Master's thesis

2021

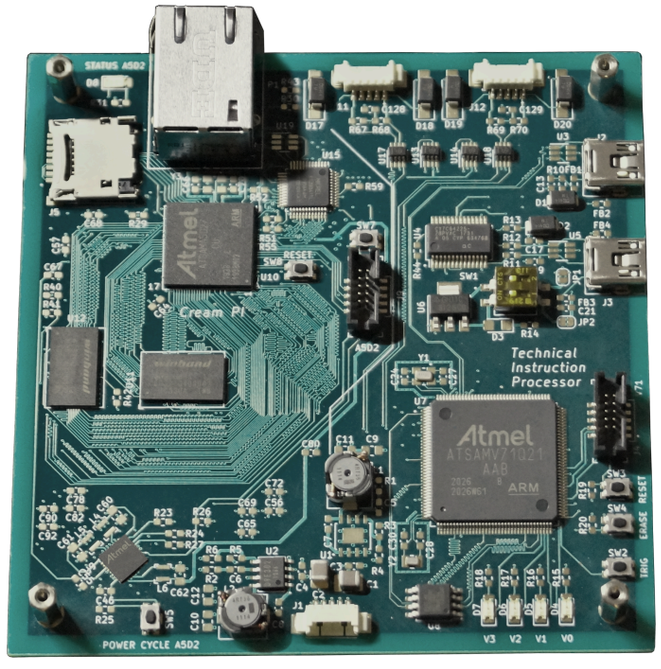
Kolbjørn Austreng

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Kolbjørn Austreng

Development of a Robust Autopilot Platform for use in Autonomous Vehicles

June 2021



NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Development of a Robust Autopilot Platform for use in Autonomous Vehicles

Kolbjørn Austreng

Engineering Cybernetics
Submission date: June 2021
Supervisor: Jan Tommy Gravdahl

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Summary

The Master's thesis describes the development work and validation efforts of a redundant modular autopilot platform for use in autonomous vehicles.

The work covers the entire process from a given requirement specification, to the end result of a complete integrated autopilot solution.

The platform is then verified to perform to a satisfying degree in respects governing hardware and software separately, as well as in an a complete integrated setting through Hardware in Loop testing.

Sammendrag

Masteroppgaven beskriver utvikling- og etterprøvingsarbeidet av en redundant og modulær autopilotplattform for bruk i autonome fartøyer.

Arbeidet dekker hele prosessen fra en gitt kravspesifikasjon, til en komplett integrert autopilotløsning som sluttresultat.

Det bekreftes også at plattformen yter i en tilfredsstillende grad innenfor både hardware og software separat, i tillegg til i et integrert scenario gjennom bruk av "Hardware in Loop"-forsøk.

Dedications

None of the work presented in this text would be possible without the right tools and knowledge. All hardware and software presented within these pages stand firmly on a foundation of mostly free and open source tools. The author extends his heartfelt thanks to anyone contributing their time and effort to the various open source communities of the world.

Furthermore, in terms of knowledge, a deep thank you is also in order to the wonderful people I had the privilege of coming in contact with during my time at Revolve NTNU. In the boundless wisdom of hindsight, there are many things I would have liked to done differently during my time with this organization. That being said, I cannot imagine a better group of people to instill an engineer with a profound sense of technological camaraderie, and nurture a love for seeing *theory in practice*, fearlessly attacking the merciless complexity of real world problems.

You guys rock!

1 Introduction

With the increasing rise of unmanned and autonomous vehicles for tasks that are either too dangerous, costly, or cumbersome for humans to do, comes the need for reliable control.

In recent years, with the advent of more affordable drones for the consumer market, as well as sustained academic interest, we have seen a flurry of autopilot solutions for the "tinkering audience", such as the ArduPilot, APM, or Pixhawk. These have all seen incredible success and continue to find use in new applications. However, common for most such off-the-shelf autopilot solutions, is that they follow a *monolithic design paradigm*, in the sense that they consist of a single board that is responsible for sensor sampling, state estimation, path planning, and low-level controller loops. In some products, such as the Pixhawk v5X, some responsibility is shared among two on-board processors, but the overarching monolithic pattern remains the same. This may have an impact on available computational resources, especially when routines of widely disparate priorities- and urgencies must be reconciled on a single target platform.

In such designs, a single point of failure is inherently built in; it is in part a defining characteristic of the architecture itself. This also necessitates an undue amount of trust in the end user code - namely, the autopilot is more often than not designed with the implicit assumption that programmer of the system will write reliable code. This has been demonstrated to be a dangerous assumption to make even in the best of circumstances, let alone regarding mission-critical control algorithms.[26][3]

The present work aims to address these two concerns by developing a novel autopilot concept that centers around a clear distinction between a low-level, high criticality part - and a high-level, lower criticality part. This two part distinction makes it possible to introduce dedicated computational resources to run unproven- and experimental control algorithms in parallel with a safer alternative (e.g. a PID), ready to take over in the case of failure. As an added benefit, this will also allow the use of more resource intensive control algorithms, that may not be suitable for a highly constrained typical microcontroller.

Additionally, the proposed autopilot system is fully *platform agnostic*, in the sense that it makes no attempts at forcing a specific sensor array upon the end user. Rather, it expects sensor information to be presented over a reliable connection with provable real-time constraints; that is, one of the two redundant Controller Area Network Flexible Datarate (CAN-FD) buses. Alternatively, the embedded Ethernet interface may be used for less critical data requiring higher transfer capacities.

The proposed design is a double edged sword. On the one hand, the nature of the design makes it harder to use as a "*one stop shop*" solution, and will likely not cater to the uninitiated that do not have at least a rudimentary background in applied control systems. On the other hand, the distribution of areas of responsibility makes the autopilot highly flexible and extendable to new and possibly more demanding applications than most existing off-the-shelf solutions.

The finished autopilot system has been named "Helmsman", as a nod to the etymological roots of the name "*Cybernetics*" - stemming from the Greek "kubernetes", namely meaning *helmsman of a ship*.[33]

1.1 Structure of work

High emphasis is placed on robustness aspects of the developed circuit, and how these may be implemented. Measures undertaken to improve these qualities are first discussed in section 3, presented with underlying theory and arguments for specific circuit design principles.

In section 4, we present the implementation endeavors of the Helmsman, complete with discussions on specific details that do not make themselves relevant in the design phase.

As far as it has been feasible to do so, all design aspects have been tested an validated for efficacy. This is presented in section 5.

Finally, from our testing efforts, we present conclusions and proposals for future work based on the Helmsman in section 6.

1.2 A clarifying note on terminology

For the most part, only nomenclature that is already established in the relevant fields to this work is used. Here, a few clarifying notes are presented where confusion might arise in readers coming from other specializations.

1.2.1 On "robustness"

The heart of the presented work is dedicated to various implementation aspects of a "robust autopilot platform". In this work, we mostly mean *fail-tolerant* and *redundant* system design when using the term "robust".

In terms of controller software running on the autopilot platform, there is nothing preventing such a scheme from being "robust" in the sense of *robust and adaptive control*, but no dedicated focus has been given to that through the development of the Helmsman autopilot system.

1.2.2 Fault vs. Error vs. Failure

The `IEC 60050` standard explains the meaning of these terms as follows:

- **Fault** A device's inability to comply with some required functionality. This will often be the *symptom* of a *failure* in the device, but may also exist independently.
- **Error** An inconsistency between a *calculated*, *observed*, or *measured* value and its *true specified* or *theoretically expected* value.
- **Failure** The cessation of a device's ability to reliably meet a specified requirement.

These terms are often illustrated as in figure 1, with the motivation being that various discrepancies may cascade to different parts of a larger system, by a *failure* eventually manifesting itself as *fault* somewhere else.

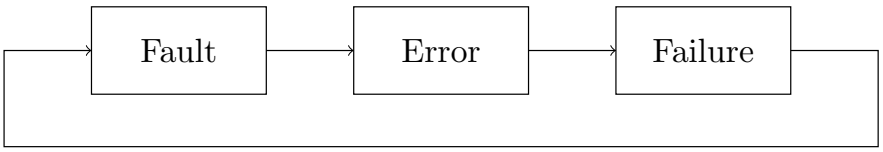


Figure 1: Illustration of deviance states according to `IEC 60050`.

`IEC 60050` makes the distinction that a *fault* is a *state* that a device is in, while a *failure* is an *event*. As such, according to this standard, the term *failure* is not applicable to systems that consist entirely of software. The

argument being that any deviation from specification and observed behavior was always built into the software itself, in form of a bug in the code, and did not spontaneously arise.[21]

The present author finds this restriction needlessly limiting in the context of the current work, and has elected to use the term *failure* freely in discussions of both hardware- and software based system components. The argument for this being that "production ready software" will oftentimes contain latent bugs that do not hamper the software's ability to perform according to specification. Hence, while the software is technically *faulty* in some sense, it firstly makes little difference to classify it as such - and secondly, the developer of the software will usually have no way of knowing that the software contains a bug before it manifests itself. Therefore, it is advantageous to colloquially refer to the manifestation of of software *fault* as a *failure*.

Readers who are proponents of the terminology used in `IEC 60050` should be aware of the this author's more liberal use of the terms therein. This should not present opportunities for confusion.

1.2.3 Circuit terminology used in this work

Printed Circuit Board (PCB) terminology seems to be well established at this point. For the few terms that differ between camps or have synonyms, we have used wording from (Horowitz 2015) and (Wilson 2018).[32][38]

2 Regarding the 2020/2021 component shortage

Following the Covid-19 pandemic, global demand for Integrated Circuits (ICs) spiked, as a result of an increasing need for computers, tablets, and phones for home office use. Long IC production times, complex production pipelines, and a market that is dominated by only a few manufacturing companies almost exclusively located in Asia - have all exacerbated the issue further.[19]

The global component shortage has hit the automotive sector particularly hard, following a temporary 2020 fall in IC demand that later surged back up, catching automotive companies off guard. Faced with lead times of up to half a year, companies have responded with panic buying and component stockpiling.[29]

This trend has heavily impacted the current work, as many components that were considered for use in the autopilot platform are also sought after by various automotive applications. Where such circumstances have most significantly impacted the work, it will be made clear in the text. Furthermore, a discussion about future autopilot iterations where components are more readily available is presented in section 6.

3 Development Process and Methodology

The high level requirement specification of the autopilot system, that has motivated all subsequent design choices, may be seen in table 1. In the following sections, we will frequently refer back to these for design guidance.

Table 1: Requirement specification

Item	Description
RS01	The circuit board shall implement a controller scheme capable of submitting corrective feedback to a simulation of an Unmanned Aerial Vehicle (UAV) in a Hardware In Loop (HIL) setup. Sensor data may be assumed to originate in the simulation and be provided to the autopilot via a data link.
RS02	The developed autopilot platform should be applicable as a future test bed for continued controller scheme experimentation, in the sense of providing the capability of running concurrent regulation algorithms. In addition to this, the platform should be able to accommodate resource intensive controller schemes requiring the use of an onboard operating system, such as embedded Linux.
RS03	The autopilot should have no single point of failure. That is, neither a single software fault, nor a single faulty IC should lead to erratic termination of operation.
RS04	Insofar that the autopilot circuit board conducts communication with other circuit boards, no single point of failure should permanently disrupt the possibility of further data exchange.

RS05	Experimental "end user code" should not pose an undue risk to continued operation of the autopilot system. That is, even in cases where supplied user code misbehaves in ways such as leaking memory, halting, attempting to set invalid control points, or even crashing, this should not permanently disable autopilot operation. At most, a graceful degradation of performance may be tolerated. By "graceful degradation" it is understood that certain functionality may become unavailable either temporarily or permanently, as long as the overall operation of the autopilot is not terminated.
RS06	Potential for undefined behavior should be prevented in both hardware and software where it is feasible to do so. That is, appropriate measures must be taken to ensure e.g. proper Signal Integrity (SI) conditions, consistent state keeping, and isolation of faults.

While in reality all requirement concerns were simultaneously taken into consideration, and a strategy for both hardware and software were laid in tandem, we will present design choices for hardware and software separately for greater clarity. This separation of discussion is possible because the presented software solutions lend themselves to a wide array of underlying hardware implementations. Thus, the discussion should be easier to follow, without loss of correctness.

In sections 3.4.1 and 3.4.2, we present alternatives for specific hardware design choices, along with the implications each alternative has on the end product.

In section 3.5 we present an overall strategy for a software architecture, before we dedicate a discussion to specific software design implications in sections 3.5.1 through 3.5.2.

For concrete discussions on how the proposed designs are implemented, the reader is referred to a presentation about hardware aspects in section 4.1, and a presentation about software in section 4.2.

Lastly, efficacy of implementation choices and autopilot performance are evaluated in section 5.

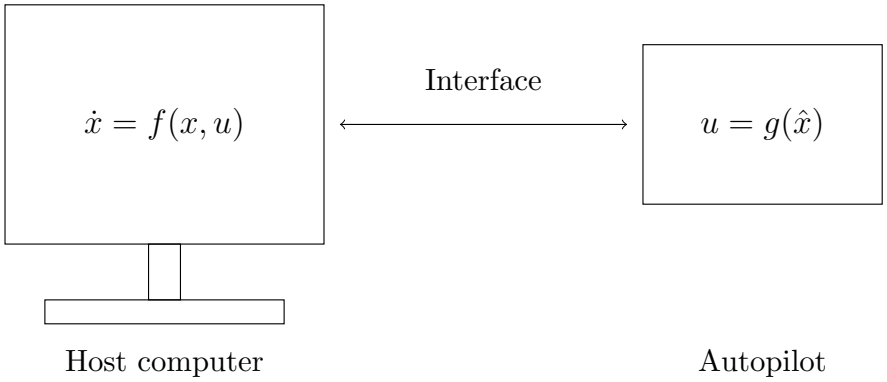


Figure 2: Schematic representation of HIL testing.

3.1 Host computer communication

Requirement specification item `RS01` dictates that the autopilot be able to communicate with a host computer and potentially additional hardware for HIL testing. This is schematically illustrated in figure 2. The discussion in this section will revolve around finding a suitable realization for the arrow market "Interface" in figure 2.

We chiefly identify two main avenues warranting our investigation. Firstly, we may introduce specific communication interfaces, such as Universal Serial Bus (USB), to the autopilot for the sole purpose of providing a *host-platform* data link.

On the one hand, this allows us to completely decouple regular operation from HIL testing, while also providing easier access to the autopilot system. This could potentially make the platform easier to develop software for, thus partly satisfying requirement `RS02`.

