3-3-2008

Development of an FPGA Based Autopilot Hardware Platform for Research and Development of Autonomous Systems

Wendy Alvis
University of South Florida

Development of an FPGA Based Autopilot Hardware Platform for Research and

Development of Autonomous Systems

by

Wendy Alvis

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Electrical Engineering
College of Engineering
University of South Florida

Co-Major Professor: Wilfrido Moreno, Ph.D.
Co-Major Professor: Kimon Valavanis, Ph.D.
James T. Leffew, Ph.D.
Paris Wiley, Ph.D.
Richard Wallace, Ph.D.
MaryAnne Fields, Ph.D.

Date of Approval:
March 3, 2008

Keywords: Field Programmable Gate Array, unmanned systems, embedded systems,
analog design, UGV

DEDICATION

To God for giving me the strength to get through many sleepless nights, the stubborn nature that kept me from ever giving up on my goals and the gift of surrounding me with such wonderful friends and family who encouraged me along the way.

To the light of my life, my daughter Danielle, who so generously gave up time with her mother in order for me to realize my dream. Her friendship and love is the greatest gift in my life.

To my husband, my one and only true love and my sole-mate Jim, for putting his goals on hold in order to support mine. I could not have completed this dream without his patience, emotional support and financial sacrifices.

To my parents, Jacqueline and Harry Trietley, for always being there to support and help me over the years.

To my grandmother, Marjorie Bechtold, for all her prayers and unconditional love; this gave me the self confidence to succeed.

To my sister and her family, Lisa, Andy and Heather Patterson, their words of friendship and encouragement were always there to bolster me.

ACKNOWLEDGEMENTS

I thank all the caring and supportive professors that I have had the privilege of working with during my time at the University of South Florida. In particular:

Dr. Leffew; who never turns away a student in need of help,

Dr. Moreno; for all the years as my advisor, encouraging words along the way and late nights revising work completed at the last minute,

Dr. Valavanis; for introducing me to robotics and his support and advice while working on my Ph.D.

I thank the staff of the Army Research Lab for the wonderful summer interning in Maryland, the financial support through a fellowship and their assistance while working on the autopilot. In particular: Dr Wilkerson for making everything possible and Dr Fields for going out of her way to be available for advice and trips to Florida.

I thank my closest friends, Kim Piper, Kim Skinner, Kathy Brown and Shashikala Murthy for their words of encouragement and understanding. An additional thank-you to Shashi for all the hours spent working with me and her excellent work with System Generator.

I thank Xilinx, for their generous contribution of software, Ron, of Advanced Circuits, for the tedious task of assembling the autopilot board and J H Technology, for the use of their equipment and electrical components.

This research was supported in part by an appointment to the Student Research Participation Program at the U.S. Army Research Laboratory administered by the Oak Ridge Institute for Science and Education through an interagency agreement between the U.S. Department of Energy and the US ARL. This work was also partially supported by grant ARO W911NF-06-1-0069 and grant SPAWAR N00039-06-C-0062.

TABLE OF CONTENTS

ii

iii

iv

LIST OF TABLES

LIST OF FIGURES

viii

ix

xi

DEVELOPMENT OF AN FPGA BASED HARDWARE PLATFORM FOR RESEARCH AND DEVELOPMENT OF AUTONOMOUS SYSTEMS

WENDY ALVIS

ABSTRACT

Unmanned vehicles, both ground and aerial, have become prevalent in recent years. The research community has different needs than the industrial community when designing a finalized unmanned system since the vehicle, the sensors and the control design are dynamic and change frequently as new ideas are developed and implemented.

Current autopilot hardware, which is available as on-the-market products and proposed in research, is sufficient for unmanned systems design. However, this equipment falls short of being able to accommodate the needs of those in the research community who must be able to quickly implement new ideas on a flexible platform.

The contribution of this research is the realization of a hardware platform, which provides for rapid implementation of newly developed theory. Rapid implementation is gained by providing for software development from within the *Simulink* environment and utilizing previously unrealized flexibility in sensor selection. In addition to the development of the hardware platform, research was performed within *Simulink*'s System Generator environment in order to complement the hardware. The software produced consists of a user template that integrates to the selected hardware. The template creates a user friendly environment, which provides the end user the capability to develop

software algorithms from within the *Simulink* environment. This capability facilitates the final step of full hardware implementation.

The major novelty of the research was the overall FPGA based autopilot design. The approach provided flexibility, functionality and generality. The approach is also suitable for and applicable to the design of multiple platforms. This research yielded a first time approach to the development of an unmanned systems autopilot platform by utilizing:

- Development of programmable voltage level digital Input/Output (I/O), ports,

- Utilization of Field Programmable Analog Arrays (FPAA),

- Hardware capabilities to allow for integration with full computer systems,

- A full Field Programmable Gate Array (FPGA), implementation,

- Full integration of the hardware within *Simulink*'s System Generator Toolbox.

CHAPTER 1

INTRODUCTION

Unmanned vehicles are better for the performance of tasks that are considered "dull", "dirty" and "dangerous" than piloted crafts. There are many potential uses that will provide a benefit to society such as traffic monitoring, search and rescue and monitoring of structures such as dams and bridges. The use of Unmanned Aerial Vehicles (UAVs), in the military dates back to the 1940s when they were used to fly into radioactive clouds to collect samples. As technology progressed Unmanned Aerial Vehicles have evolved into smaller and more efficient aircraft. UAVs have increasingly demonstrated their benefit to the military. Pioneer has flown reconnaissance missions in the Persian Gulf, Kosovo, and Bosnia since the early nineties. More recently additional types of crafts have been developed and have continued to fly these types of missions to the present, [1].

There is a great deal of work taking place in the research community to make improvements in the existing technologies. The wide diversity in unmanned vehicle designs and control as well as diversity in existing autopilots has lead to major compatibility issues among different platforms. The compatibility issues introduce an additional challenge to the research community. The platforms, sensors and control algorithms are dynamic and change frequently as new ideas are developed and implemented.

1

A search was completed for pre-developed hardware that would allow for data acquisition for system identification, testing/implementation of controller design and flexibility of platform and sensor selection. It was apparent that what is currently available requires a considerable knowledge of programming digital processing hardware and embedded control design. In addition, the hardware available only provides for a very limited choice of sensor selection with each of the specific autopilots.

Within the research community, there are two prevalent forms of processing platforms. Digital Signal Processor, (DSP), systems exist such as Mini-ITX and full computing systems such as PC-104. Neither of these implementations fully meets the needs of the unmanned system researcher. The DSP implementation requires knowledge of embedded systems design and lacks parallel processing capabilities. The full computing system requires knowledge in programming real time operating systems in order to meet tight timing requirements. Some research has been performed, which considered the inclusion of Field Programmable Gate Arrays, (FPGAs), for additional flexibility and parallel processing capabilities. Thus far none have included integration with software providing a higher level of abstraction than Very-high-speed integrated circuit Hardware Description Language, (VHDL). In addition, the advantages of hardware-in-the-loop capabilities for design verification have been explored only minimally.

The DSP and FPGA implementations have the benefit of allowing for precise real time control. This is a mandatory requirement with autopilot systems and is very carefully met with this research. However, in order to take advantage of the flexibility

2

and the parallel processing capabilities that are not available with DSP processors this precise timing was realized with a FPGA.

The lack of availability of autopilots meeting the research community requirements was the motivation behind this research. The outcome was a hardware platform, which provides two major capabilities. The platform provides for a commercial off-the-shelf, (COT), language to be utilized for both programming and hardware-in-the-loop simulation. In addition, the platform provides sufficient flexibility to allow a wide variety of sensors to be available for use with the system under study.

When proposing a new autopilot platform, issues such as sensor integration, sensor diagnostics, conventional servo and actuator control, as well as switching among, or modifying control techniques if and when necessary must be taken into consideration. In other words, consideration should be given to implementing different controllers and sensor selection based on different mission profiles and selected robotic platforms. Thus, any proposed design must include an interface module that provides for simulation, validation and verification before actual implementation. By default, such a design should be fully interfaced and integrated with MATLAB/*Simulink*, which provides for a higher level of abstraction for programming and hardware-in-the-loop capabilities.

Considering vehicle payload limitations, power consumption and requirements, cost-effectiveness and available 'space' on the unmanned vehicle are primary. Given the fact that real-time control requires very strict and fixed timing for stability purposes, the embedded system approach is preferred in designing an autopilot. This approach can be implemented in a much lighter package, which makes it suitable even for miniature vehicles.

3

Swarm formation and mission planning algorithms have been successfully designed on standard computing systems such as the Mini-ITX or PC-104. However, without an additional autopilot, the programmer must have an extensive knowledge of real-time operating system programming to ensure the signal processing and control system meets the timing requirements of the vehicle dynamics. The hardware capability for full integration with these previously developed systems was designed into the autopilot. When in use with these systems the autopilot can be programmed to become the "slave" to the "master" computer and follow specified trajectory commands. This capability provides researchers familiar with software implementations such as C-programming running on Linux to continue with their work unimpeded by the difficulty of implementing real-time programming.

The final area of concern is the protection of the hardware and any surrounding objects or humans. Hardware failure can have catastrophic effects, especially when such failures are associated with aerial vehicles. A loss of control with an unmanned helicopter can very easily cause serious injury or even loss of life. Many systems already allow for emergency takeover by a human pilot. However, this design can be taken even further when used with an external computing system. Providing the end user the ability to design fault detection and emergency control algorithms from within an external computer provides the system with another form of redundancy. In order to achieve this form of redundancy an additional safety switch circuit was designed into the autopilot platform. The safety switch circuit provides for emergency takeover by either a human pilot or a secondary daughter board. The secondary daughter board can be designed to

4

communicate with a computer and take control of the actuators under autopilot failure conditions.

The developed autopilot hardware platform complements the full computing systems by providing a separate processing system that handles the sensor signals and actuator outputs. In addition, it has the ability to be used as a standalone platform for very small scale vehicles. Some systems have been designed with flexibility and ease of implementation in mind. However, this research resulted in an improvement over what has been previously proposed or developed by allowing for full integration with *Simulink*. The integration with *Simulink* provides for a higher level of programming abstraction, hardware-in-the-loop capabilities and full FPGA implementation. These capabilities maximize parallel processing capabilities, analog signal conditioning, which can be predefined and initiated through digital communications from the FPGA processor. In addition, they provide an additional layer of safety by providing for control of the actuators by either a pilot utilizing a handheld radio or a daughter board.

The contribution of this research is a flexible, hardware-in-the-loop capable platform that benefits the area of unmanned systems design by providing for the rapid prototyping of new theory. Therefore, a reduction in the time it takes the benefits to become applicable is realized in both the private and military sectors. The improvements over previous work have resulted from the novelty of utilizing a full Field Programmable Gate Array, (FPGA), implementation, which provides full integration with *Simulink*'s System Generator Toolbox. Surrounding analog circuitry was developed to provide a more flexible interface than realized by previous work. The flexible interface was realized through the development of programmable digital ports along with utilization of

5

Field Programmable Analog Arrays for different analog inputs. In addition, software was produced to provide for a *Simulink* template, which integrates with the autopilot hardware. The software provides a user friendly environment, which provides the end user to more easily integrate the completed algorithms with the sensor and actuator hardware.

The developed autopilot platform was tested utilizing an RC-Truck like robot. Existing software for simple way point following of a robot built for a Traxxis RC-Truck was implemented on the autopilot. The autopilot was integrated to the servo controllers of a MicroStrain IMU and a Superstar II GPS unit. The RC-Truck was able to successfully follow way points, which demonstrated the effectiveness of the hardware design.

CHAPTER 2

RELATED WORK

There exist several UAV/VTOL autopilot hardware platforms, which are sold as fully developed systems. These systems have worked well for those in the private sector. However, the research community has still felt the necessity to develop their own processing systems. Some were developed as a portion of the overall research and others were the subject of the research itself. Each of the, on-the-market, autopilots will be discussed in the context of flexibility, methods of programming, hardware-in-the-loop capabilities and inclusion of parallel processing capabilities. The research based processing systems will be discussed as a generality of the different hardware types in Section 1.1. A more detailed discussion will be presented of the hardware platforms developed specifically as the subject of the research in Section 2.2. An overview and comparison of the types of platforms is presented in Section 2.3.

2.1 Commercial Autopilots

There are several autopilots on the market. Most of these autopilots have not taken into consideration all of the needs of the research community. These autopilots can be separated into several categories. Autopilots, which are proprietary and lack user design capabilities. Autopilots, which are very basic in processing power and possess limited capabilities. Autopilots, which do have flexibility in reprogramming but do not have all the capabilities of the design presented in this dissertation.

7

Two autopilots, which were designed for use with specific vehicles sold by the company marketing the entire system, are available. The Generation II by BAI only provides for minor modification, [2, 3]. The Rotomotion device provides for no modifications at all, [4]. Only minor details are provided about these designs due to the proprietary nature of the entire system. Neither is suitable as a platform for research due to the built in dependency on the company for airframe specific modifications to the software.

The Kestral by Procerus, [5], and the MP2028 by Micropilot, [3, 6], include user flexibility designed into the system. Unfortunately, both of these devices are still proprietary in nature. There are additional input/output ports included in the system software sold for reprogramming and hardware-in-the-loop capabilities. However, both designs only provide reprogramming and hardware-in-the-loop capabilities through proprietary software, which prevents use with *Simulink* and limits flexibility. Details on the Kestral are given in Table 7 and the MP2028 details are given in Table 8. Both tables are presented in Appendix A.

The Ezi-Nav, by Autonomous Unmanned Air Vehicles, was designed to be a low cost autopilot with minimal capabilities, [3, 7]. It contains eight separate microprocessors that share the computational load. The Ezi-Nav operates solely with handheld GPS units and possesses no additional ports for communications with an external processor or additional sensors. More details for Ez-Nav are provided in Table 9, Appendix A. While the Ezi-Nav has demonstrated successful flights with fixed wing vehicles, it does not have the flexibility or processing capabilities required by the research community.

8

The Phoenix, by O-Navi, is an open source, fully reprogrammable autopilot, which utilizes a 32MHz, 32-bit Motorola processor, [8, 9]. Phoenix is programmable through a provided flash kit. However, it still lacks the ability to interact with *Simulink* and there is no hardware-in-the-loop capabilities provided for in the design. Details are presented in Table 10, Appendix A.

The Piccolo II, by Cloudcap, is an open source autopilot designed specifically for fixed wing vehicles, [3, 10]. Piccolo II possesses sufficient flexibility that implementation with rotary wing vehicles appears reasonable. Piccolo II is popular with the research community. The popularity is, most likely, due to its flexibility and ability to be programmed through *Simulink*'s Real Time Workshop. In addition, it does allow for hardware-in-the-loop implementation with *Simulink* models. However, the computer running *Simulink* must be equipped with a CAN interface card. Piccolo II comes close to meeting the research community's requirements. However, it lacks the parallel processing capabilities and flexibility of a full FPGA design. Details of the Piccolo II are given in Table 11, Appendix A.

Of all the autopilots on the market, the Microbot, by Microbotics, possesses the most flexibility designed into the system, [11]. It is the only open source design on the market that includes an FPGA to provide for reconfiguration of up to 32 I/O ports. In addition, an expansion board provides for two asynchronous serial ports and twelve analog inputs to be included in the design. Unfortunately, the FPGA is only utilized for the input and output logic. Most of the autopilot's programming resides in a single microprocessor, which does not allow any parallel processing of the main functions. Another major disadvantage of Microbot is its lack of a design capability for rapid

9

prototyping. While the unit is fully reprogrammable, it does not provide for programming through *Simulink*. Additionally, Microbot does not possess any hardware-in-the-loop capabilities designed into the system. More details are provided in Table 12, Appendix A.

All of the autopilots, except the Microbot, are limited by a lack of parallel processing, which is afforded by a FPGA implementation. In addition, none provide analog input flexibility or have hardware-in-the-loop capabilities with *Simulink* specifically incorporated into the design. The Microbot design, with the FPGA being utilized for sensor sampling and data/servo output, does remove some of the computational load from the microprocessor. The Microbot design also provides considerable flexibility across platforms and sensors. While this design is superior to the others with respect to flexibility, it falls short in simple programming and hardware-in-the-loop capabilities.

2.2   Related State of the Art Research

The majority of publications studied discussed the processing system as a brief portion of a larger research project. In these papers two popular methods dominated. One involved implementing a low power DSP/microprocessor chip such as a Mini-ITX board. The other involved implementing a full motherboard type system such as the PC-104 system. The microprocessor and low power DSP chip possess minimal processing power. Both chips are used primarily for either one specific system, which does not require complex calculations, or a micro-air vehicle that has minimal payload capacity. Only the most recent publications have begun to consider the advantages of a FPGA's parallel processing and reconfigurable capabilities. Section 2.2.1 will discuss low power

10

processor implementations. Section 2.2.2 will discuss the full motherboard implementations. Section 2.2.3 will cover what has been accomplished or has been proposed for full FPGA and hybrid FPGA/DSP implementations.

2.2.1 Microprocessor/DSP Low Power Autopilots

Jung et. al., designed a simplified autopilot for use with a specific fixed wing aircraft, the Goldberg Decathlon ARF, [12]. The design was performed as a learning lab tool for undergraduate students at Georgia Tech. A Rabbit 3000 microprocessor was used along with several sensors. This microprocessor meets the requirements for easy implementation of simple algorithms, which are used for teaching basic control theory. However, the Rabbit 3000 does not provide flexibility for use across platforms. A similar design was developed by Brigham Young University with a fixed wing aircraft fabricated in foam, [13]. As with the system developed by Georgia Tech, the autopilot was small, easy to implement and did function properly. However, the autopilot suffered from inflexibility across platforms. In addition, neither designs provided hardware-in-the-loop capabilities.

Kahn and Kellogg designed an autopilot system that utilized Microchip's 16F877 microcontroller for a kite style micro-air vehicle, [14]. Since the system possessed low dynamics and utilized a minimal amount of sensors, very little processing power was required. Microchips line of microcontrollers is low cost and easy to program. However, they possess a maximum clock frequency of 20MHz, a buffer for serial communication that is limited to three characters and no hardware-in-the-loop capabilities. Microchips line of products does not meet the requirements specified for the majority of unmanned systems research.

11

Preliminary designs are presented for an MC68HCS12 microcontroller based design, [15]. The design focuses on providing a low cost and easy to modify system. The specific UAV and sensors are not mentioned. However, the processing power and flexibility across platforms will be limited due to the selection of a microcontroller for processing as opposed to an FPGA. In addition, there is no mention of intentions to design, into the system, any hardware-in-the-loop capabilities.

An area of research gaining popularity is the design of micro-air vehicles. The payload capacity for these systems is quite small, which limits the size and power consumption of the selected computing system. The majority of researchers in this field are implementing the algorithms with microcontrollers. The microcontrollers chosen are primarily from Microchip's line of processors, [16-18]. Several publications were studied, which discussed either custom sensor design or vehicle design. However, none had implemented any onboard processing. Other methods were used for control of the vehicle. A ground station was used for control processing, [19]. A ground station was also used for verification of design by simulation, [20-25]. Handheld radio control was investigated, [26, 27]. A tethered system was connected to a DSP board and MATLAB, [28]. As new vehicle and sensor designs are developed and become ready for implementation, a processing platform is required. This requirement further demonstrates the need for a small research oriented autopilot platform.

2.2.2 Full Computer Implementations

The majority of research publications discussing the design of small scale unmanned systems present full motherboard systems without dedicated hardware for the signal processing algorithms and low level controllers. The most popular is the PC-104

board running a real-time operating system such as VxWorks® [29] or QNX [30-32]. Lee, et. al., incorporated a full data acquisition card in the design, [32]. Various other computing systems have been used with real-time operating systems as well, [33, 34].

All of these implementations follow the same basic design principle, external sensors and hardware with a single standard computing system. This method has been proven to work successfully. However, great care must be taken in programming the control system or the precise timing needed for the control of the vehicle dynamics will not be met. This requires a great deal of knowledge in control systems and in the real-time programming language. Each function must be given a priority, which allows those functions with the lowest priorities to be permitted to run only when the highest priorities have completed. For a final implementation, which is designed only once, this method may prove acceptable. However, whenever the control system is modified significantly the entire low level control program changes and the timing issues must be entirely reconsidered. For example, if PID controllers are replaced by H-infinity feedback controllers all timing issues would have to be revisited. This potential software redesign can create longer delays between deriving new theory and implanting it in hardware.

One design did try to solve some of these issues by implementing a two processor system running on RT-Linux, [35]. The software was designed with a layered approach. A main board ran an x86 compatible motherboard for the wireless GPS communications and mission planning. The ATM Mega 163 chip was utilized for real-time flight control processes. This is the same basic concept of using a dedicated autopilot for the low level control system, which further argues the need for the hardware platform presented.

13

2.2.3   Implementations Utilizing FPGAs

FPGAs are very slowly gaining popularity due to the recent advances in increased number of gates and simplification of design by the manufacturer's providing intellectual properties, IPs. The IPs provide pre-developed functions such as complicated mathematical calculations and RAM, which would normally be time consuming to develop utilizing a HDL. Since these advances are fairly recent, there are only a few publications following the same philosophy of utilization of FPGAs, [9, 36-40].

Klenke combined a 40K FPGA with an 8-bit microprocessor for control of a fixed wing aircraft with a GPS unit as the only sensor, [39]. The FPGA array was utilized for the FM aircraft receiver and the servo control. The system worked successfully and proved to be a simple to implement, inexpensive design. However, it does not possess the processing power or flexibility required for research across platforms and sensors.

A proposed FPGA based design to provide a system capable of integrating a propulsion health system with a control system for VTOLs has been presented, [36]. This design recognized the strength of both a FPGA architecture and integration with *Simulink* for programming. However, the proposed design intends to implement the algorithms running inside the FPGA under a real-time operating system, VxWorks®. In order to provide user programmability, the intention is to create an ICD along with third party software for programming. The system will implement a Vibe Card for receiving some sensor data. With some simple front end analog signal conditioning and A/D converters, this card can be eliminated and all of the signal processing can be implemented entirely on the FPGA chip. In addition, the design also includes a Geode

14

DSP processor, which will run a majority of the processes. This aspect of the design ultimately limits the system to sequential processing for the majority of the processes.

A flexible FPGA/DSP based autopilot has been developed by the Georgia Institute of Technology, [37, 38]. The project has been completed and tested on two separate platforms. One system works in conjunction with a "master" computing system on the GTMax. Another system acts as a stand-alone device on the GTSpy. While a flexible hardware-proven design is defined, the full strength of the Xilinx line is not utilized to full advantage. The majority of the processing on-board the Xilinx chip is performed by a soft core DSP running on the MicroC/OS II real-time operating system. As a result of the sequential nature of the operating system, many of the tasks cannot be divided into smaller tasks running in parallel. In addition, the system includes a separate DSP chip to run any high level processing. This configuration prevents the system from being fully integrated with *Simulink* through the use of the System Generator toolbox.

Virginia Commonwealth University has recently demonstrated a successful in flight test of a FPGA based autopilot. [9]. This autopilot utilized a Suzuku V board containing a Xilinx II FPGA chip, 32 M Bytes of SDRAM, 8 M Bytes of flash memory and an Ethernet interface. The FPGA's on-chip PowerPC runs a Linux kernel for implementation of the majority of the processing. The goal of the research was to demonstrate that the software could be developed in commercial off-the-shelf hardware and then ported to any other hardware running the same Linux kernel. Since the focus of the research was not a complete hardware autopilot design, the ability to run processes in parallel, with the exception of the I/O protocol, was not considered. In addition, it did not take advantage of the Virtex II *Simulink* capabilities for programming and hardware-in-

15

the-loop verification. However, the design does demonstrate the capabilities of the FPGA as the processing hardware for an autopilot.

Continuing work on the design of an FPGA based control system for a Micro-Satellite has demonstrated the potential benefits of FPGAs for autopilots, [40]. The Xilinx series of FPGAs is utilized and full parallel processing utilized. While in-flight tests have not yet been demonstrated, lab tests have indicated that good timing and parallel communication with the devices have been obtained.

2.3   Overview of Autopilot Implementations

While standard computing systems have been proven to work successfully, great care must be taken in programming the control system or the real-time requirements will not be met. Whenever the control system is entirely changed, which occurs frequently in research, the entire low level control program changes and the timing issue must be entirely reconsidered. This leads to a longer design time between deriving new theory and implanting it in hardware. A solution to this problem is to include a separate processor. Such an autopilot is presented in this dissertation. The autopilot provides an off-board system that follows a given trajectory while handling the tight timing constraints required for sensor integration and control of the vehicle dynamics.

