a pure FPGA implementation of the flight controller is expected, which would not need RAM or FLASH storage, these components are included so that this hardware can also be used for any software approach that may be considered. The main board also includes the 9-axis IMU sensors, including accelerometer, compass, gyroscope and barometric pressure sensor. The selection of these components will be discussed in detail in IMU Sensors. Additionally this board contains supporting components including the power supplies, level translators and several connectors for peripheral devices & actuators. This main processing board, in addition to an external GPS and airspeed sensor should be enough to autonomously control many styles of unmanned aircraft.

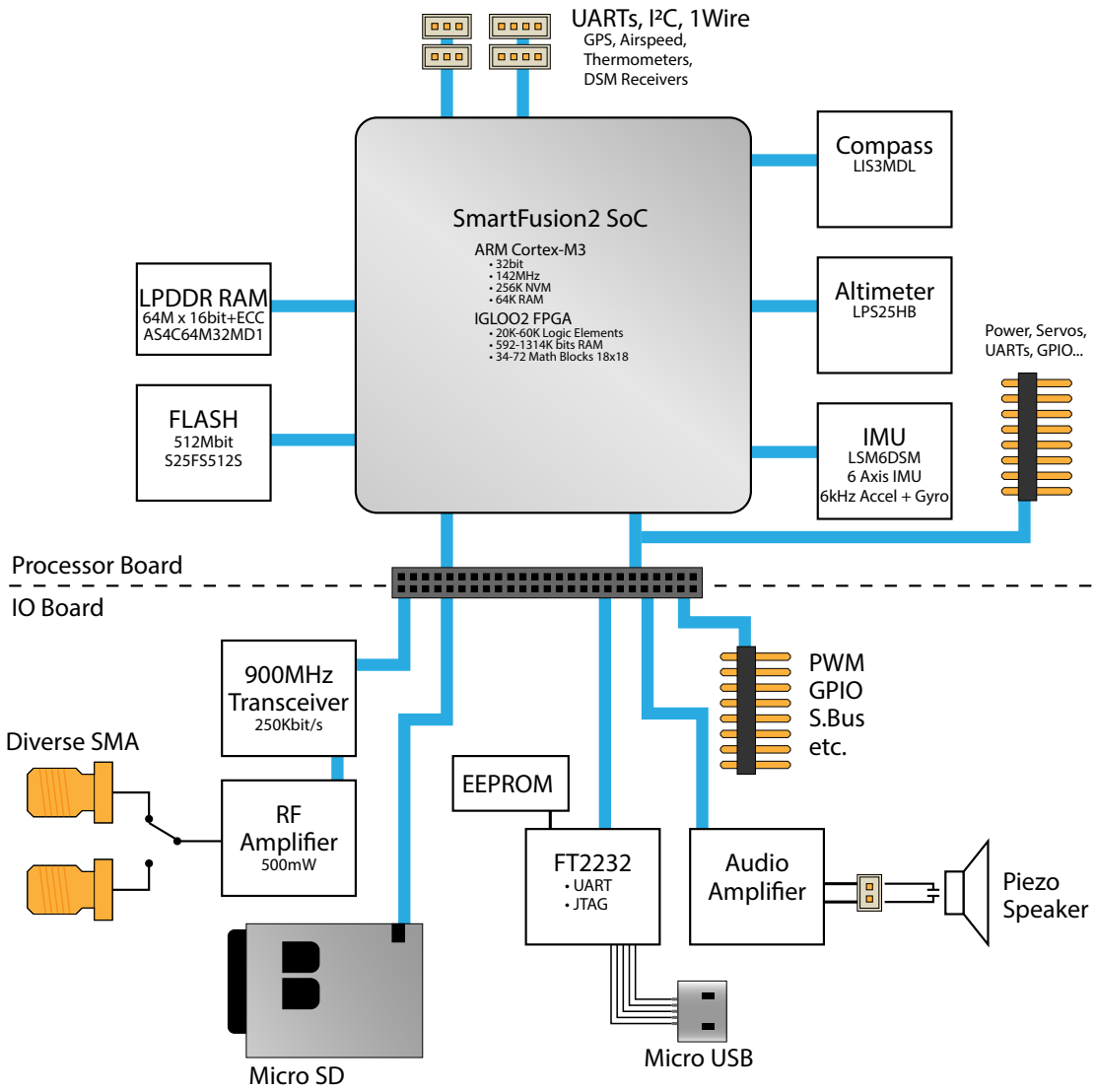

Fig. 22 System block diagram including both the processor board and breakout board.

In addition to the main processing board, I propose an initial peripheral board that includes some desired functions as well as some stated requirements. This IO board mates with the processor board through a 40pin header and provides features to ease user interaction. This includes long range RF communications, mass storage, audio output and a programming interface as well as extending connectors for more traditional actuator interfaces.

Power Converters

The goal of the power converter selection is to provide adequate power at the correct voltage levels for all the components in the design. This should be achieved with the highest efficiency and smallest physical footprint possible. The power requirements of each component in the design change dynamically depending on how they are used, and in the case of the FPGA, with the logic design it implements. It is very difficult to estimate power requirements

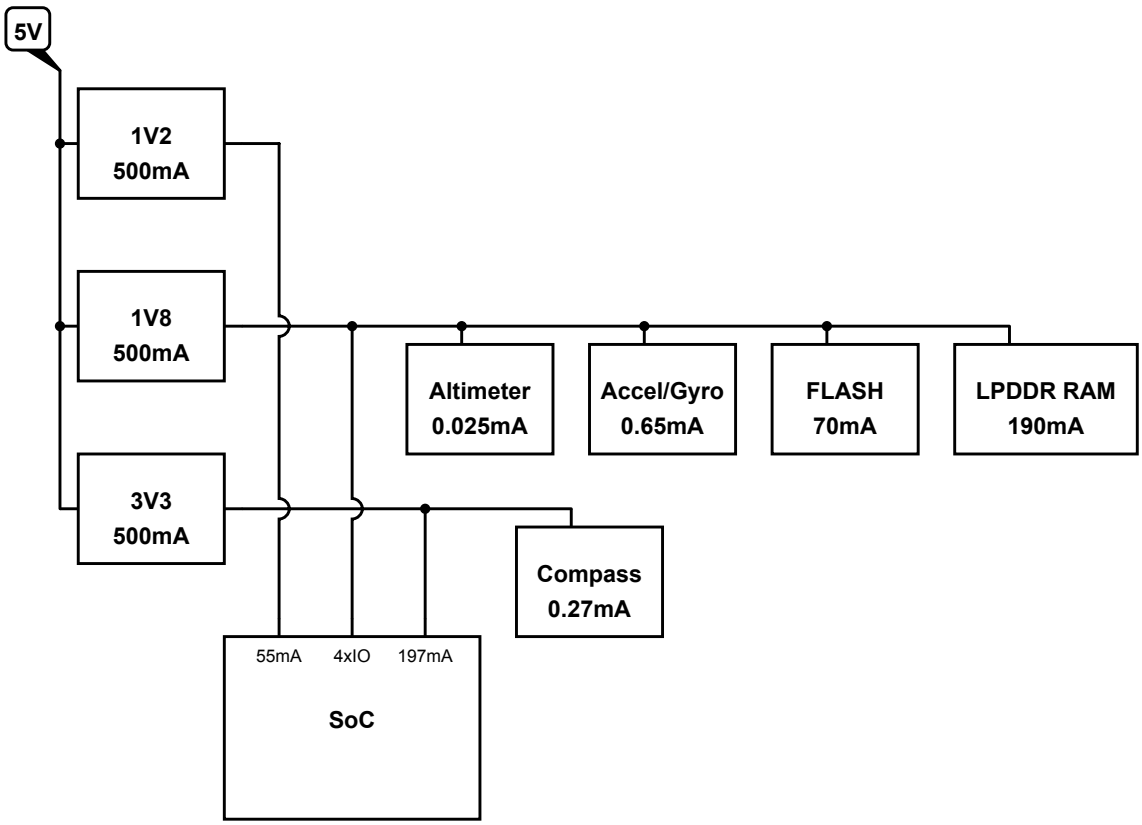

Fig. 23 Proposed power distribution with worse case current requirements indicated.

theoretically, but we can make some estimates on the worst case. The power requirements of the main processor board are summarised in Fig. 23. These current estimates were taken from the devices' datasheets in their most active state. The SoC power consumption was taken from its requirements during programming. In the case of FPGA power consumption, it is generally proportional to the number of logic elements in the design and the frequency with which they are clocked. The vendor's synthesis tools will provide a much more accurate estimate.

Power converters are a critical component and extra attention should be afforded to the reliability of the chosen solution. Like any component, they have a designed operational temperature range and may suffer de-rating towards the ends of this range. The topology and behaviour of the power converters should be considered with respect to SEEs, how they will affect the converter as well as how the converter will recover from a fault. Synchronous buck converters are a very popular choice for their efficiency and size, but they do have some drawbacks when considered in the presence of SEEs [69]. As discussed in Upsetting Logic, SEEs affecting the output switches may cause transient short circuits or in the event of SEL, a persistent short circuit until power is removed. Many modern power controllers also include features such as soft-start which limits the ramp rate of the output current as the device pow-

ers on to protect against large inrush currents. SEEs can potentially upset the controller and reset the soft-start mechanism resulting in power loss.

This design requires three distinct voltage rails at 1.2V, 1.8V and 3.3V and will operate from a single 4.5 - 5.5V supply (typical actuator/ESC voltages). The 1.8V rail has the largest current requirement of ~270mA, which is dominated by the RAM and FLASH. It should be noted that this current requirement is only when they are both "active". If the flight controller doesn't need these memories then their current draw will be insignificant.

We have selected the Semtech SC202A [70] which is a 500mA synchronous buck converter. This minimises the physical footprint of the converters by integrating the required inductor and not requiring voltage selection resistors. It includes over current protection which can sustain power to a persistent short circuit on the output. It has a power saving mode which improves efficiency at low loads, which makes it suitable for both the low and high current rails. It is desirable that the same converter is used for each rail for simplicity and a reduced BOM length.

The synchronous buck converter topology improves efficiency by switching two series transistors between power and ground with the output inductor in the middle. This is an improvement on the non-synchronous topology which used a passive diode on the low side. The danger is that the two transistors can never be conducting at the same time, otherwise a low impedance path exists between power and ground which is likely to damage the transistors. While the controller may be perfectly safe in switching these two transistors, it is possible that an SET may turn on the "off" transistor [69], creating a short circuit as illustrated in Fig. 24.

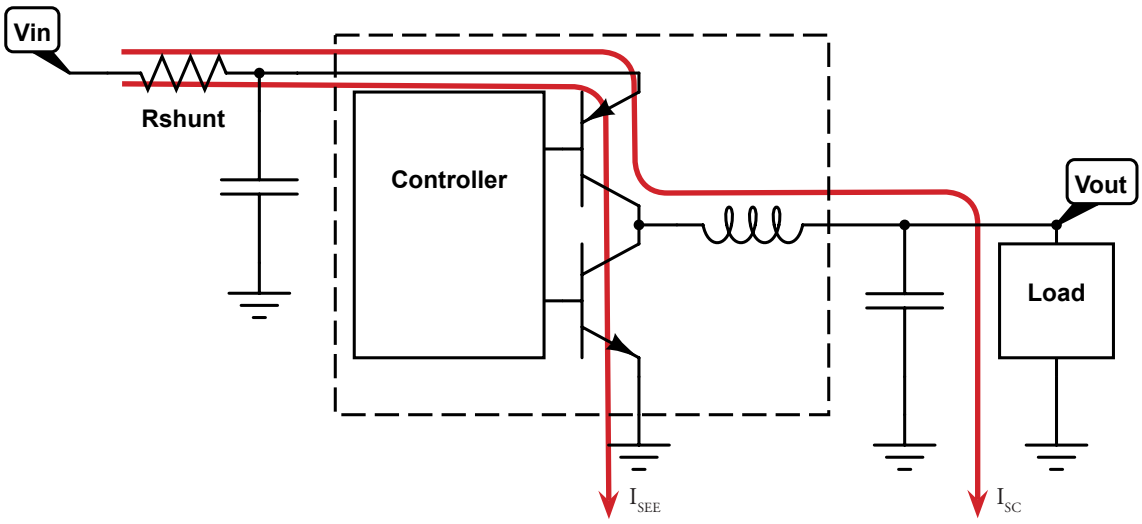

Fig. 24 Simple buck converter topology. Short circuit and SEE current paths indicated.

While these power converters have been designed to withstand persistent short-circuits on their output, the current path resulting from an SEL does not pass through the inductor and so the control logic will not have the same opportunity to modulate the output, potentially resulting in damage to the converter.

It is difficult to find another converter topology with similar specifications in the same physical footprint, especially with the appearance of devices with integrated inductors on the market. We proceed with the knowledge that there is risk of SEL in these power converters that we cannot mitigate.

Finally the soft-start and over-current features of the device must be discussed. The soft-start feature works by stepping up the current limit at 60 uS intervals, starting at 25% in increments of 25%. The soft-start can be inadvertently triggered as the result of an SEU which can lead to brownout or failure of the power rail. As the power required on each rail is nominally (except while using RAM or FLASH) below 25% of the current limit, the voltage on the rail should be maintained, avoiding a brownout condition. This is different from the more common solution of having a linear ramp from zero output power. The current limit also has a minimum time for the over-current condition to persist before any action is taken by the controller. This effectively filters out short pulses resulting from SETs from triggering an over-current condition.

Central Processing

Typically, when designing such a device the selection of a specific micro-controller is constrained by the external interfaces required to interact with the application specific peripherals, as each micro-controller has limited hardware support for various interfaces. However, when designing with FPGAs, the IO pins are much more abstract and do not directly support any specific interfaces in hardware; instead, the serial interface is implemented in the firmware running on the FPGA fabric. This provides great flexibility with interface and peripheral selection as well as the physical placement and routing of these devices on the PCB [42].

The main constraints around the selection of an FPGA for this application are the number of system gates available in the device, power consumption, physical device package and cost. Speed is not a factor as we are not anticipating any high frequency processing and we do not have a need for any specialist interfaces such as PCI.

FPGA fabric is divided up into a number of logical elements including lookup tables (LUTs), digital signal processors (DSPs), block RAM/FIFOs (BRAMs) and possibly others depending on the model and manufacturer. LUTs make up the bulk of the fabric and are used to implement custom logic. A LUT slice includes the lookup table itself, which may have anywhere from 3 to 6 inputs, and a number of flip-flops. FPGA specifications typically list the number of LUTs and their input count (or system gate equivalent) as a measure of the size of the device and can be used as an indication of how much logic can be implemented within the device. This LUT count of an FPGA is somewhat similar to the FLASH size of a programmable micro-controller.

DSP slices are dedicated hardware implementations of various arithmetic functions, typically a multiply-accumulate block and some registers; as with LUTs their design varies by model and manufacturer. DSP slices provide common functions that would otherwise take many LUT slices to implement and simplify the timing of those functions by minimising propagation delay through dedicated silicon rather than flexible routing fabric. An application that heavily relies on multiplication and large adders will require far fewer LUTs when that logic can be implemented in DSP slices instead. Finally, the fabric may also contain a number of memory elements called block RAM (BRAM). Again, their specific design varies with manufacturer and model, but each BRAM will typically be less than 20 Kbit in size with a variable aspect ratio. Many BRAMs support both RAM and FIFO interfaces. These memory elements are scattered around the FPGA fabric to act as lookup tables or buffers.

Power consumption within an FPGA is more complicated to estimate than in a micro-controller (although they can also be quite complicated with various sleep and power saving modes). A micro-controller's power consumption is a combination of the system clock frequency and the number of peripherals enabled and used by the software irrespective of the size of the software. FPGA power consumption has the same dependence on the system clock speed, but rather than turning on peripherals, the device increases power consumption by instantiating logic elements. This results in the power consumption of an FPGA being dependant on the size of the firmware implemented on it, as well as the system clock speed. FPGA power consumption is typically divided into static and dynamic consumption. Static power consumption is the quiescent power consumption of the device as well as the power required to maintain its configuration; in the case of SRAM based FPGAs the static power consumption can be significant. The dynamic consumption of the FPGA is the power con-

sumed as gates within the device switch, which is a function of the number of gates switching and the frequency with which they switch. The faster the system clock and the more logic elements instantiated within the design, the greater the power consumption of the device.

As FPGA configurations are stored as a large memory arrays they typically have large static power consumption when compared with micro-controllers. This can be avoided by using a FLASH based FPGA which stores its configuration in persistent memory as opposed to volatile SRAM which consumes power to maintain. In addition to lower static power consumption, the use of FLASH is also immune to SEUs preventing any corruption of the device configuration making them ideal for this application. At the time of writing, only one commercial manufacturer produced FLASH based FPGAs which limits the FPGA selection to Microsemi's product range. These are not to be confused with FPGAs that include persistent memory used to configure an SRAM device at power up.

This project has iterated a couple of times with different FPGAs all from MicroSemi including the ProASIC3 and Igloo. This provides us with experience regarding how large the FPGA is required to be to implement our firmware. Each time we outgrew the previous device, we moved to a device with more powerful LUT slices (4 inputs vs. 3) and now the igloo2 introduces DSP slices. Additionally the igloo2 is offered in the SmartFusion2 System-On-Chip (SoC) which co-located the FPGA on the same silicon die as an ARM Cortex-M3. The device we have selected is the M2S025 which we expect will provide adequate logic resources in the FPGA for future work; it was also the largest device we could develop for without having to purchase software licences. The CPU is not required for the flight controller or anything safety critical, but can be utilised as an interface for offline configuration such as applying coefficients and settings to the device and tasks such as calibrating the compass.

With the inclusion of a CPU it is desirable that this processing platform can perform similarly to other autopilot platforms currently available. The Pixhawk [36] includes a 32bit ARM Cortex-M4 running at 168MHz, and is considered adequate for many flight control applications. The main difference between the Cortex-M4 and the M3 on the M2S025 is the inclusion of a floating point unit (FPU). This as well as a number of other accelerators can be implemented on the FPGA so we consider this hardware capable of performing the same tasks as the Pixhawk, possibly by directly porting the open source Ardupilot software.

Memories

While the FPGA fabric is FLASH based, and therefore persistent between power cycles, the same cannot be said for the block RAMs in the FPGA. Quite often these memory elements are used for static LUTs to simplify the implementation of mathematical functions, for example sin and cosine, or to store calibration and navigation data. However the contents of these LUTs is undefined when the FPGA is first powered. It is necessary to load static memory contents from a persistent storage medium during the initial configuration. Additionally, if the CPU is to be used then somewhere to store the operating system is required. The M2S025 has a small amount (256 kb) of persistent memory available for both of these tasks, but is unlikely to be sufficient for future designs. We include a 512 Mbit FLASH device, selected mainly for its physical size, on the processor board. This memory can be read & written by the CPU and the FPGA.

There is also a need for bulk persistent data storage on board the device. Experiments requiring high fidelity telemetry can produce a significant data stream which can easily saturate a wireless link. It is desirable that this data is stored on-board for analysis offline or potentially delayed relay over a wireless link. We include a microSD card slot as this provides a simple, cheap and convenient method to store and retrieve data in the field.

The final memory requirement is for large and fast storage of data that saturates both the wireless link and local persistent storage. This is typically RAM which is scattered throughout the CPU/FPGA. While these blocks are fast, operating at the local oscillator usually generating the data, they are also quite small. The M2S025 includes 64 kb of embedded SRAM accessible to the CPU and FPGA plus 611kb scattered around the FPGA fabric. We include a 2Gb DDR device on the processor board to provide bulk RAM. This RAM can service bulk data collection at high bandwidth as well as the needs of more complex operating systems if they are required.

It is required that this RAM also be protected from radiation effects which is achieved by the ECC algorithm available to the memory controller in the M2S025. However the inclusion of extra ECC bits to each data word does make the interface a little awkward. The memory bus is 16 bits wide plus 2 ECC bits, and 18 bit wide RAM devices are not very common among COTS products. For this reason we selected a 32 bit wide device and simply don't use the remaining 14 bits. This also has the effect of reducing our effective RAM space to 1Gb.

The algorithm used for ECC should also be discussed, as the codewords are actually 64+8 with the ECC bits being spread over several data words [46]. This is apparently to take advantage of the logarithmic scaling of the hamming code used and reduce the overall memory requirements. However in this case, we have 14 bits per data word being unused anyway, and access to a single data word requires access to many physical addresses to recover the ECC bits. This may be effective for a RAM device exactly 18 bits wide, but these don't really offer size or cost advantages due to their niche uses, and otherwise standard device packages.

As an example of the RAM capabilities, if the IMU sensors discussed in this chapter (Accelerometer, Gyroscope, Compass and Barometer) are sampling at their maximum rates we can expect about 804kbs. With a 1Gb RAM available we can expect to store full resolution data for about 22 minutes. These bandwidth requirements grow rapidly when you also add control outputs, and internal state and other sensors to the required telemetry. The selected RAM is low power DDR which has a maximum interface clock of 200 MHz, giving a theoretical maximum bandwidth of 6.4 Gbps with 16 bit wide bus. However to reduce power consumption and improve reliability by avoiding the use of PLLs, the RAM can be clocked at the native oscillators 20 MHz (discussed in Oscillators). This reduces the RAM bandwidth to 640 Mbps, which is still more than enough for expected data logging rates.

Oscillators

There is a need for many different frequencies with different tolerances throughout this design, and many ways to achieve those clocks. The CPU has a maximum clock frequency of 142 MHz, the DDR RAM can go as high as 200 MHz while the FPGA fabric only needs something <50 MHz.

While the M2S025 does include two RC oscillators on its die, these are not particularly stable, with a maximum tolerance of 5% which may not be enough for higher speed interfaces such as USB. The FPGA also includes several Clock Conditioning Circuits (CCCs) with integrated PLLs which can synthesize many new frequencies from a single reference oscillator.

The effects of temperature changes, vibration and radiation must be considered carefully regarding the choice of oscillators and overall clocking scheme. As discussed in Upsetting Logic there are a few common technologies used in reference oscillators including RC filters, crystal resonators and more recently MEMS resonators. COTS MEMS oscillators provide the smallest and most stable solutions and may include active temperature compensation

and excellent vibration tolerance. However these do have one flaw when it comes to SEEs, in that the output frequency of the device is usually programmed at the factory by way of a digital divider. While the user may have no means to change this divider, SEUs may lead to a persistent change in output frequency that can only be corrected by removing power from the device [57]. Quartz crystal oscillators have been used for many decades in aviation and space applications and the effects of radiation are well understood. They are also much more stable than an RC filter and so will be the preferred technology in this design.

The PLLs inside the M2S025 make frequency selection fairly arbitrary, as most references can be used to synthesize something close to the actual required frequency and can be configured dynamically without requiring hardware changes. However it should be noted that SEEs were observed in [52] which resulted in the PLL to lose lock on its reference input. Depending on the free-wheeling behaviour of the PLL it could be argued that this is not catastrophic. The main concerns with oscillator faults are persistent (or at least significant duration) changes to frequency, and the presence of any "glitches" or shifts in phase that may cause clock periods shorter than timing paths in the design. These glitches would violate the setup & hold times of registers in the design resulting in data corruption. This is likely unrecoverable in contrast to longer clock periods which sill simply slow down computation without data corruption.

The proposed design includes a single external oscillator to provide a direct frequency reference to the FPGA fabric without requiring any PLLs. This will be a quality quartz crystal oscillator running at 20 MHz. This frequency was chosen from experience as a good starting point between speed (which is really just the margin between system clock and interface data rates) and logic complexity in the design. The CPU and RAM could be clocked directly from this reference also, but given they are not intended for any critical tasks an internal PLL can be used to generate 100 MHz and 200 MHz references.

IMU Sensors

The inertial measurement unit (IMU) usually refers to the hardware and software components that are responsible for measuring the motion of the aircraft and maintaining a "state" that represents the position and attitude of the aircraft in the world frame. Interpolating an accurate state of the aircraft is quite a complex process which will be discussed in depth in Firmware Design. Briefly the IMU works by taking high rate measurements of relative

movement and integrating them over time, and then using slower measurements of absolute references in the target frame to correct accumulated errors. Here we will describe the selected hardware components of the IMU including their types of measurement, interface and bandwidth requirements.

The gyroscopes provide the current rate of rotation of the sensor around its axes. This provides high frequency updates to the attitude of the aircraft which is required for tracking manoeuvres and maintaining stability. However errors in the magnitude of rotation and sensor noise will accumulate in the IMU so a state based on rotation alone will quickly drift. The rate of rotation is sensitive to small offsets in the measurement which will appear to be slow but constant rotation.

Linear acceleration provides information about how the aircraft is moving. Similar to the gyroscopes, the acceleration measurements can be integrated to provide estimates of velocity and position. In addition, local gravity can be measured (after being isolated from motion induced acceleration) which provides an absolute reference in the world frame as to which way is "down". This vector can be used to correct the long term drift from the gyroscopes.

Acceleration and rate of rotation measurements are often performed by a single monolithic device so as to simplify IMU calculations by at least removing the offset between the axes of rotation and acceleration. We selected the LSM6DSM [72] from ST as it was the fastest sampling device available while having comparable resolution and noise specifications to competitive products. The higher bandwidth sampling eases requirements on numeric accuracy as discussed in Firmware Design. This requires an SPI interface and when sampling at 6.6 kHz we expect 739.2 kbps of data, including the internal temperature.

Another absolute reference we require is north, which is measured by our compass. This vector is not affected by the motion of the aircraft, but it can be altered by hard & soft iron objects nearby in the airframe, or possibly on the ground if large enough. The north vector as measured by our compass also changes with location, with declination and variation changing significantly at different positions on the earth. We selected the LIS3MDL [73] again from ST for similar reasons of sample rate. The data requirements are much less significant at a modest 64 kbps when sampling at 1000 kHz.

The final sensor is not technically required for the IMU, but is very important for the operation of an aircraft. That is the barometric pressure sensor, which provides altitude information by sensing the atmospheric pressure outside the aircraft. When calibrated for

current local weather conditions this provides an accurate measurement of the aircraft above sea level, but can provide a high rate measure of vertical speed even without this calibration. We're using the LPS25HB [74] again from ST which can sample barometric pressure at 25Hz with a very modest 1 kbps data rate.

Other sensors that are required by the platform but are only optional to the function of the IMU are the GPS and airspeed sensor. These are awkward to mount on a centrally located flight controller and are instead considered external peripherals.

External Peripherals

There are a number of peripheral sensor and actuator interfaces this design is required to support. While we are designing towards fixed wing aircraft with well defined sensor and actuator layouts, it is desirable that this device is able to support other configurations. The peripherals we are anticipating are GPS modules, airspeed indicators, standard servos and ESCs (including those with telemetry capabilities). We have included several connectors for these peripherals including 3 pin and 4 pin headers, a standard 2.54 mm header as well as a 40 pin board-to-board connector for tightly integrated peripherals.

Support for arbitrary peripheral devices is actually quite easy to achieve with an FPGA, as we can implement any interface driver required in the FPGA fabric and route it to whichever header we need. The only consideration required is the number of pins the interface requires as well as the voltage level of those pins. We include level translators on each of the interface pins which can be configured via physical jumper, to drive the interface at either 3.3 V or 5 V which are the two most common logic levels among small unmanned aircraft components.

With the inclusion of the board-to-board connector, any number of configurations can be developed. As parts of this design we include a breakout board that includes a 900 MHz radio, microSD card slot, standard 8 channel servo header, and a programming interface for the main processor.

Design Summary

This design resulted in two distinct components that fit together to form a functional avionics systems for further experimental work. These components are tolerant of SEEs to the extent that could be achieved with COTS parts that fit within the required form factor.

