2014

Design of a Small Form-Factor Flight Control System

Garrett Ward
Virginia Commonwealth University

Design of a Small Form-Factor Flight Control System

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

by

Garrett L. Ward

Director: Dr. Robert H. Klenke
Associate Professor of Electrical & Computer Engineering

Virginia Commonwealth University
Richmond, Virginia
May 2014

Acknowledgments

I would first like to thank my mother for always making sure my education was a priority. I certainly would not have made it this far without her love and support. I would like to thank my friends George, Tara, and Joel for keeping me sane during graduate school; and my friends Sarah, Daniel, Ed, and Andrew for giving me a place to escape to when I needed it. I would like to thank my advisor, Dr. Robert H. Klenke, for his invaluable advice and support during this project. I would also like to thank my committee members, Dr. Wei Zhang and Dr. James Ames, for their advice. Finally, I would like to thank my fellow graduate students: Tom, Tim, Matt, Siva, and Joel for keeping our lab a fun and enjoyable place to work.

Contents

List of Figures

List of Tables

List of Abbreviations

AF	Alternate Functions
API	Application Programming Interface
ATOL	Automatic Take-off and Landing
CMSIS	Cortex Microcontroller Software Interface Standard
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CSOIS	Center for Self-Organizing and Intelligent Systems
CTS	Clear to Send
DFU	Device Firmware Upgrade
DGPS	Differential Global Positioning System
DMA	Direct Memory Access
DoF	Degree of Freedom
DRDY	Data Ready
DSP	Digital Signal Processor
EDK	Embedded Design Kit
ENAC	École nationale de l'aviation civile
FCM	Flight Control Module
FCS	Flight Control System
FDT	Flattened Device Tree
FOSS	Free and Open Source
FPGA	Field-Programmable Gate Array

FPU	Floating Point Unit
FSMC	Flexible Static Memory Controller
GCC	GNU Compiler Collection
GCS	Ground Control Station
GDB	GNU Debugger
GPIO	General Purpose Input/Output
GPS	Global Positioning System
GUID	Globally Unique Identifier
HILS	Hardware-in-the-Loop
HSE	High-Speed External
HSI	High-Speed Internal
I2C	Inter-Integrated Circuit
ICM	Instrumentation Control Module
IDE	Integrated Development Environment
ISR	Interrupt Service Routine
JAUS	Joint Architecture for Unmanned Systems
LSE	Low-Speed External
LSI	Low-Speed Internal
LUT	Look-up Table
M2M	Module-to-Module
NMEA	National Marine Electronics Association
OCP	Open Control Platform
OTG	On-the-go
PCB	Printed Circuit Board
PID	Proportional-Integral-Derivative
PLL	Phase-Lock Loop
POSIX	Portable Operating System Interface

PPM	Pulse Position Modulation
RCC	Reset and Clock Control
RFCSA	Reconfigurable Flight Control System Architecture
RTC	Real-Time Clock
RTS	Ready to Send
RX	receive
SOA	Service Oriented Architecture
SoC	System on a Chip
SPI	Serial Peripheral Bus
SVIL	Standard Vehicle Interface Library
SWD	Serial Wire Debugging
SYSCFG	System Configuration
TX	transmit
UART	Universal Asynchronous Receiver Transmitter
UAV	Unmanned Aerial Vehicle
UVI	Unified Vehicle Interface
VACS	VCU Aerial Communications Standard
VCU	Virginia Commonwealth University

List of Source Code

Abstract

DESIGN OF A SMALL FORM-FACTOR FLIGHT CONTROL SYSTEM

Garrett L. Ward

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

Virginia Commonwealth University, 2014

Director: Dr. Robert H. Klenke, Associate Professor of Electrical & Computer Engineering

This work outlines a design for a small form-factor flight control system designed to fly in a wide variety of airframes. The system was designed with future expansion in mind while providing a complete, all-in-one solution to meet present needs. This system as presented meets most needs while remaining relatively low cost. It has a completely integrated IMU solution as well as on-board GPS. It is capable of basic waypoint navigation. This solution was testing using software and hardware-in-the-loop simulation which proved its functionality.

Chapter 1: Introduction

1.1 Overview

Unmanned systems continue to grow in popularity as technology becomes more prevalent. Between military applications, recent surges in civilian law enforcement use, and continued research in academia, Unmanned Aerial Vehicle (UAV)s have become more and more a part of everyday life. They offer several key advantages over human-piloted aircraft, including the ability to use much smaller airframes and the removal of a potential source of error in a human pilot. Further, multiple autonomous UAVs can be operated by a single human operator, reducing personnel overhead. The most important aspect of any UAV system is the autopilot, commonly referred to as the Flight Control System (FCS), which is responsible for controlling the aircraft. As UAVs become more ubiquitous, there is an increasing desire for an integrated, reconfigurable platform which is capable of flying a multitude of airframes and mission types with ease.

1.2 Motivation

There are numerous autopilot systems available, ranging from Free and Open Source (FOSS) products, to products developed by University research labs, and even commercial products. However, of the systems which are available, most are prohibitively expensive for research use. Additionally, most of the available systems require a suite of external sensors to provide crucial state information to the autopilot. These sensors add additional power, space, and thermal burdens on the design of the autopilot system as well as potentially limiting the airframes which the system can be used in.

The previous generations of autopilot systems developed at the Virginia Commonwealth University (VCU) UAV lab suffer from several shortcomings which limit their future usefulness in the ongoing research there. Two systems are currently in use: one which was designed first and foremost for research applications, which offers high levels of configurability due to the integration of a Field-Programmable Gate Array (FPGA); and the other, which was designed for low-cost, small form-factor applications. The research system is fairly powerful, but also very complicated. Furthermore, parts of that system were not documented at all, which led to a situation where crucial elements of the software could not be modified without risking unintended breakage in the rest of the software. While the system functioned well, it was effectively frozen in its feature set, which was a limiting factor as research at the VCU UAV Lab moved in a different direction than was foreseen when the system was designed.

The other system used in the VCU UAV Lab was intended as a low-cost platform for small airframes. While it did not suffer from a lack of maintainability as the research platform did–indeed, the software was updated as a direct result of the work performed here–it lacked in several other areas. At the time of its development, these features were deemed too expensive, either monetarily or computationally. As neither of the systems used in the VCU UAV Lab offered desired features for a modern research platform, a new system was developed: a small form-factor system with integrated sensors, sufficient processing power, and ease of development.

1.3 Problem Statement

The primary goal of the project was two-fold. On the hardware side, a new platform was developed which integrates a powerful processor with all needed sensors and communications hardware into a single Printed Circuit Board (PCB). The hardware was chosen to be as small as possible while still being within the technical ability of the VCU UAV Lab to assemble; second to this was pure raw processing power, which allowed for the execution of complex flight control and sensor fusion algorithms simultaneously. The final design was further constrained by some of the onboard sensors; some of the design and software for the sensors was performed by another graduate student as their thesis project, and thus had to be integrated as more or less a “black box.”

On the software side, a new software interface was developed which allows for easy reconfigurability, both at a lower level hardware interface as well as a higher level algorithms interface. As a major part of this work, a generic “flight control system Application Programming Interface (API)” was developed and implemented several projects at the VCU UAV Lab. This API allows for “dropping in” code which implements guidance and control algorithms without altering the algorithms or FCS code.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 details previous FCS work, both at VCU and elsewhere. Chapter 3 details the hardware chosen for the new platform, while Chapter 4 gives an overview of the lower level software for the platform. Chapter 5 gives details on the API design mentioned above. In Chapter 6, simulation and flight test results demonstrating the capabilities of the FCS developed here are presented. Finally, Chapter 7 discusses the overall project and a selection of future work which should be performed to maximize the usefulness of the final product.

Chapter 2: Background

This chapter will outline currently used UAV autopilot systems, including VCU's previously developed autopilots, other research autopilots, and commercial autopilots. Where applicable, the faults or limitations of these systems will be highlighted to justify the development of the new FCS in this work. Additionally, as a large portion of this work is related to the autopilot software, the background of various autopilot software frameworks will be outlined.

2.1 VCU Autopilot Systems

There were two different autopilot systems which were used in day-to-day operations and research at the VCU UAV Lab.

2.1.1 NextGen FCS

The NextGen FCS [1] was developed in 2009-2010 as a high processing power system intended to support flight control research. It utilizes a dual-processor architecture which pairs an FPGA-based "soft-core" MicroBlaze Central Processing Unit (CPU) with another FPGA-based "hard-core" PowerPC 405 processor. The NextGen is not intended to be a low-power or small form-factor system, measuring 3.8" by 3.0" by 2.75".

System Architecture

The NextGen is divided into two major components: the Instrumentation Control Module (ICM) and the Flight Control Module (FCM). The ICM handles all sensor and communications tasks, while the FCM handles all guidance and navigation tasks. Both the ICM and the FCM are FPGA-based “mini-modules” which contain an FPGA and the associated support hardware (Flash memory, RAM, etc.) and provide breakouts for the available I/O pins on the FPGA.

The ICM is based on the Xilinx Spartan-3 platform. This mini-module contains a 400,000 gate equivalent FPGA, 2 MB of onboard flash, and 1 MB of SRAM. The FPGA is suitable for hosting the MicroBlaze soft-core CPU and contains numerous I/O pins for interfacing the FPGA with external hardware. The ICM is designed to run all sensor and communications interfacing, so that the FCM can concern itself only with the running of guidance and control algorithms and support code. The ICM provides the FCM with preprocessed sensor and communications data via a dedicated inter-processor bus, the Module-to-Module (M2M) bus. The ICM dispatches reports over the M2M bus to the FCM, which processes them and then runs the guidance and control algorithms.

The FCM itself is based on far more powerful platform, the Xilinx Virtex-4 FPGA. The Virtex-4 can run at much faster speeds than the Spartan-3. In addition to the FPGA fabric, the Virtex-4 contains a hard-core PowerPC 405 CPU, capable of operation at up to 300 MHz. The Virtex-4 mini-module contains 64 MV of DDR SDRAM and 4 MB of flash memory. Unlike the ICM, the FCM does not have significant interfaces to other systems, and is primarily targeted at running the flight control algorithms.

The software of the NextGen is split between the two modules as well. The ICM software runs directly on the MicroBlaze processor, and is tasked with gathering and processing of sensor and communications data. The software design is fairly flexible, thus addition of new hardware modules is not a difficult task. ICM software also handles converting different sensor data readings into a common format. For example, while different Global Positioning System (GPS) sensors will report data in different formats, the ICM software will repackage these disparate formats into a unified, abstracted data form. The FCM receives data in terms of aircraft state parameters, including latitude, longitude, roll, pitch, etc. This allows the FCM to save even more cycles by receiving data in an already usable form, without needing to further parse the data.

As the FCM has a far more powerful processor and far more RAM, it is capable of running a full OS. The FCM runs the Linux OS, using the μclibc C library and the Xenomai real-time extensions to guarantee hard real-time performance of the flight control algorithms. The flight control algorithms themselves are based on a continued evolution of Proportional-Integral-Derivative (PID) control loops which have been used at the VCU UAV Lab for nearly a decade.

Shortcomings

The NextGen FCS, although powerful, had numerous shortcomings which led to the desire to replace it.

The most egregious shortcoming is the almost complete lack of documentation regarding the FCM software. While the FCM software was developed as part of previous research work, the source code was poorly (and in some cases, incorrectly) documented. In order to allow the use of a hardware Floating Point Unit (FPU) on the PowerPC in the FCM, several modifications had to be made to the Linux kernel as well as the GNU Compiler Collection (GCC) compiler to make use of the hardware FPU. None of these modifications were documented, and though patches were made available, they do not apply correctly on newer versions of Linux or GCC. Additionally, the use of Xenomai for hard real-time performance was not well justified or documented, and again specific versions are required to actually successfully compile the FCM software. This led to the necessity of maintaining an old, insecure system just to be able to build the FCM software, much less modify it to add new functionality. While several smaller features and fixes were applied over time, by and large the FCM software has remained largely in the state it was in when the NextGen was first built. Although the system is capable of flying UAVs very effectively, its lack of documentation and ability to be upgraded has made it very difficult to use in new projects.

In addition to the arcane nature of the FCM software, the choice of an FPGA based solution has also produced several problems. While the flexibility of being able to add new hardware "on-the-fly" has in fact been used to add support for new external sensors over the lifetime of the NextGen, the requirement of using the Xilinx Embedded Design Kit (EDK) to make any changes to the FCM hardware or ICM hardware and software poses several roadblocks to modernizing the system. Most notably is the failure of several of the custom FPGA cores to function correctly (or at all) in modern versions of the EDK. While the older version continues to be used as necessary, it does not run on modern operating systems; furthermore, due to the proprietary programming interface, it is difficult to get it to function in a virtual environment. As with the FCM, the ICM is also somewhat tied to out-of-date software, although modernizing the ICM is not an intractable problem.

The NextGen is also expensive, as minimizing cost was not one of its design constraints. The total cost to build a new NextGen system is over $1,000. Additionally, the NextGen is physically large and also power hungry; it can draw almost 1 Amp at 12 Volts. The physical size makes it impractical or impossible to use in small airframes, while the power consumption makes it impractical to use in single-battery, electric engined airframes.

Note that while the NextGen has many shortcomings, there are many aspects of the design which are a marked improvement over earlier and contemporary designs. The dual-processor architecture was briefly considered for this project, and was only scrapped mostly for reasons of cost versus benefit. The software architecture of the ICM is done well and provided many ideas which were applied to this work, including the use of a driver Globally Unique Identifier (GUID) for sensor interface drivers.

2.1.2 MiniFCS

Developed contemporaneously with the NextGen FCS, the miniFCS was from the start intended as almost the polar opposite of the NextGen [2]. The miniFCS was developed to be a low-cost, small form-factor flight control system which could fly in a wide range of airframes, including small single electric engine foam gliders. It was designed primarily to allow the development and flight testing of multi-UAV collaborative systems such as those in [3]. In comparison to the NextGen, the miniFCS is a much simpler, more straightforward system.

System Architecture

The miniFCS is based on a single Atmel AVR32 processor, which has a 32-bit pipelined processor running at 64 MHz, and features 64 KB of RAM and up to 512 KB of program flash. As designed, the miniFCS only requires two external sensors: a GPS and an attitude solution, either IR sensors or a dedicated IMU. All other hardware is contained on one small PCB, measuring 3.6" by 1.8". Unlike the NextGen, the miniFCS has no hardware floating point support, which limits somewhat the complexity of the algorithms it is capable of running.

The miniFCS software is also much simpler and more straightforward than the software on the NextGen. The miniFCS runs only on the AVR32 processor, without using an OS, and is effectively a single processor solution. Additionally, in a departure from previous VCU autopilots, the guidance and control algorithms were re-written from scratch instead of being based on the extant code used on the NextGen and elsewhere.

Shortcomings

The major drawback of the miniFCS is the lack of processing power. As mentioned above, the Atmel UC3 processor which runs the miniFCS algorithms only runs at 64 MHz and does not have a hardware FPU. While there is enough slack time to run more complex guidance and control algorithms, the miniFCS is not capable of also running sensor fusion algorithms. Additionally, the miniFCS has limited expandability; adding external sensors which are interfaced via the Inter-Integrated Circuit (I2C) or Serial Peripheral Bus (SPI) busses requires physically modifying the PCB with jumper wires.

2.2 Free/Open Source Autopilots

In the time since the development of the miniFCS and the NextGen FCS, a surprising category of FOSS autopilots has appeared. These autopilots, typically sponsored by corporate interests [4] or university research [5], tend to focus on the enthusiast market and are often low cost, albeit with performance which matches their cost. Nonetheless, they are becoming more and more popular and suitable even for more complex tasks.

2.2.1 Paparazzi Autopilots

One of the first FOSS autopilot solutions was the Paparazzi. First released in 2003 by students and faculty at École nationale de l'aviation civile (ENAC) [6], the Paparazzi project has grown since then to encompass a sophisticated solution for flight control, including FCS software, a Ground Control Station (GCS) [7], a simulation environment [8], and a plethora of hardware developed by the Paparazzi community [9]. The Paparazzi core can run on any number of hardware targets–currently, the main focus of the project is on ARM based MCUs such as ST Micro's STM32 and NXP's LPC21xx families; the project also runs on some lines of the Atmel ATMega family, with limited support [9].

The Paparazzi project and the surrounding community have produced around a dozen hardware designs [9] of varying degrees of complexity, size, and cost. The Lisa series of autopilots are all based on the STM32F1 processor, and come in different configurations which optimize the design based on mission parameters [10, 11]. The largest in the family is designed to be paired with a Linux based Gumstix board for advanced control; the smaller models can be used on their own as a simple autopilot or in tandem with other autopilots depending on the desired mission. All Lisa series APs have an onboard IMU and barometric pressure sensor, but require an external GPS and modem.

Krooz and KroozSD are larger form-factor boards which strive to be as much of an all-in-one solution as possible, at the expense of a larger board [12]. KroozSD is based on the STM32F405 processor, and, as the name implies, features the addition of a microSD card slot for data logging. As with the Lisa family, the KroozSD has an onboard IMU and barometric pressure sensor. In addition, the KroozSD has variants with integrated XBee Pro wireless modems or a bluetooth modem.

Two very small form-factor devices are also available: the Apogee, measuring 5.3 cm by 2.5cm by 0.42 cm, and the Umarim Lite, measuring 4.8 cm by 2.5 cm by 0.4 cm [13, 14]. The Apogee is based on the STM32F405 and features a full 9 DoF IMU as well as a barometric pressure sensor and integrated microSD card slot [13]. The Umarim Lite is based on the NXP LPC2148 and features a more limited IMU solution based on the Analog Devices ADXL345 accelerometer and the Invensense ITG-3200 gyroscope [14]. It does not feature any onboard barometer, modem, or GPS. Both small-formfactor devices are capable of running the full Paparazzi software and can control any number of airframes.

All Paparazzi hardware makes use of the same core software, which is comprised of classic PID controllers which operate the control surfaces of a given airframe [6]. The Paparazzi project also supports a wide range of external (or onboard) sensors [15, 11]. Paparazzi community members have also created several external sensor packages featuring GPS and IMU solutions which work with the majority of the Paparazzi autopilots [16, 11] Paparazzi hardware is available from various vendors and ranges in price from $150 to $700 [17]. The available systems vary in price, size, and onboard hardware. Most require at a minimum an external GPS and modem; some also require an external attitude solution.

2.2.2 ArduPilot

The ArduPilot project [4] aims to provide a unified solution for all types of robotics applications, include ground vehicles, rotary-wing UAVs, and fixed-wing UAVs. ArduPilot, along with ArduCopter, ArduPlane, and ArduRover, is a project run by the DIY Drones community [18]. ArduPilot is sponsored by 3DRobotics, who also manufacture the hardware for the project [19].

As with the Paparazzi project, there are several different ArduPilot hardware platforms available. At the low end is SparkFun's ArduPilot [20], which, at $25 USD is by far the most inexpensive autopilot commercially available. The SparkFun ArduPilot is based on the Arduino platform, and uses an Ateml ATMega328 as its main processor. It includes no onboard peripherals, which contributes to the low cost–in order to actually fly, external GPS and IR sensors must be purchased.

For midrange prices, several autopilots are available, manufactured by 3DRobotics as part of the APM project [4]. The Pixhawk PX4 is an open-hardware platform which targets academic and hobbyist communities [21]. The PX4 is based on the high-performance STM32F427 32-bit ARM processor, similar to the lower-end STM32 processors used in the Paparazzi [9, 22]. It integrates a ST Micro L3GD20H Gyroscope and an ST Micro LSM303D accelerometer/magnetometer for an IMU solution, and a Measurement Specialties MS5611 barometric pressure sensor. Unlike the Paparazzi autopilots, the PX4 has an additional safety switch on-board, which will detect failure conditions and return the autopilot into manual control mode. The PX4 costs $200 USD without the required external GPS or modem; with both it costs $380 USD [23].

There is also an APM developed autopilot, fittingly called the APM [24]. The APM is based on the Atmel ATMega2560, and has similar features to the PX4 and Paparazzi autopilots [25]. As with the PX4, the APM has an onboard IMU solution, in this case provided by an Invensense MPU-6000 6DoF accelerometer/gyroscope and a Honeywell HMC5883L magnetometer. In addition, the APM also features the Measurement Specialties MS5611 barometric pressure sensor seen on the PX4 and several Paparazzi parts [9, 22]. The APM kit is slightly more inexpensive than the PX4–it costs $160 USD for the base model and $340 with the necessary modem and GPS modules [19].

The ArduPilot offers a single autopilot solution which can fly a wide range of airframes as well as control ground vehicles. The same physical hardware can control a multitude of different equipment by the use of different software as needed for a given mission; the ArduPilot software comes in different configurations for fixed-wing, rotary-wing, and ground based vehicles [4]. Additionally, the platform can be controlled by an external controller using an ArduPilot specific command and control protocol [26].

2.3 University Developed Autopilots

As has been done at VCU, many universities have developed in-house autopilot designs for research purposes; some of them have been turned into commercial products [27, 28], though most are only used at the university where they are developed.

2.3.1 AggieAir and AggieNav

AggieAir is a blanket term covering a system of components which make up the FCS software/hardware developed and used at the Center for Self-Organizing and Intelligent Systems (CSOIS) at Utah State University[29]. The AggieAir system is charaterized by the use of Commercial Off-The-Shelf (COTS) components for the majority of the sytems; the novelty of the AggieAir implementation is the successful marriage of these components. The primary mover in the AggieAir system is a modified version of the Paparazzi software discussed above. Deemed "AggiePilot", the Paparazzi software has been modified to support a US military developed intermodule communication standard, Joint Architecture for Unmanned Systems (JAUS). JAUS is intended as a robust command and control protocol, which makes it well suited to both ground communication as well as intermodule communication.

Another crucial component of the AggieAir system is the AggieNav INS solution [29, 30]. AggieNav integrates a 6 Degree of Freedom (DoF) accelerometer/gyroscope, a 3-axis magnetometer, a GPS unit, and static and differential pressure sensors [29, 30, 31]. All hardware is connected to an AVR microcontroller which does hardware interfacing; data from the sensors is then offloaded to a Gumstix board running Linux which handles integrating all sensor data into aircraft state information. AggieNav uses an Extended Kalman Filter to derive an attitude estimation from accelerometer and gyroscope data; other data is provided to the rest of the AggieAir system as-is [31].

2.4 Commercial Autopilots

Several commercial autopilot systems are available, with varying degrees of features and prices. Note that while both Paparazzi and ArduPilot are available commercially, they are not included here; for the purposes of this work, "commercial" is used strictly to refer to propietary, closed-source systems only.

2.4.1 MicroPilot

MicroPilot [32] manufactures a range of autopilots which are suitable for flight control of a variety of airframes. These solutions are all based on the same hardware platform, with software licensing used to restrict the feature set of the less expensive autopilots. All MicroPilot autopilots have the same basic physical dimensions, measuring approximmately 10 cm by 4 cm and weighing between 25 and 30 grams [33, 34]

The MicroPilot MPx028 range offers a small form-factor all-in-one system [33], while the MP2128 range offers a higher performance at the cost of a larger form-factor [34]. All MicroPilot systems are capable of basic flight, but only the more expensive systems allow for more useful features such as run-time alteration of waypoints, automatic take off and landing, and in-flight data logging. The cheapest system, the MP1028G [35], costs $1500 USD, and does not support basic features such as in-flight waypoint adjustment. More expensive systems add this support, as well as additional servo controls, longer data logging, and automatic takeoff and landing, but at a fairly significant price. Indeed, the most prohibitive aspect of the MicroPilot for university research purposes is the cost. The cost of a MicroPilot system with comparable features to the miniFCS or NextGen is around $6000.

2.4.2 Cloud Cap Piccolo

Cloud Cap manufactures several autopilot systems, all of which offer basic flight control functionality and contain an integrated suite of sensors necessary for flight [36]. All of the Piccolo series share a common hardware base and common software, and as with the MicroPilot allow the unlocking of more advanced features depending on the licensing purchased.