The majority of the systems presented implement a single processor. The processing power varies depending on the specific chip selected. The single processer design is at a disadvantage when compared to implementing either a full FPGA or a hybrid DSP/FPGA design. Since single processor systems cannot operate with parallel processing, care must be taken to be sure that each of the asynchronous sensor inputs are sampled at the correct time while also updating the servo outputs. In addition, the

16

majority of the implementations do not allow for *Simulink* integration or hardware-in-the-loop verification of the design.

Several of the FPGA implementations have benefited from the flexibility and parallel processing capabilities of the FPGA with regard to managing I/O functions. These designs failed to carry the parallel processing capabilities into the majority of the processes by implementing most of the algorithms within an on-board or external DSP. The only work that has utilized the full parallel capabilities for flight control was for the design of a Micro-Satellite. This work did indicate good timing when utilizing the parallel capabilities of the FPGA implementation. However, it did not explicitly design *Simulink* integration into the system.

This research included the benefit of *Simulink* integration, as with [10], the flexibility of inputs resulting from utilizing an FPGA, as with [9, 11, 36, 37], and produced a significant contribution to the field of unmanned systems by including further capabilities. These capabilities include full FPGA implementation, full integration with *Simulink* for both programming and hardware-in-the-loop, programmable signal conditioning and hardware so the human pilot can easily regain control under failure conditions. An additional layer of safety was also included. When in operation with a daughter board and second processing system, the secondary system is able to take over control of the aircraft servos when a failure in the autopilot has been detected.

CHAPTER 3

AUTOPILOT REQUIREMENTS

There are two primary areas of study, which will utilize the autopilot differently. Specifications for high level mission planning and vision systems differ considerably from those for system level applications. System level research includes the development of control systems for vehicle dynamics, development of methods for filtering and integrating the sensors and development of new micro-air vehicles.

Navigation researchers work directly with the sensor inputs in order to generate minimal noise and maximum accuracy of certain variables such as position, velocity and acceleration. Researchers within the area of controller design are investigating the most promising methods of controlling the dynamics of the vehicle. Both groups require certain measurements to be available and accurate. The researcher developing the control algorithms will require that the underlying autopilot platform provide for completed sensor filtering and integration. This provision ensures that signals have clearly defined variables and can be utilized within the control loop without modification to filtering and integration modules.

Researchers investigating micro-air vehicles are concerned with the development of new platforms requiring custom controller design. In addition, they are very concerned with the development of new smaller size and low power sensors. This group will have the same requirements as the control and navigation researchers. Additionally,

18

they are confronted with requirements of providing for various sensor inputs, which cannot be predetermined, as well as low power and light weight circuitry demands.

Since both the navigation and control researchers will be implementing and testing algorithms, many of the requirements, which simplify the process, pertain to both groups. These capabilities include modularity for separation of algorithms, a high level of abstraction to allow for simplification of design and the ability for hardware-in-the-loop verification of the software algorithms. The majority of researchers working in this area utilize *Simulink*/MATLAB for testing algorithms. Therefore, the ability to program directly from *Simulink* is advantageous. *Simulink* software design is also well suited for high levels of abstraction and the type of modularity required between the navigation system and the vehicle control system. In addition, providing for hardware-in-the-loop simulation directly with *Simulink* models is beneficial for testing designs without the risk of loss of hardware. For these reasons, full integration with *Simulink* is an extremely valuable aspect of the autopilot platform developed during this research.

The researchers working with navigation systems and vehicle design will require that data be collected for system identification of the sensors or the dynamics of the vehicle. This requires that a significant amount of data be stored while the vehicle is in flight. In addition, the hardware must have the capability to store this information at a high sampling rate without interrupting the modules controlling the navigation. This requirement clearly argues for parallel processing capabilities and creates a further requirement of additional memory for data collection.

When working with high level mission planning or vision research, it is necessary to be able to send the way points to the navigation modules and have the vehicle follow

19

the trajectories without consideration for designing around tight timing constraints. In addition, many of the researchers in this area of research prefer to work with systems such as the Mini-ITX DSP processor or the PC-104 type microprocessor. In order to meet the needs of this area of research, interaction with this second processing system must be included in the design. The autopilot must have the ability to send and receive commands through serial RS232 communication. The software controlling the dynamics of the vehicle, which provides for trajectory following, is a platform-specific design. The autopilot platform must provide for rapid development so that once the vehicle is selected, the navigation and dynamic control system can be quickly finalized. This capability provides for timely and efficient development of applications such as vision, swarm formation and mission planning while running on a complete computing system.

When working with aerial vehicles, safety requirements must be given the highest of priorities. Safety requirements demand as much redundancy for actuator control as possible. The autopilot produced during this research was developed to work with an external processor. Therefore, a redundancy of control hardware was already present. However, additional hardware was incorporated to provide for transfer of actuator control to either a human interface or a second processing system.

The list of generalized requirements, which were incorporated in the platform developed during this research includes:

- Integration with MATLAB/*Simulink* for a higher level of abstraction and modularity when programming and the capability for hardware-in-the-loop verification,

20

- Adequate memory for data collection for use with system identification research,

- Analog design to allow for reconfigurable cross platform/sensor capabilities,

- RS232 communications to provide for integration with a second computing system,

- Parallel processing capabilities & hardware level timing control,

- Emergency takeover of servos as an additional layer of safety.

CHAPTER 4

AUTOPILOT ENVIRONMENT

When designing the autopilot, careful consideration was given to both hardware and software capabilities. The hardware was designed to provide for flexibility across platforms and sensors. The software was designed within the *Simulink* environment in order to compliment the hardware. The software provides for both an autopilot hardware implementation template and an open source library. This chapter presents an overview of the autopilot hardware and the autopilot's *Simulink* software environment. In addition, a brief description of the templates, available library subsystems and how to implement them is provided.

4.1   Hardware Overview

The autopilot hardware design included port connections for most standard hardware utilized on small scale unmanned systems. These include analog inputs, Transistor-Transistor Logic (TTL) and Input/Output (I/O), ports. In addition, the autopilot possesses pressure sensors for measuring altitude and forward velocity as well as Pulse Width Modulated (PWM) outputs for controlling standard servos. The three analog inputs have additional flexibility. A Field Programmable Analog Array (FPAA) was incorporated for customized signal conditioning development, which could be programmed into the FPAA from the FPGA. The TTL I/O ports provide for variable voltage settings through the use of a digital trim pot, which is also directly programmable

22

from the FPGA. The autopilot board developed and produced during this research is

pictured in Figure 1.
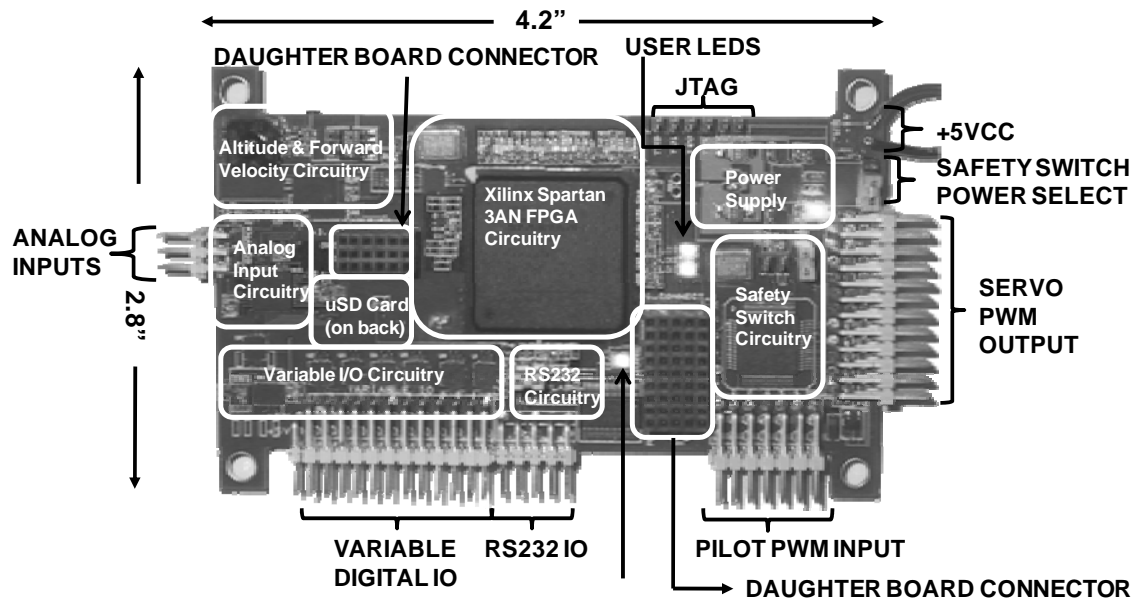
Figure 1: Autopilot Board Overview

In order to both minimize size and provide custom analog and MEMS sensors to

be developed for use with the autopilot, a stacked board design consisting of a main

board and a secondary daughter board was implemented. The daughter board can be

used for inclusion of application-specific hardware such as custom sensors. This is a

necessary requirement for micro-air vehicles since the small payload capacity requires

extremely small on-board sensors to be utilized.

The FPGA outputs PWM logic to control servos through 3.3V TTL ports. The

autopilot's servo connectors do not directly connect to the FPGA. The waveform is sent

to the input ports through a Complex Programmable Logic Device (CPLD), which is used

in the safety switch circuitry. This circuitry provides for connections from a handheld

radio receiver for human pilot takeover. In addition, the circuitry also includes PWM and

23

select lines to the daughter board connectors to provide takeover capability from an external processing system. When a daughter board is not connected, jumpers are used to disable the second takeover option. While the safety switch was programmed for the behavior described, JTAG connectors are available on the back of the autopilot board. This provides for reprogramming of the CPLD in order to gain additional functionality. One additional pin has a direct connection to the FPGA in order to send information. This was provided as a tool to assist the programmer.

In addition to the hardware connectors, three user LEDs, one user switch, a power LED and a programming completed indicator were included on the board. These hardware assets provide indicators to assist the software developer and provide an additional logic input to the board. The hardware specifications for the autopilot developed and produced during this research is presented in Table 1.

Table 1: Autopilot Specifications

I/0 ports and sensors
On-board pressure sensors for altitude and forward speed
Two large signal, single ended analog inputs
One small signal differential analog input
Twenty-Four variable voltage logic inputs
• Input voltage set in blocks of four
• 1.8V to 5V range
Four Tx and 5 Rx RS232 lines
Forty-Six 3.3V I/O FPGA connections to daughter board
Capabilities
On-board MicroSD card for data acquisition memory
A safety switch for servo control
Twelve servo outputs
• All twelve, selectable in sets of six by daughter board, (if present)
• Six critical servos, which can be taken over by a pilot
Simulink programming and hardware-in-the-loop capable

The board contains a Xilinx Spartan3-1400AN FPGA, which serves as the primary processing platform for the autopilot. By selecting from the Xilinx line of FPGAs, the autopilot was fully interfaced and integrated with *Simulink*. The reconfigurable nature of the FPGA provides the programming capabilities, which are necessary to compliment the flexibility of the hardware design. The hardware flexibility incorporates the handling of several types of communication protocol. The hardware accepts various ranges of analog sensor input and data acquisition. The hardware provides for measurement of altitude and forward speed through on-board pressure sensors. In addition, the hardware provides for releasing control of the servos to either a human pilot or a second processing system through the use of a daughter board.

4.2   Autopilot Software Environment

The user of the autopilot will have available, from within the *Simulink* environment, hardware protocol subsystems and the standard System Generator building blocks. In addition to the *Simulink*/System Generator software tools, the Xilinx's EDK environment can be utilized to develop soft core processors capable of running a user-selectable operating system. The overview of the autopilot's software environment is presented in Figure 2.

The available subsystem building blocks were developed specifically for the peripheral hardware contained on the autopilot. Other high level signal processing functions such as filtering, sensor integration and controllers can be developed using standard System Generator blocks. In addition, the soft core processors can contain a small operating system such as the Slackware version of Linux or VxWorks®. While this does seem contradictory to the argument for parallel processing, the ability to utilize

25

a DSP structure provides for the implementation of many algorithms, which have been

developed to operate within a specific software environment, such as a wireless

networking protocol. In addition, hardware may be designed into the system that utilizes

Linux drivers without the additional work of developing custom software.
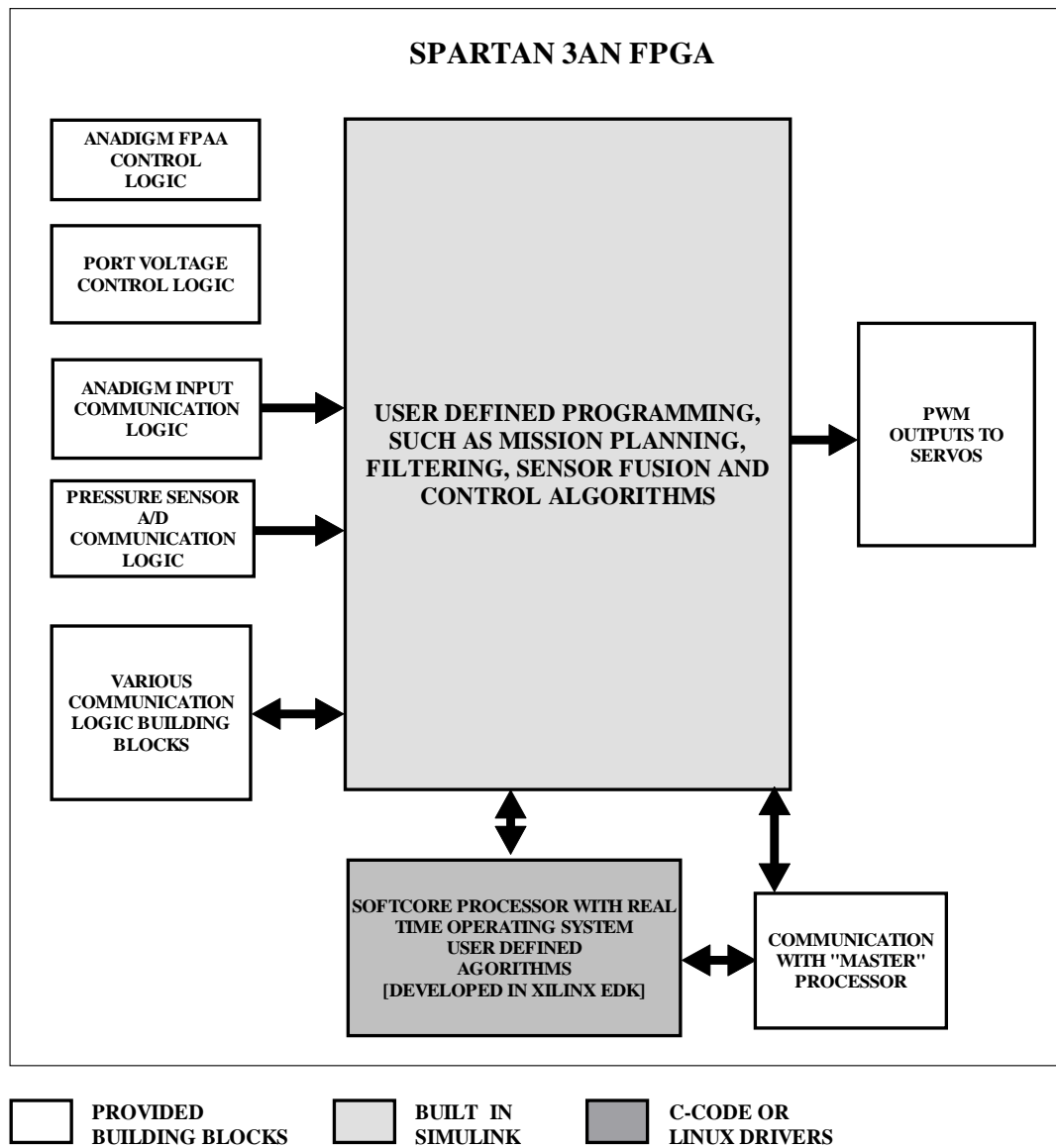
Figure 2: Software Block Diagram

26

4.2.1 Hardware Co-Simulation Timing Issues

System Generator allows for hardware-in-the-loop simulation by compiling a co-simulation block that contains the bit stream for programming the FPGA and controls the JTAG communication. After the hardware co-simulation block is generated, it can be set to single stepping or free running by double-clicking the block. When single stepping is selected, *Simulink* controls the FPGA clock signal and the hardware matches the *Simulink* clock cycle, which does not relate to real-time. This is the preferable setting when communication with external hardware is not required. However, when the autopilot is to be programmed to interact with external hardware, real-time is required. The 'free running' selection will turn control of the clock over to the FPGA's 50MHz clock. Within the system there will be blocks, which must be synchronized by the system clock. When the System Generator blocks are converted to the FPGA hardware configuration bit stream, the resulting internal rates are related to the *Simulink* update rate. The update rate is provided by equation (1). This relationship was utilized within all the developed subsystems specifying hardware level timing.

$$\text{hardware update rate} = \frac{\text{Simulink update rate*hardware clock rate}}{\text{Simulink time step}} \qquad (1)$$

A *Simulink* autopilot template was developed. The template contains masked subsystems to:

- program the FPAA,
- receive data from the A/D outputs from the FPAA,

27

- enable and receive data from the pressure sensors,

- initialize the MicroSD card,

- enable/disable the RS232 ports,

- enable/disable the variable I/O ports,

- set the desired voltage level

- generate the PWM outputs with selectable frequencies and duty cycles.

With the FPAA, the pressure sensors, all the PWM ports in use and the MicroSD card enabled the amount of slices utilized was 1362, or 12%. By disconnecting the outputs, connecting the PWM to logic low and deactivating the FPAA program subsystem, the unused logic is trimmed during hardware generation. Under these conditions the utilized slices are reduced to just 526, or 4%. The autopilot template is presented in Figure 3.
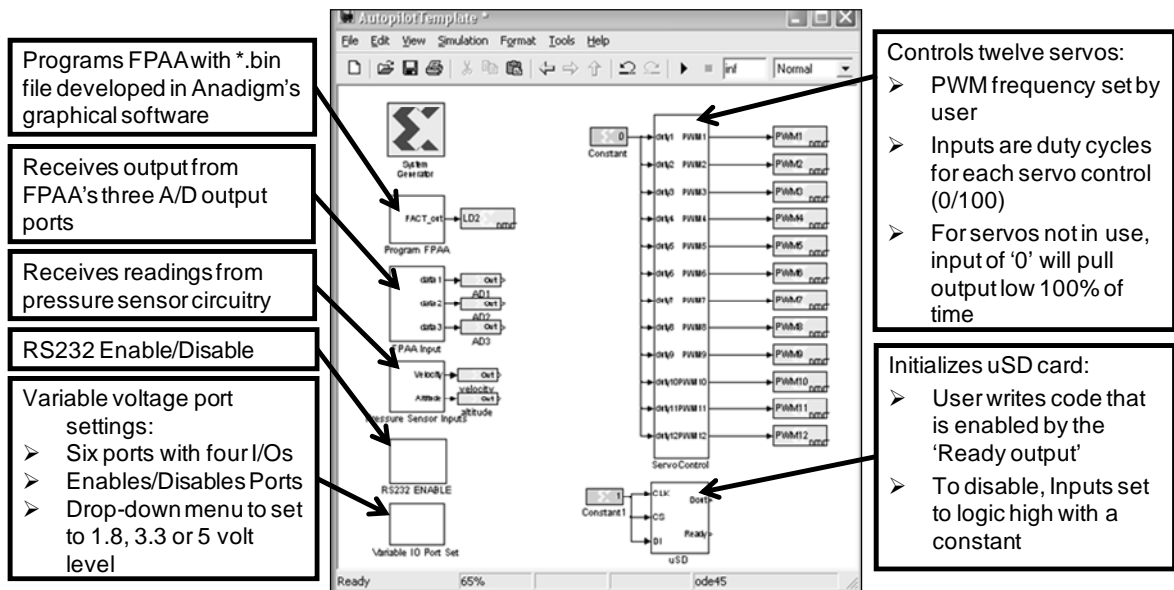
Figure 3: Autopilot Template

28

Several library subsystems were developed to accommodate both basic use of the on-board hardware and the communication requirements for the sensors utilized by the RC-Truck robot. The subsystems developed thus far include:

- an RS232 communication protocol,

- initialization the FPGA RAM for data acquisition,

- a communication protocol to receive latitude and longitude from the Superstar II GPS unit,

- a communication protocol to receive gyro-stabilized Euler angles from the MicroStrain IMU unit.

While the library is limited, the open source platform provides for continually increasing functionality as new software is developed over time by end users.

As a result of the pre-developed hardware level timing a variable declared as *SimP*, which defines the System Generator time step, must be specified by the end user. *SimP* can be defined either in the MATLAB workspace or the model explorer. Once defined, *SimP* is entered into the System Generator block. This provides for the simulation time step to be modified without affecting the final hardware level timing. The only restriction on the time step is that it must be less than 10usec. This restriction results from the communication protocol timing and the resolution of the generated PWM output. The System Generator block is presented in Figure 4.

Each of the template subsystems provided has a user interface, or mask, in order to enable/disable and select specific settings, with exception to A/D protocol of the FPAA. Since the FPAA is disabled through the programming subsystem, the outputs from the receive subsystem are left unconnected when not in use. When *Simulink*

29

generates the programming bit stream all the unconnected logic is removed and the inputs are ignored.
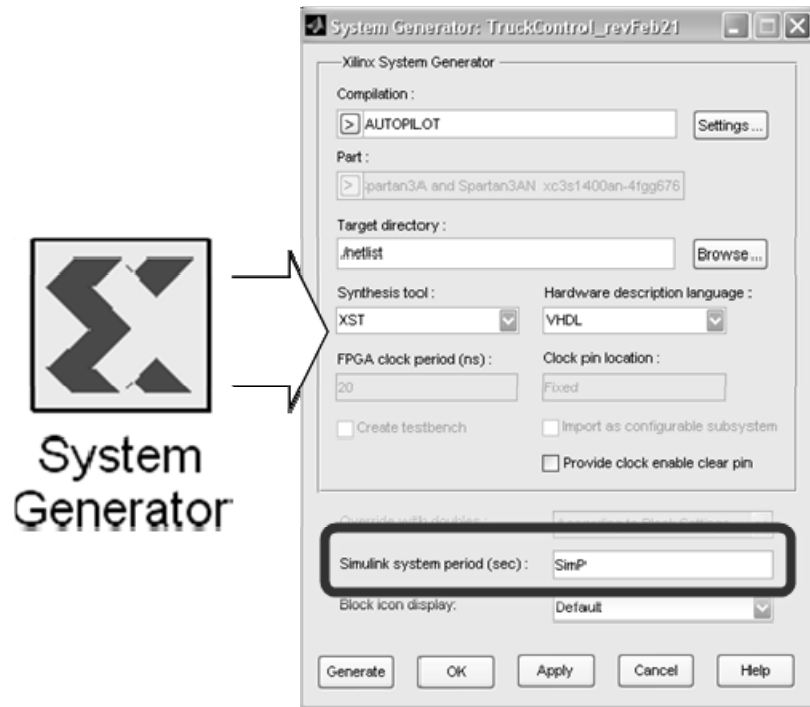
Figure 4: *Simulink* System Period Setting

4.2.2 FPAA Programming and Utilization

The FPAA subsystem program contained in the autopilot template allows the user to enable the system and enter the name of the workspace variable containing the FPAA program bit stream. Since the variable is entered into a *Simulink* block, in the underlying subsystem, the name must not be left empty. When the FPAA is not utilized, a value of '1' should be entered to prevent a *Simulink* error flag. When the 'Enable FPAA' is not selected the subsystem holds the outputs constant, which includes the clock signal to the FPAA. The FPAA program subsystem is presented in Figure 5.