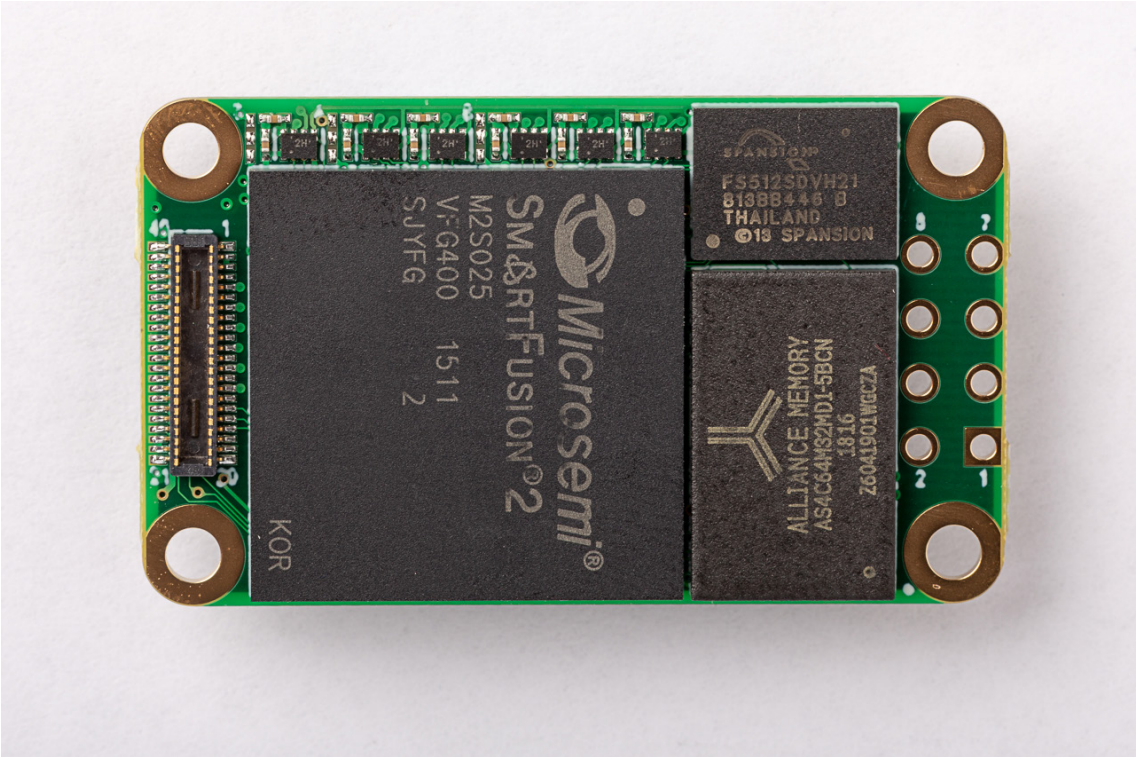

Fig. 25 The main processor SoM.

The main processor board forms a "System On Module" that is capable of controlling an autonomous platform in a stand-alone fashion, albeit without communication capabilities. This device measures 35 x 20 x 8mm making it able to fit within extremely space constrained airframes. It features a processor that combined with an FPGA can be comparable to existing autopilot platforms. It is realistic that existing software can be ported to this device. However the main advantage of this platform is the FPGA, which allows the flight controller to be implemented entirely within logic fabric. This significantly increases the determinism

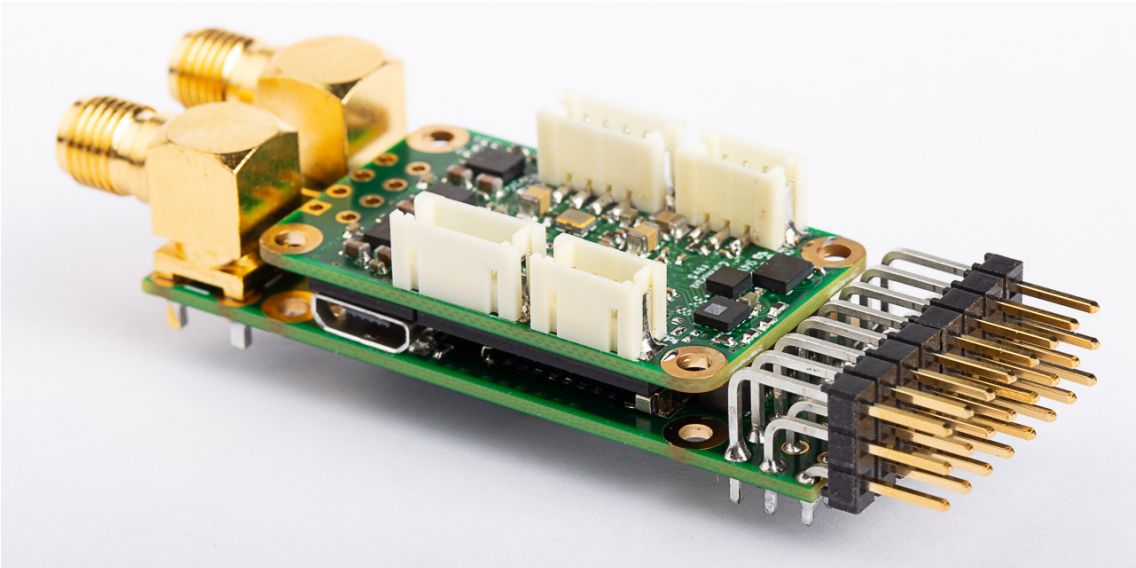

Fig. 26 The main processor SoM mounted on the breakout board.

of the system while decreasing the difficulty with which formal methods can be applied to verify the software system. In addition, extra steps can be taken to radiation harden registers within the logic fabric allowing for complete radiation tolerance in the processing system.

We also developed a breakout board which completes the design requirements for our experimental platform. This breakout board includes a 900MHz transceiver and RF amplifier that provides a 250 kbps data link with a ground control station up to 500 mW. This is expected to provide real time communications at a range of up to several kilometres. The breakout board also includes an audio amplifier to which an optional speaker can be attached. This allows the device to alert nearby operators to various modes of the device without a communication link or user interface; which is required for safe operation, especially in autonomous modes. A microSD slot is included to allow easily removable bulk storage. Finally the breakout board includes standard servo headers to ease integration with typical commercial airframes for model aircraft, as well as a programming interface to the main processor. This breakout board 67 x 20 x 14mm and the combined weight of the SoM and breakout board is 19g.

The remaining requirements of the flight controller platform are left to external peripheral devices. This includes the GPS for positioning, an external pressure sensor with a pitot tube for airspeed sensing, and an ESC that includes telemetry to monitor the battery. The GPS and airspeed indicator prove challenging to mount in a central position, as they require cabling/tubing that is sensitive to being bent through a tight fuselage. They also both require a position on the aircraft that is not typically buried within a fuselage, such as the leading edge of the wing, or somewhere with a clear view of the sky. Monitoring the battery is easily performed with the right ESC, which is far more convenient than inserting another group of sensors.

System Layout for the Pulsar 2.5E

The Pulsar 2.5E was the airframe selected for experimentation with autonomous gliding techniques and solar power integration. It was selected for its efficient aerodynamic design and relatively large wing area. However the fuselage of this airframe is very space constrained, adding to the desire for a small form factor avionics package. Fig. 27 illustrates the components included in this airframe and how they are connected to the flight controller.

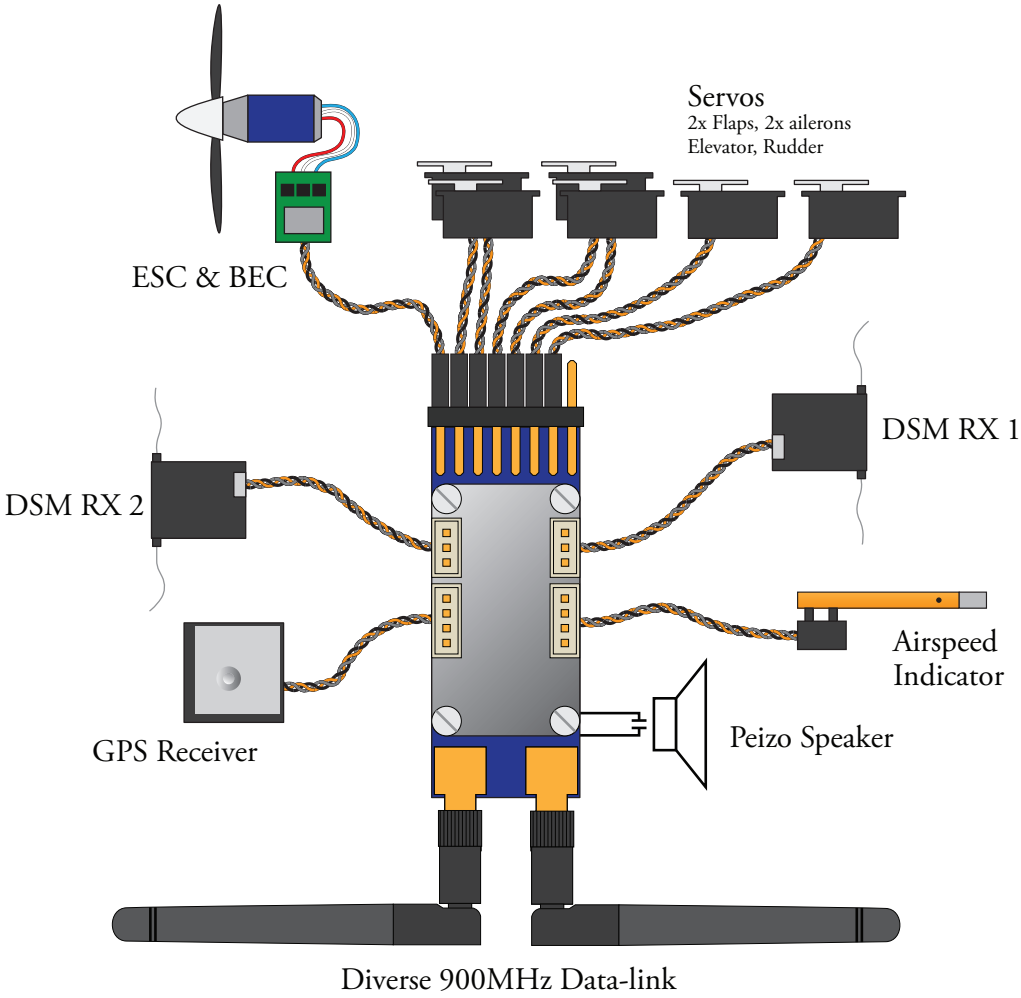

Fig. 27 The layout of system components for the Pulsar 2.5E airframe.

The power for the aircraft is provided by a 3S LiPo battery pack which connects to the ESC. The ESC includes a Battery Elimination Circuit (BEC) which acts as a step down converted from the main battery voltage to typically 5V for the avionics and servos. We're using a Castle Creations Talon ESC which controls the brush-less motor and also provides telemetry to the avionics regarding motor performance and main battery status.

We include two DSM receivers, which is the standard used by Spektrum transmitters for communicating pilot commands over a 2.4GHz wireless link. These allow for direct pilot control of the aircraft as if the avionics were a standard receiver.

To maintain aerodynamic efficiency it is desirable to mount the data link antennas within the surfaces of the airframe. This is typically within the wings or the vertical stabiliser. The tail boom is carbon fibre and acts as an RF shield. As the antennas usually have a torus shaped radiation pattern, it is impossible to orient a single antenna in a way that maintains signal strength for all attitudes of the aircraft. The radio included on the breakout board includes a switch between either of the two antennas for both receiving and transmitting

that allows it to select the antenna currently experiencing the greatest signal strength. The two antennas are mounted on orthogonal planes to maximise the aggregate radiation pattern coverage.

Power Consumption

The power consumption of the hardware design was monitored in various configurations across multiple units to quantify the power requirements for individual features. These features include the FPGA with varying utilisation, the CPU at different frequencies, the radio at different transmit power levels as well as the quiescent power requirements for the SoM and breakout board.

Measurements were made of the input voltage and current drawn through a high-side current amplifier with a digital storage oscilloscope to provide enough samples per second to isolate dynamic power changes. Further details are provided in Appendix C.

The observed power consumption, broken down by component is show in Table 5.

Component	**Power (mW)**
SoM	80
Breakout (quiescent)	23
IMU	12
CPU	58 + 1.35/MHz
Radio (active-quiescent)	44
Radio TX	309
Radio TX HGM	449

Table 5 Power consumption by component

These results show that a minimal configuration which provides for IMU activity as well as navigation and control processing capabilities using only discrete logic (FPGA), can be deployed and consume only ~92mW. This configuration does preclude the ability to communicate with the device either through telemetry or command and control signals, though data can be logged to persistent storage on the device for later retrieval. A more typical configuration would include the breakout board and radio which would require approximately ~184mW on average.

Conclusion

In this chapter we have defined the requirements for a small flight controller intended to autonomously pilot fixed-wing aircraft. There was an emphasis on small size and low power consumption such that the device could be used for a solar powered glider, the Pulsar 2.5E. The design was also required to support the formal verification and radiation hardening techniques detailed in this thesis, which dictates the use of a FLASH based FPGA as the central processor.

The resulting design of the flight controller was discussed with each substantial component detailed and justified. The main processor is the M2S025 [51] SoC from Microsemi, which includes a FLASH based FPGA and ARM Cortex M3 microprocessor. Several sensors are included in the device including the LSM6DSM [72] accelerometer & gyroscope, the LIS3MDL [73] compass and the LPS25HB [74] barometric pressure sensor. We also included 512 Mb of FLASH [71] memory and 1 Gb of DDR RAM.

We also proposed a breakout board to extend the functionality of the main processor board by including a 900MHz ISM transceiver, micro SD card holder, audio amplifier and programming interface. The modular design allows for the main processor to function as a complete system with limited capabilities, while the board-to-board connector can be utilized to develop extension boards to extend capability for specific applications.

The processor board measures 35×20×8mm and the processor board combined with our proposed breakout board measures 67×20×14mm. Their combined weight is 19g. Estimating the power consumption of the flight controller is difficult as this changes with behaviour. Power consumption of an FPGA design can be difficult to predict, as this value changes with the utilisation of the FPGA and the frequency at which it is clocked. Experiments show that a minimal configuration of this device can be deployed and consume ~92mW, while a more typical configuration would likely consume ~184mW on average. These values were above our theoretical expectations by about 40%. This is likely due to inefficiencies in the DC-DC power converters as well as optimistic estimates from vendor software.

The flight controller presented here provides an adequate platform for our experimental requirements. With the inclusions of external peripherals, such as GPS and an airspeed indicator this device will be capable of meeting our control and navigation requirements. While we are only implementing firmware to run on the FPGA, the ad-

dition of the CPU is convenient for a number of reasons. It is a convenient method for performing off-line tasks such as compass calibration, and loading persistent data into persistent memory. The resources available are comparable to other devices with similar applications in mind. This allows us to run existing open source software such as Ardupilot [38] to allow others to use this platform without requiring development for the FPGA.

Firmware Design

The flight controller firmware is the logical design implemented within the FPGA and is responsible for interfacing with hardware peripherals, analysing data, providing behaviour & control of the airframe and communications with a ground control station.

This chapter details the implementation of the firmware including everything from the serial interfaces and peripheral drivers to the higher level functional elements that implement specific behaviours of the flight controller. Only a few interesting components will include full details of the algorithms implemented.

The design is more explicitly hierarchical than typical software designs as each component maps directly to an area of physical silicon within the FPGA. I will begin discussion at the highest platform level, including the hardware context before exploring each sub component in more detail.

Formal verification and SEE mitigation will not be detailed in this chapter, though examples of this firmware design will be taken for detailed verification in Firmware Verification. However, where specific components provide interesting examples, these will be briefly explored.

Firmware Architecture

The firmware refers to the logic design running on the FPGA of the device. It is responsible for interfacing with peripheral devices, estimating the state of the aircraft and providing control values to airframe actuators. It is also responsible for communicating telemetry data to an external ground control station. The device I am targeting is the M2S025 SoC [51] which includes both an FPGA and a CPU. In this thesis I will only consider firmware running on the FPGA; not any software implemented on the CPU. Though software running on the CPU may provide some convenient functionality, it is not necessary for the functioning of the flight controller.

In order to defend against SEEs within our design, we implement hardening algorithms on all registers, BRAMs and MATH block. This is done by using the techniques described in Upsetting Logic with the help of our ECC library, which is used to wrap all register assignments with encoding/decoding circuits. These techniques are implicit with the implementation and do not warrant discussion throughout this chapter. However some interesting cases of hardening strategies not previously covered, are discussed.

The firmware has been divided into many logically distinct components, each with a specific task. These range from external device drivers to internal processing stages and various 'glue logic'. These components have been arranged to achieve two main goals, control of the aircraft and communication with the ground station. The overall architectural design of the firmware is illustrated in Fig. 28. Each component is isolated from unrelated components, interacting with others, only through explicit interfaces. This greatly simplifies verification of each component as it can actually be considered in isolation without regard for shared resources or scheduling.

The main flight control algorithm starts with device drivers for the peripheral sensors required to measure the aircraft's position and motion. Each of these drivers provides output data in their own time, minimizing sampling latency in order to maximize controller bandwidth. Raw data is provided to the inertial measurement unit (IMU) to estimate the attitude of the aircraft. This combined with the aircraft's position and next intended waypoint are provided to the controller. The controller is a collection of PID controllers aiming to achieve an attitude specified by the navigation block and aircraft performance specified by the pilot.

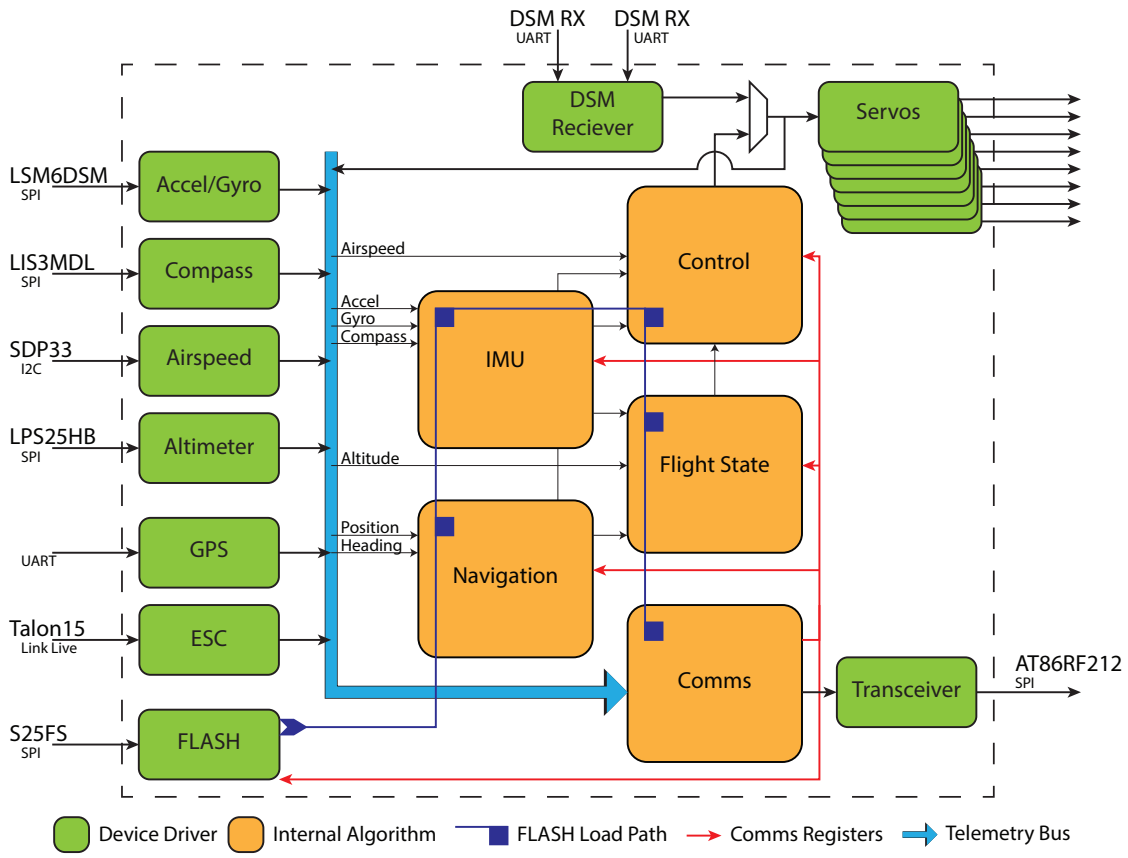

Fig. 28 Top level firmware architecture.

Finally the control outputs are provided as actuator values to a bank of PWM encoders to drive the servo actuators and throttle of the airframe.

The goal of the communications is to provide as much data about the state & behaviour of the aircraft as possible. This includes raw data from sensors, integrated data from the IMU and behavioural data from the navigation and control blocks. This data is necessarily down sampled from the raw input rates to a representative rate that fits over the wireless interface. Full rate, raw data can be collected in internal memory and transmitted in a non real-time fashion. In addition to telemetry data, the radio link can also be used for bulk, non-critical data potentially from devices external to this system.

There are two pieces of infrastructure that permeate throughout the firmware design. These are the command & control links from the communications, and the static memory initialization bus. Loading data from persistent memory at power up is required to populate tables of static data, such a look-up-tables for navigation algorithms as well as default settings for calibration values and control limits. Although the FPGA itself is FLASH and stores its configuration persistently, this is not the case for volatile memory elements such as BRAMs and flip-flops. The FLASH loading bus forms a daisy chain of elements that are loaded from

a contiguous block of persistent memory. Each element is continuously "scrubbed" to ensure any faults during runtime are removed. Separate from these static memory elements are the dynamic control registers. These registers hold values that are dynamically altered over time such as waypoint data for the current mission and aircraft performance settings. These may require a default setting from persistent memory, but cannot be scrubbed, as the value changes with mission requirements. These values are initially provided by the FLASH loader to the communications block, where they are treated like any other command received from the ground control station in order to seed registers with default values.

As pieces of telemetry information are required in many parts of the firmware, it is simplest to consider all telemetry information forming a large data 'bus' that is available throughout the system. This is reflected in the VHDL, however it should be realised that each data element is only physically routed to relevant blocks. This is one of many discontinuities between conceptual implementation in VHDL versus the physical reality of an optimized bitstream.

Device Drivers

Each device driver is responsible for interacting with a peripheral device, be it a sensor, actuator, memory device or communications interface. Drivers are required to initialize their device at power up and confirm this configuration remains correct during operation. They are also required to manage timing of data samples and the integrity of that data.

As each device driver is implemented in a dedicated section of the FPGA fabric with their own physical interface, there are no scheduling requirements either for IO buses or for CPU time. This allows each device to be serviced in its highest sample rate, lowest latency configuration while maintaining determinism. This provides the maximum bandwidth to the IMU and control algorithms.

Each driver includes the physical layer encoding (PHY) used to interface with the external hardware, as well as a state machine to define the specific behaviour of the driver and how it interacts through its own interfaces. I discuss the common template used for all drivers in this design as well as the concerns and recovery from faults in the peripheral devices.

Physical Interfaces

Each device driver contains the physical interface to the device. In this design, the interface is typically a standard serial interface such as SPI, I²C or RS232. The servos specifically have a PWM output on a single pin, and the ESC uses a single pin PWM protocol for both telemetry and control. Additionally, some peripheral devices beyond the scope of this thesis have included OneWire and other physical interfaces. All of these interfaces have been implemented with the same style and in an isolated manner, so as to be reused in other drivers.

These PHYs are responsible for serializing & de-serializing data to & from the device over an external interface and for any extra signalling required such as synchronous clock generation and specific framing signals such as chip selects or start & ack signals over I²C. Finally they are required to signal to client side logic (within the FPGA) when each transaction is completed.

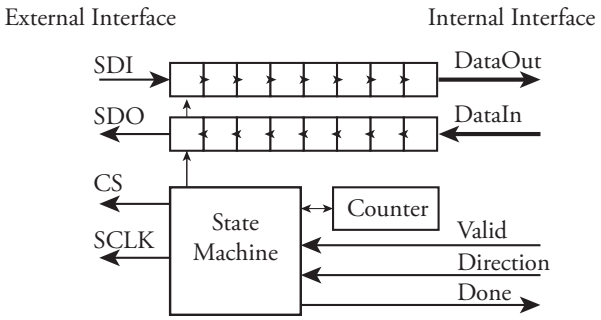

Fig. 29 Example serial bus (SPI) PHY block diagram.

Complete Driver

All the device drivers implemented in this firmware follow the same pattern, give or take some elements specific to their behaviour. There is a PHY that abstracts away the transactions over the wire to a simple byte-wise interface. There is a finite state machine (FSM) and byte counter that defines configuration and measurements steps, usually involving a periodic loop. Finally there is a lookup table of data bytes sent to the device. This lookup table is typically implemented in fabric flip-flops given its small size and complex addressing scheme. The device driver supplies bytes to the PHY from the lookup table indexed by the current state of the driver and the byte counter. Data from the PHY is latched during specified states,

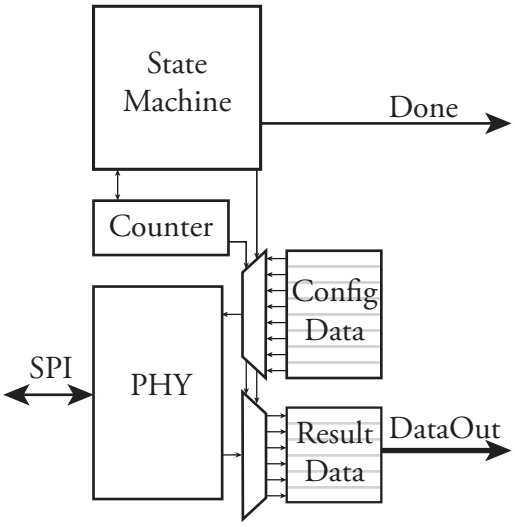

Fig. 30 Example peripheral device driver block diagram.

typically in to a shift register that maps to the final output format. This template, illustrated in Fig. 30, has proved simple and effective for all device drivers encountered.

The main differences between specific drivers are how they determine when to take specific action. It should be noted that these drivers are designed to perform one specific task with respect to the avionics system being implemented. For example the FLASH driver only loads a contiguous block of memory and does not allow arbitrary read/write access to a file-system. Sensor drivers need to know when to sample data, while the transceiver needs to know when data is received and when data is ready to transmit. This is encoded into the driver's state machine which typically includes a WAIT or IDLE state. Transitions from these states occur on interrupts from the peripheral device or from something generated internally such as a timer. Preference is given to internal signals as these are more deterministic than interrupts and come with radiation hardening advantages discussed later. However, avoiding interrupts is not always possible so we must still consider these during verification.

Configuration Scrubbing

As with any digital circuit, those in peripheral devices are susceptible to the effects of radiation which may corrupt configuration and data registers, or add noise to analogue measurements. Unfortunately these memory elements are inaccessible to our standard radiation hardening techniques. We are mainly concerned with the configuration registers of the peripheral device to ensure that they continue to provide or respond to data as expected, with the expected ranges and offsets applied. Transient faults such as the corruption of a single data bit in an output sample are expected to be tolerated and mitigated by upstream logic. This usually

involves filtering where a single data "spike" is suppressed, or checksums which detect and discard errors.

There are two main techniques we use to ensure the integrity of the configuration of peripheral devices. Both of these techniques amount to memory scrubbing of the device's control registers. The technique used depends heavily on the device in question and how its controls have been implemented.

The first scrubbing technique relies on the device having the entire configuration written preceding a measurement. This is common in simple devices such as some ADCs where there are not many configuration options. Each measurement begins with writing the entire configuration which may include gain and channel selection, and then waiting for an interrupt or a fixed duration before retrieving the data. In this case, any corruption to the configuration is not persistent as it is re-written regularly when the driver expects a new measurement. Consideration must be made in the driver when using an interrupt to time the completion of a measurement as errors in the peripheral may cause the interrupt to never be asserted. In most cases it is preferable to time the measurement independently and then conditionally retrieve the result if the interrupt is asserted or reconfigure the device if it is not. This causes the driver to act a watchdog timer and handle the missing interrupt by continuing with the next measurement. These potential bugs will become apparent in Firmware Verification.