There are currently three autopilots offered by Cloud Cap. The Cloud Cap Piccolo Nano is a small form-factor autopilot available in several configurations [37]. The Nano comes in three basic configurations, with varying degrees of complexity. The simplest "limited" model only supports 12 waypoints and a limited selection of takeoff and landing modes, while the "reduced" and "standard" configurations support 1000 waypoints and an assortment of takeoff and landing options. The "standard" also adds support for Differential Global Positioning System (DGPS) positioning and a laser altimeter, which adds the ability for traditional wheeled landings to the device. The Nano measures 4.57 cm by 7.62 cm by 1.1 cm, and weighs 22 grams. The base price for the Nano is in the $1000 range [38].

The Piccolo SL is the next step up in the Piccolo line, and offers a much more robust set of inputs [39]. In addition to all the features of the "standard" Piccolo Nano, the Piccolo SL offers differential and absolute pressure sensors, an integrated 3 axis gyroscope and 3 axis accelerometer, and more external interface options for end-user sensors. The Piccolo II is the top-end Piccolo model and offers additional I/O and peripheral support compared to the SL or the Nano [40].

All the Piccolo line also require the propietary Cloud Cap GCS to operate effectively. While pricing is not generally made available, most Piccolo systems are expected to cost on the order of $20,000 USD or more for both the autopilot and the GCS required.

2.4.3 Adaptive Flight FCS20

The Adaptive Flight FCS20 [28] is a commercially available flight control system based on a system developed at Georgia Institute of Technology [27]. FCS20 is a dual-processor system, somewhat similar to VCU's NextGen; it makes use of both a Digital Signal Processor (DSP) as well a FPGA to handle all flight control tasks. The original design goals for the FCS20 were small formfactor, high processig power, and high reconfigurability. Software tasks are split between a soft-core processor in the onboard FPGA and the DSP. All onboard hardware is interfaced through the FPGA, which communicates with the DSP via a dedicated bus. The DSP is responsible for running all flight control tasks as well as running sensor fusion algorithms when needed.

2.5 UAV Software

Much work in the field of UAV software architecture relates to reconfigurability; that is, the ability to fly the same software on as wide a range of airframes as possible without major software changes.

2.5.1 Reconfigurable Flight Control System Architecture

In [41], an event-driven, service oriented architecture for UAV software is proposed to address this concern. Deemed the Reconfigurable Flight Control System Architecture (RFCSA), this architecture is designed to be as modular and as easy to use as possible, to facilitate rapid prototyping and deployment of new software. The RFCSA architecture also goes into the hardware realm; each module in RFCSA is a self-contained hardware module. Each module has a simple interconnect which allows for module chaining or module stacking. The module does not need to know about other modules in the system.

The interconnect architecture is effectively a single communications bus with a service discovery mechanism, which allows for the addition of a new module with limited or no changes to the other modules. This occurs by the use of a Service Oriented Architecture (SOA). Each module offers one or more “services” to the system as a whole. A service discovery mechanism exists whereby new modules can announce their services to the system while also querying other modules for their services. This means that the modules do not need to know anything about each other until they are connected together.

The work in [41] also proposes a hardware interconnect for each module. Each module has a standardize hardware connector which allows stacking or daisy-chaining of modules to form a complete system. This works in tandem with the service discovery architecture by used of a shared bus.

2.5.2 Unified Vehicle Interface/Standard Vehicle Interface Library

Another example of a modular architecture is found in [42], which forms the basis for the FCS20 outlined above in Section 2.4.3 [27]. Here, the framework is split into two parts: The Unified Vehicle Interface (UVI), which is a software/hardware stack that handles low level sensor interfacing and, if necessary, basic flight control; and the Standard Vehicle Interface Library (SVIL), which offers a unified, FIFO channel based interface to higher level algorithms.

The UVI is a software/hardware stack which pairs an FPGA, DSP, and a basic sensor suite, as well as the software necessary to interface to them. It pairs both a soft-core processor in the FPGA and a hard-core processor to perform these tasks and is also capable of flight in and of itself. However, the flight control capabilities are limited to simple algorithms.

The software is unique in that it allows for the arbitrary mapping of device channels to either real hardware in the UVI or simulated hardware for software-driven testing. This is accomplished by abstracting all drivers to be packet streams; the driver code will either take hardware data from driver code or simulation data and present it to the higher-level algorithms in the same format. This approach allows for relatively painless transition between software-driven prototyping and hardware-driven hardware-in-the-loop simulation (HILS) or actual flight testing.

2.5.3 RAMS Simulator

A similar modular approach has been taken by previous work in the VCU UAV lab, with higher level decision algorithms. The RAMS simulator [43] allows for the testing and validation of collaborative algorithms in a purely software environment, without requiring modification of the algorithm code at all. This is accomplished by building the decision code as a shared library and defining initialization and run functions which are invoked by the simulator. This allows for the code to be tested as it will run on the UAV platform, while minimizing the risk of hiding bugs by an imperfect test framework. Additionally, the ability to easily test in this kind of environment helps drastically reduce turn-around time for code changes and new algorithms; expanding this ability to the lower level guidance and control algorithms is one of the motivations of this work.

2.5.4 Open Control Platform

The Open Control Platform (OCP) introduced in [44] exemplifies several modular design qualities. The OCP is designed mostly with controls engineering in mind—it allows for the abstraction of components in the system as black boxes with inputs and outputs. This allows for controls systems engineers who may not be familiar with low-level programming interfaces such as sockets to effectively program intertwined systems using more familiar controls metaphors.

Underlying the controls system metaphor is an event-driven, real-time communications framework which allows for various components to generate and respond to events. In this model, for example, a sensor could generate a position event, which contains the current latitude and longitude of the system. The guidance or control code would then receive this event and respond accordingly, without knowing any details about how this data was generated. The event generator could be a software simulator as easily as it could be a real GPS unit.

2.5.5 JULIET

The JULIET system demonstrated in [45] offers a more operating system style approach to software modularity. In JULIET, every task in the system—be it a sensor processing task or a higher level control task—is assigned a priority and a scheduling interval. The main loop of the system executes every millisecond, and selects the task to be run based on priority and schedule. The JULIET system also inserts buffers between stream-based (e.g. serial) devices and the rest of the software. This inserts a layer of abstraction on top of the device drivers. JULIET also offers inter-task communication by way of both global state variables and FIFO queue channels. The functionality of JULIET is more akin to early, cooperative multitasking operating systems; this approach allows for a more flexible implementation of various tasks that the autopilot may be required to run.

2.5.6 Borrowing from Design Patterns

Although not strictly related to UAV autopilot software, the ideas proposed in [46] are still relevant. The central thesis of this work is the application of design patterns to UAV ground control software. Design patterns were first proposed in the seminal 1995 book *Design Patterns*, commonly known as the "Gang of Four" book after its four authors[47]. While the patterns dictated within are largely focused on object-oriented software, the central tenets apply well to any software design: write software which is modular and facilitates easy reuse. The work in [46] applies these concepts to UAV ground control software. The same principles, with a bit of modification, should apply to the development of autopilot software as well. Indeed, the big take away from this paper is that design patterns are inherently abstract and must be moulded to fit a given application; some of the lessons presented within regarding the development of the ground control station have proven valuable during development of the software described herein.

Chapter 3: Hardware Design

This chapter will detail the hardware design for the Aries PCB, including a discussion of the microcontrollers used as well as a discussion of the on-board peripherals. The name Aries was chosen primarily because Aries has the astrological symbol of the Ram, which is suitable given VCU's current branding. Attempts were made to make an acronym of the name, but nothing suitable was found.

3.1 Requirements

The principal requirements for the Aries FCS were simple: design a single board flight control system with an integrated IMU solution and the processing power to run both IMU and FCS algorithms. Additionally, the board should be small enough to fit in the foam gliders used in the VCU UAV lab–ideally, the size should be no bigger than the miniFCS.

As the principal requirements specify having an on-board IMU solution, it was also desired to have all other necessary sensors integrated on the PCB. This necessitated the addition of an on-board GPS module, as well as an integrated modem connector. Although all of these can be interfaced externally if need be, the design of the Aries FCS is one that allows a single board, "plug and play" architecture that only requires an external RC receiver and optionally external antennas for the GPS and modem.

In order to be able to fly the smaller foam gliders used for collaborative testing in the VCU UAV Lab, maximum dimensions similar to the miniFCS were imposed: 3.6" by 2.0". The 2.0" dimension is constrained by the desire to have 17 0.100" spaced headers on one side of the board, giving 8 servo outputs, 8 servo inputs, as well as a Pulse Position Modulation (PPM) input from the RC receiver.

The first revision presented in this paper did not meet that size requirement: it measures 4” by 4”. Due to time constraints, a smaller design was not possible; as discussed in Chapter 7, work is ongoing to make a board which does fit the design specifications.

3.2 Microcontroller

Several microcontrollers were considered for the Aries design, including the possibility of a dual-processor system. Initially, it was hoped that a sufficiently fast DSP processor with a double precision FPU could be found and integrated alongside a commodity processor for a similar architecture as that of the NextGen. Unfortunately, research did not find a suitable DSP processor which could handle running both the IMU algorithms and the FCS algorithms; finding one with a double precision FPU in the desired price range was nigh impossible as it was.

With the possibility of a dual-processor architecture out of the question, the focus of processor research turned towards powerful single processor solutions as well as System on a Chip (SoC)s. Processor solutions from Freescale Semiconductor, Atmel, MicroChip, and ST Microelectronics were all considered. Primary considerations were the presence of an FPU, as well as the availability of peripheral interfaces such as SPI, I2C, and Universal Asynchronous Receiver Transmitter (UART). Hardware PWM generation was also considered a must. Many processors were omitted for failing to meet the above requirements, including almost all of Atmel and MicroChip’s offerings. The only Freescale offerings which met the criteria offered small amounts of RAM and Flash, and were not generally available as of Q2 2013 when processor selection was occurring.

ST Microelectronic’s STM32 line offered several compelling options. The entire STM32 line is based on ARM 32 bit Cortex processor cores. The STM32F3 and STM32F4 offer Cortex-M3F and Cortex-M4F processors, respectively, which feature the desired IEEE754 single precision floating point unit. The STM32F4 has seen great success in several Paparazzi autopilot hardware [9], and a development board can be purchased for as little as \$10. Additionally, the STM32F4 can be programmed without need of an expensive or propiertary Integrated Development Environment (IDE). The freely available ARM embedded toolchain [48], based on GCC, is quite capable of generating firmware images for the STM32 processor. A FOSS project which can program any STM32 chip using the built-in propietary ST-LINK on the aforementioned inexpensive development board is available. Indeed, partially due to the low cost of entry, a large enthusiast community has sprung up around the STM32F4 line.

The STM32F4 line has several sub-lines which offer different functionality, onboard RAM and Flash sizes, and most importantly different package sizes. The same code (provided it will fit into the smaller flash/RAM) will run unmodified on all processors in the STM32F4 line. This capability will allow the development of different boards using different processors in the future, if the need arises. All processors in the STM32F4 line have the hardware FPU and all are capable of running at up to 168 MHz core clock. For the Aries project, there were two important decisions to be made. STM32F405 and STM32F407 are mostly identical; the 405 omits a dedicated camera interface as well as omitting an integrated Ethernet MAC, and is available in smaller form-factors while the 407 is available in packages with enough external pins to allow for external RAM. Additionally, ST provides the 415 and 417 packages, which are identical in every way with the addition hardware cryptographic acceleration. After consideration, it was decided to use the STM32F407VGT6, with the possibility of swapping in a pin-compatible 417 in the future.

The selected processor for the Aries FCS features 1 MB of flash memory and 192 KB of static RAM [49]. It supports up to 140 GPIOs, although most peripheral functionality are multiplexed on the GPIO pins through an alternate functionality system which allows up to 15 different functions to be multiplexed on one pin. The 100 pin version selected only offer GPIO ports A-E, plus two port H pins, giving a total of 82 GPIOs available. For external connections, the STM32F417 has a multitude of peripheral bus connections. As configured for the Aries FCS, the processor supports an Ethernet interface (using RMII mode), 4 USARTs, 1 UART, 1 I2C, 1 SPI, and MicroSD.

The STM32F4 line is not without faults. As outlined in the detailed sections below, the several of the STM32F4's peripheral cores have numerous faults in-silicon [50]. These faults and limitations, particularly with respect to the integrated I2C core, provided a significant challenge when implementing the design of this work. Nonetheless, in spite of these faults, the STM32F4 has proven more than adequate at the tasks at hand .

3.3 Safety Switch

In order to ensure safe operation of an RC aircraft in the event of autopilot failure, a safety switch is typically used which will "fail safe" and return manual control to the safety pilot when the autopilot fails or loses power. As the design goal of the Aries was to have an all-in-one solution, the safety switch has been integrated onto the board. The design of the switch is based on the safety switch on the miniFCS, using updated parts and a new, more power processor to allow for a greater degree of configurability.

The switching mechanism is the same as used in the miniFCS, which used two sets of tri-state buffers, one triggered off of an enable signal and one off of the inverse of the same signal, to determine whether the RC pilot or the FCS controls the servos. As with the miniFCS, an active low signal and a pull-down resistor are used to ensure the switch fails safely in the event that the safety switch processor somehow fails; unless actively driven high the switch will default to allowing manual RC control. Unlike the miniFCS safety switch, the switch on the Aries does not have input hysteresis. Analog VHF transmitters are no longer used for RC control, and the 2.4 GHz radio transmitters operate a digial link which is not susceptable to noisy outputs.

Instead of an 8 pin ATTiny processor, the Aries safety switch uses a much more capable processor from another STM32 line, the STM32F050. This processor as selected has 32 pins, enabling PWM capture on all 8 input channels, as well as PPM capture and an SPI interface to the main processor for reconfigurability and PWM value reading. The safety switch maintains the same dual mode indicator output of the miniFCS safety switch, using two pins to indicate auto/manual mode and RC loss mode. In addition, the processor outputs a data ready interrupt pin which signals that new PWM values have been read.

As the new safety switch can read all 8 channels, it also supports inter-processor communcation so that the main processor can read the current input values of all PWM channels. This is accomplished via SPI, using a set of 16 “registers” in the safety switch which may be read by the main processor. The main processor may also change which channel is used for auto/manual mode and RC loss mode, as well as the thresholds for both; doing this on the miniFCS safety switch required recompiling and re-programming the safety switch processor.

3.4 Bootloader and Programming

Both the primary FCS processor and the Safety Switch processor support programming and debugging via Serial Wire Debugging (SWD). Additionally, they both support reprogramming only via a custom STM bootloader over serial [51, 52]; the main FCS processor also supports reprogramming via USB Device Firmware Upgrade (DFU) mode. To minimize board space, a single 5 pin SWD port is multiplexed between both the primary processor and the safety switch processor.

The bootloader circuitry is set up with pull-down resistors such that, by default, no processor is in bootloader mode and the onboard SWD connector is connected to the main processor. Jumpers are used to enable the processor boot modes (BOOT0) of each processor; these pins must be asserted at power up as their value is latched on the 4th rising edge after the processor clock stabilizes [51]. After this, the value of BOOT0 is ignored. DIP switches are also used to control whether the modem is available to the host computer or not. To save board space and ensure no interference from the normal operation of the main processor, the BOOT0 select is used here as well; thus, when the main processor BOOT0 is asserted, the modem is visible to the host computer. A single jumper is also used to multiplex the SWD port between both processors.

Multiplexing is handled by a 6 input mux from TI [53]. The mux does not allow independent control of each input but instead groups the inputs into two sets of three. This works well for the application here; one set of three is used to mux the correct programming pins (clock, data, and reset) between the two processors.

While the SWD interface was the main interface used during development, the ability to quickly program multiple boards with only a USB cable and a few jumpers will allow rapid deployment of future software updates to the Aries boards.

3.5 On-chip Peripherals

The STM32F4 line provides several dozen on-chip peripheral cores, each offering some form of connectivity or hardware acceleration. Only the full-size, 176-pin versions even offer enough pins to utilize all peripherals, but the Aries board makes use of around a dozen of them.

3.5.1 STM32 clock and bus architecture

The STM32F4 line has both a high-speed and a low-speed clock; the former is used for clocking the core and the latter for the real-time clock. The high-speed clock can be derived from three sources: an internal 16 MHz oscillator, an external oscillator ranging from 4 to 26 MHz, and a Phase-Lock Loop (PLL) run off of the High-Speed Internal (HSI) or High-Speed External (HSE) oscillators, which allows operation at up to 168 MHz. The low-speed clock can be derived from an internal 32 kHz oscillator or an external 32.768 kHz oscillator. The Low-Speed Internal (LSI) clock is very inaccurate; for accurate RTC performance the device must be fitted with a Low-Speed External (LSE) oscillator. There are four internal clocks of note: SYSCLK, HCLK, PCLK1 and PCLK2. SYSCLK is the fundamental system frequency and is also the core clock for the Cortex-M4F processor. HCLK is the clock frequency for the AHB busses, and is derived from SYSCLK; likewise, PCLK1 and PCLK2 are the clock frequencies for the APB1 and ABP2 busses and are derived from HCLK. For the Aries board, SYSCLK has been set to 160 MHz instead of the maximum of 168 MHz, primarily because of one of many quirks of the I2C core which requires PCLK1 to be an even multiple of 10. With SYSCLK and HCLK set to 160 MHz, this gives PCLK1

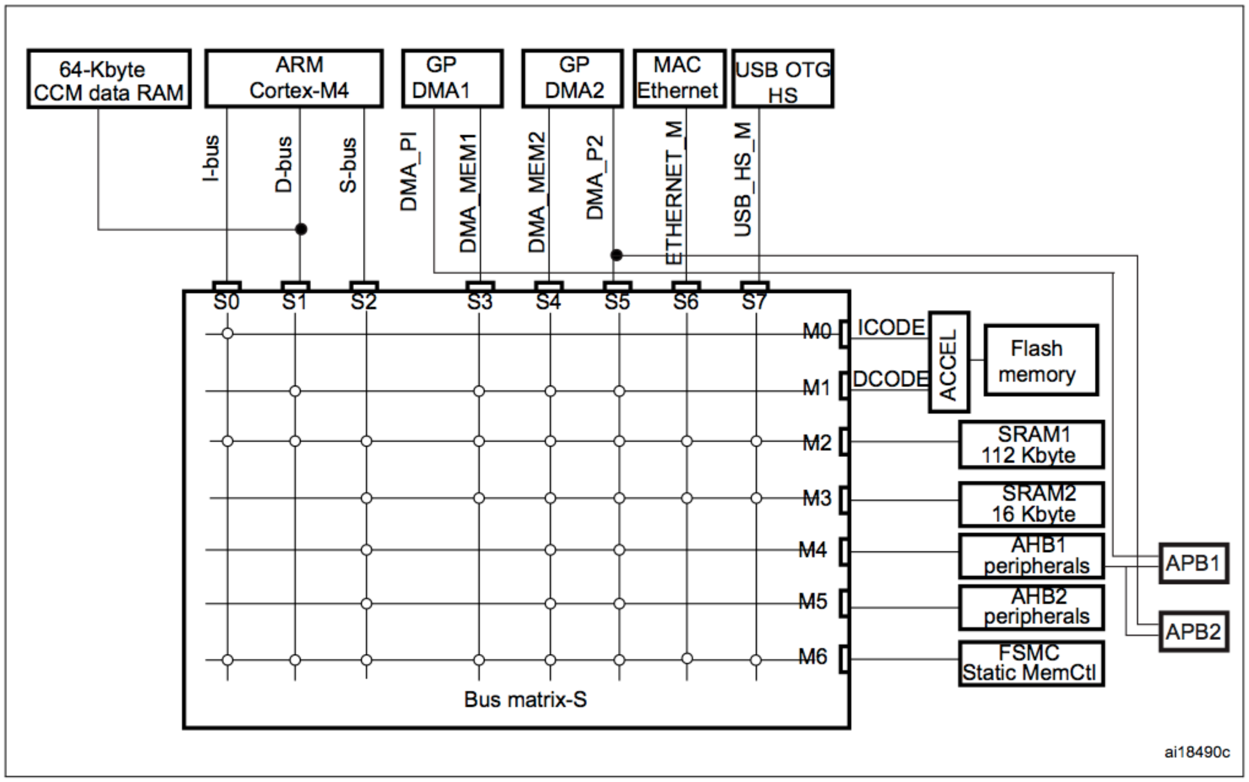

Figure 3.1: STM32F4xx Bus Matrix

as 40 MHz and PCLK2 as 80 MHz.

The STM32F407 has a multi-port bus matrix connecting the processor, memory, and internal peripheral busses, shown in Figure 3.1. Note that not all busses are interconnected; additionally, the DMA controllers have dedicated ports to the peripheral busses and, for DMA2, a separate bus matrix port for memory-to-memory transfers.

There are 5 peripheral busses in total. AHB1 has "internal" peripherals such as the Reset and Clock Control (RCC) and Direct Memory Access (DMA) cores, as well as the GPIO cores, the Ethernet MAC, and the USB On-the-go (OTG) 2.0 controller. AHB2 has the USB OTG 1.1 controller, and on the 41x line has the CRYTPO and HASH cores for hardware accelerated cryptography support. AHB3 only has the external Flexible Static Memory Controller (FSMC), which interfaces external memories to the device. All AHB busses run at the core clock of the processor. The two external peripheral busses are APB1 and APB2, which are both "nested" under the AHB1 bus. APB1 runs at half the speed of APB2, which in turn runs at half the speed of the AHB busses. APB1 contains most of the USARTs, both UARTs, all three I2C, two SPI, most of the timers, the RTC, and the DAC. APB2 contains the remaining two USARTs, SPI1, several timers, the ADC, and the SD card interface. All peripheral cores are memory-mapped and accessing their configuration registers is done via aligned memory access.

3.5.2 GPIO - General Purpose Input/Output

The STM32F4xx line provides up to 140 GPIO pins, spread across 8 ports of 16 pins each and 1 port of 12 pins. Ports are referred to by a letter, from “A” to “I”; pins are referred to by number from 0 to 15. Throughout the rest of this document, GPIO pins will be referred to by the short form “PA0” for GPIO Port A, Pin 0, “PB4” for GPIO Port B, Pin 4, and so forth. Each GPIO Port is independently configurable, and a selected device may not have all Ports, or even all the pins for a port–the STM32F407VG processor used for the Aries PCB only has GPIO Ports A-E and pins 0 and 1 of Port H, which are necessary for the HSE.

To allow the multitude of on-chip peripherals to talk to the outside world, each GPIO pin can be configured as one of 16 Alternate Functions (AF), which acts as a multiplexer to connect the GPIO pin to different internal peripherals. In addition to AF mode, each pin can be configured in IN (input), OUT (output) and AN (analog) modes–the latter used to configure pins to be ADC inputs or DAC outputs. Each AF has a specific domain–for example, AF4 is only used for the three I2C cores. A complete list of alternate functions is available in Table 3.1.

Alternate Function	**Domain**
AF0	SYS - Debugging, Clock Output, Trace Port
AF1	TIM1/2 - Timers
AF2	TIM3/4/5 - Timers
AF3	TIM8/9/10/11 - Timers
AF4	I2C1/2/3 - I2C and SMBus
AF5	SPI1/2, I2S2(ext)

AF6	SPI3, I2S3(ext)
AF7	USART1/2/3, I2S3ext
AF8	UART4/5, USART6
AF9	CAN1/2, TIM12/13/14
AF10	OTG_FS, OTG_HS - USB 1.1 and 2.0
AF11	ETH - Ethernet MAC
AF12	FSMC, SDIO, OTG_FS - External Memory controller, SD card interface, USB 1.1
AF13	DCMI - Camera Interface
AF14	Unused[1]
AF15	EVENTOUT - Event trace support

Table 3.1: STM32F4 Alternate Functions

Note that the AF system has some drawbacks–the AF mapping is not a free-for-all but rather a mapping of specific pins to specific devices. Some AF choices are mutually exclusive. For example, the SDIO AFs are only present on one set of pins. These pins are also used by UART5, which likewise only offers one set of AF pins; thus, the use of SDIO and UART5 is mutually exclusive. However, I2C1 can be used on several different pins on Port B, offering some flexibility which is more typical of the AF system. The decision of which AF to use for which pin drove several design choices for the Aries PCB.

