Development of Modular Real-Time Software for the TALARIS Lunar Hopper Testbed

by

Christopher J. Han

B.S. Aeronautics and Astronautics
Massachusetts Institute of Technology, 2009

Submitted to the Department of Aeronautics and Astronautics
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Aeronautics and Astronautics
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

Signature of Author: ....
Department of Aeronautics and Astronautics
May 6, 2011

Certified by: ....
David W. Miller
Professor of Aeronautics and Astronautics
Thesis Supervisor

Certified by: ....
Michael C. Johnson
Senior Member of the Technical Staff, Draper Laboratory
Thesis Supervisor

Accepted by: ....
Eytan H. Modiano
Associate Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

Development of Modular Real-Time Software for the TALARIS Lunar Hopper Testbed

by

Christopher J. Han

Submitted to the Department of Aeronautics and Astronautics
on May 19, 2011 in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Aeronautics and Astronautics

Abstract

Hoppers have recently emerged as a viable means for planetary exploration, and as with any new vehicle, significant testing is required to validate guidance, navigation, and control (GNC) algorithms. Furthermore, the structure, organization, and timing of the real-time software must be planned before software development begins in order to design an architecture which can match the needs and requirements of the vehicle as they evolve throughout its lifecycle. These issues are compounded in an academic environment, where software knowledge is not necessarily present and must be obtained and practiced before it can be applied. In addition, high student turnover rates can result in difficulty retaining institutional knowledge of the working software and causes further development delays while new students are trained. These problems were addressed by the TALARIS software team by implementing a flexible, modular software solution in LabVIEW on the National Instruments Real-Time Input/Output (RIO) board. After a brief introduction to the TALARIS testbed, the theory of flexibility and modularity is described as applied to the TALARIS software. In particular, the unique FPGA + PowerPC architecture and its importance to precise, real-time GNC execution are explored. Various software modules are isolated and analyzed, and several test cases are presented to illustrate the benefits of modular software with regard to development time, testing procedure, and debugging. Examples from software development, actuator characterization, and test campaigns illustrate the gradual evolution of the prototype software. Finally, a discussion of the conclusions from the work and future work is presented.

Thesis Supervisor: David W. Miller
Title: Professor of Aeronautics and Astronautics

Thesis Supervisor: Michael C. Johnson
Title: Senior Member of the Technical Staff, Charles Stark Draper Laboratory

Acknowledgements

First and foremost, I would like to sincerely thank the MIT Space Systems Laboratory and Draper Laboratory for funding my graduate work and making this thesis possible. The Draper Laboratory Fellow program has allowed me to start on an exciting new project, work amongst a great team, and experience the real world of space systems engineering. Thanks to my advisors Dr. David W. Miller and Michael C. Johnson, whose guidance and support for the past two years have been crucial to my growth as a graduate student. I would also like to recognize Lesley Yu and National Instruments for their gracious donation of hardware and software and unwavering support of the project.

Additionally, I would like to thank the TALARIS team for all their hard work. There are too many people to list them all, but I would like to recognize a few key individuals at the risk of leaving someone out. Dr. Hoffman has been a great PI and leader on the project, and it has been an honor to work under such a prestigious figure in the space community. Bobby Cohanim and Phillip Cunio have been absolutely the best systems engineers I have worked with in my six years at MIT. Their gentle but firm leadership has been the foundation of the TALARIS team, and they always had a knack for pushing just the right amount for maximum results. Special thanks go to Chris Wardman for many of the underlying themes and ideas that drove the development of the real-time software. Chris taught me about FPGAs, real-time programming, and pushed me to always stay one step ahead of the hardware team. Without his ideas and insight, this thesis would not have been possible.

Among the TALARIS students, Claas Olthoff stands out as a superman, tirelessly working 18hr days and consistently taking the last T home. His work ethic was an inspiration, and I learned much from his original software scheme even after he had left. Thanks to my only 2011 classmate on the TALARIS team, Sarah Nothnagel, for going through the pains of grad school together and providing the occasional comic relief. I will always remember the late-night pset sessions that we had. Special thanks go to my software grasshopper, Chris Rossi, for graciously accepting the LabVIEW torch and easing my workload. I hope your LabVIEW skills prosper and surpass my own. Good luck to the next DLF generation: Chris, Joe, and Ted! For anyone else on the team I have not mentioned, my sincerest thanks for all the hard work and good times. Your contributions have not gone unnoticed.

Last but not least, I would like to thank my friends and family for supporting me throughout my time at MIT. I might have gone crazy had it not been for your welcome distractions. Thanks Mom and Dad for all your love and support throughout the years, for always encouraging me to chase my dreams, and for pushing me to realize my full potential. Your teachings since I was a child, along with God's blessings, have made me into the person I am today and have allowed me to succeed in life. I dedicate this thesis to you and offer my hard work and achievements until this day as a representation of my eternal gratitude.

Assignment

Draper Laboratory Report Number T-1682

Christopher J. Han

Contents

List of Figures and Tables

Acronyms

AI – Analog Input
AO – Analog Output
CGS – Cold Gas System
DI – Digital Input
DIO – Digital Input/Output
DMA – Direct Memory Access
DO – Digital Output
DOF – Degree of Freedom
EDF – Electric Ducted Fans
FIFO – First In First Out
FPGA – Field Programmable Gate Array
GLXP – Google Lunar X PRIZE
GNC – Guidance, Navigation, and Control
GSC – Ground Station Computer
IMU – Inertial Measurement Unit
LLRV – Lunar Landing Research Vehicle
LLTV – Lunar Landing Training Vehicle
LSB – Least Significant Bit
NGL – Next Giant Leap
PPC – Power PC
PWM – Pulse Width Modulation
RIO – Real-Time Input/Output
TALARIS – Terrestrial Artificial Lunar And Reduced gravIty Simulator
TRL – Technology Readiness Level

Chapter 1

1 Introduction

Over the past two decades, rovers have been the primary vehicle architecture for mobile ground-based planetary exploration. With the success of Sojourner in 1996 and the Mars Exploration Rovers (MER) in 2003, NASA has gained valuable flight experience and reliability with the rover architecture and plans to launch its largest rover yet, the Mars Science Laboratory (MSL) in 2011. However, ground-based rovers depend on automatic path-planning algorithms which are computationally expensive and require long calculation times on radiation-hardened processors. For example, a typical 15m traverse for the MER rovers took an average of three minutes of processing time from image acquisition to position update, which limited the overall speed of the rover to approximately 10 m/hr [1]. The calculation time coupled with the communication lag and rough terrain severely limited the speed and range in which the rovers could operate. In 555 sols (1 sol = 1 Martian day = 1.02 Earth days), Opportunity traveled just 5974m [2]. These limitations of rovers have motivated a desire to use a hopper architecture for mid-to-long range traversals, which is potentially capable of traversing kilometers in a matter of minutes, instead of months or years. A planetary hopper uses impulsive jets to launch itself into either a ballistic trajectory or a fixed-attitude, rectangular flight profile and land softly in the desired location. By flying over rough terrain, hoppers may visit scientifically interesting locations otherwise unavailable to rovers, such as the interior of craters or valleys. Hoppers that use a fixed-attitude flight profile can also perform science during the traverse, for example while scaling a cliff to observe geologic features. In addition, a hopping system could be added to an existing lander platform to augment the accuracy of its entry, descent, and landing (EDL) system and potentially lower overall cost. Insertion from orbit could be performed with a larger error ellipse than conventional EDL systems, using a small hop to fine-tune the final landing site. Such planetary hoppers have already begun to garner interest for the deployment of a seismic sensor network, which requires sensors to be separated by long distances and would otherwise be costly and time-consuming for a rover to deploy [3].

Despite all their benefits, hoppers have little to no flight history and require significant prototyping and testing to demonstrate flight maturity. TALARIS, or Terrestrial Artificial Lunar And Reduced gravIty Simulator, was designed to be an Earth-based testbed to prototype hopper technologies and test guidance, navigation, and control (GNC) algorithms. Using TALARIS, hopper GNC algorithms may be validated in a relevant environment on Earth for a fraction of the cost of a space-qualified vehicle. The Next Giant Leaps team intends to use the results from TALARIS to develop its entry for the Google Lunar X-Prize, a $30 million competition to travel to the moon, traverse a set distance, and send back data, pictures, and video.

1.1 Motivation and Approach

The main value of TALARIS as a testbed is its ability to quickly adapt to a series of different contexts and testing configurations. Rapid changes in vehicle configuration are expected, and the architecture must be flexible enough to accommodate a variety of GNC algorithms, flight profiles, acceleration and deceleration commands, gravity offsets, and test stands. In order to achieve these goals, the need arose for a software architecture that is flexible, easy to learn, and which facilitates a short development cycle. Modularity was applied to the real-time software, allowing different components of the software to be developed in parallel. Also, these modules were designed to be easily switched out depending on the vehicle configuration, allowing a customized suite of sensors and functions for each test campaign.

In addition to making the software flexible, the reliable, real-time execution of GNC algorithms can be a challenging task. There is an inherent tradeoff between achieving faster control cycles, and thus the possibility of finer control of the vehicle, and computation time, communication delay, and actuator delay. Furthermore, most GNC algorithms rely on precise timing information to determine the vehicle state and issue commands, but also require complex calculations within a short control cycle. These problems were addressed through the use of a field programmable gate array (FPGA) in conjunction with a PowerPC processor to achieve both the timing needs and the calculation requirements for the hopper GNC algorithms.

The objective of this thesis will be two-fold. First, it will detail the architecture of the real-time software necessary for the robust and reliable execution of GNC algorithms, resistant to jitter, communication delays, and other runtime irregularities. Second, it will document the use of modularity in the development, operation, and maintenance of the software and provide examples from each of the three phases to demonstrate the benefits of modularity.

1.2 Literature Review

1.2.1 Flexibility Theory

Flexibility theory studies the ability of a system to respond to a change in the environment of a system while minimizing the costs of such changes. Much of the literature on flexibility is currently focused on large scale, capital-intensive projects, such as energy or transportation infrastructure, where high investments make it essential to use flexibility in systems design. Lin, et al. [4] noted four characteristics which make such systems difficult to design: long lifetime, uncertain market conditions and performance, multidisciplinary scope which could lead to emergent behaviors, and significant economic and societal impact. Decisions about system architecture must be made at an early stage and under highly uncertain environments, and high capital investment accentuates the risk. As an example, Iridium and Globalstar pioneered space-based telephony in the late 1990s, investing millions in a satellite communications network. However, neither company accounted for the rise of ground-based cellular technology in the mid 1990s, which lowered demand for space-based communication networks. The deterministic market predictions by Iridium and Globalstar as well as the inability to downsize and reconfigure its network for a different purpose resulted in losses of $5 billion and $3.5 billion respectively [5]. Instead, de Weck, et al. [5] suggest using a staged deployment, starting with a smaller, more affordable network and adding capacity by launching additional satellites and reconfiguring the constellation in orbit. In such large-scale projects, flexibility is driven by the need to adapt to future uncertainty and improves lifecycle value by mitigating downside risks.

Smaller academics projects, such as TALARIS, also benefit from employing flexibility to mitigate uncertainty. However, while large-scale projects tend to focus on changing stakeholder interest and market dynamics, uncertainty in smaller projects usually stems from changes during the development process such as shifting requirements definitions and budget and schedule constraints. Furthermore, academic projects often suffer from high personnel turnover rates as student schedules shift, and keeping institutional knowledge of the system can be a challenging task. This can be mitigated by maintaining rigorous documentation, but fluctuations in student availability stills adds additional uncertainty to future development potential.

The TALARIS theory of flexibility, first described by Cunio [6] and applied by Olthoff [7], attempts to make flexibility theory applicable to small, advanced vehicle development. Cunio describes several methods to impart flexibility to a system, each designed to either maximize the system's ability to change or minimize the costs to the system, whether they be monetary, schedule, personnel time, or complexity. These methods will be discussed in further detail in Chapter 3.

1.2.2 Modularity

Modularity has been applied for years in engineering. In 1995, Ulrich wrote a seminal paper exploring the connection between modularity in product architecture and manufacturing firm performance [8]. Ulrich defined modular and integral architectures as the following:

> "A modular architecture includes a one-to-one mapping from functional elements in the function structure to the physical components of the product, and specifies de-coupled interfaces between components. An integral architecture includes a complex (non one-to-one) mapping from functional elements to physical components and/or coupled interfaces between components" [8].

Ulrich explored the advantages of each architecture in several areas of "managerial" importance, such as product change, product variety, component standardization, product performance, and product development management.

More recently, both Gaillard [9] and Holtta [10] explored the impact of applying modularity to the automotive assembly line. Gaillard urged automotive manufacturers to adopt open standards on the assembly line and to reduce complexity through modularization. Holtta explored the tradeoff between different levels of modularity and noted that modularity may often come at a cost that engineers are not willing to pay. For example, light weighting, tight packaging, and lower power consumption may drive engineers towards more integral, less flexible architectures [10].

Software modularity is not a new idea, but the consistent application of modularity can sometimes be a challenging process. Cai and Huynh [11] stated that both aspect-oriented and object-oriented programming techniques were intended to allow one part of the software to change independently of the rest of the system. In addition, a modularization technique benefits a design only if future changes to the design can be accommodated by the technique. Thus, the application of a specific technique should be evaluated against potential future changes [11]. Tan [12] described a method which uses the Larch/C Interface Language (LCL) to encourage modular style programming by developing a formal framework with specified interfaces. LCL was used to specify modules in existing software and to drive a reengineering process which improved modularity and robustness without changing the base functionality or performance [12].

1.2.3 FPGAs in Real-Time Software

In the past two decades, FPGAs have emerged as a viable means to program flexible real-time software for small robotics. In 1995, Corba and Ninkov [13] implemented a 2D real-time image centroiding algorithm on an FPGA, noting the high parallelism, pipelineability, and modularity of the FPGA architecture. Several distinct filter configurations were pre-compiled and stored on the FPGA allowing for quick swapping after deployment. These characteristics also make the FPGA attractive for implementing GNC algorithms on experimental small robotics, where the creation of customized embedded systems can be costly and time consuming. Falsig and Soerenson [14] described a modular architecture for low level control called TosNet, which implemented modular controllers over a standard network in the FPGA to control up to 15 nodes. KrishnaKumar, Kaneshige, et al. [15] used a FPGA in a similar way, describing a "plug and play" avionics system called iPapa. The goal for iPapa was to allow new hardware to be plugged into the system and automatically configured without the need for a manual setup procedure [15]. Again, this type of architecture was valuable by avoiding the need to design individual avionics solutions for different vehicles or configurations.

All of these examples stress the reconfigurability and parallelability of the FPGA to design modular interfaces on a standard framework. However, the implementation of more complicated GNC algorithms in a FPGA remains difficult and takes up an enormous amount of resources in the FPGA. For example, an accurate divide operation consumes an immense amount of resources in an FPGA compared to a traditional processor, and much research has been done to develop an efficient division algorithm using the FPGA's fixed-point logic [16]. In addition, VHDL, the standard programming language used in FPGAs, is difficult to learn and understand for inexperienced coders. In an academic environment, this translates to valuable development and operation time spent on training new personnel. This thesis presents a new software architecture which combines a FPGA and a traditional PowerPC processor to take advantage of their respective strengths while minimizing their weaknesses. In particular, the parallelism and reconfigurability provided by the FPGA will be used to develop modular sensor and actuator interfaces, and the processing power from the PowerPC will be used to execute GNC algorithms. The software will be written in the LabVIEW environment, which is both easy to learn and understand, and the benefits of modularity will be demonstrated in different phases of the vehicle development.

1.3 Thesis Overview

This thesis is divided into six chapters. Chapter 2 provides background information regarding the Google Lunar X-Prize as well as a system overview of the TALARIS testbed. Chapter 3 introduces the theory of flexibility and modularity as they have been applied to the vehicle software and forms the theoretical foundation for the thesis. Chapter 4 begins with a description of the TALARIS avionics hardware and a discussion of GNC algorithm execution and timing, and finishes with an overview of the real-time software architecture as implemented in LabVIEW. In particular, the unique FPGA + PowerPC architecture and its importance to GNC execution will be explored in great detail. Chapter 5 focuses on the application of the modularity principle to the TALARIS software architecture. Various modules will be isolated and analyzed, and several test cases will be presented to illustrate the benefits of the modular software with regard to development time, testing procedure, and debugging. Examples from software development, actuator characterization, and test campaigns will illustrate the gradual evolution from prototype to flight software. Finally, Chapter 6 will conclude the thesis with a summary and recommendations for future work.

Chapter 2

2 Background

2.1 Google Lunar X PRIZE

The Google Lunar X PRIZE (GLXP) is an international competition organized by the X Prize Foundation and sponsored by Google to motivate privately funded teams to land on the moon, traverse at least 500m, and send back high definition images, video, and data. The first team to do so before the end of 2015 will receive a $20 million grand prize, while the second team will get a $5 million second place prize. In addition, a number of bonus prizes worth up to $4 million will be awarded for various extra goals, such as traveling ten times the baseline requirement (5000m), verifying water ice on the moon's surface, surviving the lunar night, and precision landing near the Apollo landing sites or other sites of interest [17].

To compete in the Google Lunar X Prize, the Next Giant Leap (NGL) team was founded in 2007 by Michael Joyce. Its technical partners include the Sierra Nevada Corporation, Draper Laboratory, Aurora Flight Sciences, and the MIT Space Systems Laboratory. Unlike other teams competing for the GLXP, the Next Giant Leap team has decided to use hopping technology, rather than the more conventional lander with a rover, to achieve the GLXP requirements. Hoppers, which are not limited by rough terrain, have the potential to traverse longer distances than rovers, but have no flight heritage and thus are considered as a higher risk. The Next Giant Leap team is also interested in developing hopper technology in the long term to "provide transportation and support for a variety of science and commercial payloads to low gravity bodies such as our moon, the asteroids, and Phobos" [18]. However, as an untested technology, prototype hoppers must first be developed and tested, and Draper Laboratory, as a member of the NGL team, has been tasked to develop the guidance, navigation, and control (GNC) algorithms for the NGL vehicle. To test these algorithms in a relevant environment, the TALARIS prototype testbed was developed at Draper Laboratory in conjunction with the MIT Space Systems Laboratory.

2.2 Introduction to TALARIS

TALARIS was originally conceived in Spring 2008 by the 16.89 Space Systems Engineering graduate design course and represents an MIT/Draper collaboration to develop a terrestrial testbed for testing hopper GNC algorithms. Using terrestrial testbeds such as TALARIS, GNC algorithms may be tested for relatively low cost in a simulated environment on Earth before deployment. Similar to the Apollo program's Lunar Landing Research Vehicle (LLRV) and its successor the Lunar Landing Training Vehicle (LLTV), TALARIS uses a dual propulsion system: one as the primary impulsive propulsion and a second to provide gravity offset and simulate lunar conditions. In the case of the LLRV and LLTV, a gimbaled turbofan jet engine was used to provide weight relief, and hydrogen peroxide rockets were used to simulate the thrusters on the Lunar Module [19]. On TALARIS, four electric ducted fans (EDF) are used to relieve 5/6 of the vehicle's weight, leaving the rest of the vehicle to experience 1/6th of the Earth's gravity, as it would on the Moon. The vehicle uses nitrogen cold gas thrusters as the primary propulsion system to provide impulsive thrust. A CAD model of the second generation vehicle can be seen below in Figure 1.

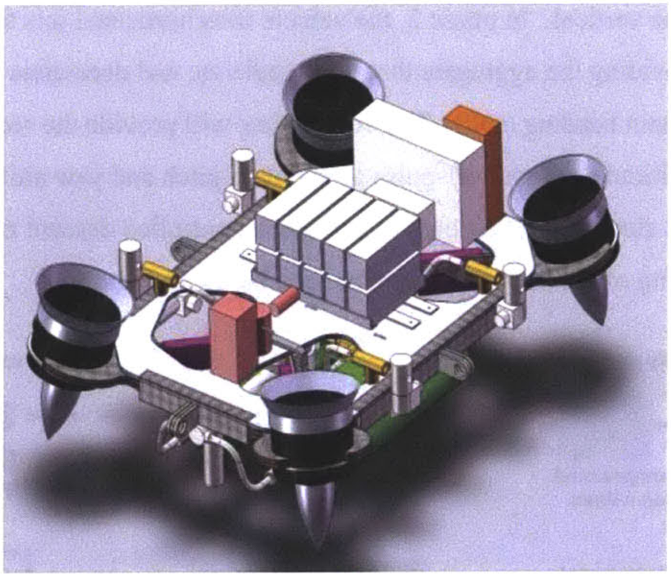


Figure 1: SolidWorks Drawing of TALARIS Testbed v2

In addition to simulating a lunar environment, the TALARIS platform was designed to simulate other astronomical bodies, including Mars, Phobos, and asteroids, as long as they have gravity lower than Earth. Furthermore, each of the propulsion systems can be switched out for more powerful upgrades, such as gas-turbine engines to replace the EDFs or hydrogen peroxide/hydrazine thrusters for the cold gas system (CGS). The avionics hardware was designed to be flexible as well, to accommodate these configuration changes [7]. This thesis will concentrate mainly on the development of real-time software to take advantage of the hardware flexibility, allowing streamlined testing on a variety of platforms and test campaigns.

By using the nitrogen cold gas thrusters with gravity offset from the EDFs, TALARIS will perform a level horizontal hover hop of 30m. In a trade-off study, 30m was chosen as a representative 1g demonstration of a lunar hover hop [20]. The distance traversed is much lower than the GLXP requirement of 500m because of higher gravity, air resistance, and the lower specific impulse of nitrogen gas thrusters compared with hydrazine thrusters. The hop will be performed in three phases. First, the vehicle will operate the EDFs in steady-state to provide 5/6g offset. The four vertical gas jets will provide the necessary thrust to ascend to 2m and obtain stable hover, while the four horizontal gas jets will provide roll control about the vertical. In phase 2, the vehicle uses horizontal jets to perform straight and level flight, providing the aggregate thrust to accelerate and decelerate laterally while off-pulsing to maintain heading (roll). The vertical jets will provide the required aggregate thrust to maintain altitude and will off-pulse to maintain pitch and yaw attitude. In phase 3, the vertical jets will slowly throttle down to perform a controlled descent and touchdown [21]. A conceptual drawing of the hop profile can be seen below in Figure 2.

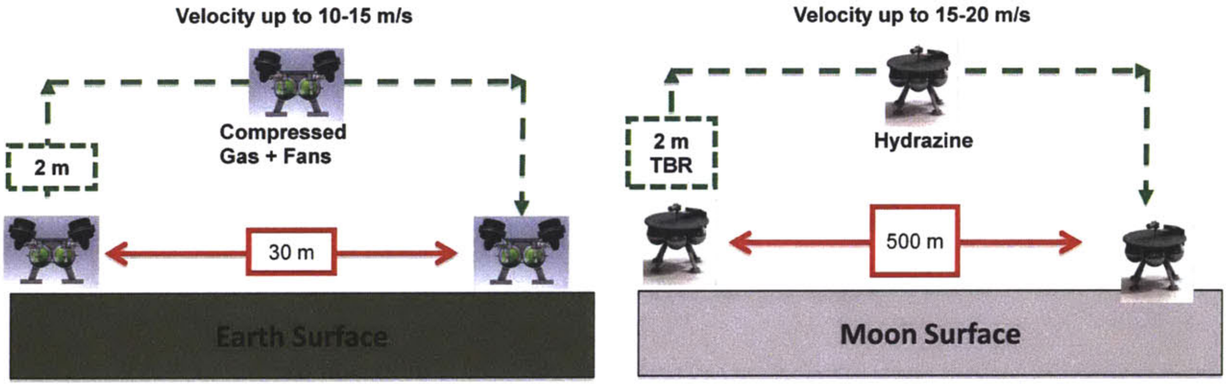


Figure 2: TALARIS (left) and GLXP (right) hop profiles

This hop profile is intended to recreate a lunar hop by providing similar forces and torques on the vehicle, as well as allowing the GNC algorithms to perform a hover hop in a controlled environment. It is important to note that the TALARIS vehicle is not the same as the final NGL vehicle and was never intended to be flight hardware or sent to the moon. TALARIS was meant for operation on Earth using analogues to the final vehicle to prove GNC algorithms.

2.3 TALARIS Systems Overview

2.3.1 Structures

The TALARIS v2 structure is composed of a single flat sheet of carbon fiber composite with additional ribbing on the underside for added strength and cutouts to reduce mass. Custom machined EDF mounts are located on the four corners, canted at 15° for controllability. The structure measures 99cm long by 76cm wide by 8.9cm tall (frame only). A picture of the underside of the carbon fiber body can be seen below [21].

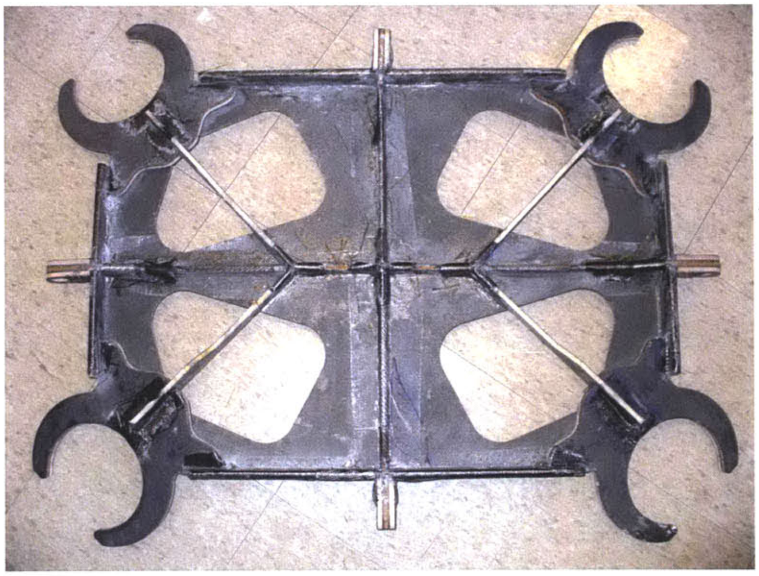


Figure 3: TALARIS v2 carbon fiber body (underside) [21]

The axis conventions on the vehicle are consistent with the Draper GNC axis conventions, which are derived from the original Apollo coordinate system. In this coordinate frame, +X is directly up from the vehicle, +Z is in the direction of the horizontal thrusters, and +Y completes the right-handed coordinate system. "Roll" will be referred to as rotation about the X axis, "Yaw" will be rotation about the Z axis, and "Pitch" will be rotation about the Y axis. These axes will be referenced in later sections and are reproduced below with respect to the body orientation.

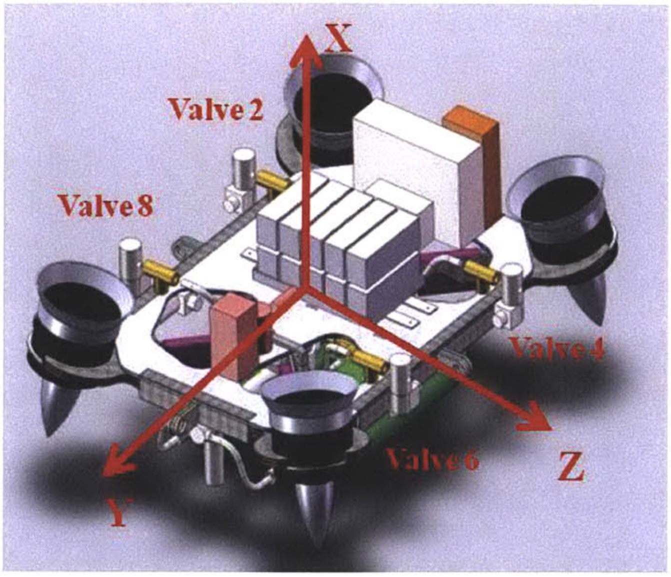


Figure 4: TALARIS Axis Conventions

2.3.2 Electric Ducted Fans (EDFs)

The TALARIS test bed uses four Aero-Naut TF8000 electric ducted fans to provide the 5/6 gravity offset. The EDFs are powered using Lehner 3060 fan motors and controlled with Schultz 40.160 motor controllers. The motors have a max rated power of 8kW, with max thrust at 6.37kW of power [20]. With an operating voltage of 45-50V from the lithium polymer batteries, the max current draw of the EDFs is 150A. In addition to the motor casing, custom inlets and fairings were designed to reduce turbulence and increase efficiency. These were fabricated using stereolytographic (SLA) 3D rapid prototyping printing. With the inlet and the fairing, each EDF produces about 100N of thrust at max thrust [20]. An exploded view as well as a completely assembled EDF can be seen in the figures below.

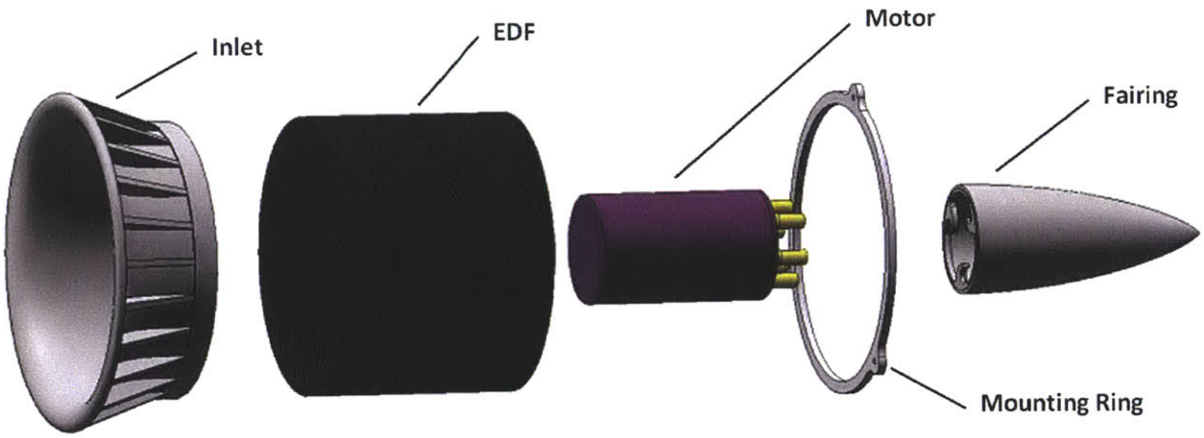


Figure 5: Exploded view of Aero-Naut TF8000 EDF with custom inlet and fairing [20]

Figure 6: Front and Back views of the assembled EDF [20]

2.3.3 Cold Gas System

The TALARIS Cold Gas System (CGS) provides the final 1/6g thrust required for takeoff as well as attitude, altitude, and horizontal control. The liftoff requirement is to traverse 2m vertically in 2s, which requires 486.4N with a 45kg vehicle. The EDFs provide 367.9N while the CGS provides 118.5N of vertical thrust [22]. Eight Omega SV128 thrusters are used on the vehicle overall, with four providing vertical thrust and four providing horizontal thrust. The valves have a rated opening and close time of 30-60ms, but through the use of a custom PCB, the open and close lag have been reduced so that the minimum overall pulse on-time or off-time is 40ms [23]. The valves will be controlled at 5Hz. Pulse-width modulation (PWM) with pulsewidths between 40 and 160ms will be used to control vehicle translation, roll during traverse, pitch, and yaw. The same range of pulsewidths will be used along with a phase plane controller to control roll during vertical rise, hover, and vertical descent; however, in this case a 200ms pulsewidth will be allowed in order to make continuous firing over consecutive Control cycles possible. The Luxfer L65G flight tanks will be filled to 4500psi, providing a flight time of 44 thruster-seconds (e.g. 4 thrusters for 11 seconds). During vertical firings on a load cell, the valves provided an average of 58N per thruster, but this value can be affected by a variety of factors including runtime, tank pressure, temperature, and multiple valve firings [22]. An Omega SV128 can be seen below, as well as a figure showing the valve orientation and numbering on the vehicle. The vertical thrusters (VTs, corresponding to valves 1, 3, 5, and 7) nominally create upward thrust, while the horizontal thrusters (HTs, corresponding to valves 2, 4, 6, and 8) nominally create lateral thrust (2 and 8 provide +z thrust, 4 and 6 provide –z thrust).

Figure 7: Omega SV128 solenoid valve [24]

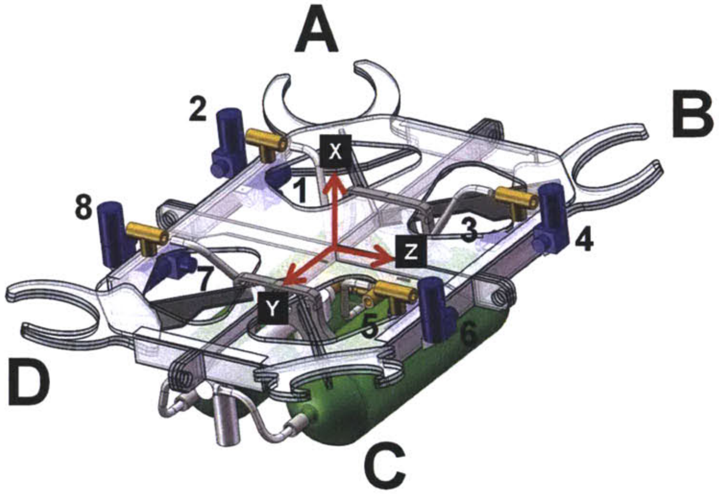


Figure 8: Valve orientation and numbering convention

2.3.4 Lithium Polymer Batteries

To power the EDFs, the vehicle uses ten Tanic 7S-1P 4500mAh lithium polymer battery packs which are capable of discharging continuously at 135A. Two battery packs are wired in series to provide a nominal voltage of 51.8V, and five sets are wired in parallel to share the current draw. At full throttle, the high power system is capable of delivering 25kW to the EDFs [20]. During testing sessions, runtime is generally limited to below 45s at full throttle to prevent overheating of the batteries and motor controllers.

2.3.5 Altimeter

An Acuity AR1000 laser altimeter will be used to sense altitude and provide an update of the vertical degree of freedom. The AR1000 has a range from 0.1m to 30m with an accuracy of 3mm. It is being operated with a 10Hz update rate and communicates using the standard RS-232 communication protocol [25]. In the software, altimeter packets are being treated as asynchronous because the altimeter does not use a sync signal to the RIO, so it does not send packets at exactly equal 100ms intervals.

Figure 9: AR1000 Laser Distance Sensor [25]

2.3.6 Gladiator IMU

The Gladiator LandMark 30 IMU was chosen to navigate the vehicle due to its high performance, low noise and bias characteristics, and small size. The IMU uses RS485 to communicate data packets at a rate of 200Hz. These packets are synchronized to the RIO's clock using a 1kHz square wave, generated by the RIO's FPGA.

Most degrees of freedom will be determined by dead reckoning, so the flight will have to be kept short to prevent IMU noise from causing the state estimate to diverge. Alternatively, two additional upgrades to the navigation algorithm are planned to improve performance. The first will use four altimeters mounted at the corners of the vehicle to provide an attitude update, and the second will use a downward pointing camera to implement vision navigation algorithms. Both updates are currently being prototyped and are not yet implemented on the vehicle.