On the other hand, introducing additional interfaces will inevitably increase system complexity - ultimately making the platform harder to maintain and develop further. Space constraints on the physical PCB may also lead to integration issues regarding components of higher criticality.

As a different approach, we may also consider using system infrastructure in non-standard ways, such as injecting messages to a bus needed for other purposes - or even utilizing a test interface such as Joint Test Action Group (JTAG) and the Flash Patch and Breakpoint unit (FPB) in ARM based systems to on the fly alter the flow of execution.

Such an approach has the advantage of not requiring additional hardware

components on the autopilot itself, but may necessitate a communications adapter between the autopilot and host computer.

The decrease in required hardware on the autopilot PCB has the potential to simplify the overall circuit layout, which could reduce entry points for hardware issues. However, if the use of existing infrastructure to facilitate HIL testing results in significant system load, this may all together invalidate the test procedure itself in a worst case scenario.

Examples of both approaches may be found in the available literature. In some cases, the HIL environment itself necessitates the addition of extra hardware to allow for meaningful host-platform communication, such as in (Huijgens 2021)[25]. In other cases, such as (Dai 2021), communication may be facilitated via existing hardware without negatively affecting the main running process.[39]

Based on such existing solutions, and motivated by a strive for a higher level of redundancy and usability, the author proposes the following strategy: Two independent USB interfaces shall be introduced to the autopilot board. These are intended exclusively for HIL usage. In addition to this, an extra validation test bed for the autopilot will be developed, allowing for bus-based communication packet injection, should the USB interfaces prove unsuitable. Care will be taken to isolate the USB components from the other components on the PCB, thus allowing for easier future board modifications and revisions.

The additional autopilot test bed is a simple USB to CAN-FD bridge, that may additionally simulate sensor noise. It is simple circuit compared to the Helmsman, and not intended for end application use. Hence, little discussion will be specifically dedicated to its development, but its use will be demonstrated in section 5.

3.2 Redundant communication

The autopilot platform will not directly control actuators and sensors; these features will be provided by other PCBs via dedicated connections.

Requirement specification item `RS04` governs this situation, and demands that we choose a communication scheme that cannot fail if, say, a single transceiver IC fails.

Furthermore, as requirement `RS03` forces us to implement redundant sub circuits, the selected communication method should allow for some form of *multi master access*, in the sense of allowing either redundant sub circuit to assume the role of message initiator.

Additionally, it would benefit the overall system reliability if the chosen communication method is in itself robust. By "robust" in this context, we understand the bus interface as being tolerant of faults, and possibly also capable of isolating faults, by method of barring faulty ICs from accessing the bus.

An alternative that satisfies these requirements, while having the additional benefit of being standardized, is the Controller Area Network (CAN) and CAN-FD bus protocols.[30] Other protocols, such as `RS-485`, `Flexray`, or `Time-Triggered Protocol` were also investigated, but ultimately were not chosen based on a confluence of hardware unavailability (see section 2) and more involved development requirements.

3.3 Number of redundant sub circuits

Requirement specification item `RS03` and item `RS04` both impose criteria for redundant hardware. Item `RS03` immediately disqualifies all solutions where control calculations are only performed on a single physical IC. Item `RS04` is a more lenient requirement, in the sense that it applies more to replication of buses, transceivers, and physical connections, etc.

There is a rich and well established literature connected to the matter of selecting and designing redundant elements in mission- and safety critical hardware. An overview may be found in (Leveson 1995).[24] Overarching strategies have not changed significantly since then, and have made their ways into several standardization bodies, such as in `IEC 61508` and `IEC 62439`.[8][9]

Arguably, one of the most successful concepts from published works governing redundancy for safety and continued operation is that of *voting*. In voting systems, critical components are implemented several times - usually duplicated or triplicated. One failing unit may thus be "voted out" by the other, still functioning, units. A popular version of such a voting system is the Triple Modular Redundant (TMR). Under this scheme, two agreeing devices will outvote a single disagreeing device. Among the areas it has seen high use is the commercial aircraft industry, such as for the fly-by-wire system of Boeing 777.[40] More recently, the same voting scheme continues to see extended use in other areas of avionics, such as aircraft positioning measurements (Grof 2020).[36]

While voting systems may be powerful under correct usage, it is also a well recognized fact that over-zealous application will lead to complex systems that are in fact less reliable than their simpler counterparts.[21]

Based on available literature, it thus seems that a TMR system is highly suitable for the presented autopilot problem. Unfortunately, this was quickly ruled out as a financial infeasibility, as such a design would result in a circuit that is prohibitively expensive to manufacture.[1]

In light of this, it was decided to approach the problem with a somewhat simpler 1-out-of-2 (1oo2) voting scheme, requiring only two hardware copies as opposed to three. In section 3.5 we will explore how we may enhance the availability aspects of such an architecture using software.

[1] The Helmsman autopilot production is personally financed by the author, on a student budget, and hence some cuts were inevitable.

3.4 Hardware configuration in redundant sub circuits

Still on the topic of requirement specification item RS03, it is necessary to determine the specific type of hardware configuration to implement the chosen 1oo2 redundancy.

Importantly, one needs to determine whether one should aim to simply copy an identical design two times - or aim for differing designs that offer similar feature sets. While focusing on a single design and then duplicating this might have a positive influence on development time, ease of circuit layout, component availability etc., it seems to be seldomly used. An an example, design considerations for the Boeing 777 fly-by-wire system explicitly states *design diversity* as a primary aim.[40]

Striving for different sub circuit designs will also indirectly aid us in more easily achieving requirements RS02 and RS05, in the sense of allowing one of the sub circuits to be significantly simpler than the other. We expect this to also have a positive impact on requirement RS06, by introducing fewer venues for undefined behavior as a result of a potentially poorly understood design.

A natural design choice would be to split the circuit in a *low complexity* part implementing the bare necessities for continued operation, and a *high complexity* part satisfying requirement RS02. These are separately treated in sections 3.4.1 and 3.4.2. Finally, a summary of both sub circuits together is given in section 3.4.3.

3.4.1 High complexity sub circuit

The high complexity sub circuit is explicitly required to provide resources necessary to run an onboard operating system, such as embedded Linux. For this purpose, an `ATSAMA5D27` was selected.

Primary selection criteria were support for external Random Access Memory (RAM) and Secure Digital (SD) cards, as this will be required to run a modestly resource intensive operating system - along with high capacity for connectivity, such as support for CAN, Ethernet, USB, and Universal Asynchronous Receiver-Transmitter (UART).

A number of other System on Chips (SoCs) and Micro Processor Units (MPUs) were considered. Common for most alternatives were that they were highly coveted by other organizations during the work of this autopilot system, as explained in section 2.

Ultimately, the choice fell on the `ATSAMA5D27`, partly because this IC has a design deviation that makes it less lucrative for the automotive industry, which ensured it was in stock during the work on the Helmsman. Namely, the IC was produced before the final revision of the CAN-FD standard (`ISO 11898`) was released. Therefore, the `ATSAMA5D27` does not fully conform to the full message format specification of the *flexible datarate* extension.[37]

This deviation will only superficially impact the Helmsman platform, as the chip is still capable of following the regular CAN protocol, which has lower transmission rates and lower packet sizes than CAN-FD. In cases where it becomes relevant to do so, the Helmsman will thus be able to gracefully downgrade bus performance by sending only critical data packets, at a lower speed than during nominal operation.

Auxiliary components that enter into the high complexity design - such as RAM chips, power supplies, and transceivers - are more interchangeable and differ less from supplier to supplier. Hence, we dedicate no additional discussion to the choice of these. For completeness sake, the Bill of Materials (BOM) is provided in appendix A.

A schematic overview of the most central components entering into the high complexity sub circuit is illustrated in figure 3.

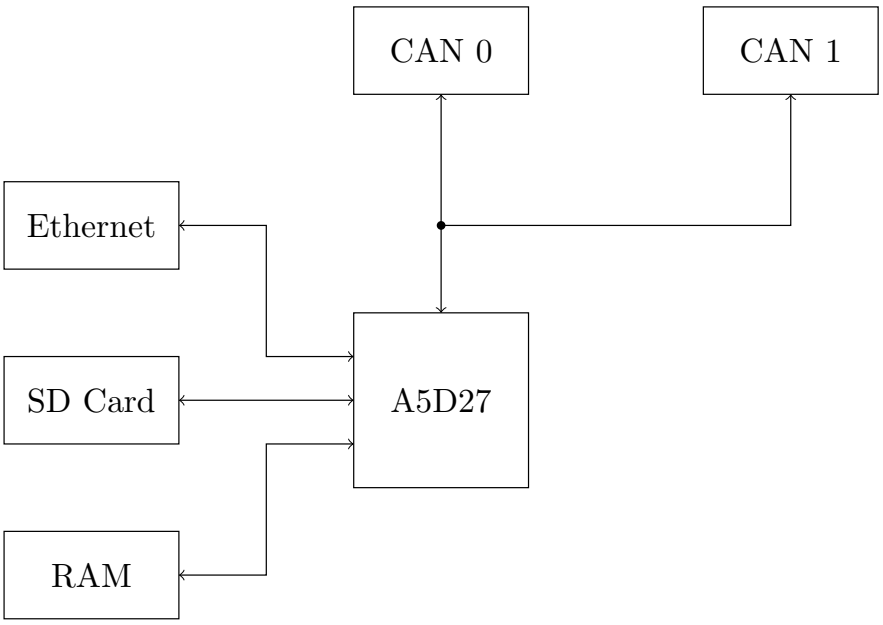


Figure 3: Helmsman high complexity sub circuit schematic.

3.4.2 Low complexity sub circuit

In choosing a suitable backbone for the low complexity circuit, we have a greater amount of freedom than in section 3.4.1, since a reduced feature set will be tolerated, as per requirement specification item `RS05`.

To limit the search space, the author set forth the following additional set of constraints that a suitable IC should meet:

- The IC should have support for two dedicated CAN-FD buses. The reason being that this will allow the low complexity circuit to interface with the same set of buses that the high complexity circuit has access to without extra bridging hardware. This will allow for a seamless transition of control in cases where one IC has to take over operation from the other.
- The IC should have an embedded double precision Floating Point Unit (FPU). This is advantageous as control code relying on floating point arithmetic may be more seamlessly integrated, without having to resort to execution time reducing tweaks - which could ultimately deteriorate code quality, hence impacting requirement `RS06`.
- The IC should be supplied in some form of a dual-inline or quad-flat package. Such a package will take up more PCB area than an equivalent Ball Grid Array (BGA) package, but is far more forgiving to misalignment issues, and is easier to visually inspect for correctness. This requirement was introduced to compliment the high complexity sub circuit, as the `ATSAMA5D27` is supplied in a 289 pin count BGA package. We also argue that this choice might have a positive effect on specification item `RS06`, as we understand a simpler package to present fewer failure vectors than a complex one.

These criteria narrowed the search space down to a handful of candidates, all implementing similar feature sets. Out of these, the choice initially fell on the `ATSAME70`. This chip was selected over similar alternatives, as it uses the same programming- and debug probe as the `ATSAMA5D27`, which limits required development hardware.

Shortly after the initial Helmsman prototype was ordered, the `ATSAME70` became unavailable at all major component distributors. This was solved by selecting the `ATSAMV71` IC instead, which is mostly feature compatible, but sold at a higher price, likely making it less appealing for the automotive industry.

As with the high complexity sub circuit, a separate discussion on auxiliary components is not warranted due to their higher interchangeability. For the

full BOM, the reader is referred to appendix A.

Lastly regarding choice of components, the author would like to draw some attention the the addition of a 128 Mbit Flash memory chip to the low complexity sub circuit. This addition will enable physical code segregation, allowing for several software suites backing up each other in case of failure, as outlined in section 3.5.2. Ultimately, this will enable better coverage of specification items `RS03` and `RS05`.

A schematic illustration of the low complexity sub circuit is provided in figure 4.

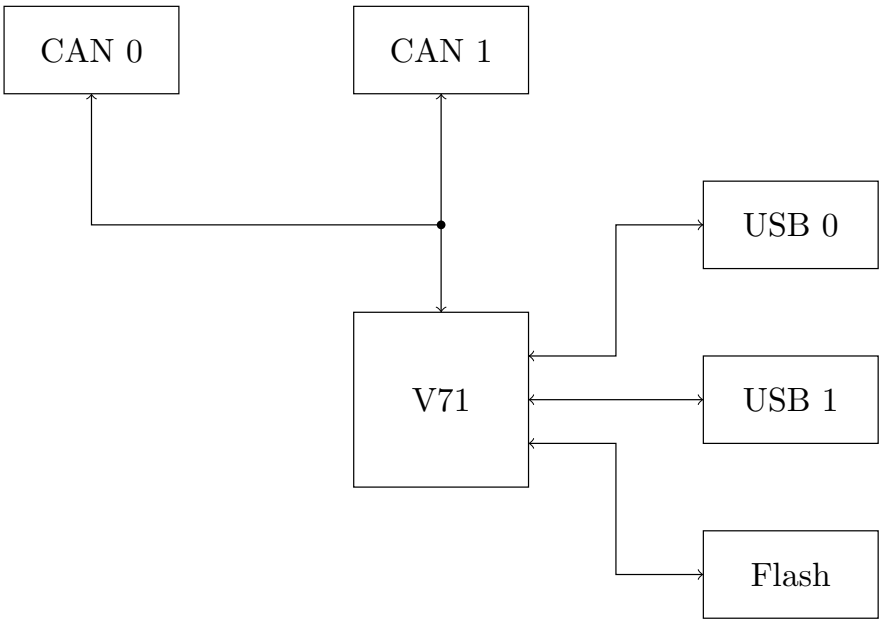


Figure 4: Helmsman low complexity sub circuit schematic.

3.4.3 Joining the redundant sub circuits

The proposed sub circuits from sections 3.4.1 and 3.4.2 need a mechanism for communication to ensure correct cooperation between the two. Ideally, this interface should provide a low overhead, meaning that it should not unduly strain other parts of the system during nominal operation. Hence it is tempting to avoid additional hardware to facilitate this communication, as that would both make the overall design more complex, and put extra load on both sub circuits.

However, we must also satisfy the requirement that no single point of failure should be possible, which must include intra-sub circuit communication, as per requirement `RS03`.