In addition to being configurable as AF, AN, IN, or OUT, each pin has additional configuration options. The speed of the pin can be changed–lower speeds give lower power draw. Each pin has configurable pull-up or pull-down resistors, which can be selected or disabled as needed. In several cases the internal pull-ups or pull-downs are sufficient to remove the need for external resistors. Pins can also be configured in push-pull or open-drain mode; in the former, the output is driven regardless of the output, while in the latter the output is only driven to 0 and kept floating in other

[1]This is used on some pin-compatible STM32F4 processors, however

modes. Open-drain mode is necessary for certain busses, notably I2C, to work properly.

3.5.3 EXTI - External Interrupt

External interrupt support is provided by the EXTI core, which allows for up to 16 external interrupt sources to be connected to 16 interrupt lines, as well as providing 7 additional interrupt sources from internal peripherals. The EXTI interrupt lines are mapped such that EXTI Line 0 maps to GPIO PA0, PB0, PC0 and so on–which does impose some design constraints as two Pin 0s cannot both be used as external interrupts. EXTI interrupts can be triggered on rising, falling, or both edges. The Aries board mostly makes use of the EXTI controller for several Data Ready (DRDY) pins from the various external devices discussed in Section 3.6.

3.5.4 DMA - Direct Memory Access

Direct Memory Access hardware allows some types of peripheral-to-memory and memory-to-peripheral transfer to occur without using any CPU cycles to mediate the data transfer. This is useful for bulk data transfer to and from byte-oriented peripherals, as well as for peripherals such as the ADC whose update speed is too great to effectively allow an interrupt-driven solution. There are two DMA controllers in the STM32: DMA1 for APB1 peripherals, and DMA2 for APB2 peripherals and memory-to-memory transfers. The DMA controllers have two ports: one for peripherals and one for memories, with DMA2 multiplexing the peripheral port with a second memory port. Connectivity of the DMA controllers to peripherals and memories is shown in Figure 3.1.

Each DMA controller has 8 streams with 8 channels each Streams can be active simultaneously as the DMA controller has a 4 tier priority system to handle the situation when two streams are vying for the same port [54]. Each stream/channel combination is specific to a given STM32 peripheral core, and typically only one side (receive or transmit) for a given core. For example, DMA1 Stream 4, Channel 4 is only for UART 4 transmit. To avoid conflicts, some peripherals are mapped to multiple DMA stream/channel combinations, in a manner reminiscent of the GPIO AF system discussed earlier. DMA requests are used mostly for USART send and receive functionality as well as ADC sampling, as discussed in Sections 4.2.7 and 4.2.9.

3.5.5 USART - Universal Synchronous/Asynchonous Receiver Transmitter

The STM32F40x line provides 4 USARTs and 2 UARTs. USARTs 1, 2, 3, and 6 are "full-featured" and provide, in addition to standard receive (RX) and transmit (TX) functionality, hardware flow control using Clear to Send (CTS) and Ready to Send (RTS) signals, synchronous clock support including SPI master emulation, and irDA and Smartcard mode emulation USART1 and USART6 can operate at baud rates up to 10.5 Mbits/second, while USART2 and USART3 can operate at baud rates up to 5.25 Mbits/second. UART4 and UART5 are simple UARTs which only provide TX/RX capability and irDA support. All U(S)ARTS are full-duplex and have full DMA support for both receive and transmit. The Aries board uses U(S)ARTs for a number of external peripherals, including the modem, GPS unit, and debugging console.

3.5.6 I2C - Inter-Integrated Circuit

All STM32F4xx processors provide 3 I2C interfaces, capable of operation at up to 400 kHz. As mentioned in Section 3.5.1, in order for I2C to function correctly at speeds of 400 kHz, the APB1 clock must be an even multiple of 40 MHz; this is among one of many "features" of the I2C core that created a large amount of work necessary to have the core functional. As outlined in the errata, the I2C core has several issues with I2C timing [50]. The I2C core is unstable at 100 kHz; it must be run either at 88 kHz or slower, or at 400 kHz. In addition, several of the registers are configured such that the act of merely reading from them causes the state of the register to change; in particular, reading the SR2 register more than once after the address acknowledge event causes the I2C core to become unresponsive. The Aries software mostly works around these bugs as explained in Section 4.2.8. I2C is used for communication with the IMU sensors.

3.5.7 SPI - Serial Peripheral Interface

The SPI core of the STM32F4xx is capable of operating at speeds up to 21/42 MHz and support both master and slave operation, 8 or 16 bit transfers, and hardware or software slave select management. It can support 2 or 3 wire SPI, although only three wire is used in the current application. SPI is used both for the onboard EEPROM as well as reading PWM values from the safety switch.

3.5.8 ADC - Analog to Digital Converters

There are three available ADCs on the STM32F4xx; however, they are configured in a master-slave configuration, with ADC1 being the “master” peripheral. Not all channels are accessible by all ADC cores. The ADC can be configured to sample up to 16 input channels in a sequential or interleaved manner. ADC channels are used for the barometric sensors and battery voltage level monitoring.

3.5.9 TIM - General Purpose Timers

The STM32F4 line provides up to 14 timers of various degrees of complexity. Each timer can output up to 4 channels (although some only have 2 or 1 channel, and two have 0 outputs or inputs), and all channels are independent. Most of the timers are 16-bit, but TIM2 and TIM5 offer 32-bit counters. Of the available timers, the Aries makes use of 3 of them: two for PWM output and 1 for timing purposes. Details on the timer usage can be found in Section 4.2.10.

3.5.10 ARM Cortex M4F Cycle Counter

In addition to the STM32F4 timers, the ARM Cortex-M4F core provides a cycle counter as part of the debug framework. This counter increments once per clock cycle if enabled, as it is a 32-bit register it overflows every 27 seconds. The cycle counter was mostly used during development to perform timing and performance analysis for several of the more complex algorithms.

3.6 External Peripherals

Using the above on-chip peripherals, several on-board and off-board peripherals are interfaced to the Aries system. A block diagram showing the connections between the on-board devices and the STM32 is shown in Figure 3.2. Details on the additional peripherals are given below.

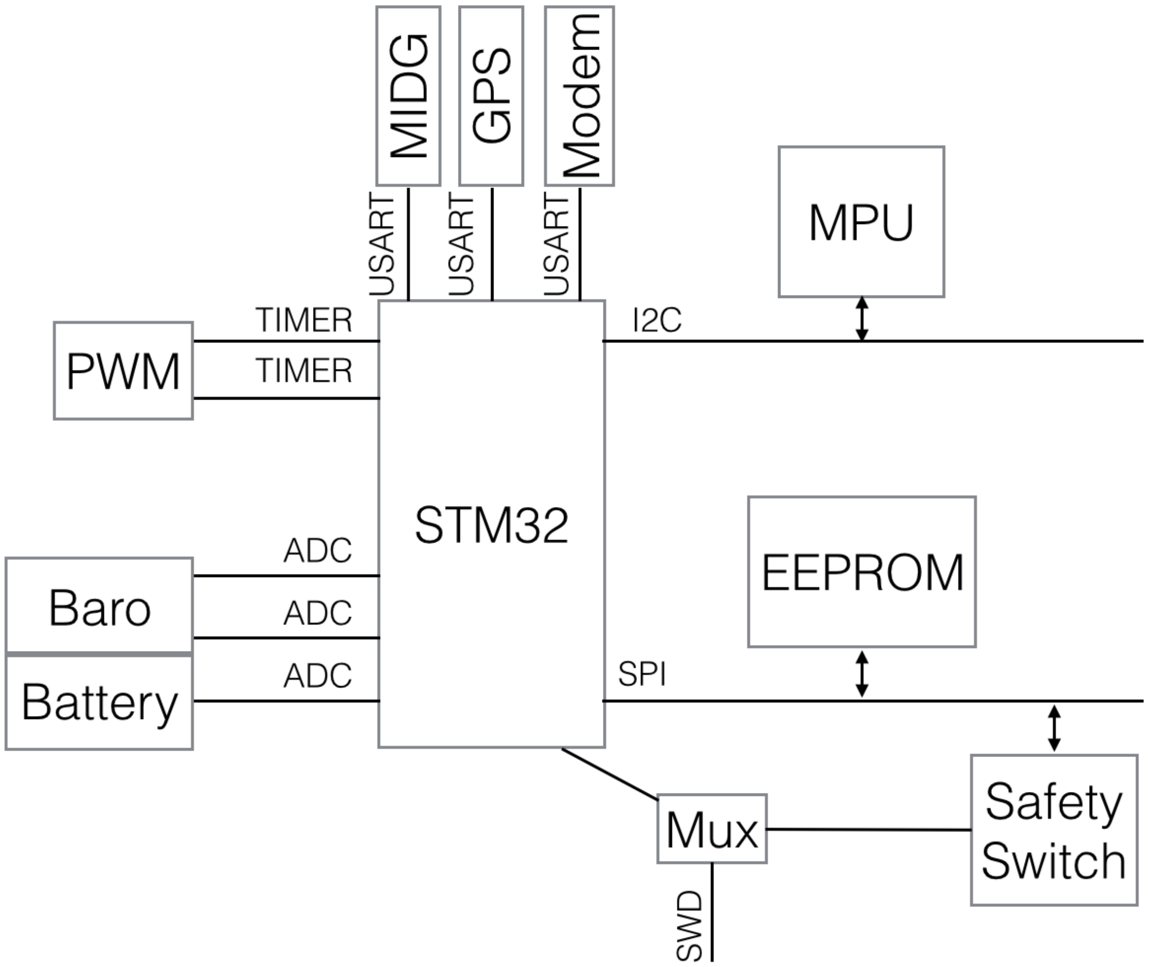

Figure 3.2: Aries Hardware Block Diagram

3.6.1 EEPROM/Parameter Flash

In order to function effectively, FCS hardware must provided some method of permanent, non-volatile storage; this allows for the preservation of tuning parameters between flights, even when the system was powered off. This is typically implemented as some kind of EEPROM or Flash storage. For both current FCS platforms at VCU, the permament storage is a special “user page” of the onboard flash memory of the respective processors.

Unfortunately, the STM32F4’s onboard flash is ill suited for this use. While it can be done [55], the process is slow and could result in locking up the entire processor during flight. Additionally, the flash in the STM32F4 is limited to around 10,000 write cycles, which is fairly low in the face of external flash which supports 1,000,000 write cycles [56].

An external flash chip was selected, a 512 Kbit SPI interfaced flash from MicroChip. The 25LC512 provides 512 pages of 128 bytes, for a total of 64K flash space, divided into 4 sectors of 128 pages each. Typical erase times for the 25LC512 chip are 5 ms per page or 10 ms per sector; page writes take an additional 5 ms for a total erase/write cycle of around 10 ms per page [56]. Each page is guaranteed to have minimum 1,000,000 write cycles. The chip is SPI-interfaced and capable of bus speeds up to 20 MHz.

3.6.2 GPS Module

The GlobalTop PA6H GPS module using the MediaTek MTK3339 chipset is provided onboard. The PA6H is an all-in-one GPS solution integrating the MTK3339 chipset with a small antenna. The module is small (16 mm by 16 mm by 4.7 mm) and does not require a large amount of external circuitry. It supports up to 66 channels with a sensitivty of -145 dBm, and has accuracy within 1.8 meters (6 feet). It can be configured to output updates at up to 10 Hz, athough this requires changing the default baud rate to something higher than the default 9600 baud. Unfortunately, the GPS does not support saving settings to its internal NVRAM, so it must be re-configured on boot unless an external battery is supplied. Additionally, supplying an external battery allows for warm and hot starts, reducing the time to first fix to around 5 seconds.

The GPS unit also supports using an external active antenna, and is compatible with GPS antennas from uBlox and others. An MCX-style antenna connector is provided; the GPS will automatically use the external antenna if it detects one.

3.6.3 Modem

For initial testing, the same Xbee Pro 900 MHz modem found on the miniFCS was used. This modem is a simple point-to-point modem capable of operation at up to 57,600 baud and additionally supports mutli-point operations, although that functionality is unused here. Although it does not offer the throughput of the modem used on the NextGen, it is much lower cost and lower power, making it more suitable for this application.

3.6.4 IMU

As mentioned before, the on-board IMU was developed by another student as part of their thesis project, and as such the choice of IMU sensor was dictated by that project[57]. The MPU9150 is an all-in-one 9-axis accelerometer, gyroscope, and magentometer, consisting of an MPU6050 accelerometer and gyroscope and an AK8975 magnetometer on the same die. The MPU6050 and AK8975 are separate sensors, but share the same die and are connected to the same I2C bus pins on the chip.

3.6.5 Barometric Sensors

As a time and cost saving measure, the barometric sensors from the miniFCS were re-used for the first revision of the Aries PCB. The board features a static pressure sensor for altitude measurement and a dynamic pressure sensor for airspeed measurement. The static pressure sensor is the Freescale MP3H6115A. It operates over a range from 15 to 115 kPa and has a linear voltage transfer function from 0V to the source voltage. Due to its wide range relative to the pressure ranges typically encountered during flight, the actual output voltage range during flight is only around $2.5V$ to $2.7V$ with $v_s = 3.3V$. To compensate for this, a gain/offset stage using an op-amp circuit is used to offset the voltage output by $-2.53V$ and apply a gain of 18.7[2].

The dynamic pressure sensor is the Freescale MP3V5004DP. This sensor can measure from 0 to 3.92 kPa pressure difference. The equations used to translate the pressures for both sensors can be found in Section 4.3.10.

Chapter 4: System Software Architecture

This chapter outlines the system software created for the Aries FCS, including both the interface to the STM32F4's internal peripherals as well as the interface to the on-board and supported off-board peripherals. Details on the FCS software itself can be found in the next chapter

4.1 Software Overview

All platform software is built using the freely-available ARM GCC Embedded toolchain[48], which is a version of GCC maintained by ARM Holdings for developing bare-metal ARM applications. It ships with a modified NewLib C standard library and fully supports the floating point unit in the STM32F4's Cortex-M4F processor. In addition, it is available and operational on all major platforms without requiring the licensing of an expensive IDE and toolchain for development. Programming the STM32F4 is possible using an open-source implementation of ST Micro's STLINK[58], which uses the commonly available libusb project and the STLINK/V2 integrated on the STM32F4-Discovery board to program and debug any compatible ST Micro processor. The combination of the STM32F4-Discovery, stlink, and the ARM GCC Embedded toolchain allows for development on the STM32F4 processor for only the cost of the STM32F4-Discovery board, and provides a complete compiling and debugging solution using GCC and GNU Debugger (GDB).

4.1.1 Software Architecture

The system software has seven primary modules:

- Low-level drivers which directly interface to the STM32F4xx processor's peripherals.
- High-level drivers which control external sensors and interfaces using the low-level drivers.
- Sensor fusion, communications, and data storage code which unifies various sensors and interfaces.
- An API which presents the unified data in a predefined format
- FCS code which actually determines the flight path of the plane.
- A configuration framework which configures all of these
- A scheduling framework which runs tasks for the high-level drivers; sensor, communication, and data storage; and FCS code.

A block diagram showing the relationship between all modules can be seen in Figure 4.1. This demonstrates the five main layers of abstraction as well as the two modules which tie all the modules together The low-level and high-level drivers share a common driver framework, detailed below in Section 4.1.2, which allows easy addition of new drivers and inter-module communication. This chapter will detail the low-level and high-level drivers; the sensor fusion, communications, and data storage code; and the configuration and scheduling frameworks. Chapter 5 details the FCS code and API code, as they are not platform-dependent.

4.1.2 Driver Framework

All driver code, both low- and high-level, conforms to the same standard interface and utilizes a common driver framework. The driver framework is based around the use of a 32-bit GUID, which is composed from four 8-bit identifiers: an 8-bit driver ID, an 8-bit instance data, an 8-bit

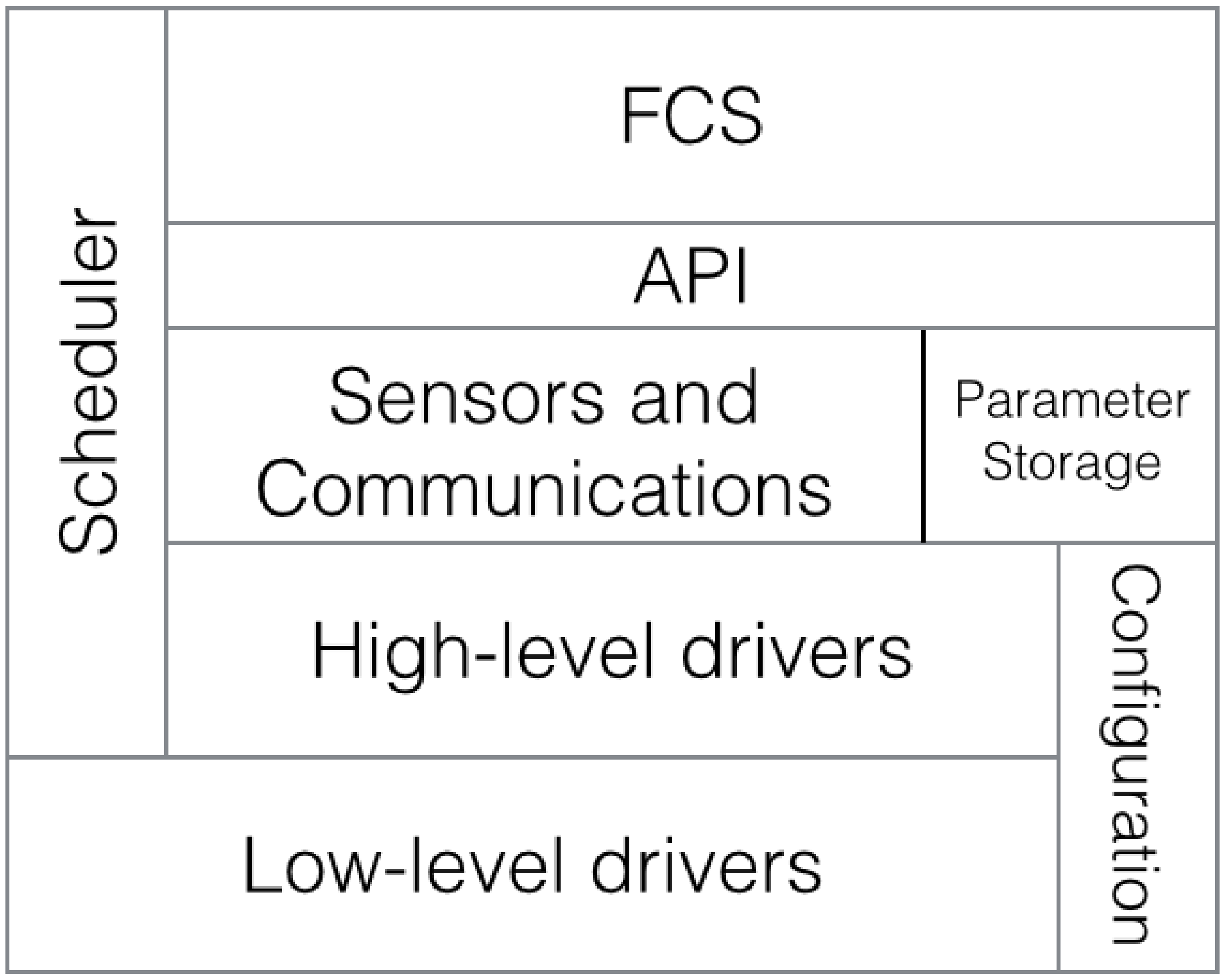

Figure 4.1: Aries Software Block Diagram

subinstance ID, and an 8-bit instance ID. This can be seen in Figure 4.2. Each driver has a unique driver ID which is specific to that driver. The instance data field can be used as needed by any driver; currently it is used in several drivers to provide more detailed interrupt source information when needed. The instance ID and subinstance ID uniquely identify specific instances of a driver. A two-tiered instance ID allows for easier configuration of drivers which have multiple different "channels" per instance. For example, a single Timer instance may have up to four independent channels which must be addressable independently and consistently. In that case, the instance ID corresponds to the timer device itself and the subinstance ID to one of the timer channels.

The use of Driver GUIDs is mainly intended to alleviate some circular dependency issues by

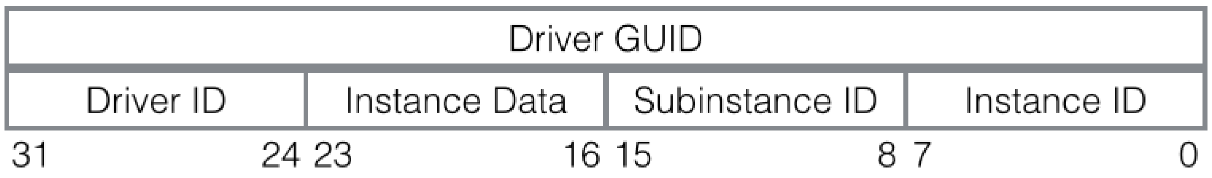

Figure 4.2: Driver GUID format

allowing drivers to be identified by a single number instead of by a pointer to a device structure. By using GUIDs everywhere, callback functions can be set up at compile-time instead of at run-time. As GUIDs are merely 32-bit unsigned numbers, they can be created during preprocessing without knowledge of the underlying driver. This is facilitated by a driver registry system, which maps driver GUIDs to callback functions at run-time. Drivers which support callbacks register themselves during driver initialzation based on their GUID. When this driver receives an event (for example, a “transfer complete” interrupt for a USART), the interrupt handler will notify the driver via the callback map and its Driver GUID. This simplifies somewhat the interrupt handing code, which can just check the validity of the pending interrupt and then trigger a callback, instead of having to deal directly with the inerrupting device itself.

4.2 Low-level drivers

The low-level drivers all correspond directly to a single type of on-chip peripheral core for the STM32F4 processor. Most of the low-level drivers are wrappers around the STM32F4 Standard Peripheral Library, which is provided by ST Micro for the purpose of interfacing to their hardware cores. The Standard Peripheral Library is written to conform to the ARM Cortex Microcontroller Software Interface Standard (CMSIS). Low-level drivers were written to interface with this library in part because of perceived user-unfriendliness and in part to work around bugs in the library code. Some of the low-level driver functions bypass the standard peripheral library and use direct hardware access due to these bugs. All low-level drivers are interrupt or DMA driven. Unlike the higher-level drivers, the low-level are not controlled by the scheduler, although they have the option of triggering a scheduled task in the high-level code in lieu of performing a callback directly in the Interrupt Service Routine (ISR)

4.2.1 Reset and Clock Control driver

The Reset and Clock Control block is responsible for configuring the internal clock dividers and PLL at boot, and subsequently controls the clock and reset options for each internal peripheral in the system. To save power, by default, no peripherals are enabled; in order to use a peripheral, it must first be reset and have its clock inputs enabled, which serves to power on the peripheral core. Responsibility for managing the 5 internal bus clocks and their peripherals is placed on the RCC block.

The RCC driver is used by all other low-level drivers, and contains the necessary interfaces to turn on all integrated peripherals, as well as interfaces to retrieve the current clock frequencies of all peripheral busses. Unlike the standard peripheral library, which has one function for each of the 5 different busses for reset and clock enable, the low-level driver has one function which takes the bus as an argument. This allows for a single driver which has devices on multiple busses (as is the case for almost all core peripherals) to only have one function call which can pass a data argument instead of determining the correct bus and calling different functions. This greatly simplified low-level driver initialization logic.