The second technique is applicable to more complex devices that require an explicit initial configuration step that is separate from their periodic behaviour. This is typical for devices such as the LSM6DSM Accelerometer/Gyroscope [72]. These devices require configuration of things like dynamic ranges, sampling rates and any filtering algorithms applied. After the initial configuration the peripheral device controls transactions by generating regular interrupts which the device driver responds to with read requests. In these cases the device driver is required to sanity check the behaviour of the peripheral device in respect to both sample timing and data integrity. A watchdog timer is a simple approach to monitoring the expected interrupt rate but data integrity is much more difficult. For example, a potential fault in the gyroscope gain register would be very difficult to detect. While a gain change would shift the measurements by a power of two, the gyroscope is nominally zero so no "jump" in the measurements would occur. It is not possible to define a general approach to detecting data integrity faults, especially as a result of configuration errors. Instead we propose sanity checking the configuration registers directly. This involves reading back the configuration as

part of the periodic behaviour of the driver and comparing it with expected values or hash. Any fault detected would result in a reset of the driver and the reconfiguration of the device. A simpler implementation would be to just write the entire configuration periodically as described in the previous technique, but this depends on how the peripheral device responds to register writes.

Inertial Measurement Unit

The inertial measurement unit in the firmware is responsible for integrating readings from the peripheral sensors and providing a best estimate of the current attitude of the aircraft. This design implements a simple complementary filter [75] which was chosen to ease development during early iterations of this design. The current state of the art inertial measurement is usually achieved with an Extended Kalman Filter [76] [77] which applies a statistical approach to combining many source of noisy data. Given the availability of MATH blocks in newer FPGA families this style of algorithm can potentially be implemented. However our simple complementary filter has proven effective in early experimentation.

The goal of this IMU is to provide an accurate attitude of the aircraft including its roll, pitch and yaw (or heading) with respect to the world frame. To keep this simple we do not attempt to integrate position, although this design can be extended to do so. Given that we are primarily targeting fixed wing aircraft operating at high altitude, there isn't much need for high resolution position information beyond what is provided by a raw GPS.

The world frame is defined as Cartesian axes aligned with the local tangent plane on the Earth's surface at the aircraft's current position by a right hand axis aligned with true north, true east and gravity, while the body frame of the aircraft is a right handed axis aligned with forward, right and down. These axes are illustrated in Fig. 31.

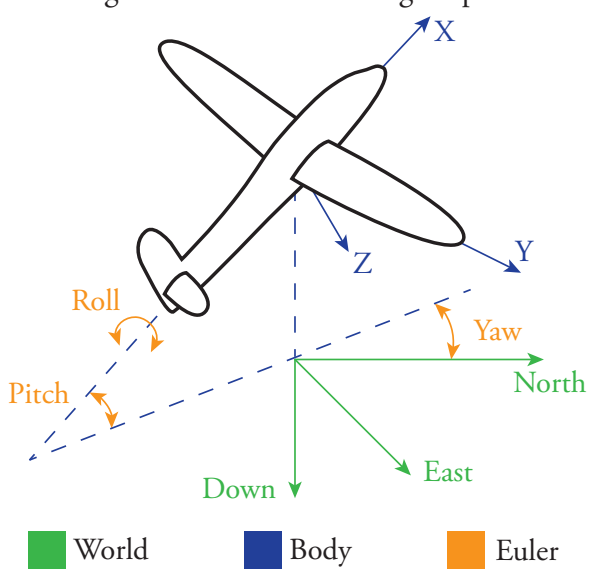

Fig. 31 World and body axes definitions.

Before the IMU, the sensor data is first filtered and calibrated. The gyroscope samples pass through a high-pass filter

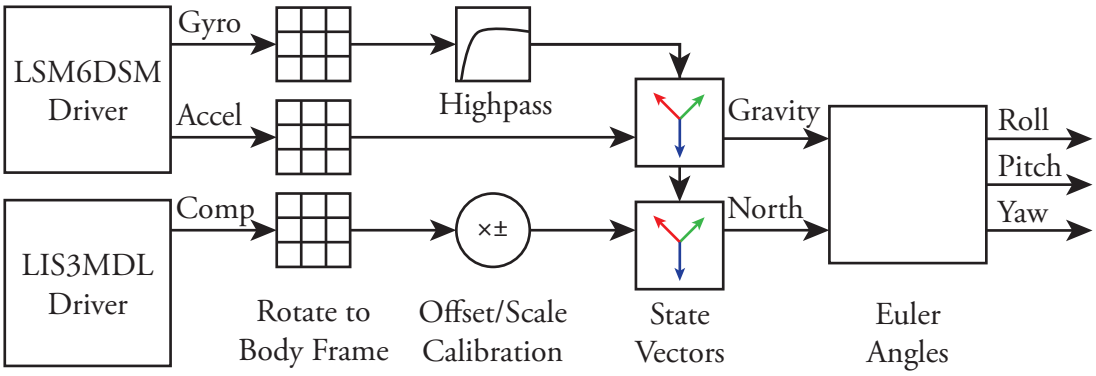

Fig. 32 IMU top level block diagram.

that removes any DC bias. The compass has calibration scale and offsets applied to compensate for hard iron effects. The gyroscope, accelerometer and compass samples are all rotated to align them with the body frame of the aircraft and correct for any offsets in the physical mounting of the sensors. After the sensor readings are corrected, they are passed to the IMU to update the current estimate of the aircraft's attitude. This estimate is then translated into Euler angles before being passed to the control algorithm.

Our IMU works by tracking two absolute vectors from the world frame in the body frame of the aircraft. The stored state in the IMU is the north $\vec{\mathbf{n}}$ and gravity $\vec{\mathbf{g}}$ vectors in the body frame of the aircraft. On each new sample from the gyroscopes $\mathbf{G}$, these two vectors are rotated by the measured rate over the time of each sample. Gravity and North are two vectors that can be measured directly by the accelerometers $\mathbf{A}$ and compass $\mathbf{N}$ (with some errors). These direct measurements are used to "correct" the stored state by drifting it towards the absolute references. The complementary filter acts by balancing high frequency rotation data with a low frequency drift towards absolute references. These two vectors are then used to find standard roll, pitch & yaw Euler angles in the world frame, which are used to control the aircraft. The overall structure of this algorithm is illustrated in Fig. 32.

The rotation of the state vectors is achieved by a monolithic HDL entity that minimizes the FPGA resources required while meeting the timing requirements for state updates. It is desirable to update the state accumulators at the highest rate, which is the native sample rate of the gyroscopes. This rotator entity simplifies the arithmetic by using small angle approximations of trig functions.

$$\begin{aligned} \sin\theta &\approx \theta \\ \cos\theta &\approx 1 \end{aligned} \qquad [6.1]$$

The faster the update rate, the smaller the angle of rotation is each step and the smaller the error introduced by these approximations. The main algorithm is simply multiplying the state vector **u** with standard rotation matrices built from the gyroscope data as shown in [6.2].

$$\begin{bmatrix} u'_x \\ u'_y \\ u'_z \end{bmatrix} = \begin{bmatrix} \cos G_z & -\sin G_z & 0 \\ \sin G_z & \cos G_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos G_y & 0 & \sin G_y \\ 0 & 1 & 0 \\ -\sin G_y & 0 & \cos G_y \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos G_x & -\sin G_x \\ 0 & \sin G_x & \cos G_x \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \qquad [6.2]$$

Where $\mathbf{G_n}$ is the angle of rotation about axis **n** in radians per time-step. Substituting in our small angle approximations from [6.1] this becomes:

$$\begin{bmatrix} u'_x \\ u'_y \\ u'_z \end{bmatrix} = \begin{bmatrix} 1 & -G_z & 0 \\ G_z & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & G_y \\ 0 & 1 & 0 \\ -G_y & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -G_x \\ 0 & G_x & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \qquad [6.3]$$

This rotation is done for both the gravity and north vectors in parallel. The drift is added with each new sample to push the current state vector towards the absolute reference for that vector. Drift is calculated by simply multiplying the sign of the difference between the current state vector and the reference vector by a constant chosen to determine the weight of the drift each step.

$$\vec{d} = \text{sign}(\vec{u} - \vec{r}) \times C_{drift} \qquad [6.4]$$

Where $\vec{\mathbf{r}}$ is either **A** or **N** scaled to make the state vector **u** drift towards about half its dynamic range.

Typical software design that aims to minimize operations would likely pre-multiply the rotation matrices into a single matrix. However our aim is optimize for FPGA resources which in this instance would be multipliers. Our state rotator takes an iterative approach, reusing the same multipliers to rotate the state about each axis sequentially. There is a trade-off between throughput and resource usage. Consuming more FPGA resources allows for more parallel operations, while reusing elements reduces resource usage but require more operations to be performed sequentiality.

As an example, our inertial sensor samples at 6.6 kHz, but the system clock in the FPGA is running at 20 MHz. Without considering the latency requirements of the data processing, the timing requirements simply need the IMU to accept a new sample as each one is provid-

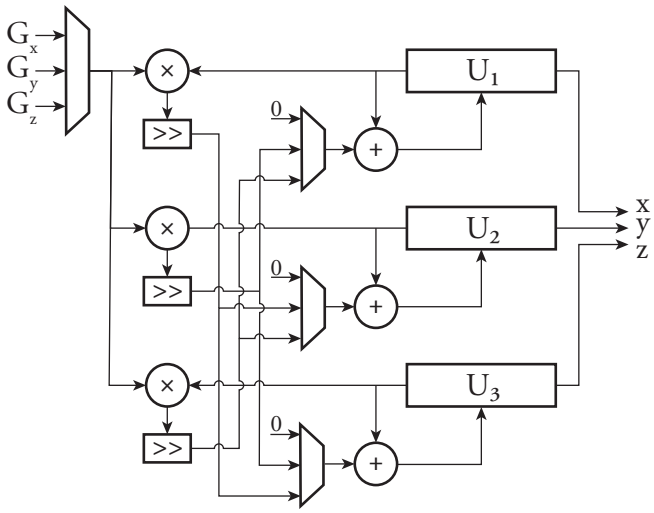

Fig. 33 State vector rotator implementation.

ed by the sensor. This can be achieved by pipe-lining if high throughput is required, but in this example we actually have 3030 clock cycles between each new sample. This IMU was originally written for an FPGA family that did not include MATH blocks and instead used the Booth iterative multiplication algorithm [78]. Even with these high latency multipliers, this IMU can still process each sample in ~75 clock samples.

This design uses high frequency rotational data to estimate the state, which is corrected by low frequency absolute data. The gyroscope provides high bandwidth information about the motion of the aircraft, but errors in the measurement and numerical errors can quickly accumulate to distort the state estimate. When the aircraft is flying straight and level, the accelerometers provide a measurement of gravity, which is used as an absolute reference to correct the state estimate. This vector however, also measures accelerations perceived by the aircraft as it manoeuvres, so high frequency components of the accelerometer vector are considered errors for this purpose. The complementary filter acts as a low-pass filter on the gravity vector, and a high-pass filter on the aircraft's rotation. The compass is used to align the state vectors with north. Given magnetic north as measured by the aircraft suffers from local declination and variation, it is difficult to use this as a vertical reference also, as it would need to be corrected based on the current position of the aircraft. It can also be distorted by large magnetic fields from object nearby.

Once the two state vectors have been established, we convert this state into Euler angles which are a more natural metric with which to control the aircraft. This is achieved by another rotator element. The rotator works to iteratively rotate the two state vectors by a fixed angular step about the body frame axes to align gravity with Z and north with X. Rotations

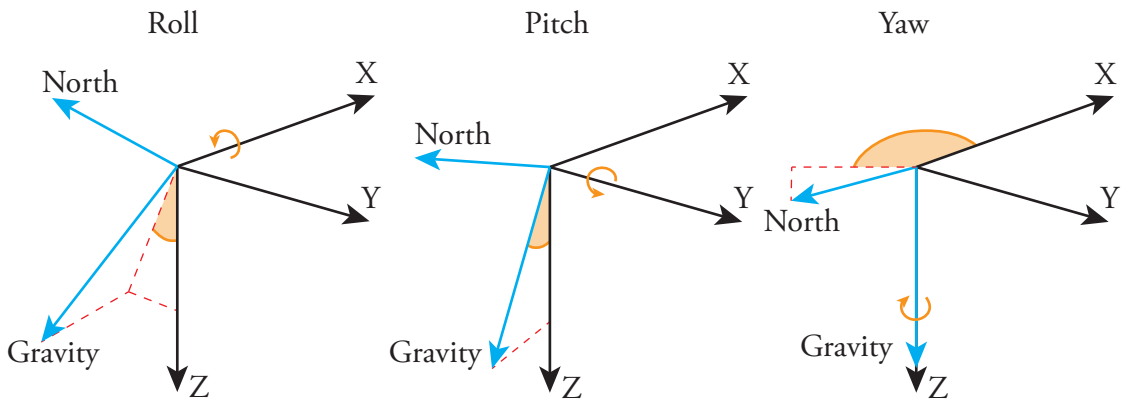

Fig. 34 Measuring Euler angles from state vectors.

are done about one axis at a time and the resulting Euler angle is the sum of steps required to reduce one component of the state vector to zero. This process is illustrated in Fig. 34. The order in which these angles are measured matters, and we follow roll, pitch then yaw. Both the gravity and north vectors are rotated together.

Using an iterative method with a fixed angular step allows us to simplify the arithmetic and avoid calculations of trigonometric functions. The rotation matrices are the same as above, but the $\sin\theta$ and $\cos\theta$ terms are now replaced by a constant value which is the magnitude of each rotation increment. This constant can be selected to be a power of two, so that now all the multiplications required can be achieved by simple bit-shifts. This rotation is repeated until the required vector component is zeroed and the sum of angular steps is recorded as the relevant Euler angle.

This process begins when the state vectors have been updated with the latest sample and at a 6.6kHz update rate, we have 3030 clock cycles to complete the process. This can be done in parallel with a new sample being processed by the state rotator. As all multiplications are reduced to simple bit-shifts, each rotation now only requires a single clock cycle. The resolution required in the output angles determines the size of the angular step and the duration of calculation. For 8 bit outputs each axis can be rotated a maximum of 256 steps, though the pitch and roll axes only need half that range before it is easier to rotate the other direction. So the worse case duration for this operation is 128+128+256=512 clock cycles.

Navigation

The navigation entity provides the controller with a bearing and distance to the next waypoint, manages the list of waypoints that form the aircraft's path and determines if the aircraft is within a defined boundary. Navigating around a spherical Earth is arithmetically complex [79] involving a lot of trigonometry, square roots and division; all operations that make an FPGA developer cringe. By optimizing the arithmetic and throwing numerical caution to the wind we were able to implement a serviceable algorithm built mainly from lookup tables. The derivation and analysis of this algorithm has been omitted for brevity as there are more effective methods for calculating these parameters; particularly using CORDICs [80] [81].

This implementation calculates the initial bearing and greater circle distance to a target waypoint from the aircraft's current position, both specified in latitude & longitude. The results do suffer significant numerical error which increases with the distance between the two points in question, however it is effective for waypoints up to 10 km away, which is more than enough for any experimentation we plan to do.

$$\mathrm{b} = \arctan\left(\frac{\sin(\Delta\phi)}{\sin(\cos(\phi_2)\cdot\Delta\theta)}\right) \qquad [6.5]$$

$$\mathrm{d} = \arccos\left(\sqrt{\sin^2(\Delta\phi) + \sin^2(\cos(\phi_2)\cdot\Delta\theta)}\right) \qquad [6.6]$$

or for small distances

$$d = \sqrt{(\Delta\phi)^2 + (cos(\phi)\cdot\Delta\theta)^2} \qquad [6.7]$$

Equations [6.5], [6.6] and [6.7] provide the initial bearing and greater circle distance in radians from position 1 to position 2 given by their latitude ϕ_n and longitude θ_n. The implementation of this algorithm relies heavily on the use of lookup-tables to implement complex functions including sin, cosine, division etc. as well as combinations of these functions. This implementation is illustrated in Fig. 35.

The mission path itself is stored as a number of waypoints in BRAM that is writeable by the communications interface. A simple state machine is used to read the current waypoint and the next waypoint in the path. The current waypoint bearing & distance are used for navigation & control, while the next waypoint bearing and the current waypoint distance are used to determine when the transition is made to begin tracking the next waypoint in the path.

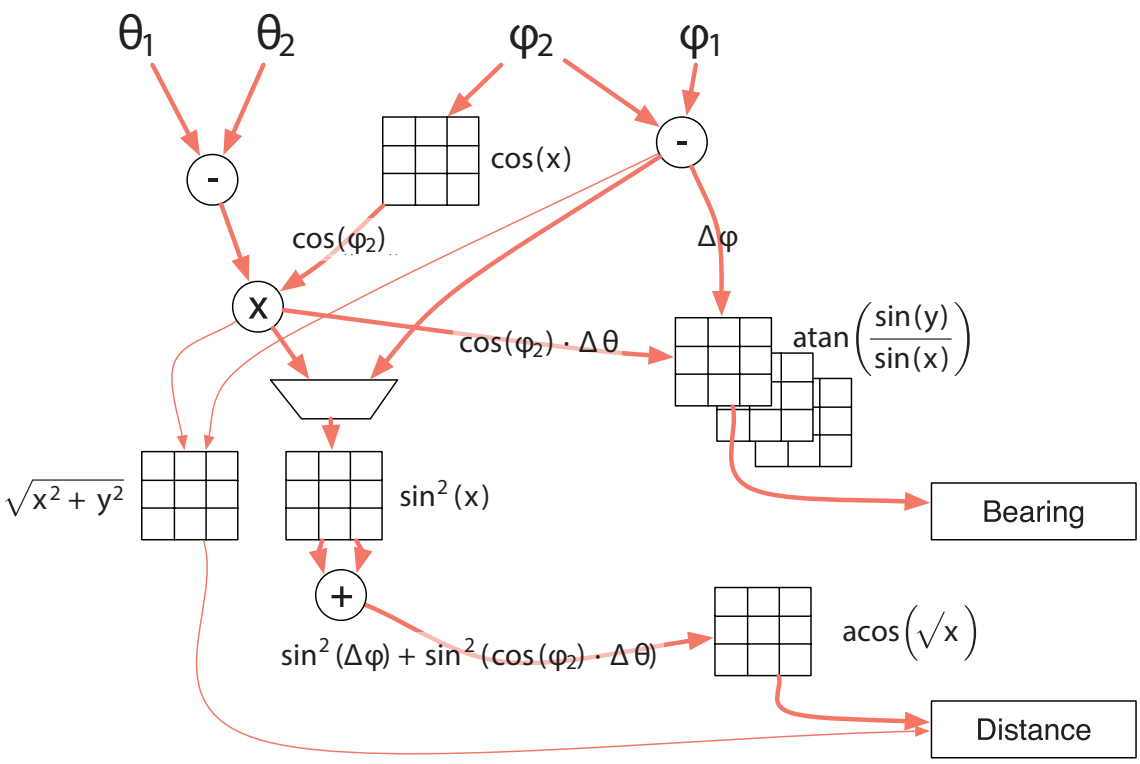

Fig. 35 Navigation computer block diagram.

The navigation computer is designed to follow a path around circular waypoints with a radius defined by the operator. An offset, scaled by the distance to the current waypoint, is applied to the calculated bearing. At a significant distance from the waypoint, the aircraft will follow a direct line from its current position to the centre of the waypoint. As it gets closer, the aircraft heading will drift towards a tangent of the circle defining the waypoint. Once the aircraft is established on the tangent it will follow the circle around until its heading matches the bearing to the next waypoint in the path.

Wind Compensation

The navigation computer is also responsible for compensating for wind to ensure the desired track is maintained. Wind is defined as a 2D vector orthogonal to gravity. We did not attempt to automatically estimate the wind vector in this design, though this is certainly possible. Instead we provide a static wind vector or a vector calculated on-line by the ground control station in response to track errors. This vector is simply applied to the target waypoint as an offset to its latitude & longitude, scaled by the distance to the waypoint. The magnitude of the wind vector includes both the strength of the wind as well as the expected airspeed of the aircraft. This causes the navigation computer to provide a bearing that compensates for the effect of wind over the track to the next waypoint.

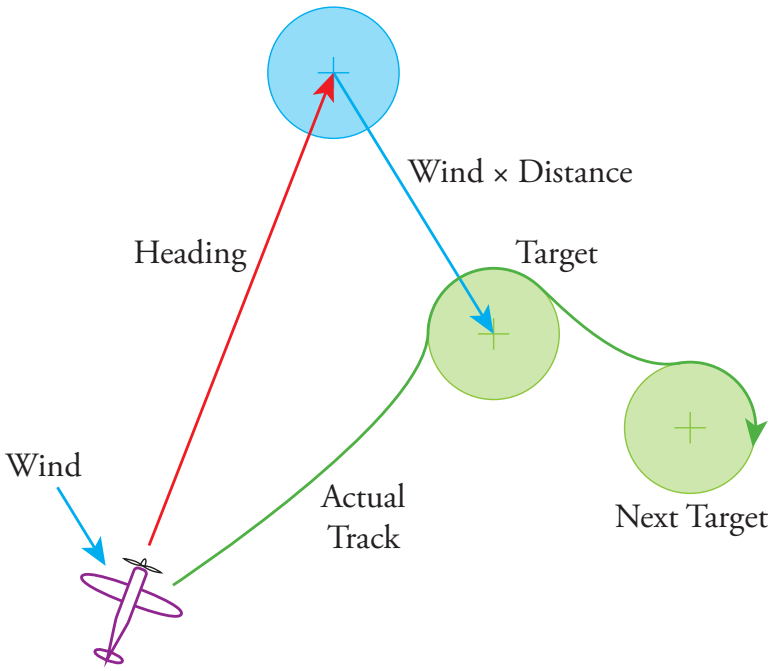

Fig. 36 Expected path between waypoints plus wind compensation.

Flight Boundary

The flight boundary is defined by a number of waypoints in exactly the same way as the mission path. On each position update, the navigation entity calculates the winding number of the boundary path by summing the differences in relative bearings to each point in the boundary. For reasonable paths, the winding number will either sum to 2π; indicating that the current position is within the boundary, or zero; indicating that it is outside the boundary. This is a simple but effective algorithm that doesn't require any significant increase in logic resources, as it simply makes use of the navigational computer already implemented.

Flight State & Control

The behaviour of the aircraft is defined by two entities. The flight state provides a high level concept of what the aircraft is trying to achieve, whether it be tracking the next waypoint, gaining altitude or even being idle on the ground. The controller provides actuator values based on the current flight state attitude and navigation results that are intended to manoeuvre the aircraft towards its current goal.

First, we will briefly describe the expected operation of the aircraft, as it isn't exactly a typical example of a fixed wing aircraft. The aircraft we are using as our test platform is a high performance, powered glider. While it does include a motor and propeller, it spends most of its time gliding, un-powered until it has descended to a threshold altitude, at which point it

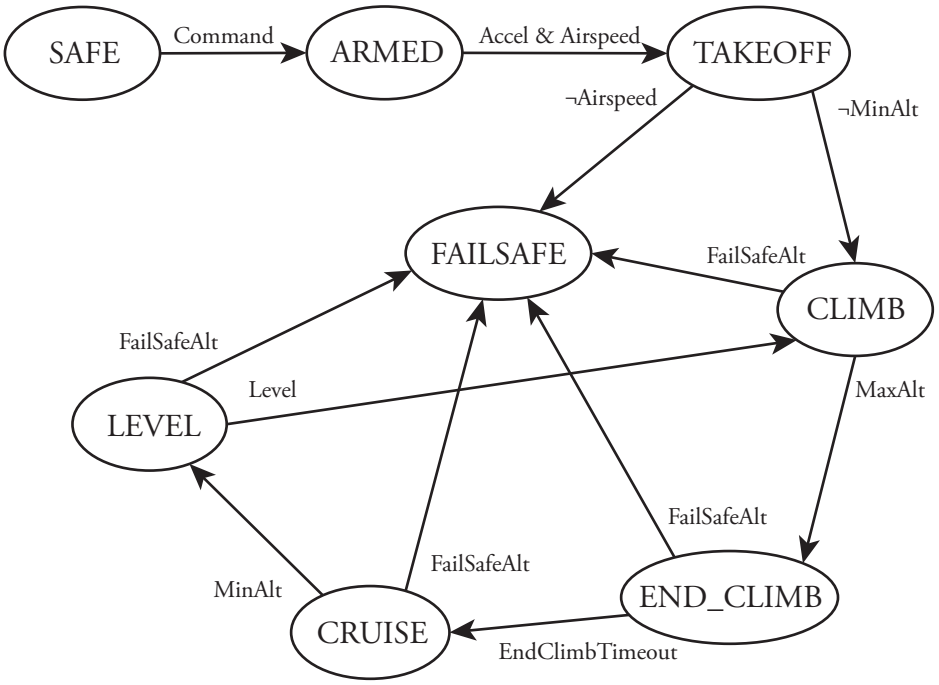

Fig. 37 Flight state finite state machine.

will increase throttle and climb aggressively until a ceiling altitude is reached. Once at altitude, the throttle is cut and the aircraft resumes gliding.

The flight state entity is a simple state machine that implements the process described above as illustrated in Fig. 37. The inputs to this state machine include the current altitude, some variable threshold altitudes for the floor and ceiling of operation as well as pilot operated signals. A failsafe threshold is also included as well as a boundary violation signal that, if either are violated, the aircraft enters a failsafe state to quickly and safely terminate the flight.

The controller is a collection of PID controller entities, each controlling some aspect of the aircraft's performance or attitude to manoeuvre the aircraft towards its goal. The topology of these controllers is illustrated in Fig. 38. The gains of the PID controllers are variable and supplied by default coefficients from persistent memory, or from the operator at runtime. The goal of the aircraft changes as the flight state changes. For example, during the CLIMB state, the throttle is set to 100%, the roll of the aircraft is controlled towards wings level and the rudder actively controls against yaw movement. The pitch of the aircraft is used to control the airspeed which is set by the operator. In a climb, the airspeed set point is usually exceeded as the pitch limit is reached. Transitional states are used to smooth out behaviour between CLIMB and CRUISE states, ie. to maintain wings level and airspeed when the throttle is cut until the aircraft is in a straight & level attitude, at which point the CRUISE behaviour takes over. During CRUISE the aircraft continues to use pitch to control its airspeed, only

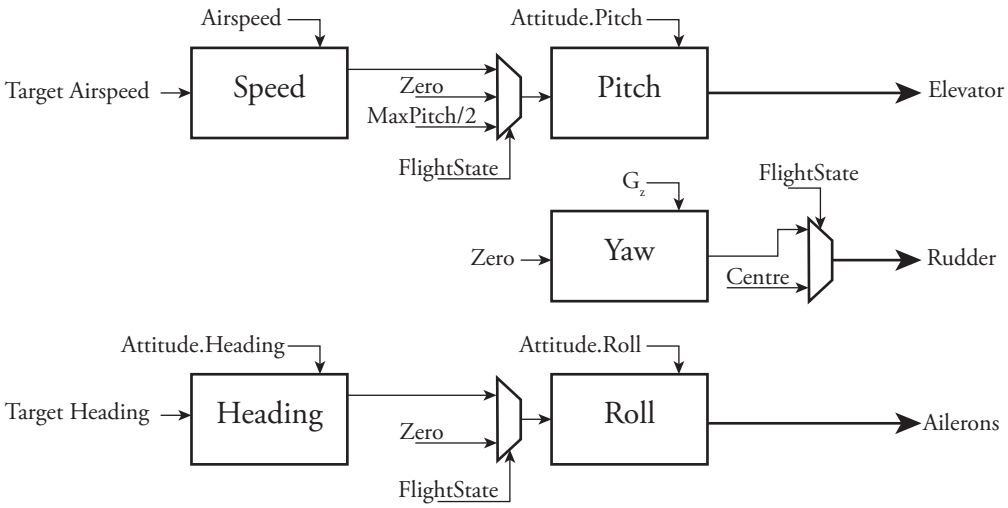

Fig. 38 Controller entity PID controllers.

this time with no power from the motor, so a gradual descent is expected. The heading of the aircraft is controlled towards the navigation computer's required bearing by rolling, which itself is achieved by actuating the ailerons. The rudder is only used to counter rotation about the Z axis during states which require wings level; it is not used during CRUISE and is left centred. This should be extended to maintain coordinated turns.