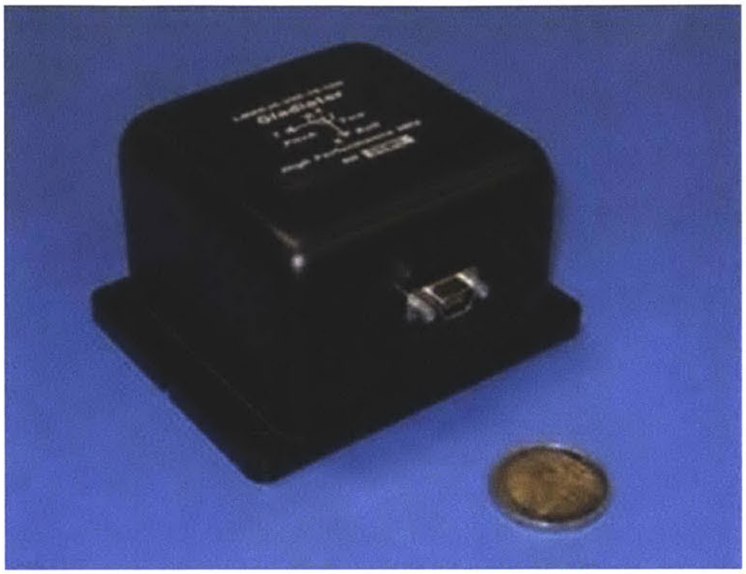


Figure 10: Gladiator LandMark 30 IMU [26]

2.3.7 National Instruments sbRIO-9642 Board

The National Instruments sbRIO-9642 (RIO) board handles all input/output functions on the vehicle and communicates telemetry data to the Ground Station Computer (GSC). This board was chosen for its low cost, small size and weight, high flexibility, and ease of development with real-time LabVIEW. At its heart is a 400MHz Freescale MPC 5200, a member of the PowerPC 5000 series of microprocessors. The processor runs the LabVIEW real-time module on the Wind River VxWorks real-time operating system and is connected to a 2M gate Xilinx Spartan-3 field programmable gate array (FPGA) through a high-speed PCI bus. The FPGA is connected to a number of digital and analog input/output choices, including 110 3.3V bidirectional digital input/output (DIO) channels, 32 24V digital input (DI)/digital output (DO) channels, 32 ±10V analog input (AI) channels, and 4 ±10V analog output (AO) channels [27]. A custom aluminum case was built for the RIO with standard D-shell connectors attaching to the various input/output lines with ribbon cable. The RIO measures 8.2" x 5.6" and weighs 292g [27].

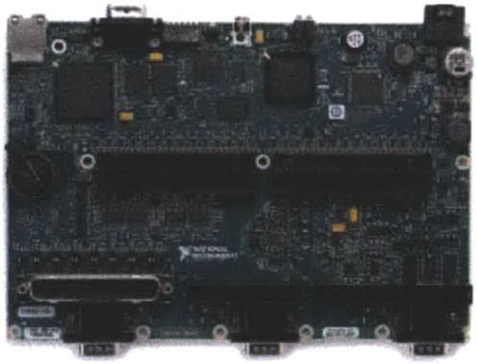


Figure 11: National Instruments sbRIO-9642 [27]

The RIO was chosen for its inherent flexibility, given the numerous input and output choices [7]. Of particular interest to this thesis is the ability to integrate c-code libraries into the LabVIEW code and run them as enclosed blocks. This feature allowed for the modularization of the Draper GNC code so that development could be done for each of the separate GNC blocks. The separate algorithms could then be compiled together with an executive and enclosed in a single LabVIEW block.

Chapter 3

3 Flexibility and Modularity

3.1 TALARIS Theory of Flexibility

> "Flexibility is a property of a system by virtue of which the system changes to gain maximum value in response to a change in the environment for the system" [6].

Flexibility is mainly a tool to deal with uncertainty. Small advanced vehicles are often affected by uncertainties in the development process, especially prototype vehicles, where changes in requirements and the project environment occur frequently and unexpectedly. In these projects, flexibility can add downstream value in two ways: by maximizing the system's capability to change or minimizing the cost of a change, whether the cost be monetary, personnel time, or schedule. As noted by Olthoff [7], software is inherently flexible in the sense that any additional changes require no changes in hardware or monetary costs. However, personnel and schedule costs must also be taken into account when making a software change, and a flexible software architecture will make these changes easy, quick, and seamless. Therefore flexibility in software will focus on minimizing the cost of a change, namely in terms of personnel time and schedule.

The TALARIS Theory of Flexibility as written by Cunio [6] states three techniques for imparting flexibility to small advanced vehicles. The first technique, maximum overhead capacity, attempts to provide more of a specific resource than is minimally required, essentially creating an overdesigned system. For example, the avionics system on TALARIS has more than twice the number of I/O pins necessary for the current sensor suite. The small amount of added mass from the extra pins must be weighed against the ability of the avionics system to incorporate a larger and more varied sensor suite in the future. The second technique, creating a defined expansion path, attempts to predict possible future development paths to target system changes. By defining a discrete number of future possibilities, the subsystems can be catered to adapt to the most probable choices or the widest possible subset of choices, thereby minimizing future cost and maximizing utility. The third technique, modularity, will be the focus of this thesis. In general, modules are designed to have specified, de-coupled interfaces, making it easy to interchange modules to form different configurations. If a change in the system function is required, this change can be isolated to a few modules, leaving the rest of the system intact. In this way, the system is maximizing its ability to change (e.g. incorporating a new sensor suite) while also minimizing the time and complexity of this change. This technique is the easiest to apply to software, as there is no "resource" to be maximized and future paths can be redesigned more efficiently with modular software. In this way, modularity can be thought of as an "enabler for flexibility" [7].

3.2 Modularity

> "A modular architecture includes a one-to-one mapping from functional elements in the function structure to physical components of the product, and specifies de-coupled interfaces between components" [8].

Modularity provides a link between the physical structure and functionality of the code, increasing organization and reducing complexity. The code is separated into modules, which are defined from the rest of the system by de-coupled interfaces. Modules should have little to no dependence on other modules, allowing them to be switched, replaced, and omitted for different configurations. In some cases, having some coupling between modules can be unavoidable. For example, the “Current Time” variable, which keeps a 1ms timer, must be shared by all FPGA modules for accurate timestamping. In the case of such interactions, the couplings must be well documented so that the module may still be replaced or removed without impacting the rest of the system.

When developing vehicle software with a large team, modularity can help maximize utility by breaking up the software into smaller tasks. Each module can be tested and validated individually, and several modules can be easily assembled into different configurations. When debugging, the problem can be isolated to a single or group of modules and debugged separately from the rest of the system. In this way, the development of the complete software can proceed in parallel, with different groups or individuals working on different modules. This scheme also decreases the complexity for students, since coders don’t necessarily need to understand the entire software to code their individual module.

In the operations phase, the development of prototype software, i.e. software that encompasses only partial applicability or that has not been fully tested for flight reliability, helps to demonstrate functionality early and buy down risk. For example, it is common on TALARIS to separate the EDF and CGS propulsion systems for individual testing. Prototype software can be quickly assembled using only the appropriate modules for each system to allow for characterization tests to continue. By having prototype software ready and quickly customizable, hardware and software work streams can proceed in parallel without one stream causing delays with the other. In addition, software functionality for each propulsion system can be validated individually, and the relevant modules can be later incorporated into the final version of the software.

Finally, modularity can also help in later phases of vehicle development. By increasing flexibility, modularity allows for the vehicle to undergo quick changes in configuration and minimize setup times. For example, common modules such as the RS-232 interface can be repurposed or reused for similar functions. New compatible sensors can be easily integrated by using a standard communication protocol such as RS-232. In the long term, if the TALARIS testbed were to be upgraded to use more powerful propulsion systems, such as hydrazine rockets or gas-turbine engines, the core framework of the software can be left in place, with only the relevant modules being replaced.

The benefits of modularity are summarized in the list below:

- Increased organization
- Increased flexibility while not compromising base functionality
- Separation of responsibility - makes debugging easier and facilitates development with large teams
- Reduced complexity - makes code easier to understand and is useful for gaining experience in an academic environment
- Ability to easily switch out GNC algorithms speeds testing sessions
- Reusable - common modules can be easily modified for similar functions (i.e. RS-232 interface)

The detriments to modularity are a bit difficult to quantify. It is usually difficult to convert integral code to modular code *a posteriori*, which means a modular coding style must be adopted before development begins. During development, it is unclear whether integral or modular code requires longer development times. In some instances, the reduced complexity of modular code may aid programmers, but it may also create some redundancy which increases the runtime or amount of resources used. The tradeoff between performance and development time is central to the decision of using integral or modular code.

3.3 Modularity in LabVIEW

The LabVIEW programming language was chosen for the TALARIS project due to its ease of use and implementation for modular embedded systems. The intuitive graphical interface is easy to learn and understand for new students, which is crucial for passing on software knowledge to the next generation. In addition, the graphical interface is inherently suited for modularity, as sections of code can be visually separated and further compartmentalized through the use of sub-vi's. The graphical user interface (GUI) is integrated into the creation of the code as a "front panel" which is paired with every vi block diagram. However, there are also some tradeoffs associated with LabVIEW. As a graphical programming language, all modules attempt to run as soon as inputs are available. Runtime execution is implied by the passing of information between modules (i.e. a module will not run until all inputs are defined) or explicitly through the use of sequences. This makes multi-threaded programming relatively easy – by default, the code will run as many modules as possible in parallel. However, this makes serial programming more difficult. The order in which modules execute is often crucial to proper performance, but this is not specified by LabVIEW unless explicitly controlled by the programmer. For many programmers used to programming in text-based languages such as C or Matlab, serial programming may be taken for granted, and some amount of adjustment may be required to enforce runtime execution order. For the TALARIS project, LabVIEW acts as a higher level environment common to both the FPGA and the PowerPC (PPC) and represents a compromise between usability and complexity. For other projects, the traditional text-based methods for programming a FPGA and PPC, such as VHDL and C, might be more attractive choices.

There are four main features of LabVIEW which make it especially suitable for developing modular code. The first feature described below is the ability to create sub-vi's, since it is possibly the most useful in terms of modularity. The other three features, described in the other subsections below, are also helpful in this regard. The graphical nature of LabVIEW makes clear the mapping between the functional and the physical structure of the code. The availability of sequences eases organization of the various modules into a cohesive body of code, and the Call C Library Function Node provides the capability of seamlessly calling pre-compiled C code as separate modules.

3.3.1 Creating sub-vi's

A sub-vi is a section of code which has been compartmentalized and is represented by a single icon on the block diagram of the top-level vi. Sub-vi's are most commonly used to represent functions, where inputs and outputs are well defined, and may be copied to produce multiple instantiations as needed. Sub-vi's can also be used to capture more complex code, which is useful in making the top-level vi less complicated and more readable. Below is an example of a sub-vi which handles the GNC execution in the PPC.

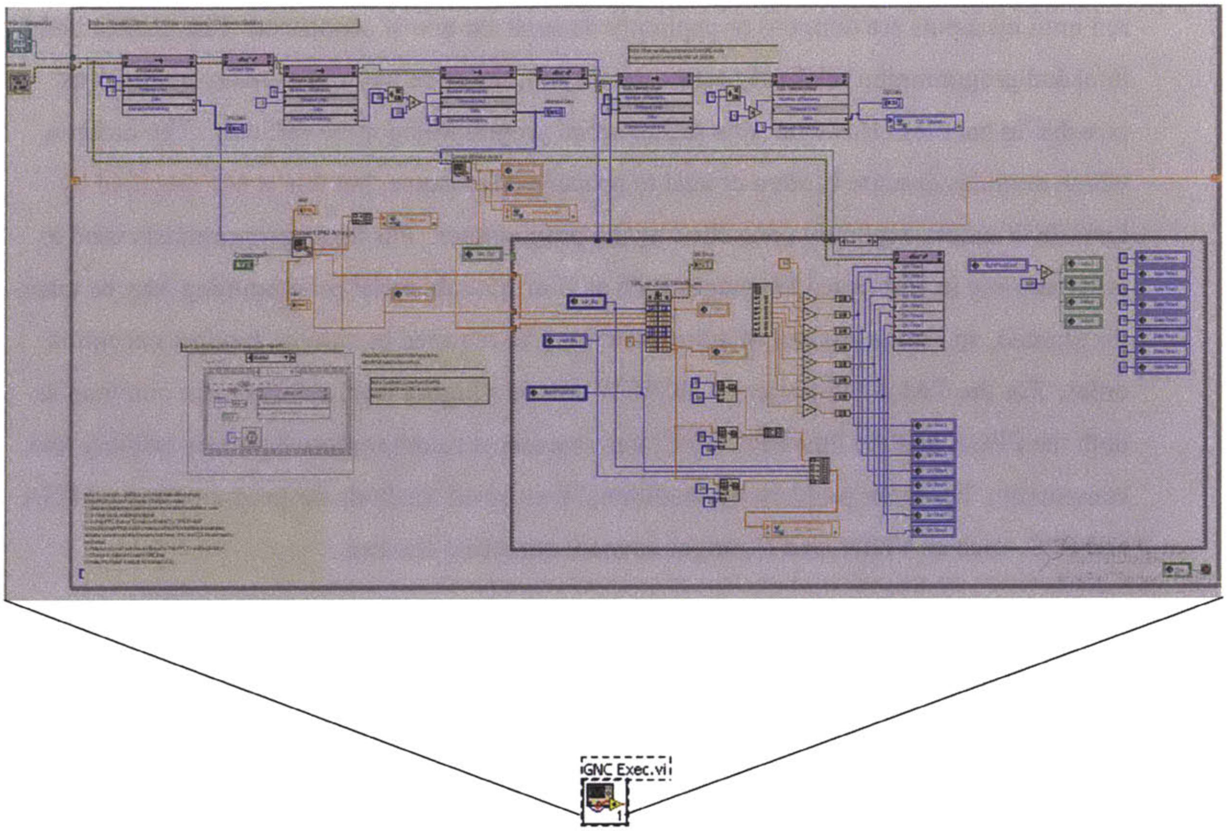
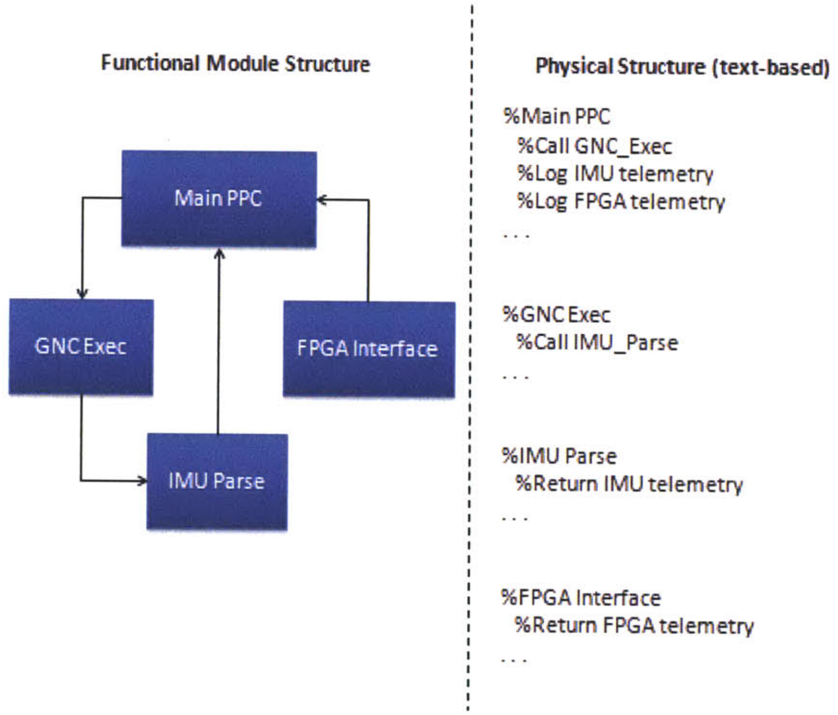


Figure 12: Capturing complexity with a sub-vi

Sub-vi's hide complexity from the top-level vi, increasing readability in both vi's, while simultaneously modularizing the code. On the top level, the sub-vi can be thought of as a black box with only inputs and outputs. When the code needs to be debugged, the bug can be narrowed to a certain sub-vi and modified separately from the main code. Sub-vi's also naturally specify module interfaces, as the inputs and outputs must be pre-determined to pass information into the sub-vi (the sub-vi and top-level vi have different variable scopes). Although a vi can often be thought of as a module, a module is not necessarily a single vi. There may be several modules inside a single vi, as is the case in the FPGA software, where all the sensor interfaces are contained within a single vi in separate modules.

3.3.2 Mapping functional and physical structure

Due to the graphical nature of LabVIEW, a block of code can be immediately understood in terms of its relation to other blocks, and their interactions can be intuitively seen. In text-based code, even if the functions are well-separated, they are still written serially, and their execution order or tree is not immediately obvious. In the figures below, a simplified example has been taken from the PPC code. The functional block diagram is shown on the left while pseudo-code is shown on the right.



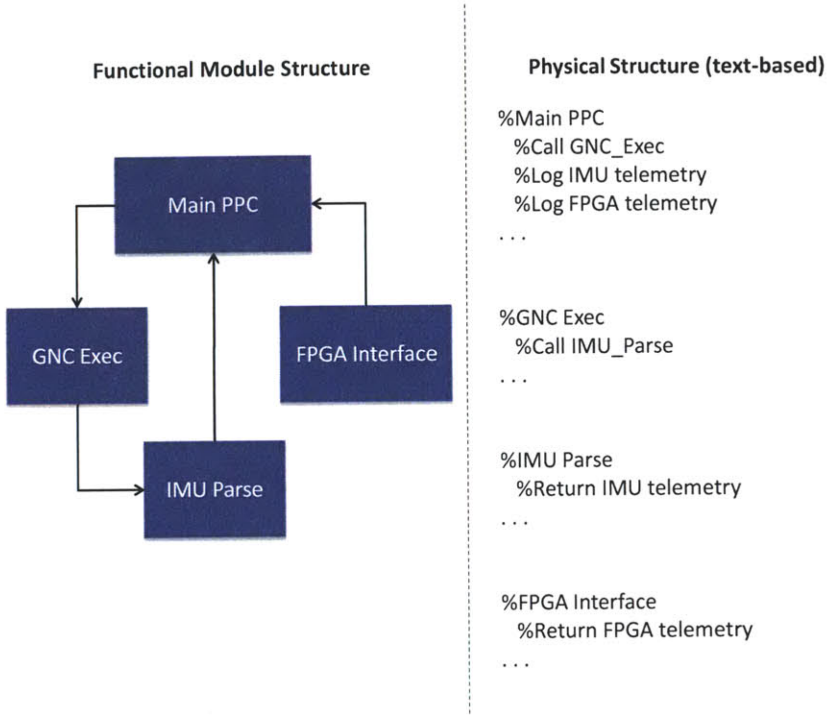
```
%Main PPC
  %Call GNC_Exec
  %Log IMU telemetry
  %Log FPGA telemetry
...

%GNC Exec
  %Call IMU_Parse
...

%IMU Parse
  %Return IMU telemetry
...

%FPGA Interface
  %Return FPGA telemetry
...
```

Figure 13: Mapping Functional and Physical Structure (text-based code)



Functional Module Structure

Physical Structure (text-based)

```
%Main PPC
  %Call GNC_Exec
  %Log IMU telemetry
  %Log FPGA telemetry
. . .


%GNC Exec
  %Call IMU_Parse
. . .


%IMU Parse
  %Return IMU telemetry
. . .


%FPGA Interface
  %Return FPGA telemetry
. . .
```

Figure 14: Mapping Functional and Physical Structure (LabVIEW)

In this example, Main PPC acts as the main executable, calling GNC Exec and receiving data from IMU Parse and FPGA Interface. In the text-based example, the programmer must keep the functional diagram in mind while coding text serially; even if the code is well-separated by function, their execution order is not immediately obvious. In LabVIEW, the programmer may make the actual code visually resemble the functional block diagram. Main PPC can be coded with an output which sends data to GNC Exec and two inputs which receive data from IMU Parse and FPGA Interface. The color of the line indicates the data type (e.g. double, int32, etc.) of the inputs and outputs, supplying further information to the programmer. In this way, the physical structure in LabVIEW can be made to closely resemble the functional structure, which is one of the defining qualities of modularity.

3.3.3 Using sequences to organize code

In LabVIEW, sequences are used to explicitly define runtime execution order. Sequences are composed of a series of frames which run in series and are arranged either horizontally, as in a flat sequence, or one behind the other, as in a stacked sequence. Frames also create natural boundaries which can be used to organize code and define modules. Information can flow into and out of a frame in much the same way inputs and outputs are defined in a sub-vi. All three of the above modularity techniques can be observed in the example below, taken from Flight Shell.vi.

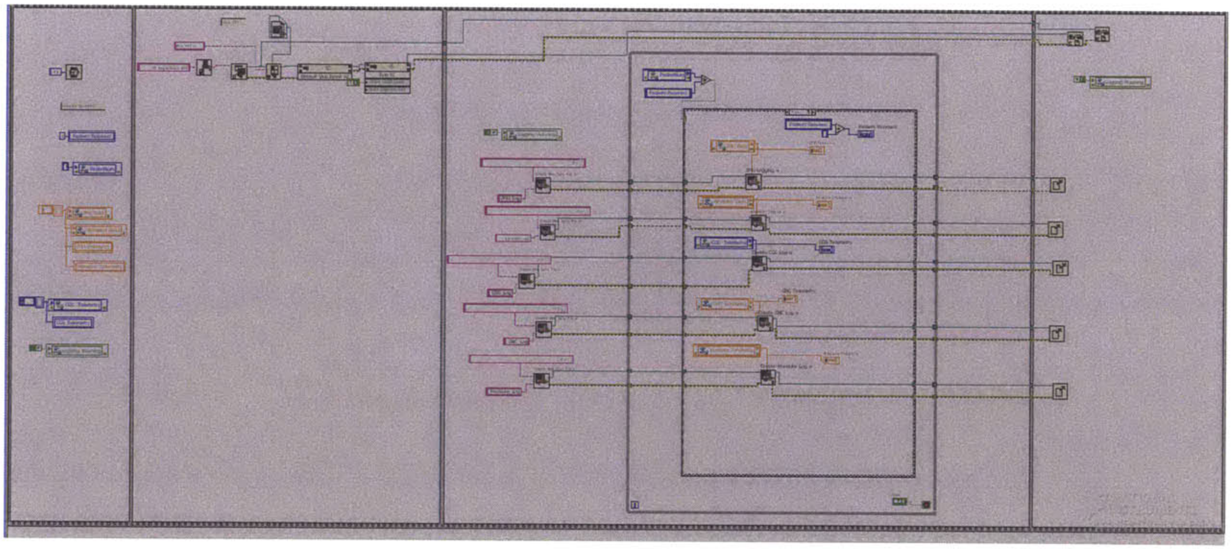


Figure 15: Overview of Flight Shell.vi

Flight Shell.vi is composed of a single flat sequence of four frames that are intended to be run in order. This ensures that the variable initialization, which happens in the first frame, happens before the PPC is called, which happens in the second frame. Logging happens in the third frame, with each of five logging text files being created by their respective sub-vi's. In the final frame, all the text files are closed and the PPC execution is stopped. In this way, each frame is defined by a specific function and separated by boundaries. Execution can be thought of as a block diagram: Initialization -> Start PPC -> Logging -> Stop PPC, with the physical structure matching its functional counterpart. Sub-vi's are used to hide the complexity of the logging routines, which are functionally simple but take up a large amount of space. The goal of these techniques is to create code which is easy to follow and understand while also easy to debug and maintain. Future software developers, even with limited LabVIEW experience, can quickly review the entire code and understand the different modules.

3.3.4 Call C Library Function Node

The fourth feature of LabVIEW that makes it useful for developing modular code is its Call C Library Function Node, which is shown below.

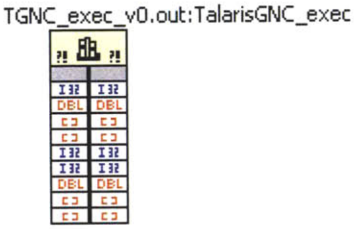


Figure 16: LabVIEW Call C Library Function Node

This block allows pre-compiled C code to be run in LabVIEW as an enclosed module with pre-defined inputs and outputs. On the TALARIS vehicle, this block is used to run the Draper GNC code, which is coded in C, Matlab, and Simulink and auto-coded to C when necessary. A "TGNC_exec_v0.out" file is compiled for the RIO in advance and may be easily switched out to test different algorithms. For example, several versions of the GNC code with different control gains can be pre-compiled and switched during operation to make the best use of testing sessions, which require long setup times and several personnel on staff. Unfortunately, C encapsulation gives no insight into bugs and creates opaque crashes. Often a bug in the C code will either freeze the entire LabVIEW software or in the worst case, crash the RIO, requiring a reboot. However, insight into the GNC code can be provided by a judicious choice of variables to be output as telemetry, and the frequency of opaque crashes was low enough that it was outweighed by the benefits of encapsulation.

Chapter 4

4 Real-Time Software Architecture

This chapter is intended to familiarize the reader with the design of the real-time software written in LabVIEW. The first three sections introduce the RIO hardware architecture, the GNC algorithms, and the GNC timing. The flight software was developed to meet the specifications summarized in these sections. The second three sections will describe the three main components of the flight software: the FPGA, PPC, and GSC code. The LabVIEW code will be discussed in detail, and the focus will be on providing a functional overview as well as a detailed documentation of the software. The information in this chapter will form the technical basis for the discussion of application of modularity to TALARIS in Chapter 5.

4.1 Avionics Hardware Overview

The software is divided between three computing entities: the FPGA, the PowerPC processor, and the GSC. This architecture is enabled by the use of the National Instruments RIO board, which includes both a FPGA and a PowerPC, as well as a number of analog and digital inputs and outputs. By implementing modular software, this architecture can become heavily customizable and quickly reconfigured, and the high number of I/O pins allows for many devices to be implemented at once. These capabilities aligned with the overall design cues of TALARIS: ease of development, flexibility, and reliability. A diagram of the avionics hardware architecture can be seen below.

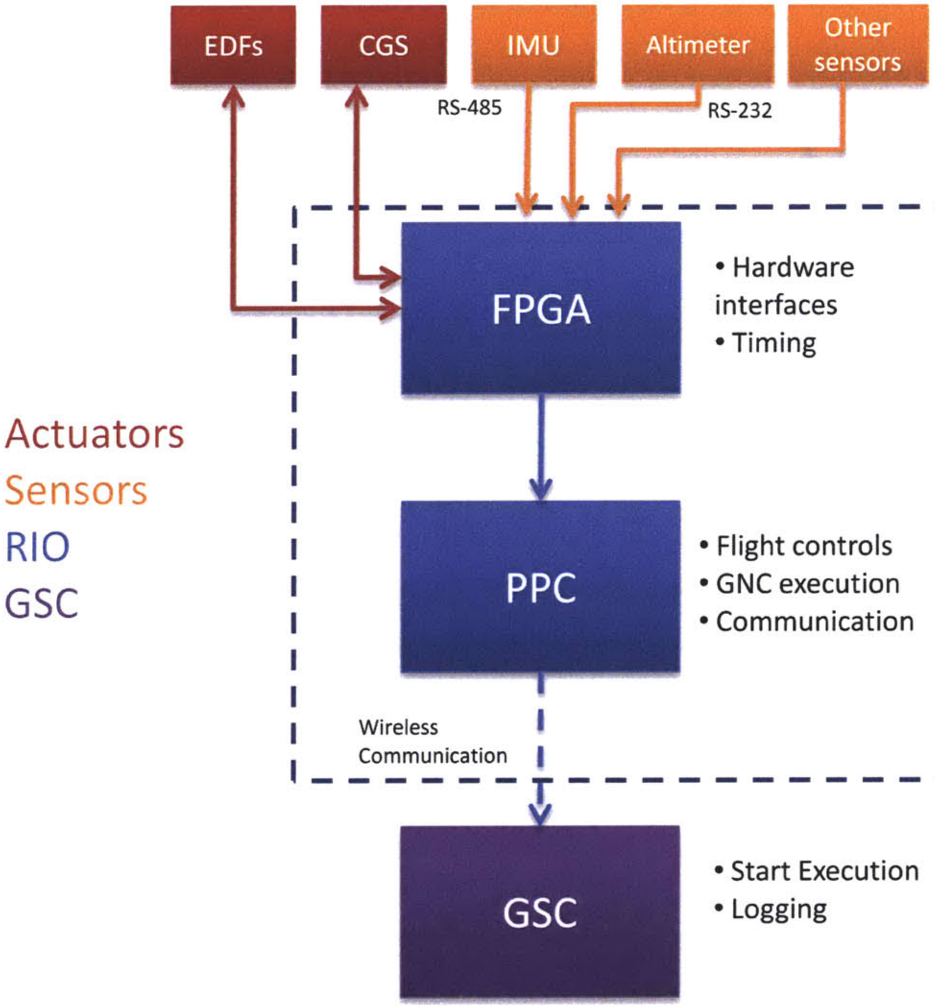


Figure 17: Avionics Hardware Architecture

The FPGA executes at the lowest level of the three computing entities and handles all sensor and actuator interfaces. It is also the most heavily modularized, as each sensor or actuator interface operates more or less independently, and can be separated, reorganized, or omitted depending on the vehicle configuration. All sensor and actuator telemetry is generated here and is timestamped on the FPGA's 40MHz clock. The data is then passed to the PPC by a Direct Memory Access (DMA) FIFO queue. This system ensures that all data is recorded with respect to a single clock, eliminating any ambiguity with the timestamp, and ensures that the data is recorded and read in the correct order through a FIFO queue. This is perhaps the most important function of the FPGA; the FPGA, which is not efficient at performing complicated computations, is able to execute all the interfaces in parallel, which makes it ideal for timing. Having all the interfaces in one place is crucial for this organization to be effective. The FPGA interacts with the real world in real-time, allowing other processes which are not time critical to be executed at a later time by a different entity.

The PPC receives the telemetry generated in the FPGA and reads the data in packets according to the control cycle frequency. The GNC code batch processes them and outputs the actuator commands, which are then sent back down to the FPGA. In this way, the GNC is executed in control "frames," with each frame lining up with the control cycle. This setup allows extremely regular execution of actuator commands, as the FPGA has precise control over actuator timing. The GNC code is only required to issue a command before the start of the next frame to ensure proper execution. The GNC timing as well as the benefits and detriments of this system will be discussed in section 4.3. The PPC code also contains the flight controls on its front panel and passes the data to the GSC for logging.

The sole purpose of the GSC code is to log the telemetry data to text files. Since it is the highest level computing entity, this logging process does not have to be done in real-time. Buffers are used to store the telemetry from one frame, and as long as the data is written before the next frame's data arrives, all data will be correctly written. The GSC communicates with the RIO through a Linksys WGA600N wireless gaming adapter, allowing the GSC and the pilot to sit away from the vehicle behind protective shielding.

This hardware architecture utilizes the advantages of each of the computing entities while minimizing the impact of their weaknesses. The FPGA is excellent for real-time applications. Its structure allows for several processes to run in parallel while tied to the same clock. However, more complicated operations take up more resources in the FPGA. For example, a divide takes a large number of logic gates, which would be executed much more efficiently in the processor [16]. The real-time processor is better at handling more complicated operations, but must rely on interrupts to obtain parallel processing [15]. The combination of the FPGA and the real-time processor (in this case, the PPC) forms a suitable solution for the vehicle's requirements. The FPGA handles timing, hardware interfaces, and simple calculations, while the PPC executes GNC algorithms, packages data for logging, and communicates with the GSC.

4.2 GNC Specifications

Even though the GNC algorithms were developed independently by Draper engineers, there are a few fundamental parameters that are crucial to the development of the real-time software. Based on the vehicle and CGS characteristics, a 5Hz pulse-width modulation (PWM) controller was developed by Michael C. Johnson of Draper Laboratory. The controller issues an "on-time" and a "delay-time" for each valve, and by using these two parameters, the pulsewidth and location can be precisely controlled for each GNC frame.

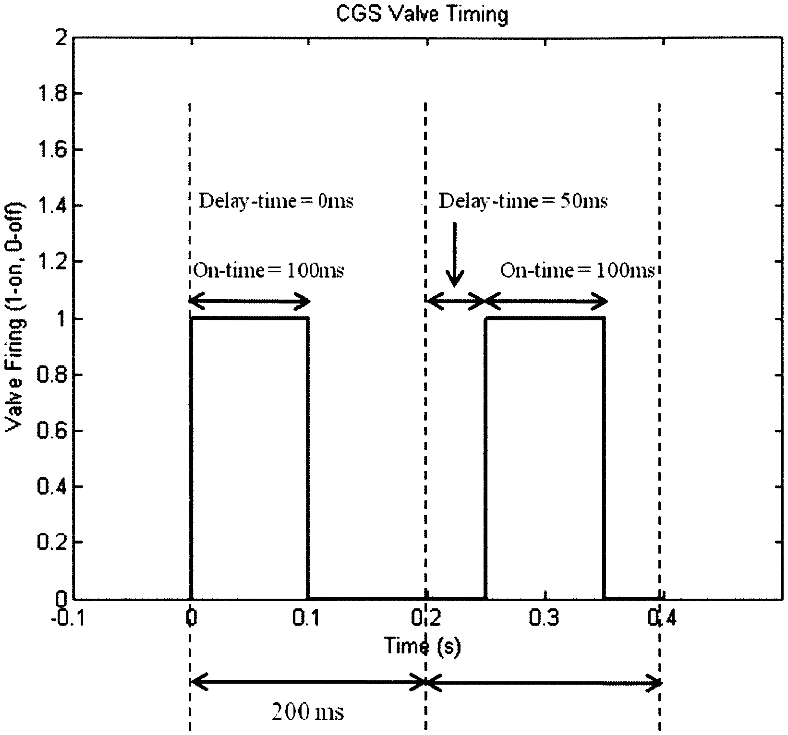


Figure 18: CGS Valve Timing

Pitch and yaw attitude control is obtained by "off-pulsing" the four vertical valves. In one frame, the aggregate on-time for the four valves will provide the total vertical force required for the desired acceleration. By then distributing the on-time (i.e. "off-pulsing") to pairs of valves, an angular acceleration can be achieved on the vehicle. For example to create a positive moment about the +z axis of the vehicle, valves 1 and 3 would fire longer than valves 5 and 7 while keeping their aggregate on-time the same within one frame. Roll control during traverse is achieved with a similar scheme using the two lateral valves on the side of the vehicle opposite from the direction of the traverse. Roll control during vertical rise, hover, and vertical descent is achieved using a phase-plane controller [28]. In this case, if necessary, continuous firing can be obtained by commanding an on-time equal to the control cycle.

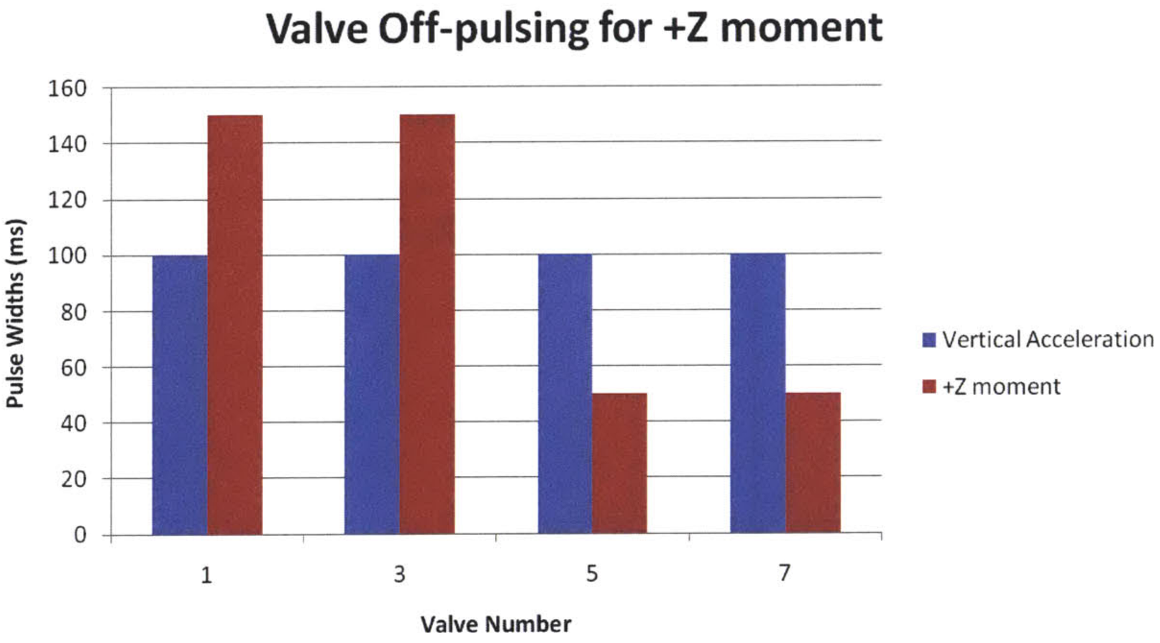


Figure 19: Valve Off-pulsing for +Z Moment

For this PWM controller, 5Hz was found to be an appropriate control frequency. A crucial parameter in this study was the CGS minimum on-off time of 40ms. A 20Hz controller would require GNC frames of 50ms, which would not leave enough time for the valve to close before the next pulse fires. Similarly, a 10Hz controller, or 100ms frames, would require pulses between 40 and 60ms to accommodate the minimum on-off time, which would not leave enough room for attitude control. Thus, a 5Hz controller was chosen and verified in simulation to be stable and suitable for controlling TALARIS.