30
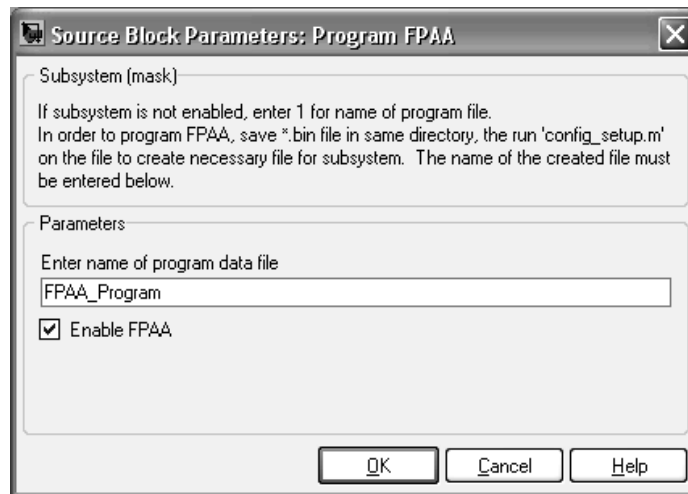
Figure 5: FPAA Program Settings

The FPAA subsystem programs the FPAA from a variable, which must be created within the MATLAB workspace. This is accomplished by first generating a binary program file with Anadigm's AnadigmDesigner2 software. Once the binary file is created an m-file is utilized to read the file into a variable in the workspace. The m-code is used to create the necessary variable, *FPAA*, which is displayed in Figure 6.

```
1 -    fid = fopen('FPAA.bin'); %config6.bin created by AnadigmDesigner2
2 -    FPAA = fread(fid,'ubit8');  % reads file into workspace
3 -    FPAA=dec2bin(FPAA);   % declares MATLAB variable containing binary values
4 -    FPAA=reshape(FPAA',[],1); % reshapes variable to correct format
5 -    FPAA=bin2dec(FPAA); % converts back to decimal so Similink can recognize
6 -    FPAA=[FPAA; 1]; % add last one to stop output port on 'logic high'
```

Figure 6: FPAA Configuration M-File

The binary file is first read into MATLAB and then rearranged from an 8-bit word length to a 1-bit length. In order to reformat the word length, the '1's and '0's are declared as binary. After reshaping, they are re-declared as decimal values. The final step is to add a trailing one that is required to hold the output line high after the last bit is transmitted.

31

4.2.3　Utilizing Pressure Sensors for Altitude and Velocity

Two on-board pressure sensors were incorporated in the design of the autopilot platform. The pressure sensors provide for the measurement of altitude and forward velocity. The pressure sensors produce an analog signal, which is sent to a dual A/D converter. The template subsystem reads the two 16-bit values into the FPGA. The resolution of the calculations for altitude and velocity are user dependent. Therefore, logic was not created to convert altitude and velocity. In addition, the end user may wish to reduce the number of gates by implementing the 16-bit values directly in the controller algorithms. The subsystem contains a mask, which will disable the system by tying all the outputs to the A/D converter to logic high. The subsystem is presented in Figure 7.
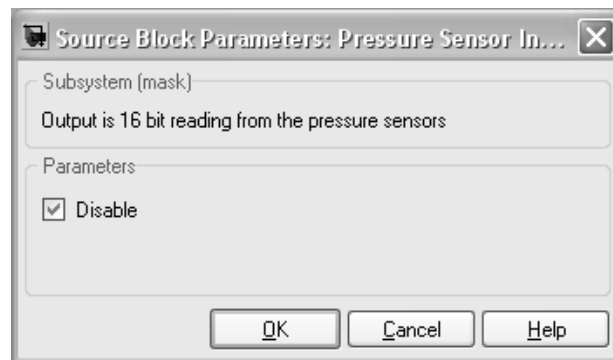
Figure 7: Disabling Pressure Sensor

4.2.4　Initializing the MicroSD Card

Since there are many potential uses for the MicroSD card, the only logic included in the library is the sequence of instructions, which must be sent in order to initialize the card. The hardware ports were incorporated inside the template subsystem and can be accessed through the ports of the subsystem. When the card is not in use the input ports must be connected to a logic high constant. The card will still initialize but it will not

32

receive any further commands. When the card is in use the user must wait for logic high

out of the *Ready* port before sending any commands. The output from the card is

available through the *Dout* port. The MicroSD card is presented in Figure 8.
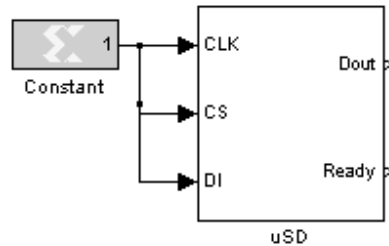
Figure 8: MicroSD Card Template Subsystem

4.2.5   Disabling RS232 Ports

The RS232 ports can be enabled or disabled from within the template. The

mechanism for manipulating RS232 enable is presented in Figure 9.
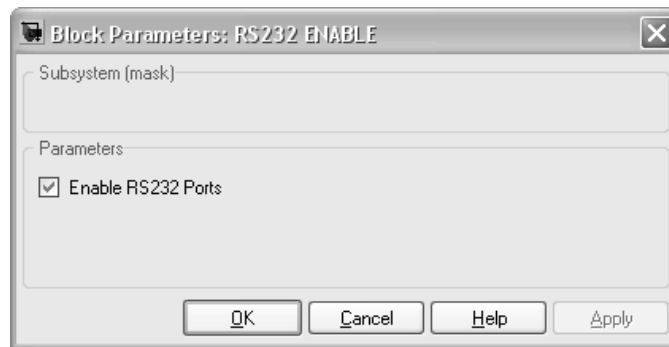
Figure 9: RS232 Enable

When disabled the FPGA output ports enabling the RS232 IC are held at logic

low. This holds the I/O lines out of the autopilot at high impedance. When in use, the

user can utilize the library blocks provided for the communication protocol.

33

4.2.6 Setting Variable Voltage I/O Ports

The template subsystem, which controls the variable I/O port settings, contains an enable and a voltage select available in six sets of four communication lines. The subsystem for I/O control is presented in Figure 10. When disabled, the voltage translator IC holds the autopilot I/O pins at high impedance. When enabled, the user can set the voltage to any of the predefined values of 1.8V, 3.3V or 5V.

Figure 10: Variable I/O Port Settings

4.2.7 Utilizing PWM Output Block

The PWM template subsystem controls the generation of the signals to the 12 output ports. The output signals are generated by converting a duty cycle input, 0 to

34

100%, into the output square wave signal. Specification of a specific frequency between
20Hz and 100Hz, an initial duty cycle and specification of hardware or simulation timing,
is user selectable. When simulation is selected the PWM frequency is matched to the
simulation time steps. When hardware implementation is selected a conversion is
included to set the PWM generated to the hardware clock. If the user sets the input to a
constant of 100 the output lines are all held logic high. The PWM subsystem parameters
are presented in Figure 11.
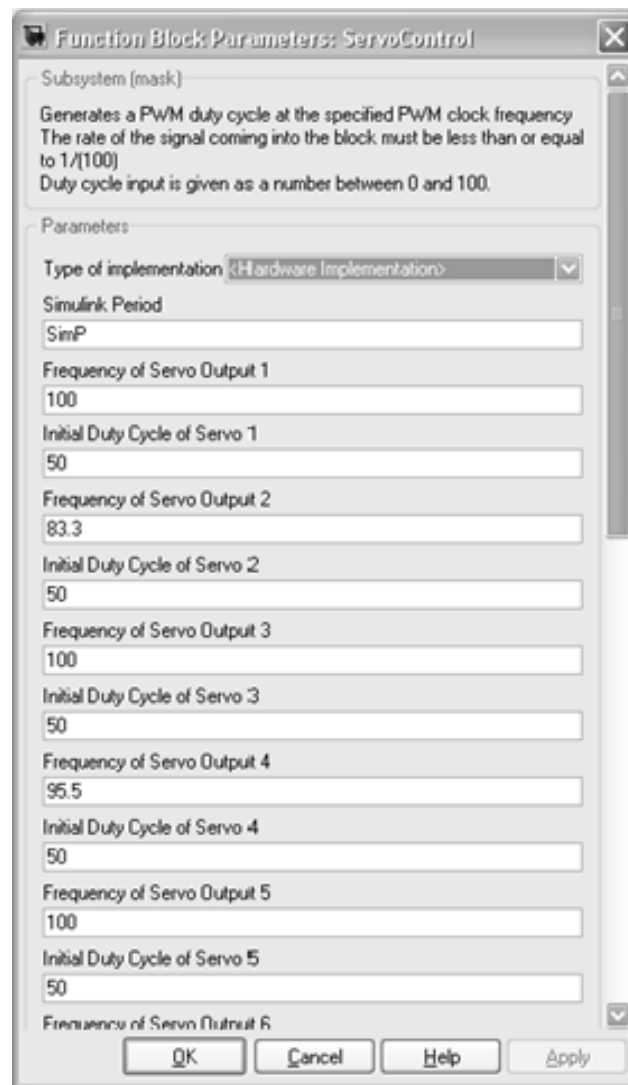
Figure 11: PWM Subsystem Settings

35

4.2.8 RS232 Communication Subsystems

Separate library subsystems were developed for sending and receiving eight bit data with no parity and a stop bit of one. The receive function provide the user a capability to select from a list of baud rates, which includes 9600, 57600, 38400, 56200 and 115200 bps. The send function also provides for the same communication baud rates. However, the rate is set by the inputs to the subsystem.

The receive subsystem over-samples the port at the clock frequency of the autopilot, which is 50MHz. This prevents an incoming byte from being misread due to clock drift or jitter. This works well for receiving data. However, it creates a very fast update rate within the System Generator. In some cases, where the incoming data is followed by only simple logic, this may not pose an issue. In other cases it sets up a timing requirement, which the hardware may not be able to meet. Therefore, a library subsystem was developed, which down-samples the output to the actual baud rate.

The library subsystem that receives the RS232 data from the I/O port has one input and two output ports. The input is the autopilot hardware port, which receives the bit level input and must be set to the clock rate under the mask. Entering the variable *SimP* will make the necessary clock adjustment for a rate of 20ns. This process is presented in Figure 12.

The two outputs from the subsystem consist of the received 8-bit character and a 1-bit flag. The 1-bit flag is held logic high for one clock cycle when a new character has been received. The baud rate is set with the drop-down menu as demonstrated in Figure 13

36
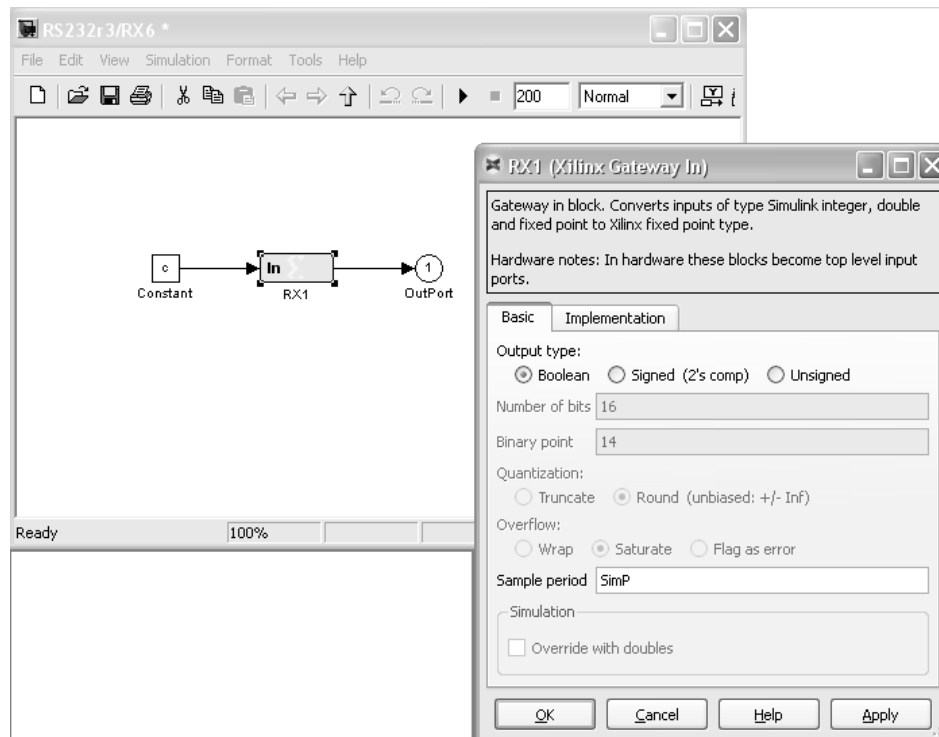
Figure 12: Setting Input Port Timing
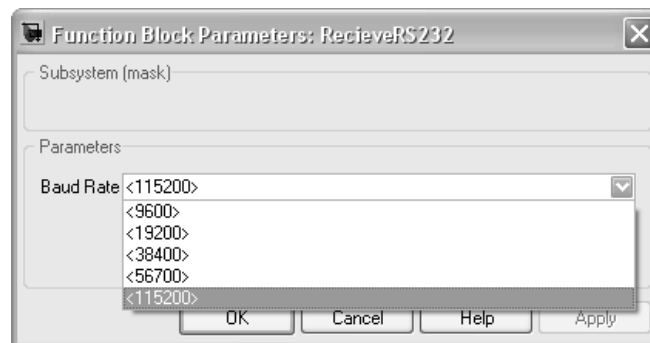
Figure 13: Setting Baud Rate

The down-sample RS232 library subsystem has two inputs and two outputs, which correspond to the outputs of the subsystem receiving the RS232 data. The 8-bit received data and the flag from the receive RS232 block are down-sampled to the selected baud rate and passed out of the function. The output rate is selected by the same style of drop-down menu as the receive subsystem. This process is presented in Figure 14.

37
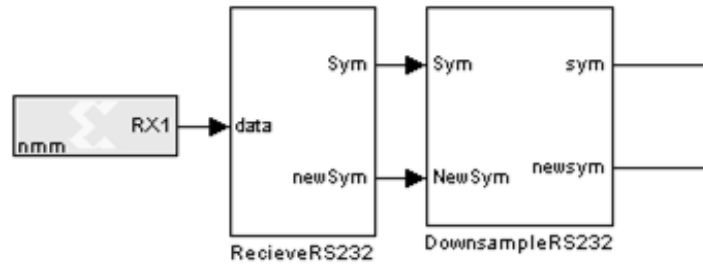
Figure 14: RS232 Down-Sample

The RS232 library subsystem, which sends the 8-bit data, has two inputs and one output port. The first input port, termed *ascii*, holds the 8-bit character to be sent. The second, termed *Out_EN*, holds a logic in, which sends the character when equal to one. The inputs set the rate of the blocks contained within the subsystem and must correspond correctly to the selected baud rate. The output port, termed *BIT*, is connected to the selected hardware port. This includes the RS232 ports and the variable level logic ports, which can be used with TTL to USB converters for receipt of the RS232 protocol. The settings are listed in Table 2.

Table 2: RS232 Send Input Timing

Baud Rate, Bits Per Second	Input Rate, Seconds
1900	$1.042(10^3)$
19,200	$5.208(10^4)$
34,800	$2.604(10^4)$
56,700	$1.736(10^4)$
115,200	$8.68(105)$

4.2.9 FPGA RAM Data Acquisition Library Block

When developing the correct logic design for communicating with external hardware the testing of the protocol must be performed with the co-simulation block set to free running. This setting will insure that the hardware timing is implemented

38

correctly. Since the update of the JTAG port is much slower than any standard

communication rate, this prevents hardware-in-the-loop verification from being utilized.

This library subsystem was developed as a solution to that issue. Values occurring within

the FPGA are stored within RAM memory. When the memory is full the values are sent

to the JTAG port at a rate, which is more acceptable. The rate must be determined by the

end user. This is due to the fact that longer word lengths require more time to receive.

Once the data has been received through the JTAG port, the values can be graphed by

any MATLAB method for analysis.

The library subsystem allows for two inputs to be recorded and also includes an

enable port so that the data can be saved at a specific time. The outputs from the

subsystem are each connected to a JTAG System Generator block, termed Gateway Out.

These outputs are the address, *addr*, and the two recorded strings of data, *data* and *data1*.

These outputs are presented in Figure 15.
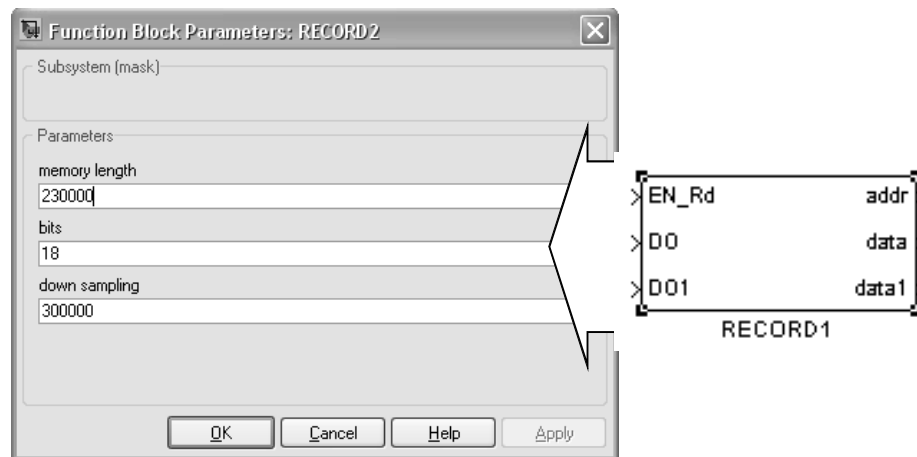
Figure 15: Record Data Library Subsystem and Settings

The available settings are the memory length, the number of bits associated with

the length and the down sampling value. The number of bits must be set to correspond to

39

the word length so the associated RAM is compiled correctly. The down-sampling value is the ratio of the output rate to the input rate.

4.2.10 Superstar II GPS Communication Protocol

The Superstar II GPS has several user selectable settings. The one, which must be selected through the GPS provided Starview software, used to implement this library block is receive LLA in binary format at 1900 baud. This sends information corresponding to the status of the receiver, position and velocity measurements. Not all of the information received from the GPS unit is sent to subsystem output ports. Output ports receive only the values of interest for simple navigation. The navigation data required consists of latitude, longitude, altitude, North velocity, East velocity, vertical velocity and the number of satellites used. The latitude and longitude are in double precision format. The altitude and velocities are in single precision format. The number of satellites exists as a standard 4-bit binary value. The final output is a 1-bit flag, which is held high for one update clock duration, when new measurement information is available. Since the baud rate of the communication block is 1900 bps, the corresponding output from the subsystem has an update rate of 1.042ms, with new measurements available at 5Hz. The only input to the function is the autopilot port selected to receive the GPS output. The RS232 library subsystem is utilized inside the GPS subsystem. Therefore, the port must be set to the hardware clock rate. The GPS unit is one example where the RS232 protocol is used with a TTL logic level. The correct voltage setting is 3.3V and can be set within the variable port setting of the template.

4.2.11 MicroStrain IMU Communication Protocol

The MicroStrain IMU sends information using the RS232 protocol and voltage levels. The 8-bit value is sent in binary, rather than ASCII. The library subsystem waits five seconds for the IMU to initialize. After the IMU initializes the library subsystem requests the IMU to continuously send the gyro-stabilized Euler angles. The 16-bit values are sent eight bits at a time. A checksum value is included for the 16-bit values. The subsystem combines the received 8-bit characters into the 16-bit measurement and calculates the checks sum. If the checksum is correct, the subsystem outputs the 16-bit values. These values include yaw, pitch, roll, ticks and a checksum error flag.

An RS232 subsystem was utilized to establish the communication protocol without the down-sample block. Therefore, the *Simulink* update rate on the subsystem's outputs is 50MHz. The information is sent by the IMU as soon as it is available. The specification sheet guarantees 50Hz. However, for a request of stabilized angles, it tends to be closer to 70Hz. The user may select any of the RS232 ports to connect to the IMU send, *CMDtoIMU*, IMU receive and *IMUin*, ports. Figure 16 displays the subsystem, which was used with the RC-Truck control.
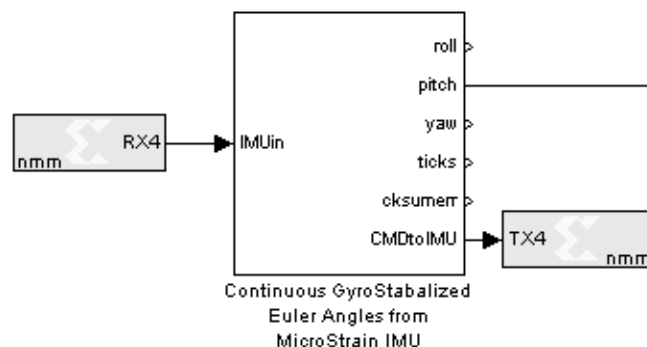
Figure 16: IMU Communication Library Block

41

CHAPTER 5

HARDWARE DESIGN

The autopilot requires dedicated hardware surrounding the FPGA in order to provide use with external devices such as sensors and actuators. In order to provide for use across multiple platforms, flexible interfaces must exist between the various hardware modules and specific hardware modules, which provide the necessary spectrum of capabilities, must exist. Flexible interfaces between the TTL logic inputs and the FPGA are mandatory. In addition, A/D conversion hardware must exist with flexible interfaces to the pressure sensors, which measure altitude and velocity. Hardware modules must be provided to realize a range of customizable analog signal conditioning and provide for RS232 communication. Dedicated hardware must provide separate circuitry for servo control selection. In addition, sufficient memory must be provided to satisfy a range of data acquisition requirements.

5.1 Processing Hardware Selection

An autopilot utilizing a full FPGA implementation is a novelty in the area of unmanned systems. The full FPGA implementation was selected since it provides a broader range of design alternatives to satisfy an expanded set of platform capabilities. Design with full FPGA provides more design versatility than DSP processors or even hybrid DSP/FPGA implementations. The full FPGA implementation provides flexibility and the ability to process different algorithms in parallel such as wireless networking,

vision algorithms, sensor integration and vehicle control implementation. The processor hardware architecture is reconfigurable. Therefore, each signal and variable can be represented using different numbers of bits as required. This allows for higher sampling rates, better accuracy and high computation speed with low power consumption. FPGAs operate at a very high frequency. When the FPGA is combined with parallel computational structures, computational speeds as much as 100 times greater than those possible with digital signal processors are realizable. The computational speed of the DSP is limited since its operation is sequential, [41, 42].

An additional advantage of a full FPGA design is the existence of a natural migration to micro-air vehicles. Once the prototype is developed and the design verified, the power and size of the processing system can be reduced by implementing the tested VHDL algorithm in a system-on-chip design.

Xilinx manufactures FPGA products and has had the foresight to work with Mathworks. This collaboration provides for programming from within *Simulink*'s graphical language, which provides hardware-in-the-loop capabilities. Working in conjunction with Mathworks, Xilinx has developed the System Generator toolbox, which provides the Xilinx FPGAs the capability of full integration with MATLAB/*Simulink*. This functionality facilitates high level abstractions to be directly compiled into an FPGA. In addition, the toolbox directly provides for hardware-in-the-loop simulation. The simulation with *Simulink* requires a standard USB or JTAG parallel port connection for synchronizing the FPGA clock to *Simulink* time.

The FPGA selected was the Spartan3-1400AN. This FPGA houses logic building blocks for 11,264 slices, 32 multipliers, 176K of distributed RAM, a 576K RAM block

43

and 1.4M system gates. Although the Spartan series does not include embedded PowerPCs, Xilinx's EDK program can be utilized to provide soft core DSPs.

5.2   Analog Input Design

As with any system involving sensors, there will be analog inputs, which require signal conditioning before being sampled by an A/D converter, for use in the processor. This leads to the challenge of including flexibility with the analog circuitry design in order to allow the same circuit to be used with different sensors. In the recent past this challenge could have only been accomplished by providing for the physical interchange of various analog components. However, the recent development of digital potentiometers, programmable operational amplifiers and Field Programmable Analog Arrays has facilitated the design of flexible analog circuitry. All of these components were considered for designing programmable analog signal conditioning. Digital potentiometers were considered for use along with either a static or programmable operational amplifier. The disadvantage of this method is the board space required for the components and the lack of analog filtering. Currently, the selection of digitally controlled capacitors is also very limited

The FPAA provides for programmable analog filtering and requires less board space. Two types of FPAAs are currently available. There are FPAAs that operate in discrete time and those that operate in continuous time. Discrete time FPAAs utilize switching capacitors to implement the resistance required in the circuit. The continuous time FPAAs utilize switches to provide for different interconnections of the components, [43]. It was determined that either would work well for this application. However, after

44

and 1.4M system gates. Although the Spartan series does not include embedded PowerPCs, Xilinx's EDK program can be utilized to provide soft core DSPs.