This simple control scheme and state machine is the specific example implemented for our own experimentation, but there is no reason other control schemes can't be implemented on this flight controller, whether it be a more typical fixed wing aircraft under constant power, or an entirely different craft such as a quad-copter or submarine.

Output Actuators

The actuators on our example airframe include 6 standard servo motors, and an electronic speed controller (ESC) for the motor, which also uses a standard servo interface. Servos are controlled by a PWM signal, typically with 1-2 ms pulses at 50Hz, with some servos extending their range to 0.5-2.5 ms and/or their speed up to 400Hz. The width of the pulse determines the absolute position of the servo.

This interface is quite simple to implement in an FPGA and at quite high resolutions given the orders of magnitude between a 20MHz system clock and a 50Hz PWM train. Each servo is given a counter with enough resolution to count 1024 increments from 1-2 ms from when it is started. A 12 bit counter incrementing at ~1.4Mhz that outputs high while the count value is below the required servo value can provide the modulation. We simply use

another counter to generate a 50Hz pulse used to reset all servos controllers causing them to produce the required pulse train.

The servo values are provided either by the controller, the pilot, or as fail-safe values. In addition, their ranges are limited digitally to match the extents of mechanical movements in the actual aircraft. These limits are provided at start-up from persistent memory but can be altered during runtime by the operator.

Manual Pilot Control

As this is an experimental platform, and because current legislation requires, it is highly desirable that a human pilot can take control of the aircraft at any point during its operation. For this reason, the DSM receivers have been included in the system design. These are a COTS receivers that pair with a Spektrum brand transmitter to provide low latency, reliable control to a human operator. In most experimental operations a human pilot is expected to be present and be able to take control of the aircraft at any time.

The DSM receivers provide the control inputs from the pilot over a unidirectional serial interface, with the framing and data resolution provided by the transmitter. These data frames are read into the FPGA and translated into servo values. One of the servo channels is dedicated to selecting the output used to drive the actuators, choosing between the output of the controller or the manual controls from the pilot. The dual DSM receivers provides redundancy which minimizes radio fading as the aircraft changes its orientation to the transmitter. When no signal is available, the MUX automatically falls back to the controller, which will either be active control or fail-safe.

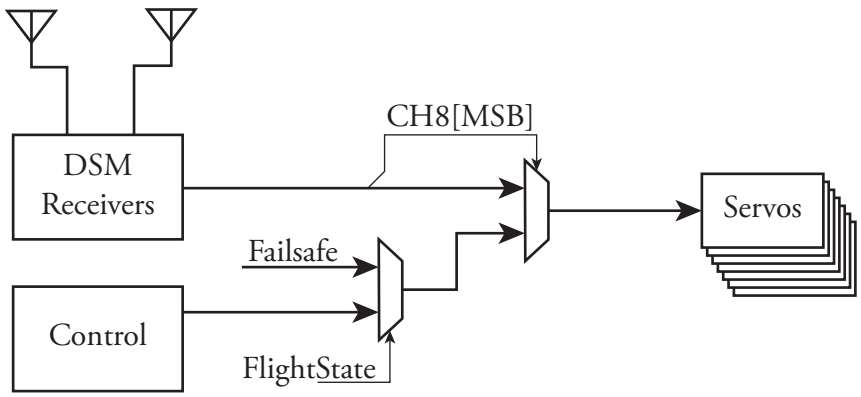

Fig. 39 Actuator control muxing.

A fail-safe state is included in the flight controller that when entered, will drive all actuators to a constant, pre-specified value that is intended to terminate the flight and impact the ground with minimal kinetic energy. These fail-safe values can also be over-ridden by manual control if the pilot wishes.

Miscellaneous Infrastructure

In addition to primary algorithms to control the aircraft, there are a few firmware components specific to implementation on an FPGA that are required to provide configuration and control to an operator. These two main pieces of infrastructure include loading data from persistent storage at start-up, as well as maintaining that data in the presence of corruption from radiation, and the communications infrastructure that allows an operator to change parameters while the aircraft is in operation. In a typical software implementation, such data registers are simply stored in RAM that doesn't have to consider the physical location of the data or the timing required to access it. However on an FPGA, one must consider the source of the data and how it is to be routed through the fabric to reach locations that are physically distant and have significant timing considerations. In a previous design, these paths actually extended beyond a single FPGA. We used two approaches to these including daisy-chaining and a prefix tree which we describe in detail.

Internal Lookup Tables

There are a number of lookup tables in the firmware that contain static data used to simplify what would otherwise be complex arithmetic. This data is typically just pre-computed and is not required to change during the operation of the aircraft, however the BRAMs are not physically capable of storing it persistently, so it must be loaded from persistent memory at start-up. Additionally this memory must be protected from SEUs.

To achieve this, we use a daisy chain technique where each memory element that requires data to be loaded is chained together sequentially with an attached "ROM Monitor" entity. The ROM monitor is responsible for loading and maintaining the integrity of the data in its attached memory element. Initially, each monitor entity is "invalid" in which case it will take serialized data from the daisy chain and store it in its attached memory element. On completion of the initial loading, it reads and stores a checksum from the daisy chain and then begins to forward all data to the next element in the chain. The checksum is included

as part of the data block stored in persistent memory, and is continuously recalculated by the ROM monitor at runtime. Any integrity issues result in invalidating the entire daisy-chain up to the point at which the fault was detected, causing all affected ROM monitors to reload their memory element. We illustrate this technique in Fig. 40. Using a checksum for an entire block rather than simply adding ECC to each word increases memory available to the algorithm in question. The BRAMs in the FPGA are all dual port, which allow read and write access to the memory by two independent processes simultaneously. As each lookup table is likely to be being accessed if/when it is reloaded, there is a possibility of corrupt data being propagated into the system, unlike ECC used elsewhere which corrects any errors as the data is accessed. As the memory is being rewritten with identical data, this will not affect most read operations. Only in the case of the corrupted word being read within the scrubbing period provided by the ROM monitor is a fault actually propagated into the rest of the system. As with all memory scrubbing techniques we can only reduce the risk of such a fault occurring but cannot mitigate it completely.

In addition to the static data lookup tables, there are also a number of registers that require default values, but that the values can be changed by the operator during runtime, and that we do not want to reload if a fault is detected upstream in the daisy chain. In this case we use a special node in place of the ROM monitor as the last element in the chain. This node is connected directly to the communications entity and provides an alternative source of command packets. This allows a number of register write commands to be included in the data stored in persistent memory which can be applied at boot. This node does not perform integrity checking after the initial loading, and without a node further upstream is never invalidated.

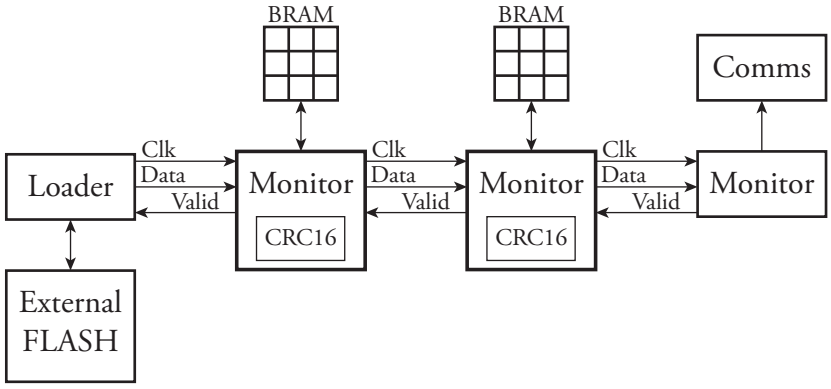

Fig. 40 ROM daisy chain.

Communications

Communication with the Ground Control Station (GCS) is achieved over a 900MHz wireless link. We use a bespoke protocol that fits neatly within the framing of the transceiver to provide deterministic telemetry, command & control as well as general purpose stream data. This protocol is describe in great detail in Real-time Wireless Communication.

The driver implementation is very similar to the drivers for any other device already discussed in that it includes a PHY to manage the physical interface to the device and a state machine to manage the behaviour of the driver itself. Telemetry packets are sent at a regular interval after which a window is provided for the GCS to respond with a command packet.

As the communications entity is a central source for many data registers throughout the FPGA, we must account for the large fanout and subsequent extended path delays. We achieve this by implementing a prefix-tree to deliver commands to their relevant locations. Essentially all commands are a write to a register address within the system. By grouping relevant registers together within the address space we can route the command packets through a physical logic 'tree' in the fabric. We can then add registers at each tree node to breakup the otherwise lengthy timing paths.

This scheme does not allow the reading back of control registers. It is expected that a GCS has complete knowledge of the configured state of the aircraft at all times. Any dynamic state to be communicated to the GCS must be included in the telemetry packets.

Radiation Hardening

As discussed in the abstract in Upsetting Logic, the general approach to radiation hardening & SEE mitigation of the system with respect to the firmware relies on providing redundant data allowing for the detection and correction of any fault caused by SEEs. A few features discussed so far in this chapter have touched on data integrity; here we will detail hardening techniques used throughout the firmware design as well as the specifics already mentioned.

Our design is synchronous with our system clock of 20MHz, this means that all inputs and outputs are driven by registers synchronous to this clock. While there is necessarily combinatorial logic between these memory elements, these paths are not considered susceptible to SEUs. Our radiation hardening focuses on the memory elements in the design,

which includes BRAMs and all flip-flops used in the design. This is not limited to elements storing data, but also those used to control behaviour and timing such as counters and delay elements.

Registers in HDL are easily distinguished by their clock sensitivity; that is their value is only altered on the rising edge of a clock. We follow strict templates for inferring registers in our coding style, and we only use the system clock directly as a clock source; there was no need for multiple clock domains within the FPGA or any gated clocks. By following standard templates it is easy to define exactly which signals in a design are registers, and it is therefore easy to extend the register behaviour to include error correcting codes.

```
process(Clk) is
begin
    if rising_edge(Clk) then
        if Reset = '1' then
            Q <= '0';
        elsif Enable = '1' then
            Q <= D;
        end if;
    end if;
end process;
```

Fig. 41 A simple inferred register.

We have implemented a library which provides for the calculation of hamming codes for arbitrary length logic vectors as well as the automatic correction of a vector encoded with error correcting bits. This library also provides helper functions to encode/decode existing standard types such as *signed*, *unsigned* and *std_logic_vector* to and from their ECC protected codewords. The VHDL implementation for this library is provided in Appendix B. Essentially, any register in the design must be encoded as an *ecc_vector* and decoded into its actual type. A typical register design in VHDL does not usually include a 'default' case when the register's reset or enable is not asserted. This is because the behaviour of an un-enabled register is simply to retain its value and does not explicitly require its old value being reassigned. However in order to maintain data integrity in the presence of SEEs, all memory elements must be "scrubbed". It is not enough to simply detect the error and correct when the register is accessed, as this increases the chance of a second error occurring in the codeword which would not be correctable. For this reason the memory element must be explicitly decoded and then re-encoded on every clock cycle. This is not as expensive as it sounds, as the logic resources are already consumed for these two activities and are simply reused.

The addition of error correcting circuitry does come at a cost of extra logic resources used, as well as increasing the timing paths between consecutive registers, limiting performance.

```
process(Clk) is
begin
    if rising_edge(Clk) then
        if Reset = '1' then
            Qecc <= (others => '0');
        elsif Enable = '1' then
            Qecc <= to_ecc_vector(D);
        else
            Qecc <= to_ecc_vector(Q);
        end if;
    end if;
end process;

Q <= to_slv(Qecc);
```

Fig. 42 ECC protected register with auto-scrubbing.

This is discussed in more detail in Upsetting Logic. The use of hamming codes reduces the need for extra memory elements as it scales with logN rather than N, but it does significantly increase the complexity of the logic to perform the encoding & decoding; increasing path delay. By limiting the data word length encoded into each codeword it is possible to balance the trade-off between logic resource requirements and timing constraints depending on what is more constrained within a specific design.

Block RAMs throughout the design must also be protected from SEEs. Most of the static lookup table elements used in the design are attached to monitors as discussed in Miscellaneous Infrastructure. These monitors use the second port of the BRAM to constantly read back the stored data and compare a pre-calculated checksum with a freshly calculated one. With the required ECC encoding of the stored checksum, this provides scrubbing of the BRAM in question. However for non-static data stored in BRAMs, such as waypoints etc. it is not possible to use a pre-calculated checksum or restore data from persistent storage. In these cases we again use our ECC library to encode data with error correcting bits as they are entered into the BRAM, and decode them as they are retrieved. These BRAMs require an extra scrubbing circuit, which also makes use of the second port of the BRAM. This simply reads the data and writes it back to the RAM after re-encoding it. This actually does double up on encoding/decoding logic, as the same logic cannot be used on both ports of the RAM.

It should be stressed that ECC encoding is used just about everywhere, not just on critical data elements. For example, the numerous counters used throughout the design each include a register to store their current value; this must also be protected as a single bit error in the count value can cause glitches in the output of the counter, which violates their safety requirements as detailed in Over Thinking Counters. There are however examples of registers that do not need ECC protection. These are usually things like buffers for asynchronous inputs which are updated every clock cycle and are expected to be tolerant of noise.

Conclusion

We have designed and implemented a complete functional firmware for the flight controller designed in Hardware Design. This includes drivers for peripheral devices and their required PHY interfaces as well as higher level control, navigation and communication algorithms. We addressed some concerns unique to working within an FPGA including loading and maintaining memory elements, managing resource consumption and protecting against SEEs.

Although we have a CPU available within the M2S025 we restricted the implementation to the FPGA to allow for verification and radiation hardening; both things we deem more difficult when implementing software on a CPU.

The resource consumption of the FPGA as reported by the Libero software suite is summarized in Table 6.

Element Type	**Total (M2S025)**	**Utilized**	**Percentage**
4LUT	27969	12107	43.71%
FF	27969	7967	28.77%
LSRAM	31	11	35.48%
uSRAM	34	0	0%
MATH	34	0	0%

Table 6 FPGA Resource Utilization.

As noted, this firmware was originally implemented on an FPGA that did not include embedded DSP/MATH blocks and instead uses iterative Booth multipliers in fabric logic. Migrating these to use hard MATH blocks would provide a significant saving of 4LUT and FF resources. Alternatively, as mentioned the ratio between system clock and sampling rate is quite large and many components that require a large number of resources (such as IMU and navigation) are sitting idle between samples. These elements could be optimised to reuse less elements in more sequential designs.

The inclusion of SEE mitigation on every memory element within the FPGA also significantly increases resource consumption. We provide a library that implements error correction for arbitrary length bit vectors. The resources required to implement these codes scales with the length of the vector as $\log(n+1)+1$. Delay paths also grow as the vector length increases, so it is usually necessary to limit the maximum length of encoded words, and break large words into smaller chunks. With a mean word length of 7 bits a design can be expected to increase 40% in FF utilization.

Real-time Wireless Communication

Wireless transmission of telemetry, command & control as well as any other acquired data is a significant consideration in most unmanned aerial platforms. Maintaining a deterministic telemetry and control link over an unreliable medium is a challenge, especially if that same link is being used to transmit image or sensor data en masse.

Here I describe the design and implementation of a real-time protocol that allows control and monitoring of an unmanned aircraft while allowing for the transfer of arbitrary bulk data collected by the aircraft. This protocol is implemented entirely within an FPGA to provide deterministic behaviour and maximise throughput. It is built upon a physical layer provided by the AT86RF212B ISM band transceiver.

The resulting protocol design allows for regular, deterministic telemetry data from the aircraft to the GCS and reliable command & control from the GCS to the aircraft. The remaining bandwidth is available for arbitrary data to be sent from the aircraft to the GCS, though no guarantees are provided for this bulk data. I also discuss the inclusion of a nondeterministic operator interface providing command & control data via the GCS.

Introduction

Unmanned aircraft operations usually require wireless communications with a ground control station (GCS) to monitor the status of the aircraft, provide any control data required and gather any data being collected by the aircraft. This leads to two conflicting uses for a wireless communications link. The first use is to maintain a deterministic command & control channel to monitor the aircraft and react to any situations in real time. The second use is to transmit large amounts of data that isn't required to be deterministic. One obvious solution would be to use two separate physical links, however I am trying to minimize space and power consumption.

The transceiver included in our avionics package is an Atmel AT86RF212B [82] with a front end 500mW amplifier [83] to extend the range of the link to at least 15 Km. This transceiver will be transmitting in the 915 - 928 MHz ISM band at a data rate of 250 Kbps. The radio employs direct sequence spread spectrum (DSSS) to spread its bandwidth and lower interference. Channel selection is fixed and no frequency hopping will be used which greatly reduces the time spent synchronizing two remote nodes. This device provides both the UHF physical layer as well as a simple "MAC" layer which will constrain out protocol design.

One challenge in working with FPGAs is that no central shared memory exists. It can exist, but is not usually an efficient use of logic resources. Instead, control parameters such as waypoint targets, servo positions and update rates are stored in registers distributed throughout the device, typically near the block they affect. A mechanism for writing commands to these distributed registers will also be discussed.

The transceiver in the aircraft communicates with an identical transceiver on the ground; I actually use the same flight controller hardware with a different firmware. The device on the ground then communicates with the operator interface, a laptop or tablet, either over a WiFi connection, or through USB.

Design

The design includes packets sent across both the wireless link and the wired link on the ground. It also includes a lot of detail of the behaviour of each node with regards to timing and reliability. As the wireless link is the bottle-neck, and the radio constrains the packet sizes, the wireless protocol is the logical place to start the design.

Wireless Protocol

The radio transmits and receives data in packets of varying length, up to 128 octets [82]. The first octet is a length field that specifies the number of octets remaining in the packet from 0 to 127. The final two octets of the packet are a checksum that is generated by the transmitting radio and checked by the receiving radio. Any packet that fails this checksum will be dropped. The remaining 125 octets will contain the following protocol.

The wireless link must transfer three different types of data. Telemetry data from the aircraft, command data from the GCS and any bulk data such as images from the aircraft. As wireless communications are unreliable I assert that all packets must be expendable, any packet can fail to reach the other end. In some cases this is acceptable while others will need to mitigate this effect with error detection and retransmission or possibly forward error correction.

To achieve deterministic packet timing and arbitrate a half duplex wireless link, a master/slave relationship will be used. In this scheme, one node broadcasts unsolicited packets at an interval of its choosing. The slave node then has a fixed window in which to respond. Outside of this window, the slave node must remain silent to avoid colliding with the master. This is a simple approach and appropriate for this point-to-point link.

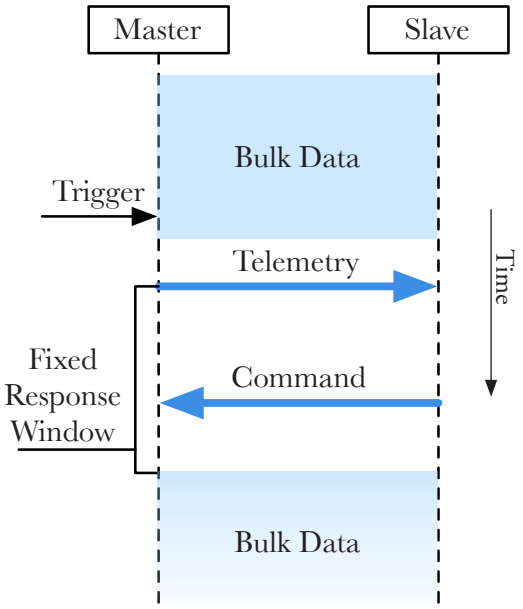

Fig. 43 Master/slave packet sequencing.

In the event of asymmetric packet loss, it is desirable to maintain as much information about the status of the communications link and of the aircraft as possible. For this reason the aircraft node will be defined as the master so that it continues to transmit telemetry data

in the absence of command packets. Both nodes must be able to determine the status of the link, which requires both nodes to send and receive packets regularly. This leads to a simple packet schedule where telemetry packets are sent in regular intervals from the aircraft, and command packets are only sent in response to telemetry packets. As the master knows exactly when the channel will be empty and has all the bulk data to send, it can use a simple priority queuing mechanism to schedule telemetry and bulk data. Bulk data packets can be transmitted continuously until a telemetry packet is required. After the telemetry packet is sent, the channel is left empty for a fixed window or until the slave responds with a command packet, after which, bulk data packets can continue.

It is desirable that the telemetry and command packets be kept as small as possible to leave bandwidth for bulk data. Telemetry data will be constrained to a single packet that will contain fixed data fields. As changes to parameters are expected to be quite rare, command packets will allow for *optional* command bytes, leaving the standard command packet to the bare minimum data required. This does allow commands to be lost, and so an acknowledgement for each command packet must be included. As telemetry packets are also susceptible to being lost, a single attempt at acknowledgement is insufficient. To enable command acknowledgments, each command packet will have a sequence number and each telemetry packet will contain the sequence number of the last command packet it received. If a command packet is lost, then the sequence number in the telemetry packet will reflect that, while if a telemetry packet is lost, then no further command packets will be sent anyway. This could actually be done by a single bit 'flag' rather than a full octet, but this enables simpler debugging. Retrying lost command packets introduces a source of non-determinism as the process is essentially random, based on the link quality at the time. The host PC will be responsible for retrying commands. However, the current behavioural 'intent' should be included in every command packet. This is a single value that specifies the high level activity or state the aircraft should be in such as 'track current path', 'loiter', or 'failsafe'. Similarly, the current action/state of the aircraft should be transmitted in every telemetry packet.

We can now define three packet types designed to communicate data over the wireless link. These are telemetry, command and bulk. Telemetry is sent from the aircraft to the GCS at

	0	1	2	3	4 … N-2	N
Telemetry	Length	Header	State	Seq	Telemetry Data	Checksum
Command	Length	Header	Seq	CMD	Optional Command Data	Checksum
Bulk	Length	Header	Bulk Data			Checksum

Fig. 44 Wireless packet structures.

a fixed rate. Command packets are sent in response to telemetry packets, and bulk packets are sent whenever the master considers the channel empty. Each packet has a length field specifying the number of bytes in the packet from 5 to 127 (excluding the length field itself). Then each packet is identified by a header byte that is unique to each packet type. The last two bytes of each packet is a checksum of the bytes 1 to N-2 and is generated by the transceiver.

The Command Tree

Delivering parameter data from the host PC to specific registers in the FPGA on board the aircraft is not a trivial task. Beyond the need to reliably communicate the data, the logic within the FPGA must deliver data to arbitrary registers throughout the device from a single point of entry. Using a centralized memory would lead to very poor 'routability' and timing of the firmware design. To simplify this I opted to organise the control registers into a prefix tree.

A command consists of an address that traverses the tree and delivers the remainder of the command, the data, to a register. The size of the address and the data are arbitrary and can vary between branches as long as sibling nodes share the same prefix length. This allows commands to write individual registers of any length, address the full width of a BRAM or just trigger an event without data. By splitting the tree over the GCS and the aircraft, the host PC can address registers in either device through the same interface. This even extends to other devices attached to the flight controller such as the radio or a CMOS camera. If

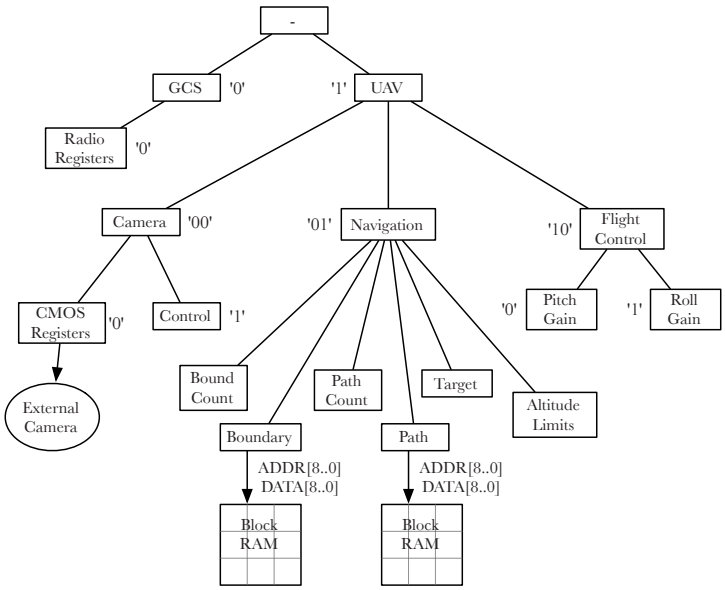

Fig. 45 An example command tree.

the drivers provide an interface to the internal registers they can be written directly from the host PC.

Command data propagates through the tree as it arrives and is immediately applied to its target register. There is no need to buffer or queue the data once it has arrived at the remote node. There is also no need to propagate the 'used' portion of the prefix, so the bus width reduces as it fans out.

Each command is a write only action of some data to an addressed register. The address space of the tree must allow enough bits to traverse the tree to the target leaf node. Fig. 45 illustrates this concept as an example tree that would actually fit the example system being implemented. To change the 'roll gain' register, for example, would use the prefix "1101". The largest commands being considered are writes to the waypoint path data. These need to include the tree prefix as well as the 9 bit address for the block RAM and the data to write. This gives the maximum length of a command 24 bits. The prefix is always left aligned in the command, while the data can be more flexible but as a rule of thumb it should always be right aligned.

Host PC Protocol

The communication from the host PC to ground control station is a 4 Mbps, full duplex serial link. As the host will most likely be running a consumer operating system, it is not expected to be capable of deterministic communications over this link. The GCS and aircraft therefore act independently of the host PC, and the host PC is not included in the hard real-time activities. This means the host PC is mainly acting as an observer of the data being sent from the aircraft and sporadically issuing commands.

This lack of responsibility greatly simplifies the interface with the GCS. Telemetry and bulk commands can simply be forwarded, unaltered to the host PC. We simply need to define a way to transfer command bytes to the GCS while maintaining their integrity and confirming that they reached the aircraft.

Commands are all 3 bytes long and contain a prefix to traverse the command tree followed by any data for the target register. It is critical that these bytes are protected by a checksum as addressing the wrong register may be catastrophic. Therefore each command will have a checksum appended which is simply the exclusive 'or' of the three command bytes. Any command quad that fails the checksum at the GCS will be dropped. The checksums are

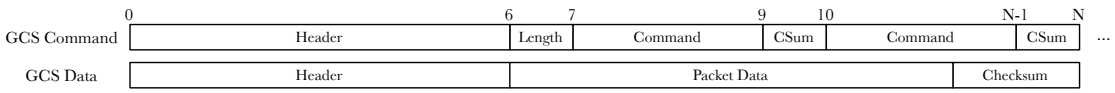

Fig. 46 Host PC to GCS packet structure.

only to protect the commands over the host PC to GCS link as the radio checksums protect the data over the wireless link.

A packet is formed by many command quads and a length byte specifying the number of command quads included. As the link is just a serial stream, there is a need to synchronise packets between the host PC and the GCS. This is achieved by adding a string of 6 bytes to be matched and recognised as a packet header. These bytes should be unlikely to occur in the data stream itself. One condition on the selection of these bytes is to make sure that no combination of 4 consecutive bytes matches the command checksum. Given the simple checksum, this simply means the bytes can't all be identical.