A decent compromise between these requirements may be reached by taking advantage of the fact that both the `ATSAMA5D27` and the `ATSAMV71` support Direct Memory Access (DMA) on their communication peripherals. This will allow us to send and receive messages between the sub circuits using less Central Processing Unit (CPU) time than we would otherwise have needed, ultimately reducing the system load.

Moreover, we may rely on message matching triggering interrupt routines with appropriate priorities on either CPU for communication that has real-time constraints.

In light of this, the author argues for the adoption of a hybrid approach. In normal conditions, a single dedicated UART pair will provide connectivity between the high- and low complexity sub circuits. In a situation where this communication venue fails, the sub circuits may fall back on using the two CAN buses for message passing. A failed UART will likely be the symptom of deeper fault, where the appropriate course of action is for one of the sub circuits to solely take over operation. As such, required communication on the CAN buses may be expected to be only transient, thus not substantially loading these buses over a long time in a partial failure scenario.

Joining the two sub circuits together, we arrive at the schematic illustrated in figure 5. Not shown in the schematic are voltage regulators and associated filters. These are, however, also unique to each sub circuit, as otherwise a power supply fault would cause a total system fault.

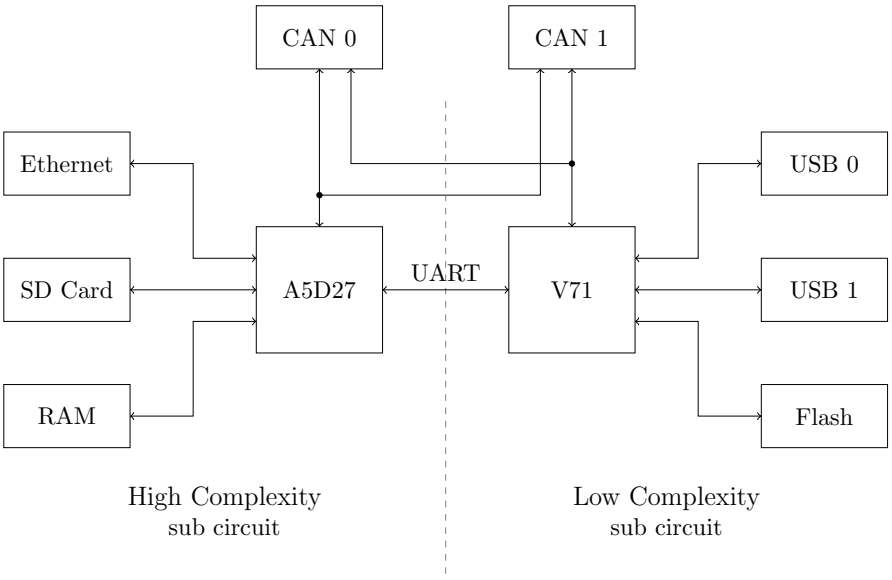


Figure 5: Overall simplified Helmsman schematic.

3.5 High level software architecture

The choice of hardware platform, as illustrated in figure 5, provides a highly flexible basis on which to develop software. As noted in section 3.3, the proposed 1oo2 hardware solution was partially motivated by the restrictive costs of a full hardware based TMR system. Nonetheless, it is possible to make up for some of the lost redundancy by carefully crafting error detecting software with fallback alternatives.

One of the most common methods for introducing software redundancy is called *N-version programming*. In this scheme, N versions of a software suite are developed from the same specification, often by different developer teams. Versions are then brought together in some voting setup, mimicking hardware redundancy implementations.

While the established literature agrees N-version programming provides *some* benefit, there seems to be little consensus regarding what *degree of benefit* one might expect. For example, when NASA investigated the merits of N-version programming in 1990, they found only a modest gain in terms of reliability. Furthermore, they concluded that independently developed versions of software alone is not sufficient to provide a meaningful reliability gain. Instead, an emphasis on reducing coincident faults that independently arise from the same specification is necessary for increased reliability.[10]

Similarly, a survey from 2001 found that although N-version programming *usually* provides a reliability benefit; still, they were able to point to TMR systems that had worse reliability than single software versions alone in pathological cases. Also this survey concluded that a higher emphasis should be placed on reducing coincident faults by increasing design diversity.[5]

Contemporary literature seems focused on specific novel implementations for N-version programming, rather than inquiries into their effectiveness at increasing redundancy. There also seems to be no established successful method of estimating the "degree of diversity", or even defining it, for a set of software solutions, although a few attempts have been made, such as (Nascimento 2012).[2]

We thus have to conclude that we are in the dark regarding a sound method of developing a guaranteed redundant software suite, a least to a somewhat higher degree than regarding hardware design. Based on observations made in the available literature, we may take an educated guess on an effective strategy. We will lay out a proposed strategy here, and later attempt to verify the merits of our work, in section 5.

- Firstly, we note that both (Eckhardt 1990)[10] and (Littlewood 2001)[5] were able to create redundant software systems with lower reliability than single software counterparts, with relative ease. Based on this, we might want to avoid complex voting systems, out of fear that this would be only be of detriment.

- Secondly, the available literature seems to unanimously agree that different and independent teams are a prerequisite of truly ensuring design diversity, insofar that true diversity is even possible. Since *all* developer teams in this work would consist solely of the author, they could hardly claim to be particularly independent by any stretch of the imagination.

- Lastly, by embracing the already existing different potential for fallback solutions in the high- and low complexity sub circuits of section 3.4.1 and 3.4.2, and further building on this difference, we may make a *heuristic guess* that an ultimately more diverse design solution will be reached. As of yet, we have no guarantee that this approach will lead to a system with fewer coincident faults, as noted by (Eckhardt 1990)[10]. However, for now we will take a leap of faith, and attempt to validate- or disprove our design in section 5.

With a background in these simple heuristics, we propose two different software fallback solutions for the two sub circuits making up the autopilot platform. These are outlined below. Details on how these approaches may be realized are deferred for section 4.2.

3.5.1 High complexity sub circuit fallback chain

For the high complexity sub circuit, we propose a simple two step fallback. Under nominal conditions, this circuit will be running a modestly powerful operating system, such as embedded Linux. Linux, as noted by (Allende 2021), is a ubiquitous operating system that is continuing to find areas of application in safety critical domains.[20]

We may reasonably conclude that this operating system will highly likely be far more stable than any fallback solution we may come up with. Still, a kernel panic cannot be ruled out - in which case we may fall back on a custom control firmware. This firmware naturally has to implement the bare necessities for continued autopilot operation - likely for the singular reason of safely ending a mission without loss of vehicle in a real scenario.

A block diagram illustrating the various software constituents, as well as the single fallback link is provided in figure 6. Here, drivers and control algorithms for the nominal case are lumped together with the operating system block, to indicate that an end user has a high degree of freedom regarding the implementation of these, and that these software components will run as tasks governed by the operating system itself.

The block indicating “Firmware” represents a much more rigid structure, that should not contain other auxiliary components than those indicated; drivers to interface with the hardware, and a basic control algorithm for continued operation.

The reader is referred to section 4.2.1 for a discussion on implementation of the proposed design.

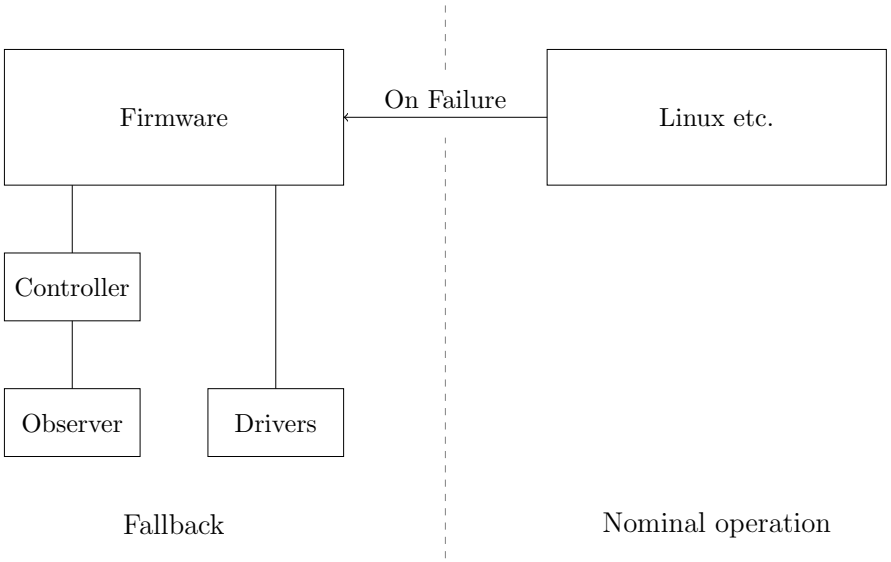


Figure 6: High complexity sub circuit fallback chain.

3.5.2 Low complexity sub circuit fallback chain

For the low complexity sub circuit, we propose a longer fallback chain than for the high complexity sub circuit. This choice is motivated by the lack of a well tested (and presumably stable) operating system like Linux.

Despite this, the avenue of introducing a lower powered real time operating system for running various control and estimation algorithms is inviting. One such operating system, that warrants a brief discussion, is FreeRTOS.

FreeRTOS is a free and open source Real Time Operating System (RTOS), that has seen widespread adoption in medium- to high end resource constrained microcontrollers, such as the `ATSAMV71`. FreeRTOS prides itself on fast execution, while still having a reasonably low memory footprint. In FreeRTOS, kernel scheduling follows a priority based preemptive algorithm, and context switching is realized by pushing task contexts to each thread's stack. These features and design details make FreeRTOS a good fit for the `ATSAMV71`, and interesting from our point of view.[14]

From the author's experience, FreeRTOS is a *mostly good* mixed bag. The project has been developed continuously for 15 years, in close collaboration with influential IC companies.[14] This cuts both ways, in making available documentation mature and overall satisfactory - while on the other hand, some of the available Application Programming Interfaces (APIs) can feel a bit "tacked on", as they were introduced to cater to the needs of a specific client or some group.

Accepting the good parts of FreeRTOS, we may leverage its maturity and stability as a basis on which to implement our first fallback link. This may be done by taking advantage that FreeRTOS tasks offer some isolation from one another, and can take over in the event that one should fail. We hence propose dividing FreeRTOS tasks into two regimes; the first of which allows for experimental features and code that can be assumed to be less reliable. In case we observe unexpected results from the first regime, we may fall back on a second regime of FreeRTOS tasks, that must by some standard be "less experimental" than the first one.

Should FreeRTOS itself fail, we may fall back on a custom firmware, similar to the approach proposed in section 3.5.1. The difference here is that we intend to create a longer chain of different code to fall back on. This is advantageous over the single fallback option, as it allows for an increasingly "strict" chain of fallback alternatives. As a simple illustration, one might imagine to start a mission off with a *sliding mode* H_∞ based algorithm - to then fall back on a more well-proven LQR Kalman based approach. Should

that fail, it would then be possible to fall back on, say, a simple least squares backed Proportional Integral Derivative (PID) solution.

As a final fallback solution, we propose a rigid, mostly non-configurable, layer of software that will be relied upon to safely end an ongoing mission. Should all other approaches fail, one must assume that a majority of the sensors have gone offline, that actuators have stopped working, or that the craft has received substantial damage. All of these would warrant a safe discontinuation of the mission. We will refer to this software layer as the "`Containment Domain`", as its purpose is solely to contain the effects of faults until the mission may be safely ended.

The definition of such a Containment Domain will necessarily be different from application to application. For example, in a UAV setting, its purpose might be to lock the craft control surfaces into a position where controlled gliding descent can be expected. For a simpler application domain, such as a positive buoyancy underwater vehicle, it might be enough to stop any and all actuation altogether while activating a location beacon, should the craft have one.

The low complexity sub circuit fallback chain may be seen illustrated in figure 7. A discussion detailing implementation aspects of the proposed architecture is provided in section 4.2.

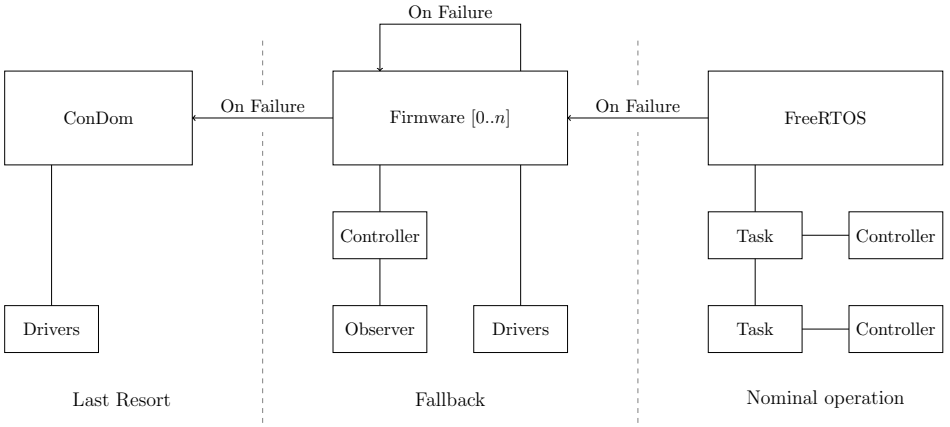


Figure 7: Low complexity sub circuit fallback chain.

4 Implementation

This section outlines implementation details of the Helmsman platform. The design that is implemented is the design that was presented in section 3. In some areas, limitations and design considerations that were not apparent or relevant on an architecture level will further refine our design. In such cases, background literature will be referred to, and implementation trade-offs and choices made will be presented where appropriate.

Mirroring sections 3.4 and 3.5, we first approach hardware implementation details in section 4.1, before presenting the corresponding software details in section 4.2.

4.1 Hardware implementation

To make the following discussions and references more tangible, we first present the finished PCB in figure 8. We then give a brief explanation that outlines the overall board structure in section 4.1.1. Next, in sections 4.1.2 through 4.1.5, we continue to discuss specific implementation aspects that motivated the finished design seen in figure 8.

For additional clarity when comparing the layout process to the complete board, the Helmsman PCB is also shown without components in figure 9 and figure 10.