4.2.2 System Configuration driver

The System Configuration (SYSCFG) block is responsible for fundamental system configuration options, such as bootloader memory remapping, PHY interface mode, and external interrupt configuration. The SYSCFG low-level driver is used only for controlling external interrupts. Each external interrupt from 0 to 15 is connected to all pins on the device, such that EXTI0 can be from PA0, PB0, etc. This of course means that only one pin number can be an external interrupt at a given time (e.g, PA0 and PB0 cannot both be configured as external interrupt lines). The SYSCFG block is responsible for mapping GPIO ports to the EXTI controller, and the low-level driver only provides that functionality.

4.2.3 General Purpose Input/Output driver

The General Purpose Input/Output (GPIO) block is responsible for all off-chip interfacing, at some level. Each GPIO pin can be configured to have up to 15 different Alternate Functions; the inputs and outputs of all on-chip peripherals are routed by the AF selection of the GPIO pins. For example, the clock line for the I2C1 peripheral can go to PB6 or PB8. Which one it is routed to is determined by the configuration for the GPIO controller. The ADC and DAC inputs/outputs are also configured by the GPIO controller, by placing supported pins in “Analog” mode. GPIO pins can also be configured in “input” or “output” mode, as expected.

Additionally, each GPIO pin can be configured for different operating speeds. Each pin also has optional internal pull-up and pull-down resistors, and can be configured in either push-pull or open-drain mode. The low-level GPIO driver supports setting all of this functionality. Unlike the standard peripheral library, the low-level driver does not require a large struct to configure pins; rather, the various pin options are bit-masked into a single 16-bit number. This also allows more easily sharing a common configuration among multiple pins.

4.2.4 Direct Memory Access driver

The STM32F4 provides two 8 channel, 8 stream Direct Memory Access controllers which allow for direct peripheral to memory, memory to peripheral, and (for DMA2) memory to memory transfers without any CPU involvement. All integrated peripherals support DMA transfer. The DMA low-level driver supports all access modes, as well as supporting circular, continuous, and double-buffered operating modes. DMA requests are implemented using a queue system, as it is possible that two peripherals will share a given DMA channel and stream; however, continuous DMA requests will never terminate and thus DMA streams which are currently serving a continuous request will not support queued DMA operations.

Although DMA was initially used for all USART communication, bugs in the STM32F4 DMA controller that resulted in data corruption forced the discontinuation of DMA support in the USART [50, 59]. Currently DMA is only used for ADC transfers; it may be used in the future for other onboard peripherals.

4.2.5 External Interrupt driver

The STM32F4 allows any GPIO pin to be an interrupt trigger, triggered on either rising or falling edge of the input signal. This is used for devices which support a “data ready” signal pin. The EXTI driver, in addition to jumping through all hoops in the system configuration and GPIO drivers to correctly enable external interrupts, supports notifying an arbitrary Driver GUID on external interrupt via the callback mechanism of the driver framework.

4.2.6 Real-Time Clock driver

The Real-Time Clock (RTC) driver handles initializing the on-board RTC peripheral and low-speed oscillator, as well as providing interface functions which translate the RTC’s registers into standard UNIX time (as well as allowing for the setting of the RTC using standard UNIX time). The RTC is synchronized by using the GPS clock as described below, and primarily used for logging purposes.

4.2.7 Universal Synchronous/Asynchronous Transmitter/Receiver driver

The U(S)ART driver for the STM32F4 handles all “traditional” serial port I/O. It is designed to emulate Portable Operating System Interface (POSIX) standards for opening and reading to/writing from serial ports. While a full file-descriptor based abstraction is not implemented, the USART driver provides `usart_open`, `usart_close`, `usart_tcsetattr` and `usart_tcgetattr`, and `usart_read` and `usart_write` functions as the primary interface. The USART driver fully supports different baud rates, as well as blocking or non-blocking I/O, read-only and write-only modes, and minimum read sizes and timeout functionality in blocking mode.

Each USART has independent 2K transmit and receive buffers, and can operate in interrupt driven or DMA mode. Each buffer is treated as a FIFO queue and has a head pointer and a tail pointer; new data is inserted at the tail pointer and removed from the head pointer. Both pointers can be updated to account for a circular buffer design, such that the 2K buffer is effectively a continuously moving buffer. In DMA mode, the receive operation is completely hands-off; once set up, the DMA peripheral continuously reads data into the receive buffer and automatically wraps as needed. Neither the transmit or receive buffers offer any sort of overflow prevention; if the USART cannot write data fast enough, eventually the buffer head pointer will "wrap" past the tail pointer and a large majority of the buffer contents will be lost–in practice, this has never happened.

As mentioned above, DMA mode for USARTs was disabled late in testing as it was discovered that there was a known DMA data corruption issue with one of the onboard DMA controllers. Currently, all USARTs work only in interrupt mode.

4.2.8 Inter-Integrated Circuit driver

The I2C driver for the STM32F4 is one of the most complicated low-level drivers, due in no small part to the numerous silicon defects/shortcomings of the hardware I2C core on the STM32F417 [50]. Among the more interesting "features" of the I2C core is that simply reading the second status register causes the device state to change, which can corrupt a transaction if done at the incorrect time. In addition to the bugs in the I2C hardware, correctly conforming to the I2C specification requires that the driver software handle certain events during the I2C transaction in an incredibly timely manner–the I2C driver is the only low-level driver which requires globally disabling interrupts during some driver operations to work correctly.

As with the SPI core, the I2C driver core is modeled on a transaction request system. A peripheral driver wishing to read from or write to an I2C device will queue a transaction request detailing the I2C address, request type (read, write, or write then read), and length of data for the request. If the I2C driver is idle, the request will be started immediately; otherwise it will be added to the request queue and processed in order. Once a request has started, the I2C start bit is sent and I2C event interrupts are enabled. Upon successful transmission of a start bit, an interrupt is generated and the driver then handles the request based on the type.

Write Request

A write request is somewhat more straightforward than a read request. After successful start bit transmission, the I2C address is transmitted with the read/not write bit reset. At this point, the I2C driver must wait for an acknowledgment signal from the addressed device. A NACK means the addressed device is not on the I2C bus or is not responding; in this case, the driver sets the state of the request to "failed" and the transaction is aborted. Once address acknowledgement has occurred, the driver sends the written data one byte at a time until all bytes have been written. This is driven by a "transmit buffer empty" interrupt. After all data is transmitted, the driver sends a stop bit, performs a callback notification for the transaction, and starts the next transaction in the queue.

Read Request

A read request is much more complicated, because the STM32F4 I2C core requires different software handling when reading 1, 2, or more than 2 bytes from the I2C bus. When reading only one byte, the core must be instructed to send a stop bit immediately after receiving the data byte, which means it must be configured to do so between the receipt of the address acknowledgment and the receipt of the first bit of the incoming data byte. As such, this is one section of the I2C driver code which globally disables interrupts.

When reading two bytes, the core must be instructed to send a single ACK and then a stop bit, for the first and second byte respectively. This requires critical sections both in the start bit generation and the address acknowledgment phase, as well as the misuse of the shadowed shift register architecture of the I2C core. While the I2C core only has one receive data register, it can actually store two bytes at once: one in the data receive register and one completed byte in the receive shift register. In order to correctly receive two bytes (and only two bytes) without putting the I2C core in an unusable state, it is necessary to wait for both bytes to be received, generate a stop bit, and then perform two successive reads to the I2C data register, which will actually read the data register and then the shift register.

Reading more than two bytes is somewhat less complicated than reading 1 or 2 bytes. In order to read n bytes and still generate the correct NACK and stop bit timing, it is necessary to use the same trick of reading two bytes at once as with reading two bytes. However, it is not the last two bytes which are read this way, but instead the $n-2$ and $n-1$ bytes.

Write then Read request

A write then read request is most commonly seen in devices which support register addressing. The transaction for reading from these devices involves writing the register to the I2C bus, and then performing an I2C restart, which drives another start bit onto the bus instead of a stop bit. The implementation of this in the I2C low-level driver is simply to switch a write then read request to a read request after the write has completed, after ensuring a restart condition instead of a stop condition.

4.2.9 Analog to Digital Converter driver

One of the challenges encountered in developing the ADC driver is the speed at which the ADC sample is completed. Even at the worst case sampling time, the ADC can generate a new sample every 20 kHz; as the ADC is not buffered, sampling multiple channels would then require an interrupt firing at that frequency. Two options were considered: an “on-demand” read which only sampled the ADC channel when requested, in a blocking fashion, or a “continuous” read which leveraged the DMA abilities of the ADC core to continuously read all ADC channels and store their latest values in memory. The latter option was chosen because it makes the ADC operation effectively hands-off. Once the ADC has been initialized and the conversion sequence set up, the DMA controller will constantly update the ADC values in memory without any interaction with the main processor.

Moving Average filter

During testing, it was discovered that the ADC inputs were fairly noisy. Figure 4.3 shows the noise variance from a constant voltage input to the FCS battery monitoring channel of the ADC. Without any filtering, the ADC sample value varies between 2311 and 2336. Using Eq. (4.1), this gives a variance of 20mV from a constant power supply with a much lower ripple.

$$v_{LSB} = \frac{v_s}{4096} \tag{4.1}$$

$$v_{LSB} = \frac{3.3\text{V}}{4096} \tag{4.2}$$

$$v_{LSB} = .806\text{mV} \tag{4.3}$$

x

In order to combat some of the ADC input noise, a 25 window moving average filter was implemented for the ADC inputs. This was done by expanding the size of the DMA ADC buffer to be 25 times larger, effectively creating a two-dimensional array holding the last 25 ADC samples for each channel. When a channel value is requested by a higher-level driver, the average of the last 25 samples is computed and returned. The results of a moving average filter can also be seen in Figure 4.3. The moving average filter reduced the ripple down to 4 mV.

More drastic noise was seen in the barometric sensors. As seen in Figure 4.4, the noise range on the absolute pressure sensor input varies from 3627 to 4029–a range of 324mV. Even with a moving average filter, the noise is only reduced to a 44mV ripple. The source of this noise is likely due to a routing issue on the PCB. Unfortunately, the barometric sensor was too noisy to use in flight, resulting in the use of GPS altitude for all flight tests.

4.2.10 Timer driver

STM32F4 timer peripherals have many different operating modes–the timer driver in the Standard Peripheral Library is the most complex driver provided. The low-level timer driver only

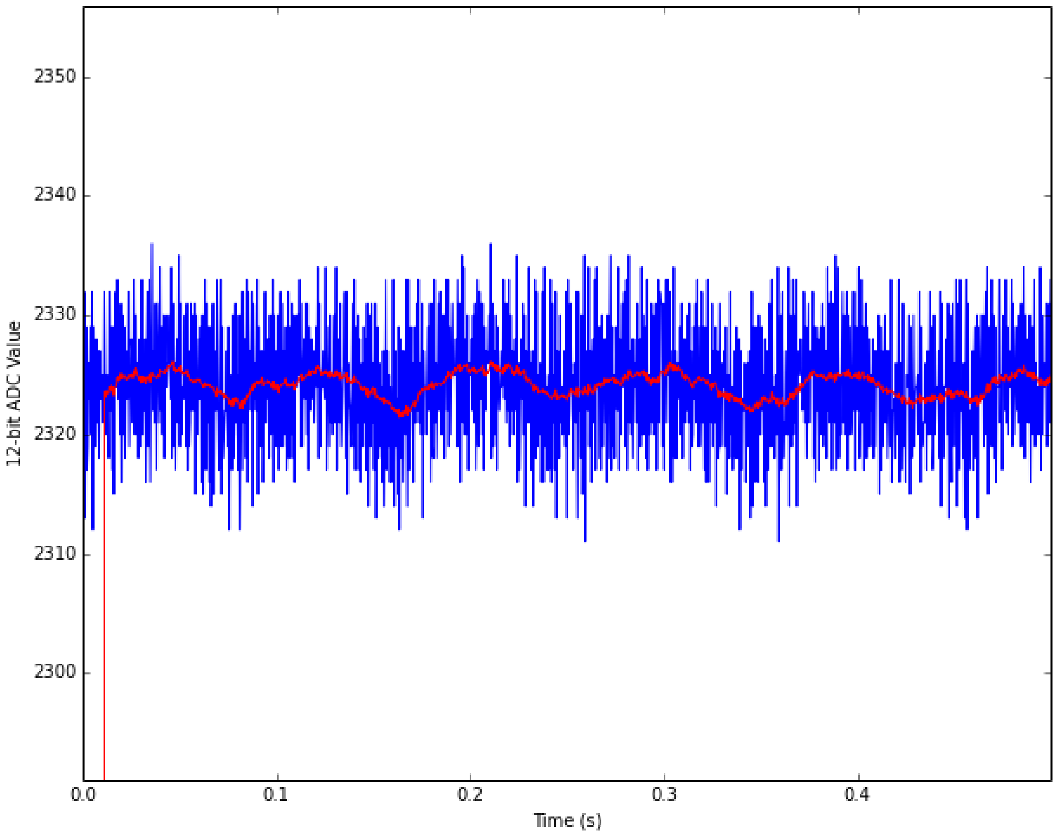

Figure 4.3: FCS Battery Monitor ADC noise

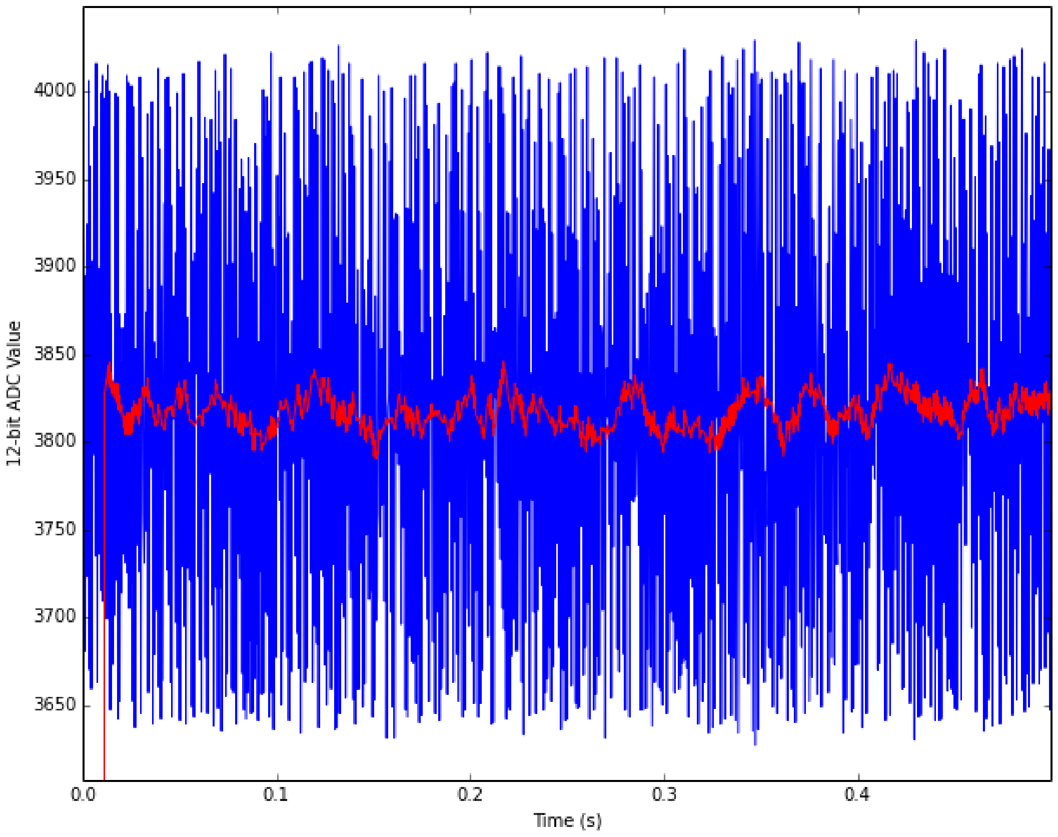

Figure 4.4: Absolute Pressure Sensor ADC Noise

supports output compare mode and normal counter mode.

Output compare

Output compare functionality is implemented in a manner that makes using the timer channel to output a PWM signal trivial. For output compare mode, two options are needed: the output prescaler and the output period. The prescaler determines how often the timer “ticks”, and is derived at run-time based on the peripheral bus clock rate. For the output compare functionality, the prescaler is set such that the timers tick at one μs. The period determines how often the timer rolls over and restarts–in output compare mode, the period roll-over determines when the rising edge of an output pulse occurs, while the output compare value determines the number of ticks after this that the falling edge occurs. For PWM output at 50 Hz, the period is set to 19,999, which gives 20,000 ticks per period. The output compare value can then be loaded as the pulse width in μs directly, which greatly reduces software complexity–by merely loading the desired pulse width into the output compare register the desired pulse width will be output.

Counter

Counter functionality is designed primarily for having a long-running microseconds counter running for use by the rest of the driver software and user software. As the 16-bit counters will overflow every 0.07 seconds, the counter functionality is only really useful for the two 32-bit counters, TIM2 and TIM5. Currently, TIM2 is used to provide a microseconds counter; it is set such that it overflows once every $3,600,000,000\mu s$–once an hour. There is currently no overflow detection; applications which must have a monotonic clock are required to check for and account for overflow between two consecutive microsecond clock samples.

4.3 High-level drivers

High-level drivers interface external peripherals using the low-level drivers. Most high-level drivers are implemented as some kind of parsing finite state machine which reads data using a low-level driver, parses it based on the peripheral's data format, and provides a number of access functions which allow the user-facing code to access the data from those peripherals. The high-level drivers can be triggered either via external interrupt (using the callback mechanism and the EXTI driver), completion of low-level driver operation (using the callback mechanism), and/or at a specific time interval (using the scheduler). Most of the more complex high-level drivers (those which involve a large amount of data parsing) are implemented only as periodic tasks. However, some drivers (such as the MPU driver) are purely interrupt-driven; the devices in question support a "data ready" signal which triggers a driver update asynchronously. Some drivers even support both; they start a sample with a periodic task, and then use interrupts to know when that sample has finished.

4.3.1 Console driver

The console driver is a simple driver which allows for a debugging console for printf()-style debugging and informational messages. Writes which occur before the console is properly initialized are stored in a buffer and written out to the console USART during initialization.

4.3.2 Modem driver

As with the console driver, the modem driver is a simple driver which is just a wrapper around reading from and writing to a single USART. It is in the current form unnecessary, but included as a separate driver to allow the future use of either UDP over Ethernet, or to allow for the implementation of the Xbee or similar framing protocol[3] to allow for multiple simultaneous UAV flight without relying on the MCS for communications.

4.3.3 PWM driver

The PWM driver is designed to map PWM outputs to timer channels, which in turn are connected to servos which drive the aircraft surfaces. As mentioned above, the timer driver has been written to allow using a raw PWM pulse with in μs as an input. The concerns of the PWM driver are two-fold: to map servo channels 0, 1, 2, 3, etc. to timer channels, and to translate normalized values in the range -1.0 to 1.0 to actual pulse widths in μs and vice-versa. Forward interpolation is done linearly. Assuming n_{min} and n_{max} as the minimum and maximum normalized values, and p_{min} and p_{max} as the minimum and maximum pulse widths, for an input n the output p is calculated like so:

$$p = (p_{max} - p_{min}) \cdot \left(\frac{n - n_{min}}{n_{max} - n_{min}} \right) + p_{max}$$

Reversing this interpolation, which is used to translate manual PWM inputs from the safety switch to normalized values for the FCS code, is done the same way with p and n swapped:

$$n = (n_{max} - n_{min}) \cdot \left(\frac{p - p_{min}}{p_{max} - p_{min}} \right) + n_{max}$$

4.3.4 NMEA driver

The National Marine Electronics Association (NMEA) 0183 standard defines a standard information interchange format between various marine sensors, including GPS units. Several commercial GPS units are available which output NMEA sentences.

The NMEA sentence format is an ASCII string, starting with an ASCII '\$' and ending with a '*', a two-digit checksum, and an ASCII CRLF. NMEA checksums are calculated by taking the bitwise XOR of all characters in the sentence between the '\$' and '*' delimiters. Each sentence includes a unique message identifier immediately following the '\$' character which consists of 4-8 ASCII characters. After the identifier, individual fields are listed separated by a ',' character.

The NMEA driver currently supports 5 different NMEA sentences:

- GPGGA - time, position, and GPS fix information
- GPGSA - satellite information
- GPRMC - minimum navigation information
- GPVTG - ground track information
- GPGSV - detailed satellite information

Each sentence is parsed the relevant position data stored in a GPS state structure. The STM32F4's RTC is updated from GPS time, if GPS time and RTC time differ by more than 30 seconds. Additionally, NED velocities are calculated from ground track and heading, as well as by calculating the rate of change of GPS altitude:

$$\begin{aligned} v_n &= s * \sin\theta \\ v_e &= s * \cos\theta \\ v_d &= \frac{(h_c - h_l)}{dt} \end{aligned}$$

where s is the ground speed, θ is the heading normalized between -180.0 and 180.0, h_c is the current altitude, h_l is the last altitude, and dt is the time delta between when h_c and h_l were sampled.

Automatic baud rate detection

As there is no one standard for NMEA device baud rates, a simple detection algorithm was implemented which attempts to automatically determine the baud rate of the attached NMEA device. The algorithm is straightforward: for each possible baud rate, the driver reads enough data for 2 NMEA sentences and tries to parse it. If no valid NMEA sentences were found, the next baud rate is tried, otherwise the current baud rate is assumed to be correct. Additionally, a timeout feature is implemented such that if no data is received (e.g. due to a misconfigured USART port) the algorithm will fail gracefully.

GlobalTop configuration support

The GlobalTop PA6H GPS module present on the Aries board supports a limited set of configuration sentences which allow customization of the baud rate, update rates for the supported messages, and enabling DPGS support. The NMEA driver attempts to write a baud rate change command to the NMEA GPS if a 9600 baud device is detected, as the default baud rate of the PA6H is 9600 baud. If the driver initialization can successfully change the baud rate to 57600, the driver will then send configuration commands to set the GPS to 10 Hz position updates and enable DGPS mode.

4.3.5 MIDG2 driver

The Microbiotics MIDG2 is an all in one GPS/IMU solution. Although the Aries has an on-board GPS and an on-board IMU, during initial flight testing and Hardware-in-the-Loop (HILS) testing it was necessary to use the MIDG2 for an IMU solution. This allowed for tuning the flight control code independently of tuning the IMU filter algorithms.

The MIDG2 binary format consists of two sync bytes, an ID byte, a length byte, up to 255 payload bytes, and a two byte checksum using the same 16-bit Fletcher checksum used in the VACS protocol. As the MIDG2 was only used for IMU data, and not GPS data, only the NAV_PV and NAV_SENSOR packets are handled by the MIDG2 driver on the Aries board. This gives attitude, acceleration, and attitude rates, as well as NED velocities.

4.3.6 MPU6000/9150 driver

The MPU6000/6050/9150 are almost identical devices; the primary difference being that the MPU6xxx series lacks an on-die magentometer and only the MPU-6050 supports SPI interfacing in addition to I2C. Only I2C mode is currently supported by the MPU driver, but otherwise it is capable of communicating with all three devices equally well.

By default, the MPU is in a power-down state. Any attempts to read any of the internal registers will return all '0's, except for the PWR_MGMT_1 register. The PWR_MGMT_1 register must be written to in order to power up the MPU; once it has been written with a correct power and clock configuration command the rest of the device may be configured and accessed. The MPU series can support several different clock sources, using both internal PLLs as well as supporting an external clock source. For the Aries board, the MPU is clocked using an internal PLL which uses the gyroscope Z axis. Additionally, the initialization sequence configures the accelerometer and gyroscope range to be $\pm 4g$ and full range respectively.