5.2   Analog Input Design

As with any system involving sensors, there will be analog inputs, which require signal conditioning before being sampled by an A/D converter, for use in the processor. This leads to the challenge of including flexibility with the analog circuitry design in order to allow the same circuit to be used with different sensors. In the recent past this challenge could have only been accomplished by providing for the physical interchange of various analog components. However, the recent development of digital potentiometers, programmable operational amplifiers and Field Programmable Analog Arrays has facilitated the design of flexible analog circuitry. All of these components were considered for designing programmable analog signal conditioning. Digital potentiometers were considered for use along with either a static or programmable operational amplifier. The disadvantage of this method is the board space required for the components and the lack of analog filtering. Currently, the selection of digitally controlled capacitors is also very limited

The FPAA provides for programmable analog filtering and requires less board space. Two types of FPAAs are currently available. There are FPAAs that operate in discrete time and those that operate in continuous time. Discrete time FPAAs utilize switching capacitors to implement the resistance required in the circuit. The continuous time FPAAs utilize switches to provide for different interconnections of the components, [43]. It was determined that either would work well for this application. However, after

44

a search of currently available FPAAs it was found that Anadigm produces a discrete time FPAA, which possesses some desirable features.

The AN231E04 includes 8-bit internal A/D converters, which can directly convert the conditioned signal to the TTL format required by the FPGA. It also comes with user-friendly software that allows the design to be tested in simulation and a binary file to be generated. The binary file can be utilized directly by the FPGA for programming during autopilot initialization. Each chip provides up to 38 CAMs, which are predefined analog circuits, and up to three A/D outputs. The CAMs include functions such as filtering, inverted gain and limited gain.

The chips do have some limitations associated with the input signal. The chip utilizes a +1.5V internal reference for circuit common in order to provide for AC signal inputs. In addition, the input is limited to 3V. The +1.5V reference creates an issue with ground referenced signals. However, with some initial voltage division and software design, ground referenced signals can be accommodated.

The autopilot design provides one input for small signal, less than 3V, differential sensors, which can be connected directly to the FPAA's input ports and two large signals up to 26V. Voltage division was used to reduce the larger signals by a factor of 8.96. The FPAA measures the input voltage with respect to the 1.5V reference. However, since the internal A/D is utilized, this can be compensated for within the FPGA software when the 8-bit received value is converted to the input voltage.

Figure 17 displays the external circuitry for the two large signal inputs, one small signal input and the AnadigmDesigner2 software configuration. V_{in1} and V_{in2} accept two input voltages, which are ground referenced and less than 26 V. V_{in3} accepts one small

45

signal voltage to be measured. The magnitude of the small signal must be less than 3V.
The internal configuration of the FPAA utilizes a low pass filter for the large signal
inputs and a gain stage for the small signal input. These circuits are followed by the three
A/D converters to provide the TTL outputs to the FPGA. The FPAA signal conditioning
blocks can be utilized to remove noise and adjust the gain of the measured voltage in
order to obtain better resolution from the A/D converters.
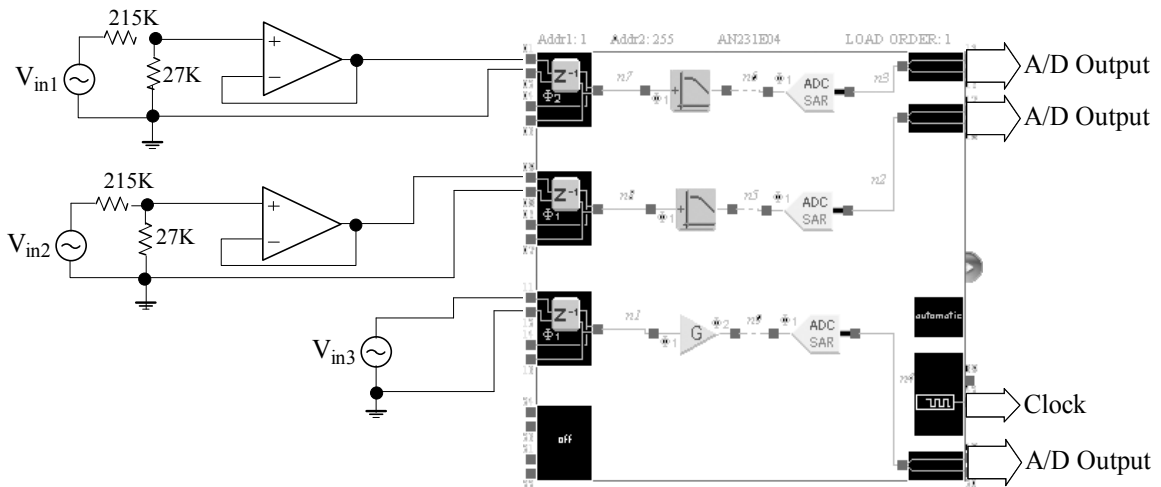
Figure 17: Voltage Measurement Circuit for Analog

5.3   Communication Voltage Level Circuitry

Two types of communication are available to the user of the autopilot.
TTL/CMOS is available at 5V, 3.3V or 1.8V and the RS232 level. There is no universal
circuitry that can accommodate both RS232 and TTL. The two are defined as a specific
type of I/O port.

The TTL I/O lines interface with a bi-directional voltage level translator, which is
Texas Instrument's TXB0104. The IC has an electrical requirement for the set of four
I/O signals to be less than or equal to the voltage after translation. By utilizing the

46

FPGA's 1.8V logic level ports and treating the input as the higher translation level, any TTL voltage between 1.8V and 5V can be accepted. A digital potentiometer, which is programmed through the FPGA processor, is used to set the logic level on the input port. The process is presented in Figure 18.
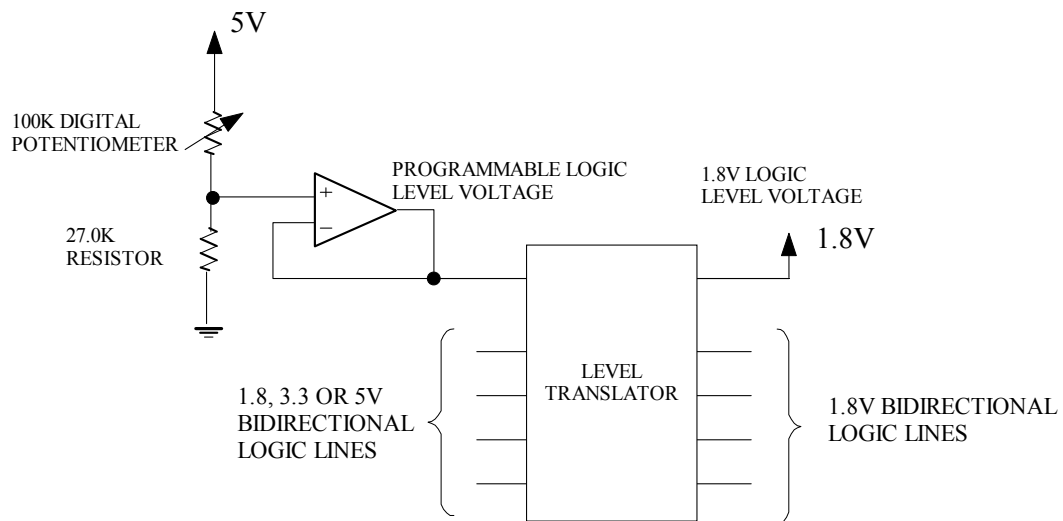
Figure 18: Adjustable Logic Level Circuitry

RS232 communication is older than TTL and does not operate at the standard 5V, 3.3V or 1.8V logic levels, which are now much more popular with processors. The voltage representing logic high can range from +5V to +15V while the logic low representation varies from -5V to -15V. There are several standard ICs that contain internal charge pumps to allow for the negative voltage levels from a single 3.3V power supply. The MAX650 was selected and provides 5 inputs and 4 outputs while requiring only four external capacitors.

In order to minimize board space, USB connectors were not directly included on the board. It is important to note that, with custom connectors, USB communication can be implemented through the variable I/O ports. The board was designed with two five

47

volt connectors near the TTL I/O ports. This arrangement provides for the supply of the five volts required to power the USB interface. The TTL port can be programmed for the 3.3V logic required by the USB specifications.

There are three data rates, 1.5Mbps, 12Mbps and 480Mbps, given in the USB specifications. The third is the high speed data rate specified by USB 2.0. However, to be compliant with USB 2.0 the highest speed in not required. A full speed interface of 12Mbps is still compatible with USB 2.0 devices, [44]. The autopilot is limited to the first two data rates due to the 24Mbps limitations of the level translators. USB communication was not developed for the autopilot template since the bi-directional communication lines required by the protocol are not yet available within *Simulink*. However, the user can still develop the protocol through the ISE program or by embedding a soft core processor using the EDK program, which contains the required drivers.

5.4   Altitude and Velocity Measurement with Pressure Sensors

Two pressure sensors were selected for measuring forward velocity and altitude. The output of these sensors is a voltage ranging from 0.2V to 5V for the altitude and 1V to 5V for the forward velocity. An A/D converter was selected, which provided for an input of up to 5V. The A/D converter also required three 5V TTL communication lines. Since the FPGA cannot produce a logic level above 3.3V, a translator IC was included in the circuit. In addition, a 4.5V reference IC was utilized in order to provide the stable reference voltage required by the converter. The pressure sensor circuit is presented in Figure 19.

48
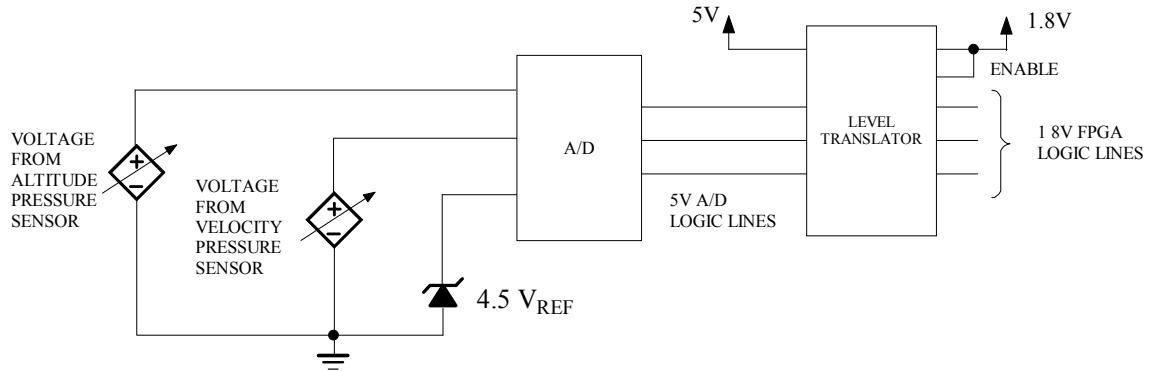
Figure 19: Pressure Sensor Circuitry

The calculation of altitude is based on the fact that pressure decreases as the altitude of an aerial vehicle increases. A pressure sensor can be used to measure this relationship and the altitude calculated. The selected pressure sensor produces a linear relationship between pressure per square inch and voltage. The pressure range is 2.2psi to 16.7psi and the voltage range is 0.2V to 4.8V. The vehicle's height can be calculated using equation (2) and equation (3). Selecting the sea level value as the initial height, the range of height measurable by the pressure sensor was calculated to be from 230 feet below sea level to 26,878 feet above sea level.

$$(psi_{initial} - psi_{measured})(2000) = UAV\_height \tag{2}$$

$$psi_{measured} = (v_{in} + 0.2)(14.5/4.6) \tag{3}$$

Since a 16-bit A/D converter is present, the resolution of the measurement is limited by the quantization steps of the A/D converter, which is 68.6656uV/bit. Relating this value to feet yields the smallest measurable change in height as 0.4329ft. However,

49

due to noise present within the circuitry this accuracy is better than can be realistically expected. Since a 4.5V reference IC was selected, the actual measurable distance below sea level is slightly less than 230 feet.

Calculating velocity using pressure measurement is slightly different. A pitot tube is used to generate a differential pressure value. The differential pressure value is derived as the difference between the static pressure, with no velocity, and the dynamic pressure generated from the wind entering the tube from the aircraft's forward velocity. The differential pressure, which is measured by a pressure sensor, is proportional to the indicated forward air speed of the vehicle. A pressure sensor was selected for this measurement, which is capable of detecting pressure in the range 0 to 3.92kPa and with an output between 1V and 4.9V. As with the altitude sensor, the range was limited to the 4.5V reference. Equation (4) gives the relationship between the measured pressure and indicated air speed in knots,[45]. The a_{sl} parameter is the standard speed of sound at 15°C, which is equal to 661.4788kts. The P_{sl} parameter is the standard pressure at sea level, which is equal to 29.92126in-Hg. The q_c parameter is the measured pressure, from the pitot tube, in-Hg.

$$V = a_{sl} \sqrt{5 \left[\left(\frac{q_c}{P_{sl}} + 1 \right)^{2/7} - 1 \right]} \qquad (4)$$

The relationship between air speed and measured pressure is non-linear. Therefore, the amount of quantization error changes with velocity. However, the smallest theoretical measured step is equal to $2.0381(10^{-5})$in-Hg/bit, which is negligible.

50

5.5 Data Acquisition Memory

The FPGA's internal RAM could be utilized for data acquisition. However, this would be inefficient due to the large number of gates, which would be required, and the fact that the RAM is a volatile memory. Therefore, the use of external memory was included in the design to provide for data acquisition capabilities. Since flash memory is available with large amounts of storage capabilities and is non-volatile, it was selected over RAM memory, which is volatile and requires more board space for the same amount of capacity. The disadvantage of flash is the limited number of write cycles, which are usually around 100,000. This limitation was overcome by selecting a MicroSD card. Therefore, the user can upgrade as write speeds and size are increased and the cards can easily be replaced should the write cycle limitation be reached.

5.6 Actuator Control Selector Circuitry

The autopilot was designed to control up to twelve servos through the use of the FPGA's 3.3V TTL ports. The output from the FPGA is not directly linked to the servo connector pins. Instead it passes through the onboard safety switch circuitry. This circuitry provides for a pilot or, when in use, an additional daughter board, to gain control of the servos in the case of a failure of either the underlying software or the FPGA. The priority order was established as pilot first, the safety board second and the FPGA third. In order to achieve this priority, without excessive use of analog switches, a CPLD was utilized. The CPLD receives each of the control lines from the three sources and selects a control source, which is outputted to the servos. This configuration is presented in Figure 20.
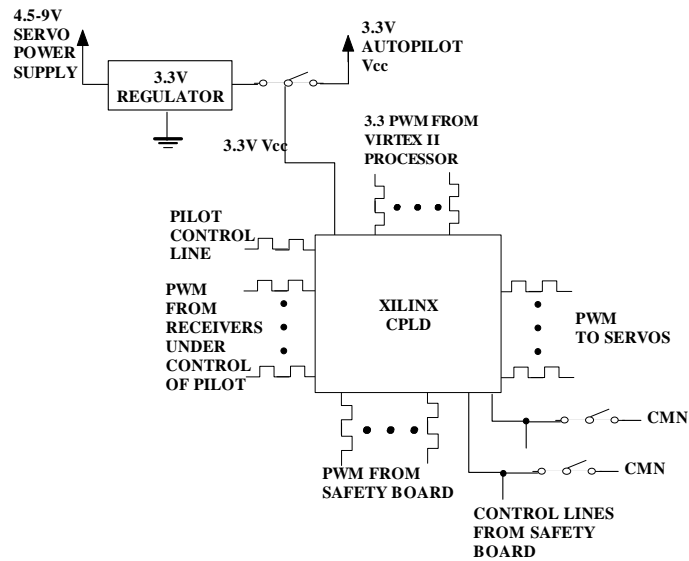
51

Figure 20: Actuator Control

There are six primary servos for control of the vehicle dynamics and six for control of any accessories such as a camera pan and tilt motors. It is unrealistic for a human pilot to control all twelve servos. Therefore, only the six primary servos are available for the pilot to control. All twelve are available to both a daughter board and the autopilot with two select lines available to the daughter board. This provides for a second processing system to control the servos running the accessories while the autopilot maintains control of the primary servos. This configuration is beneficial to systems running a second processing system to handle computationally complex algorithms such as the vision algorithms.

Standard servos run on power supplies in the range of 4.8V to 6V. The power to the control switch can be supplied by the servo connector, running from a separate supply, or the on-board 3.3V supply, which is selected by a jumper. The advantage of providing a separate supply for the servos is the additional isolation gained for the critical actuator control circuitry. It was decided to utilize a second voltage regulator for the

52

safety switch. Since the safety switch is powered by the same supply as the actuators, a loss of servo power would result in an unrecoverable failure even if the pilot were to regain control of the actuator logic.

The control line from the pilot is a 3.3V PWM signal from a receiver. The code in the CPLD monitors the frequency and gives the highest priority to the request of the pilot for control. The control lines from the daughter board are simply a logic high/low signal. Logic high on the control line will give control to the safety board but only if the pilot has relinquished control. Two connections were included in the design in order to jumper the daughter board select lines to logic low when a second board is not present. Once the human pilot has relinquished control and the safety board control lines have been set to logic low, the autopilot gains control of the actuators. The System Generator is not available for the CPLD. Therefore, the safety switch was preprogrammed as part of the autopilot design and does not need to be modified by the end user. However, the JTAG ports are accessible. Therefore, those familiar with VHDL or Verilog can modify the design for other functionality.

5.6.1   Safety Switch CPLD Logic

The safety switch was programmed within the ISE environment using VHDL. The safety switch entity has two building blocks, a frequency conversion module, *freq_conv*, to convert the pilot select line PWM signal into a single bit and a single switch module, *single_switch*, which selects the PWM input to be passed to the servo output. The safety switch configuration is displayed in Figure 21.

www.manaraa.com
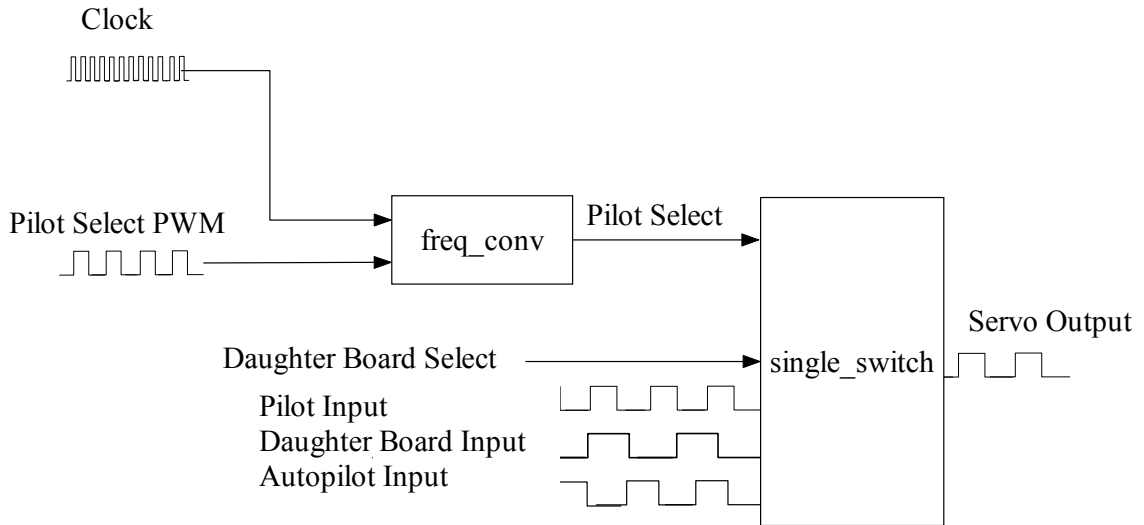
Figure 21: Safety Switch Block Diagram

The single switch component is repeated for each of the twelve PWM inputs. The six

servo outputs not affected by the pilot select have the pilot select bit passed into the

module as logic low. The truth table presented in Table 3 was used to derive the logic

function within the architectural structure of the single switch component. The single

switch logic is presented in Figure 22.

Table 3: Single Switch Truth Table

Pilot Select (ps)	Daughter Board Select (dbs)	Pilot Input (pi)	Daughter Board Input (dbi)	Autopilot Input (ai)	Servo Output (servo)
0	0	X	X	0	0
0	0	X	X	1	1
0	1	X	0	X	0
0	1	X	1	X	1
1	X	0	X	X	0
1	X	1	X	X	1

54

Figure 22: Single Switch Logic

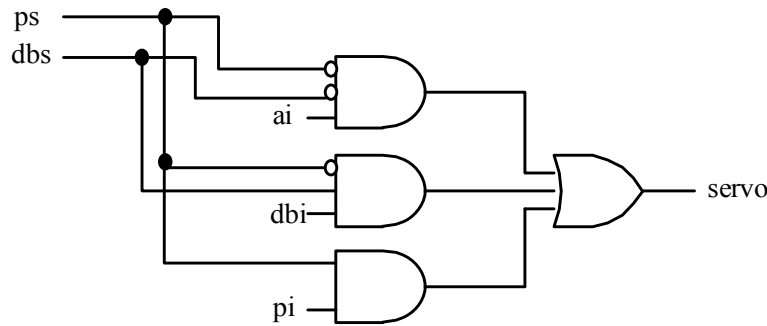The frequency conversion module converts the signal from the receiver. The resulting signal is a 50Hz PWM signal, which toggles between a 1ms and a 2ms high pulse level. The 1ms value corresponds to logic high, and the 2ms value corresponds to logic low.

The CPLD clock operates at a frequency of 50MHz and is used to calculate the time the PWM signal is logic high. A counter is utilized to determine the pulse width. The counter is initialized when the PWM input changes from logic low to logic high and reset to zero when the PWM changes from logic high to a logic low signal. The counter value is used to determine the pulse width and set the output flag accordingly.

5.7 Power Supply Circuitry

The design of the power supply circuitry was provided by Xilinx, [46]. This design was presented in a technical paper, [47]. The design was also utilized and tested on Xilinx's Spartan-3AN starter kit. Therefore, it was considered best to utilize the proven design for the power supply.

The circuitry utilizes National Semiconductor's LP3906. It is powered by the 5V autopilot supply voltage input and provides four output voltages. Two of the outputs are

55

at 3.3V, one at 1.8V and one at 1.2V. The 1.2V and one of the 3.3V outputs are buck DC-DC switch mode supplies. These are utilized to supply the FPGA 1.2V core supply and the 3.3V supply required for the I/O ports bank 0, bank 1 and bank 2. The second 3.3V supply and the 1.8V supply are linear regulator supplies. The 1.8V output is used to supply the bank 3 I/O ports, which are used for the 1.8 TTL logic protocol. Linear regulators have a lower noise characteristic than the switch mode types. Therefore, the linear 3.3V supply was utilized to supply the peripheral analog components. The analog components are involved in measurements, which could be easily corrupted by noise.

The operating voltage of the autopilot was limited to a range of 4.75V to 5.25V by the pressure sensors. Therefore, the autopilot is run from a regulated 5V supply.

CHAPTER 6

AUTOPILOT SOFTWARE DESIGN

In order for a complete design to be developed within the System Generator environment, two separate issues must be addressed. These issues are the development and testing of the software algorithms and the integration of these tested algorithms with the selected sensors and actuators. The first issue has been studied Murthy and a design flow developed in, [48]. The development of the hardware interfaces is addressed in this chapter with the developed autopilot hardware interfaces as design references.

Murthy provides an overview of the System Generator along with a recommended design flow for converting *Simulink* tested algorithms to System Generator/hardware implementation, [48]. The research discussed issues encountered while designing the algorithms at the gate level. These include quantization and overflow, difficulty implementing mathematical algorithms and timing issues. Timing issues associated with algebraic loops are of particular interest and are addressed by Murthy, [48].

Quantization and overflow are issues, which must be addressed with any form of processor utilizing a fixed word length. The required resolution must be selected along with the required precision. An advantage of working with FPGAs is that the word length and assignments of bits to represent the fractional portion can be modified at anytime within the software. Many of the System Generator building blocks provide for the re-assigning of the length at the output. In addition, the representation can be

57

modified by utilizing the reinterpret and convert blocks. The reinterpret library block assigns a different representation without adjusting the bit values. The convert library block reassigns the word length and number of bits assigned to the fractional portion, which will affect the individual bit assignment. The convert block provides for the user to select whether the value is both rounded or truncated and wrapped or saturated.