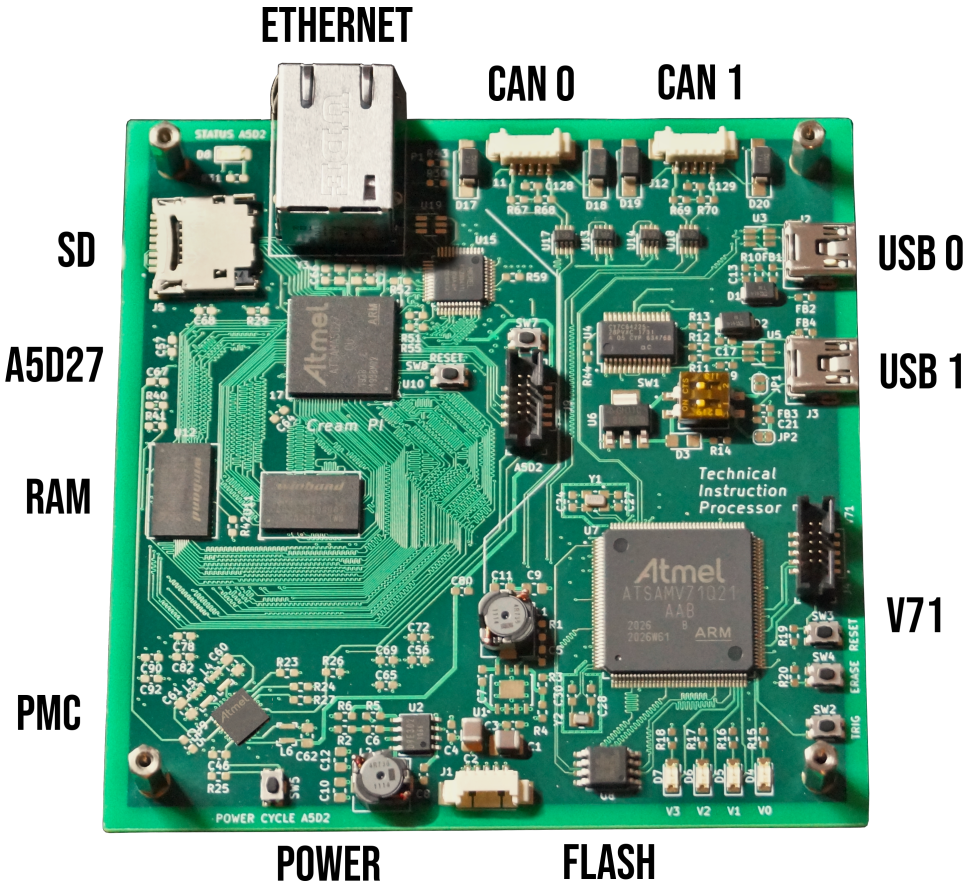


Figure 8: Helmsman PCB with components.

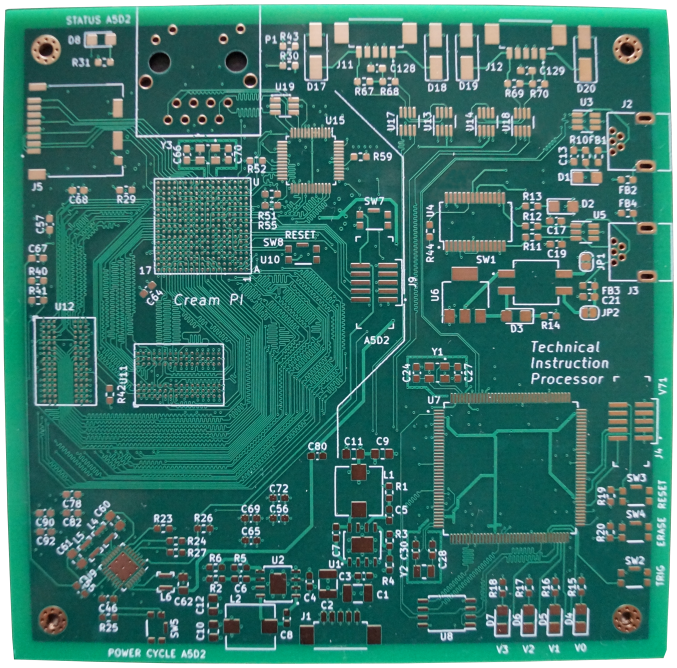


Figure 9: Helmsman PCB front side.

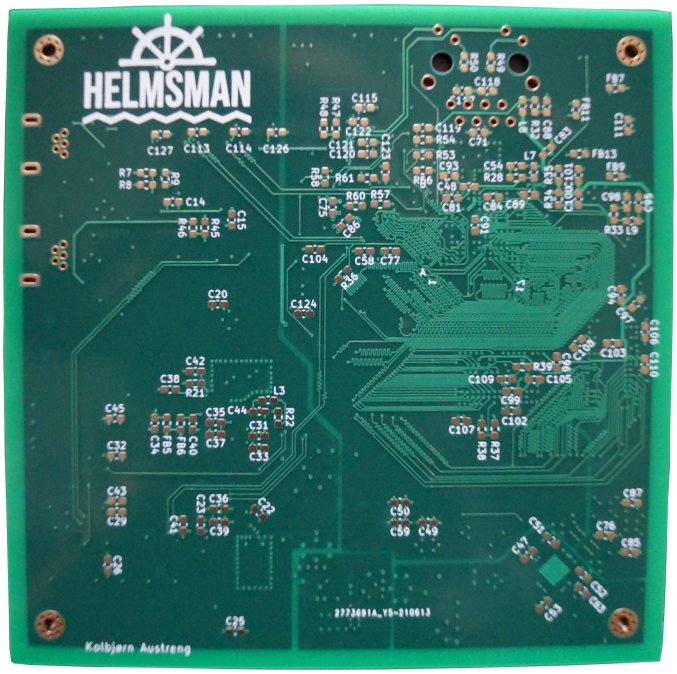


Figure 10: Helmsman PCB back side.

4.1.1 Overall board structure

At first glance, we are able to identify certain features that have a route in physical necessity or convenience. For example, all connectors and the SD card slot are placed along the PCB edges. This allows for easier access, and has relatively little bearing on the rest of the design, other than somewhat dictating where ICs that interface to said connectors should be placed. We see one exception to this, that was made out of space constraints; the Single-Wire Debug (SWD) connector for the high complexity sub circuit is placed in the middle of the PCB, rather than along any of the edges. Since the high complexity sub circuit will nominally run an operating system located on its SD storage, and not have to be programmed as often as the low complexity sub circuit, this will not significantly impact the usability of the autopilot system during software development.

After connector placement had been determined, the remaining components were roughly divided into two board halves; one for the high complexity sub circuit, and one for the low complexity sub circuit. The reasoning for this division is two-fold. Firstly, a clear sub circuit division will make board traces easier to route. Secondly, it will marginally improve circuit reliability, by reducing the risk of total board failure, if one of the sides were to suffer superficial physical damage on the outer board layers.

A wide region toward one of the board edges was then reserved for power supply ICs. This was chosen to so that the board power connector would be physically far away from communications connectors, as they share the same connector type. Secondly, by reserving a wide area on the PCB, we aim to aid thermal dissipation from the switching power regulators.

When these elements were in place, the rest of the circuit components were laid out in the remaining available space, with an emphasis on co-locating high frequency ICs that need to interface with each other.

4.1.2 Return path considerations

The popular assertion that current flows in the direction of least *resistance* is a bit of a misnomer. Particularly in PCB applications, where frequencies are higher, and the specific trace geometries introduce significant reactance, we must update instead focus on the total *impedance* of a return path.[6]

Impedance, Z, is defined by equation 1. Here, R represents resistance, and X represents the reactance; j is the imaginary unit.

$$Z = R + jX \tag{1}$$

Since both a higher capacitance, and a higher frequency, will contribute to the lowering of capacitive reactance, the reactance will be dominated by inductive effects. For PCB applications, we may then rather use equation 2 to give a reasonable approximation of the impedance in a trace.

$$Z = R + j\omega L \tag{2}$$

For very low frequencies, the only impedance contributing factor will be the copper resistance in the trace itself. In typical PCB applications, however, this typically changes radically somewhere between 1 kHz and 10kHz, where trace inductance begins to dominate entirely.[6]

This fact has deep reaching implications for how we should lay out our circuit traces. Firstly, it means that if the circuit has a dedicated ground plane underneath a current carrying trace at high frequencies, almost the entirety of the return current will be carried directly under the signal line. Secondly, it means that if the ground plane has any discontinuities under a trace, such as plane slits, that trace will experience a significant impedance discontinuity as well. It is a well established matter of fact that impedance discontinuities cause signal reflections, and contribute to Electromagnetic Interference (EMI) issues.[38][7]

This has motivated a high focus on avoiding reference plane splits where possible. In the Helmsman autopilot platform, one full, unbroken inner layer is dedicated as a ground plane. Another inner layer is dedicated as a voltage supply plane. Unfortunately, in the latter, some slits were necessary, to separate different voltage levels from each other.

4.1.3 Transmission line effects

There is no set frequency or length at which a PCB microstrip line *becomes* a transmission line. It will always behave like a transmission line irrespective of what kind of signal is sent along. That being said, for many applications, transmission line effects may be neglected.[38]

The number of applications where the effects of transmission lines become noticeable to the point that they can no longer be ignored, however, is steadily increasing. One factor is the faster switching frequencies that enter into modern circuits. Still, even a circuit with the unlikely clock frequency of one transition per year will experience transmission line effects if the signal rise time becomes short enough. This is a natural result from the Fourier transform of a quickly rising signal.[7]

In the Helmsman circuit board, we expect signal rise times on the order of a few nanoseconds. This is fast enough to justify special attention to transmission line effects such as signal reflections.[37]

We may do this by matching our microstrip line impedance to the given characteristic impedances of the IC terminals where possible. The ICs used in our application typically specify a characteristic impedance $Z_0 = 50\ \Omega$, which can easily be matched by using a PCB manufacturer's impedance calculators, which will return a recommended microstrip line width. In this work, JLCPCB was chosen as the PCB manufacturer, so their guidelines were followed.[23]

Another harsh reality of transmission lines, is that signals propagate with a finite speed through a circuit. For many signals, such as differential pairs, or collections of parallel lines, it is of importance to take steps to ensure that signals arrive at the recipient at the same time. An example where this is an essential focus is Double Data Rate (DDR) RAM.[17]

Our high complexity sub circuit incorporates 4 GB of DDR3L RAM, as such, care was taken to length- and impedance match lines where possible, and otherwise follow line grouping suggestions from NXP's application note `AN2582` and Micron's technical note `TN-41-08`.[17][11]

Length matching was achieved by manually adding meandering patterns in the individual microstrip lines that were too short. An extreme example of this may be seen in figure 11. Here, the red square indicates the location of the `ATSAMA5D27` IC, while each of the blue rectangles indicate one 2 GB DDR3L RAM IC each. Only the PCB's top layer is included in the figure.

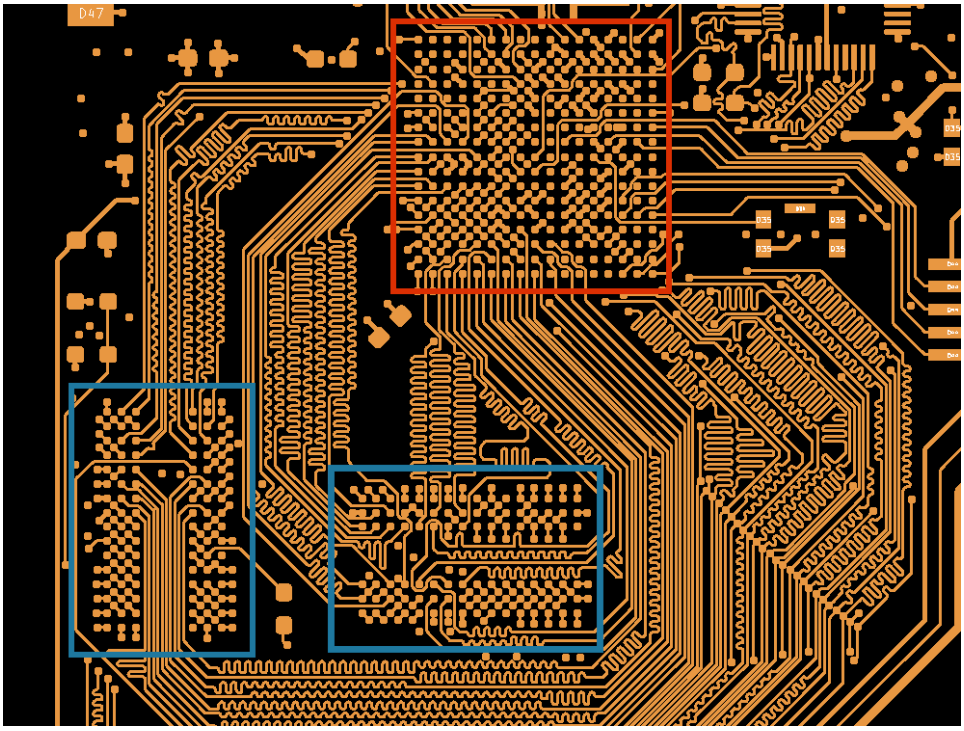


Figure 11: Meandering length matching of DDR RAM bus.

A less involved example of an application of the same technique is seen in figure 12. The red square indicates the location of the low complexity `ATSAMV71`, while the blue rectangle indicates the Flash memory module.

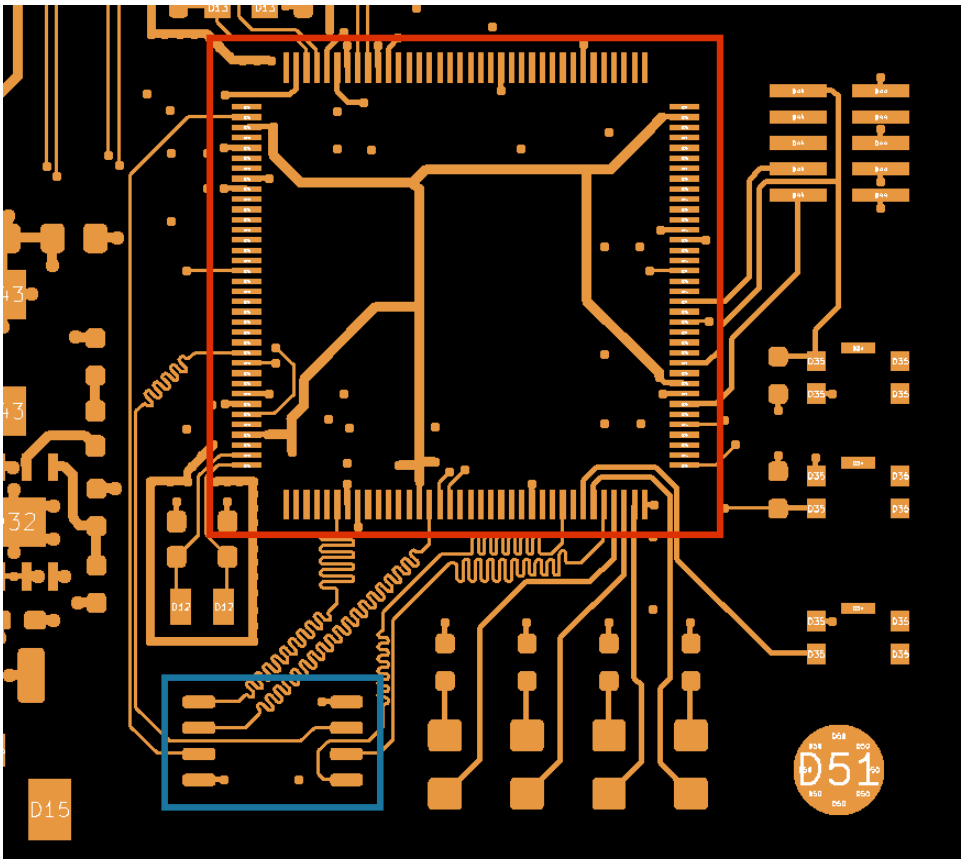


Figure 12: Meandering length matching of Flash data- and clock lines.