The MPU driver is completely interrupt driven. The MPU samples new data at a configurable rate, and asserts an interrupt pin on the completion of each sample. This interrupt is connected to an EXTI interrupt line, which invokes the MPU driver's read command. After reading in the 14 byte accelerometer, temperature, and gyroscope data, it is unpacked into the raw X, Y, and Z values, which are signed 16 bit numbers from the MPU's built-in ADC. To extract X axis accelerometer values, the following equation is used:

$$A_x = \frac{A_{x,raw}}{32768} * A_s$$

where $A_{x,raw}$ is the raw ADC reading and A_s is 1, 2, 4, or 8 g depending on the device configuration; for the Aries board it is $4g$. The same equation is applied to A_y and A_z. For gyroscope data, the X axis equation is:

$$\begin{aligned} G_{x,scaled} &= \frac{G_{x,raw}}{32768} * G_s \\ G_x &= G_{x,scaled} * \frac{\pi}{180} \end{aligned}$$

where $G_{x,raw}$ is the raw ADC reading and G_s is 250, 500, 1000 or 2000. For the Aries board, $G_s =$ 2000. The scaled value $G_{x,scaled}$ is then converted from degrees to radians. The same procedure is applied to G_y and G_z.

4.3.7 AK8975 driver

The MPU9150, as mentioned above, ships with a separate on-die magentometer, the AK8975. Like the MPU9150, the AK8975 is an I2C device and is available on the same I2C bus as the MPU9150. Unlike the MPU9150, the AK8975 does not support continuous sampling mode; it is necessary to send a “start sample” command and either wait 10 ms for the sample to complete or use the data ready interrupt pin as with the MPU. As with the MPU, the AK8975 driver is interrupt driven. During the initialization phase, a “start sample” command is send. Once the sampling is done, an interrupt pin will be asserted and trigger an external interrupt in the STM32F4. This interrupt will cause the AK8975 to read the new samples and then start a new sample, thus continuously sampling as fast as possible.

Once a sample has been read, it must be scaled from the ADC values into a value in gauss. Each channel is converted like so:

$$M_x = M_{x,raw} \cdot \left(\frac{(C_x - 128) \cdot 0.5}{128} + 1 \right)$$

where $M_{x,raw}$ is the ADC value and C_x is the calibration value for that axis. M_y and M_z are calculated identically.

4.3.8 Safety Switch driver

Interfacing with the safety switch has two main concerns: monitoring the RC loss and Auto/Manual signals from the safety switch, and reading the captured PWM input values from the safety switch. Both Auto/Manual and RC loss signals are connected to GPIO pins which are set up as external interrupts. When either changes, the safety switch driver is notified. It will then read the pin value and, if the state has changed, update its stored value accordingly. The safety switch can optionally perform a callback immediately when one of these pins changes, but as of now this functionality is unused.

Reading PWM input values is done via SPI. The safety switch supports a data ready pin which is asserted whenever the stored PWM capture values change. When this signal is asserted, the safety switch driver is notified and begins a new SPI read command to read all 8 supported PWM channels. Update frequency for the PWM values is governed by the input to the switch, but is most likely 50 Hz based on commonly available RC receivers.

4.3.9 25LC512 EEPROM driver

The 25LC512 provides the back-end for permanent parameter storage on the Aries board. It supports arbitrary length reads and writes across the entire device. Unlike most of the high-level drivers, the 25LC512 driver supports a request queuing system similar to that used with the SPI and I2C low-level drivers. This allows for multiple requests to be queued and processed one at a time.

Read requests

Reading data from the 25LC512 is fairly straightforward. A read command byte is transmitted, followed by the 16-bit address in the EEPROM to start reading from. At that point, a read will read the byte stored at that address. Subsequent read operations will read from the next address until either the chip select pin is asserted (in SPI mode) or a NACK followed by a stop condition is placed on the bus (in I2C mode).

Write requests

Writing data to the 25LC512 is somewhat more complicated. The default page size in the 25LC512 is 128 bytes, and a write request can only write one page at a time. If a write length attempts to write past a page boundary, the write is instead redirected to the beginning of the page, potentially overwriting previously written data. In order to prevent that, it is necessary in software to split up a write which would cross page boundaries into multiple writes.

Write requests are therefore split up as needed. Each request will write from the current address (initialized to the start address of the write) up to a page boundary. The total length to write is also used, and initialized to the initial length requested. After each write, the current address is incremented by the number of bytes written; the current length is decremented by the same amount. Once the current length is 0, the request is finished.

4.3.10 Barometric Pressure Sensor drivers

For the initial revision of the board, the same barometric pressure sensors used on the miniFCS were used [2].

MP3H6115A Static Pressure Sensor

As with the miniFCS, the driver for the MP3H6115A uses an equation which directly maps ADC values to altitude in feet. This equation was calculated in several steps. The transfer function for the MP3H6115A, shown in Eq. (4.4), gives the output voltage of the sensor based on the absolute pressure in kilopascals P and supply voltage v_s

$$v_o = v_s(0.009P - 0.095) \tag{4.4}$$

Solving for pressure P yields Eq. (4.5).

$$P = \left(\frac{\frac{v_o}{v_s} + 0.095}{0.009}\right) \tag{4.5}$$

The gain/offset stage of the device takes v_o and scales it from its nominal range at appropriate pressures to the full range of the ADC. With gain $g = 18.7$ and offset $o = 2.536$, the voltage output for a given v_o can be expressed as:

$$v_{gain} = g(v_o - o) \tag{4.6}$$

This can be mapped into an ADC value given the range of the ADC as:

$$s = \frac{v_{gain}}{3.3V} \cdot 4096 \tag{4.7}$$

Pressure varies with altitude according to

$$P = 101.325 \cdot (1 - 0.000022557h)^{5.25588} \tag{4.8}$$

This can be solved for h to translate pressure to altitude:

$$h = 1 - \frac{\left(\frac{P}{101.325}\right)^{\frac{1}{5.25588}}}{0.000022557} \tag{4.9}$$

Using Equations (4.8), (4.4), (4.6), and (4.7), a table mapping altitude to ADC values was created. A linear regression on that data was performed to find Eq. (4.10).

$$h = -0.404022 \cdot s + 1494.53591 \tag{4.10}$$

Equation (4.10) gives a direct mapping between ADC sample values and altitude in feet.

MP3V5004G Differential Pressure Sensor

The transfer function for the MP3V5004G, shown in Eq. (4.11), gives the output voltage of the sensor based on the supply voltage v_s and the difference in the pressure ports P. This can be solved for P, shown in Eq. (4.12).

$$v_o = v_s \cdot (0.2P + 0.2) \tag{4.11}$$

$$P = \frac{\frac{v_o}{v_s} - 0.2}{0.2} \tag{4.12}$$

Airspeed can be calculated from the differential pressure P

$$a = 661.48 \cdot \sqrt{5 \cdot \left(\left(\frac{P}{101.325} + 1 \right)^{\frac{2}{7}} - 1 \right)} \tag{4.13}$$

4.3.11 Battery Monitor drivers

Each of the batteries connected to the Aries board is also connected through a voltage divider to an ADC input, so that the voltage of the batteries can be monitored. To translate ADC samples to battery voltages, the battery monitor stores the R1 and R2 values for the voltage divider and calculates the voltage of the battery like so:

$$V_{batt} = \left(\frac{s}{4096} \cdot 3.3V \right) \cdot \left(\frac{R1 + R2}{R2} \right)$$

where s is the ADC sample value.

4.4 Configuration System

The Aries software has an integrated configuration system which allows for run-time configuration of most low-level and high-level drivers. This makes re-targeting the Aries software to different boards with different STM32F4 and on-board peripheral configurations much easier. As a given board's design tends not to change over time, this configuration data is stored as part of the application code at compile time. Additionally, the configuration system utilizes the onboard F25LC512 EEPROM to store board-specific configuration data which may change over time. This data may be accessed and changed at run-time, unlike the board configuration data itself. For consistency, the compile-time configuration data is referred to herein as the "board configuration" and the run-time, EEPROM stored configuration as the "parameter configuation."

4.4.1 Board Configuration

One of the downsides of the otherwise useful alternate function system for the STM32F4's GPIO pins is that one does not always know which pins, for example, the I2C1 core will be connected to. This precludes a driver automatically mapping GPIO pins correctly. Additionally, depending on the board, a peripheral may be connected to a different I2C core to make board routing easier.

As such, a system for describing the connections between low-level and high-level drivers, as well as configuring both of them, was desired. Several options were considered. Some were compile-time options which would update header files or provide preprocessor definitions to correctly configure pins and devices. This would likely have been implemented as a pre-preprocessor, written in Python, which would create the apporopirate header files based on an XML or JSON-formatted configuration file before compilation. A basic prototype of this system was written, but was scrapped due to complexity.

Further research into prior implementations of board-specific configurations led to the device tree format used by the Linux Kernel [60]. First developed by IBM for running Linux on the Power architecture [61], it has also been adopted by Linux ports for ARM processors as well [60]. The device tree format is a simple text-based tree format, which can be compiled into a binary device tree blob format. This is ideal, as it allows writing a human-readable text file to configure the board, which is then "compiled" into a binary format suitable with a small memory footprint. This binary format is designed to be easily "walked" and have configuration data extracted at run-time.

A simple device tree parsing/walking library, libfdt, is freely available [62]. This allows working with a binary Flattened Device Tree (FDT) easily. The code is fairly small as well (TODO find binary size). Libfdt is compiled separately and linked as a static library. This allows independent updating of the library. The FDT is stored as a static array which is loaded into the executable image at compile-time; a script is provided which takes a binary FDT and converts it to a C array representation.

Device Tree Format

As mentioned, the device tree is a simple text format. A minimal working example is shown in Source Code 4.2. Each node has a name and is delimited by "{" and "};". The "root" node has the special name "/". All other nodes can be named arbitrarily, but are typically named "device@address"; additionally, the name must be unique among other subnodes at the same level. Note that the device tree format was intended to be used for memory mapped peripherals. This leads to some potential confusion as the implementation here does not use memory addresses, but instead uses 0-based indexing for instance and subinstance IDs. Most references to "addresses" that follow are actually various parts of the Driver GUID that will be constructed for each device.

Each node can have subnodes, as well as properties. Properties are defined by the format as being either 32-bit unsigned numbers, or C-style strings. There are several "special" properties which must be present for all subnodes. The "#address-cells" and "#size-cells" properties which tell how big the address space for subnodes is. A node that has subnodes must have both of these properties. For this implementation, all nodes have "#address-cells" as "1" and "#size-cells" as 0; this means that the "address" of the subnodes has one address entry and 0 size entries. The subnode address is stored in the special "reg" property. Additionally, each subnode must contain a "compatible" property of the form "vendor,driver". The "driver" element of this property will be used to determine the Driver ID and configuration function for a given device node. Additional properties prefixed with "stm32," will be passed to the driver code in an array, in the order they are presented in the node list. To configure each driver, a configure function is called which gives each driver a parent GUID, its driver GUID, and those configuration parameters. This prototype is seen in Source Code 4.1.

```c
void configure_fn(uint32_t parent_guid,
                  uint32_t parent_guid,
                  uint32_t *config,
                  uint32_t length);
```

Source Code 4.1: Device Configuration Function Form

```
/dts-v1/;
/ {
        #address-cells = <1>;
        #size-cells = <1>;
        apb1: apb@40000000 {
              #address-cells = <1>;
              #size-cells = <0>;
              reg = <0x40000000 0x7FFF>;
              compatible = "stm32,apb1";
              I2C1: i2c@0 {
                    #address-cells = <1>;
                    #size-cells = <0>;
                    compatible = "stm32,i2c";
                    reg = <0>;
                    stm32,clock-speed = <88000>;
                    stm32,own-address = <0x00>;
                    stm32,scl-pin = <0x05000601>;
                    stm32,sda-pin = <0x05000701>;
                    sensor@0 {
                           compatible = "invensense,mpu";
                           reg = <0>;
                           stm32,i2c-address = <0x68>;
                           stm32,int-pin = <0x05000101>;
                    } ;
              } ;
        } ;
} ;
```

Source Code 4.2: Minimal Device Tree

The minimal example in Source Code 4.2 demonstrates most of these properties. At a high level, this tree specifies an MPU9150 device, which is connected to an I2C device. This I2C device is on the APB1 bus; however, this information is currently unused by the configuration. The I2C node is named "I2C1@0", indicating it is the I2C1 device, and has instance ID 0. Using the Driver ID derived from its "compatible" property and the Instance ID derived from the "reg" property, a Driver GUID will be constructed. The Parent GUID will be the address of the APB1 bus, but it is unused. In addition, there will be four configuration options passed to the I2C configuration. As the MPU is a submode of I2C1, it will be passed the Driver GUID of I2C1 as the parent GUID. Its Driver GUID will be derived in the same manner as the I2C driver GUID was. It also has two properties which will be provided to the configuration function.

Device Tree Parsing

At boot, the device tree is parsed and all peripherals are configured. This is done by “walking” the tree, starting with the root node and parsing all subnodes in depth-first order. As the tree is parsed, Driver GUIDs are built up from the driver type and driver address of each node, and each driver is configured appropriately. Both low-level and high-level drivers are configured by the device tree parser. An invalid device tree will cause the device to enter a hard fault state, as a board without valid configuration will not perform correctly.

4.4.2 Parameter Configuration

For user-facing applications which use the on-board 25LC512 EEPROM, a key/value store has been implemented to add a layer of abstraction and convenience. This uses the 25LC512 as its backing store, but could easily use other EEPROMs or even a flat file if needed. The key/value store is implemented as a Look-up Table (LUT) which maps 16-bit unsigned keys to an address and length in the EEPROM itself.

Assuming the 128 byte page size of the 25LC512, the parameter storage driver uses it for the following:

- The first 2 pages are reserved for the parameter flash configuration block
- The next 6 pages are reserved for the system configuration block
- The next 24 pages are reserved for the LUT.
- The remaining 480 pages are used for parameter storage or other use.

The actual addresses of all blocks except the parameter flash configuration block are stored in the parameter flash configuration block.

Table 4.1: Parameter Flash Configuration Block

Offset	Size	Description	Value
0	4	Flash ID	`0xbeefdead`
4	2	System Configuration Block Address	varies, default `0x0100`
6	2	System Configuration Block Length	varies, default `0`
8	2	LUT Address	varies, default `0x0400`
10	2	LUT Length	varies, default `0`
12	2	LUT Entries	varies, default `0`
14	2	Parameter Address	varies, default `0x0B00`
16	2	Parameter Length	varies, default `0`

Offset	Size	Description
0	2	16-bit unsigned key
2	2	16-bit address
4	2	16-bit length

Table 4.2: Parameter LUT Entry

Paramteter Flash Configuration Block

The parameter flash configuration block contains information about the flash and where in flash all other entries are placed. All entries in the parameter flash configuration block can be seen in Table 4.1. The Flash ID entry is a unique signature that is used to validate that an attached EEPROM has a valid configuration block; if this signature is not preset, a new configuration block is written with all default parameters. Addresses for the system configuration block, LUT, and parameters are also stored in the flash configuration block. Defaults for these values are given in Table 4.1, based on the 25LC512 default page size of 128 bytes.

System Configuration Block

The system configuration block is 6 pages (768 bytes for the 25LC512 used) which is exposed to the rest of the system for arbitrary use. Functions are provided to read the block and write a new one. Currently the system configuration block is unused; it is provided for future use.

Parameter LUT

To find entries stored in the flash, a 512-entry Look-up Table is used. Each entry in the LUT, shown in Table 4.2, has three parts: the unique key for the entry, the address of the entry in the flash, and the length of the entry. The LUT is cached in RAM, and updated on the flash whenever a new entry is inserted or changed.

To find an entry in the LUT, a binary search algorithm is used. In order for this to be effective, the LUT must be sorted by the key value; this is accomplished by using the binary search algorithm to determine the index in the LUT for a new key, shifting the rest of the entries in the LUT down one, and inserting the new entry such that the LUT is always sorted by key. The binary search algorithm is implemented as an iterative binary search to decrease the risk of stack problems. Each iteration of the algorithm halves the search space until the search space is 1 and an index has been found whose key is less than the search key. This index is the correct index for the key; it may or may not already have the key stored there.

Parameter Flash Operation

The parameter flash driver exposes read key and write key methods to the user code; these are used to look up or write entries in the flash. For the Aries software, the key used is a VACS message ID, and the entries are valid VACS packets which have been sent or received containing configuration options. Other data can be stored so long as the key does not conflict with one of the VACS packets being stored; it would be safer to use the system configuration block for these purposes.

4.5 Scheduler

Scheduling tasks is handled by a priority-based queue system. There are 8 possible priority levels, PL0 through PL7, with PL7 being highest priority and PL0 lowest. Each priority has its own queue, which can hold up to 16 tasks per priority level. At each “tick” of the scheduler, each queue starting with PL7 is iterated. Tasks which are scheduled are executed and then pushed back onto a temporary queue; this queue is then copied to the PL queue once that queue is empty. This repeats for all 8 priority levels.

The priority level system guarantees an ordered execution at each tick. Sensor processing code should be executed before the FCS code, for example, so it has a higher priority, while the ground communication reports should be sent after the FCS has run, so they have a lower priority. A full list of the priority levels and their tasks is shown in Table 4.3.

Priority Level	**Usage**
PL7	System Heartbeat Indicator, high priority one-off tasks
PL6	Unused–reserved
PL5	Periodic driver updates and communication receive updates
PL4	Sensor fusion code (IMU)
PL3	FCS code
PL2	Ground communication reports
PL1	Unused–reserved
PL0	Low-priority one-off tasks

Table 4.3: Priority Levels in the system

The task system does not provide any sort of preemption or overrun detection/prevention. Tasks which do not finish before the next tick will overrun and cause the next tick to be missed. However, in testing, this has not been a problem. Figures 4.5 and 4.6 show the processor utilization over a single second. In Figure 4.5, the IMU code is disabled. Only the FCS and hardware algorithms are running. As can be seen, the maximum usage of the processor is around 10%, typically less. Spikes can be seen at 50 Hz, corresponding to the hardware sensor updates; 5 Hz, corresponding to communications updates, and 2 Hz, also corresponding to communications updates.

Figure 4.6 shows the percentage utilization of the processor when the IMU code and Kalman Filter are running. This results in a major increase in cycles used. However, the total utilization does not go above 40%. Additionally, these benchmarks were taken using an unoptimized version of the Kalman Filter algorithm. At the time of this work, it does not seem necessary to implement the overhead necessary for context switching, as the current system seems capable of running all necessary tasks at a tick rate of 200 Hz. If a faster "tick" or more complex processes are desired,

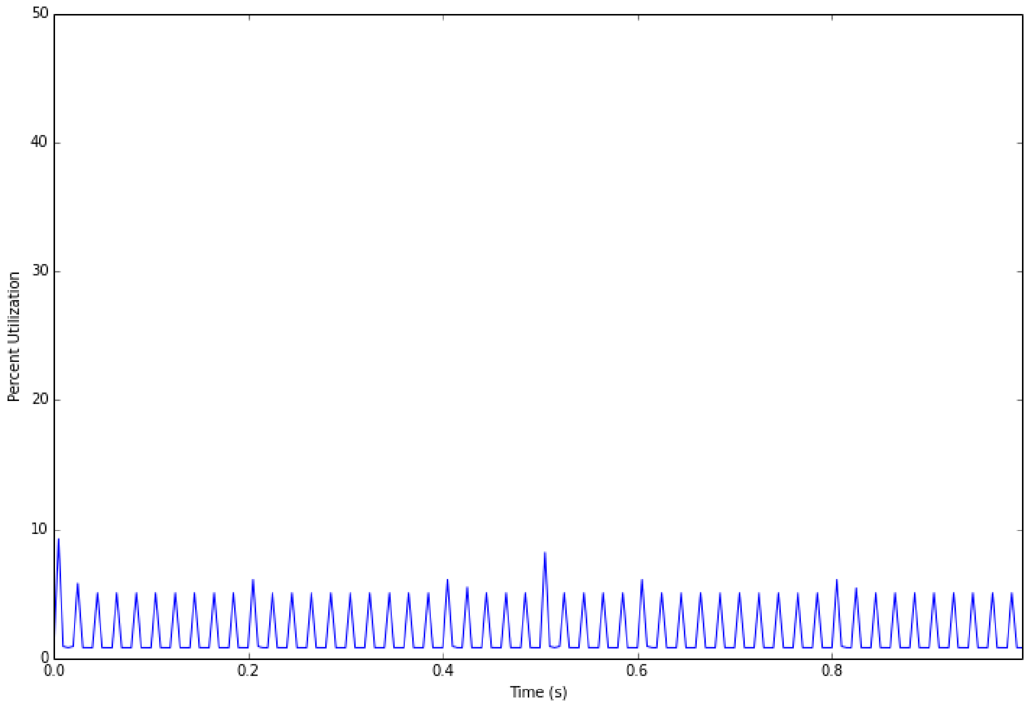

Figure 4.5: Processor utilization, without IMU code

though, this additional functionality may need to be added.

Additionally, the scheduler does not disable interrupts during its execution, and interrupts all have higher priority than any scheduled task. This is one of the motivating factors behind allowing driver code to have scheduled updates instead of being purely interrupt driven. Drivers which need to parse data can do so in a more time-predictable way. However, some drivers are still largely or completely interrupt based and may interrupt the execution of primary code at any time. Again, in current testing this has largely proven to be a non issue. Adding periodic updates to the higher-speed drivers (such as the MPU, which is generating new samples faster than the tick rate) would require a faster tick rate and context switching support. For now, the interrupt overhead is low enough that this is not a problem, but again this should be monitored in the future.

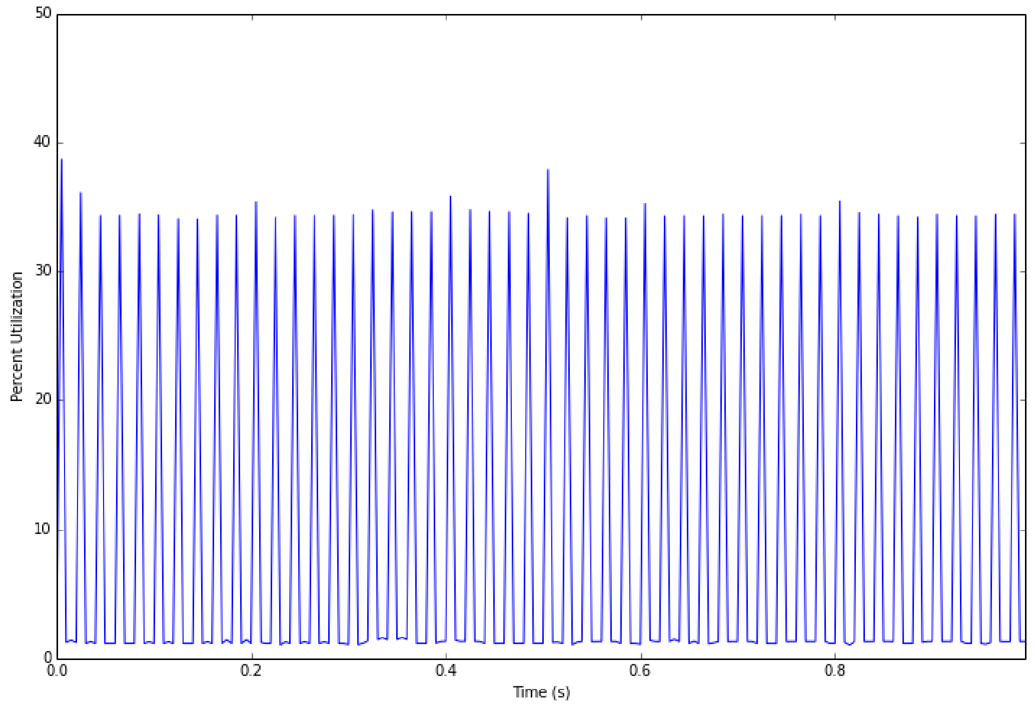

Figure 4.6: Processor utilization, with IMU code

4.6 Sensor and Communication Systems

4.6.1 Sensor Systems

The sensor systems encompass the systems which take the data provided by the high-level drivers and convert them into a usable aircraft state that is used by the API and FCS code. This predomanantly covers the Kalman Filter algorithms which take accelerometer, gyroscope, and magentometer data and determine the aircraft attitude.