Given the controller frequency, the frames must be generated in the FPGA. This is done by using the synchronous IMU measurements, which update every 200Hz, or 5ms. Thus, 40 IMU packets are received in exactly 200ms, which defines the GNC frame boundary. In addition, a 1ms square wave is generated by the RIO and sent to the IMU as a sync signal to ensure that the packets are aligned with the FPGA clock. As soon as this sync signal is sensed by the IMU, it begins sending data packets synchronous with the signal's rising edge. The frames could have been generated by a separate FPGA timed loop, but in this way, we synchronize the FPGA's clock with one sensor as well as reduce the amount of code needed. The altimeter returns packets at approximately 10Hz, but its data is treated as asynchronous because the packets are not generated at exactly 100ms intervals. In practice, between one and three altimeter packets are obtained every 200ms frame.

The navigation algorithm uses accelerations and angular rates from the IMU to propagate the state in six degrees of freedom with an Extended Kalman Filter. The altimeter is pointed in the –x direction and provides a periodic update to the altitude state. Thus, the x-axis has a fairly accurate estimate, but the other degrees of freedom are determined by dead reckoning. Their accuracy depends on the IMU drift and bias, so IMU calibration before flight is essential for a stable state solution. The navigation algorithm, designed by Paul J. Huxel of Draper Laboratory, is called at the same frequency as the control code and batch processes one frame's worth of data at a time. The state is then fed to the guidance algorithm, which was designed by Thomas J. Fill of Draper Laboratory. Guidance is called at 1Hz, that is, once every five times the control and navigation algorithms are called. Using the current state and knowledge of the target, the guidance algorithm commands an inertial thrust vector, which becomes the setpoint in the control algorithm. See below for a timeline of the GNC updates.

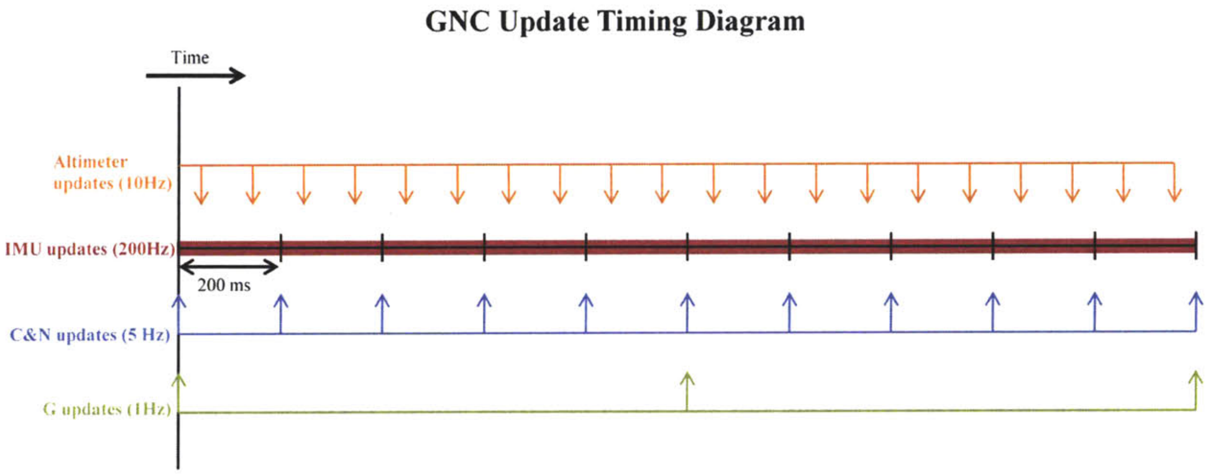


Figure 20: GNC Update Timing Diagram

There are two functions in the FPGA code which are independent of the main GNC execution above. The first is the CGS manual control, which connects the valves directly to switches on the flight controls and allows them to be fired at any time. This mode is useful for non-GNC tests such as debugging valves, CGS characterization, scripted demos, or as an emergency dump of the gas tanks.

The second function that is completely independent of the main GNC execution is the EDF module. In 6-DOF testing, the EDFs will be set to a constant RPM which provides a thrust equal to 5/6th of the vehicles initial weight, to allow the GNC and CGS to operate within a close approximation of the lunar environment, which has 1/6th the gravity of Earth. More complicated controllers are planned for the future which account for the gradual consumption of gas and the vehicle's attitude. The Schultz 40.160 motor controllers take a square wave input every 20ms to determine the throttle of the EDF. The square wave can be between 1ms and 2ms, with 1ms indicating 0% throttle and 2ms indicating 100% throttle. However, throttle percentage does not vary linearly with RPM or thrust, so extensive characterization must be performed to deliver a relatively constant 5/6g weight offset.

4.3 GNC Timing

Most, if not all, GNC algorithms rely on precise timestamps on sensor data to determine the vehicle state. This is not the same as synchronous data, i.e. updates at regular time intervals. Some navigation algorithms can deal with asynchronous sensor updates, as long as the time which the data was taken is recorded and relayed to the navigation algorithm. Similarly, control algorithms require knowledge about the precise timing of actuator firings. It is not necessary that the firing happen immediately after the command is issued, so long as that actuator delay is known and accounted for by the control algorithm. Thus, precise timing information is fundamental to the proper and reliable execution of GNC algorithms.

The GNC cycle begins with gathering of sensor data in the FPGA. Both IMU data and altimeter data are added to DMA FIFOs for passing up to the PPC. The PPC triggers when forty IMU packets have been written to the DMA FIFO, which is one GNC frame's worth of data. The PPC then reads the altimeter data, which can vary from one to three packets per frame. This data is passed to the GNC module, which outputs on-times and delay-times for the valves as well as GNC debug telemetry. The on-times and delay-times are passed back to the FPGA using a LabVIEW FPGA Read/Write Control, which overwrites local variables in FPGA Main.vi. If this write doesn't happen before the next frame boundary, the CGS loop will execute with the previous frame's pulse values.

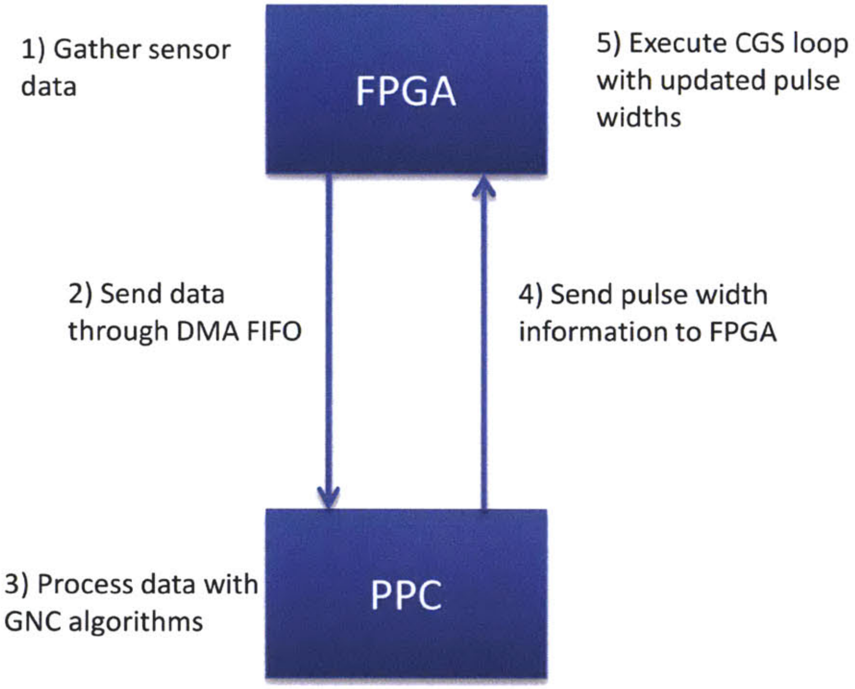

1) Gather sensor data
FPGA
5) Execute CGS loop with updated pulse widths
2) Send data through DMA FIFO
4) Send pulse width information to FPGA
3) Process data with GNC algorithms
PPC

Figure 21: Software GNC Execution Block Diagram

The frame boundary in the FPGA can be thought of as an absolute runtime requirement for the PPC. The DMA FIFO and FPGA Read/Write Control have stochastic communication delays which would normally hinder the ability to implement closed-loop control. The communication delay would add a stochastic lag which would have to be carefully characterized and accounted for in the controller margins. This setup eliminates that jitter and replaces the stochastic delays which we have no control over with a known maximum runtime requirement. It doesn't matter when the GNC issues its next command, as long as it happens before the next frame boundary. The figure below shows a timeline of GNC tasks as they are executed in the FPGA, PPC, and CGS.

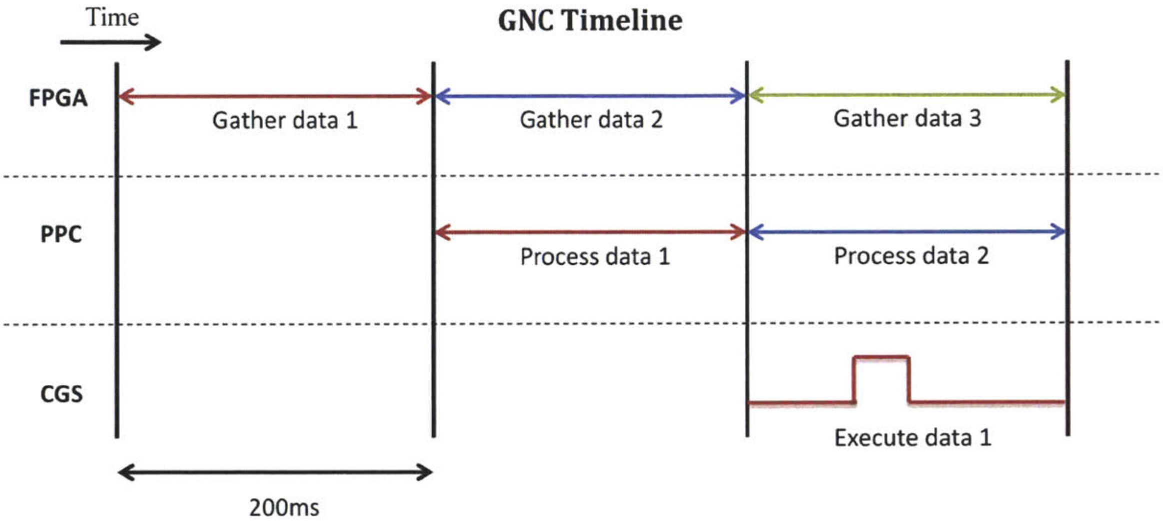


Figure 22: GNC Execution Timeline

In this way, the tasks of gathering data and processing the data are split between the FPGA and the PPC, and parallel processing can be used to utilize the maximum amount of resources. However, in this setup, the actuators are firing off of data that is up to 0.4s old, which adds much unneeded delay. After running benchmark tests, it became apparent that the GNC algorithms ran very quickly, in about 1ms, and that most of the time in the PPC was being wasted waiting for the command to be executed. To mitigate this delay, Christopher J. Wardman of Draper Laboratory suggested that the CGS loop be offset from the main GNC loop by a set number of milliseconds. By tailoring this delay to the GNC algorithm run time, the lag between the end of GNC calculation and the valve firing can be minimized, and the valves can fire much sooner after the command is issued. The updated timing scheme is shown below.

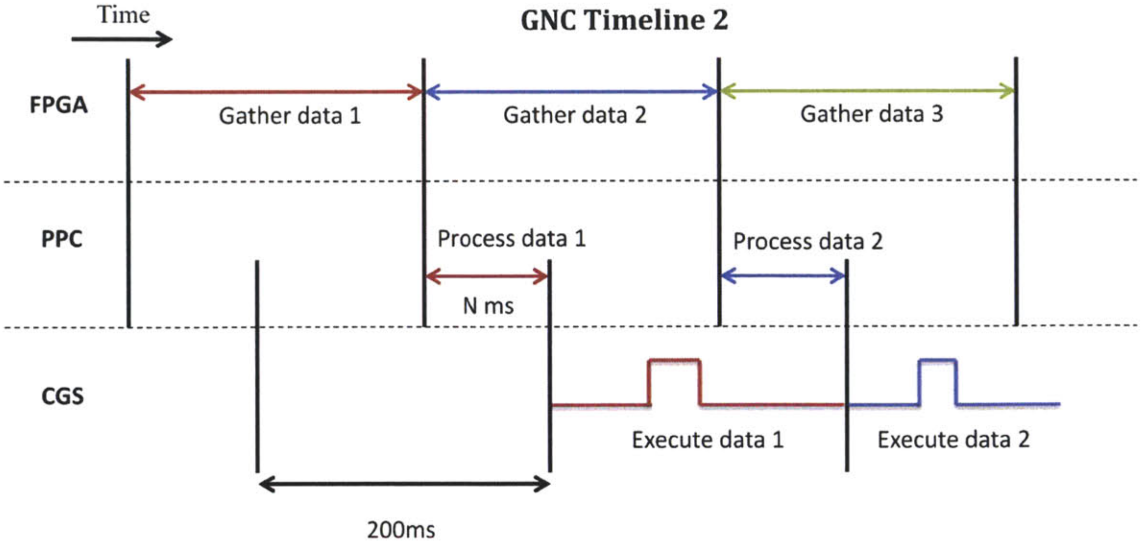


Figure 23: GNC Execution Timeline 2

This offset time must be carefully set to ensure that the GNC code still has time to complete and send the command to the FPGA. Although the actual algorithm runs on the order of 1ms, the DMA FIFO read and FPGA Read/Write Control took on the average of 22ms to complete. Benchmark tests were run to determine the average and 3σ times for three points: after the IMU read, after the altimeter read, and after the command was issued. These points happen in series in the PPC, with the first two being DMA FIFO reads and the command issue being an FPGA Read/Write Control.

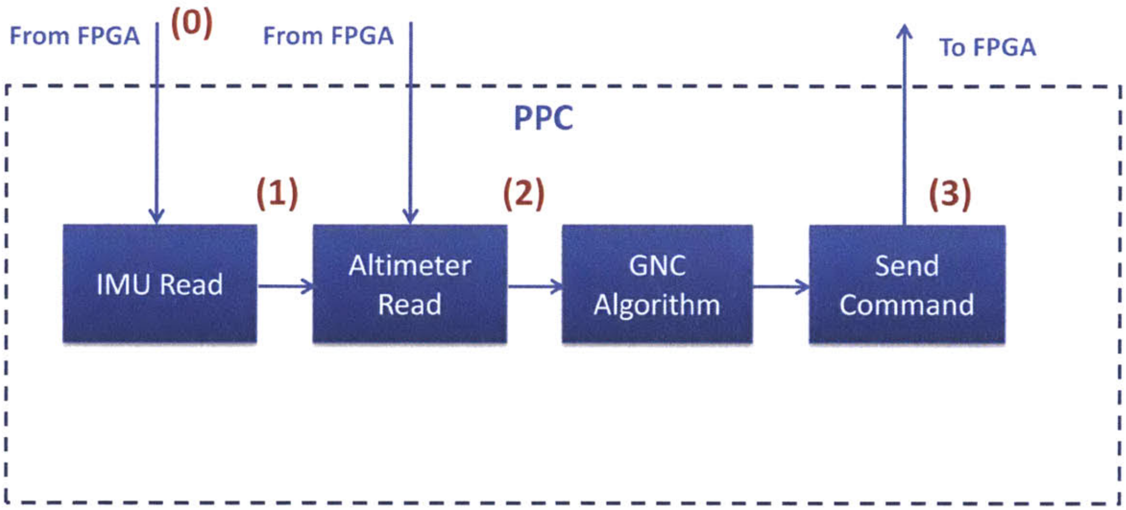


Figure 24: GNC Execution in PPC

The timing benchmark points are shown above in red, with (0) being the baseline time that the other three points are compared against. Point (2) refers to the point that the DMA FIFO read completes and information is sent to the GNC algorithm. Point (3) represents the total time it takes for the PPC to finish calculation and send the command to the FPGA. The benchmark was run for 10 minutes, or 3000 frames, and the average time and standard deviation for each point is summarized in the table below.

	(1) IMU Read	(2) Altimeter Read	(3) Command issue (total)
Average	6.21ms	14.30ms	22.46ms
Standard deviation	2.92ms	5.52ms	5.97ms

Table 1: GNC Timing Benchmark Results

The 3σ point for total execution was 40.37ms. Out of 3034 frames, 12 violated 40ms, but none violated 50ms. Thus, 50ms was taken as the CGS frame offset to ensure that no commands are missed.

4.4 FPGA Software

The FPGA software is almost solely composed of sensor and actuator interfaces in a single vi called "FPGA Main.vi." By using a single vi, local variables can be utilized to pass information between modules when needed, and FPGA Read/Write Controls can be used by the PPC code to directly access these variables. For time-sensitive telemetry, e.g. the IMU and altimeter data, DMA FIFOs are used to communicate the information to the PPC. For other sensors that are polled less frequently, e.g. voltage and pressure sensors, a FPGA Read/Write Control can be used by the PPC to pull this information.

A few simple design principles were followed when coding the FPGA. First, the code was kept as simple as possible, to reduce both resource utilization and compile times. Divide operators were avoided when possible, because a divide operation is much more efficiently and accurately performed in the PPC [16]. The reason for this is because divide often results in a rational number which an FPGA cannot accurately express with fixed-point variables. The result is an approximation of the real answer, and the implementation of larger fixed-point numbers needed to improve this approximation increases the number of logic gates, power consumption, and delay time required [16]. For these reasons, complicated calculations involving divides were kept to the PPC to minimize FPGA resources used and compile times. Next, each of the modules was developed independently and integrated at a later date. Besides being able to split the labor between multiple programmers, this also made each interface fairly independent, encouraging flexibility and different reconfigurations. Even though simplicity was stressed in the development of the FPGA code, the compile times could still reach over an hour for the full software, which was another reason to separate development into smaller pieces of code.

The FPGA code is fairly difficult to debug for a number of reasons. First, the front panel is not visible during execution, which prevents the programmer from monitoring variables during runtime. Second, bugs in the FPGA code usually cause catastrophic failures, often crashing the program, LabVIEW, and/or the RIO without any warnings or error messages. Third, any changes made to the FPGA code will require a recompile, which takes over an hour with the full testing software. With such long compile times, there is always a tradeoff between the number of changes and the number of recompiles. The tendency is to avoid long compile times by making several changes at once, but this risks running into an opaque bug which will require several recompiles to determine and fix. Finally, one of the most common errors when coding the FPGA is implementing timed loops correctly. If the loop is too complicated and cannot be completed in time, the loop will just skip the next iteration and attempt to execute as quickly as possible. This runtime violation is not caught by the compiler, so it is the responsibility of the programmer to ensure and verify that the loop is running at the rate it was intended.

The FPGA code currently consists of six modules: the main timer, altimeter transmit, altimeter receive, IMU receive, sensor acquisition, and the CGS loop. This version of the software was intended for GNC tests using the CGS only and does not include the EDF module. A separate version of the code, called the Force Balance code, includes two additional modules which interface with the EDFs and the load cell. These modules will be described below but they do not appear in the overall screenshot for FPGA Main.vi.

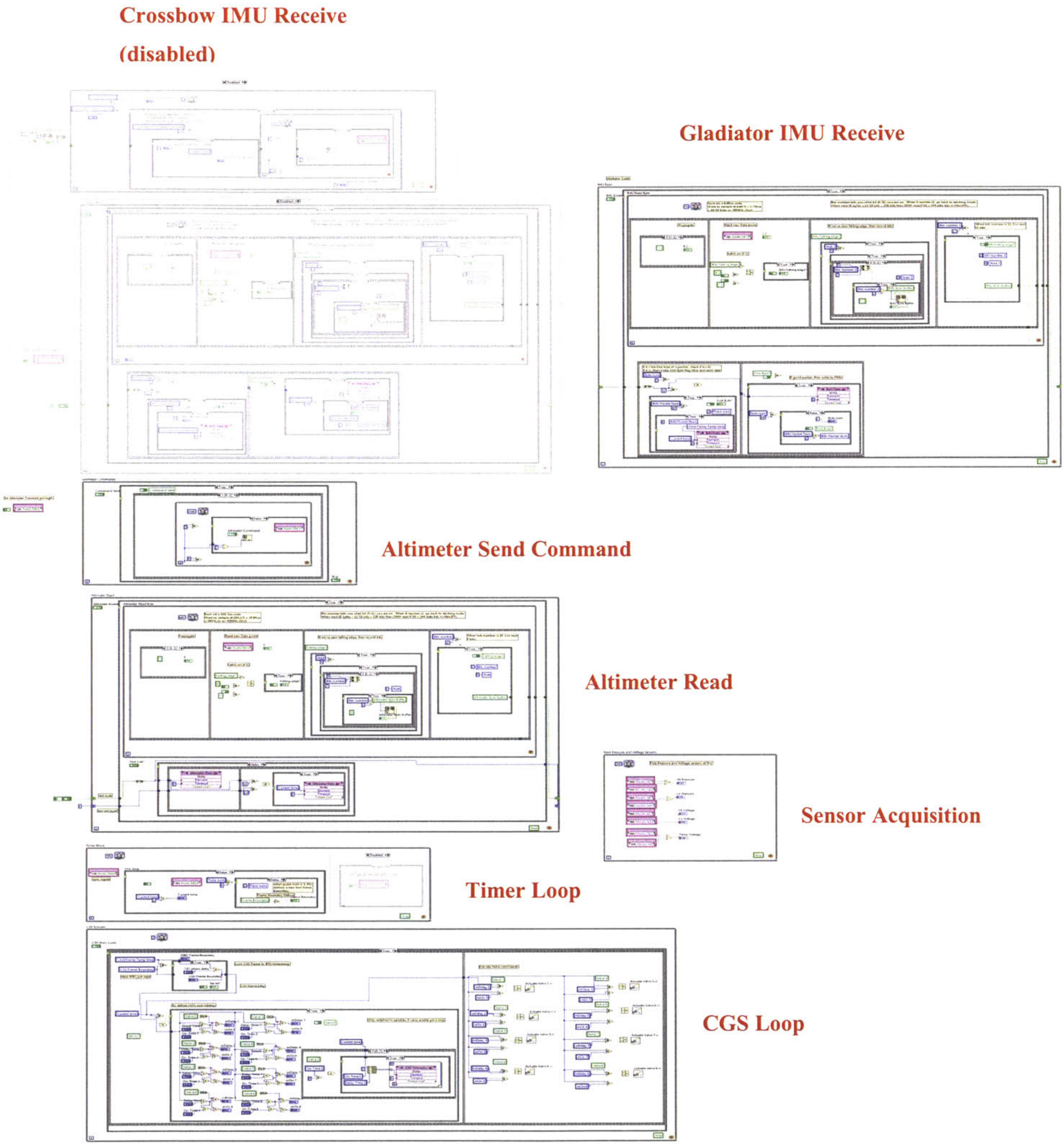


Figure 25: LabVIEW Block Diagram for FPGA Main.vi

4.4.1 Timer Loop

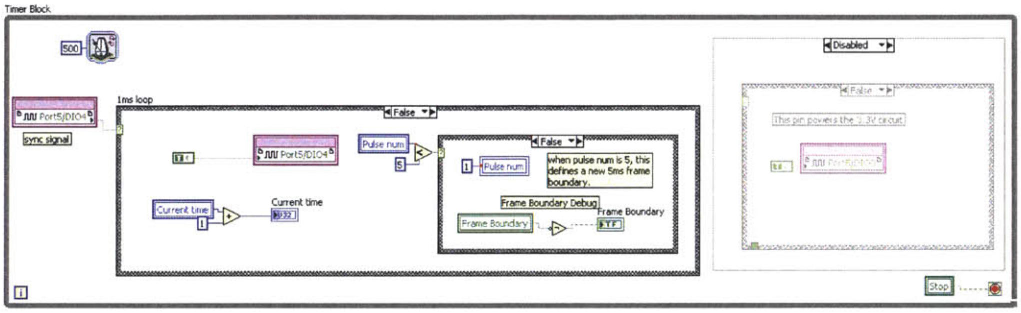


Figure 26: LabVIEW Block Diagram for FPGA Timer Loop

The main purpose of the timer loop is to create a timer variable based on the FPGA clock and generate a 1ms square wave to send to the IMU as a sync signal. The loop toggles the sync signal every 500μs and increments the variable “Current Time” on every rising edge. The “Current Time” variable will be used to timestamp all sensor data.

4.4.2 Altimeter Transmit

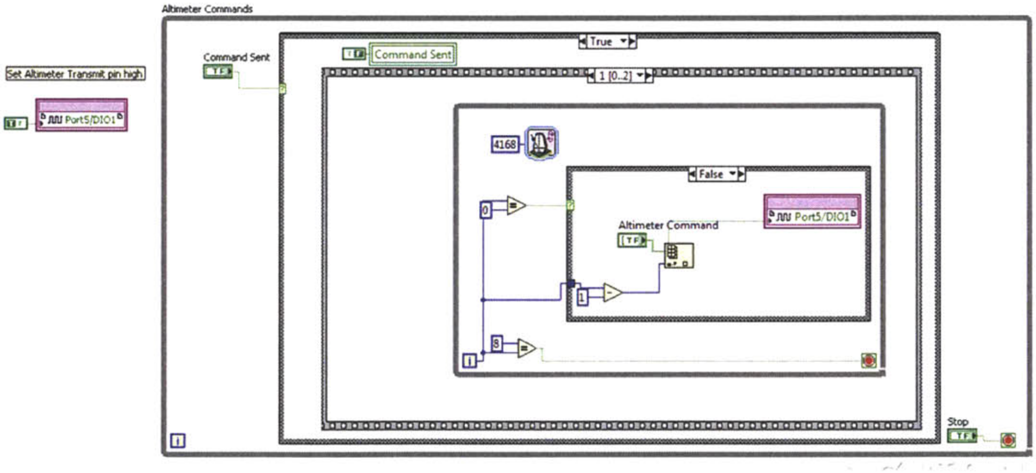


Figure 27: Altimeter Transmit

The altimeter uses the RS-232 8N1 communication protocol to send and receive data. To send commands to the altimeter, the RIO uses a dedicated DIO 3.3V pin, which it sends high to indicate standby. This occurs outside the timed loop, which happens only once and at the beginning of execution. When the "Command Sent" boolean is set to true by the PPC, the FPGA generates the bits according to the command from the boolean array "Altimeter Command." The bit width must match the baud rate of the altimeter, which is set to 9600 by default. The baud rate is related to the bit width by the equation below.

$$T_s = {}^{1}/_{f_s} \qquad T_s = Bit\ duration$$

$$f_s = Baud\ rate$$

In addition, the LabVIEW loop timer must be wired with a value in "ticks," which are number of 40MHz clock cycles. Using this information, the bit width in ticks can be calculated.

$${}^{1}/_{9600} = 104.17\mu s/bit$$

$$1\text{ tick} = {}^{1}/_{40MHz} = 25ns$$

$${}^{104.17\mu s/bit}/_{25ns/tick} = 4168\ ticks/bit$$

The 8N1 refers to 8 data bits, no parity bit, and one stop bit. The first bit, or the start bit, is always 0 to indicate a command is about to be sent. This is followed by 8 data bits and a stop bit, which returns the transmit pin to high. Commands are converted to ASCII numbers and sent using the 8N1 protocol. For example, the character "D" = 68 in ASCII, which equals 0010 0010 in binary (LSB first). Thus the full command would be ten bits including the start and stop bit: 0001000101.

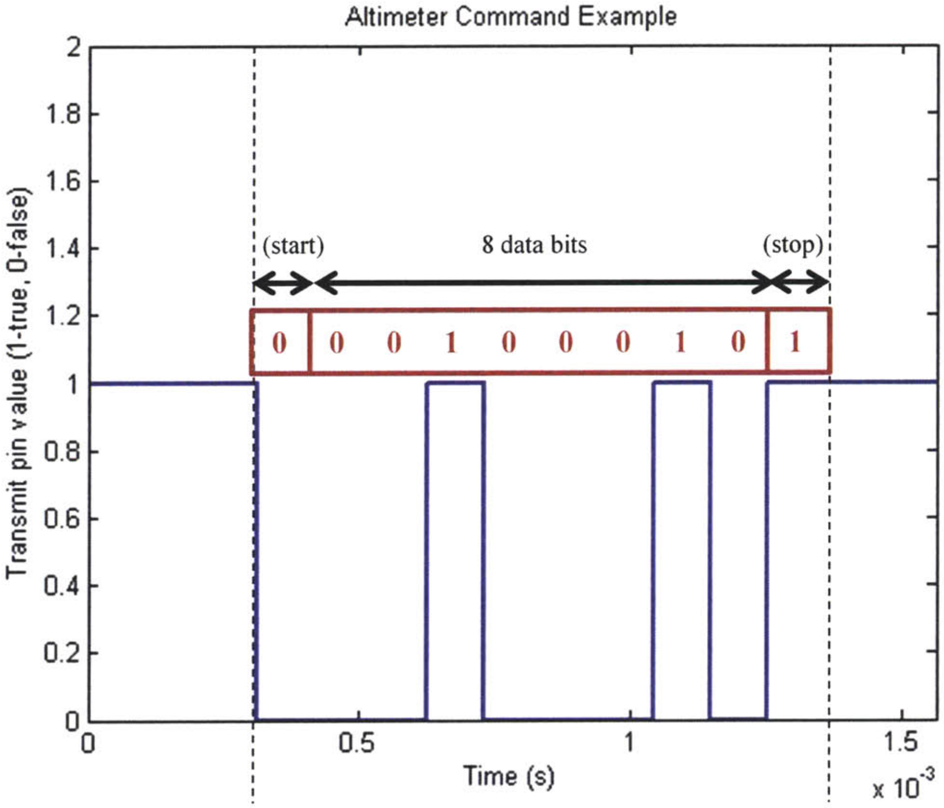


Figure 28: Altimeter Command Example

4.4.3 Altimeter Receive

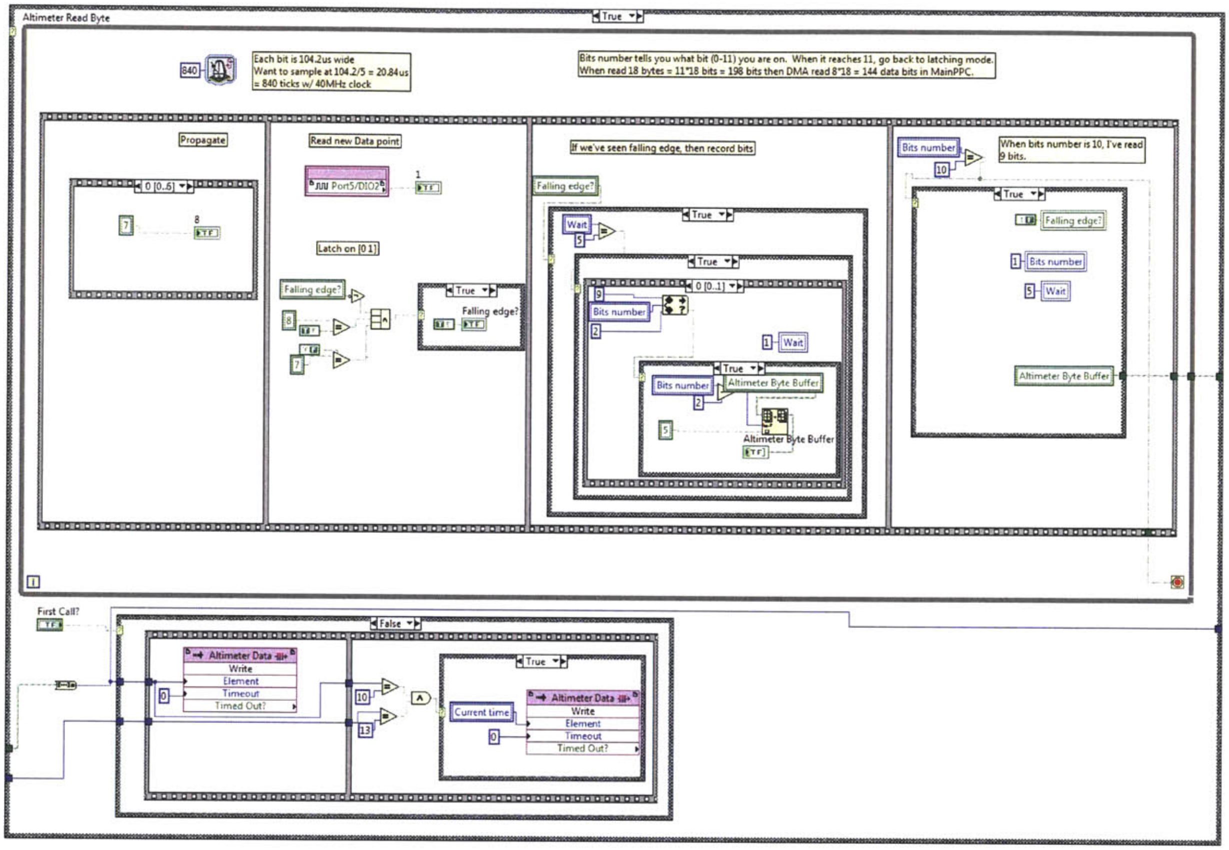


Figure 29: Altimeter Receive

The top half of the altimeter read code is composed of a timed loop and is concerned with detecting a falling edge, which indicates the start of a data packet, and populating the byte buffer. To capture the falling edge, the DIO pin must be sampled at a much faster rate than the baud rate, so the timed loop runs approximately five times faster, at 840μs. An eight boolean-wide buffer is numbered 1 through 8, and every iteration, indexes 1 through 7 are shifted to indexes 2 through 8, and a new sample is written to index 1. A falling edge is detected when the pattern "01" goes through indexes 7 and 8.