The mathematical issues addressed are not a lack of availability of System Generator blocks used for implementation. Rather the mathematical issues are concerned with the assigning of the precision and delays along the path. Simple mathematical blocks that introduce very little delay include addition, subtraction, shift, multiply, scaling by 2^n, cosine and sine functions that utilize look up tables. In addition, there are blocks that implement the division, log, sine, cosine, square root and inverse tangent functions by utilizing the Coordinate Rotation Digital Computer (CORDIC) algorithms. The CORDIC algorithms use an iterative approach by performing coordinate rotations in order to obtain an approximation of more complex functions, [49].

Algebraic loops occur when the output of a mathematical function is returned to the input of the initial calculation. Algebraic loops can create timing issues for the hardware designer. These loops require extra consideration with respect to the delay associated with the gates contained within their path. The delay can result either in an instability in the system or an incorrect result by performing the mathematical or logical calculations on samples, which have occurred at different time steps. When developing the System Generator algorithms these delays must be calculated and compensated for carefully, [48]

58

The design of hardware interfaces does not focus on the quantization, over flow or mathematical issues. The design of hardware interfaces focus primarily on the issues of timing. The communication protocol requires tightly controlled timing at the I/O ports with signals that require careful synchronization. When developing the communication protocol the best approach is to first simulate and recreate the waveforms in the *Simulink* scope. This provides for an initial determination of whether the timing and synchronization of the signals were correctly designed. Once the simulation has verified the design, the hardware implementation can be performed. If the hardware does not yield the correct results, the FPGA's RAM can be utilized to store the behavior of the system within the FPGA. This provides for reading the information through the JTAG interface and the recreation of the waveforms within the *Simulink* environment. Since System Generator is bit and cyclic true, the hardware recreation usually finds either an issue which existed in the simulation and was originally missed by the designer or an electrical issue such as an incorrectly assigned hardware port.

The majority of mistakes, which prevent the protocol from functioning, result from timing issues created by delays from the selected gates. The register library block can never have less than a delay of one clock cycle. Other gates such as comparators or logical functions may be set to a delay of zero. It is very important to look at the arrival times of each individual signal. Comparison to a counter value can be used for synchronization. Both register and delay blocks are useful for manipulating the arrival times of signals. The following sections discuss the design of all the hardware for the communication protocol in detail. The discussions in these sections are useful as a reference for similar designs.

59

6.1 FPAA Program Logic Design

The PROGRAM FPAA autopilot template subsystem sends the required clock signal to the *FCLK* port of the FPAA and programs the chip with the bit stream created by the AnadigmDesigner2 program. Counters are used to control the timing of the generated signals with surrounding gates utilized for synchronization. In addition, *Simulink*'s ability to allow subsystems to be developed with variables assigned at initialization through the mask interface was utilized. The mask was utilized for assigning of the bit stream or sequence of ones and zeros, which contain the configuration information, and disabling of the outputs to the FPAA when it is not in use.

The clock signal to the FPAA, *FCLK*, is generated by incrementing a counter between zero and one at twice the required clock frequency, which is 12.5MHz. This signal is OR'd with a Boolean value, which is assigned by the variable *EnFPAA*, set in the mask. The additional Boolean variable provides for disabling the clock signal out of the FPGA when the FPAA is not utilized. The FPAA clock signal is presented in Figure 23.
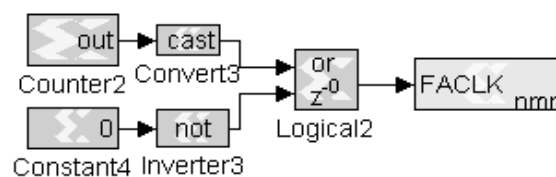

Figure 23: FPAA Clock Signal

Six hardware ports are utilized to program the FPAA. The reset port, *FRES*, enables the FPAA when at logic high. The program chip select port *FCS2B*, which is set to logic low while the configuration bit stream is sent. The bit stream is clocked out of the chip on the data port *FSI*. These output signals are synchronized by the

60

communication clock port *FCLK*. After the chip has been successfully programmed, the

FPAA outputs *FACT* and *FERRB* are pulled t logic high values.

The *FACT* and *FERRB* ports are not required within the logic since these inputs

do not affect the output signals. The port hardware blocks cannot be left unconnected or

the compiler will remove the unused logic. This includes the assignment of these ports as

inputs. In order to prevent this removal, the ports are tied into an unused FPGA output

port. This output is not connected to any surrounding hardware and is defined as

TERM1. The configuration is presented in Figure 24. In addition to *TERM1*, three

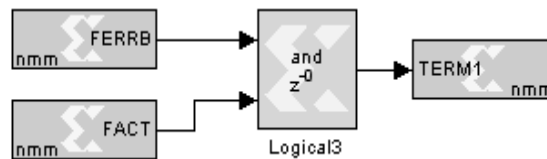additional unused ports *TERM2*, *TERM3* and *TERM4* were defined for future use.

Figure 24: Terminating Input Ports

The bit stream varies in length and bit values for different FPAA configurations

must be entered into the subsystem's mask as a variable. This variable contains a vector

of ones and zeros and is stored in the FPGA's ROM memory, which is set to a width of

1-bit and a depth equal to the number of bits to be sent. The variable is assigned in the

MATLAB workspace through the use of an m-file, which configures the

AnadigmDesigner2 generated binary file to the required vector. Within the mask's

initialization commands an intermediate variable, *A*, is set equal to the user declared

variable to be used within the subsystem's internal blocks. Then the variable *A* is entered

into the ROM as the initial value vector. A counter is utilized to increment the ROM

address at the communication rate. When the final value of the bits, to be sent, has been

61

reached, the counter latches at the last address value. This is accomplished by setting a variable to the length of A less one and compared to the count value. When the comparison outputs logic high, the communication clock is also disabled. Figure 25 displays an overview the system and the variables utilized within the logic blocks.
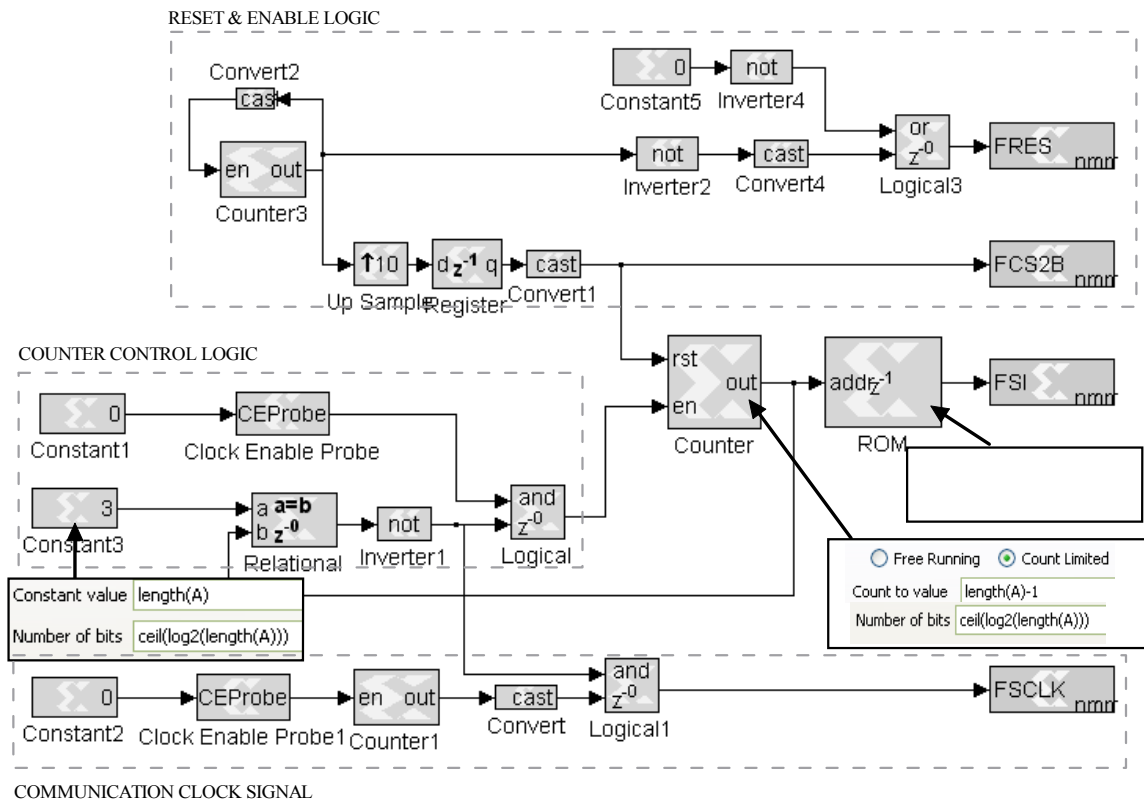

Figure 25: Program FPAA Logic

6.2 FPAA Receive Logic Design

The FPAA contains three internal 8-bit A/D converters, which are utilized for providing the analog information to the autopilot's FPGA. The protocol utilizes three output ports *data*, *synch* and *clk* from the FPAA. The *synch* port is set to logic low during the time when the FPAA is sending the eight data bits. The individual bits are updated when *clk* is logic high and stable for the duration of logic low. The clock

62

frequency, *clk*, is set to 3.125MHz from within the AnadigmDesigner2. The FPAA
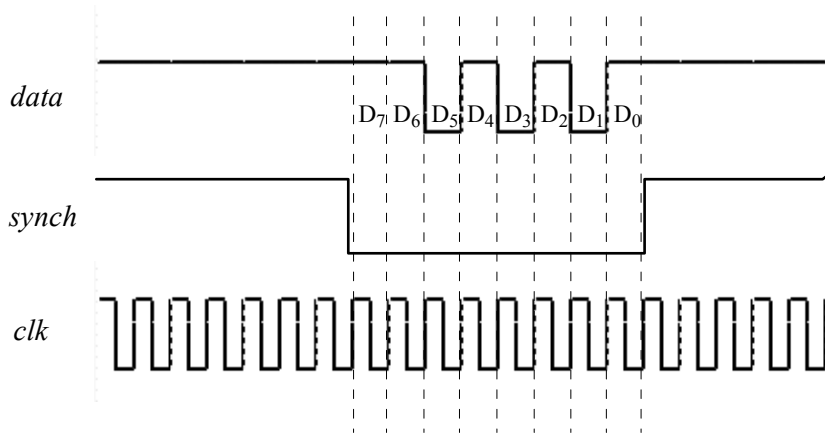
generated waveform is displayed in Figure 26.

Figure 26: FPAA A/D Communication Protocol

Figure 27 displays the System Generator logic for the first A/D input. The

FSYNCH1 port corresponds to *synch*. The *FDATA1* port is set to *data*. The *FSYNCH*

port is set to *synch*. This same logic is repeated for the second and third A/D inputs,

which utilize their individual *synch* and *data* ports and the shared *clk* port as the inputs.
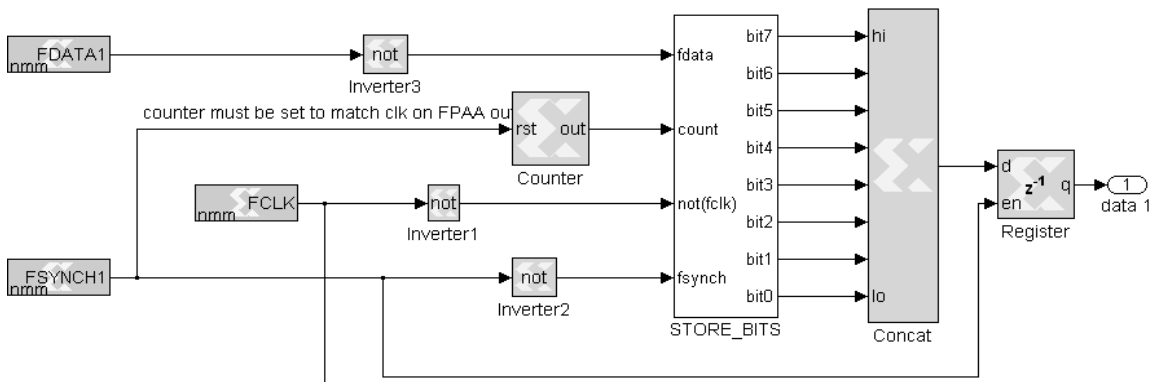

Figure 27: FPAA Receive Logic

A counter is utilized to synchronize storing each of the individual bits arriving

sequentially into the corresponding registers. This counter is held in reset when *synch* is

63

logic high. When synch is logic low the counter increments at the update rate of *data*. The subsystem, STORE_BITS, contains eight 1-bit enabled register blocks. The registers are enabled according to the logic presented in equation (5). The parameter *en* is the Boolean input to the register's enable port. The parameter *bit_number* is the bit's sequential position in the serial input data. The parameter *counter* is the count value.

$$en = (bit\_number = counter) \text{ AND } synch \text{ AND } (\text{NOT}(clk)) \tag{5}$$

The output from each of the 1-bit registers is concatenated into an 8-bit word by the concate block. The 8-bit word is stored in a register, which is enabled by the synch input and is logic high when no data is being received.

6.3 Pressure Sensor A/D Logic Design

The selected A/D converter, the LTC1865, utilizes a standard Serial Peripheral Interface (SPI) protocol at a clock frequency of 500 KHz. The *sdi* input to the A/D converter specifies the settings for the next conversion cycle. In order to set the A/D for channel 0 with single ended measurements, the sequence '1 0' is written. For the same setting but with channel 1, the sequence is '1 1'. The serial data port, *sdo*, provides the readings from the A/D for the previously specified channel. The input line *conv*, to the A/D, is held high to start the conversion cycle and is held high for the minimum required conversion time. The *clk* signal synchronizes the bit transfer, which is stable when the *clk* signal is logic high.

A block diagram, which provides a functional overview of the subsystem for the A/D communication protocol, is presented in Figure 28.
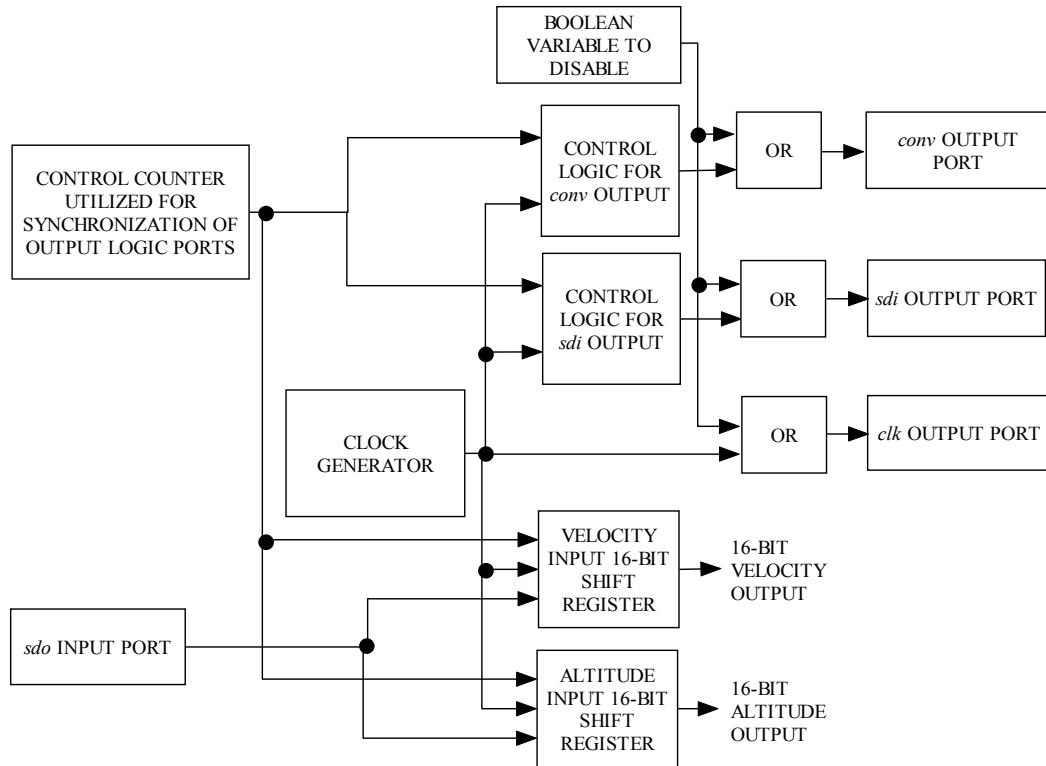
64

Figure 28: A/D Communication Block Diagram

A control counter is utilized in order to synchronize the output waveforms required for the communication protocol and storing of the input. The counter increments from one to thirty-seven and provides the control count value. The required waveforms from the output with respect to the control count value are presented in Figure 29.

The generation of the *sdi* and the *conv* outputs are controlled by utilizing the System Generator's relational block. Each of the relational blocks compares the control counter output to a specific control count value for the required logic high output. The outputs from the relational blocks are then OR'd in order to produce the required waveform. The logic is given for the *conv* output in equation (6)

65

$$conv = (count = 19)\text{OR}(count = 20)\text{OR}(count = 21)\text{OR}(count < 3) \qquad (6)$$

and the logic for the *sdi* output is given by equation (7)

$$sdi = (count = 3)\text{OR}(count = 22)\text{OR}(count = 23) \qquad (7)$$
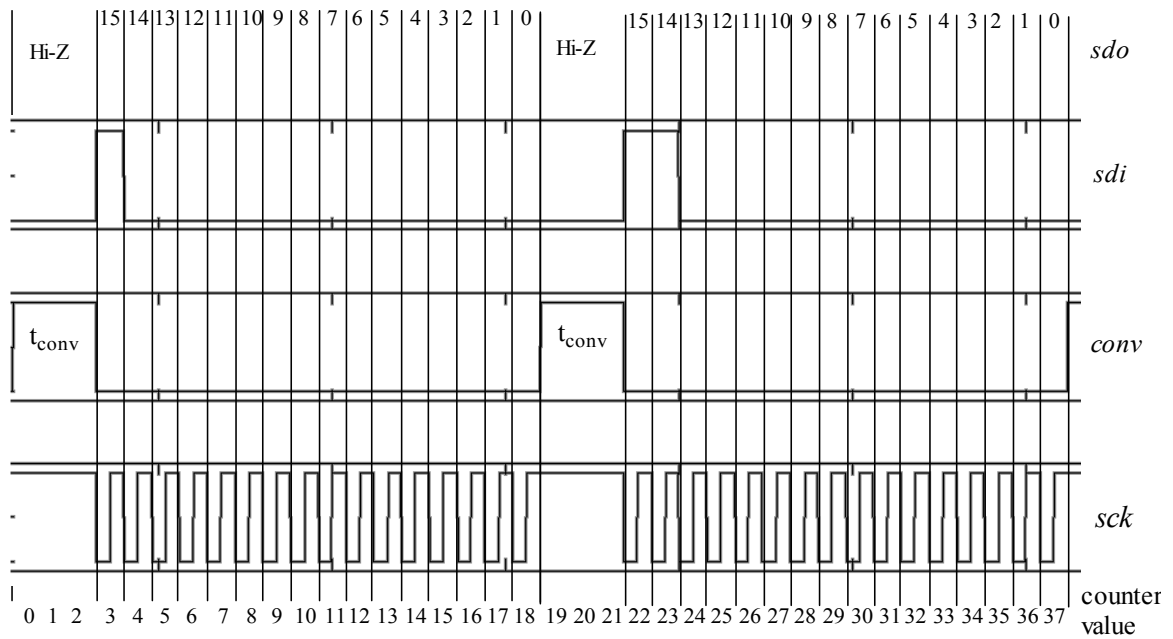

Figure 29: A/D Converter Timing

In order to produce a stable output, during the time when the clock is logic high, a
register is utilized with an enable port. The register is activated with the inverted value of
the generated clock signal, which is received from the *RegEn* subsystem input. This
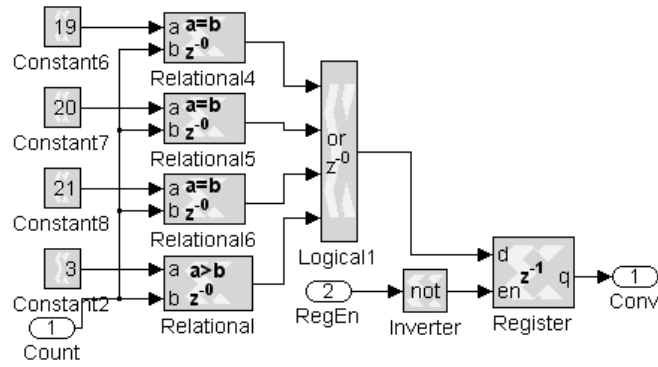configuration is presented in Figure 30.

66

Figure 30: Logic to Generate Convert Output

The System Generator CEProbe block outputs a logic high pulse equal to the time when the hardware clock is logic high. These pulses occur at the update rate of the constant block input. This is used to generate the communication clock signal, *clk*. The constant is set to an update rate of twice the required communication clock frequency, which is 2usec. The generated pulse enables a counter to toggle between 1 and 0 to produce the required frequency with a 50% duty cycle. The generated clock signal is converted to a logic value and OR'd with the *conv* waveform. The or-gate is used in order hold clock signal logic high for the duration of the A/D conversion cycle. A delay of half a communication clock cycle, 1usec, is necessary in order to synchronize the *sck* logic low signal with the *sdi* and *conv* waveforms. This delay is created by utilizing a register block immediately before the output port. The System Generator implementation is presented in Figure 31. The PS_SCK hardware port block corresponds to the *sck* output.
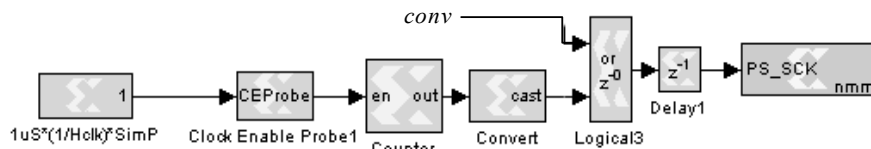

Figure 31: A/D Clock Generator

67

The 16-bit shift registers contain two of the 8-bit registers, which were created for use in the FPAA communication logic. The shift registers were modified to collect each individual bit at the correct counter corresponding to the A/D timing, which was presented in Figure 29. The outputs of these registers are concatenated to form the 16-bit word. An enabled register is utilized to store the concate block output. The register is only permitted to update following the arrival of the last bit. For the 16-bit shift register collecting the channel 1 input the update occurs when the count value is equal to zero. This process is presented in Figure 32.
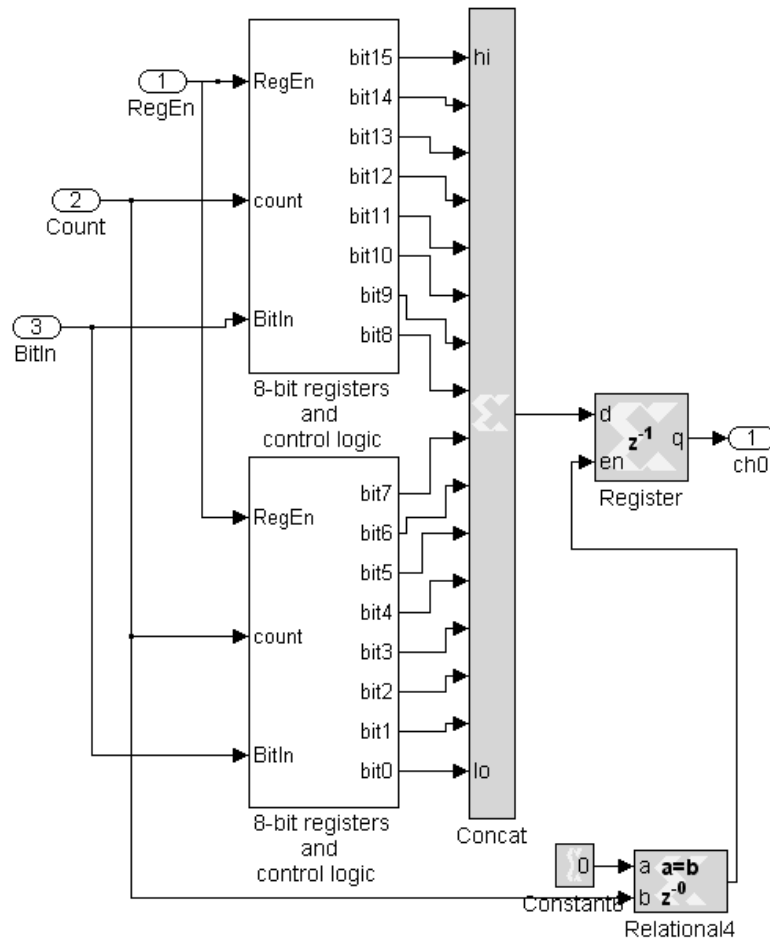

Figure 32: Pressure Sensor A/D Input Logic

68

In order to allow the user to disable this template subsystem, a mask is used to set a Boolean variable that is OR'd with each of the outputs in order to hold all the outputs logic high when the system is disabled.