4.1.4 Decoupling

Decoupling is the practice of reducing the effects of inherent inductance from microstrip lines, thereby limiting the repercussions of events such as power demand spike transients. In a worst case scenario, such spikes may lead to a significant drop in voltage at the input pin of an IC, which may then misbehave, or sporadically reset. The most effective means of combating this issue from a PCB design point of view, is by adding low-inductance capacitors close affected IC pins.[38]

For a decoupling capacitor the be effective, inductive contributions from the interconnects between the IC pin and the capacitor must be kept to a minimum. This means limiting the distance from the IC pin to the capacitor legs as much as possible. Different schools of though exist regarding exactly how to achieve this, and on which side of a PCB each capacitor should be placed. For example, the `ATSAMV71` datasheet explicitly recommends against routing decoupling capacitors through circuit vias.[1]

Industry SI experts offer definitive and opposing advice, however. The reader is referred to for example (Johnson 2003)[18]. The reasoning given is that while vias themselves introduce inductance to some degree, they will overall reduce inductive effects with appropriate placement.

If vias are placed close together, they will effectively squish the inductive loop area, as seen through the cross section of the PCB. This alone presents a convincing argument to rethink advice from e.g. the `ATSAMV71` datasheet. Another contributing factor is the dedicated ground- and power planes found in most modern PCB applications; by replacing a microstrip wire with a via to a plane, a much wider path is available for the supply current (namely the whole plane), thus limiting the overall resulting inductance.[18]

Specifically, what this means for the layout of the Helmsman autopilot PCB may be seen in figure 13. To the left in this figure, we see a "traditional" microstrip line connection; this is what we have been avoiding in our layout process on the basis of the stated arguments. To the right in figure 13, we see connections from IC pins and capacitors alike, going immediately to the power- and ground planes. In many cases, we have also placed capacitors on the back side of the board, to further limit the distance between ICs and decoupling components.

For the most part, this strategy has been easy to abide by. Nonetheless, due to the physical limitations of the board manufacturer's capabilities, this approach could not always be satisfied around the `ATSAMA5D27` IC in the high complexity sub circuit. In these cases, inductive loops have inevitably

gotten somewhat bigger. We attempt to assess the repercussions of this in section 5, and give a brief discussion on how this aspect of the design may be improved upon with different manufacturing technologies in section 6.

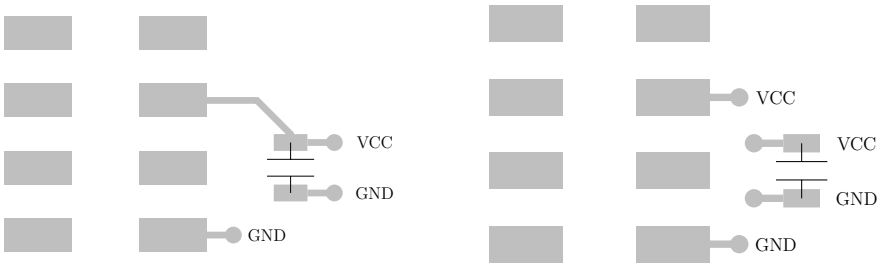


Figure 13: Decoupling strategies; microstrip line connection on the left, vias to dedicated planes on the right. Rectangles represent the IC connection footprints on the physical board.

4.1.5 Clock signal integrity preservation

Clock signals are a vital part on most digital circuitry. While clock signals may be generated from simple *RC* oscillators for applications that demand only modest precision, better clock signal regularities require components such as ceramic resonators or quartz crystal feedback circuits.[38]

Of these, quartz crystal feedback circuits offer the highest precision. One of the chief ways to derive a clock signal from a quartz crystal is to use an inverting amplifier on thermal noise, feed it through the crystal - which in this configuration may be viewed as a very narrow bandpass filter, and then through a Schmitt-trigger.[32]

When a frequency is needed that is higher than what one can reasonably generate from a quartz crystal alone, a common technique is to use a Phase-Locked Loop (PLL). A PLL used in this fashion will generate a multiple of the clock signal supplied by the quartz crystal, with the caveat being that the resulting signal will of lower regularity than its input.[32]

These physical realities contribute to make clock signals somewhat fragile, and care must be taken in the layout process of a quartz crystal to ensure a viable clock signal. In the design process of the Helmsman autopilot system, crystal layout was largely based on suggestions made in STMicroelectronics' `AN2867` application note.[31] Suggestions here are specifically given for their `STM8` and `STM32` series, but mostly applies to any quartz crystal containing circuit.

Central to the `AN2867` application note is an aspiration for isolating the oscillator circuit as much as possible from outside interference, which also has the additional benefit of keeping clock pulses from "escaping" and from affecting other circuitry. This amounts to limiting trace lengths, and using ground "guard traces" around each crystal.

Moreover, care is taken to not reference the ground plane from the other side of the PCB close to the crystals on the board. An example is given in figure 14. Here, the pads marked *D12* designate the placement of the crystal, while the orange enclosing line is a guard ring, tightly coupled to the ground plane with numerous vias shown in blue.

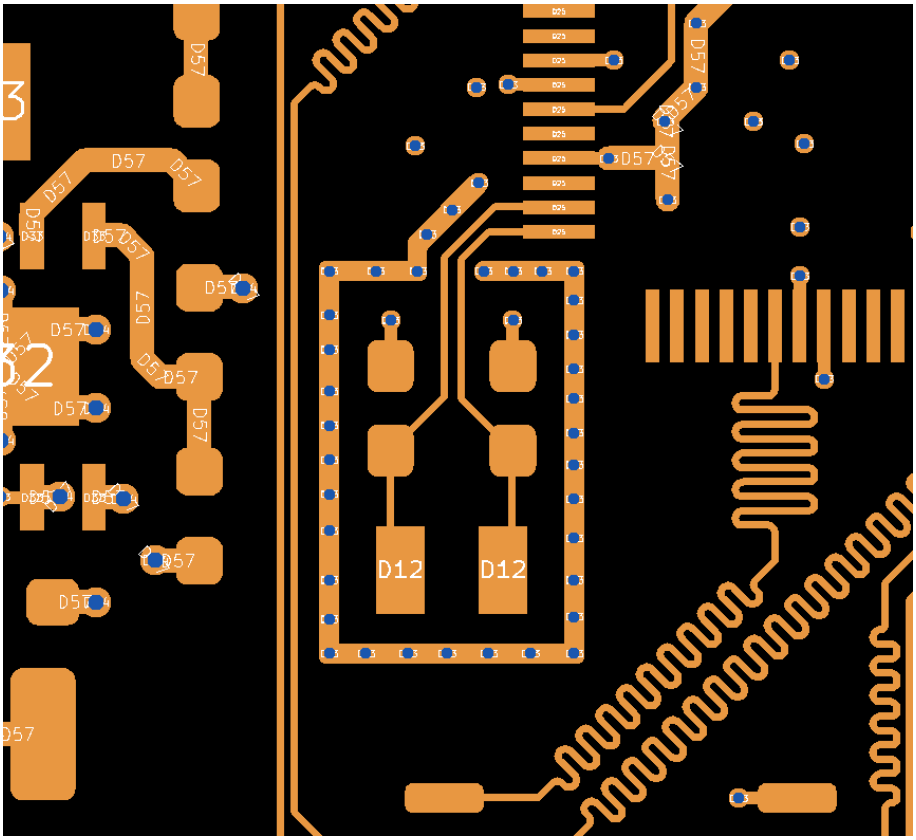


Figure 14: Example of crystal layout in low complexity sub circuit.

Finally, we mention that the Helmsman has separate low- and high frequency clocks, for a separation of *real time counting* and system clock signals. Both the high- and low complexity sub circuits also boast internal RC oscillators, that the controllers will automatically fall back on should the crystals fail. This is a feature of the `ATSAMA5D27` and `ATSAMV71` themselves, and thus not subject to dedicated design considerations from the author's side.

4.2 Software implementation

We begin our discussion with implementation details regarding the high complexity sub circuit, as these are, perhaps somewhat ironically, far less complex from a software point of view. The fallback chain is simple, as noted in section 3.5.1, and leans heavily on a feature rich operating system to do its work.

Next, from section 4.2.3 onward, we present software solutions for the low complexity sub circuit. These are more involved, and is also where most of the implementation effort went in terms of software.

4.2.1 High complexity sub circuit: Linux

The process of integrating Linux on the high complexity sub circuit is straight forward. The operating system image is already supplied by the manufacturer via a dedicated distribution website.[27]

Interfacing with other hardware and utilizing the `ATSAMA5D27`'s internal peripherals is made easy by using drivers supplied by the manufacturer. Concerns such as which drivers to load, and which peripherals to enable, are handled by a platform agnostic hardware identification language named *Open Firmware Device Tree*, most commonly referred to as "Devicetree" or simply "DT".[12]

In the Devicetree specification, individual hardware components may be identified with parameters such as their addresses in a memory mapped Input Output (IO) scheme, their interrupt graphs, DMA support, and their preferred programming models. If necessary, an end user may modify the Devicetree to add support for new hardware, or customize drivers that are being used. Yet, writing custom Linux drivers tend to be a lengthy process, and was not relevant for the work of this project.

Before the Devicetree is loaded and the operating system started, some custom program must initialize the resources of the `ATSAMA5D27`, and the operating system code itself must be read and copied to RAM. A program carrying out this operation is typically called a "bootloader". Under normal conditions, a simple fuse configuration may be set in the `ATSAMA5D27`'s non volatile memory, that will automatically load code from the platform's SD card and run this. However, for our purposes, we need the ability to detect whether an abnormal restart has occurred, as this is a signal that we should fall back on the custom firmware solution as described in section 3.5.1. This is simply done by first checking a reserved area in the chip's volatile memory for a flag we define. The flag is set on Linux boot, so if the flag is already present when we start the MPU, it implies that the previous run of Linux crashed without clearing the startup flag. To reset this flag, it is necessary to either manually clear it, or power cycle the high complexity sub circuit for long enough that the volatile memory be invalid.

This strategy amounts to an exceedingly simple custom bootloader, that only needs to enable the `ATSAMA5D27`'s power management controller, check the volatile boot flag, and then jump to either the SD card or the on-chip persistent memory accordingly. This bootloader was implemented in C, with manual memory mapping, and leveraging the "branch and execute" (`bx`) ARM instruction to swap code execution from the bootloader to the designated software solution. A thorough discussion on the mechanisms of

such techniques may be found in (Zhu 2018).[41]

Regarding the actual Linux environment, little extra code was added, due to time and scope constraints of the project. A simple program sending messages to the low complexity sub circuit over UART was used to verify that the operating system had successfully booted. Apart from that, only code to intentionally cause a crash was added to the Linux implementation, as this was used for fallback validation in section 5.

4.2.2 High complexity sub circuit: Custom firmware

As stated in section 3.5.1, the overarching design of the high complexity sub circuit fallback firmware follows a central doctrine of minimalism. During the work of the project, the simplified class structure illustrated in figure 15 was implemented in the C programming language. This structure is seen to be essentially identical to the proposed fallback architecture of figure 6, with two exceptions worth mentioning.

Firstly, figure 6 makes no mention of the block indicated as "`setup.h`" in figure 15. In actuality, this is really just a driver for the `ATSAMA5D27` Power Management Controller (PMC); responsible for enabling the correct clock signals internally. As such, it could have been lumped together with the "`drivers`" module. Due to the universal need for PMC setup before any other drivers, it was decided to implement it as a standalone entity in the final implementation.

Secondly, while figure 6 alludes to the addition of an observer, this was in fact not implemented in the high complexity sub circuit fallback firmware due to time- and scope limitations. During the project work, the observer block was simply replaced with "`stub.h`" in figure 15, which is a dummy implementation that passes sensor data unfiltered through to the "`pid.h`" block.

The two drivers, "`can.h`" and "`uart.h`" are present to simulate sensor- and actuator communication over CAN, and inter sub circuit communication over UART, respectively. Importantly, when the `can.h` driver activates, it also forces both CAN buses into strict CAN 2.0B mode (as opposed to CAN-FD). The reasoning is that the `ATSAMA5D27` implements a message format that does not conform to the latest CAN-FD specification, as noted in section 3.4.1.