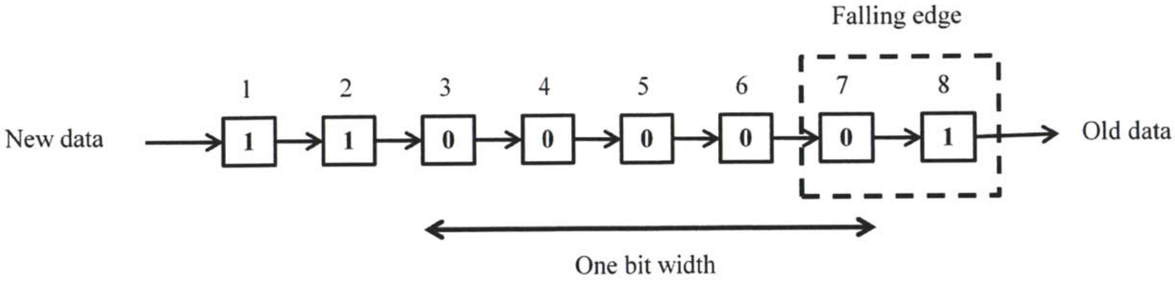
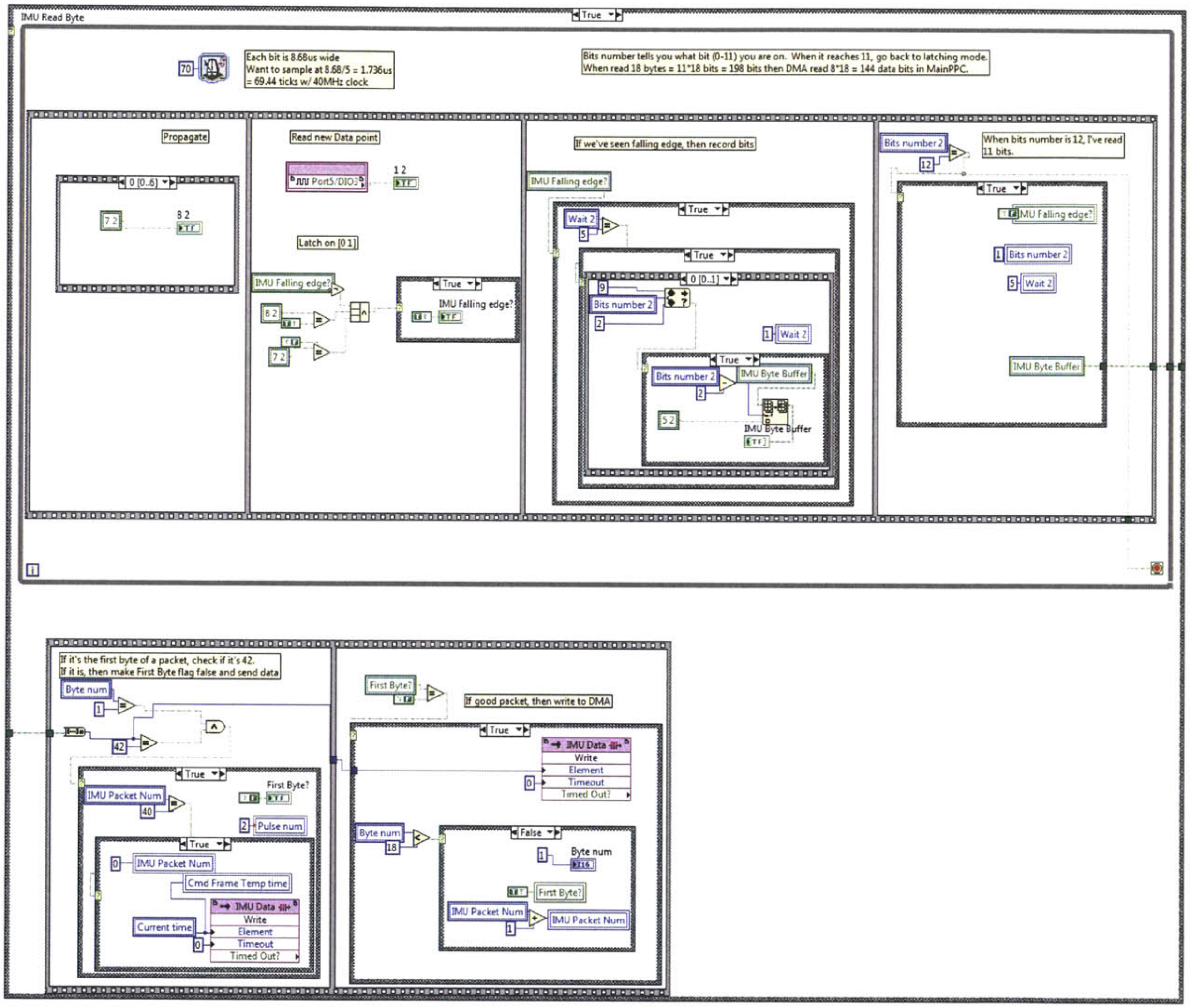


Figure 30: RS-232 Falling Edge Detection

When a falling edge is detected, the value at index 5 is read and written to the byte buffer. Index 5 is used because it represents the middle of the next bit. Five iterations are passed before index 5 is read again and added to the byte buffer. This is repeated for ten bits, which represent the start bit, the eight data bits, and the stop bit. When the byte buffer is full, it is passed to the bottom half of the code, which writes the byte to the DMA FIFO. All flags are reset, and the loop is ready to detect another falling edge.

The altimeter typically sends nine bytes at a time, each representing an ASCII character. For example, to send the altitude "0.934," the altimeter would send the characters "48 48 48 46 57 51 52 13 10" which translated to "0 0 0 . 9 3 4 <CR> <LF>." The altimeter will send a carriage return and a line feed to indicate the end of a measurement. The FPGA will insert a timestamp before every nine characters, so a typical altimeter measurement will be ten bytes long: one timestamp + nine data bytes. In one 200ms frame, we expect 1-3 altimeter packets, depending on how the sensor timing is aligned with the FPGA clock.

4.4.4 IMU Receive



Each bit is 8.68us wide
Want to sample at 8.68/5 = 1.736us
= 69.44 ticks w/ 40MHz clock

Bits number tells you what bit (0-11) you are on. When it reaches 11, go back to latching mode.
When read 18 bytes = 11*18 bits = 198 bits then DMA read 8*18 = 144 data bits in MainPPC.

Propagate

Read new Data point

Latch on [0 1]

If we've seen falling edge, then record bits

When bits number is 12, I've read 11 bits.

If it's the first byte of a packet, check if it's 42.
If it is, then make First Byte flag false and send data

If good packet, then write to DMA

Figure 31: IMU Receive

The IMU interface is similar to the altimeter interface in concept with a few key differences. There is no need to send commands to the IMU except for the 1ms sync signal, so we have only a receive module. The IMU uses the RS-485 8E2 communication protocol, with 8 data bits, an even parity bit, and two stop bits. The default baud rate is 115200, giving us a bit width of 8.68μs. To oversample at five times the bit width, the timed loop must be performed at 70 ticks per iteration. The edge detection is performed in exactly the same manner as the altimeter receive, with the byte being compiled and sent to the bottom half of the module to be added to the DMA FIFO. One full IMU data packet is composed of 18 bytes with values seen in the table below.

Index Number	Data Value
1	Packet start (42)
2	Message counter
3-4	Gyro x
5-6	Gyro y
7-8	Gyro z
9-10	Accel x
11-12	Accel y
13-14	Accel z
15-16	Temperature
17	Test/status indicator
18	Checksum

Table 2: Gladiator IMU Data Packet [26]

To ensure that the correct 18 bytes are added to the DMA FIFO, the FPGA will ensure that the value of the first byte read is 42. This trips the boolean flag "First Byte?" and the FPGA will write the time stamp, then add the next 18 bytes to the DMA FIFO.

4.4.5 Sensor Acquisition

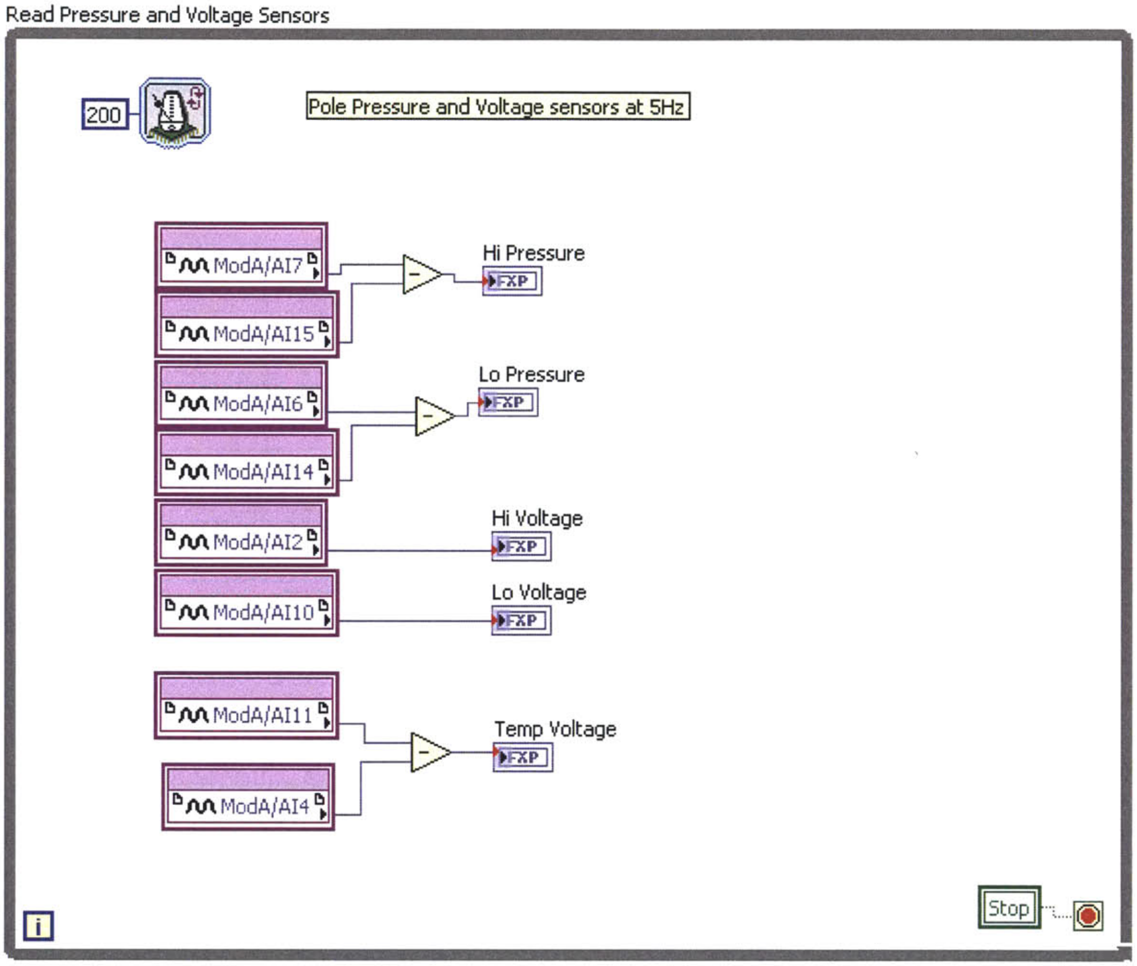


Figure 32: Sensor Acquisition

The pressure, voltage, and temperature sensors are connected to the analog input (AI) pins of the RIO and are read at a rate of 5Hz. These sensors are mainly used for system health and are output to the pilot interface in the PPC for monitoring. Since these measurements are not as time critical to the GNC execution, they are read by FPGA Read/Write Controls in the PPC code.

4.4.6 CGS Loop

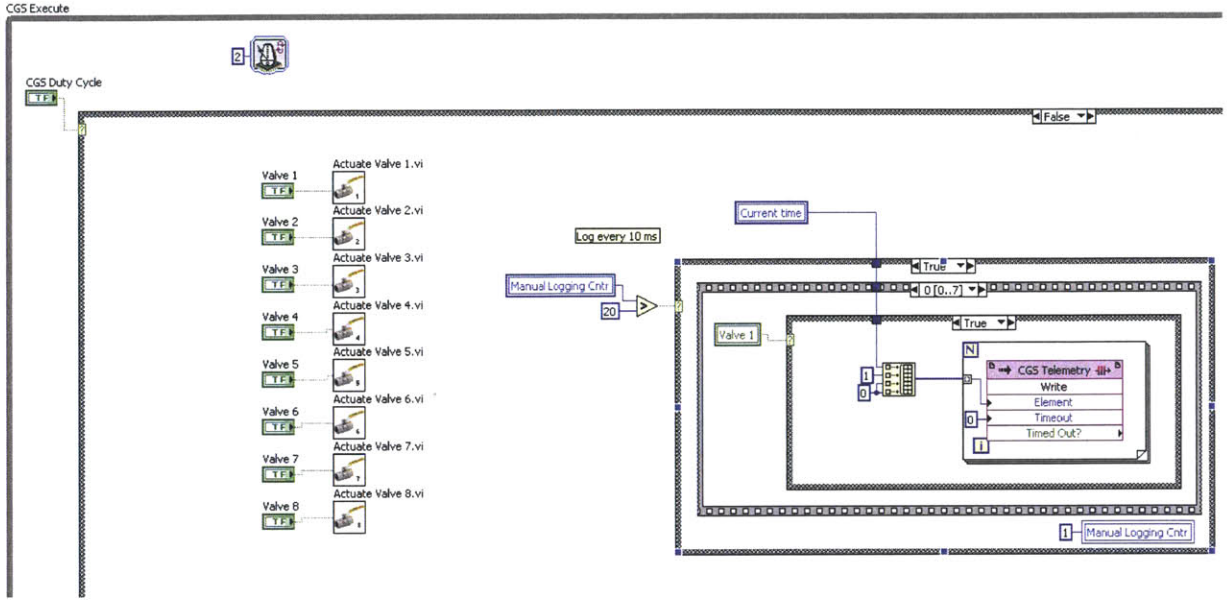


Figure 33: CGS Manual Fire Loop

The CGS loop operates in two modes depending on the value of the "CGS Duty Cycle" boolean. When set to false, the loop operates in manual fire mode, and eight boolean variables are used to control the on/off status of the valves. These booleans are then controlled by the PPC using FPGA Read/Write Controls. Every 2ms, the value of these booleans is sent to their respective sub-vi's named "Actuate Valve #.vi" which activates the correct digital out pins to actuate the correct valve. The stacked sequence on the right logs the values of each of the booleans every 20ms, adding the timestamp, valve number, and zero for the on-time and delay-time (to indicate a manual fire). This mode is most commonly used for CGS characterization tests, demos, or dumping the low pressure gas from the tanks.

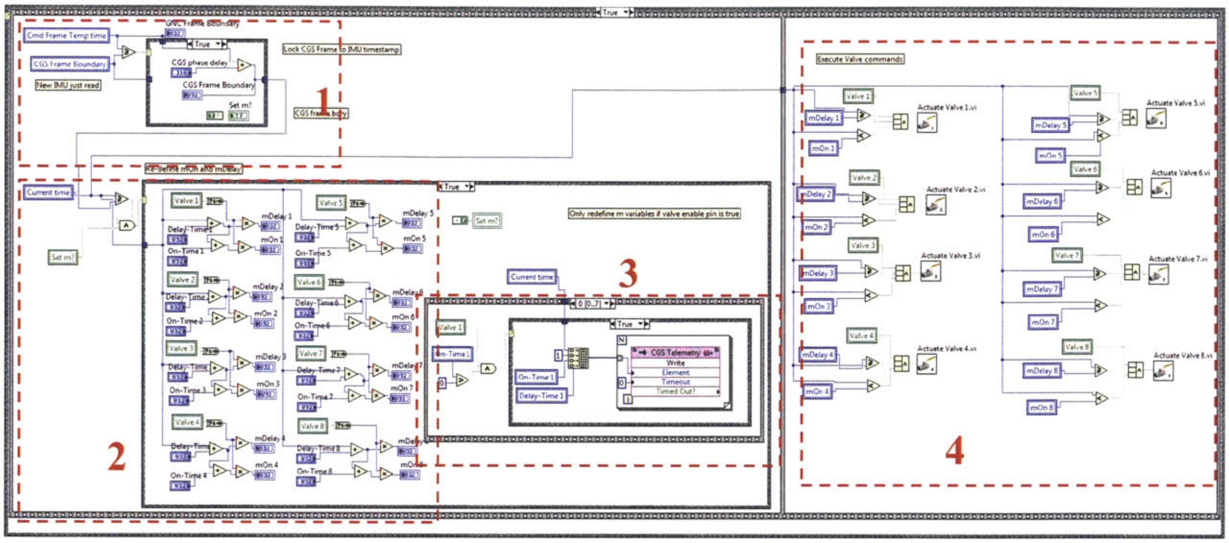


Figure 34: CGS Duty Cycle Fire Loop

The second mode of operation is the duty cycle fire mode, which is crucial to the GNC execution. The code can be divided into four parts, indicated by the numbers above, with one, two, and four executed in series and two and three executed in parallel.

1) The first section of the code determines whether a new 200ms frame has started. If so, then "CGS Frame Boundary" will be updated to be offset from the start of the GNC frame. This is how the frame offsetting discussed in section 3.3 is achieved.

2) The second section will trigger only if the current time is greater than "CGS Frame Boundary." This condition indicates the start of the CGS frame. Only once at the beginning of the CGS Frame, the "m" variables will be updated. The m variables, mDelay and mOn for each valve, indicate the times between which the valve should be open. Defined from the beginning of the CGS frame, the m variables are defined as:

$$mDelay\ 1 = CGS\ Frame\ Boundary + Delay\text{-}Time\ 1$$

$$mOn\ 1 = mDelay\ 1 + On\text{-}Time\ 1$$

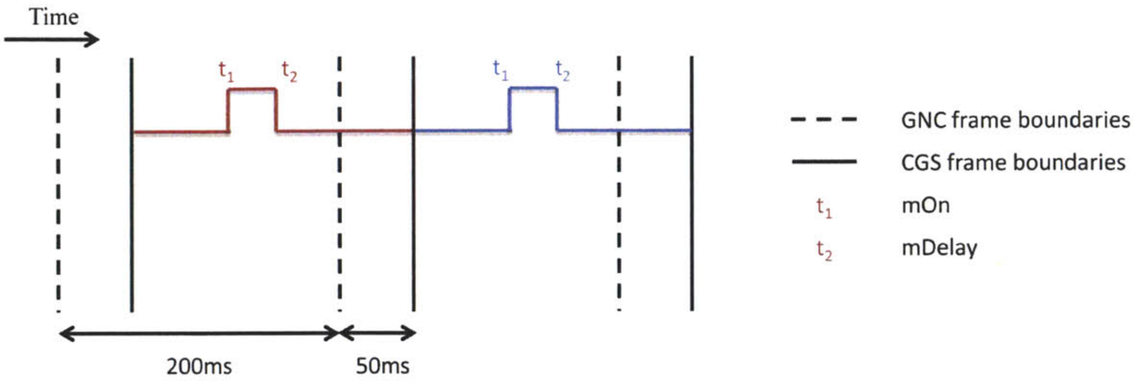


Figure 35: CGS frame offset and m variables

3) At the same time that the m variables are updated, the CGS telemetry is logged in section 3 of the code. This logging routine is similar to the one used in the manual fire mode, reporting four values for each valve: timestamp, valve number, on-time, and delay-time.

4) The fourth section of the code checks two conditions to determine whether the valve will be actuated. In this mode, the valve boolean acts as an enable flag. If the flag is false, set by the PPC, then the valve will never fire. If the flag is true and the current time is between mDelay and mOn, then a value of true will be sent to "Actuate Valve #.vi" and the valve will open.

4.4.7 Load Cell Receive

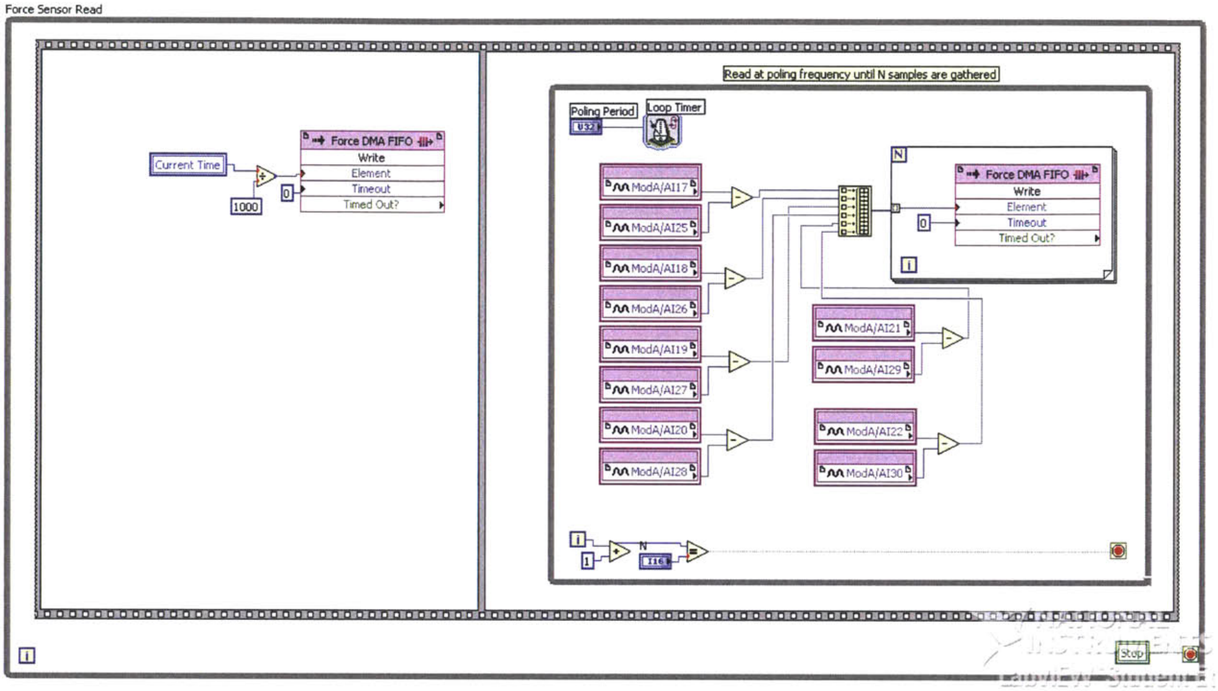


Figure 36: Load Cell Receive

Like the system health sensors, the load cell interfaces to the RIO using AI pins. Twelve AI pins are set up to take differential readings which are added to the DMA FIFO at a rate of 500Hz. A timestamp is placed at the beginning of every 200ms frame, or 100 samples.

4.4.8 EDF Loop

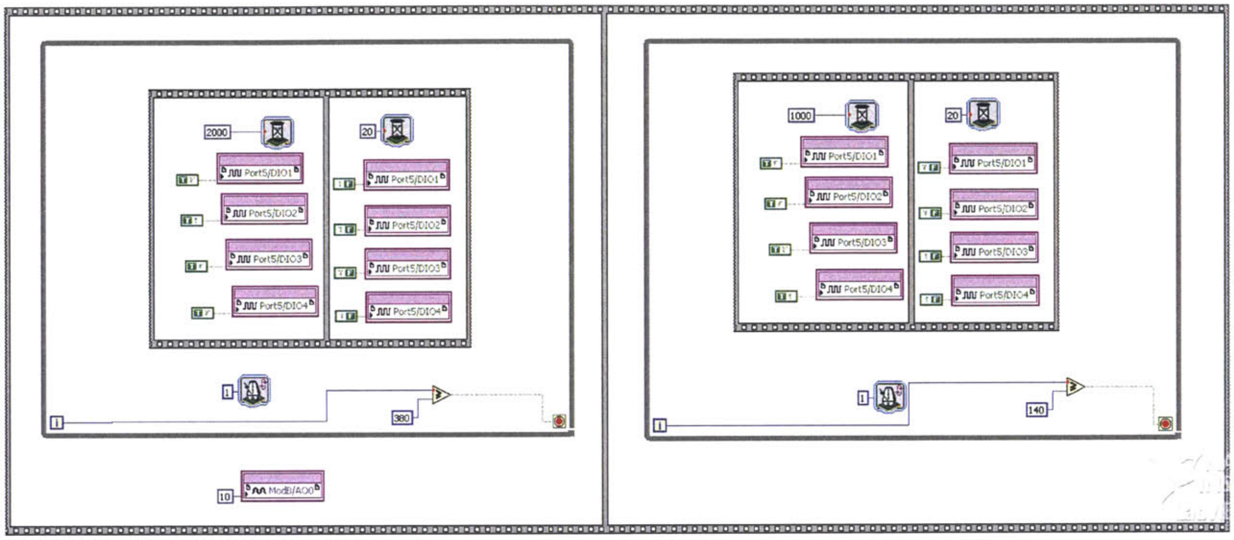
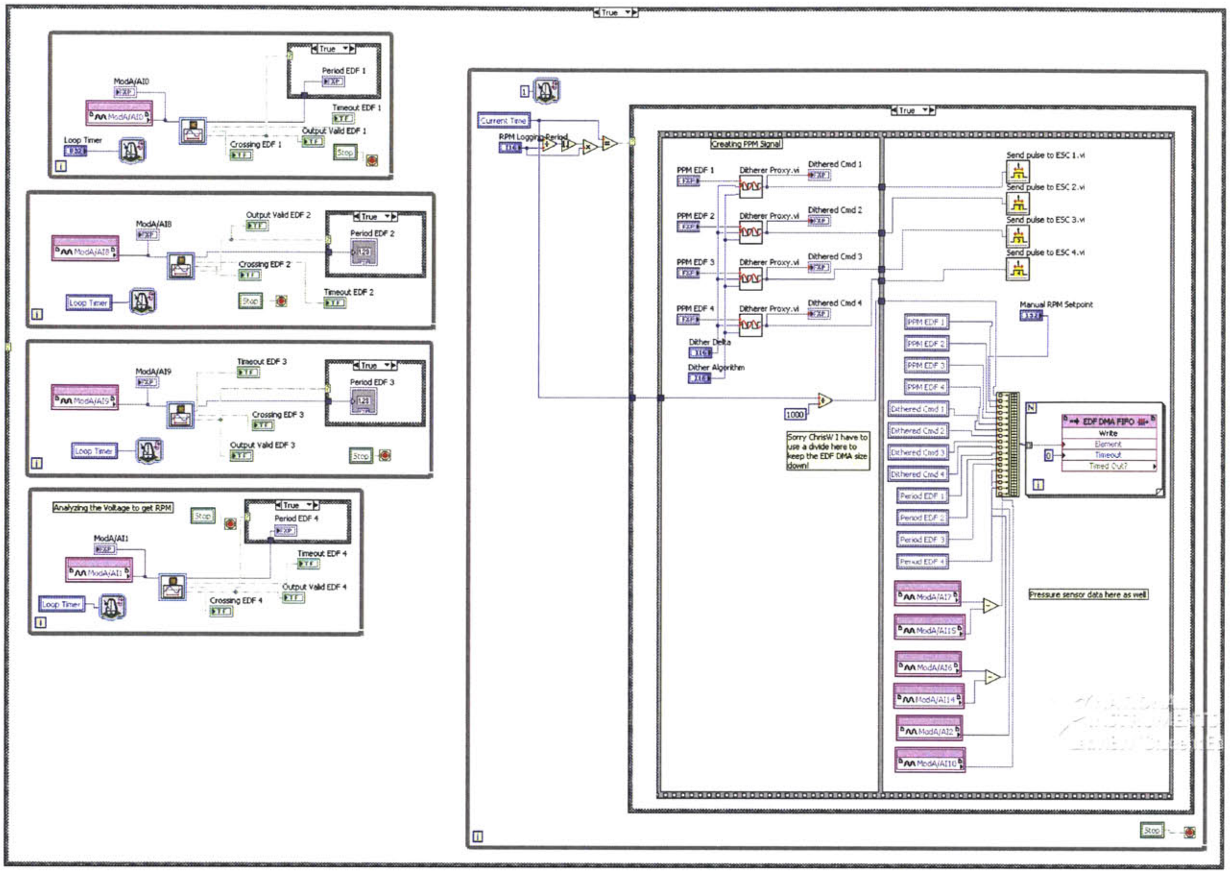


Figure 37: EDF Initialization

The Schultz motor controllers require an initialization procedure at power-up to calibrate the signals for no throttle and full throttle. This is done at the start of FPGA execution, before any of the other modules begin. First, a 2ms square wave is sent to the motor controllers to indicate full throttle. After 380ms, the signal is changed to a 1ms square wave to indicate no throttle. During this time, the motor controller will go through a series of beeps to indicate that calibration has completed successfully. Caution is recommended when using this initialization routine outside of normal calibration procedures. If the motor controller is not in calibration mode, then a full throttle command will be sent to the EDFs, which could be potentially hazardous to personnel.



Analyzing the Voltage to get RPM

Sorry ChrisW I have to use a divide here to keep the EDF DMA size down!

Pressure sensor data here as well

Figure 38: EDF Module

The EDF module can be separated into four EDF read timed loops, located on the left side of the module, and one EDF send timed loop, which is the large loop on the right side. The read loops monitor the RPM signal on an AI pin for a set amount of time. A LabVIEW "Analog Period Measurement" block is used to approximate the period of the oscillating signal. In the steady state, this period should be aligned with the RPMs of the EDF. The period is normalized by the length of the loop so that the units of the period is [iterations/revolution]. For example, a period value of 12 would mean that one EDF revolution equals 12 loop iterations. To convert this to RPMs, use the following relation:

$$\frac{120000\left[\frac{Iterations}{min}\right]}{Period\left[\frac{Iterations}{Revolution}\right]} = RPMs$$

This relation is for a loop frequency of 500μs. If the loop timer is changed, then the scalar in the numerator will have to change as well. That scalar is calculated by:

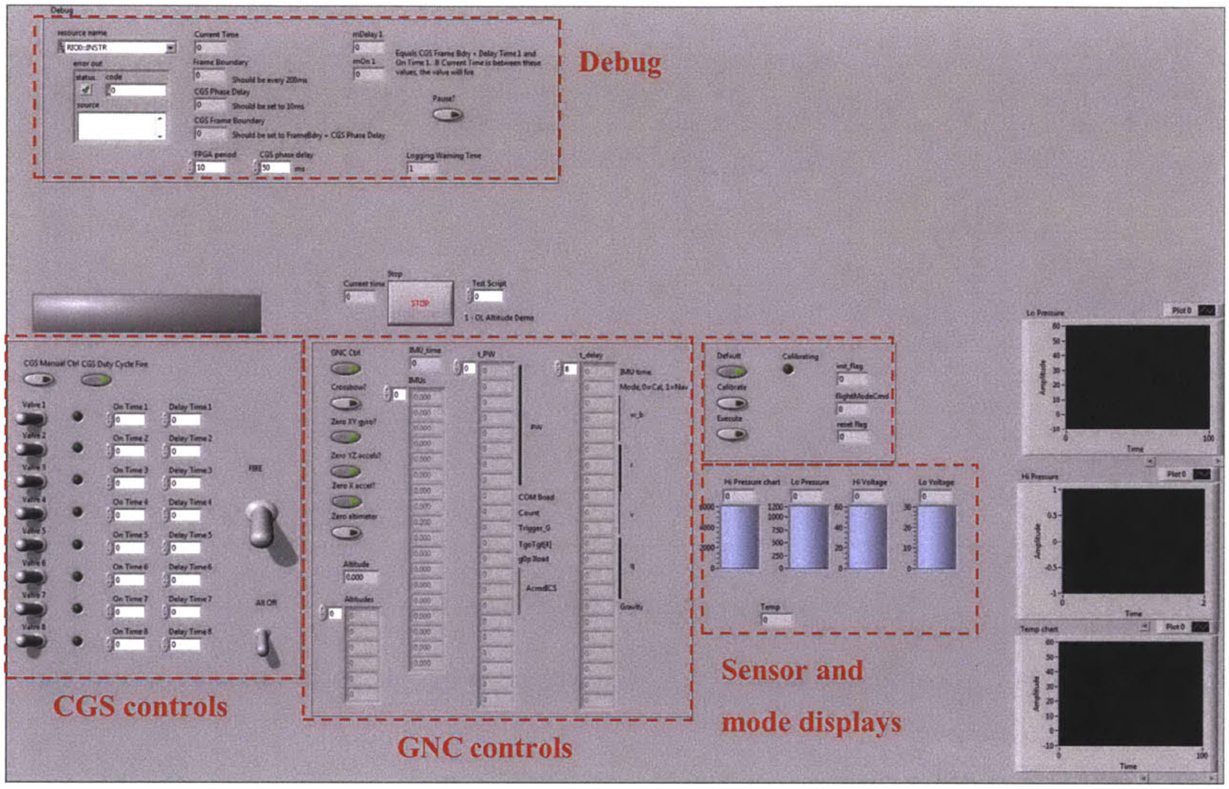
$$\frac{1}{Loop\ Timer\left[\frac{\mu s}{iteration}\right]} \times \frac{6*10^{7}\mu s}{1\ min} = \left[\frac{Iterations}{min}\right]$$

The command portion of the module is executed every 20ms. Commands are set by the PPC through FPGA Read/Write Controls and range from 0 to 1000 μs. This is added to the base 1ms pulse and sent to the motor controller in the sub-vi "Send pulse to ESC #.vi." At the same time, EDF telemetry and some analog sensor data are written to the DMA FIFO, to be later read by the PPC and logged.

4.5 PPC Software

The PPC software is primarily composed of three vi's. The first, "Main PPC.vi," acts as the main executable. Its front panel consists of the main flight controls and is the primary interface between the pilot and the vehicle. "GNC Exec.vi" is called from within "Main PPC.vi" and contains the DMA FIFO Read blocks as well as the autocoded Draper GNC code. After the GNC code runs, the CGS commands are sent to the FPGA using an FPGA Read/Write Control. "FPGA Interface.vi" is also called from within "Main PPC.vi," and sends the remaining commands with a single FPGA Read/Write Control. System health sensors are also polled in this vi. Global variables are used to communicate information between these three vi's, and their scope is only within the PowerPC. Shared variables are used to send telemetry to the GSC for logging, and can be accessed in both the PPC and the GSC.

4.5.1 Main PPC.vi



resource name: RIO0::INSTR

error out — status ✓ code 0, source

Current Time: 0

Frame Boundary: 0 Should be every 200ms

CGS Phase Delay: 0 Should be set to 10ms

CGS Frame Boundary: 0 Should be set to FrameBdry + CGS Phase Delay

FPGA period: 10

CGS phase delay: 50 ms

mDelay1: 0

mOn1: 0

Equals CGS Frame Bdry + Delay Time1 and On Time 1. If Current Time is between these values, the valve will fire

Logging Warning Time: 1

Current time: 0

Test Script: 0 — 1 - OL Altitude Demo

On Time 1: 0	Delay Time 1: 0
On Time 2: 0	Delay Time 2: 0
On Time 3: 0	Delay Time 3: 0
On Time 4: 0	Delay Time 4: 0
On Time 5: 0	Delay Time 5: 0
On Time 6: 0	Delay Time 6: 0
On Time 7: 0	Delay Time 7: 0
On Time 8: 0	Delay Time 8: 0

IMU_time: 0

Altitude: 0.000

init_flag: 0

flightModeCmd: 0

reset flag: 0

Hi Pressure chart: 0

Lo Pressure: 0

Hi Voltage: 0

Lo Voltage: 0

Temp: 0

Figure 39: Main PPC.vi Front Panel

The front panel of "Main PPC.vi" consists of the main controls used by the pilot. The controls are divided by function, with CGS controls on the left, GNC controls and displays in the center, and system health sensors displayed on the right. Above the controls is the all-stop button, which may be pressed at any time to shut down the vehicle, and a large warning light which will flash when telemetry logging is not taking place, to remind the pilot. These controls provide a variety of operating modes to the pilot. The CGS may be operated in manual mode, duty cycle mode, or by automated scripts. The GNC has three automated steps which must be pushed in sequence to properly activate the GNC: standby, calibrate, and execute. Standby is depressed by default, calibrate is usually depressed for 30s, and execute is depressed only when valves are clear to fire. When the GNC is activated, the CGS manual controls are deactivated to prevent conflicting controls. System health sensors are provided to the pilot and updated at 5Hz. These are useful for determining whether the gas tanks need to be refueled or the batteries recharged before the next test. Please refer to the Software Operations Guide in Appendix A for detailed instructions on piloting the vehicle.