4.6.2 Communications System

The communications subsystem handles both sending and receiving data. All data is packaged using the VCU Aerial Communications Standard (VACS) format, which was created at VCU to be a simple packet format for plane-to-ground and plane-to-plane communication. Incoming communication is received asynchronously and stored in a read buffer in the USART driver. This buffer is processed at a 50 Hz rate; data is processed until the read buffer is empty using the VACS Framework outlined below. Additionally, the communications system handles sending report packets to the ground station at a configurable rate. Three types of reports are sent out, each with an independently configurable send rate: standard reports, hardware reports, and FCS reports. Standard reports are basic aircraft state information: position, velocity, and attitude. Hardware reports are more detailed hardware information: GPS information, barometric sensor information, battery voltage levels, and detailed IMU information. FCS reports are generated primarily by the FCS code itself; in addition, the current PWM values from the FCS and from the inputs to the safety switch are reported.

4.6.3 VACS Framework

The VACS framework implements a parser for the VACS protocol, as well as method for dispatching received message based on their message ID.

Protocol

The VACS protocol is a simple serial packet format which contains an eight-byte header, a variable length payload, and a two-byte checksum. All VACS packet data, both header fields and data fields, are little endian. The VACS header consists of two sync bytes, the destination and source addresses, a two-byte message id, and a two-byte length. Although the header supports lengths up to 65,536 bytes, a maximum length of 1024 bytes of payload data has been imposed; no existing VACS packet contains enough data to even fill this. VACS checksums are 16-bit Fletcher checksums, calculated over the header (excluding the sync bytes) and payload data. The full VACS packet format can be seen in Table 4.4.

Index	Field
0	Sync 1 Byte (`0x76`)
1	Sync 2 Byte (`0x63`)
2	Destination Address
3	Source Address
4	Message ID (LSB)
5	Message ID (MSB)
6	Length (LSB)
7	Length (MSB)
8	Data (N Bytes)
$N + 8$	Checksum A
$N + 9$	Checksum B

Table 4.4: VACS Packet Format

Parser

A VACS parser is implemented here as a simple finite state machine parser. The parser can detect and resync if there are duplicated sync bytes, but it does not offer the same recursive resync seen on the NextGen code–in testing, there has been no issue with the parser thus far. Additionally, the parser tracks valid, invalid, and unhandled packet counts. The unhandled count is the number of packets received whose ID does not have a valid callback function.

VACS Callbacks

In order to allow for multiple different components of the system to handle incoming VACS packets without requiring the communications framework to know the details of each VACS packet, a callback system was implemented. This callback system maps VACS message IDs to callback functions. Once incoming packets have been parsed, a hash map of message IDs to callbacks is searched. If a callback has been registered, the callback function is called with the message ID, VACS payload, and VACS payload length as arguments. The signature for a callback function is seen in Source Code 4.3. This callback is based on the form specified in the FCS API, seen in Chapter 5.

```
uint8_t callback_fn(uint16_t message_id,
                    const uint8_t *data,
                    uint16_t length);
```

Source Code 4.3: VACS Callback Function signature

The VACS callback map is implemented as a simple hash map. This hash map is a simple "array of linked lists" type hash map; there are 16 hash buckets and each bucket is the head of a linked list. As the STM32 does not effectively support dynamic memory allocation, the entries in the linked lists are not allocated with malloc but instead come from a backing store of 128 hash map entries. This backing store acts as a rudimentary heap of sorts; new entries are taken from it until the last entry is taken, at which point the insertion algorithm will iterate over the backing store until it finds an unused entry in the store. The actual hash function is a simple modulo 16; this function can be easily altered in the future, but currently it provides a relatively uniform hash distribution.

4.7 Application Code

The application code is responsible for scheduling and executing all periodic tasks in the system. Primarily the scheduler has four task areas to run:

1. Periodic driver updates
2. Communications tasks
3. Sensor fusion tasks (IMU)
4. Control tasks (FCS)

Each of the four tasks can be scheduled independently at any divisor of the fundamental scheduler frequency; currently this frequency is set to 200 Hz. The scheduler frequency is implemented by the use of the ARM SYSTICK timer, which is expressly intended for this purpose.

4.7.1 System Initialization

At boot, code provided by ST-Micro copies data from flash into RAM, enables the FPU, and configures the system clock as necessary. At this point, control is handed to the application main method, which handles low-level, high-level, and applications system initialization. The main method then invokes the main control loop.

The initialization sequence is fairly straightforward. Before any drivers are initialized, both the driver framework and the VACS framework are initialized–they both make use of a hash map system as outlined in Section 4.6.3 which must be configured at run-time. The board configuration is then parsed from the FDT and all drivers are configured appropriately. After these systems are initialized, the low-level drivers are initialized; this involves configuring and enabling all on-chip peripherals. The high-level drivers are then initialized. As there are potential driver dependencies (especially with the MPU9150 and the AK8975), the peripheral driver initialization is required to be blocking; once a driver initialization has started, it must complete before the next driver is initialized. Initialization order is user-defined. The scheduler is initialized next, which involves the configuration and activation of the SYSTICK timer. Finally, the communications, IMU, and FCS systems are initialized. At this point, the main function performs a wait loop for a signal from the SYSTICK timer, which causes a single iteration of the scheduler to run.

4.7.2 Sensor Fusion and Control Tasks

Sensor fusion and control tasks are implemented as external libraries which are compiled separately and linked in as static libraries at compile time. They must conform to a standard set of interface method names, which are invoked by the main software as needed to run the code as well as to trigger any periodic communications. This implementation allows for the FCS and IMU code to be built independently and tested in simulation, then recompiled for ARM and “dropped in” with no other code changes.

Chapter 5: FCS Software Architecture

The FCS software architecture can be broadly divided into two parts: the FCS API, which interfaces the flight control code with the outside world, and the FCS code itself. Note that both the API and the FCS code are platform independent, and are capable of running on a multitude of platforms.

5.1 Application Programming Interface

Typically, both in the VCU UAV lab and elsewhere, the development of autopilot software has followed a monolithic development model. While some code or algorithm design may persist between autopilots, by and large each new hardware platform comes with a reimplementation of the autopilot software as well. While a ground-up rewrite is necessary for low-level drivers, which by necessity vary between platforms, high-level drivers and algorithms do not need to be re-invented for every new platform. As an example, the driver code which parses GPS data can, if written well, be run on a wide range of platforms, even if the underlying RS-232 driver which provides it data changes. This is one of the fundamental design principles behind the modern day operating system, which differentiates between hardware-specific device drivers and hardware-agnostic applications.

During the development of this work, it was decided that, instead of re-inventing the wheel and writing the control algorithms from scratch, the NextGen FCS algorithms would be ported to run on the Aries board. At the same time, it was also desired that miniFCS code be runnable on the Aries as well, due to the use of the miniFCS code base in other projects in the VCU UAV Lab. Out of this desire was born the idea of creating an API for flight control system code: a stable, universal point of entry that all FCS algorithms could use, and that would allow the algorithms from one system to be “dropped in” to another system and run unmodified.

In order to do this, the FCS code was treated as a “black box” which took as inputs various bits of aircraft and hardware state data, and output aircraft control commands. From this abstraction, a set of requirements was created for the FCS API. The API requirements were determined to be as follows:

- Provide aircraft state data to the FCS
- Provide communications channels from and to the FCS
- Provide access to the aircraft control surfaces

These requirements were later expanded to include:

- Provide data storage/retrieval for the FCS
- C and C++ compatibility

based on initial implementation results on the miniFCS and in the RAMS simulator.

5.1.1 API Design and Architecture

The API software framework has two major components: the API *providers*, which implement and present the components of the API; and the API *consumers*, which take that API data and use it to perform flight control tasks. API providers are not typically tightly bound; an API provider can be a combination of several different sources. Most likely, however, there will be a single canonical API provider for a given system, which will implement all API methods and, when the API provider itself is not the source of the requested information, call other modules instead. This creates an API provider which is effectively a layer of intermediate code between the hardware drivers on one side and the FCS code on the other.

There are 5 major coverage areas for the API code, based on the requirements outlined above:

- Methods for retrieving aircraft state
- Methods for setting aircraft control surfaces
- Methods for sending data via the communications link
- Methods for registering the code as a receiver for certain communications packets
- Methods for storing and retrieving communications data

It is necessary to ensure that the API providers and API consumers agree on what data is provided and in what format. To that end, an API specification and reference definition was created, which formalizes the functionality of the API by specifying method names, arguments, and return values for all API methods. The specification also dictates which units are used, as well as valid ranges for all arguments and return types. The following sections detail the 5 coverage areas of the API given above.

Aircraft State

Knowing the current aircraft state, both in terms of position and attitude, is obviously crucial for correct operation of guidance and control algorithms. The majority of API methods are related to retrieving the current aircraft state. In addition to the necessary position (latitude, longitude, and airspeed), attitude (pitch, roll, and yaw), and airspeed data, the API also provides velocity, acceleration, and angular rates. For the API design, the units were chosen to be aviation standard units: altitude in feet, airspeed in knots, and velocities in feet per second.

Aircraft Control Surfaces

As mentioned above, the output of the "black box" model of the guidance and control code are control commands which move the airframe control surfaces to actually fly the plane. The API provides an interface to set aircraft control surfaces. To abstract the actual surface output mechanism from the control code, the API interface deals with normalized values (e.g. -1 for full negative deflection and 1 for full positive deflection). The convention used with all current implementations maps channels 0, 1, 2, and 3 to the aileron, elevator, throttle, and rudder, respectively; this can be changed as necessary so long as the provider and consumer agree on which channel corresponds to which surface. Translation to PWM pulse widths, or to some other method of controlling surfaces, is assumed to be the responsibility of the driver code itself, based on the physical limits of the device being controlled. The API also provides a method for retrieving the normalized value of a given channel.

Communication

Dealing with ground communication is a complex design problem, especially given the requirements of a modular software framework. It is necessary for the guidance software to communicate with the ground station to receive, for example, a target flight path. Likewise, the control software must be able to receive control parameters from the ground station to allow for in-flight adjusting of aircraft performance. However, allowing the API provider to completely handle communications would require it to have some knowledge of the internals of the guidance and control algorithms. This would break the abstraction given by the API.

Consideration must also be given to communications format. As communications links evolve, the format may be changed or altered as well. However, it would be ill-advised to design an API to expect or assume certain things about the communications format. As such, the API design relies only on the idea of a unique message identifier and assumes all data will be treated as a byte array. This offloads the burden of handling communications formats to the API provider itself, which must be able to communicate with other elements in the system, and to the API consumer, which must be able to decode or unpack the data it receives. The API itself remains ignorant of the underlying format of the data being exchanged.

The API provider handles sending and receiving data in a fairly protocol-independent manner. It is assumed that each message will have some form of unique message ID, which will be used to correctly route the message. Sending data is the more straightforward approach. From the standpoint of the API consumer, the code need only serialize data into a vector of unsigned bytes, and provide the correct message ID; the API provider will deal with any additional packaging and then transmit it accordingly.

Receiving data requires a more complex approach. As the API provider cannot know any details about the data packet, it cannot process the data packet on its own. It is assumed that the communications protocol exposes enough information about the data, namely a single unique message identifier, that allows the API provider to detect and dispatch incoming messages without inspecting them. This is handled with the use of callback functions. During initialization, the guidance and control codes will call an API method which will "register" a function in the guidance or control code as the handler for a given message type. Upon receipt of a message with the correct message type, the API provider will call all functions associated with the message type and provide them a copy of the message data. In this way the API provider can avoid needing to know the details of the communications packets being sent or received.

Configuration Storage

It is often useful to be able to save and retrieve internal configuration data. For example, the parameters used for PID loops in the control algorithms should not be lost every time the system is powered down. While it would be possible for the system to require upload of control parameters from a ground station on every flight, it is much easier and less error prone to store these parameters locally. It is therefore necessary to offer some form of non-volatile storage for arbitrary data blocks to the API consumers, to allow them to store and retrieve data as necessary. One approach is to allow the storage and retrieval of evenly sized blocks, identified by a unique ID. This would allow, for example, the controls block to store its internal state information in the block with unique ID "1234". On subsequent runs of the autopilot system, it would request that block using the unique and unpack it as needed. Of course, care must be taken to ensure that the IDs are, in fact, unique.

5.1.2 API Implementations

During the course of this work, all major platforms in the VCU UAV Lab, with the exception of the NextGen FCS hardware, were ported to the API provider/API consumer model. All of these implementations were done in C or C++; although the API concept can be applied to any language which features functions, all of the work in the VCU UAV lab is done primarily in these languages. The process of implementing API providers and API consumers revealed several flaws in the original design of the API, which were remedied as they were discovered–for example, the original API specification lacked a data storage method. A list of all methods implemented for the API can be seen in Table 5.1.

Initially, the API was intended to be implemented much like standard C libraries: include a header declaring all the API methods, and they will be resolved at link time to the correct API functions. This would work fine for FCS hardware. However, integrating this into the RAMS simulator posed a significant challenge. The RAMS simulator, as one of its primary features, supports multiple "agents". A system which assumed only one API provider would remove the ability of the RAMS simulator to simulate multiple planes with independent state and therefore independent API providers. Several ideas were considered. The simplest was to add some form of context pointer to all API calls. However, this introduced unnecessary overhead on hardware implementations. Other ideas were also briefly considered but rejected due to unnecessary overhead.

After consideration, a simple solution was discovered: encapsulate the API in an object and provide it to the FCS code at run-time, during the initialization phase. All API methods are defined as a structure of function pointers, each with the expected signature as defined by the API specification. The provider is responsible for correctly resolving these function pointers to appropriate methods, either implemented by the provider itself, or, where applicable, implemented by the underlying driver code. A pointer to this structure is then passed to the API consumer during initialization; the consumer uses this struct whenever it makes an API call.

This implementation provides an interesting side effect which makes implementing a C++ API

Function	**Description**
`get_latitude`	Return current latitude in decimal degrees
`get_longitude`	Return current longitude in decimal degrees
`get_altitude`	Return current altitude in feet
`get_heading`	Return current heading in degrees
`get_airspeed`	Return current airspeed in knots
`get_groundspeed`	Return current ground speed in knots
`get_yaw`	Return current yaw angle in degrees
`get_pitch`	Return current pitch angle in degrees
`get_roll`	Return current roll angle in degrees

`get_velocity_north`
`get_velocity_east`
`get_velocity_down`
	Return current NED velocity in feet per second

`get_accel_x`
`get_accel_y`
`get_accel_z`
	Return current body accelerations in g

`get_yaw_rate`
`get_roll_rate`
`get_pitch_rate`
	Return current angular rates in degrees per second

`get_fcs_voltage`	Return current FCS battery voltage
`get_rc_voltage`	Return current RC battery voltage
`get_pwm_value`	Return the normalized PWM value for a channel
`set_pwm_channel`	Set PWM channel to a normalized value
`comm_register_callback`	Register a callback function as the handler for messages with a given ID
`comm_send_message`	Send a message over the communications channel
`store_data`	Store data in a permanent storage location
`get_data`	Retrieve data from a permanent storage location.
`gps_valid`	Return true if the system has valid GPS data
`rc_loss`	Return true if there is an RC loss event
`comm_loss`	Return true if there is a communications timeout event

Table 5.1: FCS API Functions

provider relatively easy. With the implementation described above, all API consumer code will ultimately access API methods like so:

```
api->method();
```

This is the valid syntax for calling a function pointer method from a pointer to structure. It is also valid syntax for calling a method from a class pointer in C++. Assuming the rest of the FCS code base is valid C++, a C++ class pointer can be substituted for a C structure pointer and be compiled as a C++ program. Initially, the RAMS simulator used a similar implementation; this has since been updated to use the reference implementation described below.

5.1.3 Reference API Implementation

In order to standardize the API across different API providers and API consumers, a single reference "implementation" was created which specifies both C and C++ API implementations, as well as common complex types (enumeration and typedef) for both implementations. By doing so, a project which wishes to use the API can include this reference implementation and ensure compatibility across the entire range of producers and consumers. The reference implementation provides both C and C++ examples, shown below in Source Codes 5.1 and 5.2. The C API reference is implemented as a struct of function pointers, while the C++ API reference is implemented as an abstract base class.

Additionally, a unified API header is provided which will define the type `api_t` correctly for C and C++ code. Code which does not expect to intermingle C and C++ can simply include this header, shown in Source Code 5.3. This will always provide the correct API header for a given compiler. In addition, this will work when the same code base is compiled as C or C++; the compiler will use the correct definition regardless.

```c
struct _api_c_t {
float (*get_latitude)(void);
float (*get_longitude)(void);
float (*get_altitude)(altitude_mode_t);
float (*get_airspeed)(void);
float (*get_groundspeed)(void);
float (*get_heading)(void);
float (*get_yaw)(void);
float (*get_pitch)(void);
float (*get_roll)(void);
float (*get_accel_x)(void);
float (*get_accel_y)(void);
float (*get_accel_z)(void);
float (*get_pitch_rate)(void);
float (*get_roll_rate)(void);
float (*get_yaw_rate)(void);
float (*get_velocity_north)(void);
float (*get_velocity_east)(void);
float (*get_velocity_down)(void);
float (*get_fcs_voltage)(void);
float (*get_rc_voltage)(void);
float (*get_pwm_value)(uint8_t channel);
int (*set_pwm_value)(uint8_t channel, float new_pwm_value);
void (*comm_register_callback)(uint16_t message_type,
                                   comm_callback_fn callback);
void (*comm_send_message)(uint16_t message_type,
                              uint8_t destination_address,
                              const uint8_t *data,
                              uint16_t length);
int8_t (*store_data)(uint16_t key, uint8_t *data, uint16_t length);
int8_t (*get_data)(uint16_t key, uint8_t *data, uint16_t length);
bool (*gps_valid)(void);
bool (*rc_loss)(void);
bool (*comm_loss)(void);
};
```

Source Code 5.1: C API Reference Implementation

```
class _api_cpp_t {
public:
    virtual float get_longitude(void) = 0;
    virtual float get_latitude(void) = 0;
    virtual float get_altitude(altitude_mode_t) = 0;
    virtual float get_airspeed(void) = 0;
    virtual float get_groundspeed(void) = 0;
    virtual float get_heading(void) = 0;
    virtual float get_yaw(void) = 0;
    virtual float get_pitch(void) = 0;
    virtual float get_roll(void) = 0;
    virtual float get_accel_x(void) = 0;
    virtual float get_accel_y(void) = 0;
    virtual float get_accel_z(void) = 0;
    virtual float get_roll_rate(void) = 0;
    virtual float get_pitch_rate(void) = 0;
    virtual float get_yaw_rate(void) = 0;
    virtual float get_velocity_north(void) = 0;
    virtual float get_velocity_east(void) = 0;
    virtual float get_velocity_down(void) = 0;
    virtual float get_fcs_voltage(void) = 0;
    virtual float get_rc_voltage(void) = 0;
    virtual float get_pwm_value(uint8_t channel) = 0;
    virtual int set_pwm_value(uint8_t channel, float new_pwm_value) = 0;
    virtual void comm_register_callback(uint16_t message_type,
                                        comm_callback_fn callback) = 0;
    virtual void comm_send_message(uint16_t message_type,
                                   uint8_t destination_address,
                                   const uint8_t *data,
                                   uint16_t length) = 0;
    virtual int store_data(uint16_t key, uint8_t *data, uint16_t length) = 0;
    virtual int get_data(uint16_t key, uint8_t *data, uint16_t length) = 0;
    virtual bool gps_valid(void) = 0;
    virtual bool rc_loss(void) = 0;
    virtual bool comm_loss(void) = 0;
};
```

Source Code 5.2: C++ API Reference Implementation

```
#ifdef __cplusplus
#include "api_cpp.hpp"
typedef _api_cpp_t api_t;
#else
#include "api_c.h"
typedef struct _api_c_t api_t;
#endif
```

Source Code 5.3: API Reference Implementation Header

API C/C++ Shim

As mentioned above, the use of the reference API implementation allows the same code to be compiled as C and C++ and use the correct API implementation regardless. However, this is not always desirable. For example, designated initializer lists for structs are not valid C++ and will not compile as C++; however, their use in C is desirable to more clearly initialize structures. There is a desire to run FCS code (written in C) in the RAMS simulator (written in C++), though. To provide the best of both worlds, an API shim was written in C++. This shim provides a C API struct and C functions which all call their C++ equivalent. At initialization, the shim is passed a C++ API. This is stored in a local variable and used in the C functions. The C API struct is then passed into the FCS code which has been compiled by a C compiler. In this way, a C FCS code base can use the C++ API from the RAMS simulator without also needing to be a valid C++ program.

5.2 Flight Control System Software

The basic flight control system used in the Aries is a heavily rewritten version of the algorithms used in the NextGen FCS [1], which themselves are rewritten versions of the code originally developed in [63]. Fundamentally, the FCS code uses PID control loops to achieve flight control. It supports waypoint and loiter navigation, as well as safety fallbacks in the event of GCS loss or RC loss.

5.2.1 FCS Software Architecture

The NextGen FCS software was largely implemented as two giant C++ classes: a "waypoint navigator" class which ran all FCS algorithms and maintained all FCS state, and a VACS communication class which handled ground communication. As a result, the code for the FCS was in one file of about 2,000 lines–it was not an easy task to modify the code to support additional features. One of the goals of the FCS rewrite was to make the FCS software easier to modify for future work, especially to support the addition of Automatic Take-off and Landing (ATOL) control, which at the time of this work was being implemented on the miniFCS by another graduate student despite the ill-suited nature of the miniFCS algorithms for larger airframes. To that end, the FCS software was re-written from the ground up as a number of separate modules. Several variations of inter-module communication were considered. Ultimately, more complex methods such as chaining functions together with void-pointer data, akin to UNIX pipes, were scrapped in favor of a simple system of shared data structures. A table of the FCS modules can be found in Table 5.2. More detail on each module is given in the following sections.

Module	**Description**
fcs	Main FCS module
nav	Navigation and flightpath control
control	Aircraft control
data	Shared state data
math	Common FCS math functions
comm	Ground communication code

Table 5.2: FCS Software Modules

5.2.2 FCS Module

The FCS module is the primary interface to the FCS software. It provides the initialization and run functionality that will be invoked by the system software to actually run FCS code. For the most part, the FCS module itself does very little; most of the heavy lifting is done by the other modules. The FCS module is responsible for initialize the FCS data structures as well as the other modules. Data initialization is performed by reading all parameter data from the permanent flash provided by the API; if this fails, default values are used as a fallback.

The FCS module also is responsible for running the FCS code. For the most part, the FCS module merely invokes the nav and control modules to accomplish this. However, it does handle setting a rally point for GCS and RC loss modes, as well as actually checking the GCS and RC loss modes using API functions. It also updates an aircraft state structure, again using API functions. It then invokes the nav module run function and the control module run function. Finally, if the FCS is in autonomous mode, the FCS module will output the PWM values calculated by the control module to servos using API servo commands.

5.2.3 Nav Module

The nav module is responsible for maintaining and updating the flight path of the plane, as well as implementing loiter functionality. In general, the nav module inputs present aircraft state and outputs target course, airspeed, and altitude parameters to be used by control code.

The flight path for the navigation code is stored as a list of waypoints; each waypoint has a target latitude, longitude, altitude, and airspeed associated with it. Altitude and airspeed are optional parameters and, if they are omitted, the navigation code will instead use global parameters for target altitude and airspeed. Navigation is implemented as several different navigation modes which calculate target course in different ways. Right now only two such modes are implemented– waypoint and loiter. Waypoint mode flies a flight path specified by several waypoints, while loiter mode circles a given waypoint at a specified distance. Additional modes may be easily added in the future, so long as they correctly generate target course, altitude, and airspeed.

5.2.4 Control Module

The control module is responsible for taking a target course from the nav module and actually controlling the aircraft. As previously mentioned, the control code is currently implemented using PID control loops. There are two “sets” of PID loops. Outer control PID loops take the desired course of the plane and output a target attitude (roll, pitch) and airspeed. Inner control PID loops take these targets and output target PWM values. All PID parameters are linearlly interpolated between two set points, based on the airspeed. If this interpolation fails, the default parameters "fall back" to the lower-speed parameters.