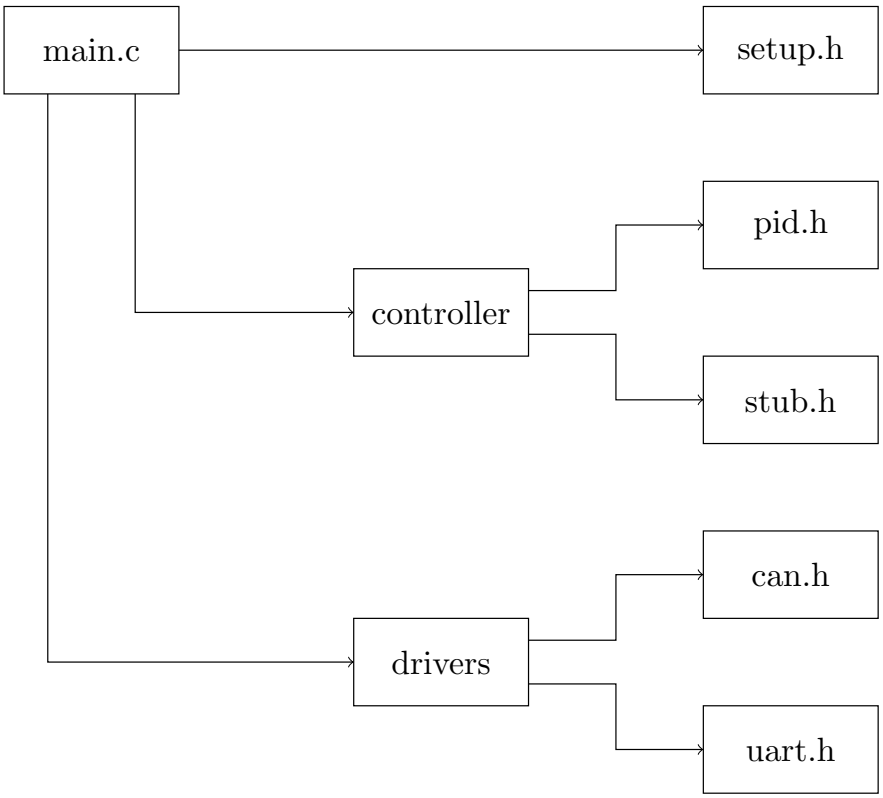


Figure 15: High complexity fallback firmware structure.

4.2.3 Low complexity sub circuit: FreeRTOS

The first step in the low complexity sub circuit's fallback chain is FreeRTOS, as discussed in section 3.5.2, and illustrated in figure 7. The implementation is straight forward, and does not warrant an in depth discussion.

At system start, a single task is statically created, designated as the system "Supervisor". It is this task's responsibility to react to exceptions caused by failing code, and activate fallback solutions. This is implemented by the top level task dynamically creating a single controller task; if this task fails, another task of a different controller implementation is created and added to the scheduler, while the failed one is removed.

In this implementation, only two tasks are specified; the same simple PID algorithm used in section 4.2.2, and a dummy "open loop" algorithm that only sends empty set point messages over CAN at equal intervals.

Considering the implementation is identical to that of section 4.2.2, nothing is gained in terms of design diversity - and having an open loop algorithm as a fallback solution is clearly questionable from a control engineering point of view. However, since the scope of this project is first and foremost to verify the robustness of the autopilot platform implementation, simple algorithms were chosen so focus could be concentrated on the fallback details over the control theory aspects.

4.2.4 Low complexity sub circuit: Firmware fallbacks

Significantly more development went into custom firmware solutions for the low complexity sub circuit than its primary FreeRTOS strategy. FreeRTOS remains a very well known, and widely used operating system, and from an implementation point of view it is not that interesting in the eyes of this work. The story is quite different regarding the custom firmware solutions, which will necessarily be of a more experimental nature. Hence, a more in depth discussion is presented on this topic.

The primary challenge of conditionally running several firmware versions is that of code separation. A simple approach would be to compile a single binary from a code base comprising all fallback solutions, and then selecting the branch to execute on startup. This would work, but requires the entire "fallback binary" to be recompiled, linked, and flashed every time a single fallback solution is modified. As such, this approach was quickly deemed to be difficult to scale efficiently. Another issue is that of physical code separation. Following this method, there are no guarantees that "firmware version 1" does not use routines from "firmware version 2" and so on, as all responsibility is left to the programmer.

To get around both these limitations, an approach leveraging the Flash memory IC introduced in section 3.4.2 was developed. Since the `ATSAMV71` has *execute in place* capabilities, code from the Flash may be run directly, without first shadowing to RAM. Hence, we may thus dedicate the whole internal persistent memory of the `ATSAMV71` to a combined bootloader and the FreeRTOS binary from section 4.2.3. The bootloader may then transfer execution to separate codes from Flash, should the FreeRTOS strategy fail.

To get us started, we present the `ATSAMV71` memory map in figure 16. Details such as memory mapped IO and reserved sections are not included. The various sections warrant a quick explanation to make our strategy clear.

The section called "Code" on the `ATSAMV71` (ranging from `0x00000000` to `0x20000000`) contains our simple bootloader and FreeRTOS. Following this, the region "Internal SRAM" contains 384 kB actual space, with a beginning at address `0x20c00000`. Nothing is located here when the IC starts up; it must be copied from somewhere else, such as a data section on the external Flash, or the internal Read Only Memory (ROM).

Of most interest to us is the 512 MB address space starting at `0x80000000` for our external Flash chip. Our chip only physically has 128 Mb of capacity, which is why the entire memory region is not used. We further segment this memory into five regions allowing for completely separate fallback firmware

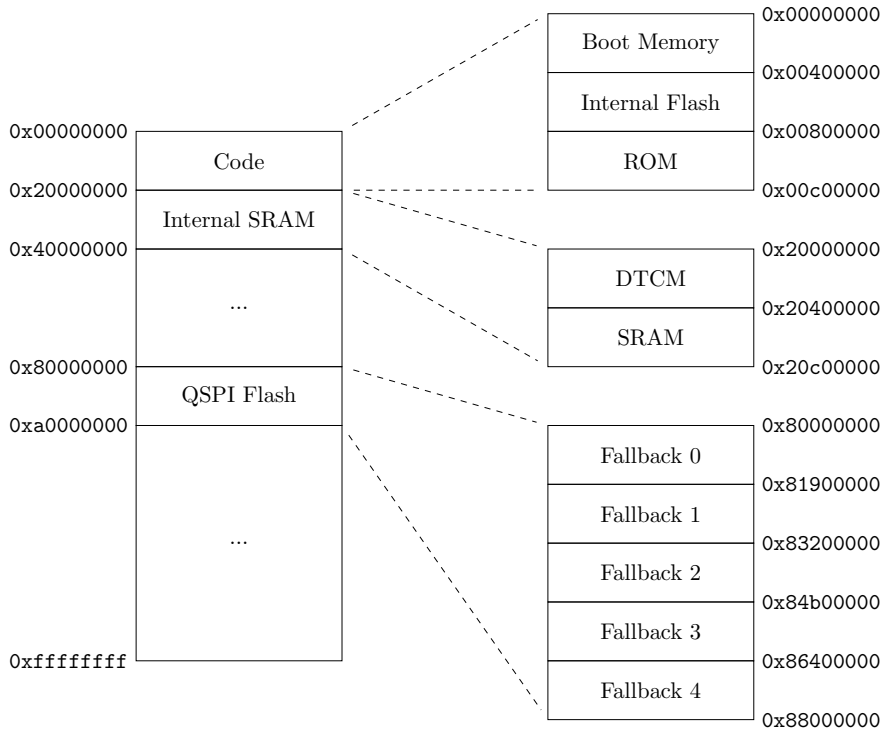


Figure 16: Simplified `ATSAMV71` memory map.

solutions. This is implemented as four regions à 25 Mb, and a single region à 28 Mb. This is done so the regions line up correctly with entire Flash memory pages.

To effectively use the proposed memory map illustrated in figure 16, it was necessary to write a custom linker script, and ensure that the compiler would place different functions and data elements in different sections. The latter is easily achieved in the `gcc` compiler via the `-ffunction-sections` and `-fdata-sections` flags. The former is more involved, and justifies a more in depth discussion.

Firstly, we specify the memory map as linker sections, as in listing 1. This informs the linker about the available regions our finished binary should consist of.

Next, it is necessary to ensure that each specific part of our fallback code lands in their designated areas. This will be partly the responsibility of our toolchain - namely compilation and binary merging, but also touch on the domain of our startup code; which in our case will be a small stub of C and assembly that will get executed before we reach the `main` function of our

```
MEMORY
{
  bootrtos   (rx)  : ORIGIN = 0x00400000, LENGTH = 0x00200000
  ram        (rwx) : ORIGIN = 0x20400000, LENGTH = 0x00060000
  fallback0  (rw)  : ORIGIN = 0x80000000, LENGTH = 0x01900000
  fallback1  (rw)  : ORIGIN = 0x81900000, LENGTH = 0x01900000
  fallback2  (rw)  : ORIGIN = 0x83200000, LENGTH = 0x01900000
  fallback3  (rw)  : ORIGIN = 0x84b00000, LENGTH = 0x01900000
  fallback4  (rw)  : ORIGIN = 0x86400000, LENGTH = 0x01c00000
}
```

Listing 1: Linker script memory regions for low complexity sub circuit.

bootloader. To achieve this, we must "back export" symbols from *link time*, back to *compile time*, which we accomplish by appending listing 2 to our linker script.

```
__bootrtos_start__ = ORIGIN(bootrtos);
__bootrtos_size__ = LENGTH(bootrtos);
__ram_start__ = ORIGIN(ram);
__ram_size__ = LENGTH(ram);
__fb0_start__ = ORIGIN(fallback0);
__fb0_size__ = LENGTH(fallback0);
__fb1_start__ = ORIGIN(fallback1);
__fb1_size__ = LENGTH(fallback1);
__fb2_start__ = ORIGIN(fallback2);
__fb2_size__ = LENGTH(fallback2);
__fb3_start__ = ORIGIN(fallback3);
__fb3_size__ = LENGTH(fallback3);
__fb4_start__ = ORIGIN(fallback4);
__fb4_size__ = LENGTH(fallback4);
```

Listing 2: Linker "back exporting" of link time symbols.

Listing 2 provides a foundation our startup code may build on, but it is still necessary to place compilation artifacts in their correct areas. This will be discussed presently. Building further on our linker script, we place the bootloader code in its correct location, as shown in listing 3.

```
STACK_SIZE = 0x2000;

SECTIONS
{
  .text :
  {
    KEEP(*(.vector*))
    *(.text*)
  } > bootrtos

  .bss (NOLOAD) :
  {
    *(.bss*)
    *(COMMON)
  } > ram

  .stack (NOLOAD) :
  {
    . = ALIGN(8);
    . = . + STACK_SIZE;
    . = ALIGN(8);
  } > ram

  .data :
  {
    *(.data*)
  } > ram AT > bootrtos
}
```

Listing 3: Linker section placement of bootloader code components.

Most of what is shown in listing 3 is standard in low level microcontroller applications.[41] Nonetheless, for completeness, we point out two details in particular, as these will differ from other architectures. Firstly, we see that we use an 8-byte alignment for the `.stack` section in our linker script. This might differ from microcontroller to microcontroller, but will be specified in each respective technical reference manual. In our case, we have followed the ARM Architecture Procedure Call Standard (AAPCS).[34]

Secondly, we see that the the `.data` segment has specified both a *storage region* and a *load region* via the linker `AT` command. This tells us that the `.data` segment should reside in non-volatile internal Flash at load, but be copied into RAM when we are running. This is significant, because we may use the same technique for our fallback firmware versions. One additional thing to note is that this technique represents a promise to the linker; we are still responsible for implementing the loading from internal Flash to RAM, as the linker itself has no idea of how to accomplish this. Incidentally, this is also where we must rely on our startup code.

Building further on the techniques of listing 3, we may employ the same tactics on fallback code, as demonstrated in listing 4 for fallback code 0. The procedure is identical for the other fallback firmware versions.

Both listings 3 and 4 present us with the obligation of ensuring that data loading happens before control is handed over to the relevant code. As stated, this will fall under the responsibility of our startup code, which is presented in listing 5. Certain universal parts of a startup code have been omitted for clarity; these include disabling interrupts at the beginning of the startup sequence, setting chip errata workarounds, and zeroing the `.bss` segment. Firstly, our startup code loads segments from internal memory into chip RAM, by use of pointers to our "back exported" linker symbols, and the C runtime function `memcpy`, which copies `size` number of bytes from internal Flash to RAM. Secondly, we exploit the assembly instruction "`bx`" to do an unconditional branch to an arbitrary point in our code - which happens to be the beginning of our bootloader. To achieve this, we resorted to the unorthodox method of leveraging the placements of arguments in the AAPCS; functions having four arguments or fewer will use the registers `r0` to `r3`, without creating a new stack frame. This method was selected to limit the amount of inline assembly needed, as the author is more familiar with C than the Thumb-2 ARM instruction set. The procedure is identical for our fallback firmware versions, replacing the `__bootrtos_start__` and `__bootrtos_size__` symbols with the symbols `__fb[0..4]_start__` and `__fb[0..4]_size__`.

```
SECTIONS
{
  .fb0_text :
  {
    KEEP(*(.fb0_vector*))
    *(.fb0_text*)
  } > fallback0

  .fb0_bss (NOLOAD) :
  {
    *(.fb0_bss*)
    *(COMMON)
  } > ram

  .fb0_stack (NOLOAD) :
  {
    . = ALIGN(8);
    . = . + STACK_SIZE;
    . = ALIGN(8);
  } > ram

  .fb0_data :
  {
    *(.fb0_data*)
  } > ram AT > fallback0
}
```

Listing 4: Linker section placement of fallback code 0 components.

```c
/* Pointers to source and destination of bootloader */
uint32_t * p_s = (uint32_t *) &__bootrtos_start__;
uint32_t * p_d = (uint32_t *) &__ram_start__;

/* Size of bootloader */
int size = (int) &__bootrtos_size__;

/* Copy byte for byte from internal Flash to RAM */
memcpy(p_d, p_s, size);

/* Load initial Stack Pointer and Program Counter */
uint32_t sp = p_d[0];
uint32_t pc = p_d[1];

/* Call static function to place SP and PC in ARM registers */
execute_branch(pc, sp);

/* ... */

/* Transfer execution from startup code to loaded code */
static void execute_branch(uint32_t pc, uint32_t sp){
  __asm("
    msr msp, r1 /* SP lives in register R1 per AAPCS */
    bx r0       /* PC lives in register R0 per AAPCS */
  ");
}
```

Listing 5: Startup code responsible for code loading and branch execution.

4.2.5 Low complexity sub circuit: Containment Domain

Should all the fallback solutions of sections 4.2.3 and 4.2.4 fail, our design from section 3.5.2 calls for a "Containment Domain", which sole area of responsibility will be to contain faults, and hopefully lead to a controlled mission abort.

As this is the last resort, it is imperative that the code it is built on be structured differently than all other firmware, and implemented by using different technologies, to maximize our chances for successfully leveraging the principles of design diversity.