6.4   Micro Secure Digital Software Design

The MicroSD card is included in the hardware design to provide for the storage of information at run time. With data acquisition, the final selection of the word length and sampling time are dependent on the individual design, which creates the possibility for many different logic configurations to exist. Therefore, only the initialization routine was included in the template. The clock rate is set to 25MHz and the data length to 512, 8-bit memory locations.

The card must go through a sequence of commands in order to initialize. The first command CMD0 sets the card to the idle state and SPI protocol. The second command CMD8 requests information regarding the card state. The third command CMD1 tells the card to initialize. The fourth command CMD16 sets the data length to 512 bytes. After each of these commands is received the card sends a specific response, which must be checked. CMD8 sends a 40-bit response, which is detailed in Figure 33, [50]. CMD0, CMD1 and CMD16 send back a 7-bit response, which is the highest byte of the CMD8 response.

An individual subsystem was built for each of the commands. As the correct response is received, a register is latched high to enable the next sequential command. After the final command has been received successfully the CMD16 response latches an enable flag, termed *Ready*. This flag is outputted from the subsystem to indicate that the

69

card is ready to receive the next instruction such as read or write. The subsystem is
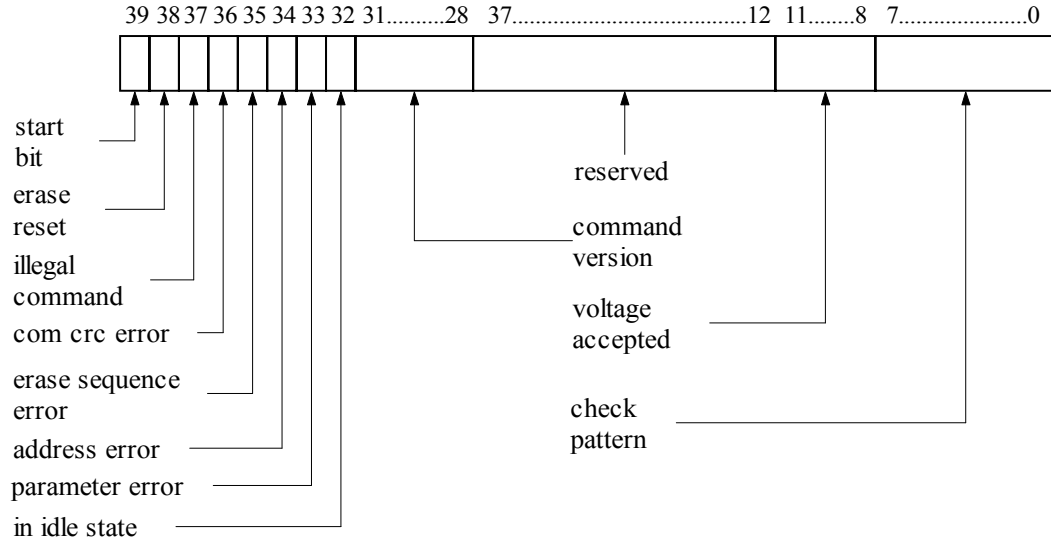
presented in Figure 34.
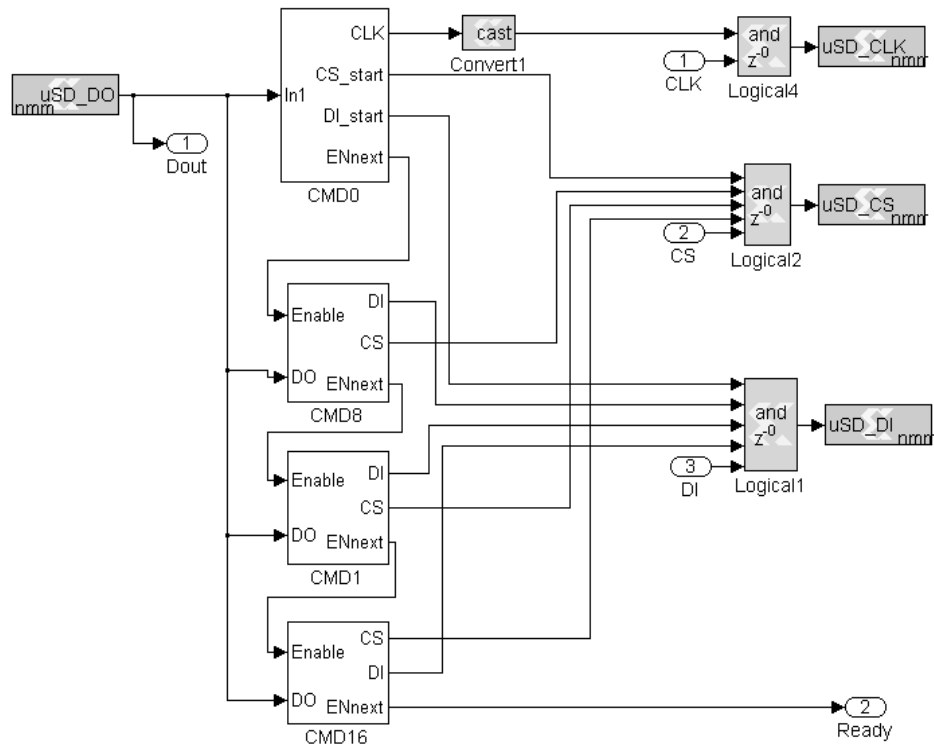

Figure 33: MicroSD Card Response

Figure 34: MicroSD Card Initialization Logic

70

All the commands sent to the card share the same output hardware ports and must be able to take control, without conflict, when active. Therefore, all of the command outputs along with an input port to the subsystem are AND'd just before each of the corresponding hardware ports. The input ports to the subsystem are for user read and write commands. The subsystems are designed so that the output is logic high when inactive. The hardware ports *uSD_CLK*, *uSD_CS*, *uSD_DI*, and *uSD_DO*, given in Figure 34, correspond to the communication clock port *clk*, the chip select *cs*, the data port to the MicroSD card *di*, and the data port from the MicroSD card *do* respectively.

The subsystem that sends CMD0 waits 3ms to provide the memory card time to power up. After this delay, it sends the correct bit sequence for CMD0, receives the response and sets the enable flag output. This process is presented in Figure 35.
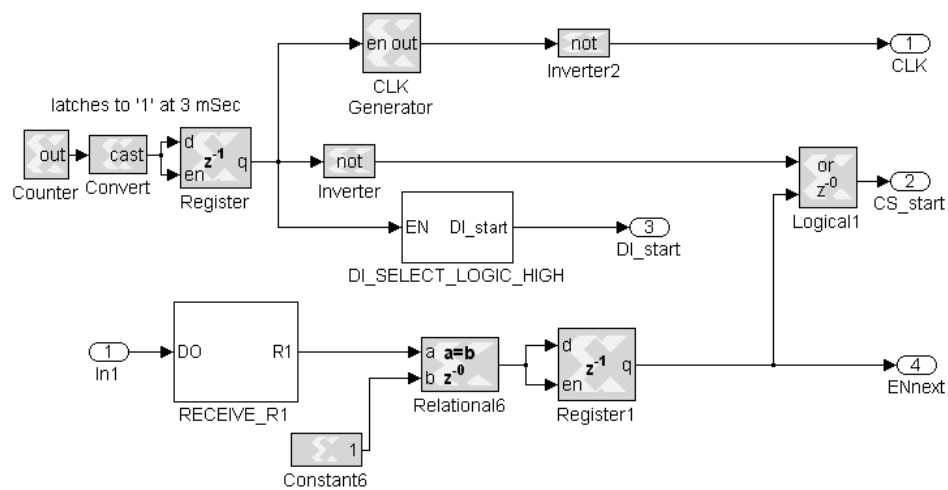

Figure 35: CMD0 Subsystem

In order to create the 3ms delay a counter is utilized, which counts from 0 to 1 with a 3ms update rate. When the counter has incremented to the value one a register is latched to logic high. This register output enables the counter utilized as the communication clock output, *clk*, and the subsystem that sends CMD0, which is

71

D1_SELECT_LOGIC_HIGH. The subsystem, which receives the card's response, RECEIVE_R1, is always enabled. The MicroSD card always sets the first bit sent equal to zero to indicate the start of the transmission. When this occurs, the RECEIVE_R1 subsystem starts a control counter, which is set at the communication rate. The counter is used to enable eight registers when the corresponding register enable bits arrive. When the last bit has been received a register storing the concatenated 8-bit value is enabled. The relational block is utilized to compare the received word to the correct response, which is equal to the value one. When the correct response is successfully received a register is latched logic high, which is the enable output of the subsystem.

The DI_SELECT_LOGIC_HIGH subsystem contains a counter, which counts from 0 to the value of 147. When the final value is reached, the output is latched to logic high. The count values of 0 to 100 are required to provide the card with a clock input for a short time before the command is sent. The count values of 100 to 147 represent the 48-bit word, which is to be sent. The DI_LOGIC subsystem contains relational blocks to compare the count value to the location of the logic high bits within the 48-bit word and OR the results. The function given in equation (8),

$$di = (count=100)\text{OR}(count=101)\text{OR}(count=140)...\\ \text{OR}(count=143)\text{OR}(count=145)\text{OR}(count=147) \tag{8}$$

produces the signal sent from the *di* port to the MicroSD card, which is the correct sequence of logic high pulses.
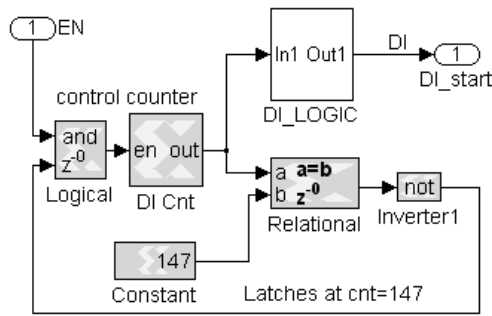
72

Figure 36: CMD0 Logic Output Subsystem

The next subsystem to be enabled is the CMD8 in Figure 34, which sends the
CMD8 command. This subsystem follows the same design flow as the command CMD0
discussed previously, with three exceptions. The MicroSD subsystem for sending CMD8
is presented in. Figure 37.

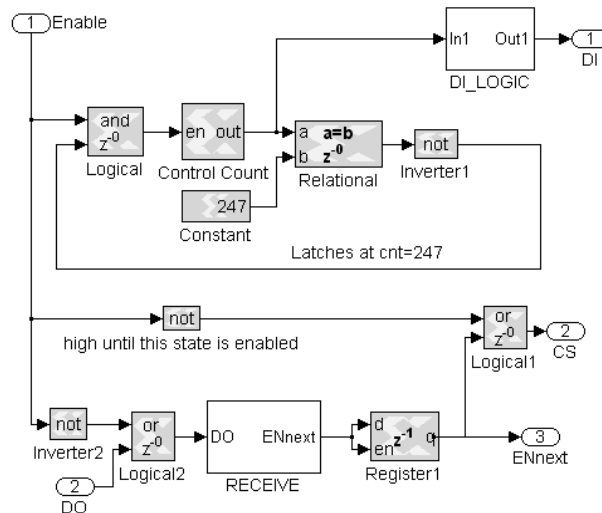Figure 37: MicroSD Send CMD8 Subsystem

The CMD8 subsystem is enabled from the output of CMD0 instead of a timer
delay. The 48-bit command has a different sequence sent than CMD0. The response is
forty bits rather than seven. It was found that if the next command was sent too soon, the
card did not respond properly. Therefore, the command is not sent until 200 clock cycles
after the subsystem is enabled.

73

As with the output of CMD0, the MicroSD send CMD8 subsystem's *di* output is generated with relational blocks to compare the count value to the required logic high sequence followed by an OR gate. The function, which produces the *di* output port signal for CMD8 is given in equation (9) as:

$$di = (count=200)\text{OR}(count=201)\text{OR}(count=204)\text{OR}(count=231)\text{OR}...$$
$$(count=232)\text{OR}(count=234)\text{OR}(count=236)\text{OR}(count=238)\text{OR}...$$
$$(count=240)\text{OR}(count=245)\text{OR}(count=246)\text{OR}(count=247) \quad (9)$$

Contained within the 48-bit word sent to the MicroSD card is an 8-bit value equal to 170. This value is a pattern check and is returned within the response from the card.

The subsystem that receives the MicroSD response to the CMD8 command, RECEIVE, checks that the first 8-bit word received from the MicroSD card is equal to 1 and that the pattern check, bits 32 through 39, is equal to 170. The MicroSD subsystem for receiving the CMD8 response is presented in Figure 38.
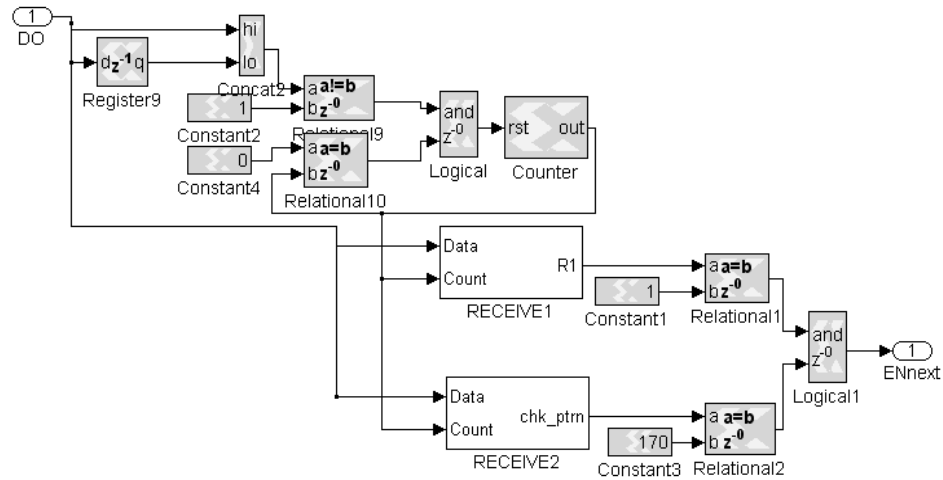

Figure 38: MicroSD Receive CMD8 Response Subsystem

74

Once the conditions checked are true a register is latched to logic high. The register output is the subsystem flag *ENnext*, which provides the next command to be sent. When enable input signal *Enable* is logic high and the *di* signal is logic low a counter is enabled. The counter is used to synchronize storing of the arriving bits. The subsystems, RECEIVE1 and RECEIVE2, contain registers, which are enabled by corresponding counter values. The values are concatenated to form the 8-bit word, which is the output of the subsystem.

The next subsystem is MicroSD CMD1, which sends the CMD1 signal. This subsystem follows the same design as the CMD8 subsystem by utilizing a counter for synchronizing the sending of the *di* sequence, which represents the 48-bit command. The MicroSD CMD1 subsystem is presented in Figure 39.
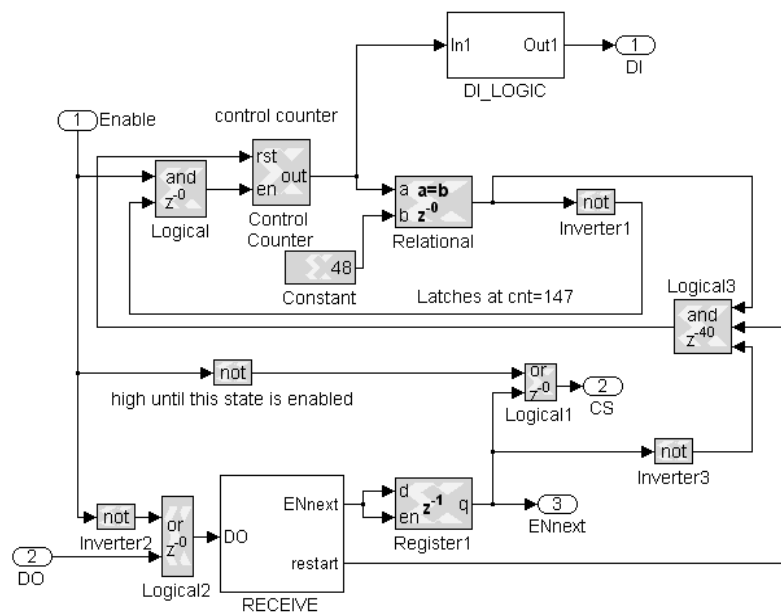

Figure 39: MicroSD CMD1 Subsystem

The counter is enabled when the *Enable* input is logic high. Unlike the previous commands discussed, the response to this command is slightly different. The card reacts

75

by returning an 8-bit response equal to 0 while it is still initializing and equal to 1 when it is ready for use. The communication protocol requires the card to be polled for the information. In order to accomplish polling of the card, a reset is included in the circuit, which restarts the send CMD1 control counter when a response equal to 0 is received. The subsystem for receiving the MicroSD CMD1 response, RECEIVE, which is depicted in Figure 39, works the same as the previously discussed MicroSD CMD1 send subsystem except for one addition. The MicroSD CMD1 receive subsystem performs an additional comparison to the value 0, which when true, sets the *reset* output to logic high in order to resend the command.

The final command is sent by the MicroSD CMD16 subsystem. Once the data is sent, the subsystem waits for an 8-bit response, which is equal to one. Once the response is received, a register is latched producing the signal out of the initialization subsystem that provides the enable flag to the next subsystem, *ENnext*. The MicroSD CMd16 subsystem is presented in Figure 40.
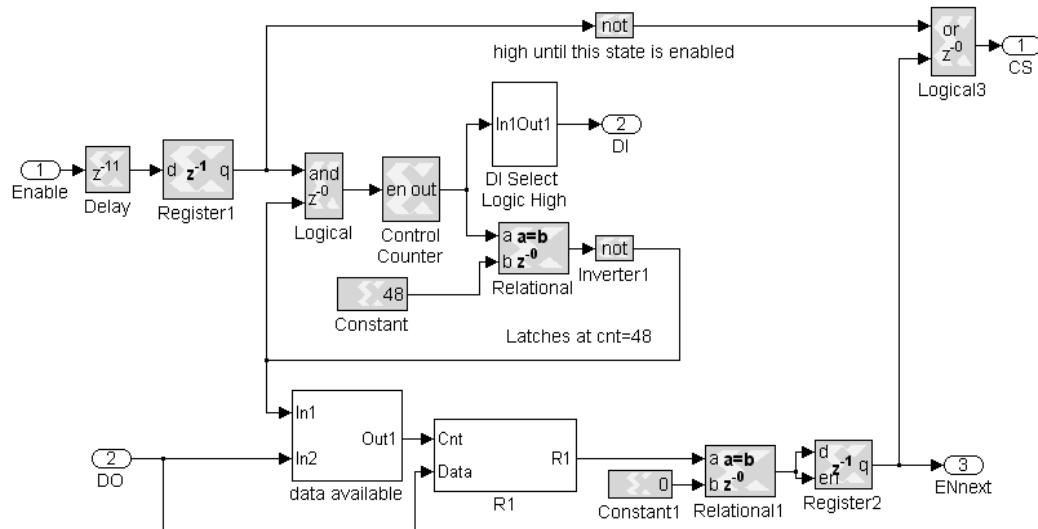

Figure 40: MicroSD CMD16 Subsystem

76

6.5   RS232 Logic Design

Communication subsystems were created to provide user capability for the RS232 communication protocol. A subsystem is included within the autopilot template to enable/disable the autopilot ports. In addition, three subsystems are included within the library for sending, receiving and down-sampling the received data.

6.5.1   RS232 Disable Logic

Communication via RS232 is disabled by setting the enable and the shutdown pins on the MAX561 transceiver IC to logic low. When these IC control lines are logic low, the transceiver holds all the I/O pins as high impedance. The subsystem is masked and the inputs to the ports set as a variable constant. The RS232 disable logic is displayed in Figure 41.

Figure 41: RS232 Enable Logic

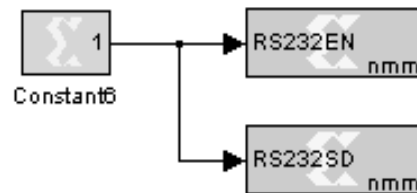### 6.5.2   RS232 Send Logic Design

A library subsystem was designed to send ASCII characters utilizing a standard protocol of no parity and one stop bit. The subsystem for sending RS232 protocol is presented in Figure 42.

The inputs to the subsystem, *ASCII* and *Out_EN*, set the sampling rates to the following blocks. The *ASCII* input is the 8-bit character to be sent. It is concatenated to

77

the start bit, equal to 0, and the stop bit, equal to 1. A parallel to serial System Generator

block is used to rotate the 10-bit result, which causes the lowest bit to be sent first. The

parallel to serial converter increases the update rate by a factor equal to the number of

bits being sent. Therefore, the input must be set to one-tenth of the bit rate for the

selected baud rate. The output of the parallel to serial block is converted from a numeric

to a logical representation by utilizing the cast block. The logic output is OR'd with the

up-sampled and inverted *Out_EN* bit. This forces the output *BIT*, of the subsystem, to
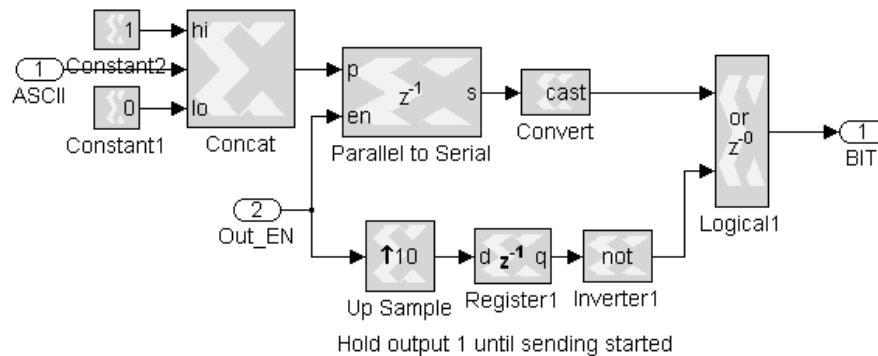
logic high when a character is not sent.

Figure 42: Send RS232

6.5.3   RS232 Receive Logic Design

A library subsystem was designed to receive 8-bit data utilizing RS232 with no

parity and one stop bit protocol. A functional block diagram of the system is displayed in
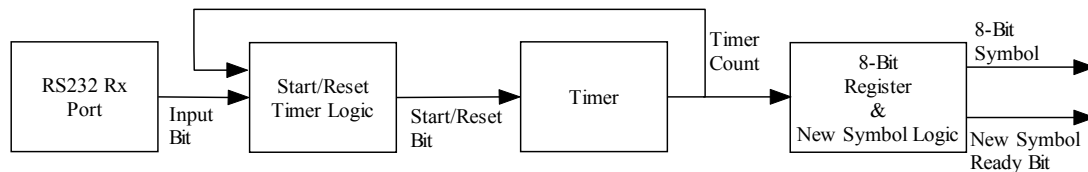
Figure 43.

Figure 43: RS232 Receive Diagram

78

The rate is set by the user selecting the baud value from a drop-down menu, which is provided through the use of a subsystem mask. The selection sets a variable, which is used within the subsystem, to synchronize the logic to the corresponding baud rate.

The System Generator program requires the sampling time on the input port to be a multiple of the clock rate. This requirement necessitated a more complex design for this subsystem. Rounding the update rate to the nearest allowable value creates a slight timing offset for each of the baud rates. The baud rates are listed in Table 4.

Table 4: RS232 Bit Timing

Baud Rate (bps)	Used Bit Rate (uSec)	Actual Bit Rate (uSec)
9600	104.20	104.1667
19200	52.10	52.0833
38400	26.04	26.0417
57600	17.36	17.3611
115200	8.68	8.6806

For a single byte the offsets are small enough to be negligible. However, over time the offset are cumulative, which eventually leads to a communication error. Therefore, the port was oversampled at the clock rate and a counter, utilized as a timer, synchronizes the reception of each bit. The timer is started when the input changes from logic high to logic low, which essentially realigns the timing for each character received. Figure 44 displays a recorded waveform for one byte.