The main control run function performs this interpolation, and then runs the outer loop and then inner loop controls in order. It can support different FCS modes which would have different outer or inner loops, but the current implementation uses the same loops for all supported flight modes.

5.2.5 Data Module

The data module contains all shared FCS data, as well as some functions for serializing and deserializing this data.

5.2.6 Math Module

Several math functions were repeatedly used during the execution of the FCS algorithms–limiting and slew limiting, bearing and distance calculations, and linear interpolation. These functions were placed in their own module. This module could be easily extracted and make into a library in and of itself in the future, should the desire arise.

5.2.7 Comm Module

The comm module handles all external communications. It implements the callback function passed into the API comm register function, as well as providing functions which generate and transmit a standard set of FCS reports which give the GCS information on the current state of the FCS.

Chapter 6: Simulation and Testing Results

In order to validate correct functionality of the flight control system, testing was performed throughout the development process. All FCS code was first tested and validated in the RAMS simulator. The FCS code and the Aries hardware were then tested using VCU's Hardware-in-the-Loop simulation.

Both the RAMS simulator and the VCU HILS setup make use of the open-source flight simulator FlightGear [64]. This allows for a more accurate physics simulation than a simple point mass model, as well as providing visual feedback via FlightGear's cockpit or model views. FlightGear supports a network interface, based on UDP sockets, which allow an external application to send control commands to the simulator; the position of the aircraft in the simulator is updated accordingly and a new aircraft state packet is sent back to the external application.

6.1 Software Simulation

By being a valid API provider, the RAMS simulator supports software simulation of FCS code. It integrates with the open-source flight simulator FlightGear to provide a relatively realistic physics environment for the plane to fly in. Software simulation was a vital part of testing the initial rewritten FCS algorithms, as it allowed for a rapid "find bug, fix bug, recompile" sequence due to it using loadable modules for FCS code. It was not necessary to recompile the entire simulator for every change in the FCS software.

The simulator provides a full interface to the VCU GCS, allowing for flight observation and flightpath upload as though the simulator was an actual plane. A series of flightpaths were uploaded and flown in the simulator to demonstrate basic flight functionality and ensure that the FCS code would be able to fly an actual aircraft.

6.1.1 Flightpath tracking

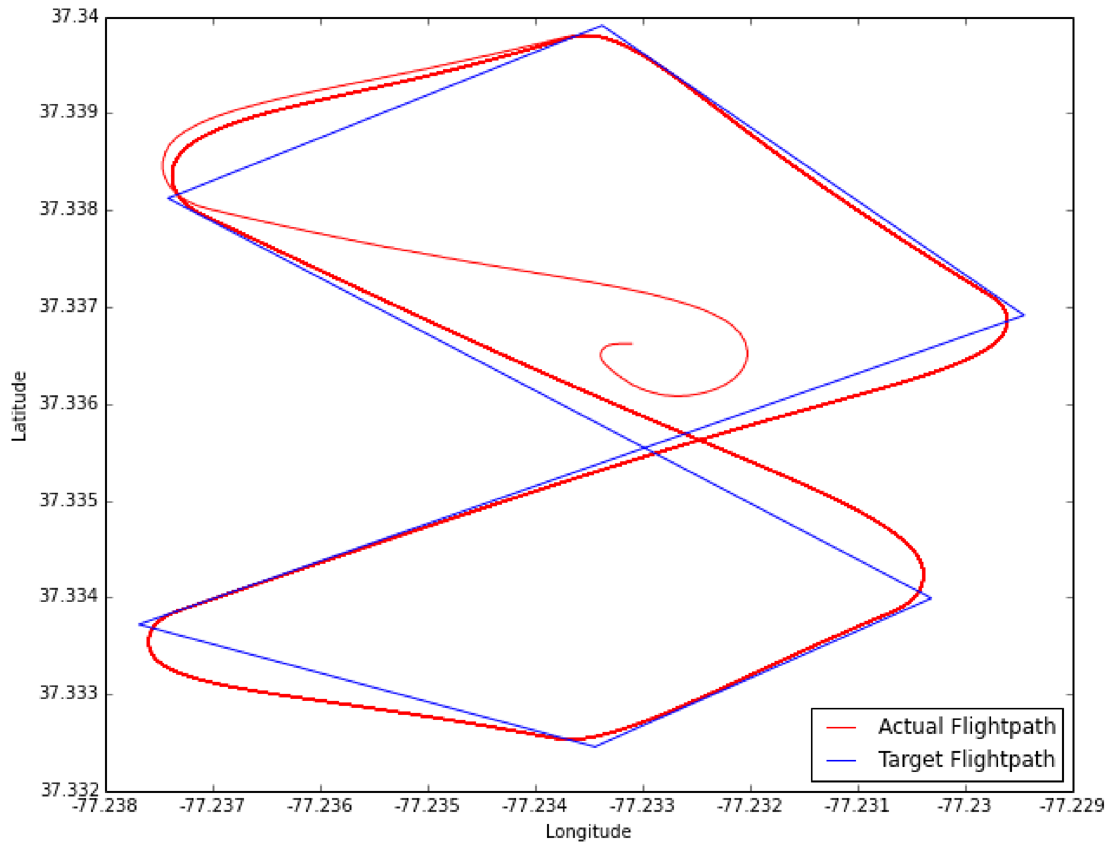

Figure 6.1: Flightpath data from RAMS simulator

As a basic test, a 6-waypoint flightpath making a basic figure-8 was created and sent to the FCS code running on the RAMS simulator. This pattern tests both left-hand and right-hand turns, and provides a variety of turn angles. A plot of the flightpath over a period of around 40 minutes is shown in Figure 6.1. As can be seen, apart from the initial leg of the flightpath, the FCS tracks the same path consistently. The path is slighty off from the “ideal” path, as the FCS does not currently support crosstrack mode. For all flight tests, the “arrival range” (distance at which the plane is considered to have reached a waypoint) was set to 150 feet.

6.2 Hardware-in-the-Loop Simulation

HILS testing allows for testing both flight control software as well as hardware and driver software. The HILS board used at VCU provides hardware emulation of an NMEA GPS, a MIDG2 GPS/IMU, and analog pressure sensors used on the Aries board. As with the RAMS simulator, FlightGear is used as the simulation back-end; the HILS board translates FlightGear data to actual sensor outputs, and reads the servo outputs of the FCS board to be sent back to FlightGear. A diagram showing these connections is given in Figure 6.2.

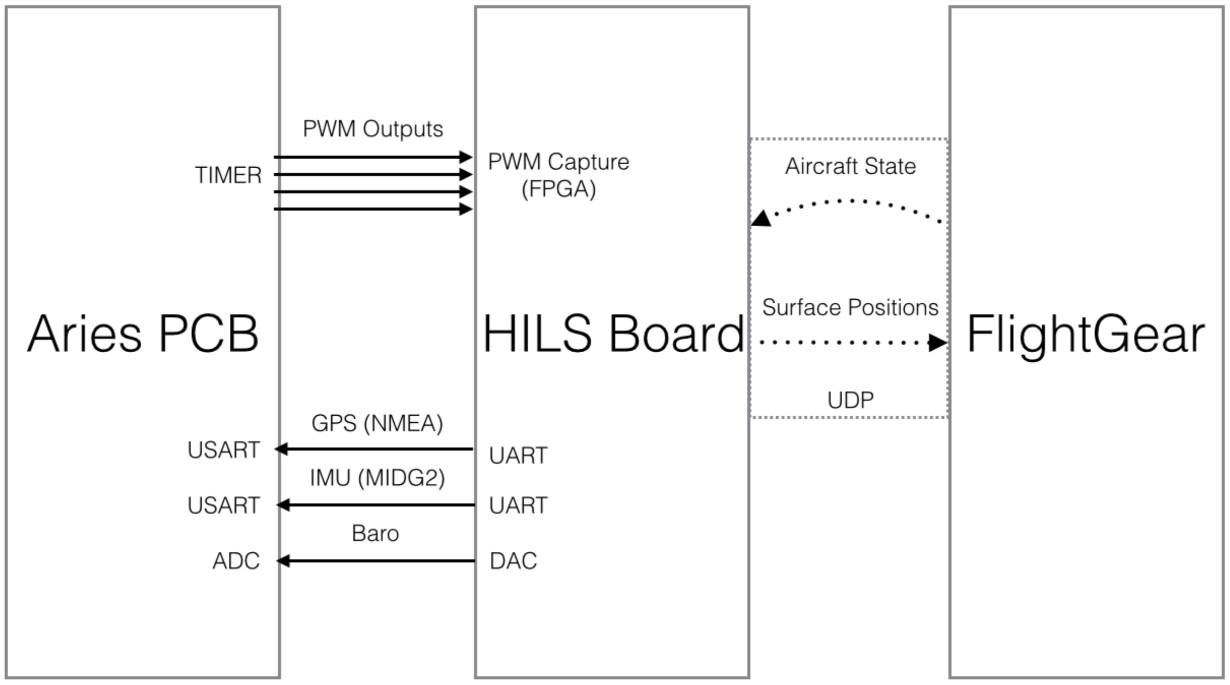

Figure 6.2: Connection between Aries PCB, HILS board, and FlightGear

Several overnight runs of the HILS simulation setup were conducted, to ensure that the FCS would be capable of running for long periods of time without issue. In addition, HILS flights were used to tune some of the control parameters of the FCS before real-world flight testing. The use of

HILS testing revealed several issues with the prototype revision of the Aries PCB, including noise issues which rendered the analog sensors effectively unusable for flight.

6.2.1 Flightpath tracking

The same flightpath flown by the RAMS simulated FCS code was flown using HILS testing as well. A plot of this flightpath over approximately the same period of time is shown in Figure 6.3. As with the results from the RAMS simulator, the flightpath tracking is very accurate after the initial pass to start following waypoints.

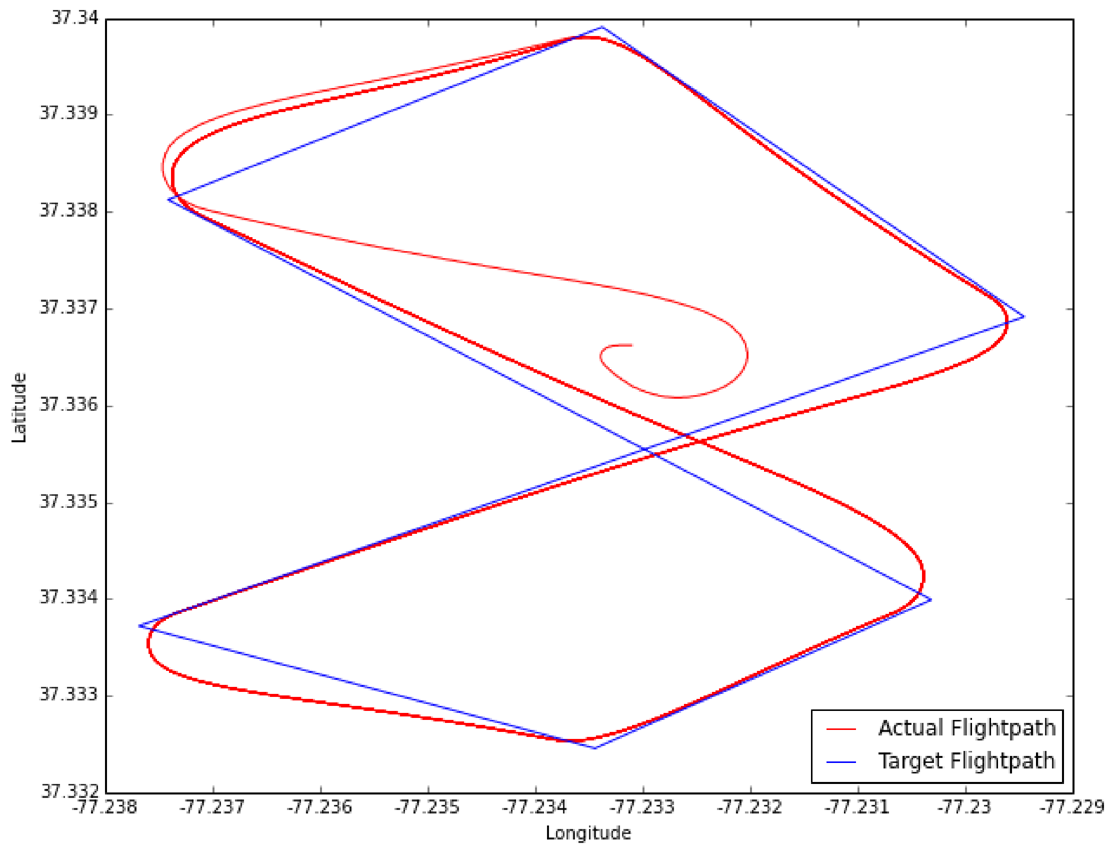

Figure 6.3: Flightpath data from HILS testing

6.2.2 Altitude Hold

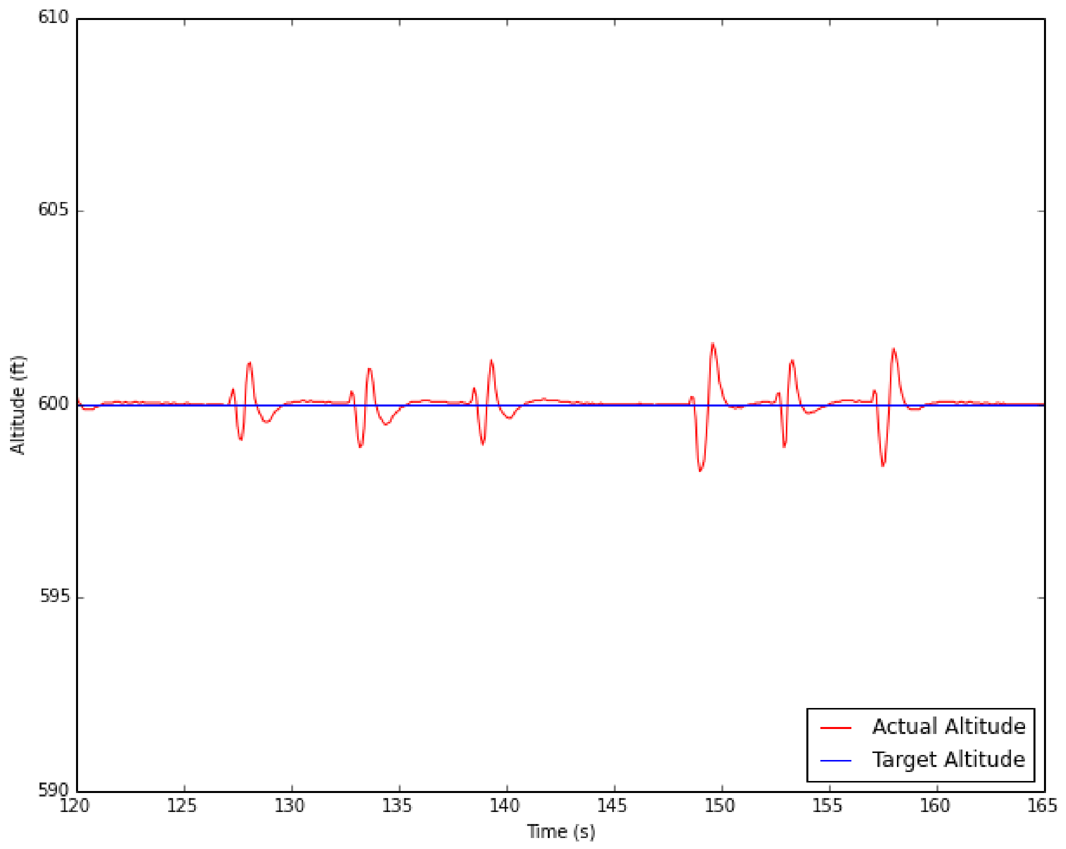

Figure 6.4: Target versus Actual Altitude from HILS testing

Altitude hold functionality of the FCS code is shown in Figure 6.4. Although the actual altitude varies slightly and never quite comes close to the target altitude, it still performs relatively well. Slight peaks and valleys can be seen when the aircraft hits a given waypoint and begins banking towards the next waypoint. This should be able to be smoothed out by careful tuning of the feed-forward parameters for target altitude and target airspeed.

Figure 6.5 shows the climb/descent performance of the FCS in simulation. The graph shows a hold of 400 ft, descent to 200 ft, and then climb to 600 ft. In all cases the aircraft is able to climb/descent and hold altitude without oscillation or overshoot.

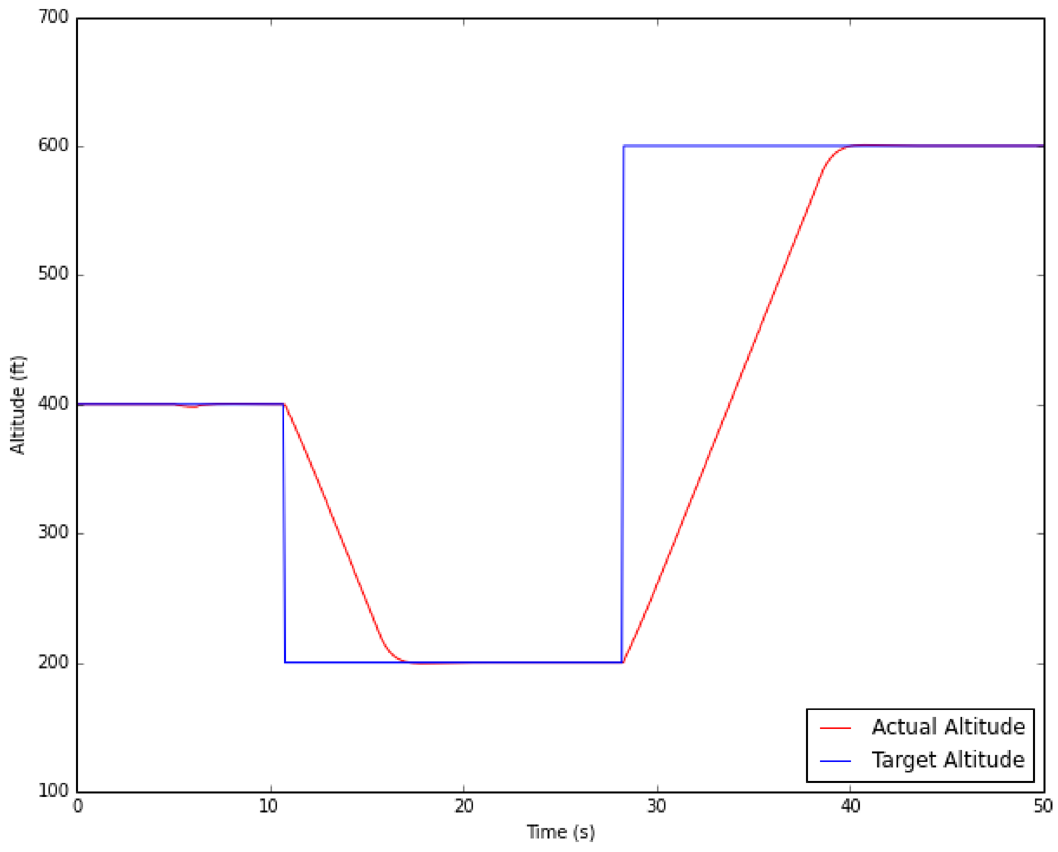

Figure 6.5: Target versus Actual Altitude from HILS testing, climb and descent

6.2.3 Airspeed Hold

Airspeed hold functionality of the FCS code is shown in Figure 6.6. The airspeed control holds relatively well over the course of a single flight of the flightpath; again there are some minor variations as it changes roll angle when it hits a given waypoint. These minor variations amount to less than 1 knot and the controller recovers within 5 or so seconds. Careful tuning of the airspeed controller and tweaking of the feed-forward parameters should allow the elimination of these small errors.

Figure 6.7 shows the behavior of the FCS when directed to change airspeed. As can be seen, there are small amounts of under/overshoot as the FCS attempts to change airspeed lower or higher.

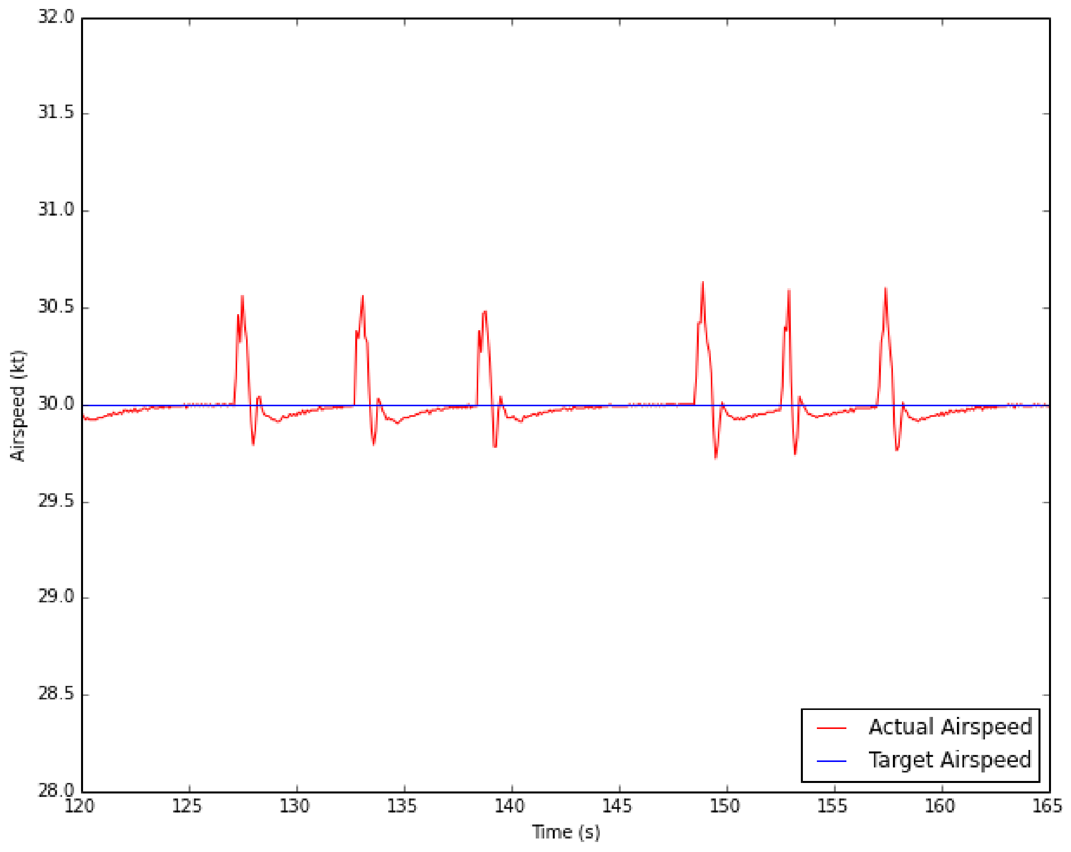

Figure 6.6: Target versus Actual Airspeed from HILS testing

This is likely due to an improperly tuned airspeed controller.

6.2.4 Roll Control

Roll control functionality of the FCS code is demonstrated in Figure 6.8. As can be seen, the target and actual roll track very closely; with a minor amount of overshoot as the roll approaches the desired target. This again can likely be eliminated with careful tuning of the roll controller PID parameters.