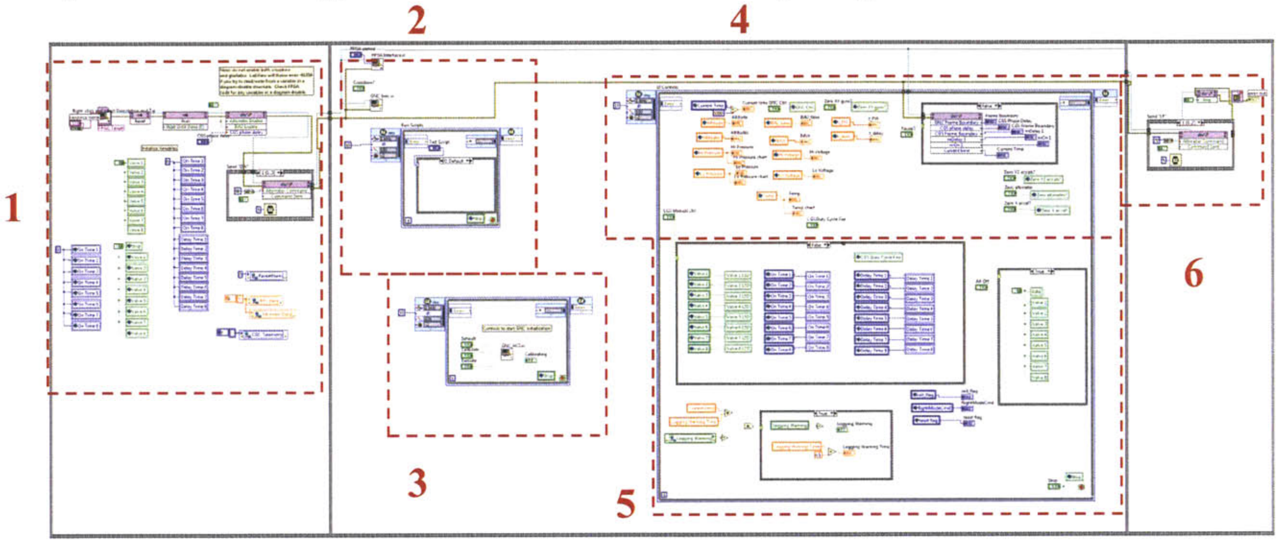


Figure 40: Main PPC.vi Block Diagram

1) This portion initiates "FPGA Main.vi", initializes all global variables to zero, and starts the altimeter.

2) "FPGA Interface.vi" and "GNC Exec.vi" begin execution as soon as the FPGA is initialized. Automated scripts are also executed inside a timed loop, which is controlled from the front panel.

3) The three GNC modes (standby, calibrate, and execute) are monitored in this timed loop and sent to "GNC Exec.vi" to be input into the Draper GNC code.

4) Global variables are written to local variables to be displayed to the front panel.

5) In manual control, the controls on the front panel are read and set to global variables, which will then be sent to the FPGA in "FPGA Interface.vi." When automatic scripts or the GNC code is activated, the front panel controls are set from the global variables to reflect the current state of the CGS. In addition, when logging is not taking place (i.e. when the GSC did not start Main PPC.vi), a warning light will flash on the front panel.

6) After the all-stop button has been pressed, the altimeter is deactivated, and the FPGA is stopped.

4.5.2 FPGA Interface.vi

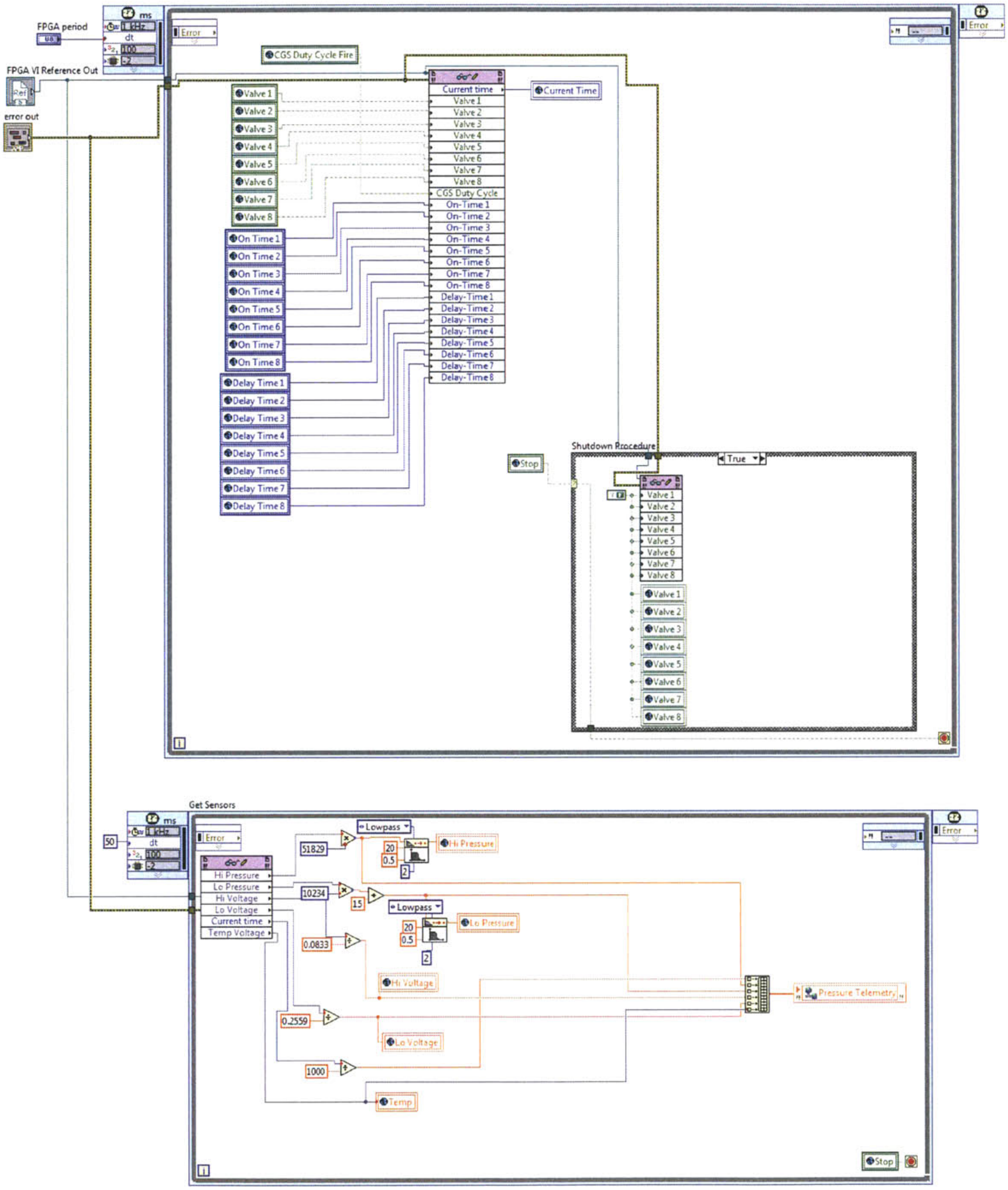
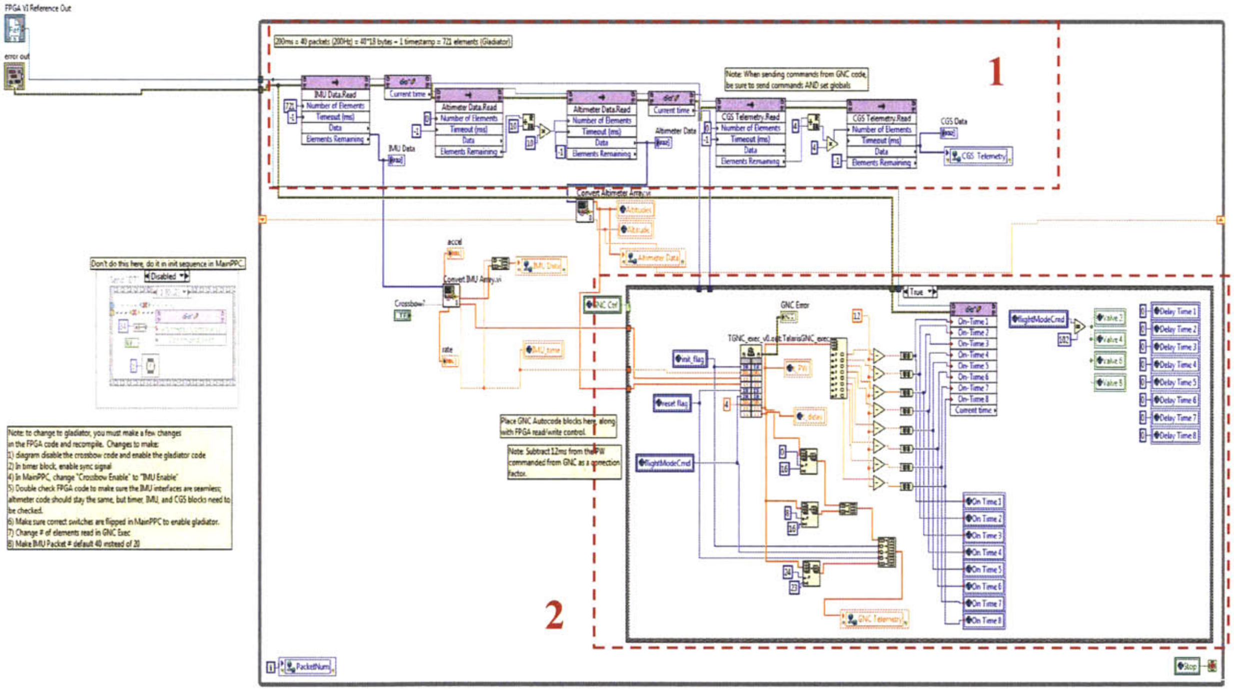


Figure 41: FPGA Interface.vi Block Diagram

"FPGA Interface.vi" sends commands through a single FPGA Read/Write Control every 10ms. In this version of the software, the valve enable booleans, on-times, and delay-times are being sent, although EDF commands can also be sent here. When the all-stop button is pressed, all the valves are closed before stopping the timed loop to ensure vehicle safety after execution. The system health sensors are also read in this vi, converted from volts to appropriate units, and written to global variables so that they may be accessed in "Main PPC.vi." In the case of the pressure sensors, a 2nd order Butterworth filter was also added to smooth out the data. These sensors are also written to the shared variable "Pressure Telemetry," which can be accessed in the GSC for logging.

4.5.3 GNC Exec.vi



200ms = 40 packets (200Hz) = 40*18 bytes + 1 timestamp = 721 elements (Gladiator)

Note: When sending commands from GNC code, be sure to send commands AND set globals

Don't do this here, do it in init sequence in MainPPC.

Note: to change to gladiator, you must make a few changes in the FPGA code and recompile. Changes to make:
1) diagram disable the crossbow code and enable the gladiator code
2) In timer block, enable sync signal
4) In MainPPC, change "Crossbow Enable" to "IMU Enable"
5) Double check FPGA code to make sure the IMU interfaces are seamless; altimeter code should stay the same, but timer, IMU, and CGS blocks need to be checked.
6) Make sure correct switches are flipped in MainPPC to enable gladiator.
7) Change # of elements read in GNC Exec
8) Make IMU Packet # default 40 instead of 20

Place GNC Autocode blocks here, along with FPGA read/write control.

Note: Subtract 12ms from the PW commanded from GNC as a correction factor.

Figure 42: GNC Exec.vi Block Diagram

1) The top portion of the code consists of a series of DMA FIFO Read blocks which read the appropriate number of elements from the IMU, altimeter, and CGS telemetry queues. The first DMA FIFO Read block is the IMU, which waits until 721 elements have been

written to the queue. This represents one 200ms frame's worth of IMU data (i.e. 1 timestamp + 40 IMU packets = 721 elements), and the rest of the code will not execute until this read has completed. In this way, the timing of this loop is not reinforced by a loop timer, but instead by the DMA FIFO that holds the IMU data. After the IMU data is read, it is sent to "Convert IMU Array.vi" and the altimeter DMA FIFO Read block proceeds. Instead of waiting for a certain number of elements, this read block simply reads the number of complete altimeter packets (i.e. 10 elements) that are in the queue. Finally, all the complete CGS telemetry packets (i.e. 4 elements each) are read from the CGS queue.

2) The raw data packets are processed, written to global and shared variables, and input into the Draper GNC algorithm. The outputs of the algorithm consist of commands to be sent to the FPGA and debug telemetry, which are written to shared variables for logging. Please refer to the "Decoding Log Files" documentation [29] for more information regarding the GNC log files.

4.6 GSC Software

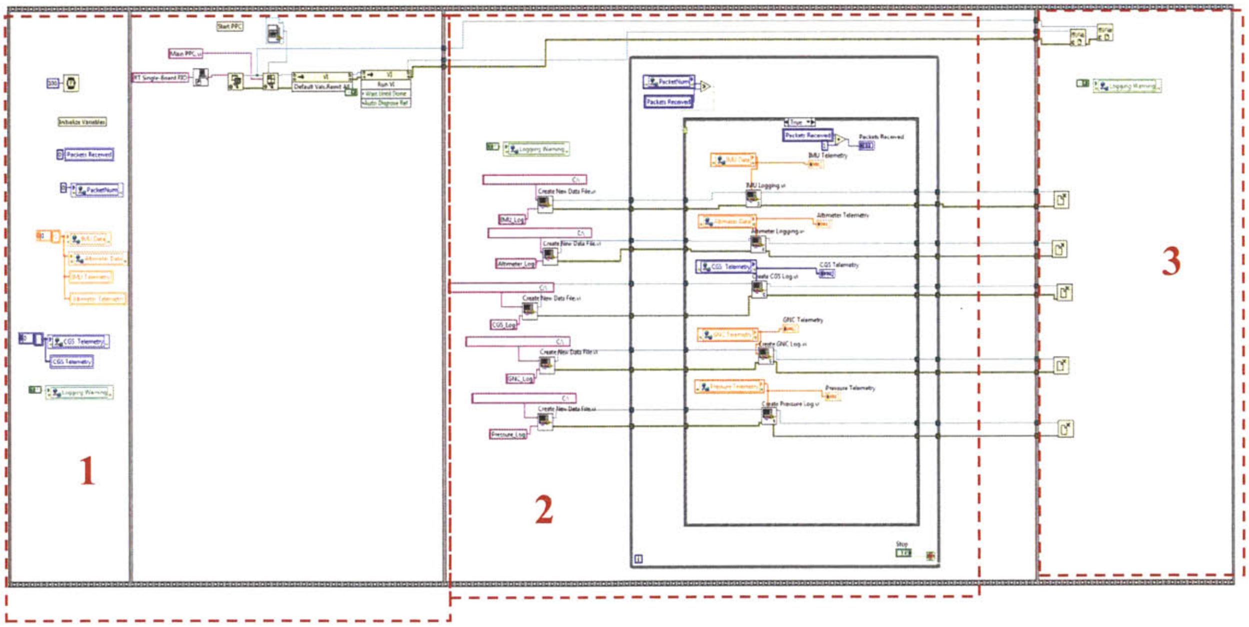


Figure 43: Flight Shell.vi Block Diagram

The main executable for the GSC is called "Flight Shell.vi." Its main purpose is to start "Main PPC.vi" and to write the information in shared variables to txt files on the GSC.

1) The shared variables are initialized to zero. "Main PPC.vi" is started.

2) Logging takes place from information from the shared variables. Separate log files for the EDFs, CGS, IMU, altimeter, GNC, and pressure telemetry are created. The data is written one frame at a time as it is being passed up from the PPC.

3) All text files are closed and "Main PPC.vi" is stopped.

Chapter 5

5 Applying Modularity to TALARIS Software

In this chapter, the application of modularity to the TALARIS software will be explored. The benefits of modularity as stated in Chapter 3 are:

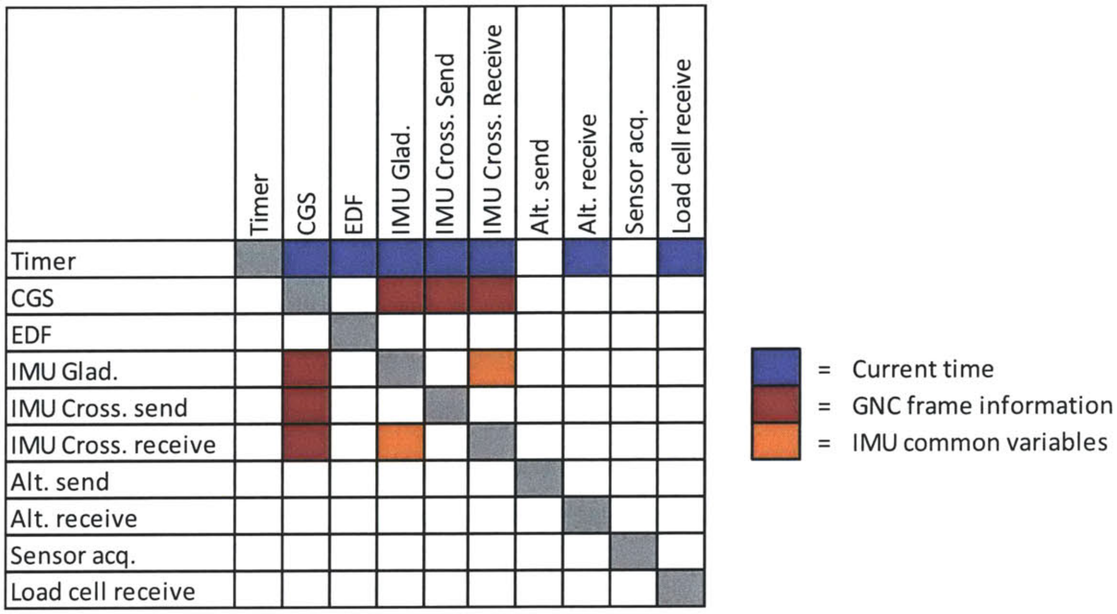
- Increased organization
- Increased flexibility while not compromising base functionality
- Separation of responsibility - makes debugging easier and facilitates development with large teams
- Reduced complexity - makes code easier to understand and is useful for gaining experience in an academic environment
- Ability to easily switch out GNC algorithms speeds testing sessions
- Reusable - common modules can be easily modified for similar functions (i.e. RS-232 interface)

Nearly all these benefits will be applied in the three vehicle phases: development and debugging, operation, and maintenance. In the development and debugging phase, modules will be isolated and analyzed in the FPGA to map their couplings and demonstrate their reconfigurability. Modularity in the PPC will be studied in terms of increasing organization and implementation of sub-vi's to reduce complexity, and debugging examples will show how modularity was utilized to modify GNC timing and aid in benchmark testing. In the operations phase, a series of test campaigns from actuator characterization to GNC tests will illustrate the gradual evolution of the prototype software. Each campaign will be characterized by its main purpose, significant telemetry logged, and FPGA modules. Finally, examples from sensor upgrades will demonstrate the ease of maintaining the modular software and possibility for future expansion.

5.1 Modularity in Development

5.1.1 FPGA Development

The FPGA is characterized by several separate modules for actuators and sensors in a single top-level vi. The FPGA code mainly benefits from the increased organization, separation of responsibility, and increased flexibility. Each sensor or actuator module can be developed and tested individually in a separate project and integrated into the full system at a later time. This ability also increases the flexibility, where modules can be reconfigured into a custom configuration for different test campaigns. For this reconfiguration process to be performed smoothly, the modules must be fairly decoupled so that the removal or addition of a module does not disturb the execution of another module. However, some interactions are inevitable, such as access to a single clock variable. When these interactions occur, the couplings must be thoroughly mapped and documented so changes can be made correctly. The figure below shows the current FPGA modules and their interactions.



	Timer	CGS	EDF	IMU Glad.	IMU Cross. Send	IMU Cross. Receive	Alt. send	Alt. receive	Sensor acq.	Load cell receive
Timer	▨	🟦	🟦	🟦	🟦	🟦		🟦		🟦
CGS		▨		🟥	🟥	🟥				
EDF			▨							
IMU Glad.		🟥		▨		🟧				
IMU Cross. send		🟥			▨					
IMU Cross. receive		🟥		🟧		▨				
Alt. send							▨			
Alt. receive								▨		
Sensor acq.									▨	
Load cell receive										▨

🟦 = Current time
🟥 = GNC frame information
🟧 = IMU common variables

Figure 44: N^2 diagram of FPGA module interactions

There are currently 10 modules coded for the FPGA, and the matrix above shows which modules exhibit coupled interactions. The matrix is relatively sparse, which indicates that most interfaces are de-coupled. The blue entries indicate the variable "Current time" being passed from the timer module to the relevant sensor and actuator modules for timstamping telemetry. The three IMU modules, connected through the orange entries, represent a special case. At any time, only one IMU is used on the vehicle, either the Gladiator or the Crossbow. The Gladiator IMU uses a single RS-485 receive module and an external sync signal generated in the timer block. The Crossbow IMU uses a standard RS-232 interface similar to the altimeter. These IMU modules are intended to be swappable, so they share a few common variables, namely IMU Packet Num and Byte Num, indicated by the orange entries. The final category of interactions, shown in red, involves sharing GNC Frame information. The IMU module generates 200ms frames based on the number of IMU packets it receives (the IMU is synchronized using an external sync signal). This frame boundary is passed to the CGS module in the variable "Cmd frame temp time." The desired CGS frame delay is added to this boundary to form the "CGS frame boundary." The red entries indicate the interaction between the CGS module and the IMU module, either Gladiator or Crossbow.

Depending on the test campaign, different modules may be chosen and compiled in the FPGA software. By using only the modules necessary for the given test, the FPGA compile time is minimized, reducing the turnaround time for software modifications and customizations. For example, during vehicle integration, a series of isolated sensor tests were compiled to test each of the sensors separately. For CGS characterization, only the timer, CGS, sensor acquisition, and load cell receive modules were required. For EDF closed-loop altitude tests, the timer, EDF, altimeter send, altimeter receive, and sensor acquisition modules were needed. Section 5.2 will document several test campaigns, from actuator characterizations to GNC tests, to demonstrate the evolution of TALARIS into a full 6DOF testbed.

5.1.2 PPC Development

The PPC benefits from modularity through the use of sub-vi's to increase organization and reduce complexity. A single flat sequence is used to divide execution into three phases: initialization, execution, and shutdown. The execution phase consists of three main modules: FPGA Interface.vi, GNC Exec.vi, and the user interface. Both FPGA Interface.vi and GNC Exec.vi are sub-vi's to save space on the top-level. The figure below shows an overview of the Main PPC.vi block diagram, with the phases and major modules labeled.

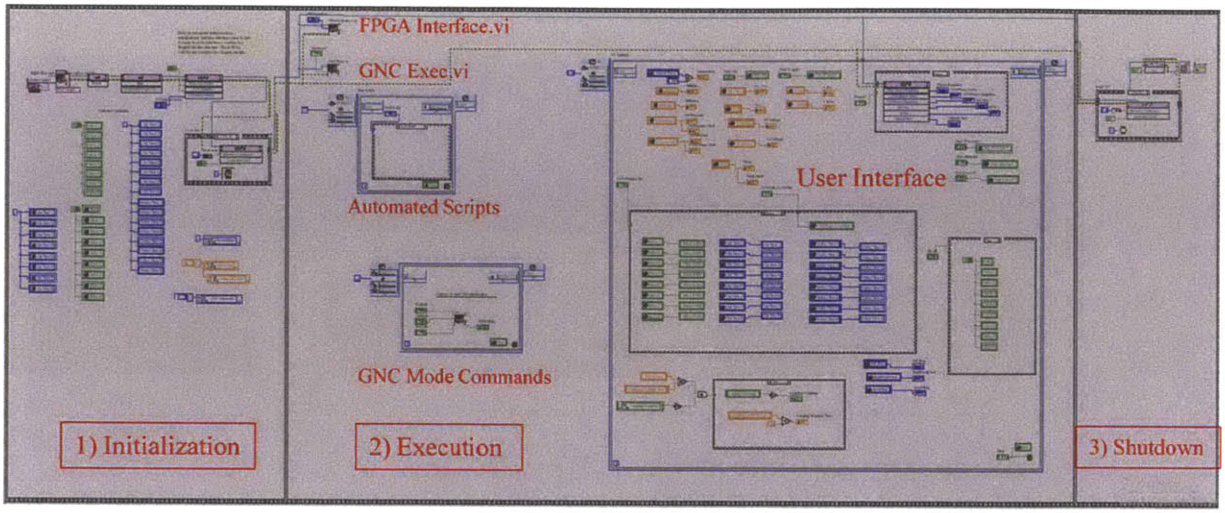


Figure 45: Overview of Main PPC.vi

The sub-vi execution can be thought of as a tree, with Main PPC.vi acting as the main executable and calling sub-vi's in lower levels. GNC Exec.vi in particular has a number of lower level functions to aid in the conversion of raw sensor data, with each forming a separate module. The Call C Library Node also resides in GNC Exec.vi and represents the final benefit of modularity in the PPC code. GNC algorithms may be switched out while keeping the rest of the software unchanged, minimizing the time required for GNC software changes. A block diagram of the sub-vi execution may be seen in the figure below.

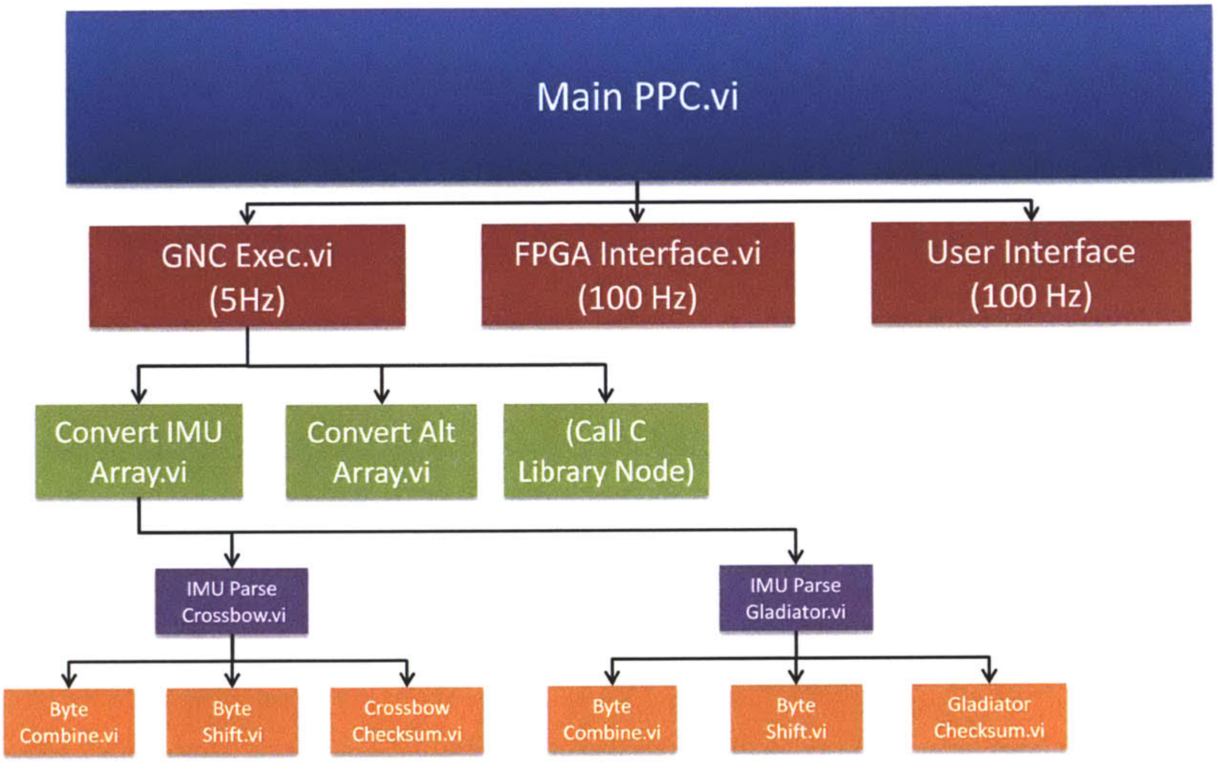


Figure 46: PPC Sub-vi Execution Tree

5.1.3 GNC Timing

The timing modification mentioned in section 4.3 was able to be smoothly and easily implemented through the use of modularity. For this modification, the IMU and CGS modules need to be time-offset to allow time for the PPC to execute the GNC code and send CGS commands back to the FPGA.

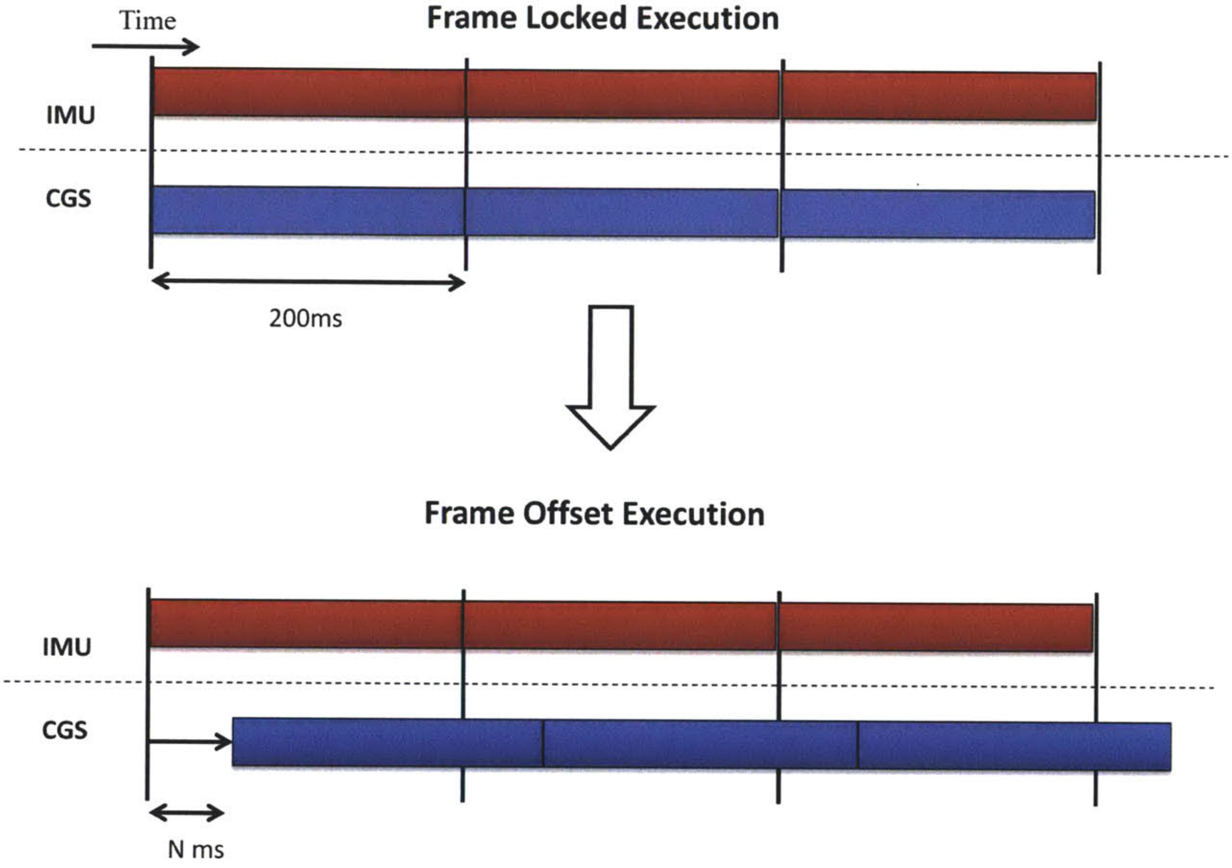


Figure 47: Frame-locked vs. Frame offset execution

Because the CGS and IMU modules were already separated, the delay was implemented using a single case statement. To further increase flexibility, the variable "CGS phase delay" was created to provide a variable offset. This variable could be later tweaked based on benchmark timing tests to minimize delay and maximize performance. The frame offset code can be seen below, as taken from the CGS module. The case statement executes upon the condition that a new 5Hz IMU packet has just been read. The variable "Cmd frame temp time" is written to "GNC Frame Boundary" and "CGS phase delay" is added to it and written to "CGS Frame Boundary." The CGS will then execute its next pulse when "Current Time" passes "CGS Frame Boundary."

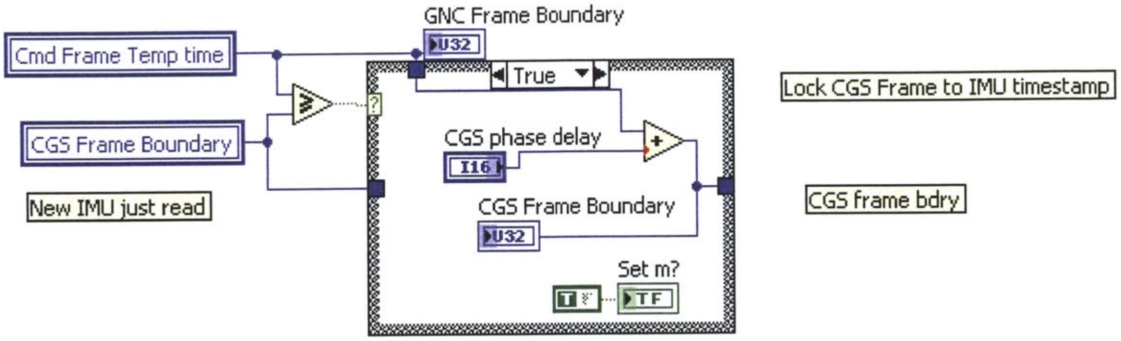


Figure 48: CGS Frame Offset Code

Modularity was useful not only in implementing the frame offset, but also in running the benchmark tests. Module boundaries were used to time intermediate points in the closed-loop control to determine the module that caused the most delay. By sizing the CGS frame offset based on this information, the actuator delay after the GNC command could be minimized while also minimizing the uncertainty from communication delay. Control commands are still executed at exact regular intervals as assumed by the GNC algorithm, resulting in a system which actuates in a manner resembling the model simulations.

5.2 Modularity in Operation

The purpose of this section is to demonstrate the flexibility of the software by providing a brief overview of the number of test configurations of the TALARIS testbed, starting with actuator characterization and ending with GNC algorithm testing. Each test campaign will have customized prototype software composed of only the appropriate FPGA modules. By using only necessary modules, prototype software can be quickly assembled and compile times can be kept low. Also, the FPGA contains only low-level functionality, so all higher level functionality such as swapping controllers, user interface, and logging can be customized in the PPC, which does not require a recompile after modifications. For example, the FPGA will return range data from the altimeter, and the PPC will interpret this range either as an altitude from the ground or translational displacement based on whether the altimeter is pointed downward or horizontally during that test. This also minimizes the number of compiles required for the FPGA code, lowering turnaround times for customized prototype software. To create a discrete embedded system for each of these test campaigns would be very expensive, especially for an academic project, but by utilizing modularity and the flexibility of the RIO platform, a suitable embedded system for each campaign can be created for a fraction of the time and money.

The two main hardware streams on TALARIS are the EDF and CGS propulsion systems. These systems were split at an early stage to perform characterization tests and will be integrated before the full six degree of freedom GNC test. By performing control tests with only one propulsion system, functionality can be demonstrated at an earlier stage, reducing downstream risk. As each propulsion system matures, the prototype software develops as well until the two propulsion systems are combined to perform the final 6DOF traverse. For each campaign below, the main purpose, telemetry, and FPGA modules will be listed, as well as a short summary of the significant results.

5.2.1 Single EDF Testing

Purpose: Demonstrate functionality of single EDF using power supply or LiPo batteries. Characterize RPM and thrust vs. throttle command.

Telemetry: EDF command, RPMs, force data, system health

FPGA Modules: timer, EDF, sensor acquisition

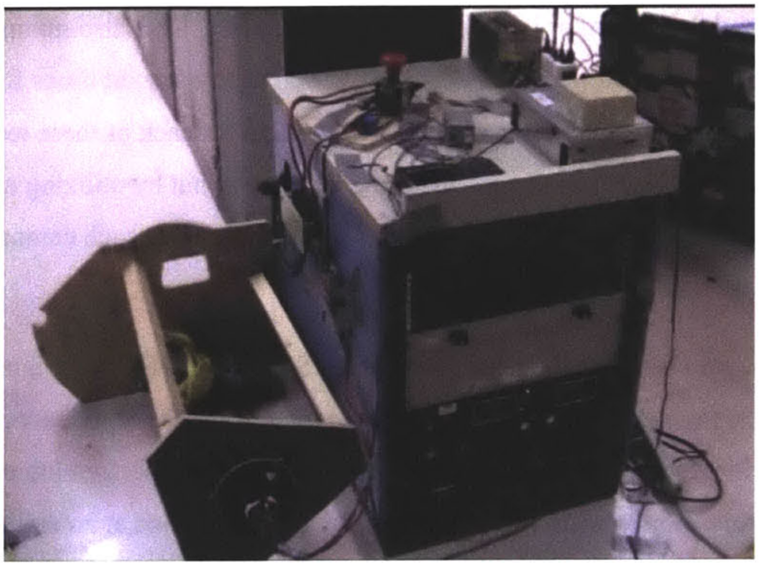


Figure 49: Single EDF Test Setup

The EDFs were the first of the two propulsion systems to be built and tested. The single EDF stand was designed to hook up to either a power supply or the vehicle's lithium polymer battery stack. Open loop control was tested using a 0-100% throttle command, and an RPM vs. throttle curve and thrust vs. RPM curve were characterized.