To guide our attempt, inspiration was taken from the "Formula Student" organization "Revolve NTNU". According to the 2019 team's developer of the Vehicle Control Unit (VCU), the most elusive type of bug to root out for that system was the dreaded *segmentation fault*, where illegal memory is being accessed erroneously.[28]

While modern tools such as the `gcc` *address sanitizer* may be helpful in tracking down such faults, we must seek out a completely different approach all together to stay true to the tenets of design diversity. The decision was therefore made to implement the `Containment Domain` in Rust; a language that makes segmentation faults virtually impossible, if the language is used idiomatically.[22]

As of the time of writing, Rust for embedded systems such as the `ATSAMV71` has a "developer feeling" comparable to that of a desktop system. Some key differences include that one cannot necessarily expect all features of the standard library, and one also has to implement the ARM `Reset Handler` function through Rust, by using the so-called `unsafe` directive, for direct access to memory locations. As this is the most non-standard part of the `Containment Domain`, our Rust `Reset Handler` is presented in listing 6.

Briefly explained, the code presented in listing 6 works similarly to the code in listing 5, by accessing symbols defined by the linker, to copy necessary raw data into place from internal Flash to RAM, and then branching to the programs `main` function.

The other aspects of our Rust `Containment Domain` differ little from the strategy we have had for our other firmware versions, and do not demand an in depth presentation.

```rust
pub fn reset_handler() -> ! {
  extern "C" {
    // Use the same symbols from our Rust linker script
    static mut __sbss: u32;
    static mut __ebss: u32;
    static mut __sdata: u32;
    static mut __edata: u32;
    static __sidata: u32;
  }

  // Zero the .bss segment
  unsafe {
    // Start and end of .bss segment
    let mut sbss: *mut u32 = &mut __sbss;
    let ebss: *mut u32 = &mut __ebss;

    while sbss < ebss {
      core::ptr::write_volatile(
        sbss,
        core::mem::zeroed()
      );
      sbss = sbss.offset(1);
    }
  }

  // Load data
  unsafe {
    // Start and end of .data segment
    let mut sdata: *mut u32 = &mut __sdata;
    let edata: *mut u32 = &mut __edata;
    let mut sidata: *const u32 = &__sidata;

    while sdata < edata {
      core::ptr::write_volatile(
        sdata,
        core::ptr::read(sidata)
      );
      sdata = sdata.offset(1);
      sidata = sidata.offset(1);
    }
  }

  // Transfer execution to main Containment Domain
  main()
}
```


Listing 6: Rust based `Reset Handler` section of the `Containment Domain`.

5 Validation and Results

Continuing the split theme from before, we attempt to treat hardware- and software validation as separate topics to the degree that this distinction makes sense. Ultimately, however, the Helmsman autopilot must be seen as a complete system, to which we have dedicated section 5.3.

Regarding the Helmsman platform, hardware validation may broadly be divided into *signal integrity* and *power distribution*. The tests done must be classified as quite rudimentary, but have still provided some level of insight. Suggestions for future hardware validation tests are presented in section 6.

For software validation, most of the effort is focused on various forms of software linting and run time testing. Possible alternative future test venues are briefly outlined in section 6.

5.1 Hardware validation

Both signal integrity and power distribution are essentially concerned about some signal in the general sense arriving from a source to a destination, without being mangled too badly.

For the domain of signal integrity, this equates to a (typically) low voltage signal being allowed to travel through an interconnect without experiencing disruptive reflections, skews, distortions, or significant cross talk from other interconnects.

For power distribution, we must drive a supply voltage to the pins of some IC, and tolerate a specified amount of power demand spiking. Furthermore, we must normally also satisfy some specified bounds on supply ripple for reliable performance.

We are able to measure both of these aspects to some degree using an oscilloscope with a high impedance probe. In our work, we have used a Tektronix TBS 1052B, which is capable of 1 giga-samples per second, and specified for signal bandwidths of up to 50 MHz.

Additionally, we have utilized the electric field solver in the Elmer Finite Element Method (FEM) package, which is distributed freely by the Finnish CSC; IT Center for Science.[35] This was done in an effort to compliment our oscilloscope results, considering its rated signal bandwidth is quite low for our application.

5.1.1 Oscilloscope: Power distribution

Power distribution tests were carried out by sampling the supply pins of each IC on the Helmsman PCB. For the `ATSAMA5D27` and `ATSAMV71`, this was done both under nominal load, and specially constructed code that turned on all internal peripherals and busied the CPU for the duration of the test.

In all cases, our tests seem to suggest that we are within the respective specified maximum ripple limitations. Nonetheless, the author would like to attach a reasonably large caveat to this claim; in the cases of the `ATSAMA5D27` and its two RAM ICs, these tests might as well be discarded. The reason is that all of these three chips are BGAs, and were thus difficult to test on. The only extent that this was possible, was in fact by carefully touching untented vias on the PCB with the oscilloscope probe. In some cases, these vias were comparatively far away from the IC itself, which might add to a larger measurement uncertainty.

5.1.2 Oscilloscope: Signal integrity

The maximum sampling frequency of the oscilloscope used in the tests placed significant limitations on what signals could be measured. One simple test that was carried out, was to indirectly measure the PCB's clock signals, by programming the `ATSAMA5D27` and `ATSAMV71` to directly output their system clocks on a given pin. This clock signal was then measured, and compared to the prescribed frequencies of 16 MHz for the high frequency oscillator, and 32.768 kHz for the real time clock. For both the high frequency clocks, deviations were typically within ~ 40 ppm, while for the low frequency real time clock the deviation was typically within ~ 60 ppm.

Again, these results should be taken critically, as for example (Nakagawa 2009) recommends using an oscilloscope capable of measuring at least five times the expected clock frequency.[15] Our TBS 1052B is well on the low side of this.

Additionally, we were able to measure the CAN buses under regular load conditions. Unsurprisingly, at the relatively low frequencies of 1 MHz in CAN mode and 4 MHz in CAN-FD mode, no discernible signal degradation could be observed. Under test conditions, these signals always stayed within recommendations given by the `ISO 11898-1:2015` CAN standard.[30]

5.1.3 Elmer FEM: Signal integrity

In an attempt to validate our microstrip line routing strategies outlined in sections 4.1.2 and 4.1.3 we employed basic field solver routines from the Elmer FEM package.

Figures 17 and 18 both show similar simulation setups. Here, we see a PCB in top view; the black line extending across the board is on the top layer, and carries a 10 MHz signal from the top via to the bottom via. The gray background represents a ground plane on the bottom layer of the PCB.

Relative electric fields strengths are shown by field concentrations ranging from red (being the most concentrated), to violet, blue, cyan, green, yellow, and finally orange (being the least concentrated).

While it was difficult to get concrete results in terms of absolute expected fields strengths, we are able to see their relative spreading. In figure 17, we see that almost all of the signal return current is carried directly under the microstrip line. We do observe some amount of fringing, but even at a PCB thickness of 1.6 mm, this is strongly bounded.

In figure 18, we observe a radically different scenario. Return current is forced to follow the edge of the ground plane slit, and as a result of this sharp change in impedance, we detect the resulting electric field virtually everywhere on the PCB, including to some degree in the dielectric space making up the slit itself.

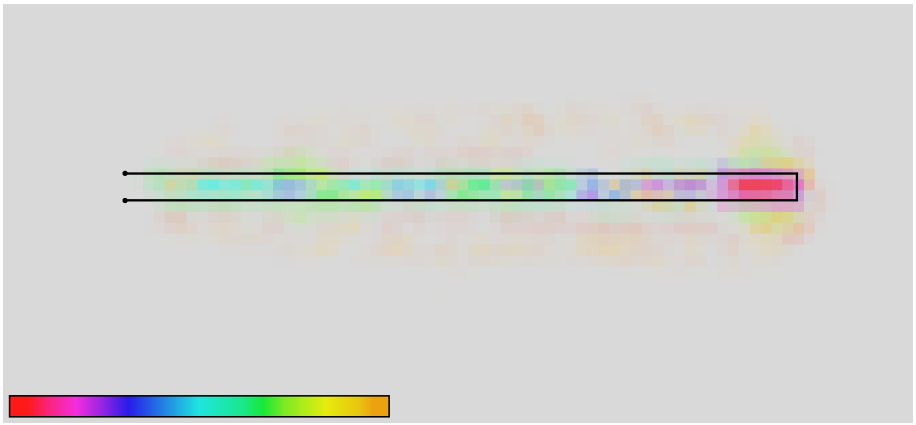


Figure 17: Relative electric field strengths for current return path over an unbroken ground plane at a signal frequency of 10 MHz.

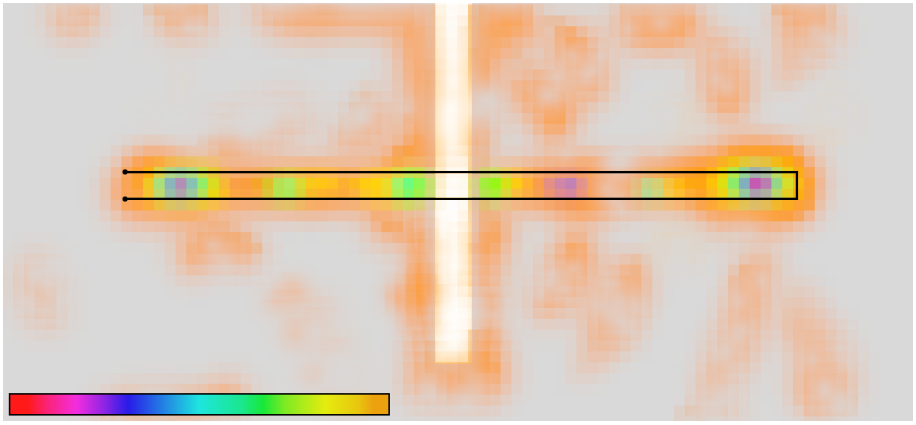


Figure 18: Relative electric field strengths for current return path over a ground plane with a 5 mm wide slit covering most of the height of the board. The simulated signal frequency is 10 MHz.

5.2 Software validation

For software validation, a handful of different tools were used. Included in these ale the Clang Static Analyzer for linting and checking of C code, the `gcc` compiler toolchain itself, Rust's "Clippy" static checker, the tool Valgrind for memory correctness, and custom written unit tests for critical functions and common functionality.

5.2.1 Linting for C and Rust

Linting, the process of detecting and flagging known problematic language constructs or suspicious code, is a similar process for both C and Rust. The difference mainly being that Rust's "Clippy" is able to uncover marginally more common errors than the Clang Static Analyzer, due to the stricter language form.

Linting was always run for every file modification of any source code during the work with the project. Compiler "*pedantic*" modes were enabled, and flags turning *warnings* into *errors* were used. This ensured that dubious code would be harder to run.

Since no code was developed without these tools, it is difficult to estimate numbers on how many potential bugs were stopped using this method, based on this work alone. However, we may compare the code developed here with empirical studies on bug characteristics found in contemporary open source projects, for example (Li 2006) and (Catolino 2019).[3][16]

This yields some comparative metric, but as these studies focus on widely different software domains than the one presented in our work, we do not attempt to make a one to one comparison. Types of bugs that we ignore from (Li 2006) and (Catolino 2019) include graphical issues, some network code, and most security related bugs.

To get a more representative bug characteristic comparison, we thus also compared our code to static analyses of code from repositories maintained by Revolve NTNU.

Against both comparison groups, our code had significantly fewer linting errors, and presented no detectable serious issues like memory faults from a static analysis point of view.

5.2.2 Address sanitizer and Valgrind

A static analysis can usually only hope to uncover a portion of potential faults. Linting results suggested that our code be unlikely to contain major

memory corrupting faults, but to investigate this more thoroughly, both an address sanitizer and Valgrind were used.

An address sanitizer compiles in extra error checking code that enters into the finished binary, which may then be be run on an emulator such as QEMU (qemu.org). The completed binary will then perform a host of verifications, such as out-of-bounds checking, double free detection, memory leaks, and use-after-free violations.

Valgrind works similarly, but instead of adding code into the binary itself, it implements a whole simulated target processor, where the original binary is run without modifications.

Both tools supported the previous assertion that our code does not contain common errors observed by (Li 2006), such as NULL pointer dereferences; or the segmentation faults that were common in subsets of code developed by Revolve NTNU.[28]

5.2.3 Unit testing

Unit testing is the process of writing constructed cases for specific functions, methods, or related discrete pieces of code under set circumstances. In our work, unit testing was used extensively to validate correctness of functions in drivers, communication interfaces, and control code.

The method of implementation differs between our C and Rust code. In Rust, unit testing is a natural extension of the language framework, and may be introduced by decorating code with the `#[test]` attribute. Next, assertion macros may be used to verify the correctness of some condition.

In C, there is no accepted "one way" of writing unit tests, and the author resorted to manually implementing assertions that would be conditionally compiled in- or left out, depending on compiler flags.

Unit testing proved valuable in the work of this project. Not only was it able to further reaffirm the case for our code likely being memory fault free, it was also able to greatly aid in the development of control code. One example of this, is a set of linear algebra routines that were ported from some of the author's previous work (Austreng 2020).[4] Under pathological conditions, these routines would produce NaNs (the floating point "not a number"). Unit testing aided in isolating the culprit, and validated that the adjusted implementation worked for the tested scenarios.

5.3 Helmsman validation

The Helmsman autopilot platform would be useless without the capability of functioning as an integrated whole system. As outlined in section 3, we outlined that an important part of the application as a whole is to act as the central controlling unit in a HIL setup.

To facilitate this, the on board USB interfaces were used for autopilot to host communication. A somewhat simplified version of the simulator due to (Oland 2018) was used during testing.[13] Simplifications comprised "hard coding" their noise free "perfect knowledge" version of simulation. This was done in an effort to restrict scope, and allow more rigorous testing on the platform concept itself, as explained in section 1. The HIL tests carried out consisted of alternating between keeping steady heading and altitude, and performing 180° coordinated turns. Disturbances were kept, but were limited to one tenth of their original magnitudes.

The Helmsman system performed satisfactorily under nominal conditions. To validate the platform's fail-over capabilities, which is the heart of this work, controller code was forced to catastrophically crash from user input. This consistently triggered the fail-over routines, without loss of autopilot availability. In fact, the redundant takeover proved to happen too fast to be discernible from the MATLAB/Simulink environment the simulator was running in.

In an attempt to agitate noticeable deterioration of performance, a small custom interface driver was written to speak with the Helmsman from the host computer. This driver would register as a file under the file system, and be read and written by MATLAB/Simulink. The driver would sporadically drop packets needed for the HIL simulation. When this happened frequently enough, the Helmsman would register the fault, and switch to the next fall-back solution. MATLAB/Simulink, however, remained blissfully unaware of this in all tests, due to its significantly lower execution speed.