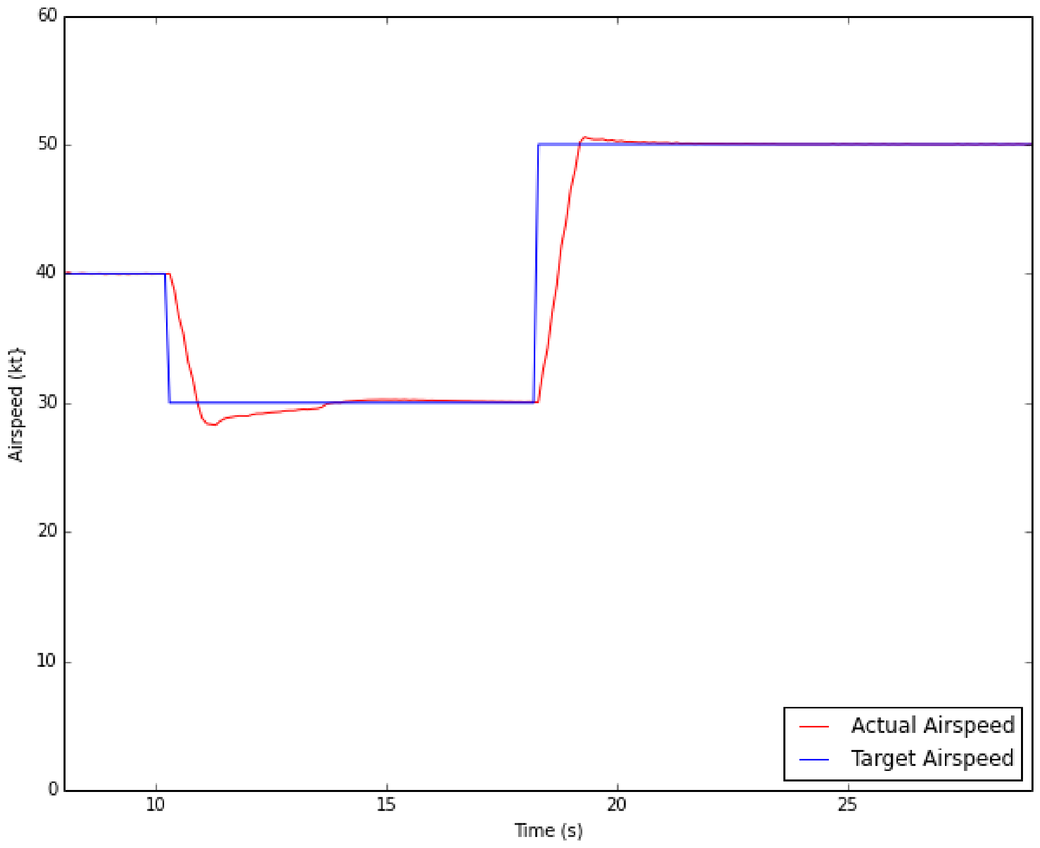

Figure 6.7: Target versus Actual Airspeed from HILS testing, changing set point

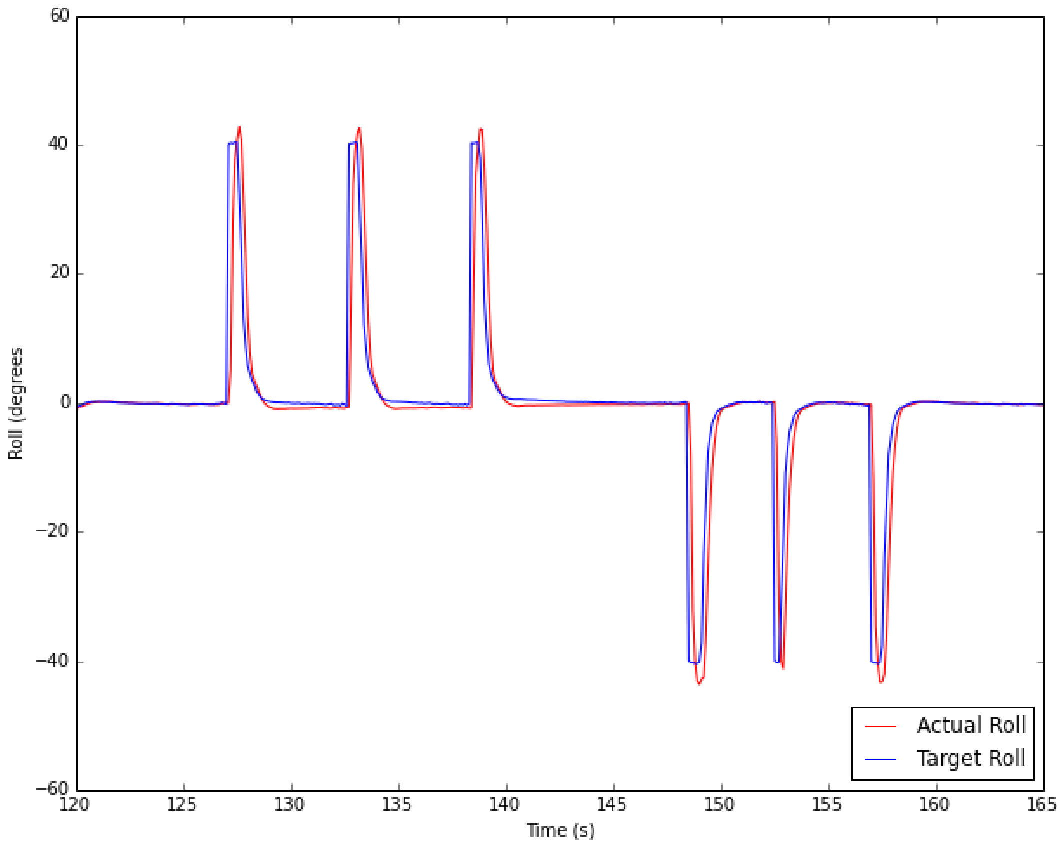

Figure 6.8: Target versus Actual Roll from HILS testing

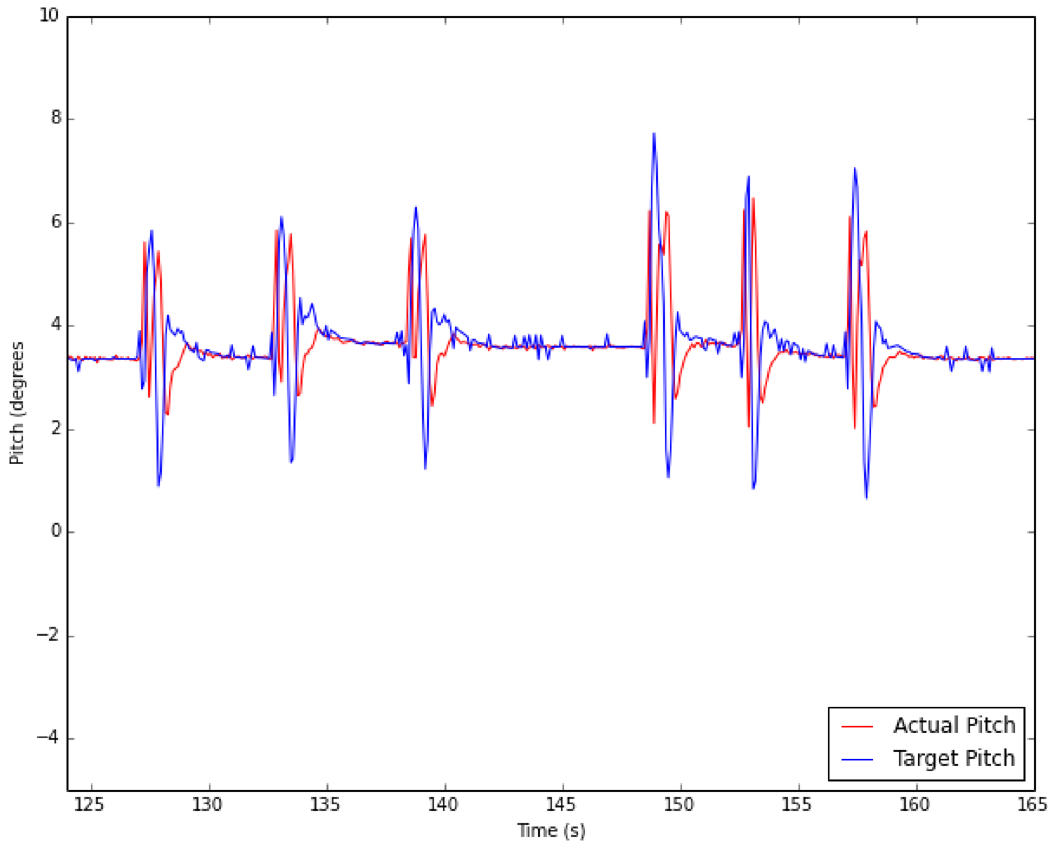

Figure 6.9: Target versus Actual Pitch from HILS testing

6.2.5 Pitch Control

Pitch control functionality of the FCS code is demonstrated in Figure 6.9. Some error is seen due to the sharp changes in target pitch over a small amount of time; this suggests that the target pitch controller is in need of tuning. The actual pitch itself follows fairly well given the rapid change in target pitch. In all likelihood, the actual inner loop pitch control is sufficiently tuned.

Chapter 7: Conclusions and Future Work

7.1 Conclusion

As presented here, an adequate replacement for both the miniFCS and NextGen has been created. The platform is far more powerful and feature-rich than the miniFCS, while avoiding some of the pitfalls caused by the complexity of the NextGen software. While the platform has been successfully tested in simulation, a large amount of work remains to be done to make the Aries a true replacement for the VCU UAV Lab's existing FCS platforms. In addition to exhaustive flight testing, an outline of ongoing and future work for the Aries platform is outlined below.

7.2 Ongoing Work

There is significant room for improvement of the Aries system. Indeed, several intended features of the board were left uncompleted due to time constraints, such as successful integration of the Ethernet and SD card hardware. The sections below outline some of the features that will be added to what is at the time of writing the second revision of the Aries hardware, as well as outlining some potential future improvements to the board hardware and software.

7.2.1 Second Board Revision

At the time of this writing, work was ongoing on creating a second revision of the Aries board which meets the size constraints outlined in Section 3.1. This board will feature the STM32F417IGH6, a 176-pin BGA version of the STM32F4 used on the current board, that also features hardware accelerated cryptography options. Additionally, the board will feature 8 MB of external PSRAM.

In addition to most of the peripherals mentioned in Chapter 3, the second revision will make use of the on-board SD card and Ethernet MAC. The SD card will be used for data logging at a much faster rate than is supported by the ground communication modem. Configurable trace log levels will allow the use of the Aries board as a data gathering system, although it is currently unknown if the board will be able to operate as a full FCS while tracing all peripheral data. An on-board Ethernet PHY and RJ-45 connector will provide network connectivity; paired with the lwIP lightweight IP stack [65] it will allow simple TCP and UDP connections. A working example is provided by ST Micro [66] which should allow for a relatively quick implementation.

The second board revision will also implement USB programming functionality and USB mass storage for accessing the SD card. USB programming functionality will allow the programming of the on-board processor directly via USB DFU mode, as well as programming the safety switch and Xbee via on-board FTDI USB to Serial chips. This will allow the device to be programmed (but not debugged) without the need for an external SWD debugging interface. In addition, there will be a USB serial console which can be used for printing information to the screen.

The barometric pressure sensors will also be replaced on the second board revision. Due partially to the noise issues described in Section 4.2.9, and partially to a desire to move away from analog sensors in general, two digital pressure sensors have been chosen to replace the current analog ones. The Measurement Specialties MS4525DO series offers both absolute and differential pressure sensors in the same form-factor. These sensors are available in a variety of pressure ranges as well. All sensors use the same basic interface format to read data; only the transfer equations differ.

Backup battery power will also be supplied on this revision. A small lithium coin battery will be used to provide backup power to the main processor and the GPS module. This will allow RTC time and GPS fix data to be kept for up to 15 days in standby. This should greatly reduce time to first fix for the GPS module as well as providing accurate date and time information for the SD card logging system.

7.2.2 Final Board Revisions

Once the second board revision has been manufactured and thoroughly tested, two final revisions will be made: one which is hopefully relatively unchanged from the second revision, and a “tiny” version, without most on-board peripherals, which is intended for use in very small airframes.

Final Revision

The “final” revision of the Aries FCS will hopefully be unchanged from the second revision. Some concerns exist about the reliability of the on-board USB setup, as well as the SRAM. Furthermore, the use of a large number of BGA components also poses a risk; while successful BGA soldering has been accomplished in the VCU UAV Lab, it has never been tried on a board with multiple closely-spaced components. Any issues with BGA may result in the scaling back and re-designing of less important components to not use BGA; however, all attempts will be made to keep the processor and SRAM as BGA components. The final revision may also offer daughter board functionality similar to the Aries Lite, as explained in the section below.

Aries Lite

The Aries Lite will be a very small form-factor board, ideally measuring 1” by 2” or less. This size constraint obviously reduces the number of peripherals which can be used. The Lite will only feature the bare minimum number of components on-board: the main and safety switch processors, the IMU, and (if space allows) the microSD card system. It will likely feature a daughter board system which connects barometric sensors. Ethernet connectivity will likely be made available either a daughter board or a break-out board, as the planes the Lite will be flying in will also likely be using the Ethernet/Wi-Fi based communications system developed in [67].

7.3 Future Work

In addition to the ongoing revisions discussed above, there is potential for future changes which have not been partially implemented already. Many of the rough edges of the current design were only discovered after it was too late to realistically change them. The following sections outline some of the changes that could be implemented, both hardware and software, in version 2 of the Aries FCS or beyond.

7.3.1 Better IMU Sensors

As mentioned above, the choice of IMU sensor was dictated by their use in another graduate student's work. Indeed, the sensors themselves are quite capable–the problem is mostly with the poor implementation of the I2C core on the STM32F4. A better IMU solution would likely be to use an MPU6050 in SPI mode with an external, SPI interfaced magnetometer. The AK8975 on the MPU9150 die, when packaged separately, supports SPI mode. Additionally, many of the open-source autopilots make use of the HMC5883L; this is another option worth exploring.

7.3.2 Integration of ATOL code

The current FCS algorithms have no support for automatic take-off and landing. As previously mentioned, another graduate student was implementing such algorithms on the miniFCS platform at the time of writing. These algorithms should be ported to the Aries platform. Additional sensors may be needed for successful ATOL; these too should be integrated into the Aires platform as needed.

7.3.3 Real-Time Operating System

Currently, the Aires software platform runs "bare-metal" with no OS overhead. At the time of design, there were no RTOS solutions available which met the needs for the Aries. However, in the year since the Aries design was determined, two contenders have emerged as potential RTOS solutions: NuttX [68] and ChibiOS/RT [69]. Both fully support the STM32F4 and all on-board peripherals. NuttX is used in the ArduPilot Pixhawk PX4 [22], which runs a slightly better version of the STM32F4 used in the Aries. However, ChibiOS/RT seems more "friendly" to low-level manipulation where NuttX attempts to draw a hard line between user code and driver code. For example, to port the MPU driver, or any I2C or SPI driver, to NuttX, it would be necessary to write a NuttX kernel driver for them. Regardless, though, an RTOS should be further explored as a possible upgrade to the current driver/scheduler interface used on the Aries. Additionally, the presence of an RTOS would make the board suitable for other uses–perhaps even as a replacement for the current HILS board.

Bibliography

[1] R. C. DeMott II, “Development of a flexible fpga-based platform for flight control system research,” Master’s thesis, Virginia Commonwealth University, 2010.

[2] J. E. Ortiz, “Development of a low cost autopilot system for unmanned aerial vehicles,” Master’s thesis, Virginia Commonwealth University, 2010.

[3] L. B. Mize IV, “Development of a multiple vehicle collaborative unmanned aerial system,” Master’s thesis, Virginia Commonwealth University, 2011.

[4] (2014). [Online]. Available: http://ardupilot.com/

[5] P. Project. Paparazzi. [Online]. Available: http://wiki.paparazziuav.org/wiki/Main_Page

[6] P. Brisset, A. Drouin, M. Gorraz, P.-S. Huard, and J. Tyler, “The paparazzi solution,” *MAV2006*, 2006.

[7] [Online]. Available: http://wiki.paparazziuav.org/wiki/GCS

[8] [Online]. Available: http://wiki.paparazziuav.org/wiki/Simulation

[9] [Online]. Available: http://wiki.paparazziuav.org/wiki/Autopilots

[10] [Online]. Available: http://wiki.paparazziuav.org/wiki/Lisa

[11] B. Gati, “Open source autopilot for academic research - The Paparazzi system,” in *American Control Conference (ACC), 2013*, June 2013, pp. 1478–1481.

[12] [Online]. Available: http://wiki.paparazziuav.org/wiki/KroozSD

[13] [Online]. Available: http://wiki.paparazziuav.org/wiki/Apogee/v1.00

[14] [Online]. Available: http://wiki.paparazziuav.org/wiki/Umarim_Lite_v2

[15] [Online]. Available: http://wiki.paparazziuav.org/wiki/Hardware

[16] [Online]. Available: http://wiki.paparazziuav.org/wiki/AspirinIMU

[17] [Online]. Available: https://www.ppzuav.com/osc/index.php?cPath=2&osCsid=3gkm5sa7n8ne8fusjiodv69mu6

[18] [Online]. Available: http://diydrones.com/

[19] [Online]. Available: http://store.3drobotics.com/t/parts/autopilots

[20] [Online]. Available: https://www.sparkfun.com/products/8785

[21] [Online]. Available: http://pixhawk.org/

[22] [Online]. Available: http://pixhawk.org/modules/pixhawk

[23] [Online]. Available: http://store.3drobotics.com/products/3dr-pixhawk

[24] [Online]. Available: http://store.3drobotics.com/products/apm-2-6-kit-1

[25] 3DRobotics. (2014, 02) ArduPilot Mega 2.5 Schematic. [Online]. Available: http://3drobotics.com/wp-content/uploads/2014/02/APM_v252_RELEASE.zip

[26] M. Coombes, O. McAree, W.-H. Chen, and P. Render, "Development of an autopilot system for rapid prototyping of high level control algorithms," in *Control (CONTROL), 2012 UKACC International Conference on*. IEEE, 2012, pp. 292–297.

[27] H. B. Christophersen, W. J. Pickell, A. A. Koller, S. K. Kannan, and E. N. Johnson, "Small Adaptive Flight Control Systems for UAVs using FPGA/DSP Technology," in *AIAA 3rd "Unmanned Unlimited" Technical Conference*. AIAA, September 2004.

[28] [Online]. Available: http://www.adaptiveflight.com/products/fcs

[29] C. Coopmans and Y. Han, "AggieAir: An integrated and effective small multi-UAV command, control and data collection architecture," in *Proceedings of the 5th ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications (MESA09)*, 2009.

[30] C. Coopmans, "AggieNav: A small, well integrated navigation sensor system for small unmanned aerial vehicles," in *Proceedings of the 209 ASME Design Engineering Technical Conference Computers and Information in Engineering*, 2009.

[31] C. Coopmans, H. Chao, and Y. Chen, "Design and implementation of sensing and estimation software in AggieNav, a small UAV navigation platform," in *Proceedings of the ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference. IDETC/CIE*, 2009.

[32] [Online]. Available: http://www.micropilot.com/

[33] [Online]. Available: http://www.micropilot.com/products-mp2028g.htm

[34] [Online]. Available: http://www.micropilot.com/products-mp2128g.htm

[35] [Online]. Available: http://www.micropilot.com/products-mp1028g.htm

[36] [Online]. Available: http://www.cloudcaptech.com/piccolo_system.shtm

[37] Cloud Cap Technology Piccolo Nano. [Online]. Available: http://www.cloudcaptech.com/Sales%20and%20Marketing%20Documents/Piccolo%20Nano%20Data%20Sheet.pdf

[38] [Online]. Available: http://www.uasevent.com/cloud-cap-technology-launches-piccolo-nano-autopilot-for-small-uas/

[39] [Online]. Available: http://www.cloudcaptech.com/Sales%20and%20Marketing%20Documents/Piccolo%20SL%20Data%20Sheet.pdf

[40] [Online]. Available: http://www.cloudcaptech.com/Sales%20and%20Marketing%20Documents/Piccolo%20II%20Data%20Sheet.pdf

[41] Z. Deng, C. Ma, and M. Zhu, "A Reconfigurable Flight Control System Architecture for Small Unmanned Aerial Vehicles," in *Systems Conference (SysCon), 2012 IEEE International*, 2012, pp. 1–4.

[42] S. K. Kannan, A. A. Koller, and E. N. Johnson, "Simulation and development environment for multiple heterogeneous UAVs," in *AIAA Modeling and Simulation Technology Conference*, 2004.

[43] T. Bakker, G. L. Ward, S. T. Patibandla, and R. H. Klenke, "RAMS: A Fast, Low-Fidelity, Multiple Agent Discrete-Event Simulator," in *SCS Summersim 2013*, 2013.

[44] L. Wills, S. Sander, S. Kannan, A. Kahn, J. V. R. Prasad, and D. Schrage, "An open control platform for reconfigurable, distributed, hierarchical control system," in *Digital Avionics Systems Conference, 2000. Proceedings. DASC. The 19th*, vol. 1, 2000, pp. 4D2/1–4D2/8 vol.1.

[45] J. Ferruz, V. Vega, A. Ollero, and V. Blanco, "Reconfigurable Control Architecture for Distributed Systems in the HERO Autonomous Helicopter," *Industrial Electronics, IEEE Transactions on*, vol. 58, no. 12, pp. 5311–5318, 2011.

[46] M. Jovanović, D. Starčević, and Z. Jovanović, "Improving Design of Ground Control Station for Unmanned Aerial Vehicle: Borrowing from Design Patterns," in *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, 2010, pp. 65–73.

[47] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[48] [Online]. Available: https://launchpad.net/gcc-arm-embedded

[49] ST. (2013, Jun) STM32F405xx/STM32F407xx Datasheet (DM00037051). [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00037051.pdf

[50] ——. (2013, Oct) ES0182: STM32F405/407xx and STM32F415/417xx device limitations (DM00037591). [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/errata_sheet/DM00037591.pdf

[51] ——. (2013, May) AN2606: STM32^{TM}microcontroller system memory boot mode (CD00167594). [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/application_note/CD00167594.pdf

[52] ——. (2013, May) AN3155: USART protocol used in the STM32^{TM}bootloader (CD00264342). [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/application_note/CD00264342.pdf

[53] Texas Instruments. (2009, May) TS3A27518E Datasheet. [Online]. Available: http://www.ti.com/lit/ds/symlink/ts3a27518e.pdf

[54] ST. (2014, February) RM0090: STM32F405xx/07xx, STM32F415xx/17xx, STM32F42xxx and STM32F43xxx advanced ARM-based 32-bit MCUs (DM00031020). [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/reference_manual/DM00031020.pdf

[55] ——. (2011, Oct) AN3969: EEPROM emulation in STM32F40x/STM32F41x microcontrollers (DM00036065). [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/application_note/DM00036065.pdf

[56] Microchip. (2010) 25LC512 Datasheet (DS22065C). [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/22065C.pdf

[57] M. T. Leccadito, “A Kalman Filter Based Attitude Heading Reference System Using a Low Cost Inertial Measurement Unit,” Master’s thesis, Virginia Commonwealth University, August 2013.

[58] [Online]. Available: https://github.com/texane/stlink

[59] [Online]. Available: http://blog.frankvh.com/2012/01/13/stm32f2xx-stm32f4xx-dma-maximum-transactions/

[60] [Online]. Available: http://www.devicetree.org/Main_Page

[61] [Online]. Available: https://www.power.org/documentation/epapr-version-1-1/

[62] [Online]. Available: http://git.kernel.org/cgit/linux/kernel/git/jdl/dtc.git/

[63] J. C. McBride, “Flight Control System for Small High-Performance UAVs,” Master’s thesis, Virginia Commonwealth University, May 2010.

[64] [Online]. Available: http://www.flightgear.org/

[65] [Online]. Available: http://savannah.nongnu.org/projects/lwip/

[66] ST. (2013, July) AN3966: LwIP TCP/IP stack demonstration for STM32F4x7 microcontrollers (DM00036052). [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/application_note/DM00036052.pdf

[67] S. T. Patibandla, “Development of Mobile Ad-Hoc Network for Collaborative Unmanned Aerial Vehicles,” Master’s thesis, Virginia Commonwealth University, August 2013.

[68] [Online]. Available: http://nuttx.org/

[69] [Online]. Available: http://www.chibios.org/dokuwiki/doku.php