Commands arriving at the GCS will be delivered in order to the aircraft, however they will be organised into new packets for the wireless link. Each command arriving at the GCS will be added to a queue to be included in the next free command packet. No guarantee can be made as to when each command will be transmitted, or if it arrives at the aircraft. To ensure a command was delivered we must first make sure it was sent. Each command packet sent to the aircraft will also be echoed to the host PC verbatim. The host PC can then check the commands included in that packet, and match the sequence number to a future telemetry packet. As all commands will be delivered in order if they are in fact delivered, this will provide confirmation.

In order to provide the host PC with a complete picture of the status of the link and ground control station, a new packet type is added to those listed in Fig. 44. This is a simple GCS telemetry packet and is identical to the aircraft Telemetry packet with the sequence number and state fields removed.

All packets sent over the wireless link and GCS telemetry packets are sent to the host PC, this includes command packets being sent to the aircraft. They are echoed verbatim over the serial link, with the addition of a 6 byte synchronisation header and a 16 bit checksum of the entire packet. While packets coming from the wireless link will already have a checksum, not all packets came over the wireless link, so for simplicity, all packets are given an extra checksum (the existing one is left in place).

Implementation

The implementation of the protocols described above will be completely written in VHDL and synthesised to run on the avionics and GCS platforms. The aircraft includes the telemetry gathering and packet generation, priority queuing and an interface to the radio hardware. The GCS includes an interface to the host PC, command gathering and packet generation, GCS telemetry generation and an interface to the radio hardware. Fig. 47 illustrates a block diagram of the intended implementation of the master end of the communication channel. Similarly Fig. 48 illustrates the implementation of the GCS firmware. The colouring indicates the protocol layer in a similar fashion as the OSI model [34]. Blue blocks implement the physical/transport layers, red blocks are similar to a session layer and green blocks show the application layer.

The blue 'transport' layer blocks include the radio interface and the serial link interface of the GCS. These blocks are responsible for managing the physical link to the hardware as well as packetising and scheduling data transfers over those links. They manage the rate at which data transfers occur.

The red 'session' layer blocks are only required on half duplex links such as the radio. They manage the direction of the link and only allow data to pass in the slots allocated to that data stream.

The green 'application' layer blocks are where the data packets originate. They gather all the data relevant to a particular packet type and arrange it into a packet as described in Fig. 44. They can be triggered to transmit their data at certain intervals or in response to certain event or even just left to transmit continuously. When a green block is ready to send, it signals the red block it's attached to and waits for it to pull data forward.

Master Communication

The master side of the communication link is the simpler of the two. It needs to gather telemetry and bulk data and transmit them both to the ground while occasionally stopping to listen for commands.

TelemetryPack has many inputs for all the registers that contain telemetry data to communicate. There are 122 bytes available in each packet and as of the time of writing only 71 of these are being used in our system. TelemetryPack is triggered by a timer that generates a

pulse at the required frequency of telemetry packets. When the trigger fires, all data inputs are registered in a single, long shift register and TelemetryPack waits for its data to be shifted out.

BulkPack has a similar job to do, but the data sources are typically simpler byte-wide streams. Each source of bulk data must be scheduled and identified which does depend on the application of the bulk data streams. We won't go into detail here and instead assume a single source of bulk data that is always ready to provide a constant data byte.

MasterComms manages the timing of telemetry and bulk packets as well as holding the window open for command responses. Outside of the command window, MasterComms will pass through data from either TelemPack or BulkPack as it is available. However if TelemPack is ready it will receive priority and be allowed to send after the end of the current packet. This forms a priority queue that sends telemetry packets as soon as possible. After each telemetry packet the channel is left open for a fixed window. In that window a command packet should have begun transmission and will then hold the radio module until it completes. Therefore the command window only needs to last until the expected start time of the command packet; it does not need to include the duration of the command packet itself.

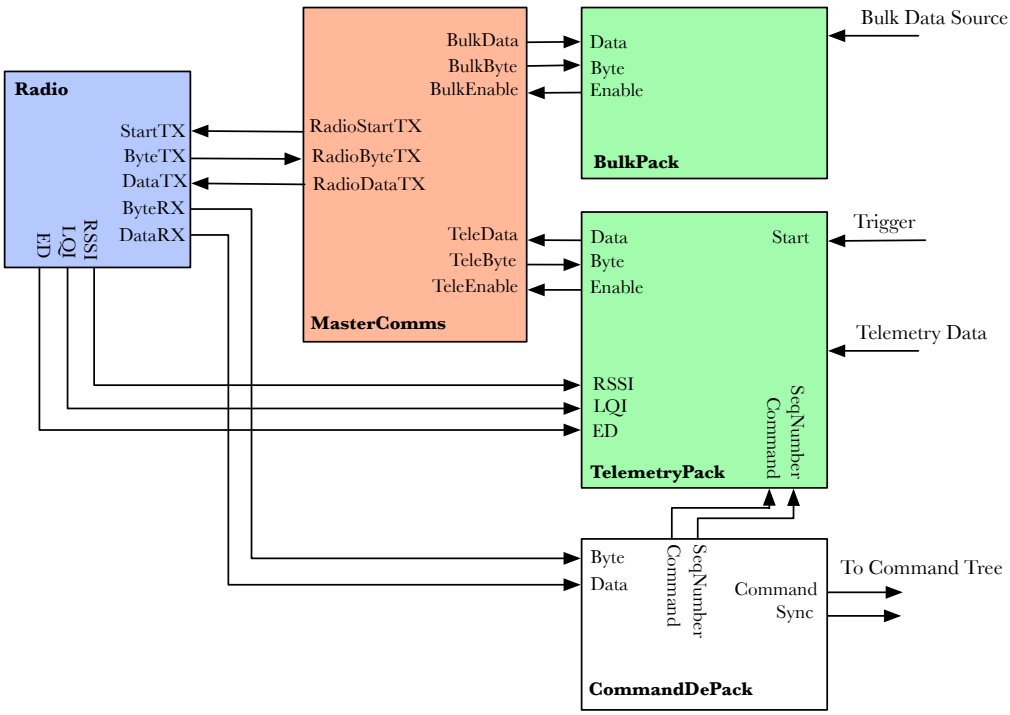

Fig. 47 Block diagram of master implementation.

CommandDepack interprets command packets as they are received and issues commands to the command tree. As this is the only packet expected by the aircraft it is attached directly to the radio output.

Finally the radio block manages the SPI interface and interrupt lines to the radio hardware. This block is identical in the master and slave implementations and will be covered in more detail later.

GCS Communications

The GCS node needs to talk to both the host PC and the aircraft. Communication with the aircraft is required to be deterministic while trying to maximise throughput. Communication with the host PC is simply relaying data received from the aircraft while gathering and forwarding any commands received. As the host PC link is much faster than the wireless link to the aircraft, the host PC interface is kept simple.

The GCSCommsRX block listens to the host PC serial link and waits to hear a synchronisation header. Once a packet is received, commands are integrity checked and either propagated down the GCS command tree or stored in a FIFO buffer to await transmission to the aircraft.

CMDPack takes any data available in the FIFO and stores it in a large shift register to build up the next command packet. It will keep appending commands as they are available until the packet is full or it is triggered to send. If the command packet is not yet full, a short sorting occurs to align gathered commands with the start of the packet. This allows command packets to be dynamically sized depending on how many commands are available. Each command packet can contain up to 40 commands. Any commands remaining in the FIFO or arriving while CMDPack is sending will wait in the FIFO. The command sequence number is incremented on each packet and wraps around to 0 from 255.

SlaveComms does very little to manage the timing of packets from the GCS. In fact command packets can be transmitted through SlaveComms at any time. The actual timing is left to TeleDePack. SlaveComms simply re-synchronises the signals from the radio clock domain and passes all data into the FIFO buffer. This includes command packets which are simultaneously written to the radio and the FIFO buffer. Once a packet is completely received or transmitted, SlaveComms sends a pulse to signal GCSCommsTX to relay it to the host PC.

TeleDePack listens to all packets moving through the radio looking for telemetry packets. When it detects a telemetry header it signals CMDPack and GCSPack to generate and send a

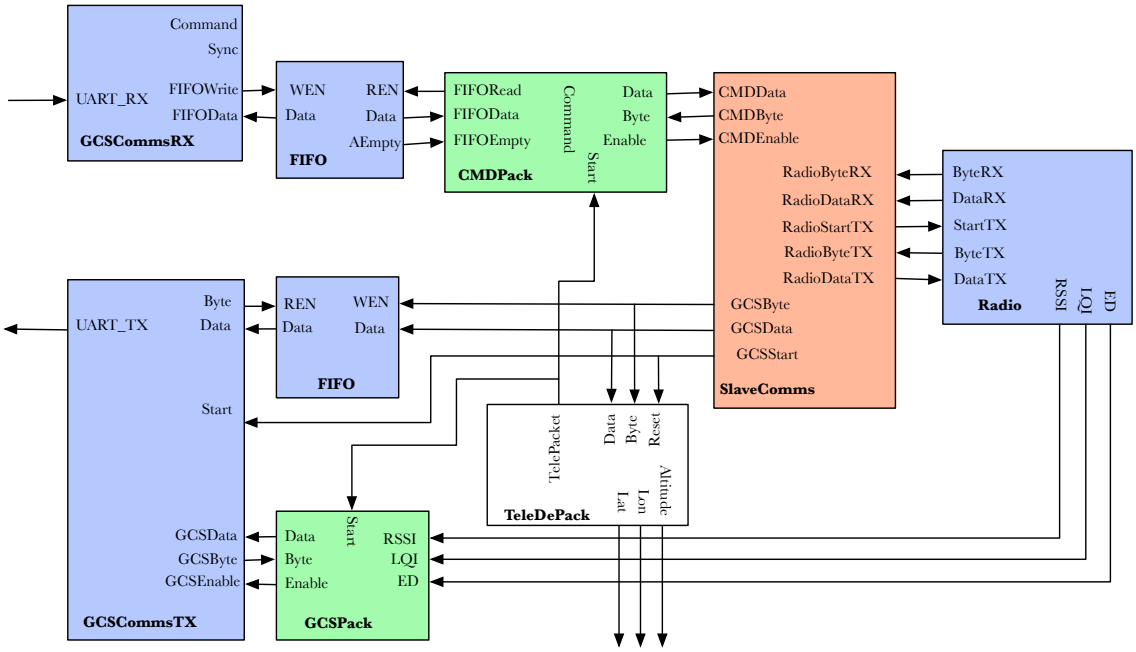

Fig. 48 Block diagram of the slave implementation.

packet. This is the cause of command responses to telemetry packets. Timing here is not important as CMDPack will be ready to send before the telemetry packet has been completely received and so will transmit at the earliest opportunity. TeleDePack also gathers some fields from the telemetry packet for use within the GCS device. This includes aircraft position information for use in active antenna tracking etc.

GCS telemetry packets are generated by GCSPack which simply gathers radio status information and sends them to the host PC. It is triggered whenever TeleDePack finds a telemetry packet.

The GCSCommsTX block has the simple job of simply adding a synchronisation header to the packet data and appending a checksum. As the serial link is full duplex there is no need to schedule data transfers. The GCSCommsTX block will wait for a start signal indicating the radio has finished writing a packet into the FIFO, or that a GCS telemetry packet is ready to go. As multiple packets can arrive in the FIFO while one is being transmitted, the start triggers are queued in a small shift register to track multiple pending packets.

Synchronous Radio Controller

In order to maximise throughput and lower power consumption in the FPGA, a controller clocked by the radio can be used. This involves taking the CLKM output from the AT86RF212B and using it to clock registers inside the FPGA. This effectively lowers the power consumption of those registers by clocking them at a much lower frequency than the

system clock. It frees up logic that would be needed to synthesise correct bus speeds. Finally it allows data to be read and written to the frame buffer inside the radio at the exact same rate it is received or transmitted over the air, reducing latency and improving throughput. The AT86RF212B uses a dual port RAM for its frame buffer, allowing data to be read or written while the packet is still being received/transmitted.

In order to maximise throughput, the time spent not transferring packet data to or from the frame buffer should be minimised. After the initial setup, the data bus is only used for transferring packet data, clearing the interrupt register and initiating some state changes. The state changes and interrupt clearing are both a single register access, and so do not occupy much time, but it is still significant. The order of operations needs to be carefully selected to align data bus uses with inactive states of the radio, such as while it's transitioning between states.

Fig. 49 illustrates the state machine implemented in the synchronous controller, overlaid with the intended state transitions of the radio hardware. The device initially goes through a setup phase that writes the required configuration to the registers. Once set up, the controller will begin a receive loop which sits idle until either a packet is received or a packet is ready to be sent. Transitions through the states TO_RX or TO_TX both require a register access to initiate the state transition in the radio. Likewise, entering the states WAIT_RX or TRANS_END both access a register to clear the interrupt pin.

The data interface sits idle while a packet is being received. Once the checksum is verified, the packet data is retrieved from the frame buffer. Another packet could quite possibly be received while the first is still being read from the frame buffer. This would generate its own interrupt on completion and also be retrieved.

A packet transmission is initiated as soon as the frame buffer access to write it begins. The radio must transmit a synchronisation header first, giving the controller a head start to complete writing the packet. In a continuous transmission mode, where no response is permitted the radio transmitter is very rarely idle, maximising data throughput. The most significant factors to throughput are the packet overhead, synchronization header, and unavoidable radio turn around times.

When a response is expected, the radio must transition to the opposite state. This requires non-data related access and a non-trivial turn around time. The transmitting side must also

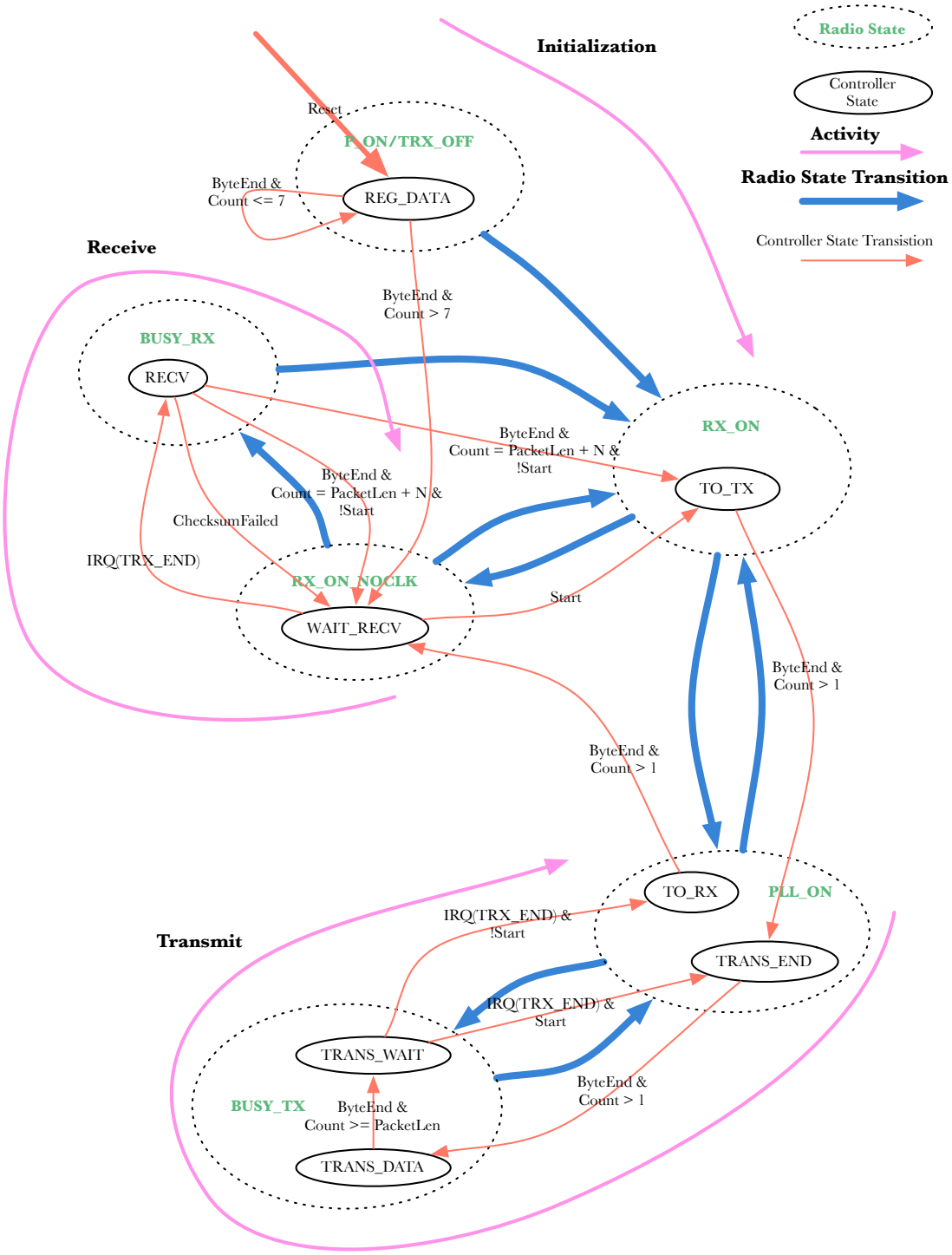

Fig. 49 Radio driver state machine overlaid with transceiver states.

wait idle while the receiver retrieves the packet from the frame buffer after checksum validation before it begins to transmit its response.

In a continuous transmission, based on simulation results and timing data about the AT86RF212B, it takes 4344 us to begin transmission of one packet of maximum length, complete its transmission including synchronisation headers and return the radio to the state it was before transmission started. This allows 230.203 packets of 127 bytes to be transmit-

ted per second. This is a theoretical throughput of 233.89 Kbps. Not including SPI accesses and radio state transitions, a telemetry exchange requires 4 packet time slots to complete. With telemetry and command packets of maximum length, exchanged at 10 Hz, the bulk data throughput is reduced to 193 Kbps. However using the current size of telemetry packets as 71 bytes, and the minimum (and common) size of command packets of 5 bytes a bulk data rate of 221 Kbps is achievable.

Results

The firmware for both the GCS and the aircraft was successfully implemented and run on their respective devices. Both met the timing constraints of a 50 MHz system clock and a 1 MHz radio clock. The master design occupied approximately 12 % of a ProASIC31000, while the slave occupied around 20 %. This logic consumption was a bit higher than expected but still comfortable to use in the complete system.

Wireless communications were tested, as was the serial link to the host PC. Both communication links operated successfully and data could traverse the network as designed.

The time between telemetry packets was measured by the GCS FPGA's system clock to give an estimate of jitter within 20 ns. A thousand samples were taken at each 1 Hz update interval from 3 Hz to 100 Hz. From 3 Hz to 49 Hz jitter was bounded to about 2 us. This is consistent with half the clock period of the radio and reflects the fact that the system clock and radio clock are not synchronous. There was also a slight jitter of 20 ns. This is likely caused by the counter used to trigger telemetry packets occasionally rounding one clock edge further than the last, which is expected behaviour for this technique.

After 49 Hz, the wireless link began to drop packets, seemingly at random. This issue was far more apparent when continuous transmission was attempted, with the link losing roughly 1 in 4 packets. I believe this to be a timing bug or inconsistent radio state and not an indication of poor link quality. This did prevent me taking any meaningful measurements of bulk data throughput or telemetry packet jitter in the presence of bulk data.

The command tree proved to be a very simple concept to implement, easy to route and very flexible with the data being addressed. Commands were able to be issued from the host PC and their affects observed on both the GCS and the aircraft nodes.

Discussion

We successfully implemented the required communications between an aircraft and a ground control station. While there are still a few bugs in the firmware, it appears that this work will be effective to observe and control an autonomous aircraft in flight.

Implementing a synchronous radio module has many advantages and disadvantages and it is unclear if it is the best solution at this time. By making it synchronous to the radio clock, it is made asynchronous to the rest of the device. While this requires less power due to the much lower clock rate, it requires extra effort and logic resources to re-synchronise.

The radio clock is also kept from being useful to other parts of the design given that it is not available when the radio is in its sleep state. The small increase in power consumption may be worth the simplification in the radio state machine as well as providing a clock for other low frequency components. There is still a possibility that running the SPI interface to the radio at a higher rate than the radio baseband (up to 8 MHz) could improve on the performance achieved in this implementation.

Waiting for checksums to be verified impacts throughput. Without the checksums, packets can be read from the radio as they are being received, leaving the radio free to transmit as soon as the packet is complete. This could reduce the turn around time during a telemetry/command handshake by the entire length of a packet. As it is implemented, the radio hardware sits idle while the received packet is being retrieved. A solution to this is to verify the checksums inside the FPGA, taking the packet out-of-band as soon as possible and freeing up the radio hardware. Of course responses could not make use of data received in the most recent packet.

Firmware Verification

In previous chapters we have discussed formal verification of digital logic designs using generic examples in order to demonstrate the theory and provide a context for the design work that followed. Now that we have a complete design of the avionics firmware, we can extend the formal verification to concrete examples as well as cover a few more concepts as they can be used in practice.

While verifying every component of the firmware is beyond the means of this author, a select chain of components will be used to illustrate the concepts required to perform a complete verification if time allowed. These components will follow the data path from the altimeter, including the serial interface, device driver & higher level behavioural structures that interact with this data, then into the control algorithms and finally the output actuators.

Verification will be performed using first order logic and automated satisfiability solvers as well as metric temporal logic as a more human friendly specification language.

Introduction

Here I provide the formal verification of safety properties of a few selected components in the firmware design detailed previously. The selected components form a chain from a sensor input through to a behavioural function and actuator output. I am only focusing on safety and liveness properties of each component; while verification of functional behaviour can be achieved with these same techniques, such proofs are very application specific and don't serve the purpose of these examples.

Our verification demonstration centres around the fail-safe behaviour of the aircraft. During autonomous operation of the aircraft, any deviation below a minimum altitude should cause the flight controller to enter a FAILSAFE state. The required behaviour during FAILSAFE is that all servo actuators are driven to operator specified, constant values. I will start with the altimeter driver and its SPI PHY and work our way through the firmware system, ending with the servo actuators.

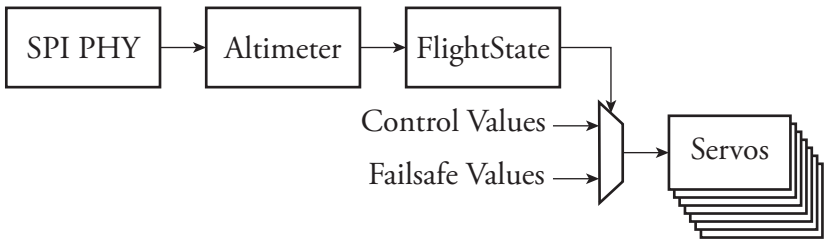

Fig. 50 Entity path for example verification.

The verification of the safety properties of the selected components will demonstrate techniques that can be used to verify common patterns in HDL design. These patterns include finite state machines, counters (timers) & MUXs.

The formal methods I use here are simple first order logic [5] as introduced in Over Thinking Counters. I also use metric temporal logic [6] as a more concise shorthand for the same constructs I express in first-order logic.

On Error Correcting Codes

As much of this thesis has been in discussion of defending data storage elements against SEEs we must quickly address how various error correcting circuits affect our formal methods. The first point to be made is that they don't have any effect on the timing of the circuits. This is not to say that they don't increase propagation delay, but they do not include any new syn-

chronous elements that would add clock cycles worth of delay. For codeword lengths up to 24 bits (the longest in the firmware design) it is not unreasonable to exhaustively test every input value with every single bit error on a typical desktop computer. This allows us to prove the encode/decode functions perform as designed, and are the inverse of each other simply by brute force. Finally, as the formal methods used in this chapter are only applicable to purely rational designs, any SEEs cannot be reasoned about, as they are outside the rational scope of a standard HDL design. More formally

$$\text{decode}(\text{encode}(x)) = x \qquad [8.1]$$

and unless stated otherwise

$$\forall t\,(x(t) \Leftrightarrow x(t+1)) \qquad [8.2]$$

As the encoding and decoding of error correcting codes is essentially a null operation with regards to our formal methods we will omit them from further discussion. It should be noted that some synthesizers might also take this approach to redundant logic elements in a design.

SPI Physical Interface

The SPI PHY provides an interface between a device driver and the physical peripheral device over an SPI bus. This element consists of a simple state machine and a few counters that translate a simple internal interface into a synchronous serial bus with complex timing requirements.

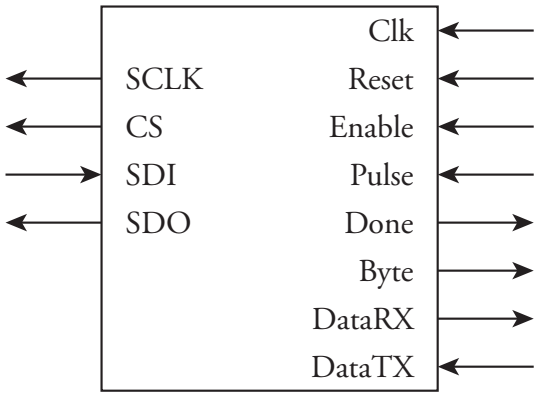

Fig. 51 SPI PHY Entity interface design.

The SPI PHY works by generating 'transactions' which include one or more bytes sent over the serial link. Transactions are started by asserting the *enable* pin, which will commence sending the first byte. At the completion of each byte, the PHY asserts the *byte* pin, which signals the driver to either provide the next data byte or end the transaction by de-asserting *enable*. As the SPI interface is full-duplex, receiving bytes occurs at the same time as sending them, so the same *byte* assertion signals valid data on *DataRX*. If the driver signals to end the transactions, the *done* pin is asserted a short time later and the peripheral chip is deselected by the PHY.

In order to meet the timing specifications of an SPI interface, transitions on the wire are aligned with the *SCLK*. To achieve this, a *pulse* is provided by the driver which defines the baud rate of the interface. All transitions of the state machine and output pins are only performed on assertions of the *pulse* input. There are timing requirements between the CS pin and valid data on SDI and SDO which are managed by the state machine.

We begin by specifying the interactions between the client of the PHY (typically a device driver) and the PHY itself, which involves the *enable*, *pulse*, *done* and *byte* pins of the entity. We assume a valid Clk is provided and that reset is always de-asserted. The client must provide a *pulse* input which defines the baud rate of the PHY by asserting at regular intervals. This input can be driven by the output of a counter as discussed in Over Thinking Counters. To prove liveness of the SPI PHY, we are not too concerned about how long things take, only that they happen eventually. To keep things simple we make the assumption that a pulse will always eventually occur, which is consistent with our previous counter implementation:

$$\forall t\big(\exists \tau : \tau \geqslant t \wedge \mathrm{pulse}(\tau)\big) \qquad [8.3]$$

The client controls transactions through the *enable* pin and is notified of events through the *byte* pin, which is asserted at the end of each byte within a transaction; and the *done* pin, which is asserted after the PHY has completed the transaction. A transaction starts when enable is asserted along with pulse while the PHY is in the IDLE state. The transaction will continue for many bytes as long as enable is asserted. When the client de-asserts enable, the PHY will complete the transaction and return to the IDLE state. There is a brief 'cool-down'

as the PHY completes the transaction during which time new transactions cannot be started. This leads to this requirement on the enable pin:

$$\forall t(\text{enable}(t) \wedge \neg\text{byte}(t) \Rightarrow \text{enable}(t+1)) \qquad [8.4]$$

and these expectations

$$\forall t(\text{enable}(t) \wedge \text{pulse}(t) \Rightarrow \exists \tau : \tau > t \wedge \text{byte}(\tau)) \qquad [8.5]$$

$$\forall t(\text{byte}(t) \wedge \neg\text{enable}(t+1) \Rightarrow \exists \tau : \tau > t \wedge \text{done}(\tau)) \qquad [8.6]$$

Briefly, the client begins a transaction by asserting pulse and enable simultaneously and continuously asserting enable until byte is asserted. The client is guaranteed that byte will eventually be asserted after a transaction is started, but is only permitted to de-assert enable on the next clock edge after byte is asserted.