Notice that the sampling occurs close to the center of the time the bit is available. This provides for the system to overcome the expected slight offset and any small amount of jitter, which may occur.
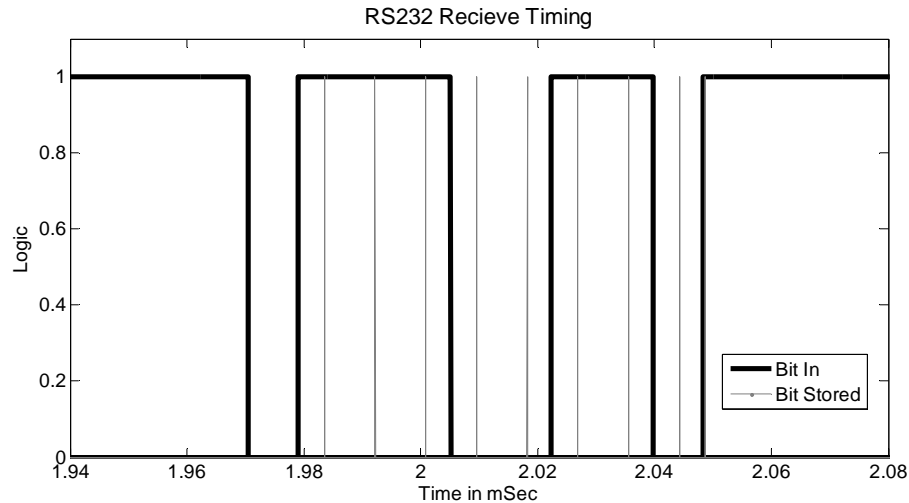
79

Figure 44: RS232 Receive One Byte

The timer that synchronizes the storing of the individual bits is implanted within a

counter block, which is incremented at a sampling rate of 50MHz. The count is started

when the input changes from logic high to logic low, which signals the arrival of a new

byte. This is detected by comparing the input, which is down-sampled to 25MHz, to

itself with a delay of one sample. The down-sampling is required to meet timing

constraints. The two bits are then concatenated and compared to a value of 1. When this

condition is true a register is latched so that the system cannot reset until all eight bits

have been received. The register output is then up-sampled to force the counter to run at

50MHz. After up-sampling the register is inverted in order to hold the counter in the

reset condition while waiting on a byte to arrive. The register controlling the counter is

reset to logic low when the final count value is reached. This value is dependent on the

baud rate selected for the counter and a constant block. The baud rate is set by the *baud*

variable. The constant is used as a comparison to the count value in order to control the

reset of the system. The variable is set inside the subsystem mask when the user selects

the baud rate from the menu. The logic and use of this variable is presented in Figure 45.

80

RESET TIMER

19*baud

START TIMER
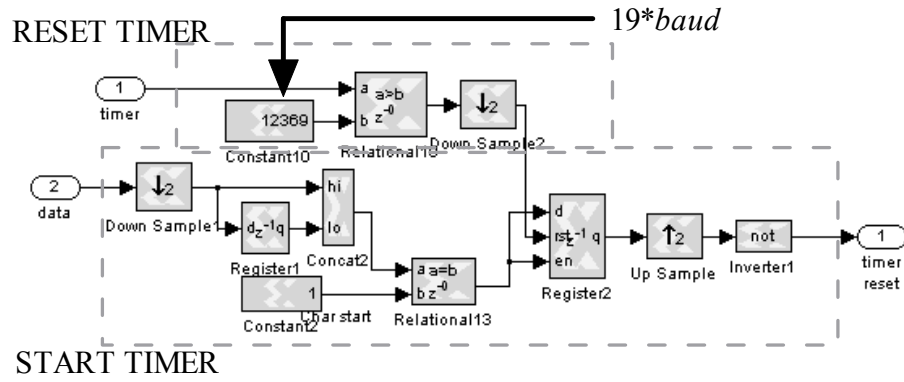
Figure 45: RS232 Timer Control Logic

The individual bits are received into one of eight registers when the counter equals the correct value. This value is calculated by equation (10),

$$counter = baud * 2 * bit\_number + 1 \tag{10}$$

where *baud* is the bit rate for the corresponding baud rate and *bit_number* corresponds to the order of the received bit. The outputs of these eight registers are then concatenated into one 8-bit word, which is stored in an additional register. After the last bit has arrived, the output of the register holding the 8-bit word is then updated and a byte ready flag, *newSym*, is set for one clock cycle. The process is presented in Figure 46.

Figure 46: RS232 Receive Byte Subsystem

81

6.5.4   RS232 Down-Sample Logic

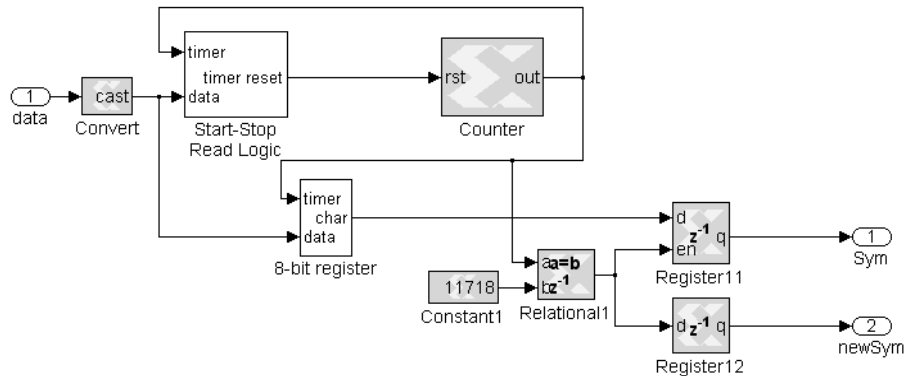System Generator synchronizes the output of any block to the incoming rate. Therefore, the output of the RS232 library subsystem, by default, is 50MHz. Attempting to run complex algorithms at this rate creates unrealizable timing constraints. In order to avoid such constraints, an additional library subsystem was developed to down-sample the incoming byte and the bit ready flag to the communication baud rate. This system is masked with a user selectable drop-down list, which sets a variable, *baud*, based on the communication baud rate chosen. The variable is incorporated within the blocks of the subsystem in order to synchronize the timing to the selected baud rate.

System Generator does supply a down-sample function. However, it cannot be utilized alone with the bit ready flag. Since the flag is only available for one hardware clock cycle of 2ns, it will not be down-sampled correctly. The bit ready flag input, *NewSym*, is used to control the timer. The timer provides the output, which can be down-sampled to provide the bit ready flag, *newsym*. This flag will remain at logic high for a period of one clock cycle of the communication rate chosen. The timer is implemented with a counter. The update rate is the hardware clock rate set by the counter's reset input. The timer value is controlled by setting the counter to increment to the value *baud* reduced by one. Since the counter starts the increment at zero, the reduction of one is included. The timer enable/reset technique used in the receive subsystem was also repeated for this implementation. The logic used to down-sample the bit ready flag is displayed Figure 47.

The 8-bit word is held in a register, which is stable between updates. Therefore, it can be correctly down-sampled by *baud*. However, an additional register is required in

82

order to have the updated value transmitted on the same clock cycle as the byte ready
flag. This is due to the delay introduced by the register controlling the counter. The
logic for down-sampling the 8-bit word is illustrated in Figure 48.
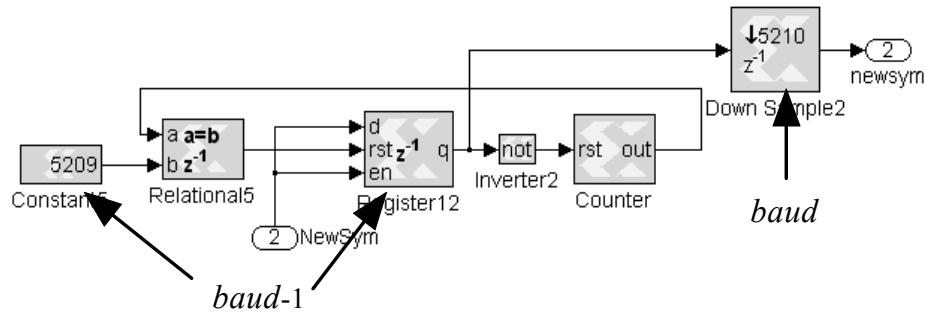
Figure 47: Down-Sampling New Bit Logic
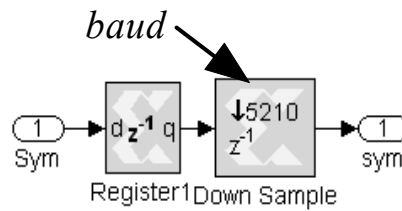
Figure 48: Down-Sampling RS232 Symbol

6.6   Variable I/O Port Voltage Set Logic

The logic level setting for the TTL variable voltage ports is controlled by a digital
potentiometer, which sets the voltage control pin on the level translators. A single IC
containing six individual potentiometers was utilized, as discussed in Section 5.3. The
communication protocol to the IC is a standard SPI protocol, which possesses a clock
input line, a data input line and a chip select line. The data is sent serially as an 11-bit
word. The highest three bits specify the potentiometer to be set and the lower eight bits
specify the trim setting in 255 incremental steps. The subsystem is presented in Figure
49.

83

Figure 49: Variable Port Voltage Set Subsystem

The subsystem is masked to allow the user to easily select between 1.8V, 3.3V and 5V logic. Within the mask, the initialization code was written to assign the correct value to the c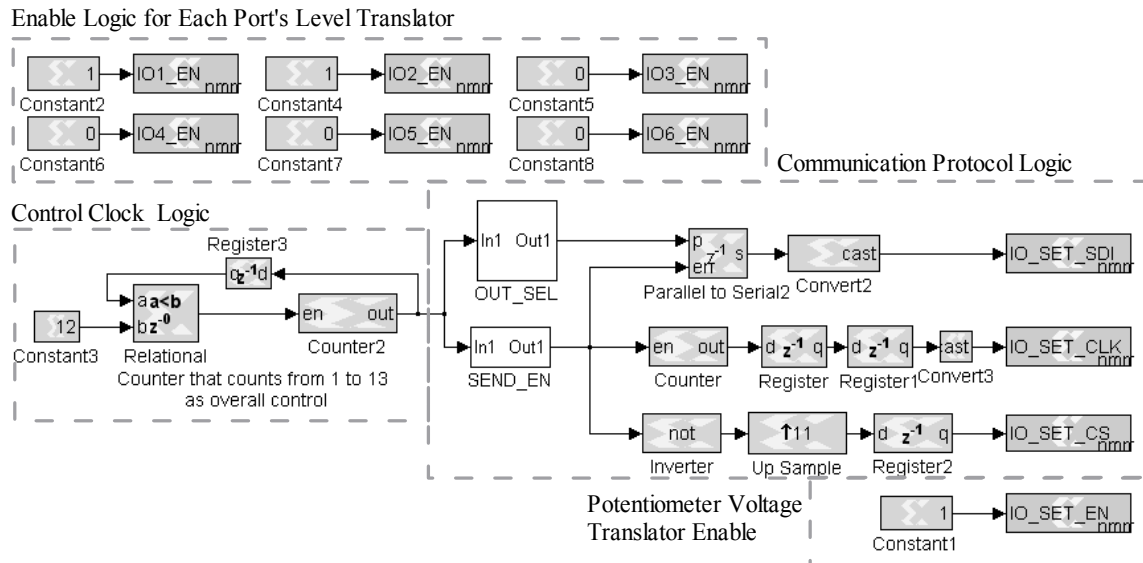onstant containing the trim setting for each port. If the enable block is not selected, the corresponding enable output for each level translator is set to zero. This holds the current unused autopilot I/O ports in a high impedance state. Figure 50 demonstrates the protocol for one potentiometer setting.

The clock frequency is set to 50KHz, and controlled by a counter, which toggles between zero and one. The 11-bit word is then sent to the data input line through a parallel to serial converter. In order to allow the potentiometer select and trim setting to be specified separately, two constant blocks are utilized and then concatenated to a single 11-bit word. This process must be repeated six times for each of the internal potentiometers. In addition, the chip select must be set to logic low for a short period after each setting is received. A counter is utilized to control when each of the 11-bit words is sent. The counter is stopped when it reaches the value of thirteen. This is

84

accomplished by channeling the output back through a relational block, which sets the

counter enable input to logic low when the final value is reached. Table 5 displays the

required action for each count value.
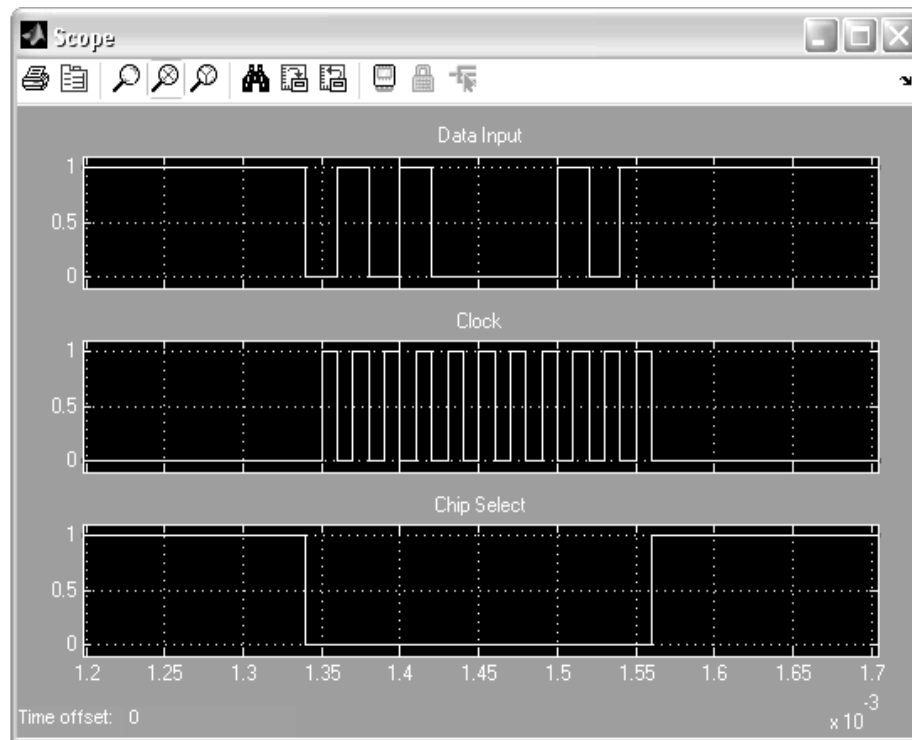
Figure 50: Potentiometer SPI Protocol

Table 5: Port Setting Control Counter

Counter value	Action
1	Chip not selected
2	Send data for port 1 potentiometer
3	Chip not selected
4	Send data for port 2 potentiometer
5	Chip not selected
6	Send data for port 3 potentiometer
7	Chip not selected
8	Send data for port 4 potentiometer
9	Chip not selected
10	Send data for port 5 potentiometer
11	Chip not selected
12	Send data for port 6 potentiometer
13	Chip not selected, counter is now disabled.

85

The communication protocol is enabled by the logic contained in the SEND_EN subsystem and is given in equation (11)

$$(count=2)\text{OR}(count=4)\text{OR}(count=6)\text{OR}(count=8)\text{OR}(count=10)\text{OR}(count=12) \quad (11)$$

When the output is logic high it enables the logic responsible for sending the 11-bits. The parallel to serial converter is enabled to provide the output data on the IO_SET_SDI port. The counter, which outputs the clock signal on the IO_SET_CLK port, is enabled and inverted to directly provide the chip select output on the IO_SET_EN port.

The data sent to the PARALLEL TO SERIAL block is controlled by the OUT_SEL subsystem. The subsystem is presented in Figure 51.
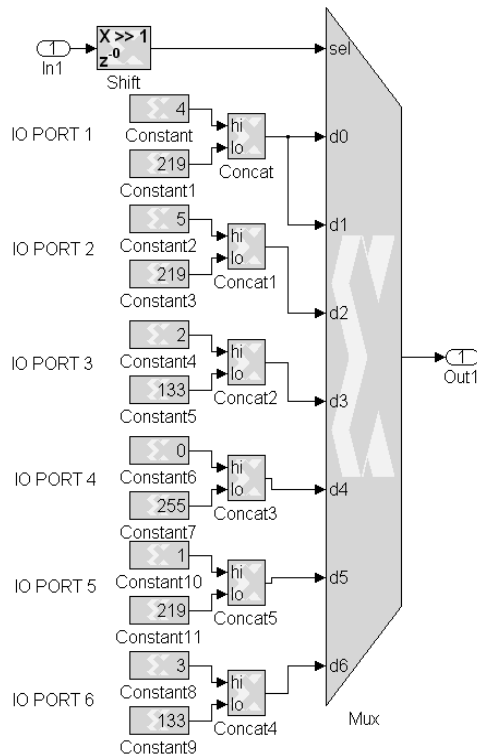

Figure 51: Variable Port Data Output Multiplexer

86

Individual constant blocks are utilized for specifying each of the individual potentiometer trim setting. This provides for each setting to be specified through a user selection variable contained in the subsystem's mask. The count value is shifted left by one bit in order to specify the correct multiplexer output. Since the multiplexer's output starts with reference 0, a seven input multiplexer was selected. The multiplexer input line referenced to 0 is tied to the first potentiometer setting. This first value is never sent. The first value is used to provide for the selection to occur between outputs at every other count value. This establishes the required delay between each 11-bit word being sent.

6.7 Servo PWM Output Logic

The autopilot is designed to provide for servo control. Therefore, a subsystem was developed to generate the required PWM signals. The output frequency is specified by the user to be between 20Hz and 100Hz. The system is designed for an input of 0.00% to 100.00% duty cycle. A functional block diagram of the PWM generate block is presented in Figure 52.
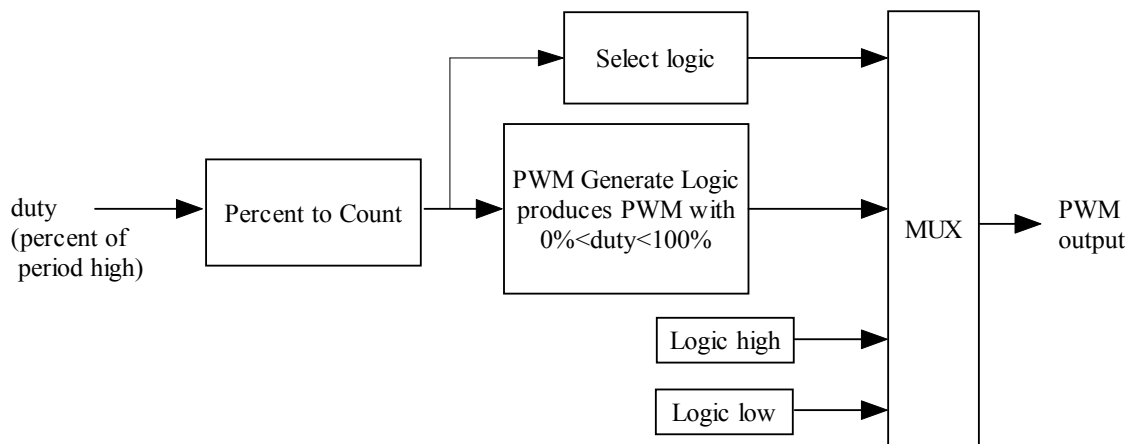
Figure 52: PWM Generate Block Diagram

87

A counter is utilized as a timer for the generation of the PWM output. The maximum count value for 100% duty cycle is assigned from a variable calculated within the subsystems mask, which uses the user specified frequency. The select logic controls the multiplexer output selection. PWM output is logic high when the duty cycle input is 100% and logic low when the duty cycle input is 0%.

The percent duty cycle must be converted to a count value. The ratio given in equation (12)

$$count = duty \frac{\text{update frequency}}{\text{PWM frequency*100}} \tag{12}$$

is contained within a multiplier block. The converted value is loaded in the counter at the start of the clock cycle. When the value is reached the register containing the output bit is enabled. The register is forced to logic low through the use of an inverter block, which is contained within a feedback loop. When the counter reaches the final count value the process is started again. The detailed logic for the generated PWM output is presented in Figure 53.

The input to the PWM generator is set to fourteen bits. Seven bits are used to represent the integral portion and seven bits represent the fractional portion. The quantization error is limited to $1/2^7$, or +/- 0.0078. This provides an accuracy of 1us for the pulse width. This accuracy was required since the standard operating range of a 50Hz servo is 1msec to 2msec. The update rate of the *duty* input is limited to 100Hz in order to guarantee the multiplication stage will meet the timing constraints.

88
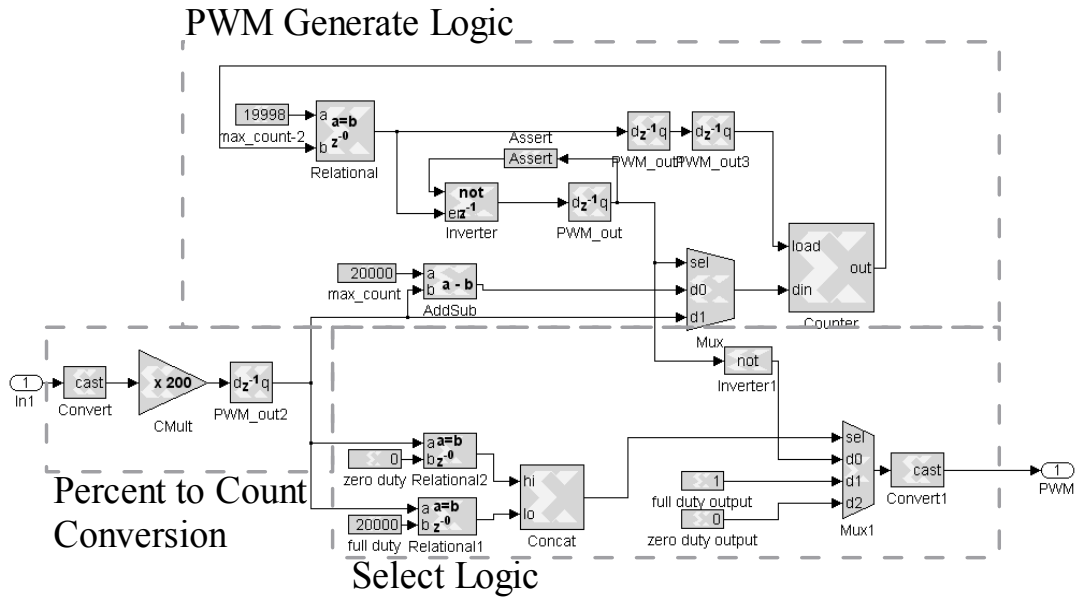
Figure 53: PWM Generator

A test was run for a setting of 15.25% duty for a 100Hz period, which will produce a 1.525ms pulse output. The generated wave form was stored to RAM at a 1us sampling rate and retrieved through the JTAG port. A check of the exact time, which the waveform was logic high, demonstrated the timing requirements were met. This subsystem was repeated twelve times to provide the control logic for each of the servos and combined to create a masked subsystem with the required user settings. The recorded waveform is presented Figure 54.