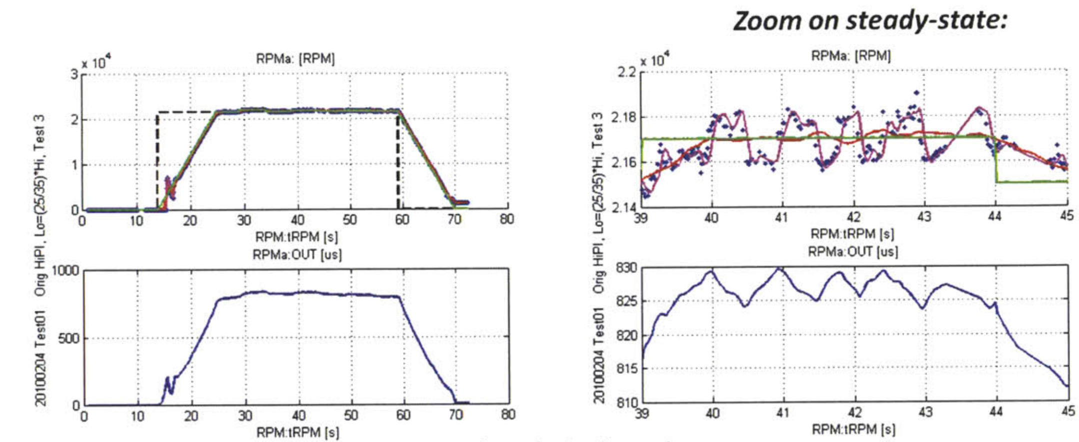
5.2.2 Multi-EDF RPM Testing

Purpose: Integrate four EDFs onto the vehicle. Characterize open-loop RPM response to delta throttle command at various operating points. Continue to characterize thrust vs. RPM. Design and validate closed-loop proportional-integral (PI) RPM controller.

Telemetry: EDF command, RPMs, force data, system health

FPGA Modules: timer, EDF, sensor acquisition

An extensive series of tests were conducted, with all four EDFs installed on the vehicle and the vehicle strapped down to characterize the open-loop response of the RPM to changes in throttle. Then, further characterization of thrust vs. RPM was done, also with all four EDFs installed on the vehicle. Using data from both of these tests, a closed-loop PI RPM controller was designed with variable gains. A sample closed-loop response is shown in the figure below.



Zoom on steady-state:

Top (RPM) plot legend:

Black – Desired profile

Green – Commanded profile

Blue – Raw data/command

Magenta – Filtered data (post-processed)

Bottom plot: "OUT" is the output of the closed-loop controller, i.e., the command to the motor, aka "additional micro-s" beyond 1 ms

Figure 50: Closed-loop RPM Response

The overall profile is shown on the left and a zoomed view of the steady state can be seen on the right. At the start of the ramp, a small spike in RPM can be seen when the motor controller starts the EDFs. After that, during the rest of the ramp up (and subsequent ramp down), the RPMs (blue) appear to nicely follow the commanded ramp profile (green). However, when zooming in on the "steady state", a 200RPM deadband can be seen. The command is varying by about half a percent, but the RPMs stay at discretized levels. The integral action in the closed loop controller introduced behavior similar to dithering, in which the average RPM over a period of time approximately follows the command. This granularity will become important for later altitude tests.

5.2.3 Multi-EDF 1DOF Altitude Testing

Purpose: Integrate altimeter on the vehicle and perform a closed-loop altitude hop on the 1DOF test stand.

Telemetry: EDF command, RPMs, altitude, system health

FPGA Modules: timer, EDF, altimeter send, altimeter receive, sensor acquisition

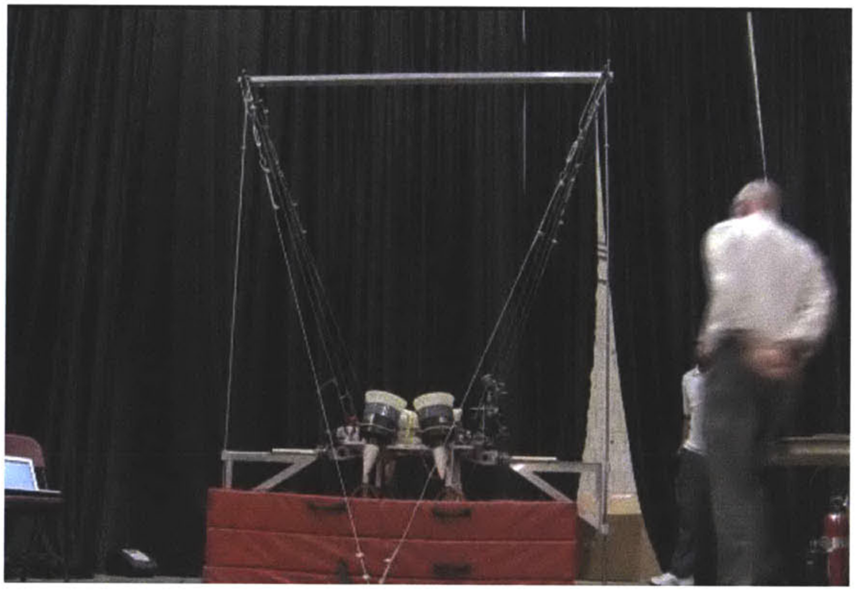


Figure 51: Multi-EDF Altitude Test Stand

For this test, a downward-pointing altimeter was integrated onto the vehicle and was used to wrap a closed-loop PID controller around the RPM inner loop to control the vehicle's altitude. The outer loop received height information from the altimeter and commanded an RPM to the inner closed-loop PI controller based on current vertical position and velocity estimates. For these tests, several outer-loop (altitude) controllers were pre-compiled with different PID gains, ranging from a "gentle" controller to a relatively "aggressive" controller. This streamlined the testing procedure, since prepping for a test requires a few hours and several personnel, but running an altitude test only takes under two minutes of runtime.

Using this controller, a 30cm altitude hop was achieved in 1DOF using the EDFs only. During 6DOF operations, the EDFs will simply be maintained at a set RPM to provide weight offset. However, although not currently in development, a 6DOF closed-loop controller which uses only EDFs could be used as a "parachute mode" in case the CGS system fails. In this case, the EDFs could act as an emergency descent system, controlling both altitude and attitude for a safe landing.

5.2.4 CGS Single Stream

Purpose: Test integrity of flight tanks and plumbing system. Fire CGS valves and determine thrust per valve.

Telemetry: CGS commands, force data, system health

FPGA Modules: timer, CGS, sensor acquisition, load cell receive

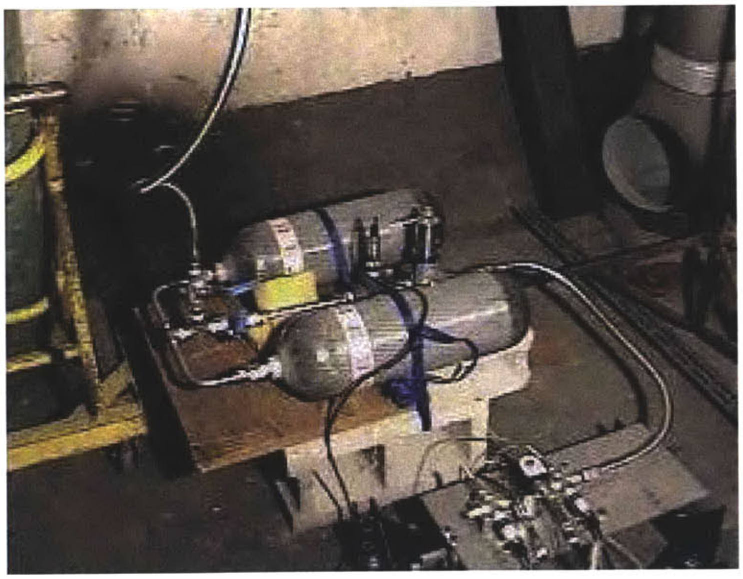


Figure 52: CGS Single Stream Test Stand

The CGS system was first built and tested inside a blast chamber for safety reasons. The single-stream setup prototypes the plumbing system from the tanks to a single valve.

The purpose was to demonstrate CGS valve on/off functionality as well as take load cell data on individual valves. At this point, only the "manual control" mode of the CGS module was coded. The "duty cycle" mode was coded at a later date. In this way, CGS software matured alongside the CGS hardware and was able to be used to test at significant milestones to demonstrate intermediate functionality.

5.2.5 CGS Characterization and Load Cell Testing

Purpose: Integrate CGS hardware onto the vehicle, take force and moment data while on static test stand.

Telemetry: CGS commands, force and moment data, system health

FPGA Modules: timer, CGS, sensor acquisition, load cell receive

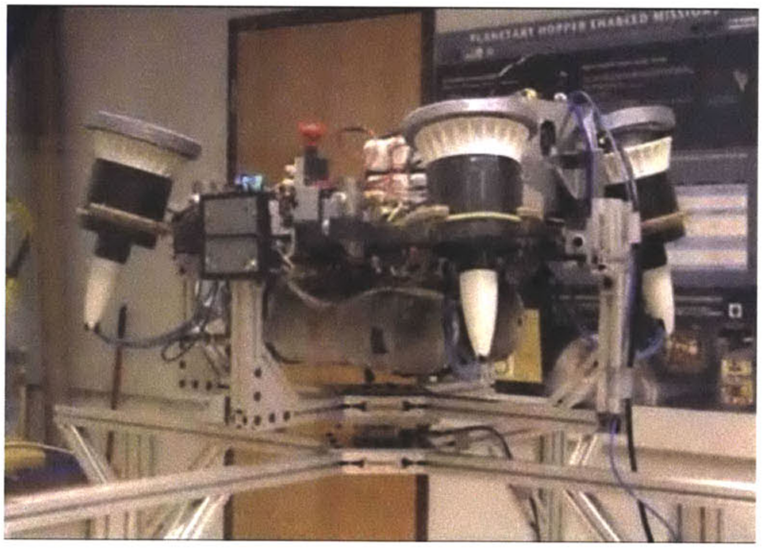


Figure 53: CGS Static Test Stand

After single stream testing, the CGS was integrated onto the vehicle and attached to the static test stand for force and moment characterization. Valves were fired as singles, pairs, and quads to determine the forces and moments imparted onto the vehicle. Also, the entire test stand was lifted two meters off the ground to prevent ground effects from artificially affecting force telemetry. At this point, the CGS "duty cycle" mode was finished and tested to determine the open and close lags associated with pulsing the valves. Of particular interest were the opening and closing time of the valves. The opening time was defined as the time between the commanding of the valve open and the moment maximum steady-state thrust is achieved. The closing time was defined as the time between the off command and time it takes for thrust to reach zero. These parameters determine the minimum pulsewidth and granularity of the CGS controller. A diagram of these parameters can be seen in the figure below.

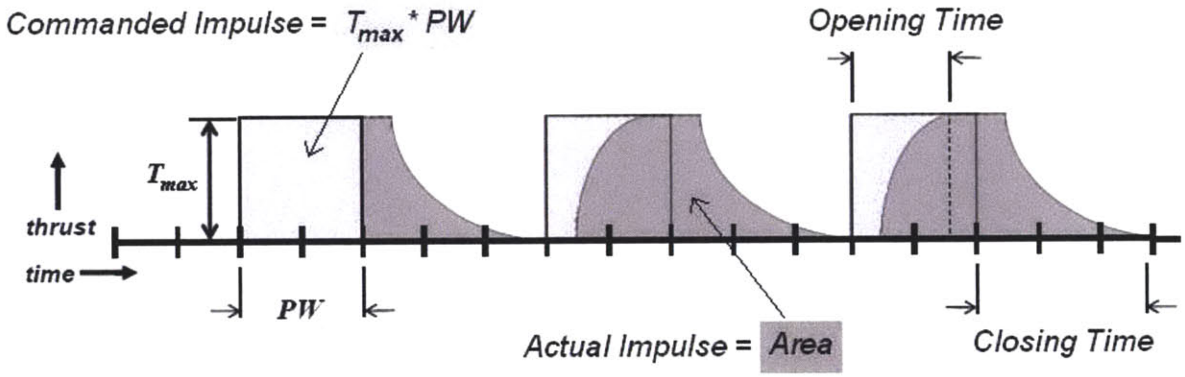


Figure 54: CGS valve firing profile

This version of the software contained many non-flight sensors, such as the 6 DOF load cell, which was returning data points at 500Hz. As such, the software became very telemetry heavy, passing much more data than required for an actual flight. An example of a force profile for a single valve can be seen in the figure below.

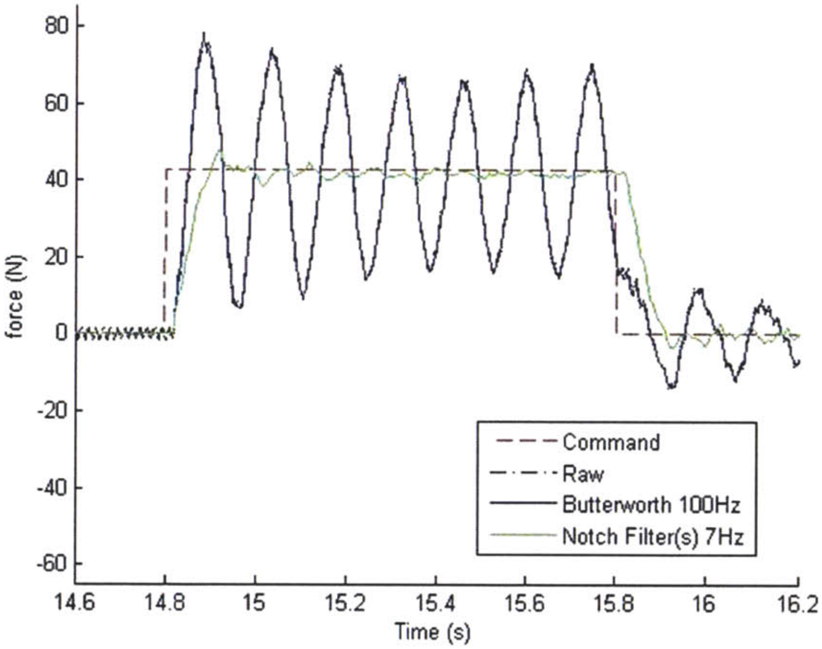


Figure 55: CGS valve force profile

The valves averaged about 40N per valve. The blue measurement above represents the raw force measurement, which displays a significant amount of oscillation. This was determined to be caused by the dynamics of the static stand plus load cell. When applying a notch filter at 7Hz, the fundamental frequency of the ringing, a much cleaner force signal was achieved, as shown in green.

5.2.6 CGS Testing 1DOF Traverse

Purpose: Verify CGS operation in a 1DOF horizontal traverse, assuming level attitude

Telemetry: CGS commands, altimeter telemetry, system health

FPGA Modules: timer, CGS, alt send, alt receive, IMU Gladiator receive, sensor acquisition

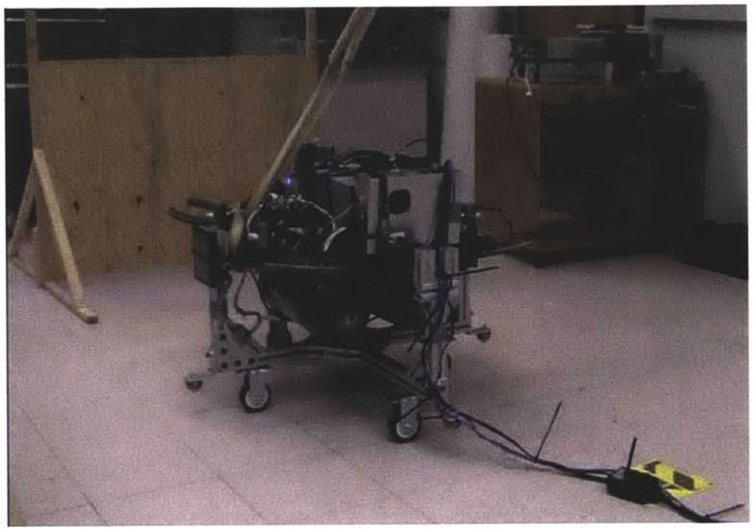


Figure 56: GNC 1DOF Traverse Stand

The first GNC test was a horizontal 1DOF traverse. The cradle from the CGS load cell characterization stand was modified with wheels which were constrained to move only in one direction. The vertical valves were disabled for this test, and EDF weight offset was not required. This would simulate the hop stage after stable hover was achieved, where both altitude and attitude were unchanging. An altimeter was pointed towards the wall to sense position, from which velocity was also derived. This information was passed to a 1DOF PID controller designed by Joseph M. Morrow, a fellow graduate student and Draper Laboratory Fellow, which closed the loop on range and commanded the horizontal thrusters to successfully drive the vehicle to the target. This test validated that the CGS system delivered the thrust expected, and that it could perform well in a closed-loop controller.

5.2.7 GNC Testing 3DOF Traverse

Purpose: Demonstrate a 3DOF roll plus horizontal traverse with Draper GNC software

Telemetry: CGS commands, altimeter telemetry, IMU telemetry, GNC debug telemetry, system health

FPGA Modules: timer, CGS, alt send, alt receive, IMU gladiator receive, sensor acquisition

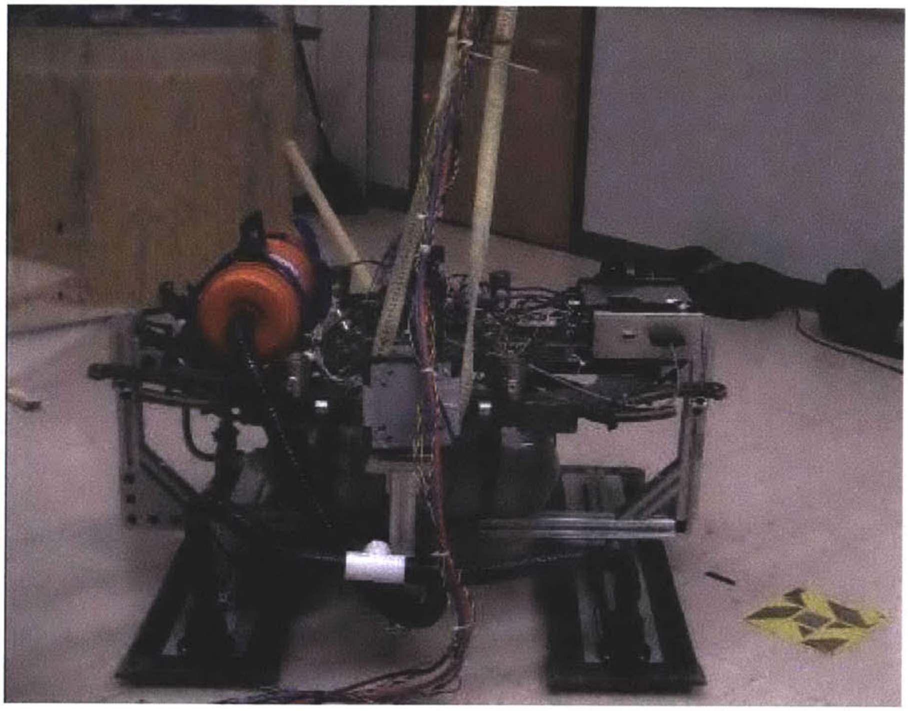


Figure 57: GNC 3DOF Traverse Test Stand

After the success of the 1DOF traverse, a 3DOF test stand was created. An air sled replaced the wheels on the cradle, allowing the vehicle to slide with reduced friction along the floor. First, the IMU was added for attitude and attitude rate about the vehicle's X (vertical) axis, and the Draper GN&C software was loaded onto the vehicle. The vehicle was then commanded to roll 90° without traversing, to mimic roll control during vertical rise, hover, or vertical descent. It successfully rolled 90° in each of four consecutive closed-loop tests. Then the altimeter was also mounted on the vehicle, but pointed towards the wall for position and velocity updates. The vehicle was then oriented 45° away from the wall, and was commanded to re-orient itself towards the wall and traverse 0.8m towards the wall. This also was done successfully for four different closed-loop tests. Not only did these tests successfully demonstrate a portion of the hop profile, but it was the first time the Draper GNC code, which is written in C (Guidance and Navigation) and auto-coded from Simulink (Control), was run in LabVIEW. Further modifications could be made to the Guidance or Navigation C code, or modifications could be made to the Control Simulink model and then auto-coded. Then, the modified GNC C code could be re-compiled along with the executive C code to upgrade the GNC module called by LabVIEW.

5.2.8 GNC Testing 1DOF Attitude

Purpose: Demonstrate ability to maintain stable attitude in 1DOF using CGS pulsewidth firing

Telemetry: CGS commands, altimeter telemetry, IMU telemetry, GNC debug telemetry, system health

FPGA Modules: timer, CGS, alt send, alt receive, IMU gladiator receive, sensor acquisition

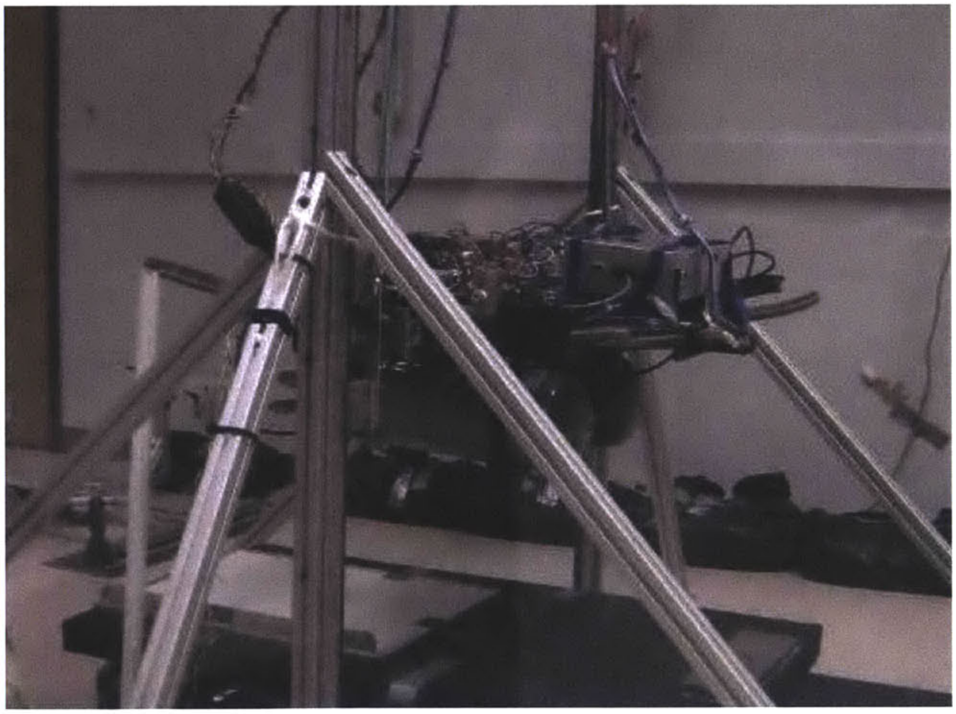


Figure 58: 1DOF Attitude Test Stand

For altitude and attitude tests, a new test stand was built which allows the vehicle to rotate about one axis and traverse upwards and downwards. For this test, the altitude degree of freedom was constrained, constraining the vehicle to only one degree of freedom in attitude. IMU data was used to determine attitude and augmented by a downward-pointing altimeter for regular updates. By firing pairs of thrusters, the vehicle successfully maintained an attitude within 2°. For subsequent tests, the vehicle was artificially weighted towards one end to create a disturbance which the CGS system would have to overcome. Despite this offset, the attitude control algorithm brought the attitude to within 3°. Euler angles from balanced and unbalanced runs are shown in the figures below.

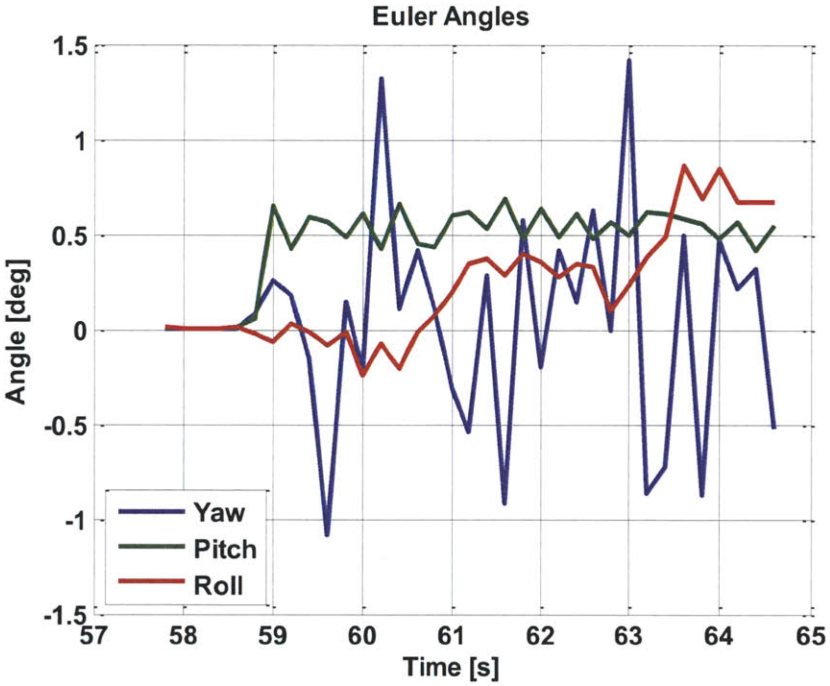


Figure 59: Euler angles for balanced attitude test

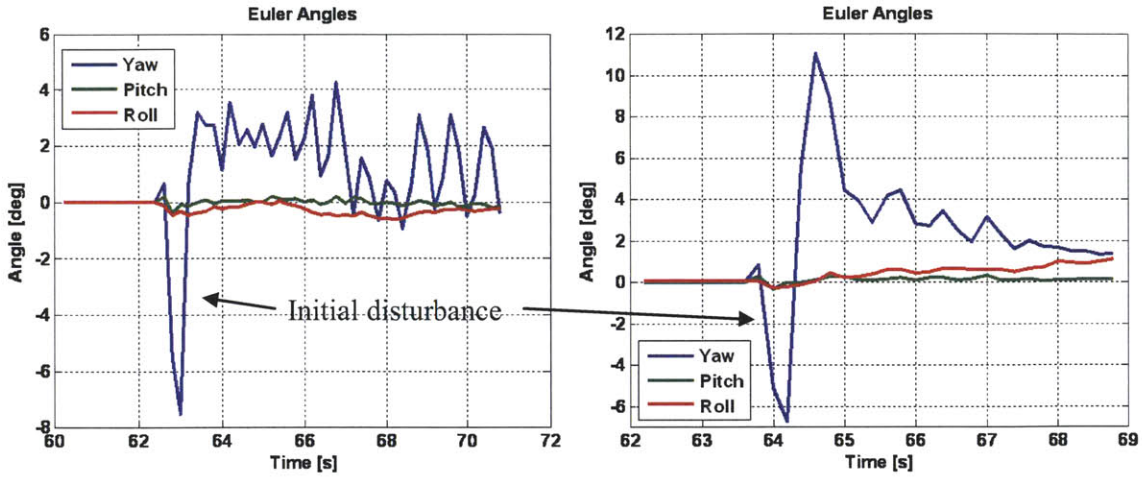


Figure 60: Euler angles for two unbalanced tests

5.2.9 GNC Testing 2DOF Attitude + Altitude

Purpose: Demonstrate ability to maintain stable attitude while performing an altitude hop

Telemetry: CGS commands, altimeter telemetry, IMU telemetry, GNC debug telemetry, system health

FPGA Modules: timer, CGS, alt send, alt receive, IMU gladiator receive, sensor acquisition

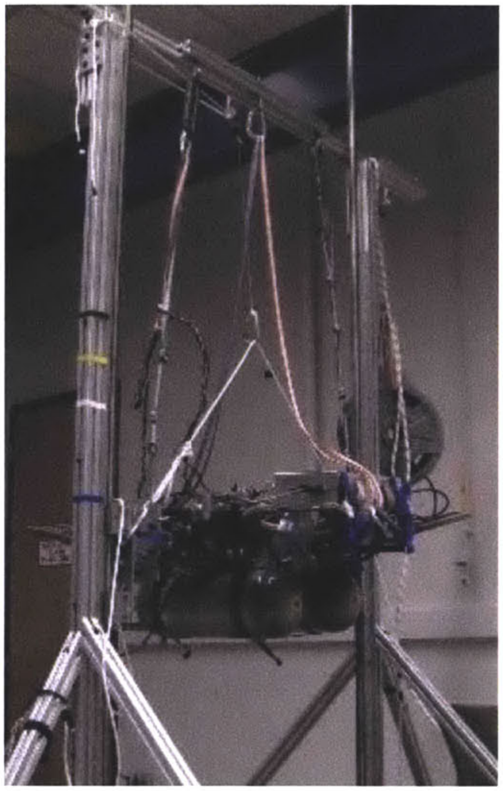


Figure 61: 2DOF Altitude + Attitude Test Stand

The next test freed the altitude degree of freedom, leaving the vehicle to control altitude and one attitude degree of freedom. A weight offset was provided by a pulley system and attached to the four sides of the vehicle. The vehicle was able to maintain a relatively steady attitude, but friction in the rails introduced dynamics in other degrees of freedom which were supposed to be constrained. In practice, the test stand performed more like a 3DOF stand (2 attitude, 1 altitude) rather than a true 2DOF stand. At the time of this writing, the EDFs are being integrated onto the vehicle to replace the pulley weight offset system. This will hopefully alleviate some of the external stand dynamics and result in a cleaner flight. These tests will lead up to a full 6DOF test, where the vehicle will be supported only by a safety harness that is slack during flight. This final flight would represent the full hop profile as described in section 2.2 and is planned for late Spring 2011.

5.3 Software Maintenance Using Modularity

In addition to the benefits in development and operational flexibility, modularity can also have benefits during later phases of vehicle development. In particular, upgrading and customizing the software are facilitated by narrowing the focus to a section of the code. Modules can be exchanged without affecting the rest of the system, eliminating the need to re-validate unrelated portions of the code. This section consists of a series of examples from the TALARIS project demonstrating these benefits during the later stages of development.

5.3.1 Flexibility of the RS-232 interface

The RS-232 communication interface is one of the most prevalent interfaces, with applicability to a wide range of sensors and actuators. On the TALARIS vehicle, both the altimeter and the Crossbow IMU use the RS-232 communication interface, and once one version of the module has been coded, additional sensors can be integrated in a fraction of the time. For example, the Crossbow interface was modified from the altimeter module, as seen below.

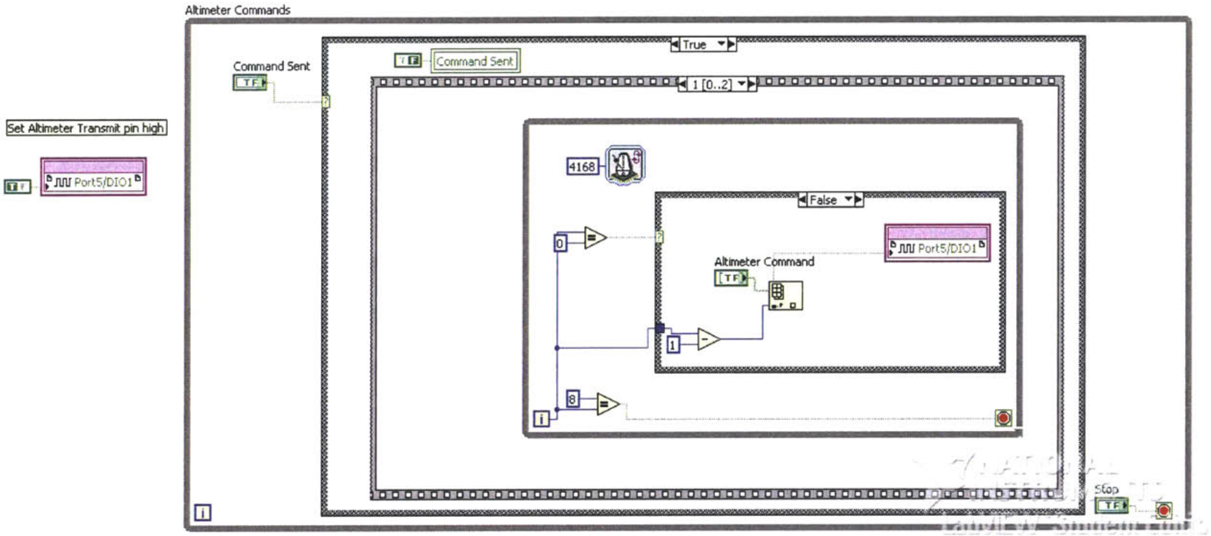


Figure 58: Altimeter Send Module

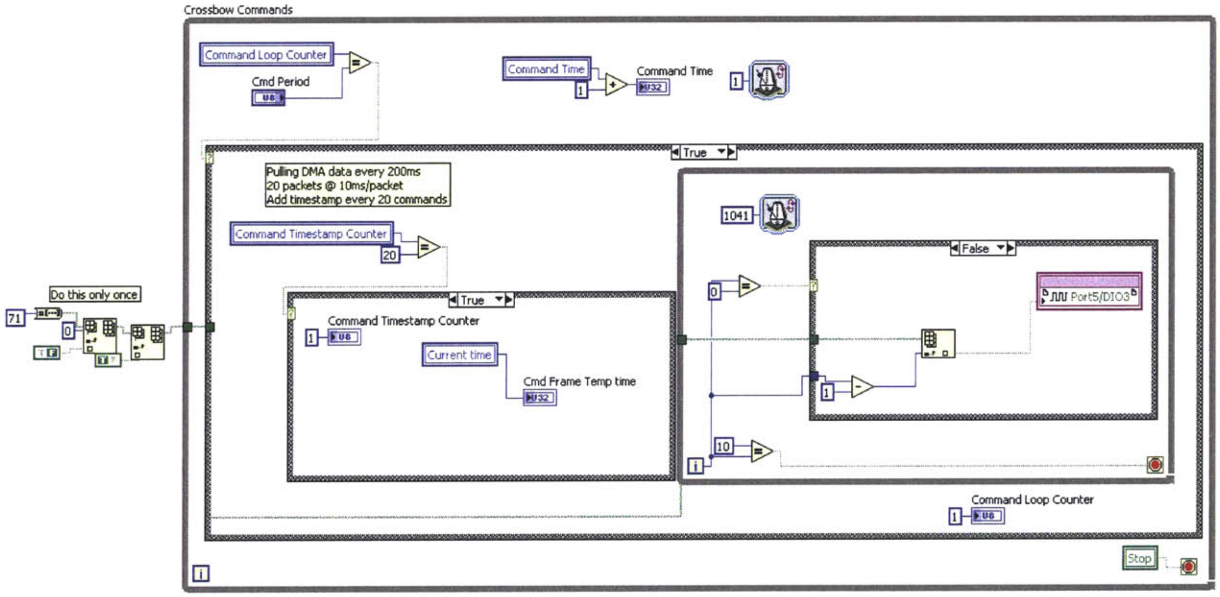


Figure 59: Crossbow IMU Send Modules

The Crossbow IMU send module has additional logic relating to defining the GNC frame and a slightly different command structure, but otherwise is identical to the altimeter send module. Similarly, the RS-232 receive block for both devices is nearly identical, differing only in the I/O pinout accessed, baud rate, and data packet format. In the future, the time to integrate a new device using the RS-232 communication interface is a fraction of the time to code a new module from scratch.