We now come to the implementation of the SPI PHY which includes a state machine and a counter. There are other elements such as shift registers and other logic to manage the actual data propagation, but we will focus only on the behavioural elements.

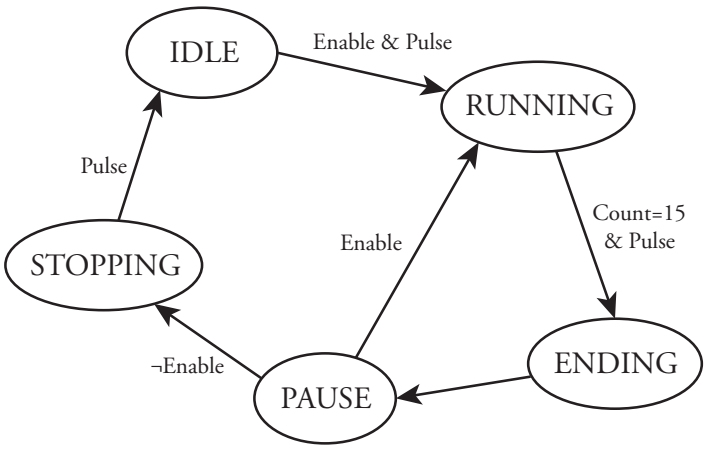

Fig. 52 SPI PHY Finite State Machine.

```
type STATE_T is (IDLE, RUNNING, ENDING, STOPPING, PAUSE);
signal NextState, State : STATE_T;

process(Clk) is
begin
    if rising_edge(Clk) then
        if Reset = '1' then
            State <= IDLE;
        else
            State <= NextState;
        end if;
    end if;
end process;

process(State, Enable, Count, Pulse) is
    begin
        case State is
            when IDLE =>
                if Enable = '1' and Pulse = '1' then
                    NextState <= RUNNING;
                else
                    NextState <= IDLE;
                end if;
            when RUNNING =>
                if Pulse = '1' and Count = 15 then
                    NextState <= ENDING;
                else
                    NextState <= RUNNING;
                end if;
            when ENDING =>
                NextState <= PAUSE;
            when PAUSE =>
                if Enable = '1' then
                    NextState <= RUNNING;
                else
                    NextState <= STOPPING;
                end if;
            when STOPPING =>
                if Pulse = '1' then
                    NextState <= IDLE;
                else
                    NextState <= STOPPING;
                end if;
            end case;
    end process;
```

Fig. 53 Example FSM Implementation.

The finite state machine of the SPI PHY is illustrated in Fig. 52 and more formally specified in equations [8.7], [8.8], [8.9], [8.10] & [8.11].

$$\begin{aligned} &IDLE_t \wedge enable(t) \wedge pulse(t) \Rightarrow RUNNING_{t+1} \\ &IDLE_t \wedge \neg(enable(t) \vee pulse(t)) \Rightarrow IDLE_{t+1} \end{aligned} \quad [8.7]$$

$$\begin{aligned} &RUNNING_t \wedge count(t) = 15 \wedge pulse(t) \Rightarrow ENDING_{t+1} \\ &RUNNING_t \wedge (count(t) \neq 15 \vee \neg pulse(t)) \Rightarrow RUNNING_{t+1} \end{aligned} \quad [8.8]$$

$$ENDING_t \Rightarrow PAUSE_{t+1} \quad [8.9]$$

$$\begin{aligned} &PAUSE_t \wedge enable(t) \Rightarrow RUNNING_{t+1} \\ &PAUSE_t \wedge \neg enable(t) \Rightarrow STOPPING_{t+1} \end{aligned} \quad [8.10]$$

$$\begin{aligned} &STOPPING_t \wedge pulse(t) \Rightarrow IDLE_{t+1} \\ &STOPPING_t \wedge \neg pulse(t) \Rightarrow STOPPING_{t+1} \end{aligned} \quad [8.11]$$

We have used the following shorthand to simplify specification of the state variable:

$$s_t \equiv \text{state}(t) = s \qquad [8.12]$$

A counter is used to count the number of bits in each byte and forms part of the state. This counter's behaviour is specified as:

$$\begin{aligned} &\neg(\text{RUNNING}_t \vee \text{ENDING}_t) \Rightarrow \text{count}(t+1) = 0 \\ &(\text{RUNNING}_t \vee \text{ENDING}_t) \wedge \text{pulse}(t) \Rightarrow \\ &\qquad \text{count}(t+1) = (\text{count}(t)+1)\text{mod}16 \end{aligned} \qquad [8.13]$$

The last implementation details we have to specify are the byte and done signals:

$$\text{ENDING}_t \Leftrightarrow \text{byte}(t) \qquad [8.14]$$

$$\text{STOPPING}_t \wedge \text{pulse}(t) \Leftrightarrow \text{done}(t) \qquad [8.15]$$

It should be noted that the equations above are a direct translation of the VHDL illustrated in Fig. 53 used to implement the SPI PHY, in which the combinatorial components are essentially written as first order logic.

To verify the liveness of this design and the specification of the interface, the designer must prove that [8.5] and [8.6] hold, given the assumptions and conjecture provided. This is usually achieved with the use of an automated theorem prover [7] [8] [9]. The conjecture above would be translated to a specification language such as SMT2 and a theorem prover would be used to check for consistency. The theorem provers I tried often struggled with non-linear algebra such as that used for the counter in this example. One possible work around was to bound the time interval over which this theorem should hold, but this has consequences with regard to rigour. Alternatively the designer could perform this verification by hand, or at least convince themselves that it should hold by examining the signals involved in the state machine transitions. It should be noted that all transitions from all states except IDLE will happen regardless of input to this entity, given that pulse will always eventually be asserted.

As mentioned previously, this verification relies on the assumption of time being quantized by a global system clock to which all logic is synchronized. One example of this assumption being violated and leading to a bug in the final implementation is in the CS pin on the physical bus interface. This pin can easily by driven by combinatorial logic sensitive to the

state machine and counter leading to a complex combinatorial circuit. When the inputs to this circuit change on a clock edge the logic levels are not instantly propagated to the output; instead logic levels propagate in parallel through many paths in the circuit which causes transient outputs from circuit elements such as LUTs before the inputs to the element settle. These transient levels can lead to 'glitches' on inputs to other logic elements or in the case of the CS pin, the physical device we're interacting with. Glitches on the CS pin of an SPI interface can be seen as the start/end of transactions, which can invalidate all further interaction until the actual start of a new transaction. To remove this bug, the designer must include a register on the output of the CS pin so that no asynchronous logic exists between the design that we can verify, and the external interfaces.

In addition to liveness it is often useful to know the specific timing of a behavioural element. In the case of this SPI PHY we would like to know exactly how many clock cycles each byte takes to send, and how long the cool down between transactions is. As these are both driven by the pulse input we first need to specify its periodic nature:

$$\begin{aligned}&\forall t(\mathrm{pulse}(t) \Rightarrow \mathrm{pulse}(t+D))\\&\forall t(\mathrm{pulse}(t) \Rightarrow \forall t'(t < t' < t+D \Rightarrow \neg\mathrm{pulse}(t')))\end{aligned} \qquad [8.16]$$

This matches the behaviour of a simple counter discussed previously, with a period of D clock cycles. As we don't require a specific period on the physical interface, we can avoid jitter by choosing a power of two period.

From here we begin counting the clock cycles required from a valid transaction start to the first byte assertion, assuming we begin in the IDLE state:

$$\begin{aligned}&\mathrm{IDLE}_t \wedge \mathrm{enable}(t) \wedge \mathrm{pulse}(t) \Rightarrow \mathrm{RUNNING}_{t+1} && [8.7]\\&\mathrm{count}(t+1) = 0 && [8.13]\\&\mathrm{RUNNING}_\tau \wedge \mathrm{pulse}(\tau) \Rightarrow \mathrm{count}(\tau+1) = \mathrm{count}(\tau)+1 && [8.13]\\&\therefore \mathrm{count}(t+15\times D) = 15 && [8.16]\\&\mathrm{RUNNING}_y \wedge \mathrm{count}(y) = 15 \wedge \mathrm{pulse}(y) \Rightarrow \mathrm{ENDING}_{y+1} && [8.8]\\&\therefore \mathrm{ENDING}_{t+15\times D+D+1}\\&\therefore \mathrm{byte}(t+16\times D+1) && [8.14]\end{aligned}$$

Fig. 54 SPI PHY Byte Duration Proof.

This can be verified through behavioural simulation.

Altimeter Driver

For this example we will use the driver for a peripheral pressure sensor, the LPS25HB [74]. This device contains an absolute air pressure gauge, ADC and logic to convert to a pressure value in hPa. It communicates with the master FPGA via an SPI interface with one optional interrupt.

The goal of the driver is to configure the sensor, periodically trigger a pressure reading and gather the resulting measurement. The design of the driver includes a state machine, Fig. 56, and basic block diagram. This driver is also responsible for sanity checking the behaviour of the peripheral. While we have an interrupt available, we will instead use an internal timer to manage timing of samples. The device provides a status register that signals when a measurement interrupt occurs, as well as if an interrupt occurs before the previous sample was retrieved. This allows us to confirm that exactly one measurement has occurred since the last reading. We also manually trigger each reading.

In this example we will use Metric Temporal Logic [6] to formalize the driver behaviour and its specifications. This provides a grammar capable of describing the temporal relationships between events as well as quantifying the time between them in a more concise way than the first order logic in previous examples.

There are two specifications worth noting in this example: the liveliness of the design; that is it continues to drive the device and does not get trapped in any state, and the timing of each reading; which should be performed periodically.

$$\forall x \in S : x \Rightarrow F_{\delta} \neg x \qquad [8.17]$$

This states that for all states in the design, each state will eventually lead to a different state. In other words the design will never stop and be trapped in a single state.

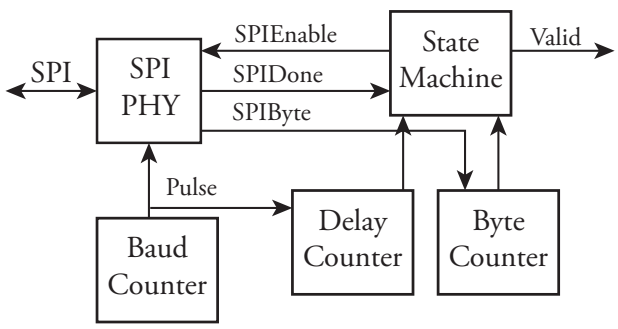

Fig. 55 Altimeter driver control blocks.

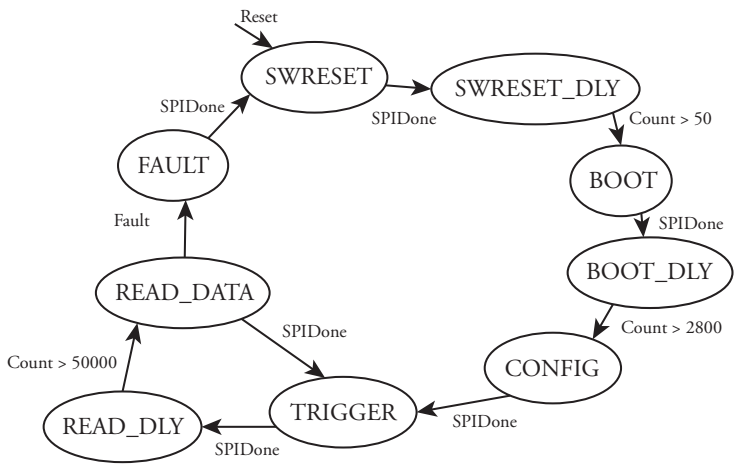

Fig. 56 Altimeter State Machine.

Additionally we can assert that the state machine not only doesn't stop, but that it operates through the desired transitions to operate the device.

$$\Box F_{<\delta} \text{TRIGGER} \wedge \Box F_{<\sigma} \text{READ\_DATA} \qquad [8.18]$$

This states that the state machine shall always enter the TRIGGER state within some δ time, and always enter the READ_DATA state within some σ time. This provides a liveliness specification in that the state machine must continue to transition between triggering and reading the device.

To make the alternating relationship between these two states more explicit, this can instead be written as

$$TRIGGER \Rightarrow F_{<\sigma} READ\_DATA \qquad [8.19]$$

and

$$\text{READ\_DATA} \Rightarrow F_{<\delta} \text{TRIGGER} \qquad [8.20]$$

specifying that READ_DATA will lead to TRIGGER within some δ and similarly that TRIGGER will lead to READ_DATA within σ.

Beginning with the reset condition, given that the Reset signal will always force the state machine into the SWRESET state

$$\text{Reset} \Rightarrow F_C \text{SWRESET} \qquad [8.21]$$

we assume that it is only asserted once when the device is powered on, and we omit the implied $\neg Reset$ from the following expressions.

While it is good to conceptualise this in continuous time, in practice time is usually quantized by a system clock. Therefore all transitions will be made on rising clock edges. So an expression may become true on one rising edge, but will not lead to any transition until the following rising edge (the first rising edge on which the expression was true). Hence the use of the F_C operator, where C is the system clock period.

As this state machine is designed to manipulate the SPI interface to the peripheral sensor, each state is associated with either some transaction to be performed, or simply pause between transactions. The SPI PHY is controlled by the *SPIEnable* signal, which is combinatorially driven based on the state and current *ByteCount,* for example:

$$\mathrm{SWRESET}_t \wedge \mathrm{bytecount}(t) \leq 1 \Leftrightarrow \mathrm{spienable}(t) \qquad [8.22]$$

Where *ByteCount* is a counter that is reset on each state transition, and incremented each time the SPI PHY asserts *SPIByte:*

$$\begin{aligned} &\forall t(x_t \neq x_{t+1} \Rightarrow \mathrm{bytecount}(t+1) = 0) \\ &\forall t(x_t = x_{t+1} \wedge \mathrm{spibyte}(t) \Rightarrow \mathrm{bytecount}(t+1) = \mathrm{bytecount}(t) + 1) \end{aligned} \qquad [8.23]$$

where $x_t \in S$ is the state at time t.

Combined with the timing proof from Fig. 54, we can define the duration of each state based on its expected SPI transactions or delay, in the case of the SWRESET transaction, we write two bytes. We must also include up to one more *pulse* duration, as the state transition does not necessarily align with *pulse*.

$$\mathrm{SWRESET} \Rightarrow F_{<2D+32D+C}\mathrm{spidone} \qquad [8.24]$$

This metric temporal logic specifies that anytime in which the state variable equals SWRESET, the spidone signal will be asserted within 2D+32D clock cycles, where D is the period of the counter defining the baudrate of the SPI PHY. This includes the duration of the number of bytes required for the state transaction plus transaction overhead.

As we can observe from the FSM in Fig. 56

$$\mathrm{SWRESET} \wedge \mathrm{spidone} \Rightarrow F_C\mathrm{SWRESETDLY} \qquad [8.25]$$

which leads us to the simplification:

$$\text{SWRESET} \Rightarrow F_{34D+C}\text{SWRESETDLY} \qquad [8.26]$$

The delay states can be specified in a similar fashion, though they simply use another counter driven by the baudrate counter instead of the SPI PHY to determine when they are exited.

$$\begin{aligned} &\forall t(x_t \neq x_{t+1} \Rightarrow \text{count}(t+1) = 0) \\ &\forall t(x_t = x_{t+1} \wedge \text{pulse}(t) \Rightarrow \text{count}(t+1) = \text{count}(t)+1) \end{aligned} \qquad [8.27]$$

Which simply gives us:

$$\text{SWRESETDLY} \Rightarrow F_{<50D+C}\text{BOOT} \qquad [8.28]$$

The remaining states can be specified the same way, as each transition is either on *SPIDone* or on a *Count* delay as observed in Fig. 56. Given S as the set of all states in the design:

$$S = X \cup Y \qquad [8.29]$$

where X and Y are the subsets of states that have a transition on *Count* or *SPIDone* respectively. Then

$$\begin{aligned} &\forall x \in X: x \wedge \text{count} > n \Rightarrow F_C \neg x \\ &\therefore x \Rightarrow F_{<nD+C} \neg x \end{aligned} \qquad [8.30]$$

and

$$\begin{aligned} &\forall y \in Y: y \wedge \text{spidone} \Rightarrow F_C \neg y \\ &\therefore y \qquad\qquad\qquad \Rightarrow F_{<16Db+2D+C} \neg y \end{aligned} \qquad [8.31]$$

where n is the required delay, and b is the number of bytes in the SPI transaction. Given that *Count* is driven by the periodic *pulse*, and that *SPIDone* is assumed for each relevant state it can be shown that [8.17] holds, proving our liveness requirement.

The last interesting state to consider is READ_DATA as it may encounter a fault condition. While READ_DATA is included in the set of states which exit on SPIDone, it also has a transition based on data from the peripheral device that indicates if a fault has occurred. We provide this simple definition of the fault signal:

$$\forall t(\text{fault}(t) \Rightarrow \text{spibyte}(t) \wedge \text{bytecount}(t) = 1 \wedge \text{fault}'(t)) \qquad [8.32]$$

where $\text{fault}'(t)$ indicates a fault condition exists at time t. This provides us with the timing of the fault indication relative to the SPI transaction that receives it, which is the end of the second byte (but before the SPIDone assertion). This immediately transitions the state machine to the FAULT state in which

$$\text{FAULT}_t \Rightarrow \neg\text{spienable}(t) \qquad [8.33]$$

We now expect the SPIDone to be asserted:

$$\begin{array}{lll}
\text{READDATA} \wedge \text{fault} & \Rightarrow F_C\,\text{FAULT} & \\
\text{fault} & \Rightarrow \text{spibyte} & [8.32] \\
F_C\,\text{FAULT} & \Rightarrow F_C\,\neg\text{spienable} & [8.33] \\
\text{spibyte} \wedge F_C\,\neg\text{spienable} & \Rightarrow F_{<D}\,\text{spidone} & [8.10, 8.15] \\
\therefore F_C\,\text{FAULT} & \Rightarrow F_{<D}\,\text{spidone} &
\end{array}$$

So we can now assume

$$\text{FAULT} \Rightarrow F_{<D+C}\,\text{SWRESET} \qquad [8.34]$$

To prove [8.18]:

$$\begin{array}{ll}
\Box F_{<\delta}\,\text{TRIGGER} \wedge \Box F_{<\sigma}\,\text{READDATA} & \text{RTP} \\
\text{SWRESET} \Rightarrow & \\
F_{<34D+C}\,F_{<50D+C}\,F_{<34D+C}\,F_{<2800D+C}\,F_{<34D+C}\,\text{TRIGGER} & [8.30, 831] \\
\therefore \text{SWRESET} \Rightarrow F_{<2952D+5C}\,\text{TRIGGER} & \\
\text{TRIGGER} \Rightarrow F_{<34D+C}\,F_{<50000D+C}\,\text{READDATA} & \\
\text{READDATA} \Rightarrow F_{<98D+C}\,\text{TRIGGER} \vee F_{<34D+C}\,\text{SWRESET} & \\
\therefore (\text{TRIGGER} \Rightarrow F_{<50034D+2C}\,\text{READDATA}) & \\
\quad \wedge (\text{READDATA} \Rightarrow F_{<2986D+6C}\,\text{TRIGGER}) &
\end{array}$$

This provides us with an upper bound for the delay between these states, which is driven by the fault condition path. The much shorter sampling loop can be shown if we assume $\neg\text{fault}'$.

$$\begin{array}{l}
(\text{TRIGGER} \Rightarrow F_{<50034D+2C}\,\text{READDATA}) \\
\quad \wedge (\text{READDATA} \Rightarrow F_{<34D+C}\,\text{TRIGGER})
\end{array} \qquad [8.35]$$

Finally we can show the periodic nature of the valid output, which signals when the pressure data output from the altimeter is valid to downstream components.

$$
\begin{aligned}
\text{valid}(\,t\,) &\Leftrightarrow \text{READDATA}_t \wedge \text{TRIGGER}_{t+1} \text{ defn.} \\
\therefore \text{valid} &\Rightarrow F_C\,\text{TRIGGER} \\
\text{TRIGGER} &\Rightarrow F_{<50034D+2C}\,\text{READDATA} \qquad [8.35] \\
\text{READDATA} &\Rightarrow F_{<34D+C}\,\text{TRIGGER} \qquad [8.35] \\
\therefore \text{valid} &\Rightarrow F_{<50068D+3C}\,\text{valid}
\end{aligned}
$$

Flight State

The Flight State Machine is responsible for the high level behaviour of the flight controller and defines the current goal that the actuator controllers are trying to achieve. This includes things such as arming before take-off, climbing, levelling the wings and cruising. The Flight State Machine is discussed in detail in Firmware Design and illustrated in Fig. 37 on page 92. In this example we are only concerned by the Flight State's behaviour in enforcing the fail-safe altitude threshold.

The Flight State has been designed to automate the high level behaviour of the aircraft while providing control to the operator. For this reason it includes some features not expected in an FSM such as 'sink' states and arbitrary transitions. This relates to the operators ability to override the state value with any state at any time. The means the SAFE state does not have a natural transition to ARMED other than being forced by the operator. This also means that unnatural transitions exists from any state to any other state. This makes it impossible to verify the overall behaviour of the Flight State Machine with regards to liveness, so we limit our discussion to the relevant behaviour for this example.

The fail-safe altitude threshold is not relevant until the aircraft has been established in flight above the minimum altitude. During ground handling and take-off, the aircraft is obviously below the fail-safe threshold. Once the aircraft has exceeded the mission 'floor' altitude and transitioned from TAKEOFF to CLIMB, the FAILSAFE state is accessible. The behaviour expected is simply that at any time the measured altitude is less than the threshold altitude specified, the Flight State enters FAILSAFE. Once the FAILSAFE state has been entered it cannot be overridden without operator intervention. This state value is then used in downstream control entities to enact the fail-safe behaviour. We assume no overriding control from the operator.

There are three requirements that we wish to verify involving the fail-safe behaviour of the Flight State Machine.

1. After TAKEOFF, if the aircraft descends below a fail-safe altitude threshold, the Flight State Machine shall enter the FAILSAFE state.
2. The FAILSAFE state is accessible from all states that are accessible after TAKEOFF.
3. FAILSAFE cannot be exited once it is entered.

Of course these requirements preclude the ability to autonomously land, requiring operator intervention to descend below the fail-safe threshold. Requirement 2 here may be considered redundant next to requirement 1, but we state it explicitly to emphasize that after TAKEOFF, the state machine should not enter any state that cannot access the FAILSAFE state, as that could potentially make the fail-safe behaviour unavailable at unpredictable times.

As no states after TAKEOFF include transitions to states before, or to TAKEOFF we need only consider the subset of states after TAKEOFF.

$$\begin{gathered}\forall x \in S : x_t \wedge \text{failsafealt}(t) \Rightarrow \text{FAILSAFE}_{t+1} \\ S = [\text{CLIMB}, \text{ENDCLIMB}, \text{CRUISE}, \text{LEVEL}]\end{gathered} \tag{8.36}$$

$$\text{state}(t) \in S \Rightarrow \text{state}(t+1) \in S \cup [\text{FAILSAFE}] \tag{8.37}$$

$$\forall t(\text{FAILSAFE}_t \Rightarrow \text{FAILSAFE}_{t+1}) \tag{8.38}$$

These three axioms specify the required behaviour. With regard to verification, there are no complex timing relationships in this design, and no reliance on subcomponents to perform actions. These axioms can be observed directly in the VHDL implementation.

Servo Actuators

The final stage of the flight controller in our example is the output to the actuators. As illustrated in Fig. 39 on page 94 the values applied to the servo array are selected by the value of Flight State, assuming no operator control. This combinatorial process is a simple MUX that selects either the values from the Controller, or specified fail-safe values. There are no synchronous processes involved here, and so the VHDL implementation translates directly to its logical specification.

```
process(AutoServos, ManualServos, FailSafeServos,
        ManualOverride, FlightState) is
begin
    if ManualOverride = '1' then
        ServoValues <= ManualServos;
    elsif FlightState = FAILSAFE then
        ServoValues <= FailSafeServos;
    else
        ServoValues <= AutoServos;
    end if;
end process;
```

Fig. 57 VHDL Implementation of actuator value MUX.

The servo entities are responsible for generating the PWM signal to the servo actuators. This output is a 0.5ms to 2.5ms wide pulse at 50Hz, with the width of the pulse specifying the absolute position of the actuator. Any pulses with the incorrect width, or any missing pulses will cause the actuator to move towards an incorrect position which could be catastrophic in an aircraft system, and a potential cause of injury in the context of the throttle control. It is important that our actuator controllers meet the following specifications:

4. The actuator controller shall assert a high pulse at 50hz ±1Hz.
5. The width of the pulse shall be at least 0.5ms and at most 2.5ms.
6. Between pulses, the actuator controller shall assert a low output.
7. The width of the pulse shall be proportional to a value supplied to the actuator controller, with an 0.5ms offset.

More formally:

$$\forall t\left(\begin{array}{l}\neg sig(t) \wedge sig(t+1) \Rightarrow \\ \exists \tau\left(\begin{array}{l} t + \frac{1}{50Hz} < \tau < t + \frac{1}{49Hz} \wedge \\ \neg sig(\tau) \wedge sig(\tau + 1)\end{array}\right)\end{array}\right) \quad [8.39]$$

$$\forall t\left(\begin{array}{l}\neg sig(t) \wedge sig(t+1) \Rightarrow \\ \forall \tau\left(t < \tau < t + 0.5ms \wedge sig(\tau)\right)\end{array}\right) \quad [8.40]$$

$$\forall t(\neg sig(t) \wedge sig(t+1) \Rightarrow \neg sig(t + 0.5ms + value(t))) \quad [8.41]$$

$$\forall t\left(\begin{array}{l} sig(t) \wedge \neg sig(t+1) \Rightarrow \\ \forall \tau\left(t < \tau < t + D - v \wedge \neg sig(\tau)\right)\end{array}\right) \quad [8.42]$$

where $0 \leq v \leq 2ms$.

These axioms fully specify the required behaviour for the output of the servo actuator controller. This includes the period of the pulse, the minimum pulse width, the expected behaviour of the value provided to the controller and the time between pulses. Unlike previous examples, we can't assume our *sig* output is only one clock cycle wide and so we instead

specify timing with respect to rising and falling edges. This specification is much more than the safety requirements discussed in previous examples and covers the complete functional behaviour.

We now look at the VHDL implementation of the servo actuator controller

```
architecture Behavioural of Servo is
    signal Count : unsigned(11 downto 0);
begin
    process(Clk) is
    begin
        if rising_edge(Clk) then
            if Reset = '1' then
                Count <= (others => '0');
            elsif Pulse = '1' and Count < 4095 then
                Count <= Count + 1;
            end if;

            Sig <= '1' when Count < (Value + 512) else '0';
        end if;
    end process;
end architecture;
```

Fig. 58 VHDL implementation of Servo Actuator Controller.

Note that the *sig* output is registered to maintain our discrete time assumption. The controller is driven by two external counters, one that drives the *reset* pin at 50Hz and is responsible for the periodic timing, and another that drives the *pulse* pin that provides a timing reference for converting an integer value into a pulse width. Beginning with a 20MHz system clock we make these assumptions about the two counter inputs:

$$\begin{aligned}&\forall t(\mathrm{reset}(t) \Rightarrow \mathrm{reset}(t+399457) \vee \mathrm{reset}(t+399458))\\&\forall t(\mathrm{reset}(t) \Rightarrow \forall \tau(t < \tau < t+399457 \wedge \neg\mathrm{reset}(\tau)))\end{aligned} \qquad [8.43]$$

This provides for a counter with an exponent of 23 and an increment of 21 which results in an output period of 50.068 Hz.

$$\begin{aligned}&\forall t(\mathrm{pulse}(t) \Rightarrow \mathrm{pulse}(t+19) \vee \mathrm{pulse}(t+20))\\&\forall t(\mathrm{pulse}(t) \Rightarrow \forall \tau(t < \tau < t+19 \wedge \neg\mathrm{pulse}(\tau)))\end{aligned} \qquad [8.44]$$

This provides for a counter with an exponent of 14 and an increment of 839 which allows for an 11bit value to span the 2ms window with an accuracy within 0.1%.