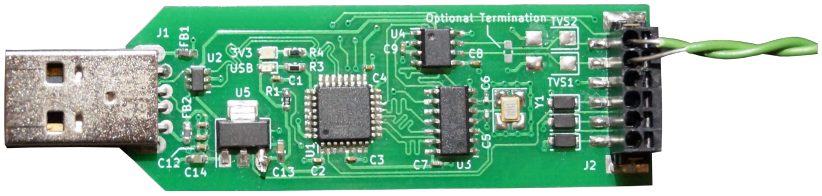


Figure 19: USB to CAN-FD bridge.

Lastly, to simulate Helmsman performance under conditions where its HIL interface would not consume processor time, a simple auxiliary CAN-FD to USB bridge was developed. This bridge may be seen in figure 19. The twisted pair going off to the right side are the CAN-FD lines, while the USB section plugs directly into a computer. Power is provided via USB. The development of this bridge is not central to the Helmsman circuit, so we dedicate no extra discussion regarding it.

The use of this bridge allowed the Helmsman USB interfaces to remain off, instead relying on the always-on CAN-FD buses for host communication. During this stage, additional sensor noise was also injected on the CAN-FD buses. No change in performance could be detected, indicating that the Helmsman has superfluous computational resources even when carrying out subordinated tasks such as HIL communication.

6 Conclusions and Future Work

In this work, we have presented the process of developing a redundant and fault tolerant modular autopilot platform designated as the "Helmsman". Validation efforts suggest that our platform meets or exceeds all relevant development criteria that were set forth.

Hardware In Loop (HIL) testing has shown promise in further use and potential development of the autopilot system, with a sizable and mostly unused capacity for integrating additional features, even those demanding extensive computational resources.

Many topics have been left untouched, and potential avenues for further technical and academic inquiry have uncovered in great numbers. Some of the ones that present themselves as immediately of interest to the author are outlined below.

6.1 Future work in hardware

Based on observations from section 5, we identify several areas that may be improved upon. Firstly, as noted during the development process and testing alike, it was difficult to verify power supply integrity for the `ATSAMA5D27` and its two RAM ICs. This may be addressed in future revisions by selecting decoupling capacitors or smaller form factors and opting for a manufacturer allowing thinner microstrip lines and smaller vias. This would significantly ease the task of adding decoupling to the board's BGA components. One might also opt for the use of so-called "blind vias" - namely vias that do not penetrate the PCB entirely, but is only exposed on one side. This is a more expensive solution to manufacture, but would greatly increase routability.

For easier and better design validation, the board may also be produced at a facility providing x-ray inspection of the solder connections. This is among the only viable options for proper inspection of BGA connections.

An oscilloscope with a higher signal bandwidth would also greatly aid the trustworthiness of signal integrity results collected in section 5.1.

A board revision that is not constructed around components that happen to be available could also benefit the overall design. Particularly, this applies to the `ATSAMA5D27`, `ATSAMV71`, and the CAN-FD transceivers. None of these components were ever intended as the first choice for their respective uses, but the state of the industry (as outlined in section 2) forced some of the development to take a perhaps sub optimal direction in certain respects.

In section 3.3, we pointed out that it was sadly not feasible to produce a full TMR 2-out-of-3 voting system due to the production expenses this would incur. However, for even higher availability requirements, benefits might outweigh the costs.

Lastly regarding the hardware of the Helmsman itself, striving for a higher level of component integration, and an ability to more strictly meet real time requirements, basing future designs around Field Programmable Gate Array (FPGA) technologies presents an attractive area of exploration.

Apart from the Helmsman hardware, testing the autopilot system in an actual UAV application is the only way of obtaining irrefutable empirical validation for our design. Hence, such an endeavor would be appealing for future work.

6.2 Future work in software

We identify vast potential for software advances in the Helmsman autopilot. Firstly, the potential power of running high-level control routines on the embedded Linux platform were never explored deeply. Furthermore, in investing more development time for this part of the high complexity sub circuit, that fallback chain might be eliminated entirely; leaving fail-over capabilities to the low complexity sub circuit.

Secondly, while on the topic of embedded operating systems, it was seen early on that the FreeRTOS solution is needlessly complex. There is little reason to create tasks *dynamically*, if a single control task will be running at any given time. This warrants an exploration into an implementation of reduced scope, perhaps leveraging a set of statically created tasks for each of the controller- and observer software components.

Common to both the high- and low complexity sub circuits, we see some tendency to over-zealous fallback solution implementation. An example of this is the low complexity sub circuit's "`Containment Domain`". This could likely be combined with a single custom firmware fallback, provided the RTOS implementation is given more sustained development attention.

If one were to strip some fail-over capabilities, this could also open up the possibility of using the high complexity sub circuit to for example carry out model predictive control on a large time horizon, while the low complexity sub circuit would handle direct control.

Apart from software running on the Helmsman directly, we also see a promising avenue for more extended use of simulation software. For example, Elmer FEM was only superficially used in the validation work of this project, on a somewhat constructed scenario. If would be of interest to expand its use into other aspects of the circuit, such as thermal dissipation from switching mode power supplies. A more deliberate use of simulation techniques would moreover be imperative if an FPGA were to be introduced to the Helmsman platform.

Finally, we acknowledge that testing has benefited the project work in all matters regarding software. While unit testing has been the chief focus in this pursuit, it is by no means the only form of testing could have been explored. In the future, it might hence be of interest to explore for example property based testing solutions, or more rigorous integration testing.

In closing: Personal comments

In the author's view, a matte black PCB is about six times cooler than a standard green PCB. When an *electron nickel immersion gold* plating is applied to the exposed copper on the circuit, instead of a standard tin based hot air solder leveling, the coolness factor increases to about eight times its original value.

While the author wants to stay true to his Scandinavian roots in embracing the central doctrine that *function* must precede *form*, he would also like to add that there is nothing wrong with reaching for better form when perfect function has been achieved.

In light of this, the author would like to personally, and heartily, encourage the use of matte black PCBs in areas where consistent operation has already been secured by previous revisions.

In the author's humble opinion, this serves as a signature that the developer is ready to take complete ownership over- and pride in the finished solution. PCB design is an arduous and taxing effort, fraught with contradicting and perilous advice from schools of old. A matte black signature PCB hence serves to embolden the beneficiaries of an impeccably designed circuit, and foster courage and a stoic attitude in the generations of engineers to come after us.

The importance of matte black,

Kolbjørn Austreng

References

[1] *32-bit ARM Cortex-M7 MCUs with FPU, Audio and Graphics Interfaces, High-Speed USB, Ethernet, and Advanced Analog*. SAM E70 S70 V70 V71 Family. Microchip. Dec. 2020.

[2] et al. A.S. Nascimento. "An Empirical Study on Design Diversity of Functionally Equivalent Web Services". In: *2012 Seventh International Conference on Availability* (2012), pp. 236–241.

[3] Zhenmin Li et al. "Have things changed now?: an empirical study of bug characteristics in modern open source software". In: *Proceedings of the 1st workshop on Architectural and system support for improving software dependability* (2006), pp. 25–33.

[4] Kolbjørn Austreng. "A novel web based framework for the digitalization of teaching materials at Engineering Cybernetics in response to the COVID-19 pandemic". NTNU, 2020.

[5] Lorenzo Strigini Bev Littlewood Peter Popov. "Modeling software design diversity: a review". In: *ACM Computing Surveys* 33 (2001).

[6] Eric Bogatin. *A Simple Demonstration of Where Return Current Flows*. URL: `https://www.signalintegrityjournal.com/articles/1771-a-simple-demonstration-of-where-return-current-flows` (visited on 06/17/2021).

[7] David K. Cheng. *Field and Wave Elecromagnetics, 2nd Edition*. Pearson, 2014. ISBN: 978-1-29202-656-5.

[8] International Elecrotechnical Commision. *Functional safety of electrical/electronic/programmable electronic safety-related systems*. IEC. 2010.

[9] International Elecrotechnical Commision. *Industrial communication networks - High availability automation networks*. IEC. 2016.

[10] et al. Dave E. Eckhardt Jr. "An Experimental Evalutaion of Software Redundancy As a Strategy for Improving Reliability". In: *NASA Technical Memorandum* (1990).

[11] *Design Guide for Two DDR3-1066 UDIMM Systems*. TN-41-08. Rev. B. Micron. Jan. 2011.

[12] *Devicetree Specification*. Release v0.3. devicetree.org Technical Steering Committee. Feb. 2020.

[13] et al. Espen Oland. "A Comparative Study of Different Control Structures for Flight Control with New Results". In: *IEEE Transactions on Control Systems Technology* 28 (2 2018), pp. 291–305.

[14] FreeRTOS. *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extension*. URL: https://www.freertos.org/ (visited on 06/17/2021).

[15] *Frequency Deviation of Quartz Crystal Units in Oscillator Circuits*. NK-TIT-03 Rev. A. Nakagawa Electronics Limited. May 2009.

[16] et al. Gemma Catoline. "Not all bugs are the same: Understanding, characterizing, and classifying bug types". In: *Journal of Systems and Software* (2019), pp. 165–181.

[17] *Hardware and Layout Design Considerations for DDR Memory Interfaces*. AN2582. Rev. 6. Freescale Semiconductor. Apr. 2007.

[18] Martin Graham Howard Johnson. *High Speed Signal Propagation*. Prentice-Hall, 2003. ISBN: 978-0130844088.

[19] Demetrois Pogkas Ian King Debby Wu. "How a Chip Shortage Snarled Everything From Phones to Cars". In: *Bloomberg* (Mar. 29, 2021). URL: https://www.bloomberg.com/graphics/2021-semiconductors-chips-shortage/ (visited on 06/15/2021).

[20] et al. Imanol Allende. "Towards Linux based safety systems - A statistical approach for software execution path coverage". In: *Journal of Systems Architecture* 116 (2021).

[21] *Instrumenteringssystemer*. 6th edition. Institutt for Teknisk Kybernetikk. Jan. 2019.

[22] Jason Orendorff Jim Blandy. *Programming Rust: Fast, Safe Systems Development*. O'Reilly, 2018. ISBN: 978-1-491-92728-1.

[23] JLCPCB. *Controlled Impedance PCB Layer Stackup*. URL: https://cart.jlcpcb.com/impedance (visited on 06/17/2021).

[24] Nancy G. Leveson. *Safeware: System safety and computers. A guide to preventing accidents and losses caused by technology*. Addison-Wesley, 1995. ISBN: 0-201-11972-2.

[25] Hans Hopman Lode Huijgens Arthur Vrijdag. "Hardware in the loop experiments with ship propulsion systems in the towing tank: Scale effects, corrections and demonstration". In: *Ocean Engineering* 226 (2021).

[26] Steve McConnell. *Code Complete 2*. Microsoft Press, 2004. ISBN: 978-0735619678.

[27] Microchip. *Linux and Open Source for AT91 Microchip Microprocessors*. URL: https://www.linux4sam.org/bin/view/Linux4SAM (visited on 06/19/2021).

[28] Nikolai Zaitsev Nymo. "Implementation of an Event-Driven Architecture for Multiprocessor Embedded Systems". MA thesis. NTNU, 2019.

[29] Dave Opsahl. "The Semiconductor Shortage: Are Car Companies Now Consumer Electronic Companies?" In: *Forbes* (Apr. 14, 2021). URL: https://www.forbes.com/sites/forbesbusinesscouncil/2021/04/14/the-semiconductor-shortage-are-car-companies-now-consumer-electronic-companies/ (visited on 06/15/2021).

[30] International Standards Organization. *Road vehicles - Controller area network (CAN)*. ISO. 2015.

[31] *Oscillator design guide for STM8AF/AL/S, STM32 MCUs and MPUs*. AN2867. Rev. 13. STMicroelectronics. Dec. 2020.

[32] Winfield Hill Paul Horowitz. *The Art of Electronics*. Cambridge University Press, 2015. ISBN: 978-0-521-80926-9.

[33] Gard Paulsen. *Alltid Rabiat*. Fagbokforlaget, 2019. ISBN: 978-82-450-2546-0.

[34] *Procedure Call Standard for the Arm Architecture*. Release 2020Q2. Arm Ltd. June 2020.

[35] CSC - IT Center for Science. *Elmer Finite Element Method*. URL: https://www.csc.fi/web/elmer (visited on 06/28/2021).

[36] Peter Bauer Tamas Grof. "Voting-based Fault Detection for Aircraft Position Measurements with Dissimilar Observations". In: *IFAC-PapersOnLine* 53 (2020), pp. 14724–14729.

[37] *Ultra-Low-Power Arm® Cortex®-A5 Core-Based MPU, 500MHz, Graphics Interface, Ethernet 10/100, CAN, USB, PCI 5.0 Pre-Certified*. SAMA5D2 Series. Microchip. Mar. 2021.

[38] Peter Wilson. *The Circuit Designer's Companion*. Newnes, 2018. ISBN: 978-0-08-101764-7.

[39] et al. Xunhua Dai. "RFlySim: Automatic test platform for UAV autopilot systems with FPGA-based hardware-in-the-loop simulations". In: *Aerospace Science and Technology* 114 (2021).

[40] Y.C. Yeh. "Design Considerations in Boeing 777 Fly-By-Wire Computers". In: *Proceedings Third IEEE International High-Assurance Engineering Symposium* (1998), pp. 64–72.

[41] Yifeng Zhu. *Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, 3rd edition*. E-Man Press LLC, 2018. ISBN: 978-0-9826926-6-0.

A Bill of Materials

For ease of future revisions and reproducibility, we supply the following BOM containing critical components. Each entry is the exact part number used by the Digikey component supplier, where all parts were ordered.

Common components such as chip resistors and ceramic capacitors have been left out, to avoid unnecessary cluttering. For such components, the 0603 form factor (1608 metric) was used. Capacitors were selected from the X5R series, with the exception of capacitors used for clock circuitry, where the C0G series was employed.

1253-1698-1-ND	ATSAMV71Q21B-AAB-ND
1092-1228-ND	WM7623CT-ND
296-48677-1-ND	ATSAMA5D27C-CU-ND
SMAJ7.0CA-FDICT-ND	SER4081CT-ND
A135698-ND	AP2111H-3.3TRG1DICT-ND
256-W632GU6NB-12CT-ND	732-4988-1-ND
HR1972CT-ND	1175-1735-ND
KSZ8041TL-CT-ND	497-4492-1-ND
1278-1011-ND	BD9E302EFJ-E2CT-ND
428-4121-1-ND	WM1723-ND
732-12918-ND	WM1142CT-ND
150-MCP16502TAA-E/S8BCT-ND	