5.3.2 Upgrading IMUs

The vehicle has used a number of IMUs, with the latest upgrade being from the Crossbow IMU to the Gladiator IMU. According to their data sheets, the Gladiator IMU has a higher update rate as well as lower gyro bias, noise, and a lower weight. However, the Gladiator IMU uses the RS-485 protocol instead of RS-232 protocol that the Crossbow uses. The RS-485 module was based off the RS-232 module, using the same falling edge detection, bit read, and DMA FIFO procedure. The data packet of the Gladiator started with a set value of "42," so this was added as an error checking case at the start of each packet read. In the PPC code, the number of elements read from the DMA FIFO was changed to match the Gladiator's data packet format. The logging routine was also updated. Even though the IMU module was one of the most coupled modules, it was still exchanged fairly easily by using an existing module as a baseline and taking into account all interactions with other modules.

The Crossbow module still resides in FPGA Main.vi as a disabled module, so the two IMUs may be interchanged with minimal downtime. A list of changes is summarized in the list below. To switch back to the Crossbow, simply reverse these changes.

In the FPGA:

1) Current time, IMU Packet Num, Byte num, and GNC frame data are shared between the two modules.

2) Baud rate is changed from 38400 (Crossbow) to 115200 (Gladiator)

3) Additional error checking was added to check for a "42" as the first byte in a data packet

4) Gladiator IMU uses a sync signal generated by the RIO to return data. Crossbow IMU requires a regular command to poll data at a synchronized rate.

In the PPC:

1) Number of elements read from the DMA FIFO was updated.

2) Logging routine was updated to reflect new data packet format.

5.3.3 Upgrading RPM sensor

One of the latest upgrades to the vehicle was the RPM sensing circuit. Previously, an analog signal was taken from one phase of the three-phase motor signal output by the motor controls, and the period of the oscillating signal was estimated by the FPGA using zero crossing logic provided by LabVIEW. There were a number of problems with this setup. Theoretically, the period of the analog signal should be proportional to the RPM of the EDF. However, this only holds true at steady state and not during ramp-up or ramp-down transients. Also, the analog signal is prone to noise, especially if the wire carrying the signal was long or was near other electrical lines. Inside the RIO box was especially prone to introducing noise, as close interaction with other signal wires was inevitable. Finally, as mentioned in section 5.2.2, there was a 200RPM granularity apparent in the test data. It was unsure whether this granularity was inherent to the motor controller, noise in the analog signal, or a combination of the two. Several periods were averaged together to act as a rudimentary low pass filter, but seemed to only alleviate, not solve the problem.

A digital RPM circuit was developed by the TALARIS EDF team which returned a square wave with a period proportional to the EDF RPMs. The source of the signal still comes from a single motor phase, but this circuit should reduce the noise inherent to the analog signal. The software module also uses the same edge detection scheme from the RS-232 modules, which has proved to be very accurate due to the FPGA's excellent timing characteristics. These features should make the digi-RPM circuit robust to external noise and so that the RPM telemetry reflects the actual motor phases throughout the test.

5.3.4 Integration of new sensors

The past three sections have focused on upgrading or replacing certain sensors. This section is intended to summarize the process of adding new sensors into the existing modular FPGA code. Here are the appropriate questions to ask before integrating a new sensor:

1) At what frequency do you want to poll the sensor?

2) Does it use a standard communication protocol? At what baud rate do you want to read data packets (digital only)?

3) What is the data packet format that the sensor returns?

4) What is the logging format (e.g. timestamp + 6 data elements) desired?

5) How will you communicate the data? FIFO or FPGA read/write?

6) Can you put it in an existing timed loop? For example, system health sensors log data at 5Hz, which is suitable for plotting overall trends for tests under a minute.

7) Do you need to lock timing to the GNC or CGS frame?

Analog sensors are easily incorporated into one of the RIO's dedicated AI lines. If exact timing is critical, then a DMA FIFO would need to be used to pass up the data at regular intervals. However, the RIO has only three DMA FIFO channels in hardware which are currently being used by the IMU, altimeter, and CGS telemetry. Additional telemetry would need to be augmented onto one of these existing FIFO channels. For example, the IMU channel, which is populated at 200Hz, could consist of a timestamp, one IMU packet, and one data packet from the additional sensor. The PPC code would have to be modified to take into account this extra telemetry as well. Digital sensors are a bit harder to incorporate, but integration could also be very quick if the sensor uses a standard communication protocol. Even if a custom interface is required, many of the basic tools, such as baud rate calculations and edge detection, can be taken from the existing RS-232 modules. By using previous models as a baseline, new modules can be coded in a shorter amount of time than their predecessors, facilitating the learning curve as the software matures.

Chapter 6

6 Discussion and Conclusions

6.1 Thesis Review

This thesis explored the use of an FPGA in conjunction with a PowerPC processor to execute real-time GNC algorithms. By taking advantage of the FPGA's excellent timing characteristics as well as the PPC's processing power, complex algorithms may be executed without sacrificing precision timing. The negative effects from communication delay and associated jitter are minimized through the use of a unique timing procedure which utilizes frames to execute commands at regular intervals. Modularity is used to obtain functional and operational flexibility, allowing the prototype software to mature as the vehicle hardware matures. LabVIEW is used as a high-level programming language to program the FPGA and PPC and is useful in an academic environment for quickly instructing new students to become software developers for embedded systems. Draper GNC algorithms may be coded in their native language and later incorporated as a module separate from the surrounding support software.

After a brief introduction and literature review in Chapter 1, Chapter 2 provided background information regarding the Google Lunar X-Prize as well as a system overview of the TALARIS vehicle. Chapter 3 introduced the theory of flexibility and modularity and discussed techniques to implement them in LabVIEW to facilitate development. Chapter 4 described the architecture of the real-time software with detailed discussions about GNC timing and benchmarking as well as the FPGA, PPC, and GSC code. Chapter 5 concluded the thesis with several examples displaying the benefits of software modularity in the development, operation, and maintenance of the vehicle.

6.2 Future Work

There are several areas of improvement for this study, summarized below:

1) The benefits of modularity are often hard to quantify. These benefits are usually development time, performance, consumption of resources or the "-ilities" such as flexibility and reusability. Estimating the increase in downstream value based on these -ilities can be quite difficult and subject to bias. For example, how does one value the ability to switch IMUs, if required? The personnel time savings for performing such a task could be estimated for a modular and integral architecture, but such estimates will always be a stochastic measure. Similarly, to determine the savings in development time, an integral and a modular software could be coded and the development time recorded. Not only is this an inefficient use of developer time, but such an endeavor would be subject to programmer skill, subtle changes in programming style, and even variations in day-to-day mental fitness. It is clear that formal specifications for determining the benefits of modularity need to be studied as well as time-efficient ways of verifying these benefits.

2) Similarly, the detriments of modularity to a project are also hard to quantify. Modularity may cause a performance hit, consume more resources, or increase initial development time, but these detriments must also be weighed against future reduced development time and platform flexibility. Thus, hard metrics need to be studied to determine the net value of applying modularity to a project.

3) The current FPGA code requires over one hour to compile. While this may not sound like a long time, compiles must be performed after any change in the FPGA code (i.e. any time the "save" button is pressed, even if no functional change was made). This results in a tradeoff between the numbers of changes one can make to the code and the compile time.

For example, if there is a crash and many changes to the FPGA code had been made, the bug cannot be easily narrowed to a single change, especially if the crash is opaque and doesn't return an error code. The problem is somewhat alleviated through modularity, as modules can be compiled separately and tested individually, but the long compile times still limits the forward progress that can be made between compiles.

4) Bugs in the LabVIEW code often cause opaque crashes which don't provide any information to the nature of the crash. This could come in the form of a software freeze, where the pilot is locked out of the controls and must abort LabVIEW, or a full RIO crash, where the RIO must be rebooted afterwards. This problem is especially problematic when coupled with the long FPGA compile times, which provide a severe time penalty for debugging code. The autocoded GNC module in the PPC often causes opaque crashes, as LabVIEW is unable to provide an error code for an error in C. Thus, the GNC software should ideally be debugged before implemented in LabVIEW.

5) The communication of telemetry between the PPC and GSC is handled through shared variables, which are not meant for high frequency logging purposes. Shared variables often write slowly, causes missed or repeated packets. In the past, this has not severely limited the amount of telemetry that could be logged, but in the future, it is possible that the amount of telemetry the vehicle requires exceeds the ability of shared variables to relay the information. An alternative, such as real-time FIFO queues, should be researched and implemented.

References

[1] Mark Maimone, Yang Cheng, and Larry and Matthies, "Two Years of Visual Odometry on the Mars Exploration Rovers," *Journal of Field Robotics*, vol. 24, no. 3, 23 March 2007.

[2] J.J. Biesiadecki and M.W. Maimone, "The Mars Exploration Rover Surface Mobility Flight Software: Driving Ambition," in *IEEE Aerospace Conference*, Big Sky, MT, 2006.

[3] S.D. Howe et al., "The Mars Hopper: an impulse-driven, long-range, long-lived mobile platform utilizing in situ Martian resources," *Proc. IMechE, Part G: J. Aerospace Engineering*, vol. 225, no. 2, pp. 144-153, 2011.

[4] Jijun Lin, Olivier de Weck, Richard de Neufville, Bob Robinson, and David MacGowan, "Designing Capital-Intensive Systems with Architectural and Operational Flexibility Using a Screening Model," *Complex Sciences*, vol. 5, no. 1, pp. 1935-1946, 2009.

[5] Olivier de Weck, Richard de Neufville, and Mathieu Chaize, "Staged Deployment of Communications Satellite Constellationsin Low Earth Orbit," *Journal of Aerospace Computing, Information, and Communication*, vol. 1, no. 3, pp. 119-136, March 2004.

[6] Phillip Cunio, "TALARIS Theory of Flexibility," Massachusetts Institute of Technology, Cambridge, MA 2010, (Unpublished, available on TALARIS file repository).

[7] Claas Olthoff, "Application of Flexibility Principles and Strategies to the TALARIS Avionics System," Massachusetts Institute of Technology, Cambridge, S.M. Thesis 2009.

[8] K.T. Ulrich, "The Role of Product Architecture in the Manufacturing Firm," *Research Policy*, vol. 24, pp. 419-441, 1995.

[9] Christopher-Loic Gaillard, "An Analysis of the Impact of Modularization and Standardization of Vehicles Electronics Architecture on the Automotive Industry," Massachusetts Institute of Technology, Cambridge, MA, SM Thesis 2006.

[10] Katja Holtta, Eun Suk Suh, and Olivier de Weck, "Tradeoff Between Modularity and Performance for Engineered Systems and Products," in *International Conference on Engineering Design*, Melbourne, 2005.

[11] Yuanfang Cai and Sunny Huynh, "An Evolution Model for Software Modularity Assessment," in *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, 2007.

[12] Yang Tan, "Formal Specification Techniques for Promoting Software Modularity, Enhancing Documentation, and Testing Specifications," Massachusetts Institute of Technology, Cambridge, MA, PhD Thesis 1994.

[13] Massimiliano Corba and Zoran Ninkov, "Modular Architecture for Real-Time Astronomical Image Processing with FPGA," Rochester Institute for Technology/Center for Imaging Science, Rochester, NY, 1995.

[14] Simon Falsig and Anders Soerensen, "An FPGA Based Approach to Increased Flexibility, Modularity, and Integration of Low Level Control in Robotics Research," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Taipei, 2010, pp. 6119-6124.

[15] K. KrishnaKumar, J. Kaneshige, R. Waterman, C. Pires, and C. Ippoloito, "A Plug and Play GNC Architecture Using FPGA Components," NASA Ames Research Center, Moffett Field, CA, 2005.

[16] Jianua Liu, Michael Chang, and Chung-Kuan Chen, "An iterative division algorithm for FPGAs," in *ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, 2006.

[17] (2011, February) GLXP Website. [Online]. http://www.googlelunarxprize.org

[18] (2011, February) Next Giant Leap Website. [Online]. http://www.nextgiantleap.com/

[19] G.J. Matranga, C.W. Ottinger, and C.R. Jarvis, and D.C. Gelzer, "Unconventional, Contrary, and Ugly: The Lunar Landing Research Vehicle," Monographs in Aerospace History #35 NASA SP-2004-4535, 2005.

[20] 16.89 Design Team, "16.89 Spring 2008 Design Document," Massachusetts Institute of Technology, Cambridge, MA 2008, (Unpublished, available on TALARIS file repository).

[21] Babak Cohanim et al., "Further Development and Flight Testing of a Prototype Lunar and Planetary Surface Exploration Hopper: Update on the TALARIS Project," Massachusetts Institute of Technology, Pasadena, CA, AIAA Space 2010.

[22] Sarah Nothnagel, "Development of a Cold Gas Spacecraft Emulator System for the TALARIS Hopper," in *Space 2010*, Anaheim, CA, 2010.

[23] Sarah Nothnagel, "New CGSE Circuit Documentation," Massachusetts Institute of Technology, Cambridge, MA 2011, (Unpublished, available on TALARIS file repository).

[24] Omega. (2011, February) Omega SV120 Series Data Sheet. [Online]. http://www.omega.com/Pressure/pdf/SV120_Series.pdf

[25] Schmidt Measurement Systems, Inc., "AR1000 Laser Distance Sensor Specification Sheet," Portland, OR, 2008.

[26] Gladiator Technologies, Inc., "LandMark 30 IMU "LN Series" Data Sheet," Snoqualmie, WA, 2011.

[27] National Instruments Corporation, "sbRIO-96xx Factsheet," 2008.

[28] Michael C. Johnson, "A Parameterized Approach to the Design of Lunar Lander Attitude Controllers," in *AIAA GN&C Conference*, Keystone, CO, 2006.

[29] Christopher Han, "Decoding Log Files," Massachusetts Institute of Technology, Cambridge, MA 2010, (Unpublished, available on the TALARIS file repository).

[30] National Instruments Corporation, "sbRIO-96xx User Guide," 2008.

Appendix A: Software Operations Guide

A.1 Quickstart Guide

Before Test:

1) Run LabView as Administrator
2) Open LabView project file
3) Turn on RIO and plug in all sensors
4) Connect to RIO
5) Deploy MainPPC.vi
6) Open Flight Shell

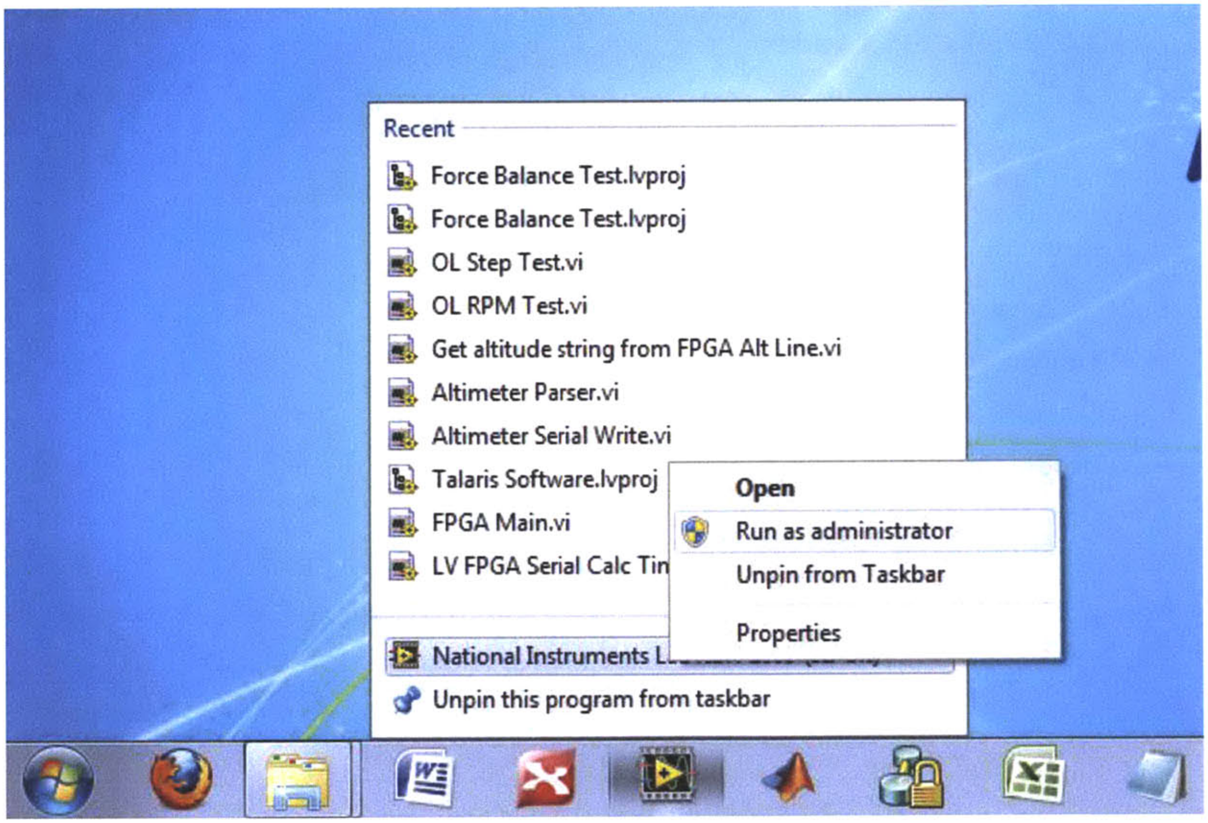
During Test:
1) Start Flight Shell
2) Check for EDF calibration music
3) Run Test
4) Stop both Flight Shell and MainPPC

After Test:
1) Reinitialize Flight Shell and MainPPC to default values
2) Disconnect and power down RIO
3) Upload data to SVN

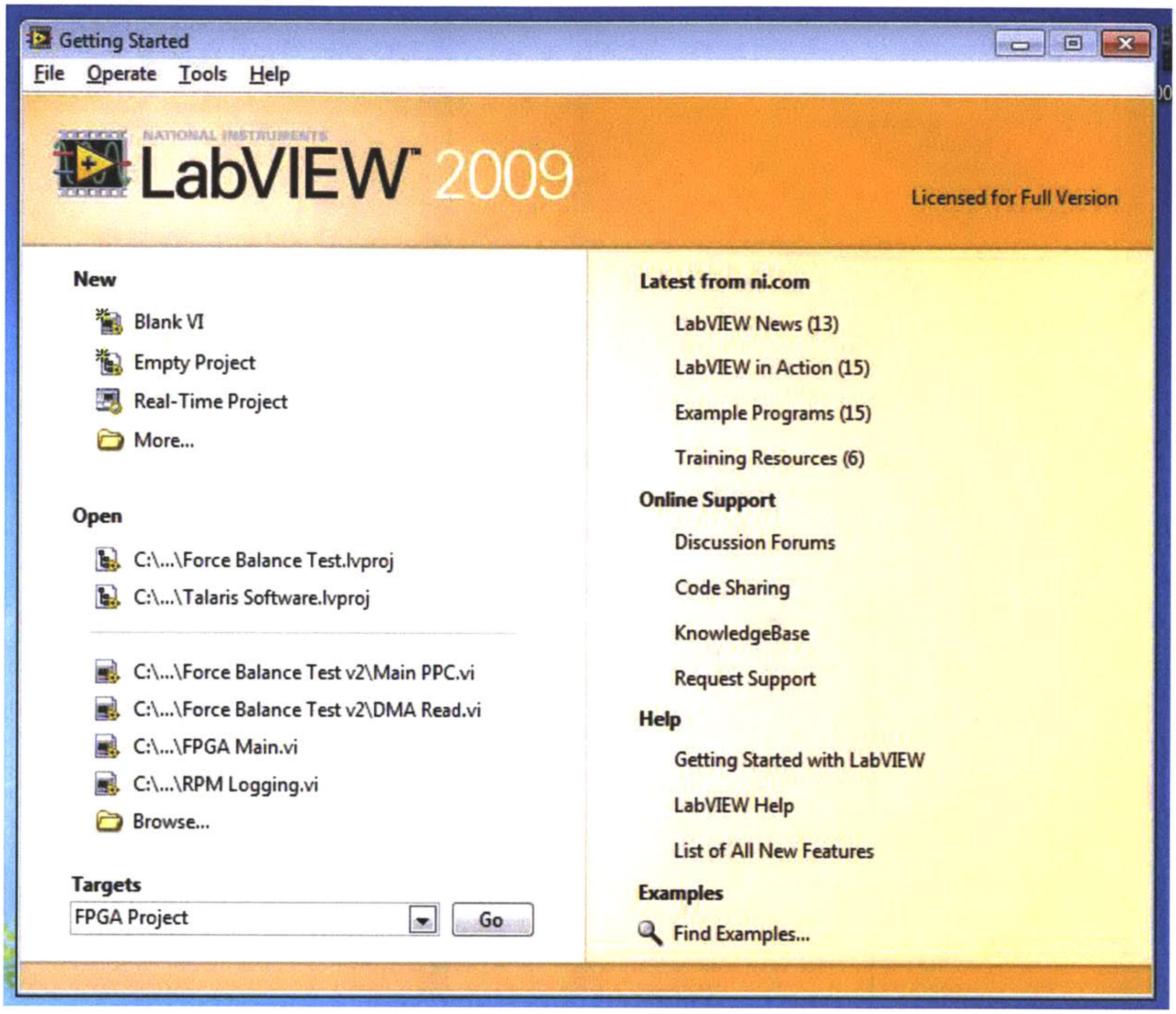
A.2 Loading Software

1) Plug in any sensors you need and turn on RIO. For a list of connectors and pinouts, refer to "RIO Pinout Documentation" on the svn under 3.4-Avionics -> Documentation

2) On the flight computer, run LabView as Administrator: Right click on the Labview icon in the taskbar. Right-click Labview in the list that comes up, and click on "Run as Administrator."



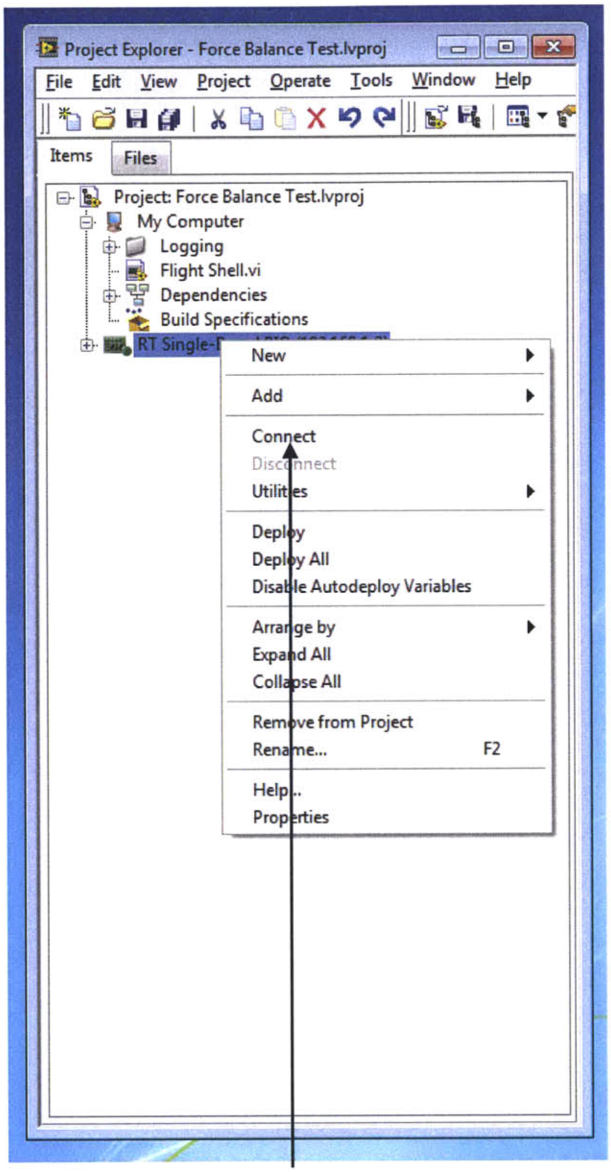
You should see the following LabView window pop up:



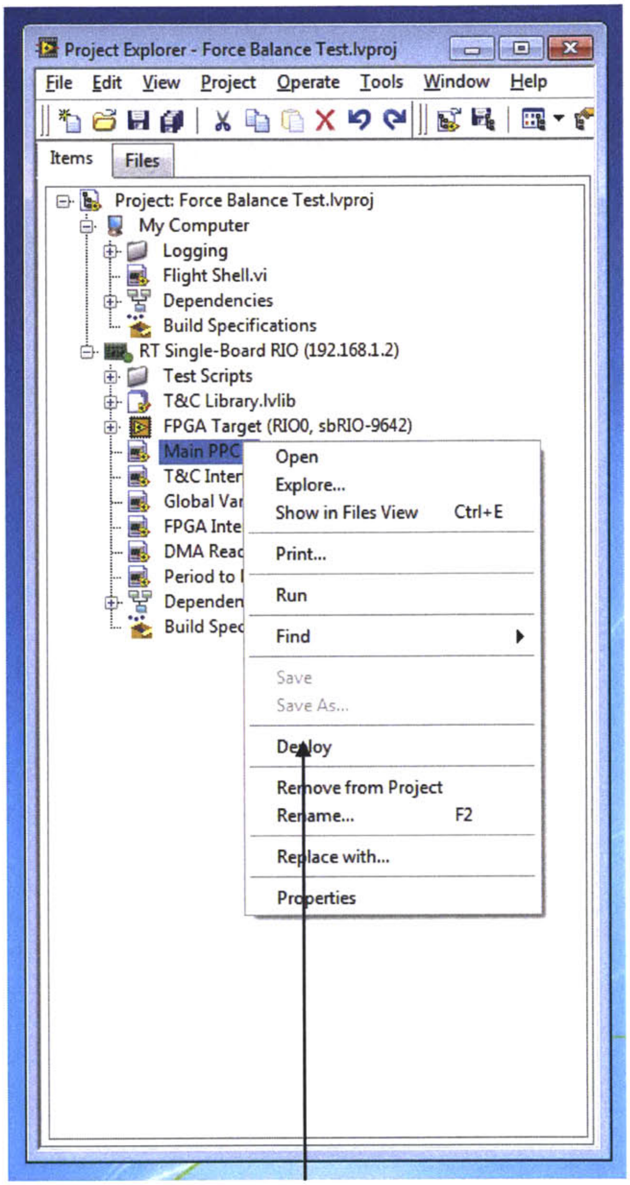
3) Open "Force Balance Test.lvproj", or the appropriate project, which should be located on the desktop. The project explorer should come up.

4) Right-click on the RIO icon, hit "Connect." The green LED in the RIO icon should light upon successful connection.

5) Under the RIO tree, right-click on "MainPPC.vi." Hit "Deploy." Deployment should take about 30 s.



Hit "Connect"



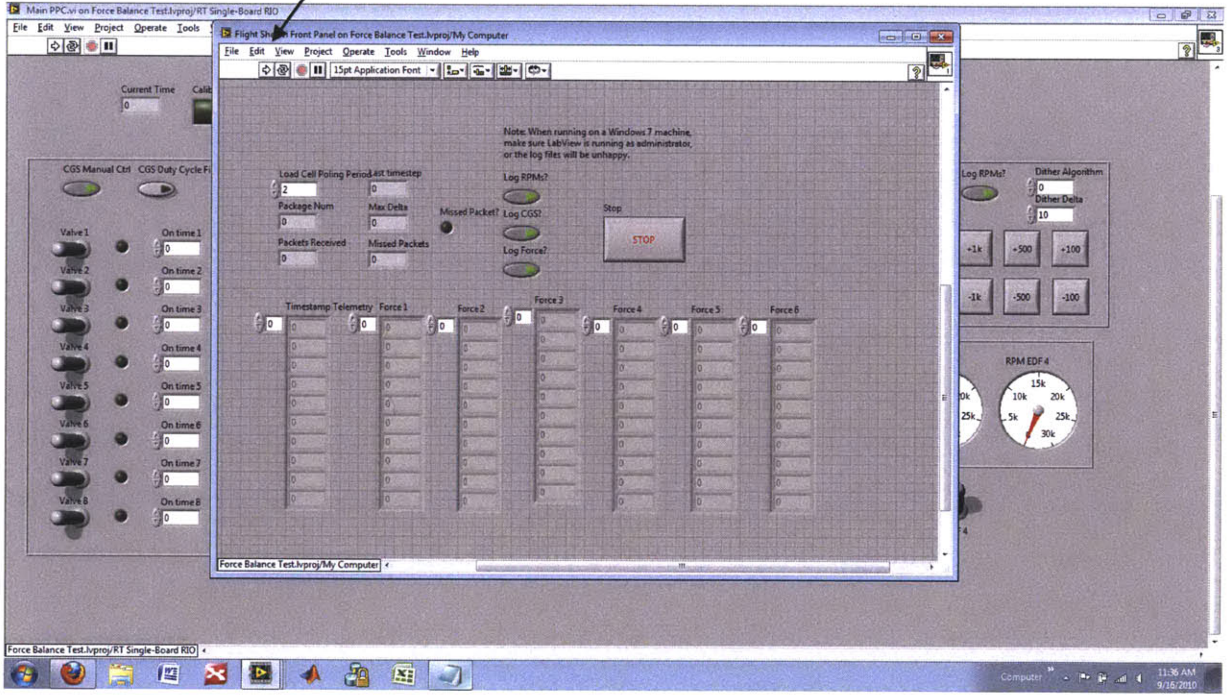
Then "Deploy"

6) Under "My Computer," double-click on "Flight Shell.vi." You are now ready to run software.

A.3 Running Tests

A.3.1 Starting Software

1) Hit the white arrow on "Flight Shell.vi" to start the software. Switch to MainPPC.vi. MainPPC should automatically start running.



Hit this white arrow

to start software

Note: When running on a Windows 7 machine, make sure LabView is running as administrator, or the log files will be unhappy.

Load Cell Poling Period: 2

Last timestep: 0

Package Num: 0

Max Delta: 0

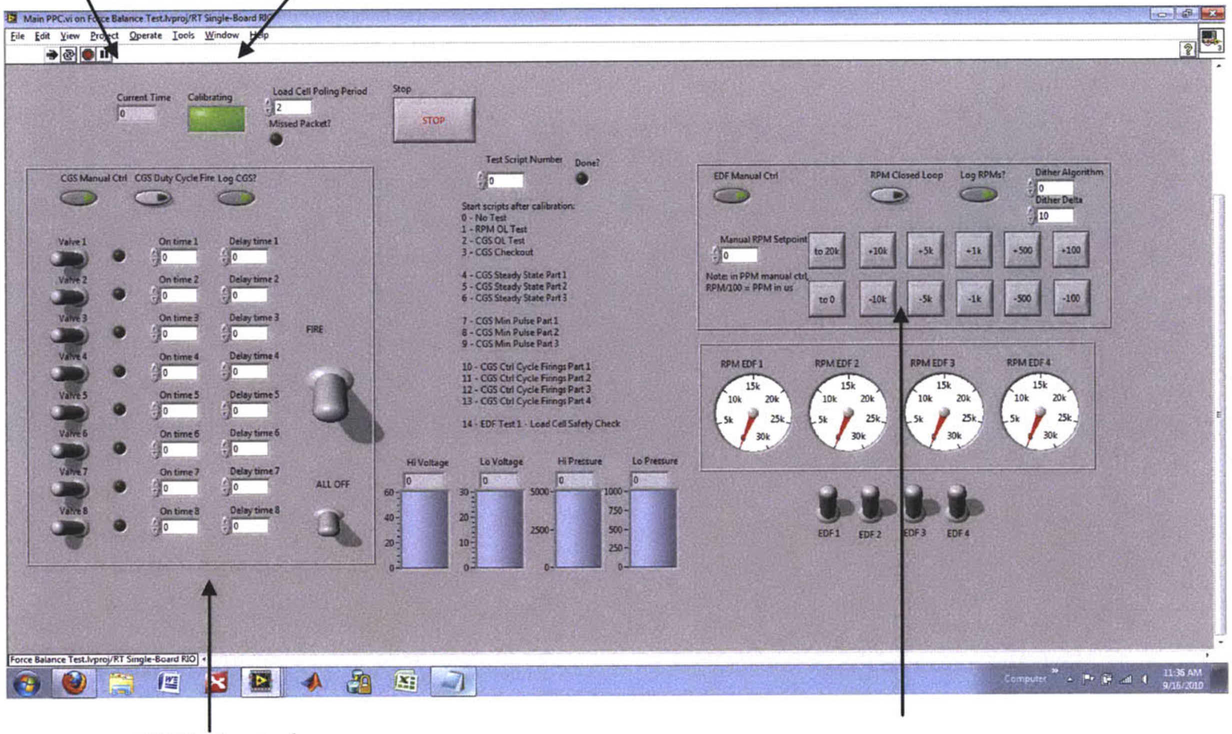
Packets Received: 0

Missed Packets: 0

2) The calibration light should come on. After 10s, time should start running.

Current time will start after calibration is complete.

This light should come on for about 10s



Current Time: 0

Calibrating

Load Cell Poling Period: 2

Missed Packet?

Stop

Test Script Number: 0

Done?

Start scripts after calibration:
0 - No Test
1 - RPM OL Test
2 - CGS OL Test
3 - CGS Checkout

4 - CGS Steady State Part 1
5 - CGS Steady State Part 2
6 - CGS Steady State Part 3

7 - CGS Min Pulse Part 1
8 - CGS Min Pulse Part 2
9 - CGS Min Pulse Part 3

10 - CGS Ctrl Cycle Firings Part 1
11 - CGS Ctrl Cycle Firings Part 2
12 - CGS Ctrl Cycle Firings Part 3
13 - CGS Ctrl Cycle Firings Part 4

14 - EDF Test 1 - Load Cell Safety Check

Valve	On time	Delay time
Valve 1	On time 1: 0	Delay time 1: 0
Valve 2	On time 2: 0	Delay time 2: 0
Valve 3	On time 3: 0	Delay time 3: 0
Valve 4	On time 4: 0	Delay time 4: 0
Valve 5	On time 5: 0	Delay time 5: 0
Valve 6	On time 6: 0	Delay time 6: 0
Valve 7	On time 7: 0	Delay time 7: 0
Valve 8	On time 8: 0	Delay time 8: 0

Hi Voltage: 0

Lo Voltage: 0

Hi Pressure: 0

Lo Pressure: 0

Dither Algorithm: 0

Dither Delta: 10

Manual RPM Setpoint: 0

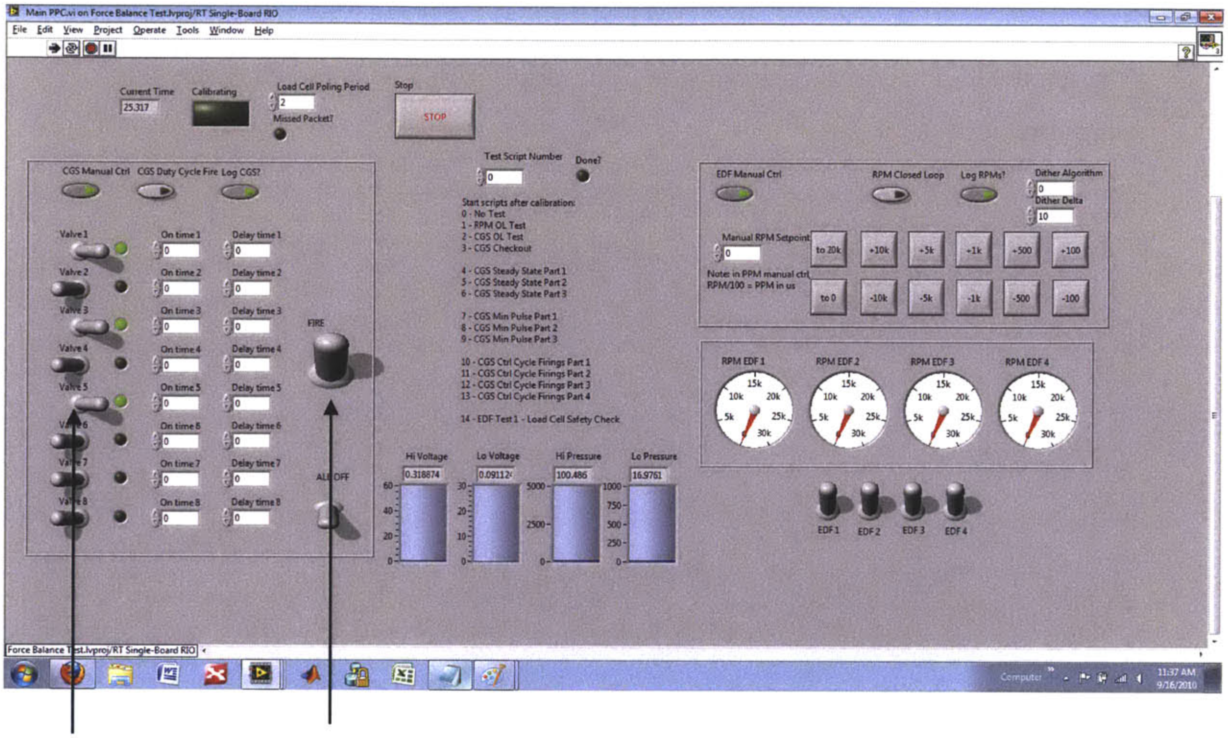
Note: in PPM manual ctrl, RPM/100 = PPM in us

CGS Controls

EDF Controls

3) Manual controls are enabled by default. CGS controls on the left, EDF controls on the right.