Of course we assume

$$\exists t(\mathrm{reset}(t)) \wedge \exists t(\mathrm{pulse}(t)) \qquad [8.45]$$

We now have the requirements for the output *sig* as well as the definitions of the two inputs pulse & reset. There are also timing requirements on the value input such that changing the

value during a pulse may result in erroneous output. We limit changes on the value input to instances of reset:

$$\forall t(\neg reset(t) \Rightarrow value(t) = value(t+1)) \qquad [8.46]$$

we now translate the VHDL implementation into formal logic.

$$\forall t(reset(t) \Rightarrow count(t+1) = 0) \qquad [8.47]$$

$$\forall t\begin{pmatrix} \neg reset(t) \wedge pulse(t) \wedge count(t) < 2^{12} - 1 \Rightarrow \\ count(t+1) = count(t) + 1 \end{pmatrix} \qquad [8.48]$$

$$\forall t\begin{pmatrix} \neg reset(t) \wedge \neg(pulse(t) \wedge count(t) < 2^{12} - 1) \Rightarrow \\ count(t+1) = count(t) \end{pmatrix} \qquad [8.49]$$

$$\forall t(count(t) < value(t) + 513 \Leftrightarrow sig(t+1)) \qquad [8.50]$$

We now wish to show that our conjecture holds over the model of the servo controller. This can be done by translating the formal logic expressions into SMT and handing it to a theorem prover, at which point we discover our first inconsistency. Our definitions for our counter periods in [8.43] and [8.44] only specify the minimum and maximum to account for jitter, and make no assertion about the average period over multiple iterations. This allows for the instance where all periods of *pulse* are the minimum duration and we violate our minimum output duration specified in [8.40]. In this case, where we are really interested in the sum of periods required to increment a counter to reach a specified value we can define, by design, that the total duration required:

$$\begin{aligned} &count(t) = 0 \Rightarrow \\ &\quad count\left(t + floor\left(\frac{2^E x}{I}\right)\right) = x \vee \\ &\quad count\left(t + ceil\left(\frac{2^E x}{I}\right)\right) = x \end{aligned} \qquad [8.51]$$

where E and I are the exponent and increment of the counter driving the pulse signal and x is the number of periods. This provides a more precise specification of the aggregate time interval counter that is consistent with previous specifications. This model proves consistent, verifying our servo actuator controller.

Conclusion

We have discussed in detail a specific behavioural path through the firmware design that collects a relevant data sample, makes decisions based on its value, and then performs appropriate output actions such that the aircraft platform exhibits the required behaviour. This path followed the fail-safe behaviour required by the flight controller in the event that the aircraft descends below a specified threshold altitude.

We formally specified the safety requirements of the initial device driver including its physical serial interface that is required to trigger a measurement in the external peripheral before retrieving the sample and making it available to the rest of the system. This included the case in which a misconfiguration of the peripheral device existed resulting in a detectable, erroneous behaviour and the subsequent correction by the firmware driver.

We then briefly discussed the Flight State Machine in regards to its required behaviour with respect to the fail-safe altitude threshold. The Flight State Machine is responsible for the high level behaviour of the flight controller. We must be sure that it is receptive to the fail-safe condition at all times during operation and that it responds correctly if and when the fail-safe condition occurs.

Finally we discussed the response of the servo actuator controller to the FAILSAFE state from the flight state controller. We verified that the value applied to the actuator was correct and that the PWM signal from the actuator controller was always within specified limits.

The use of first order logic to reason about circuits implemented in VHDL allowed for precise, formal specifications of the components and their requirements. It also allows for the verification of those models against their specifications. VHDL is almost completely written in a syntax similar to first order logic, with the addition of synchronous timing elements. By quantifying statements over a discrete time-step the translation from the VHDL to formal logic is almost trivial. It is the specification of the required behaviour that requires the most thought in this process. It is not trivial to define requirements in such a way as they can be verified while also being a valid model for upstream components, however the technical skills involved are very similar to the skills required to implement a complex design in VHDL in the first place, so it is not unreasonable to expect designers to be capable of verifying their designs as they develop them.

The use of automated theorem provers was useful in the development of formal specifications and models of VHDL implementations. Often, they struggled with quantification over all time for complex models, especially those including non-linear arithmetic such as modulo-2^N counters. Re-phrasing conjecture in a logically equivalent way, or constraining some signals such as resets or the time domain often helped, but limited the results. They did provide confirmation that a model was at least consistent with itself, if not with its specifications. Often, it was simple enough to perform verification proofs by hand. For simple VHDL components it may be possible to implement exhaustive unit tests without need for first order logic. This is typically limited to components without any, or with very simple timing, related logic.

Conclusion

In this thesis I aimed to demonstrate the design and implementation of a novel flight controller for small unmanned aerial systems. The anticipated use of this controller was for High Altitude, Long Endurance solar powered aircraft, so there was a focus on reliability, size and power consumption. This design differed from traditional solutions by implementing all functionality within an FPGA rather than software on a CPU. This allows for formal verification of the implementation of the flight controller as well as radiation hardening using COTS parts. We demonstrated techniques for verifying various components within the design using first-order and metric-temporal logic. We included mitigation of Single Event Effects using our ECC library to implement Hamming codes around all registers within the implementation, greatly reducing the resources required when compared with more common Triple Modular Redundancy (TMR). However the effectiveness of our ECC hardening suffers due to excessive propagation delays when compared with TMR. We successfully built and tested the proposed flight controller in a real aircraft.

A hardware device was designed that included all the necessary components to autonomously control a small aircraft, including an SoC with both a CPU & FPGA, compass, altimeter, IMU and FLASH & RAM memories. Considerations were made to include external components such as GPS receivers, airspeed indicators and RF transceivers. Components were split between two PCBs to allow flexibility in its use. An SoM including the main processor, memories and IMU components which is intended to be a functional flight controller on its own; and a breakout board which included extra peripherals for programming, storage and communication. The completed device maintained a very small form factor to fit within the constrained fuselage of the Pulsar 2.5E airframe that had been selected for further experimentation. The complete SoM and breakout board assembly is 67 x 20 x 14mm, weighs 19g and consumes <200mW on average in a typical configuration.

The use of an FPGA as the main processing element required new firmware to be implemented from scratch, as existing solutions all target CPUs. Various control, navigation and communication algorithms had to be (re)designed to suit implementations on programmable logic. I detailed the design of this firmware which included device drivers for all the included peripheral components on the hardware, the control systems that implemented the required

behaviour of the autonomous aerial system and the communications with a remote ground control station. Considerations were made to harden memory elements within the firmware against the effects of radiation, as well as formal verification of firmware components to protect against implementation faults.

One advantage of implementing a flight controller with an FPGA instead of a CPU is that formal methods are significantly simpler to apply to verify that the firmware components meet their specifications. In the examples discussed in this thesis, translating the implementation to first order logic was a trivial exercise, as the combinatorial logic statements in the VHDL are already formal logic statements. The main challenge with applying formal methods was the consideration of timing within the implementation. This is simplified by assuming a design that is completely synchronous with a single system clock. This allows us to use a discrete time interval common to all elements over which we can quantify our conjecture. Formally specifying the behaviour of a firmware entity proved a valuable exercise, not just for the resulting verification of the entity, but also in redesigning and refining the entity itself. During this project, VHDL implementations were produced which were then subsequently subjected to verification. As mentioned, the translation of the entity into first order logic was usually a trivial exercise. When it was not trivial, however, this was usually an indication that the implementation in question was not appropriate; that is, a more elegant or simpler solution for the same behaviour could be achieved. With experience, this lead to the use of standard templates for various behavioural elements including state machines and entity interfaces. Use of these templates led to simpler designs, easier verification and much shorter development time.

As an aircraft gains altitude it is exposed to increasing levels of radiation from high energy particles interacting with the atmosphere [16]. Neutron flux in the range of 1 - 10 MeV is commonly experienced within Earth's atmosphere and has the energy required to effect electronic devices. These effects are grouped as SEEs and can cause transient logic levels within silicon devices and potentially alter the value of memory elements such as flip-flops. SEEs can also affect the memory used to configure an FPGA, resulting in the behaviour of the FPGA itself being modified. This is an issue for SRAM based FPGAs that we thought was best avoided, as we didn't require the flexibility or performance currently offered. We instead used a FLASH based FPGA as the configuration memory is stored in persistent memory and immune to SEUs. This limited the requirement to harden the FPGA to focus only on the

data elements in the design, and not the FPGA configuration itself. Typical approaches to SEE mitigation involve redundancy by replicating hardware. In small aircraft applications, it is not always practical to include multiple copies of the same flight controller, due to space and power constraints. We approach this from the firmware implementation by encoding all registers with some form of redundancy, usually a type of hamming code. We discuss the balance between the consumption of logic resources and elongated timing paths which occurs with the introduction of hamming codes to our implementation. We created a VHDL library for the purpose of encoding and decoding various bit vectors with error correcting codes so that they can easily be used throughout the firmware system. In our design, the number of logic slices required increased by 1.42x when adding radiation hardening using ECC, compared with a typical increase of 3.2x when using TMR. Our ECC implementation also allows the average length of a codeword to be "tuned" to allow the balance between LUTs and FFs required in the design to be shifted as needed.

We estimate, based on neutron testing performed on the M2S025 that a fully utilized FPGA with no protection for SEEs can experience 4.76E-5 upsets per hour at an altitude of 40,000 ft and 45° latitude. When using ECC and other techniques to protect all elements in the FPGA fabric from SEEs, this expected upset rate drops to 2.54E-22 in ideal RTL simulation. However, given the asynchronous nature of SEEs they are able to upset memory elements with no regard for our system clock. When implementing hardening algorithms in the FPGA fabric, these circuits themselves have a non-zero propagation delay meaning that erroneous, intermittent glitches from unstable logic could be latched into a register, effectively subverting the protection offered. This is limited to a fraction of the clock period depending on the maximum frequency the design is capable of running and the actual clock frequency. In our design, this gave us an upper bound that 1 in 10 SEUs may escape our protection circuits. This effect is exacerbated by using Hamming codes which have significantly more complex protection circuits than TMR, however this should be considered whenever there is significant propagation delay with respect to the clock period.

We successfully manufactured the hardware device as designed and implemented the basic flight controller firmware. Despite a few manufacturing faults, the hardware functioned as designed, though the power consumption was about 40% higher than estimated. The peripheral devices functioned as advertised and the device was able to track its attitude, navigate between waypoints and control the aircraft semi-autonomously. We were also able to

communicate with the aircraft while airborne with a similar hardware device on the ground, including telemetry, command & control as well as larger data payloads. We were able to verify a selection of firmware components and describe the techniques required to perform a complete verification of the system.

Flight Tests

While actual flight performance was not a topic in this thesis, experimental flights were conducted for short periods and at a low altitude to sanity check the design and attempt to tune various control parameters. The aircraft used was a high performance glider, the Pulsar 4E and later the Pulsar 2.5E. During these tests the pilot was always able to manually control the aircraft via a DSM transmitter as if it were a standard radio controlled aircraft. Waypoint data forming a simple path was provided to the aircraft over its wireless link through a graphical interface on a laptop, which was relayed through the GCS. Telemetry data was reliably received at 10Hz and provided attitude and performance information for later analysis. When the aircraft was left in an autonomous mode, it attempted to navigate through the path it was provided. This was largely unsuccessful due to the lack of wind compensation and inclusion of magnetic variance in early implementations. This caused the aircraft to wander in circles in the vicinity of its target waypoint, but never follow the path precisely. In contrast

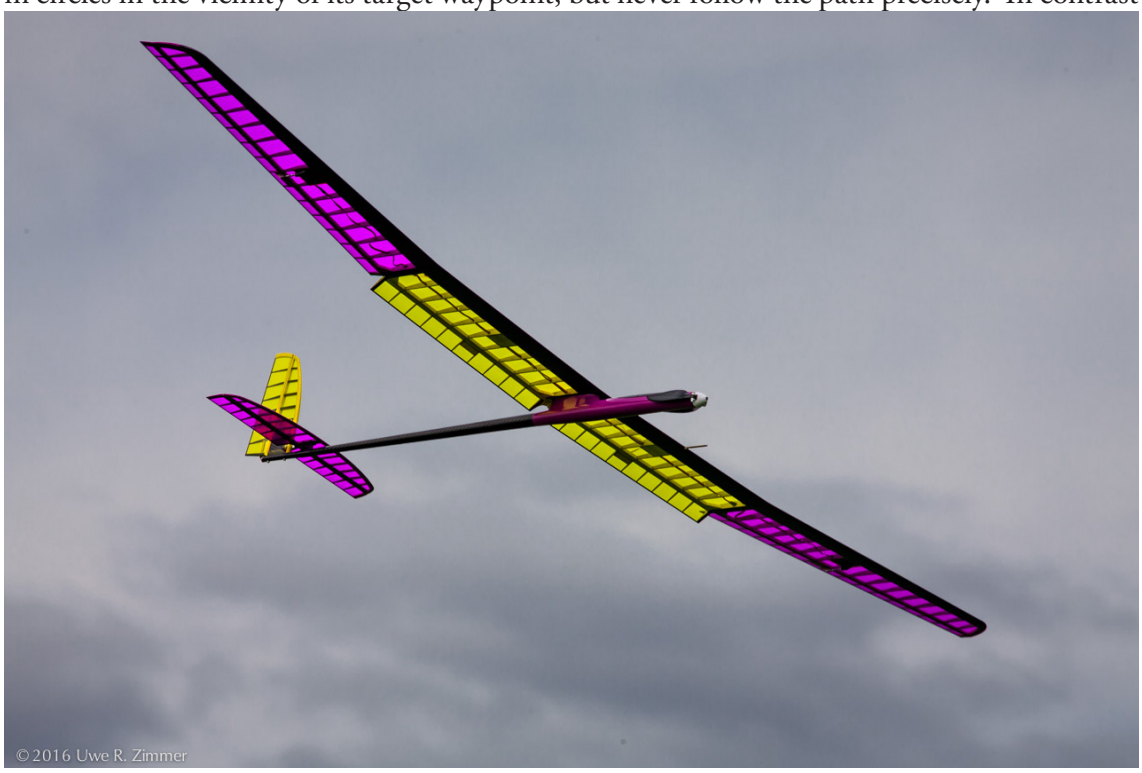

Fig. 59 The Pulsar 2.5E in flight.

to its navigation abilities, the flight controller was very capable with vertical navigation and control of its airspeed and attitude. It successfully reacted to altitude thresholds and would autonomously climb and sink between them as designed. The longest autonomous period of flight was in excess of 30 minutes during which the aircraft climbed and sank repeatedly while wandering around the area of operation.

There were two accidents during flight testing which resulted in damage to the aircraft. The first was due to operator error during a "hand launch" in which the pilot maintains manual control of the aircraft while throwing it into the air. The switch to toggle between autonomous and manual control was inadvertently switched during the launch, which resulted in no control of the aircraft, which immediately dove, impacting the runway. The second accident occurred during autonomous flight testing. The aircraft entered a shallow dive below its floor altitude. Inappropriate PID controller coefficients prevented the aircraft from recovering before reaching its failsafe altitude. Upon reaching its failsafe altitude, all control surfaces were driven to failsafe positions at a significant airspeed. This was a significant control surface deflection above the aircraft's apparent maximum manoeuvring speed which resulted in the failure of the main wing spar. Both of these accidents occurred during early testing, and are the result of insufficient design of the flight controller rather than failures of the implementation.

Future Work

I mentioned in the introduction of this thesis that standards exists to provide guidelines regarding the development of high integrity software & hardware systems for use in aviation applications. DO-178C [2] and DO-254 [3] are two such standards. However, much of the certification requirements of both of these standards relates to the processes used in developing a system as well as the management of the project teams tasked with performing the work. System development projects in academic settings do not usually fulfil the requirements for certification against such standards, and I will admit this one certainly did not. However effort was made to demonstrate that the techniques and implementation style used in this project could easily be used to ensure the integrity of the output products, and to support certification against such standards.

What we did not address in this thesis was the verification of vendor tools used to compile, synthesize and place & route a design on an FPGA fabric as mentioned in [22]. One of the

motivations for using dissimilar implementations of what would otherwise be logically equivalent components, as in [1], is to isolate faults introduced in the assembly process even when the development team's implementation is otherwise flawless. This is more of a challenge with FPGAs as the current state of the industry still couples vendor tools with their own hardware, and we are not aware of any third party tools for developing on an FPGA.

We also discussed various concerns about SEEs from a hardware perspective, including the use of FLASH based FPGAs instead of SRAM; and the effects on switch mode power supplies, oscillators and peripheral components. Where we could, we outline strategies to mitigate radiation effects on these devices. Much of the discussion on radiation hardening in this thesis was theoretical. While we were able to use real test results (provided by a manufacturer) for the neutron upset cross sections, it is difficult to verify our mitigation strategies in a real world radiation environment. The effects of SEEs on specific elements within an FPGA are well understood, and the mitigation strategies are theoretically sound, however there are many aspects of the complete system that may be susceptible to SEEs. It would be of great benefit to perform neutron testing on the completed devices to quantify the tolerance of radiation and discover any other potential upset vectors.

Our ECC hardening circuits were not as effective at mitigating SEUs as we had hoped. This was due to significant propagation delays in the decoding circuits themselves, allowing poorly timed SEEs to subvert the protection entirely. TMR has an advantage over ECC in this regard as the voting circuits are quite simple and each bit can usually be implemented within a single logic slice, allowing apparent skew between codeword bits to be masked. It was difficult to ascertain from the literature if this is intentional. I believe further examination is required to quantify this timing effect on TMR circuits, especially when synthesized in combination with other, more complex downstream combinatorial circuits.

The hardware device in this thesis went through three iterations during this project. The main difference between each version was the FPGA used, as the firmware grew it exceeded the bounds of previous hardware platforms. Simultaneously, FPGA technology improved moving from 3 input LUTs to 4 input LUTS, and the inclusion of MATH blocks on the FPGA fabric. We began with the ProASIC3 from Actel before moving on to the Igloo family from what became Microsemi. Finally we arrived at the SmartFusion2 design which includes an Igloo2 FPGA from Microsemi, who has recently become Microchip. This third version is what was presented in this thesis and is the product of many experiences and insights from

previous mistakes. That being said the first two version both successfully controlled aircraft at various levels of autonomy in flight. During this time the availability and capability of sensors and peripherals available for this type of application improved significantly, and new versions included updated components.

There are many improvements that can be made to the flight controller hardware and firmware that resulted from this work. Many of the algorithms in the firmware we simplified to speed development time and provide only a demonstration of capability. Obviously the integrity of a flight control involves more than just computational rigour, there must also be robust algorithms and defensive design decisions. Having demonstrated a high integrity computational platform in this thesis, the remaining work is to build a robust flight controller on top of it.

Finally, a significant amount of work during this PhD was spent investigating, designing and building technologies in support of High Altitude, Long Endurance, solar powered aircraft. This includes maximum power point tracking, DC converters to combine the optimal output of several, heterogeneous photo-voltaic arrays; a small, efficient battery charger with the ability to automatically and continuously balance the individual cells of lithium polymer battery packs; as well as behavioural techniques for a solar powered aircraft to maximise isolation of its solar cells while attempting to optimise flight attitude. This proved to be a

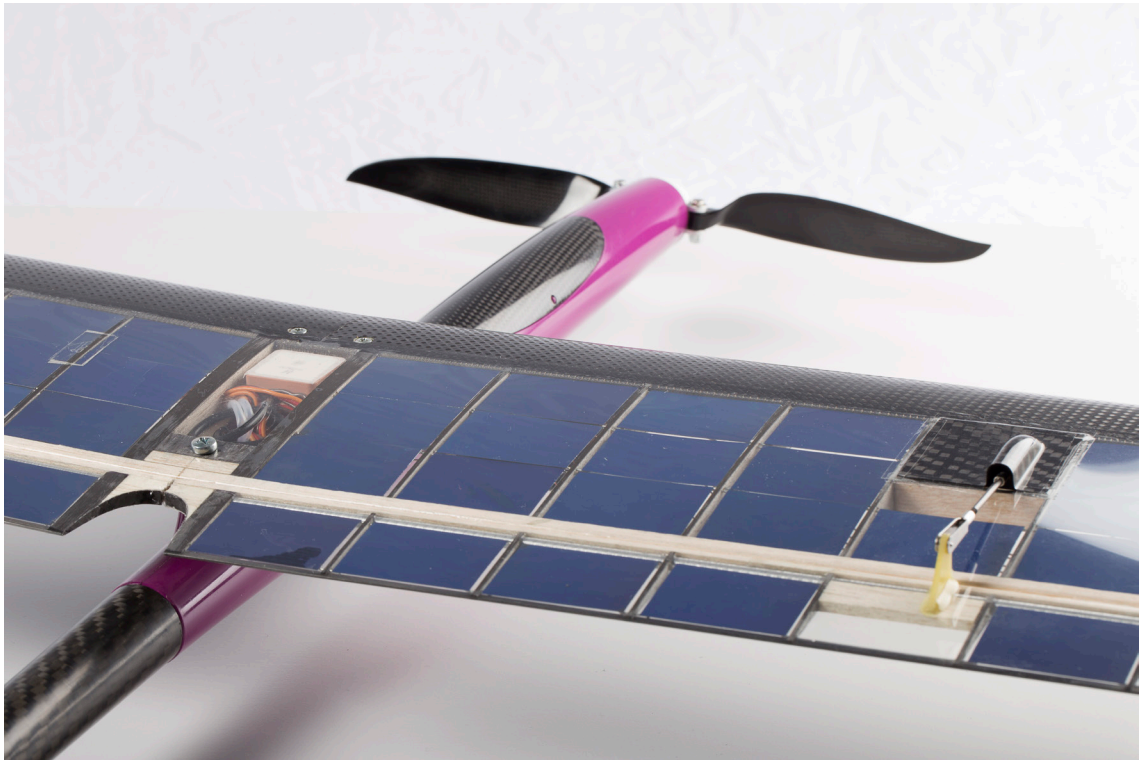

Fig. 60 The Pulsar 2.5E's photo-voltaic array.

significantly large, multi-disciplinary area which could not be covered in a single thesis in any great detail, so these topics were omitted for brevity.

References

[1] Yeh, Y. C., "Triple-Triple Redundant 777 Primary Flight Computer," presented at the IEEE Aerospace Applications Conference, 1996.

[2] RTCA, Software Considerations in Airborne Systems and Equipment Certification. 2011.

[3] RTCA, Design Assurance Guidance for Airborne Electronic Hardware. 2000.

[4] Bornebusch, F., Lüth, C., Wille, R., Drechsler, R., "Towards Automatic Hardware Synthesis from Formal Specification to Implementation" 25th Asia and South Pacific Design Automation Conference, pp 375-380. 2020.

[5] Claessen, K., Hahnle, R., and Martensson, J., "Verification of hardware systems with first-order logic," presented at the Problems and Problem Sets Workshop, 2002.

[6] Koymans, R., "Specifying real-time properties with metric temporal logic," Real-Time Systems, vol. 2, no. 4, pp. 255–299, Nov. 1990.

[7] Baumgartner, P., Bax, J., and Waldmann, U., "Beagle – A Hierarchic Superposition Theorem Prover," presented at the Automated Deduction - CADE, 2015

[8] De Moura, L., and Bjørner, N., "Z3: An Efficient SMT Solver," Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340, 2008.

[9] Barrett, C. and Tinelli, C., "CVC3," Computer Aided Verification, pp. 298–302, 2007.

[10] Riener, H., Haedicke, F., Frehse, S., Soeken, M., Große, D., Drechsler, R., Fey, G. "metaSMT: focus on your application and not on solver integration," International Journal on Software Tools for Technology Transfer 19(5), pp 605-621. 2017

[11] Taber, A. and Normand, E., "Single event upset in avionics," IEEE Transactions on Nuclear Science, vol. 40, Apr. 1993.

[12] Taber, A. and Normand, E., "Investigation and Characterization of SEU Effects and Hardening Strategies in Avionics," DNA-TR-94-123, Feb. 1995.

[13] Baumann, R. C., "Radiation-induced soft errors in advanced semiconductor technologies," IEEE Transactions on Device and Materials Reliability, vol. 5, pp. 305–316, Sep. 2005.

[14] Shoga, M. and Binder, D., "Theory of Single Event Latchup in Complementary Metal-Oxide Semiconductor ICs," IEEE Transactions on Nuclear Science, vol. 33, no. 6, 1986.

[15] Sterpone, L., Violante, M., "Analysis of the Robustness of the TMR Architecture in Sram-Based FPGAS," IEEE Transactions on Nuclear Science 52(5), pp 1545-1549. 2005.

[16] Normand, E. and Baker, T. J., "Altitude and latitude variations in avionics SEU and atmospheric neutron flux," IEEE Transactions on Nuclear Science, pp. 1484–1490, Dec. 1993.

[17] Pellegrini, P., Euler, F., Kahan, A., Flanagan, T. M. , and Wrobel, T. F., "Steady-state and transient radiation effects in precision quartz oscillators," IEEE Transactions on Nuclear Science, vol. 25, no. 6, pp. 1267–1273, Dec. 1978.

[18] Stroud, C., Barbour, A., "Design for testability and test generation for static redundancy system level fault-tolerant circuits," proceedings 'Meeting the Tests of Time', International Test Conference, 1989.

[19] Kastensmidt, F., Sterpone, L., Carro, L., Reorda, M. "On the Optimal Design of Triple Modular Redundancy Logic for SRAM-based FPGAs Design," Automation and Test in Europe, 2005.

[20] Xilinx. "RT Kintex UltraScale FPGAs for Ultra High Throughput and High Bandwidth Applications," White paper, May 2020.

[21] Xilinx. "Radiation-Hardened, Space-Grade Virtex-5QV Family Data Sheet: Overview" Datasheet, Jan. 2018.

[22] Bernardeschi, C., Cassano, L., and Domenici, A., "SRAM-Based FPGA Systems for Safety-Critical Applications," Journal of Computer Science and Technology, vol. 30, no. 2, pp. 373–390, Mar. 2015.

[23] Maillard, P., "Neutron, 64 MeV proton & alpha single-event characterization of Xilinx 16nm FinFET Zynq® UltraScale+TM MPSoC," presented at the IEEE Radiation Effects Data Workshop, 2017.

[24] Rockett, L., Patel, D., Danziger, S., Cronquist, B., Wang, J. "Radiation Hardened FPGA Technology for Space Applications," IEEE Aerospace Conference. 2007

[25] Bolchini, C., Quarta, D., Santambrogio, M. "SEU mitigation for SRAM-based FPGAS through dynamic partial reconfiguration," Jan. 2007.

[26] Bolchini, C., Miele, A., Santambrogio, M. "TMR and Partial Dynamic Reconfiguration to mitigate SEU faults in FPGAs," 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems, 2007.

[27] Muñoz-Quijada, M., Sanchez-Barea, S., Vela-Calderon, D., Guzman-Miranda, H. "Fine-Grain Circuit Hardening Through VHDL Datatype Substitution," Electronics 8(1), pp. 24, Dec. 2018.

[28] Huhn, S., Frehse, S., Wille, R., Drechsler, R. "Enhancing robustness of sequential circuits using application-specific knowledge and formal methods," 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), 2017.

[29] Ebeid, E., Skriver, M., and Jin, J., "A Survey on Open-Source Flight Control Platforms of Unmanned Aerial Vehicle," presented at the Euromicro Conference on Digital System Design (DSD), 2017, pp. 396–402.

[30] Hamming, R. W., "Error detecting and error correcting codes," The Bell System Technical Journal, vol. XXIX, no. 2, pp. 147–160.