4) For manual CGS control, select any number of valves and hit "Fire." They should click on/off with the Fire button.



Main PPC.vi on Force Balance Test.lvproj/RT Single-Board RIO

Current Time 25.317

Load Cell Poling Period 2

Test Script Number 0

Start scripts after calibration:
0 - No Test
1 - RPM OL Test
2 - CGS OL Test
3 - CGS Checkout

4 - CGS Steady State Part 1
5 - CGS Steady State Part 2
6 - CGS Steady State Part 3

7 - CGS Min Pulse Part 1
8 - CGS Min Pulse Part 2
9 - CGS Min Pulse Part 3

10 - CGS Ctrl Cycle Firings Part 1
11 - CGS Ctrl Cycle Firings Part 2
12 - CGS Ctrl Cycle Firings Part 3
13 - CGS Ctrl Cycle Firings Part 4

14 - EDF Test 1 - Load Cell Safety Check

On time 1 0, Delay time 1 0
On time 2 0, Delay time 2 0
On time 3 0, Delay time 3 0
On time 4 0, Delay time 4 0
On time 5 0, Delay time 5 0
On time 6 0, Delay time 6 0
On time 7 0, Delay time 7 0
On time 8 0, Delay time 8 0

Hi Voltage 0.318874
Lo Voltage 0.091124
Hi Pressure 100.486
Lo Pressure 16.9761

Dither Algorithm 0
Dither Delta 10
Manual RPM Setpoint 0

Note: in PPM manual ctrl RPM/100 = PPM in us

Select valves

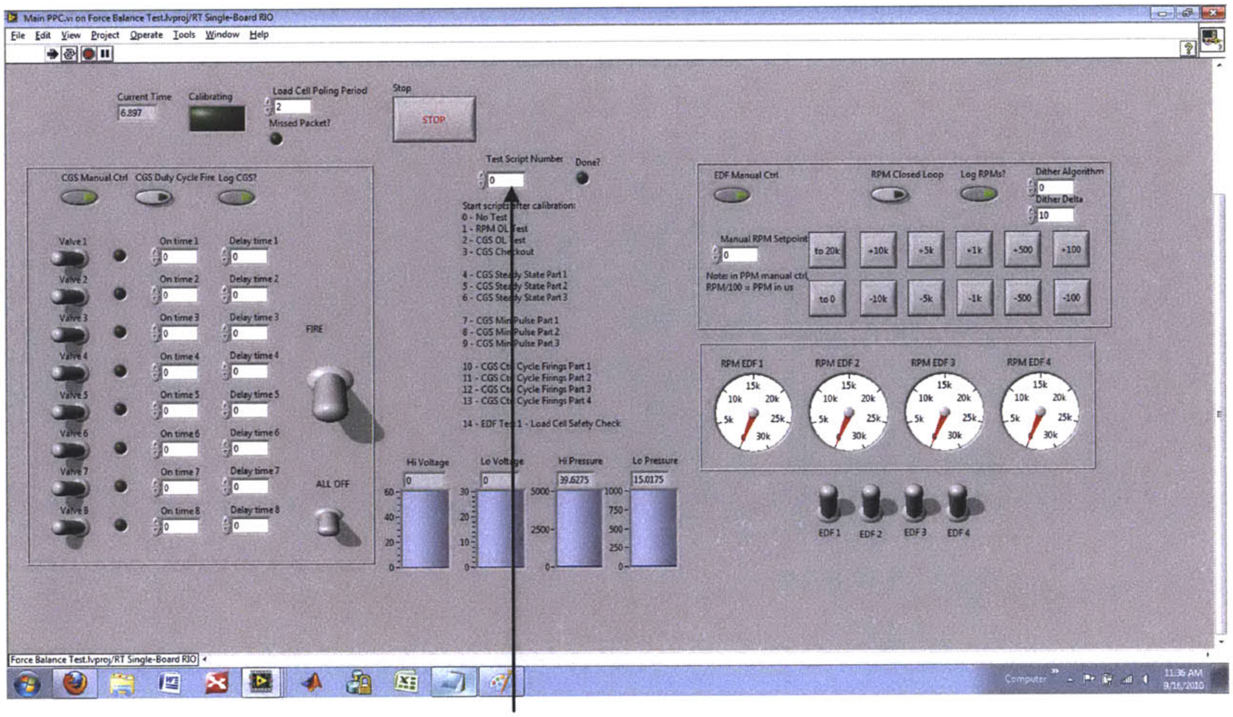
Fire Button

5) For manual duty cycle CGS control, hit the switch "CGS Duty Cycle Fire." Flip the appropriate number of valves, enter their "On-Time" and "Delay-Time" and hit Fire. They should pulse according to the defined parameters.

6) For manual EDF, hit the buttons or enter a number to increase or decrease the throttle command. The conversion is 1000RPM = 1%.

A.3.2 Running Scripts

1) To run scripts, enter the appropriate script number into the "Test Script Number" input. The script will immediately start running. All scripts have a 5s delay before the first actuation.



Main PPC.vi on Force Balance Test.lvproj/RT Single-Board RIO

Current Time: 6.897

Load Cell Poling Period: 2

Test Script Number: 0

Start scripts after calibration:
0 - No Test
1 - RPM OL Test
2 - CGS OL Test
3 - CGS Checkout

4 - CGS Steady State Part 1
5 - CGS Steady State Part 2
6 - CGS Steady State Part 3

7 - CGS Min Pulse Part 1
8 - CGS Min Pulse Part 2
9 - CGS Min Pulse Part 3

10 - CGS Ctrl Cycle Firings Part 1
11 - CGS Ctrl Cycle Firings Part 2
12 - CGS Ctrl Cycle Firings Part 3
13 - CGS Ctrl Cycle Firings Part 4

14 - EDF Test 1 - Load Cell Safety Check

Hi Voltage: 0
Lo Voltage: 0
Hi Pressure: 39.6275
Lo Pressure: 15.0175

Manual RPM Setpoint: 0
Note: in PPM manual ctrl, RPM/100 = PPM in us

Dither Algorithm: 0
Dither Delta: 10

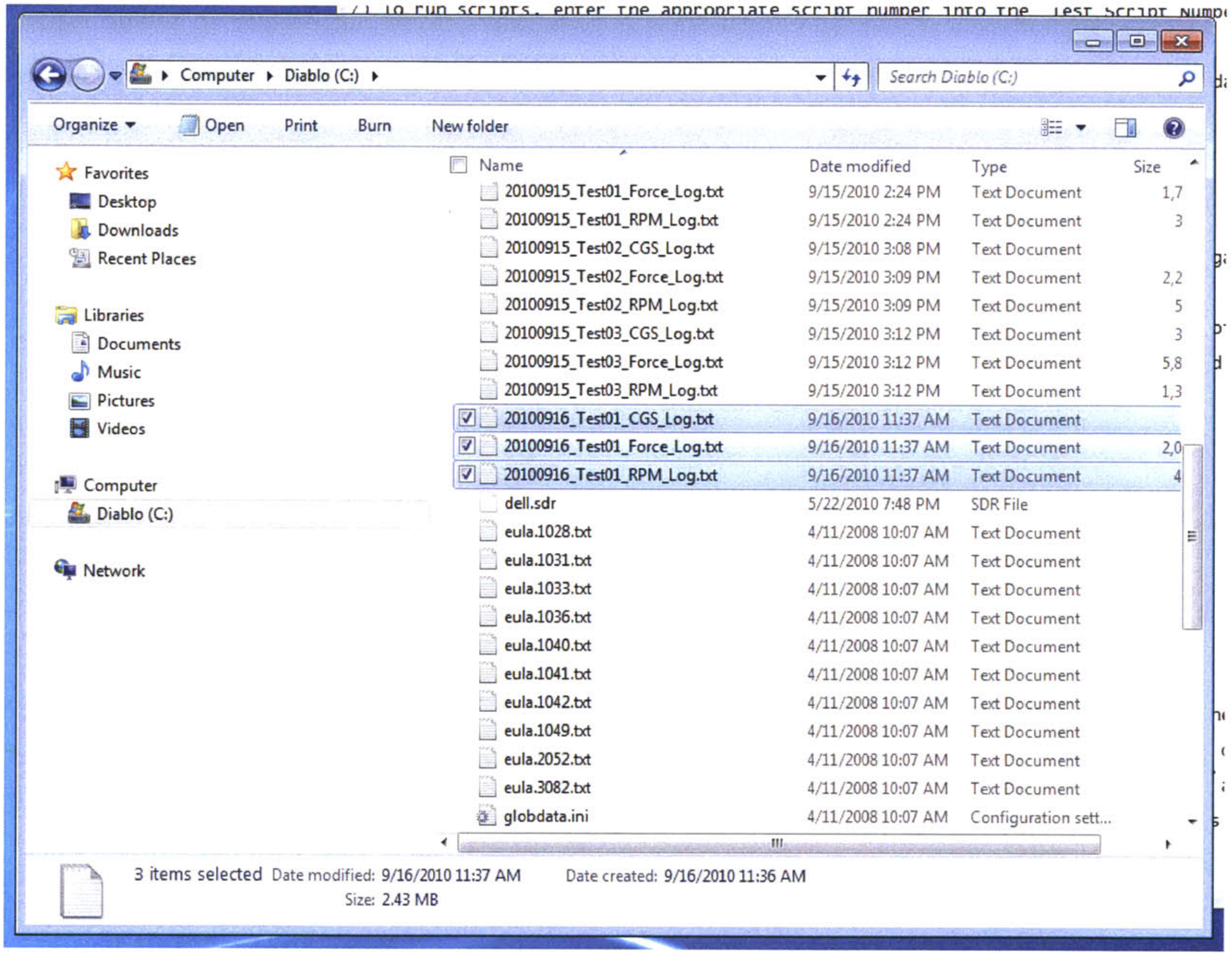
On time 1–8: 0
Delay time 1–8: 0

Force Balance Test.lvproj/RT Single-Board RIO

Test Script Input

2) To stop a test, you must hit the "Stop" button on BOTH MainPPC.vi and Flight Shell.vi.

3) Log files are located under the C:\ directory. They are automatically dated and numbered to avoid overwriting files.

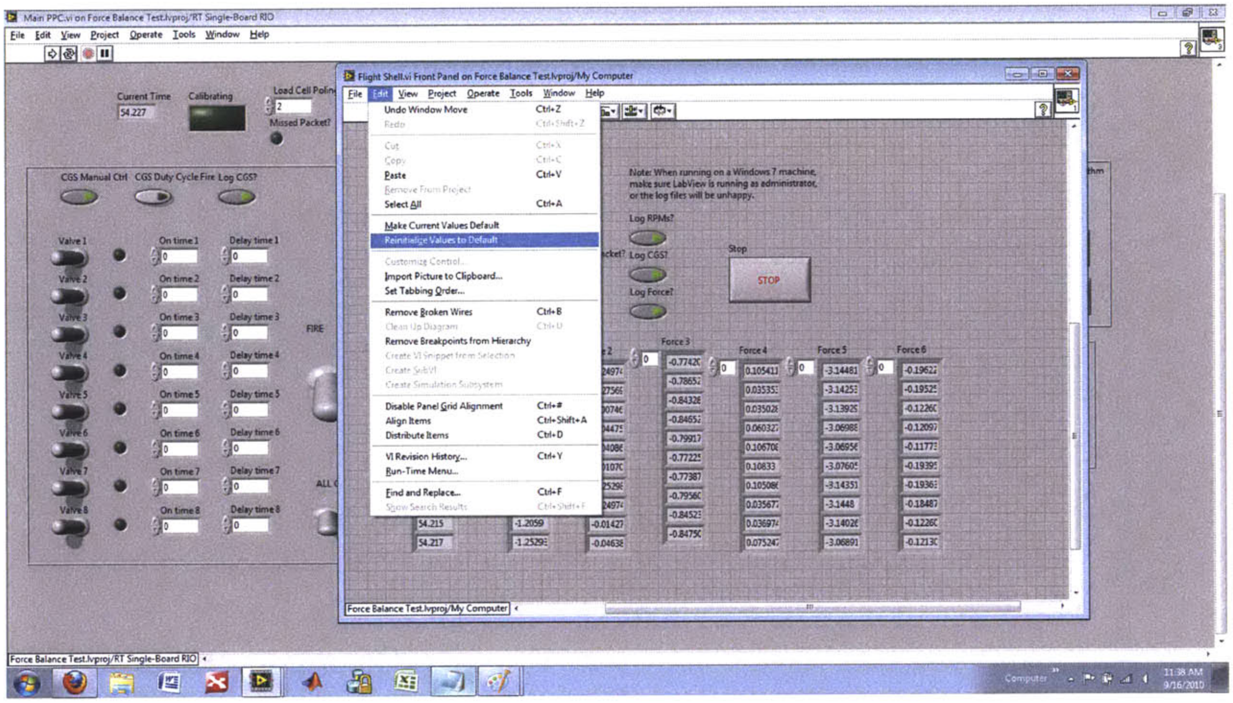


□ Name	Date modified	Type	Size
20100915_Test01_Force_Log.txt	9/15/2010 2:24 PM	Text Document	1,7
20100915_Test01_RPM_Log.txt	9/15/2010 2:24 PM	Text Document	3
20100915_Test02_CGS_Log.txt	9/15/2010 3:08 PM	Text Document	
20100915_Test02_Force_Log.txt	9/15/2010 3:09 PM	Text Document	2,2
20100915_Test02_RPM_Log.txt	9/15/2010 3:09 PM	Text Document	5
20100915_Test03_CGS_Log.txt	9/15/2010 3:12 PM	Text Document	3
20100915_Test03_Force_Log.txt	9/15/2010 3:12 PM	Text Document	5,8
20100915_Test03_RPM_Log.txt	9/15/2010 3:12 PM	Text Document	1,3
☑ 20100916_Test01_CGS_Log.txt	9/16/2010 11:37 AM	Text Document	
☑ 20100916_Test01_Force_Log.txt	9/16/2010 11:37 AM	Text Document	2,0
☑ 20100916_Test01_RPM_Log.txt	9/16/2010 11:37 AM	Text Document	4
dell.sdr	5/22/2010 7:48 PM	SDR File	
eula.1028.txt	4/11/2008 10:07 AM	Text Document	
eula.1031.txt	4/11/2008 10:07 AM	Text Document	
eula.1033.txt	4/11/2008 10:07 AM	Text Document	
eula.1036.txt	4/11/2008 10:07 AM	Text Document	
eula.1040.txt	4/11/2008 10:07 AM	Text Document	
eula.1041.txt	4/11/2008 10:07 AM	Text Document	
eula.1042.txt	4/11/2008 10:07 AM	Text Document	
eula.1049.txt	4/11/2008 10:07 AM	Text Document	
eula.2052.txt	4/11/2008 10:07 AM	Text Document	
eula.3082.txt	4/11/2008 10:07 AM	Text Document	
globdata.ini	4/11/2008 10:07 AM	Configuration sett...	

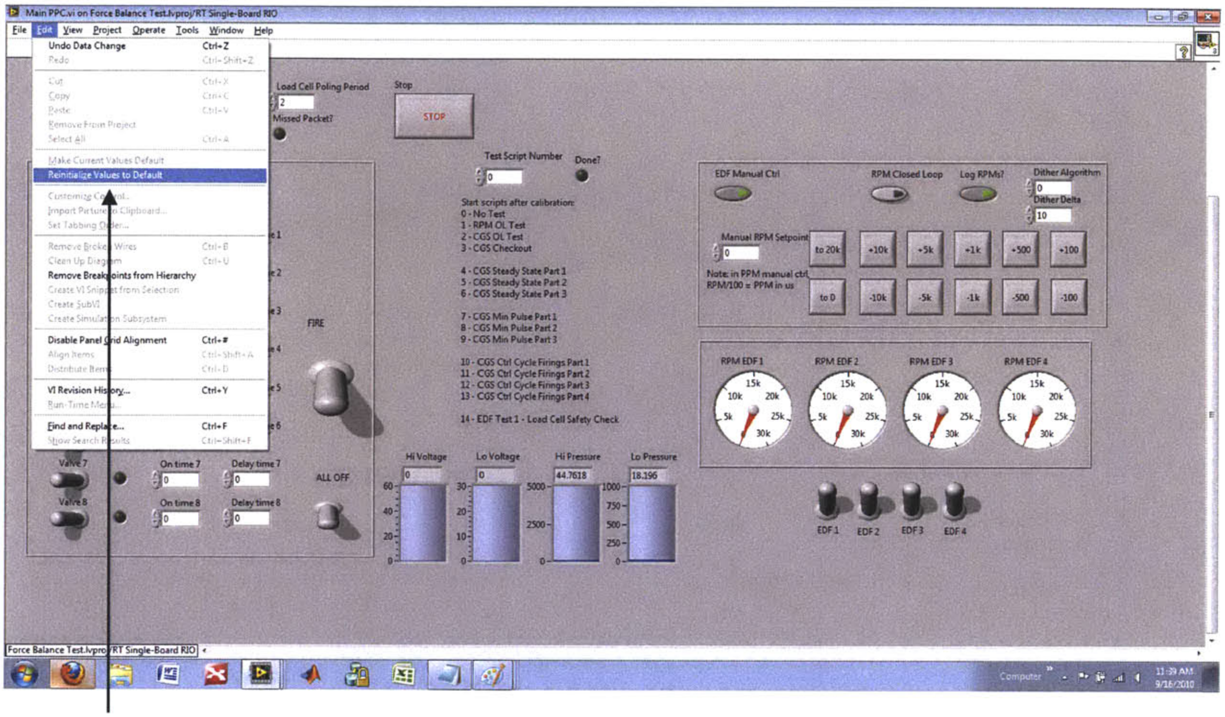
3 items selected Date modified: 9/16/2010 11:37 AM Date created: 9/16/2010 11:36 AM
Size: 2.43 MB

A.3.3 Resetting Software

1) On both MainPPC.vi and Flight Shell.vi, go to Edit -> Reinitialize values to default. All switches and outputs should return to their normal settings.



Note: When running on a Windows 7 machine, make sure LabView is running as administrator, or the log files will be unhappy.



Load Cell Poling Period 2

Test Script Number 0

Start scripts after calibration:
0 - No Test
1 - RPM OL Test
2 - CGS OL Test
3 - CGS Checkout

4 - CGS Steady State Part 1
5 - CGS Steady State Part 2
6 - CGS Steady State Part 3

7 - CGS Min Pulse Part 1
8 - CGS Min Pulse Part 2
9 - CGS Min Pulse Part 3

10 - CGS Ctrl Cycle Firings Part 1
11 - CGS Ctrl Cycle Firings Part 2
12 - CGS Ctrl Cycle Firings Part 3
13 - CGS Ctrl Cycle Firings Part 4

14 - EDF Test 1 - Load Cell Safety Check

On time 7 0

Delay time 7 0

On time 8 0

Delay time 8 0

Hi Voltage 0

Lo Voltage 0

Hi Pressure 44.7618

Lo Pressure 18.196

Dither Algorithm 0

Dither Delta 10

Manual RPM Setpoint 0

Note: in PPM manual ctrl, RPM/100 = PPM in us

Reinitialize on BOTH flight shell and MainPPC

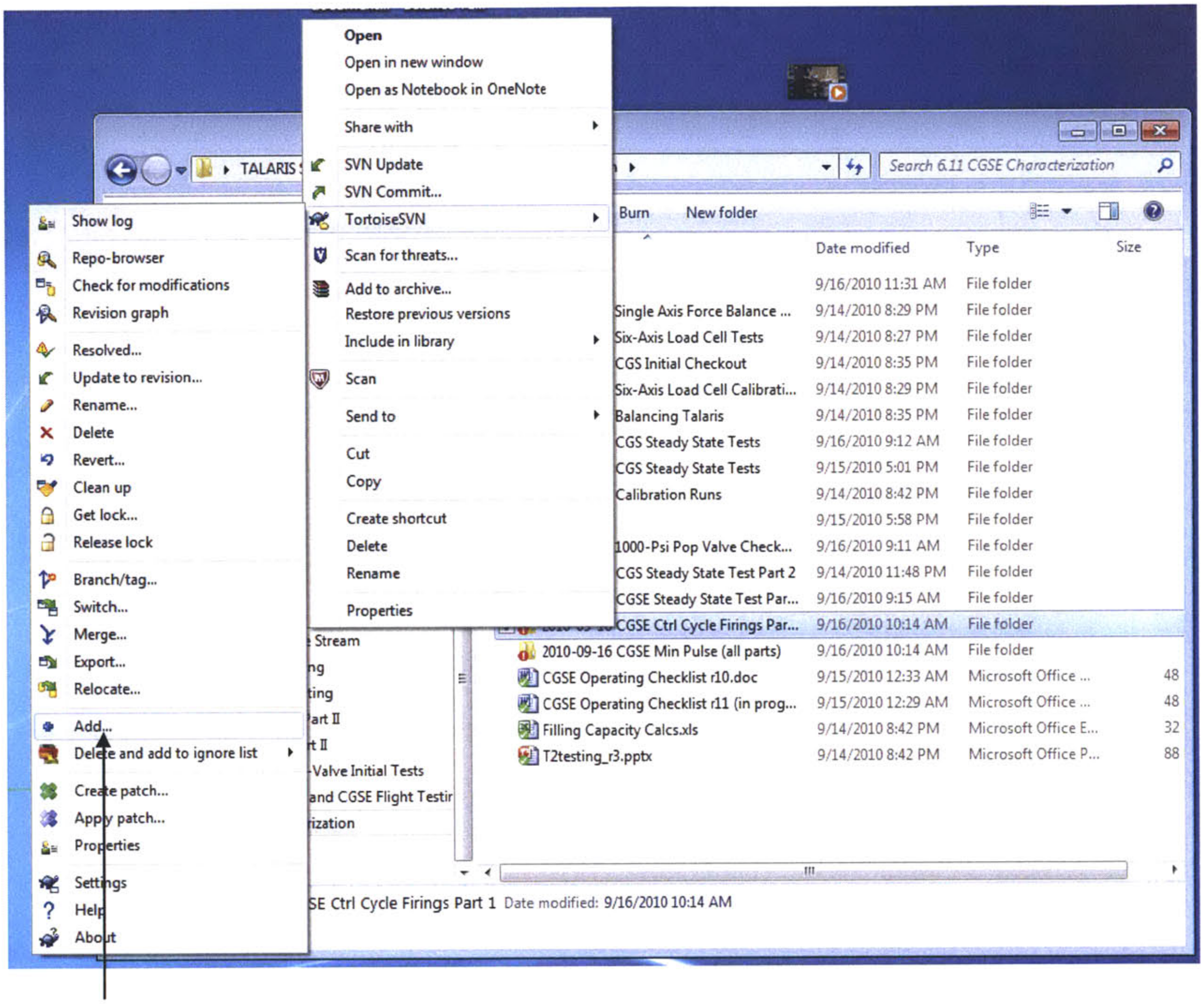
A.4 Uploading Data to the SVN

1) Open the svn folder on the desktop. Navigate to the appropriate folder, which is probably in 6-Testing.

2) Make a new folder for today's date if there is not already one.

3) Copy the data files to this folder.

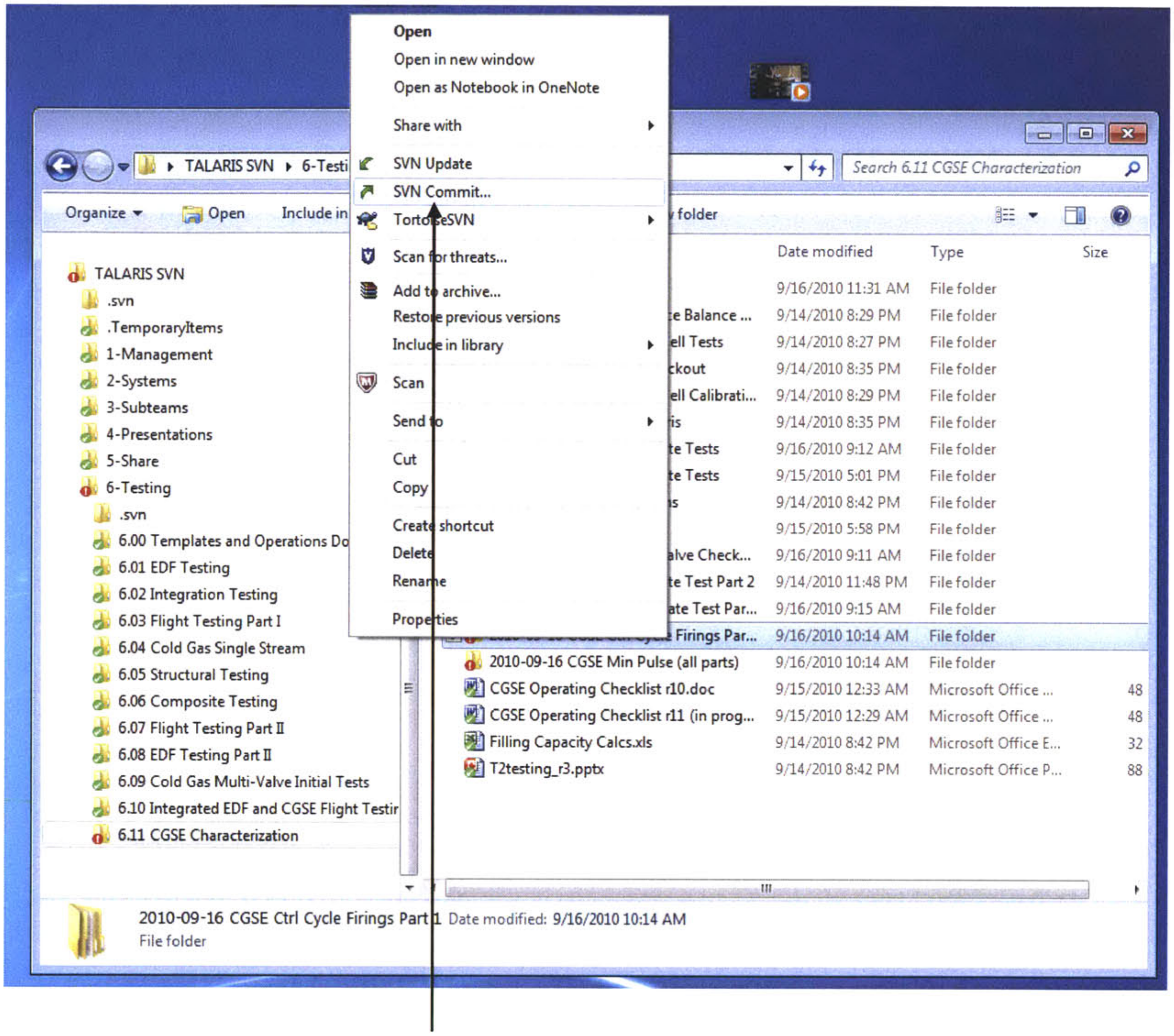
4) Right-click on the folder you just made. Go to TortoiseSVN -> Add. A blue plus should appear next to the folder.



	Date modified	Type	Size
	9/16/2010 11:31 AM	File folder	
Single Axis Force Balance ...	9/14/2010 8:29 PM	File folder	
Six-Axis Load Cell Tests	9/14/2010 8:27 PM	File folder	
CGS Initial Checkout	9/14/2010 8:35 PM	File folder	
Six-Axis Load Cell Calibrati...	9/14/2010 8:29 PM	File folder	
Balancing Talaris	9/14/2010 8:35 PM	File folder	
CGS Steady State Tests	9/16/2010 9:12 AM	File folder	
CGS Steady State Tests	9/15/2010 5:01 PM	File folder	
Calibration Runs	9/14/2010 8:42 PM	File folder	
	9/15/2010 5:58 PM	File folder	
1000-Psi Pop Valve Check...	9/16/2010 9:11 AM	File folder	
CGS Steady State Test Part 2	9/14/2010 11:48 PM	File folder	
CGSE Steady State Test Par...	9/16/2010 9:15 AM	File folder	
CGSE Ctrl Cycle Firings Par...	9/16/2010 10:14 AM	File folder	
2010-09-16 CGSE Min Pulse (all parts)	9/16/2010 10:14 AM	File folder	
CGSE Operating Checklist r10.doc	9/15/2010 12:33 AM	Microsoft Office ...	48
CGSE Operating Checklist r11 (in prog...	9/15/2010 12:29 AM	Microsoft Office ...	48
Filling Capacity Calcs.xls	9/14/2010 8:42 PM	Microsoft Office E...	32
T2testing_r3.pptx	9/14/2010 8:42 PM	Microsoft Office P...	88

“Add” is here.

5) Right-click the folder again, hit SVN commit. Type any comments you have, and hit commit.



	Date modified	Type	Size
	9/16/2010 11:31 AM	File folder	
e Balance ...	9/14/2010 8:29 PM	File folder	
ell Tests	9/14/2010 8:27 PM	File folder	
ckout	9/14/2010 8:35 PM	File folder	
ell Calibrati...	9/14/2010 8:29 PM	File folder	
is	9/14/2010 8:35 PM	File folder	
e Tests	9/16/2010 9:12 AM	File folder	
e Tests	9/15/2010 5:01 PM	File folder	
s	9/14/2010 8:42 PM	File folder	
	9/15/2010 5:58 PM	File folder	
alve Check...	9/16/2010 9:11 AM	File folder	
e Test Part 2	9/14/2010 11:48 PM	File folder	
ate Test Par...	9/16/2010 9:15 AM	File folder	
Firings Par...	9/16/2010 10:14 AM	File folder	
2010-09-16 CGSE Min Pulse (all parts)	9/16/2010 10:14 AM	File folder	
CGSE Operating Checklist r10.doc	9/15/2010 12:33 AM	Microsoft Office ...	48
CGSE Operating Checklist r11 (in prog...	9/15/2010 12:29 AM	Microsoft Office ...	48
Filling Capacity Calcs.xls	9/14/2010 8:42 PM	Microsoft Office E...	32
T2testing_r3.pptx	9/14/2010 8:42 PM	Microsoft Office P...	88

2010-09-16 CGSE Ctrl Cycle Firings Part 1 Date modified: 9/16/2010 10:14 AM
File folder

"SVN Commit" is here.

6) If done correctly, a green check should appear in the folder icon.

7) When uploading video, open the video in Windows Live Movie Maker, edit to include only the relevant test, and save as .wmv or .avi. Unless an abnormal event occurred during the test, standard definition is preferred to keep file sizes down.

8) Rename the video to the corresponding test, which should agree with the log numbers.

A.5 Troubleshooting

A.5.1 Software Troubleshooting

If you can't connect to the RIO

1) The RIO takes about 10-15s to boot up. Wait a little bit and try again.

2) Make sure the RIO is on and the Ethernet is plugged into BOTH the RIO and computer

3) Make sure the VPN client is not connected. The VPN uses the same port as the Ethernet, so both cannot be on at once. Unfortunately, this means that at Draper, you cannot have Matlab up while connected to the RIO.

4) When all else fails, restart computer and try again.

If software isn't running correctly:

1) Try re-deploying MainPPC. Any time you power down or disconnect from the RIO, MainPPC must be re-deployed, even if it was deployed and working previously.

2) Something might have been changed in the code. Download the latest version from the svn by right-clicking on the software folder on the desktop, and hit "SVN Update" and/or "SVN revert."

If a read/write error pops up right when software starts running:

1) LabView is probably not running as administrator. Exit out of LabView completely and start again from step 1 in the "Quickstart Guide."

A.5.2 SVN Troubleshooting

When the local SVN on the flight computer is synced with the central SVN, there will be a green check on the folder icon. If there is a red !, that means there have been some changes made to the local SVN since the last update. It's a good idea to periodically update the local SVN to make sure that both versions are synced.

To sync the SVN:

1) Identify the lowest level folder that has a red !.

2) If these are changes that you made and want to upload to the central SVN, right-click on the folder and hit "SVN Commit."

3) If you would like to get the latest version from the central SVN, right-click and hit "SVN Update." If there is still a red ! after updating, hit "SVN Revert." Be VERY CAREFUL with this command; it will delete any local versions of the files, and they will not be recoverable. Make sure you are ok overwriting the local version before reverting.

Sometimes the SVN locks, corrupts, or throws an error that prevents proper committing or updating. When the SVN is having issues:

1) First COPY the data someplace else. The SVN sometimes gets hungry and has eaten files on more than one occasion.

2) Hit "SVN Cleanup." If this completes successfully, try another commit/update.

3) If this doesn't work, isolate troublesome folders and move them somewhere else (like the desktop).

4) Right-click the parent folder and hit "SVN Commit." Make sure the "missing" folders are checked. This deletes the troublemakers from the central SVN.

5) Hit “SVN Update” then “SVN Revert.” Both should complete without having changed anything. This confirms that the SVN is synced with all and only the files on the local computer. Double check there is no missing data.

6) Go into the troublemaker folders and delete all .svn folders inside the troublemaker folders, even in subfolders. They will be the transparent folders at the top of the list. This will make them "clean" of SVN versioning. If you've done this right, there should be no SVN icon on the folder (the little green check or red ! should not be there).

7) Copy the now cleaned and gutted folders back to the desired folder: try another add->commit cycle

8) If there are still problems, then repeat cleanup step or get a more senior person to take a look.

A.5.3 Other Notes

1) The “Stop” button is equivalent to an e-stop. Pressing this button will immediately close all valves, set the EDFs to zero, and prevent any additional commands from being sent to the vehicle. This can be done at any time.

2) Script files must complete before MainPPC can be stopped. The Stop button will still shut down all actuators and commands at any time, but MainPPC won’t stop and be resettable until the script is done. Operationally, this is not a critical issue, but good to be aware of.

3) Refer to the logging vi's (found in the project explorer) to determine format of logfiles.

4) Refer to test scripts (found in project explorer) to see/modify test scripts.

5) Be extremely careful after a software crash. There is a small possibility that variables will retain their values from the last test, and in the worst case, a valve will click open at an unintended time. Clear all personnel from flight zone, run software, and stop normally to ensure all valves and EDFs are properly shut and reset.

6) Five data files will be generated each time the software is run. Please make a note of which test each data file refers to and upload to the svn with the original data files.

Always keep backups of data or files to be uploaded to the svn, in case there is an error.

7) Before performing a test with load data, it is customary to take one minute of standstill data to determine the load cell bias values.

8) Before powering down the RIO, Disconnect by right-clicking and hitting "Disconnect" in the project explorer. Not a big deal, but LabView will throw an error.

9) Please only upload relevant test video. Videos of the fill process or routine checks can be deleted. If possible, clip the video to include only the relevant test.