[31] Sanchez-Macian, A., Reviriego, P., Maestro, J. "Hamming SEC-DAED and Extended Hamming SEC-DED-TAED Codes Through Selective Shortening and Bit Placement," IEEE Transactions on Device and Materials Reliability 14(1), pp 574-576. 2014.

[32] Vargas, F., Amory, A., Velazco, R. "Estimating circuit fault-tolerance by means of transient-fault injection in VHDL" in proceedings 6th IEEE International On-Line Testing Workshop, 2000.

[33] Azambuja, J., Nazar, G., Rech, P., Carro, L., Kastensmidt, F., Fairbanks, T., Quinn, H. "Evaluating Neutron Induced SEE in SRAM-Based FPGA Protected by Hardware and Software-Based Fault Tolerant Techniques," IEEE Transactions on Nuclear Science 60(6), pp 4243-4250. 2013.

[34] Zimmermann, H., "OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection," IEEE Transactions on Communications, vol. 28, no. 4, pp. 1–8, Apr. 1980.

[35] Tsiligiannis, G., Dilillo, L., Bosio, A., Girard, P., Todri, A., Virazel, A., McClure, S., Touboul, A., Wrobel, F., Saigné, F. "Testing a Commercial MRAM Under Neutron and Alpha Radiation in Dynamic Mode," IEEE Transactions on Nuclear Science 60(4), pp 2617-2622. 2013.

[36] "Pixhawk," pixhawk.org, viewed 23 Aug 2018.

[37] Meier, L., Tanskanen, P., Heng, L., Lee, G. H., Fraundorfer, F., and Pollefeys, M., "PIXHAWK: A micro aerial vehicle design for autonomous flight using onboard computer vision," Autonomous Robots, vol. 33, no. 1, pp. 21–39, Aug. 2012.

[38] "Ardupilot," ardupilot.org, viewed 23 Aug 2018.

[39] "OcPoC," aerotenna.com/ocpoc-zynq, viewed 23 Aug 2018.

[40] "Phenix Pro" en.robsense.com, viewed 7 Sep 2018.

[41] Alvis, W., Murthy, S., Valavanis, K., Moreno, W., Fields, M., and Katkoori, S., "FPGA Based Flexible Autopilot Platform for Unmanned Systems," presented at the 15th Mediterranean Conference of Control Automation, 2007.

[42] Christophersen, H. B., Pickell, W. J., Koller, A. A., Kannan, S. K., and Johnson, E. N., "Small Adaptive Flight Control Systems for UAVs Using FPGA/DSP Technology," presented at the AIAA 3rd Unmanned Unlimited Technical Conference, 2004.

[43] Ratti, J., Moon, J.-H., and Vachtsevanos, G., "Towards Low-Power, Low-Profile Avionics Architecture and Control for Micro Aerial Vehicles," IEEE Aerospace Conference, 2011.

[44] Gao, X.-Z., Hou, Z.-X., Guo, Z., Liu, J.-X., and Chen, X.-Q., "Energy management strategy for solar-powered high-altitude long-endurance aircraft," Energy Conversion and Management, vol. 70, no. C, pp. 20–30, Jun. 2013.

[45] Rapinett, A., "Zephyr: A High Altitude Long Endurance Unmanned Air Vehicle" Apr. 2009. Masters Thesis, University of Surrey.

[46] Microsemi Corporation, "UG0446 User Guide SmartFusion2 and IGLOO2 FPGA High Speed DDR Interfaces," Feb. 2017.

[47] Wang, F. and Agrawal, V. D., "Single Event Upset: An Embedded Tutorial," presented at the 21st International Conference on VLSI Design 2007, pp. 429–434.

[48] Dodd, P. E., Sexton, F. W., Hash, G. L., Shaneyfelt, M. R., Draper, B. L., Farino, A. J. and Flores, R. S., "Impact of technology trends on SEU in CMOS SRAMs," IEEE Transactions on Nuclear Science, vol. 43, no. 6, pp. 2797–2804, 1996.

[49] Pickel, J. C., Blandford, J. T., "CMOS RAM Cosmic-ray-induced-error-Rate Analysis," IEEE Transactions on Nuclear Science, vol. 28, no. 6, Dec. 1981.

[50] Rezgui, S., Wang, J. J., Sun, Y., Cronquist, B., and McCollum, J., "Configuration and Routing Effects on the SET Propagation in Flash-Based FPGAs," IEEE Transactions on Nuclear Science, vol. 55, no. 6, pp. 3328–3335, Dec. 2008.

[51] Microsemi Corporation, "UG0445 User Guide SmartFusion2 SoC FPGA and IGLOO2 FPGA Fabric," datasheet, pp. 1–124, Sep. 2017.

[52] Microsemi Corporation, "TR0020: SmartFusion2 and IGLOO2 Neutron Single Event Effects (SEE)," white paper, Aug. 2015.

[53] Rezzak, N., Wang, J. J., DSilva, D., Huang, C. K., and Varela, S., "Single Event Effects Characterization in 65 nm Flash- Based FPGA-SOC," presented at the SEE symposium, 2014.

[54] Quinn, H., Fairbanks, T., Tripp, J., Duran, G., Lopez, B. "Single-Event Effects in Low-Cost, Low-Power Microprocessors," in proceedings IEEE Radiation Effects Data Workshop (REDW), 2014.

[55] Fairbanks, T., Quinn, H., Tripp, J., Michel, J., Warniment, A., Dallmann, N. "Compendium of TID, Neutron, Proton and Heavy Ion Testing of Satellite Electronics for Los Alamos National Laboratory," in proceedings IEEE Radiation Effects Data Workshop (REDW), 2013.

[56] Bloch, M., Mancini, O., and McClelland, T., "Effects of Radiation on Performance of Space-Borne Quartz Crystal Oscillators," presented at the IEEE International Frequency Control Symposium Joint with the 22nd European Frequency and Time forum, 2009.

[57] Tararaksin, A. S., Kessarinskiy, L. N., Pechenkin, A. A., Demidova, A. V., Yanenko, A. V., Boychenko, D. V., and Nikiforov, A. Y., "Experimental Investigation of SELs in SiT8003 MEMS-Oscillators," presented at the IEEE Radiation Effects Data Workshop, 2015.

[58] Santos, M., Ribeiro, H., Martins, M., and Guilherme, J., "Switch Mode Power Supply Design Constraints for Space Applications," presented at the Telecommunications - ConfTele, 2007.

[59] Zeinolabedinzadeh, S., Ying, H., Fleetwood, Z. E., Roche, N. J. H., Khachatrian, A., McMorrow, D., Buchner, S. P., Warner, J. H., Paki-Amouzou, P., and Cressler, J. D., "Single-Event Effects in High-Frequency Linear Amplifiers: Experiment and Analysis," IEEE Transactions on Nuclear Science, vol. 64, no. 1, pp. 125–132, Jan. 2017.

[60] Microsemi Corporation, "DS0128 Datasheet IGLOO2 FPGA and SmartFusion2 SoC FPGA," datasheet, 2016.

[61] Penzin, S. H., Crain, W. R., Crawford, K. B., Hansel, S. J., Kirshman, J. F., and Koga, R., "Single Event Effects in Pulse Width Modulation Controllers," IEEE Transactions on Nuclear Science, vol. 43, no. 6, Dec. 1996.

[62] Bozzano, M., Bruttomesso, R., Cimatti, A., Franzén, A., Hanna, Z., Khasidashvili, Z., Palti, A., and Sebastiani, R., "Encoding RTL Constructs for MathSAT: a Preliminary Report," Electronic Notes in Theoretical Computer Science, vol. 144, no. 2, pp. 3–14, Jan. 2006.

[63] Aguirre, M. A., Tombs, J. N., et al, "FT-UNSHADES: A new system for SEU injection, analysis and diagnostics over post synthesis netlist," presented at the Military and Aerospace Programmable Logic Devices, 2005.

[64] Hilton, A. and Hall, J. G., "Refining Specifications to Programmable Logic," Electronic Notes in Theoretical Computer Science, vol. 70, no. 3, pp. 37–49, 2002.

[65] Magnus, P. D., "Forall X: An Introduction to Formal Logic," 2017.

[66] Delgrande, J. P., "An Approach to Default Reasoning Based on a First-Order Conditional Logic: Revised Report*," Artificial Intelligence, no. 36, pp. 69–90, 1988.

[67] Actel Corporation, "ProASIC3 Flash Family FPGAs Datasheet", datasheet, 2010.

[68] Microsemi, "IGLOO nano Low Power Flash FPGAs Datasheet," datasheet, Mar. 2012.

[69] Placinta, V. M., Cojocariu, L. N., and Ravariu, C., "Evaluating the Switching Mode Power Supplies used in Radiation Hardness Tests of Integrated Circuits," presented at the International Semiconductor Conference, 2017.

[70] Semtech, "SC202A 3.5MHz, 500mA Step-down Regulator," datasheet, 2011.

[71] Cypress, "S25FS512S, 512 Mbit, 1.8 V Serial Peripheral Interface with Multi-I/O Flash," datasheet, pp. 1–136, Apr. 2018.

[72] ST, "iNEMO inertial module: always-on 3D accelerometer and 3D gyroscope," datasheet, Sep. 2017.

[73] ST, "Digital output magnetic sensor: ultra-low-power, high-performance 3-axis magnetometer," datasheet, pp. 1–33, May 2017.

[74] ST, "MEMS pressure sensor: 260-1260 hPa absolute digital output barometer," datasheet, pp. 1–50, Aug. 2016.

[75] Euston, M., Coote, P., Mahony, R., Kim, J., and Hamel, T., "A Complementary Filter for Attitude Estimation of a Fixed-Wing UAV," presented at the International Conference on Intelligent Robots and Systems, 2008.

[76] Grewal, M., Henderson, V., Miyasako, R. "Application of Kalman filtering to the calibration and alignment of inertial navigation systems," 29th IEEE Conference on Decision and Control, 1990

[77] Sabatini, A. M., "Quaternion-Based Extended Kalman Filter for Determining Orientation by Inertial and Magnetic Sensing," IEEE Trans. Biomed. Eng., vol. 53, no. 7, pp. 1346–1356, Jul. 2006.

[78] Booth, A. D., "A Signed Binary Multiplication Technique," The Quarterly Journal of Mechanics and Applied Mathematics, vol. IV, no. 2, pp. 236–240, Aug. 1950.

[79] Robusto, C. C., "The Cosine-Haversine Formula," The American Mathematical Monthly, vol. 64, no. 1, pp. 38–40, Jan. 1957.

[80] Volder, J. E., "The CORDIC Trigonometric Computing Technique," IRE Transcations on Electronic Computers, vol. 8, no. 3, pp. 330–334, Sep. 1959.

[81] Andraka, R., "A survey of CORDIC algorithms for FPGA based computers," presented at the ACMA/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, 1998, pp. 191–200.

[82] Atmel, “AT86RF212B Low Power, 700/800/900MHz Transceiver for ZigBee, IEEE 802.15.4, 6LoWPAN, and ISM Applications AT86RF212B,” pp. 1–212, Feb. 2015.

[83] Texas Instruments, “CC1190 850 – 950 MHz RF Front End,” pp. 1–16, Nov. 2010.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.Numeric_std.all;

entity QuorumCounter is
    generic (
        I : integer := 1;
        N : integer := 8
    );
    port (
        Clk : in std_logic;
        Reset : in std_logic;
        Enable : in std_logic;
        Strobe : out std_logic;
        Quorum : in std_logic;
        CarryIn : in unsigned(N downto 0);
        CarryOut : out unsigned(N downto 0)
    );
end entity;

architecture Behavioural of QuorumCounter is
    signal Count, NextCount : unsigned(N downto 0);
    signal Fault : std_logic;
begin
    process(Clk) is
    begin
        if rising_edge(Clk) then
            if Reset = '1' then
                Count <= (others => '0');
            else
                Count <= NextCount;
            end if;
        end if;
    end process;

    process(Fault, Enable, Count, CarryIn) is
    begin
        if Fault /= '0' then
            NextCount <= CarryIn;
        elsif Enable = '1' then
            NextCount <= ('0' & Count(N-1 downto 0)) + I;
        else
            NextCount <= '0' & Count(N-1 downto 0);
        end if;
    end process;

    Fault <= Quorum /= Count(N);
    CarryOut <= NextCount;
    Strobe <= Count(N);
end architecture;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.Numeric_std.all;

entity TripleRedundantCounter is
    generic (
        I : integer := 1;
        N : integer := 8
    );
    port (
        Clk : in std_logic;
        Reset : in std_logic;
        Enable : in std_logic;
        Strobe : out std_logic
    );
end entity;

architecture Behavioural of TripleRedundantCounter is
    signal Quorum : std_logic;
    signal StrobeA, StrobeB, StrobeC : std_logic;
    signal CarryInA, CarryInB, CarryInC : unsigned(N downto 0);
begin
    Quorum <= (StrobeA and StrobeB) or
              (StrobeB and StrobeC) or
              (StrobeA and StrobeC);
    Strobe <= Quorum;

    COUNTERA : entity WORK.QuorumCounter
        generic map (I, N)
        port map (
            Clk => Clk,
            Reset => Reset,
            Enable => Enable,
            Strobe => StrobeA,
            Quorum => Quorum,
            CarryIn => CarryInA,
            CarryOut => CarryInB
        );

    COUNTERB : entity WORK.QuorumCounter
        generic map (I, N)
        port map (
            Clk => Clk,
            Reset => Reset,
            Enable => Enable,
            Strobe => StrobeB,
            Quorum => Quorum,
            CarryIn => CarryInB,
            CarryOut => CarryInC
        );

    COUNTERC : entity WORK.QuorumCounter
        generic map (I, N)
        port map (
            Clk => Clk,
            Reset => ‘0’,
            Enable => Enable,
            Strobe => StrobeC,
            Quorum => Quorum,
            CarryIn => CarryInC,
            CarryOut => CarryInA
        );
end architecture;
```

Appendix B. ECC Library Implementation

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

package ECC is
    type ecc_vector is array (natural range <>) of std_logic;
    attribute keep : string;
    attribute keep of ecc_vector : type is “true”;

    function to_ecc_vector(s : std_logic_vector) return ecc_vector;
    function to_ecc_vector(s : unsigned) return ecc_vector;
    function to_ecc_vector(s : signed) return ecc_vector;


    function to_slv(e : ecc_vector) return std_logic_vector;
    function to_unsigned(e : ecc_vector) return unsigned;
    function to_signed(e : ecc_vector) return signed;

    function code_length(l : positive) return positive;
end package;

package body ECC is
    function xor_reduce(s : std_logic_vector) return std_logic is
        variable x : std_logic := ‘0’;
    begin
        for i in s’range loop
            x := x xor s(i);
        end loop;

        return x;
    end function;


    --Given the length of a Data word, how many ECC bits are needed
    function num_ecc_bits(l : positive) return positive is
        variable i : integer := 0;
    begin
        while 2**i <= l loop
            i := i + 1;
        end loop;

        return i;
    end function;


    -- Given the length of an ECC Word, how many bits are data
    function num_data_bits(l : positive) return positive is
        variable i : integer := 0;
    begin
        while (2**i) + i < l - 1 loop
            i := i + 1;
        end loop;

        return l - i - 1;
    end function;


    --Given the length of a data word, how long is the codeword
    function code_length(l : positive) return positive is
    begin
        return l + num_ecc_bits(l) + 1;
    end function;
```

```
    --Given a data word, calculate the ECC bits + Parity bit
    function calc_ecc_bits(s : std_logic_vector)
                                    return std_logic_vector is
        variable n : integer := num_ecc_bits(s'length);
        variable ecc_bits : std_logic_vector(n-1 downto 0)
                                        := (others => '0');
        variable i : unsigned(31 downto 0) := to_unsigned(1, 32);
    begin
        while i <= s'length loop
            for u in ecc_bits'range loop
                ecc_bits(u) := ecc_bits(u) xor (s(to_integer(i-1)) and i(u));
            end loop;
            i := i + 1;
        end loop;

        return ecc_bits & xor_reduce(ecc_bits);
    end function;
    --Given a data word return the resulting code word
    function to_ecc_vector(s : std_logic_vector) return ecc_vector is
    begin
        return ecc_vector(std_logic_vector'(s & calc_ecc_bits(s)));
    end function;

    --Given a codeword return the corrected data word.
    function to_slv(e : ecc_vector) return std_logic_vector is
        variable l : positive := num_data_bits(e'length);
        variable data_in : std_logic_vector(l-1 downto 0);
        variable ecc_in : std_logic_vector(e'left-l downto 1);
        variable ecc_calc : std_logic_vector(e'left-l downto 0);
        variable ecc_xor : std_logic_vector(ecc_in'range);
        variable data_out : std_logic_vector(data_in'range);
        variable p_bit : std_logic;
    begin

        data_in := std_logic_vector(e(e'left downto e'length-l));
        ecc_in := std_logic_vector(e(ecc_in'range));
        ecc_calc := calc_ecc_bits(data_in);
        ecc_xor := ecc_in xor ecc_calc(ecc_calc'left downto 1);
        p_bit := xor_reduce(ecc_in);

        for I in data_in'range loop
            if to_unsigned(I, ecc_xor'length) = unsigned(ecc_xor) - 1
               and p_bit = e(0) then
                data_out(I) := not data_in(I);
            else
                data_out(I) := data_in(I);
            end if;
        end loop;

        return data_out;
    end function;

    function to_ecc_vector(s : unsigned) return ecc_vector is
    begin
        return to_ecc_vector(std_logic_vector(s));
    end function;

    function to_ecc_vector(s : signed) return ecc_vector is
    begin
        return to_ecc_vector(std_logic_vector(s));
    end function;

    function to_unsigned(e : ecc_vector) return unsigned is
    begin
        return unsigned(to_slv(e));
    end function;

    function to_signed(e : ecc_vector) return signed is
    begin
        return signed(to_slv(e));
    end function;

end package body;
```

Method

Power consumption was measured my monitoring the input voltage to the device as well as the current draw from the source. Power was delivered from a Keithley 2231A-30-3 DC Power Supply and both current and voltage were monitored with a Moku:Lab. Voltage was monitored directly, while current observations required a high-side shunt & amplifier as illustrated in Fig. 61. The amplifier was calibrated with respect to the power supply at DC and was accurate to ±2mA.

In order to isolate various components in the design, the SoM (Cormorant) and breakout board were tested in isolation, as well as in combination while the FPGA/CPU implemented varying designs that stimulated components in differing modes of operation. Subtracting sets of results allows us to estimate the power consumption of various features in the design as well as total power consumption.

Several software designs were used to stimulate specific hardware features:

- **Empty** An empty design that does not but drive pins to a static, sane default value. Differs from Erased in that Erased does not assign default pins values.
- **IMU** Runs the IMU components (Accelerometer, Gyroscope and Compass) at their intended rates as well as running state estimation algorithms.
- **Radio** Runs the telemetry radio with differing amplifier power. The radio transmits with a duty cycle representative of it's expected behaviour.
- **IMU+Radio** A combination of the IMU and Radio designs.

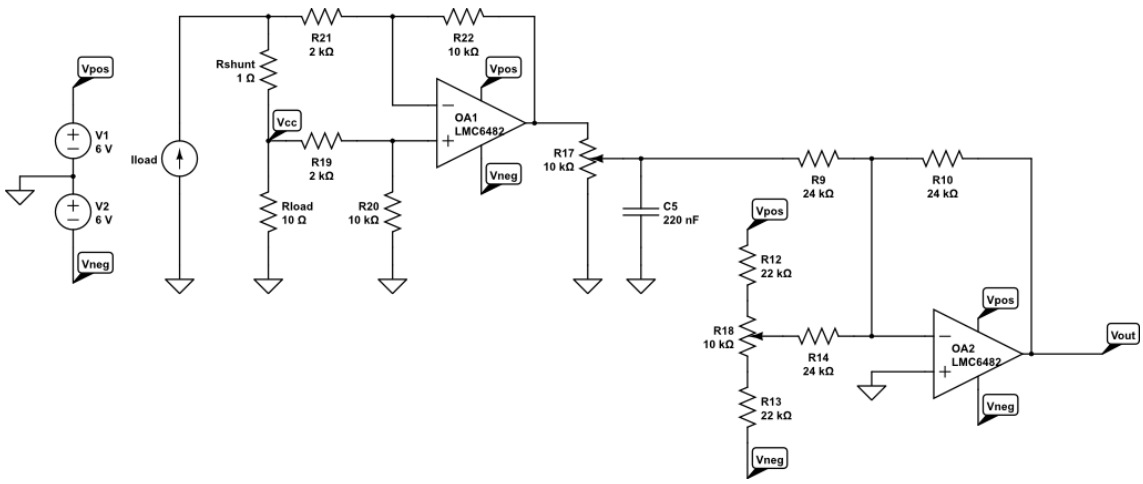

Fig. 61 High side current amplifier.

• **CPU** Enable the CPU at differing frequencies. An empty software implementation is provided to simply loop infinitely.

Results

Eight prototype units were manufactured, of which 2 had uncorrectable faults (3 and 8). Unit 1 appeared to function before failing all tests involving the radio. These three units were omitted from further tests. The remaining 5 units were each programmed with each of the software designs in sequence and their power consumption measured. These 5 units provided consistent results which are shown in Fig. 62 and Fig. 63.

		Breakout Only	Cormorant Only					Combined											
Cormorant	Breakout		Erased	Empty	CPU 100MHz	CPU 140MHz	IMU	Erased	Empty	IMU	Radio Base	Radio Peak	Radio Mean	Radio HGM Base	Radio HGM Peak	Radio HGM Mean	IMU + Radio Base	IMU + Radio Peak	IMU + Radio Mean
1	1	25	143	78	215	260	89	160	116	125									
2	2	25	136	77	288	341	93	156	125	137	172	498	196	169	642	200	185	505	207
3	3	25																	
4	4	25	136	80	277	338	91	157	97	108	142	440	165	141	570	172	156	454	178
5	5	25	160	85	299	346	97	178	103	114	149	452	171	145	592	175	161	462	183
6	6	24	154	80	280	341	91	190	94	101	140	427	164	137	551	166	149	437	170
7	7	25	159	77	274	331	89	137	100	111	145	481	172	142	626	176	160	491	184
8	8																		
	Average (mW)	**25**	**148**	**80**	**272**	**326**	**92**	**163**	**106**	**116**	**150**	**460**	**174**	**147**	**596**	**178**	**162**	**470**	**184**

Fig. 62 Power consumption results.

In a configuration deemed the minimum required to control and navigate and aircraft autonomously, the SoM was found to draw ~92mW. This provides only for the SoM using the FPGA and IMU components and excluded the breakout board.

In a more typical configuration which includes the breakout board and the radio for telemetry the complete device drew ~184mW on average and could reach peaks of ~596mW when the radio was transmitting in High Gain Mode (HGM). This still excludes the use of the CPU which was found to draw 58mW + 1.35mW/MHz. The breakdown of power consumption of each component if shown in Table 7.

Component	**Power (mW)**
SoM	80
Breakout (quiescent)	23
IMU	12
CPU	58 + 1.35/MHz
Radio (active-quiescent)	44
Radio TX	309
Radio TX HGM	449

Table 7 Power consumption by component.

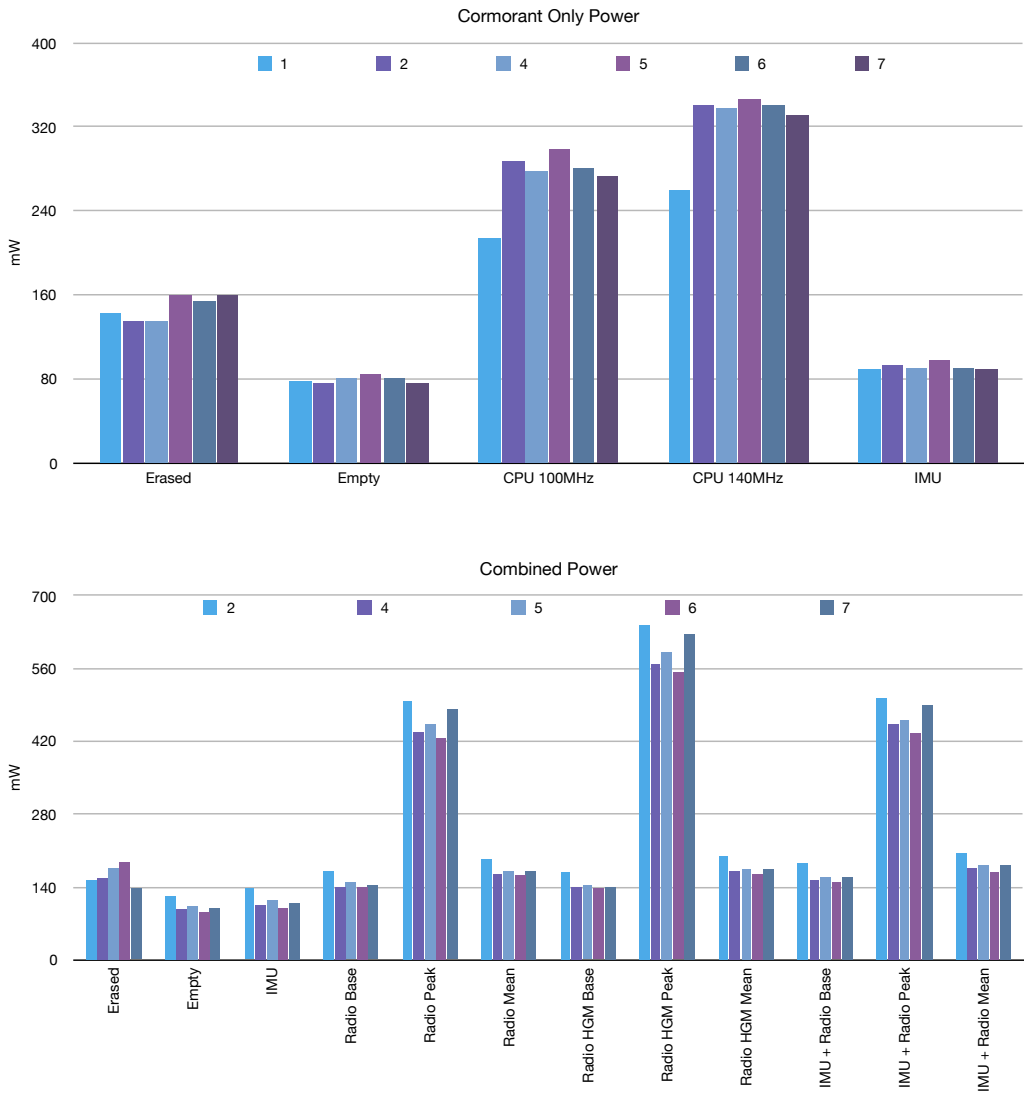

Fig. 63 Power consumption results graphed.

Some of these components power consumption can be estimated after the software design has been synthesized using the vendors tools.

FPGA Design	Power Increase Estimated (mW)	Power Increase Measured (mW)	Factor
CPU 100MHz	80	192	0.42
CPU 140MHz	105	246	0.43
IMU	5	12	0.42

Table 8 Estimated vs Measured power consumption.

Conclusion

The original intended design was to produce a processing system capable of autonomously controlling an aircraft while providing the ability to formally verify the software components as well as harden those components against SEEs, while maintaining a power budget of 100mW. These results show that this design does achieve those power consumption goals in a minimal configuration.

A minimal configuration provides for IMU activity as well as navigation and control processing capabilities using only discrete logic (FPGA). This configuration does preclude the ability to communicate with the device either through telemetry or command and control signals, though data can be logged to persistent storage on the device for later retrieval. A more typical configuration would include the breakout board and radio which would require approximately ~184mW on average.

Some discrepancies between theoretical and measured power consumption figures which indicate that the device is consuming more power than its design intended by a factor of ~0.42. This could be explained several ways; the vendor tools could under estimate power consumption, possibly due to favourable assumptions included in these calculations. The power supplies could also contribute as there design efficiency is only expected to be >80%, but could in fact be much lower in their actual operating conditions.