Figure 54: Generated PWM Output

89

6.8 FPGA RAM Data Acquisition Software Design

The JTAG port, while efficient when running the hardware under *Simulink* control, is unable to send information at most communication protocol rates. Therefore, a library subsystem was written in order to write data values to two single port RAM blocks and then read back these values at a much slower rate. The logic design is presented in Figure 55.
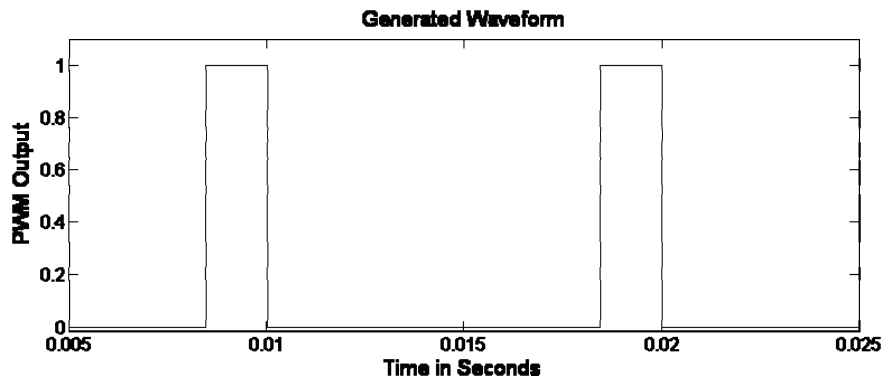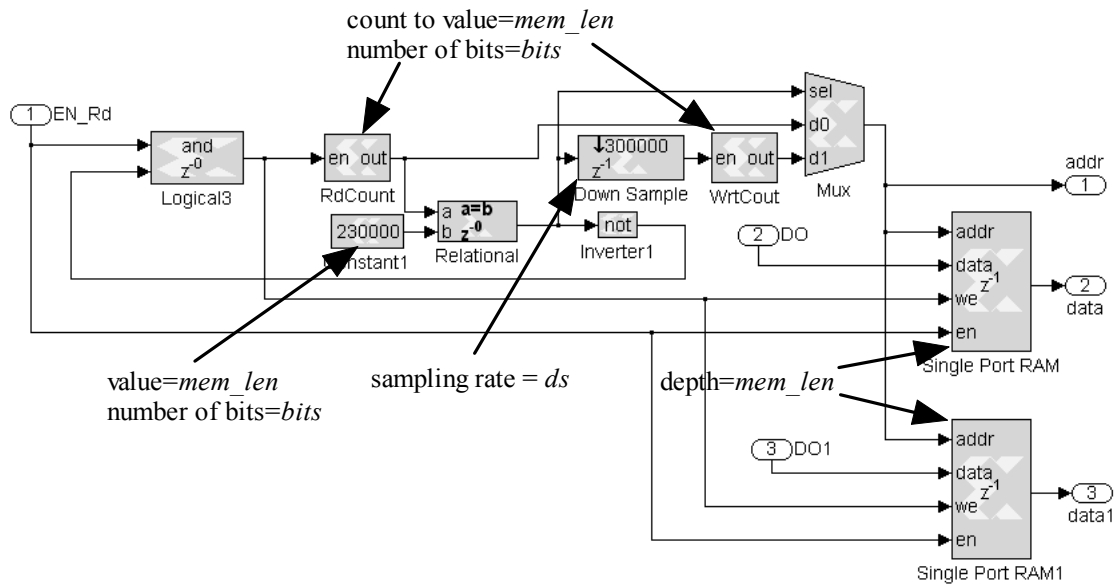

Figure 55: RAM Data Acquisition Logic

The subsystem logic input, *EN_Rd*, starts the read process under user control. There are two inputs, *DO* and *DO1*, for data acquisition. These values are saved to two separate RAM blocks at the update rate of the incoming signal. In order to synchronize the write process correctly, *EN_Rd* must have the same update rate as *DO* and *DO1*. A counter is incremented at the date input update rate in order to assign the memory location to both of the single port RAM blocks. When this counter has reached the final value, a second counter is enabled at a much slower rate in order to increment through the

90

memory locations during the read process. Both counter outputs are multiplexed before the RAM memory input in order to control the switching between the write and read processes.

Control logic was utilized in order to synchronize the selection of the multiplexer output and the write control line to the RAM blocks. Equation (13)

$$EN\_Rd \text{ and (not (write counter=final value))} \tag{13}$$

provides the logic for the enable to the write counter and the read/write select input to the RAM. Since the counter is disabled as soon as it reaches the final value, the output of this equation is held constant.

The multiplexer select and the read counter logic is simply a comparison of the counter value to the final count value. When the write counter reaches its final value the multiplexer output switches and the read counter starts the increment through the memory addresses. A down-sample block was placed just before the read counter enable input. Therefore, the counter rate is reset to a user selectable value. The outputs from the subsystem are the address, *addr*, the first set of data, *data*, and the second set of data, *data1*. These are sent to the JTAG port at the slower rate, during the read cycle, to be stored in *Simulink*.

The subsystem was masked with user inputs for three variables. The three variables are memory length of the RAM blocks, *mem_len*, the associated number of required bits, *bits*, and the down-sample rate for reading the RAM, *ds*. These variables are then entered into the blocks associated with them.

91

6.9   GPS Unit Communication Protocol

The SuperstarII GPS unit supplies latitude and longitude information. This
information is sent as a series of 8-bit values using the RS232 protocol. The TTL logic
level is selected as 3.3V. The library subsystems for receiving and down-sampling the
RS232 protocol are both set to 1900 baud. The SUPERSTARII block receives and
combines the 8-bit values into the correct format. The last block of the subsystem
prevents the information from being passed out of the subsystem if the correct CRC is not
received. Figure 56 displays the four lower level subsystems that comprise the system.
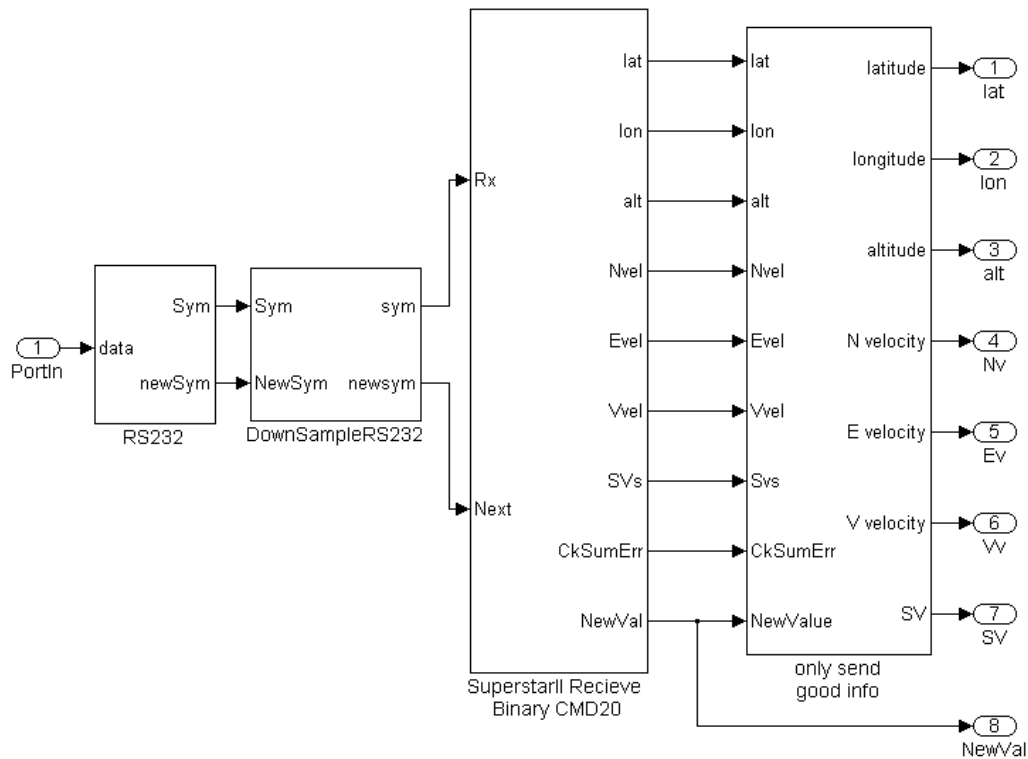

Figure 56: Receive Superstar II Library Block

The subsystem receiving the down-sampled 8-bit information utilizes a control
counter to keep track of the byte number of the received 8-bit word then combines the
information as necessary.

92

The SuperstarII GPS unit when set to send the latitude/longitude information in binary format, returns the values listed in Table 6, [51]. The subsystem contains a block for each piece of information received or each individual line in Table 6.

Table 6: GPS Information Formatting

Byte	Description	Units	Type
1-4	Header	N/A	Binary
5	Hours, Correction, Reserved	N/A	Binary
6	UTC Minutes	Minutes	Binary
7-14	UTC Seconds	Seconds	Double
15	UTC Day	Day	Character
16	UTC Month	Month	Character
17-18	UTC Year	Year	Unsigned Short
19-26	Latitude	Radians	Double
27-34	Longitude	Radians	Double
35-38	Altitude	Meters	Float
39-42	Ground Speed	Meters/Second	Float
43-46	Track Angle	Radians	Float
47-50	North Velocity	Meters/Second	Float
51-54	East Velocity	Meters/Second	Float
55-58	Vertical Velocity	Meters/Second	Float
59-62	HFOM	Meters	Float
63-66	VFOM	Meters	Float
67-68	HDOP	N/A	Unsigned Short
69-70	VDOP	N/A	Unsigned Short
71	Navigation Information	N/A	Binary
72	Bits 0-3 Number of SVs Bits 4-7 Coordinate System	N/A	Binary
73	System Mode Information	N/A	Binary
74	Elapsed Time	Hours	Character
75	Reserved	N/A	N/A
76-77	Checksum	N/A	Unsigned Short

Not all the information received from the GPS unit is passed out of the subsystem. The only information passed out of the subsystem is data necessary for simple navigation. This includes position readings, velocity readings and number of satellites available (*SV*). The subsystem calculates the CRC checksum as each byte arrives. The checksum format

93

requires that each of the bytes be combined, added, into a 16-bit word with overflow neglected. As each byte is summed, the addition block output is set to truncate the sixteen bits with no fractional representation. Each of the subsystems receives the partially added value, *CkSumIn*, and passes out the updated value to the next subsystem, *CkSum*. The final sixteen bits received is the checksum value sent by the GPS. The checksum is subtracted from the calculated value and compared to the value of 0 in order set an error flag out of the subsystem, *CkSumError*. A comparator is utilized to set a new value available flag, *NewVal*, when the counter reaches the value of 77. The blocks, which make up the receive subsystem are presented Figure 57. The control counter logic and each of the subsystems for combining the specific measurement values are contained in the lower level systems for clarity.
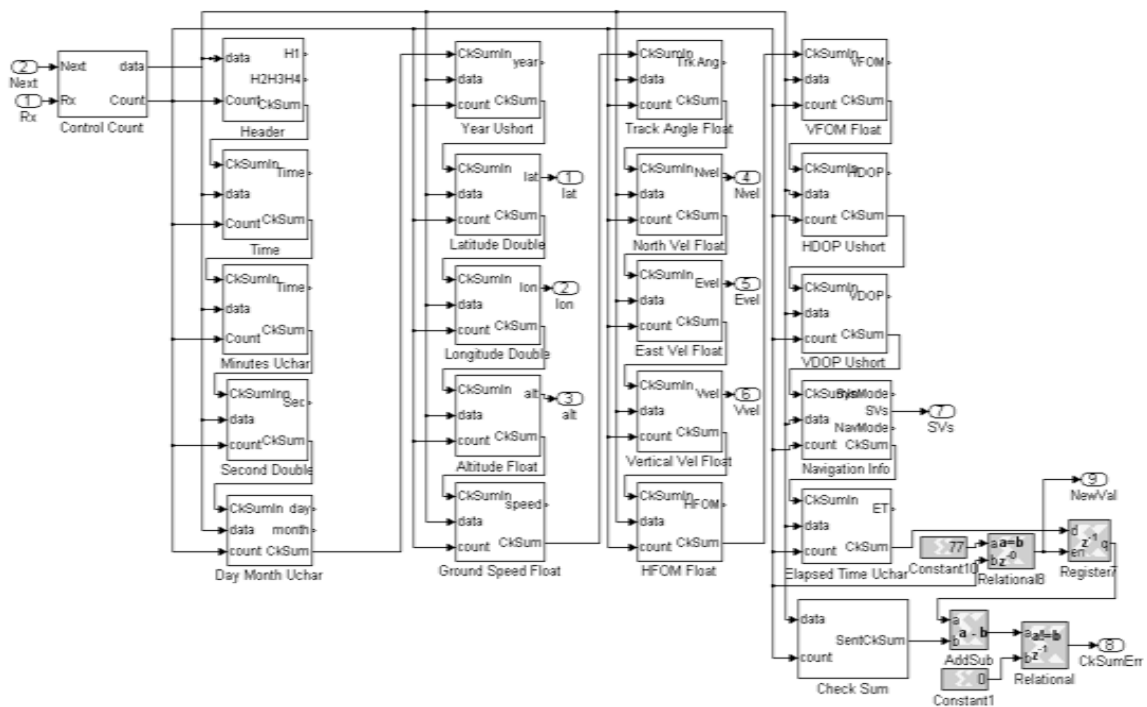

Figure 57: Superstar II Receive Subsystem

The control counter logic is presented in Figure 58. The first byte sent by the GPS unit is the beginning of the header and is equal to 1. A relational block is used to compare the incoming byte to this value. When the value is received a register is latched to one. Latching of the register to one enables the counter, which will synchronize storing of the received bytes.
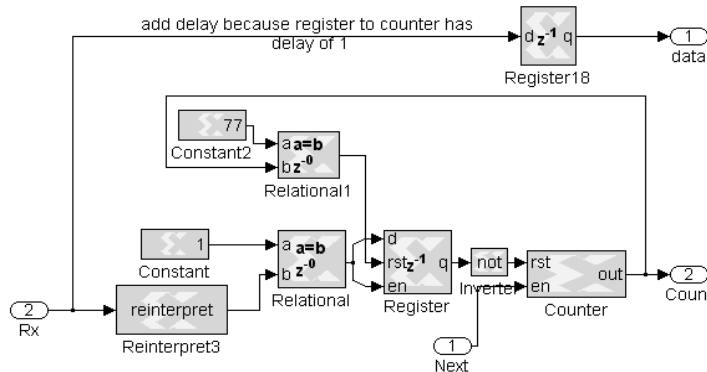
Figure 58: GPS Communication Control Counter Subsystem

The final subsystem prevents the output from updating when a checksum error occurs by utilizing a series of registers for each of the values sent out of the subsystem. This subsystem is presented in Figure 59. These registers are enabled when the checksum error, *CkSumErr*, is logic low and the new value flag, *NewValue*, has been set.
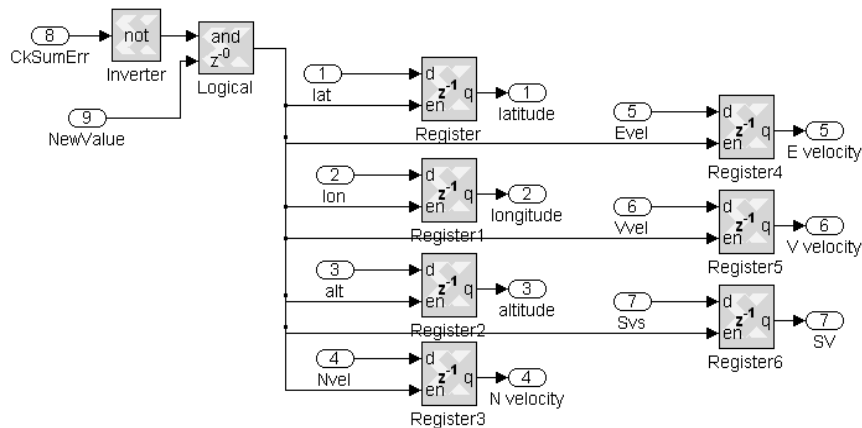

Figure 59: GPS Communication Subsystem Update Output Subsystem

95

6.10 IMU Unit Communication Protocol

The MicroStrain communication library subsystem is comprised of two lower level subsystems. The library subsystem for receiving RS232 protocol is set to a baud rate of 38400. The MicroStrain subsystem controls the initialization command requesting the stabilized Euler angles be sent continuously. After initialization, it combines each of the bytes into the correct format as the information is received. This subsystem also checks to see if the correct checksum value is received. If the checksum is correct, the output is updated. If the checksum is incorrect, the error flag, *cksumerror*, is set to one. The IMU protocol block is illustrated in Figure 60.
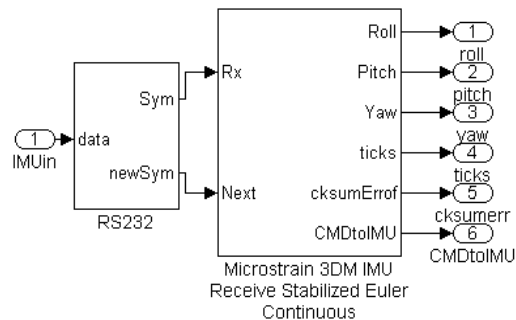
Figure 60: IMU Protocol Library Block

The subsystem receiving the 8-bit data from the RS232 subsystem is made up of lower level subsystems. The MicroStrain subsystem is presented in Figure 61. The control counter logic block synchronizes the receiving of the individual bytes by providing a count value for each one received. The send command subsystem sends the correct sequence for the requested information. The remaining subsystems combine the bytes making up each of the measurements received. Logic is included to sum the value of the bytes received in order to provide the calculated checksum. This value is

96

compared to the received checksum. If the correct value is received, the output registers are enabled and the output of the subsystem is updated.
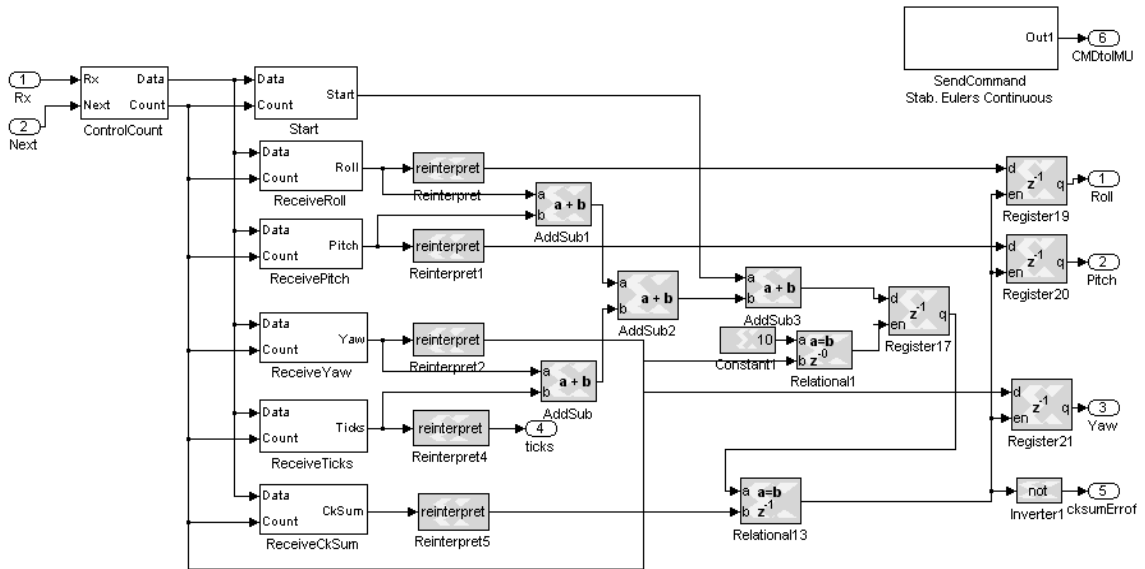
Figure 61: MicroStrain Receive Stabilized Euler Angles Subsystem

The subsystem providing the control count contains a counter, which is used to synchronize the reception of the individual bytes. The counter is enabled by latching a register to logic high when the first header byte is received, which is equal to 14. When the counter has incremented to the final count value of 11, the same register is reset back to the initial value of 0. Reset of the register disables the counter until the next block of data is received. The control count subsystem is presented in Figure 62.
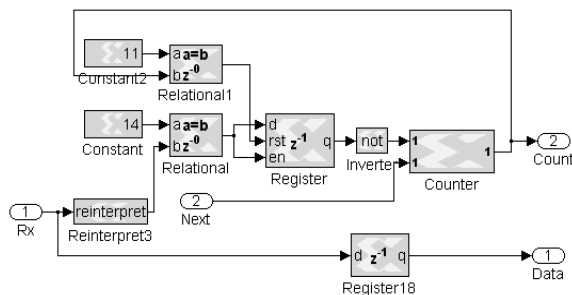
Figure 62: IMU Control Count Subsystem

97

The subsystem for sending the command sequence is presented in Figure 63. The subsystem uses a counter to synchronize the sending of three sequential 8-bit commands. The first value is equal to 16 and indicates to the IMU that a command is coming. The second value is equal to the value 0 and sets send to continuous mode. The third value is equal to 14 and requests the stabilized Euler angles to be sent. These values are held in individual constants, which are applied to a multiplexer. A slight delay is occurs after each value is sent. Therefore, the counter is increment to the value 6 at the baud rate and the output shifted right in order to produce the 0 to 3 count values required by the multiplexer select lines. The library block created for sending the RS232 protocol is utilized for sending the multiplexer output. This block is enabled by comparing the counter output to the required constants for sending each of the values.

Figure 63: IMU Send Command Subsystem

98

CHAPTER 7

RC-TRUCK IMPLEMENTATION
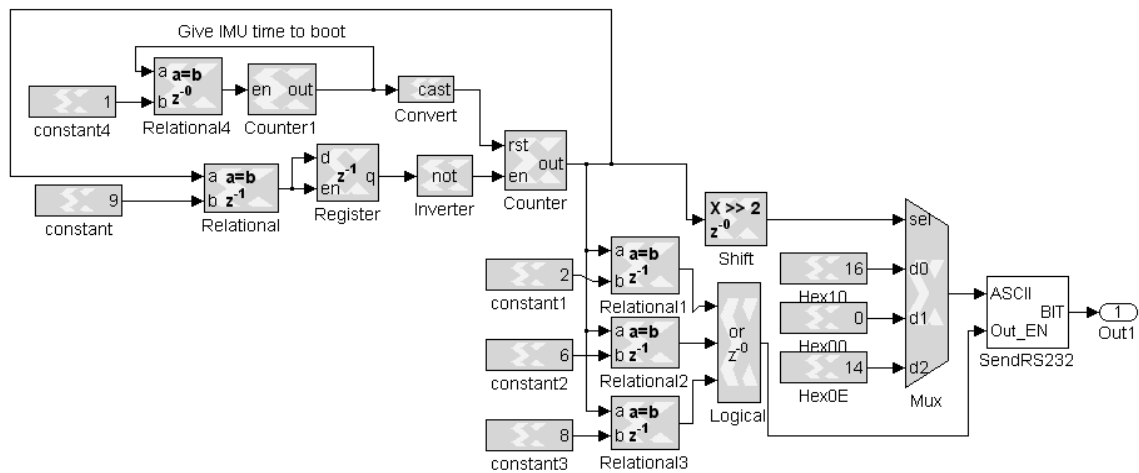
In 2007, Murthy developed a control system for an RC-Truck model. The model was converted to FPGA implementation using System Generator and verified through hardware-in-the-loop on a Xilinx Virtex II development board, [48]. In order to demonstrate the effectiveness of the developed autopilot platform, Murthy's research was emulated, by implementing the software design of this research on a similar RC-Truck robot utilizing the autopilot designed and developed in this research.

The simulation RC-Truck model was a simplified mass on wheels model that included a single motor equation and kinematic equations for Ackermann steering. The RC-Truck model is illustrated in Figure 64.
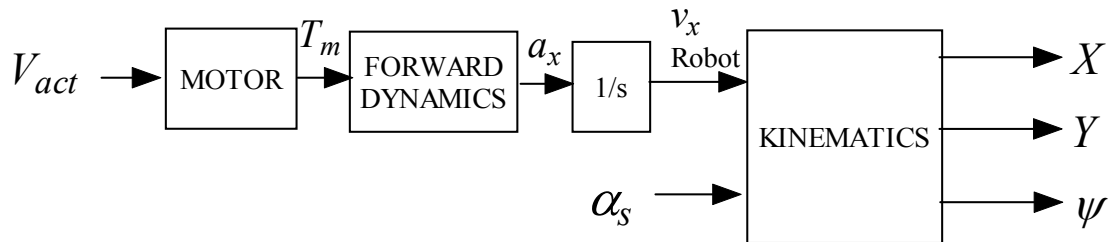

Figure 64: RC-Truck Model Block Diagram

V_{act} is the control input to the motor. T_m is the torque output of the motor. a_x is the forward acceleration along the body reference x-axis. v_x is the forward velocity along the body reference x-axis. X and Y represent the world reference position. ψ is the heading. α_s is the steering angle.

99

The control system consisted of a simple mission planner to send in a new way point when the robot was close to the current one. The control system was based on a PI controller for velocity and a P controller for the heading. The RC-Truck control system is presented in Figure 65.
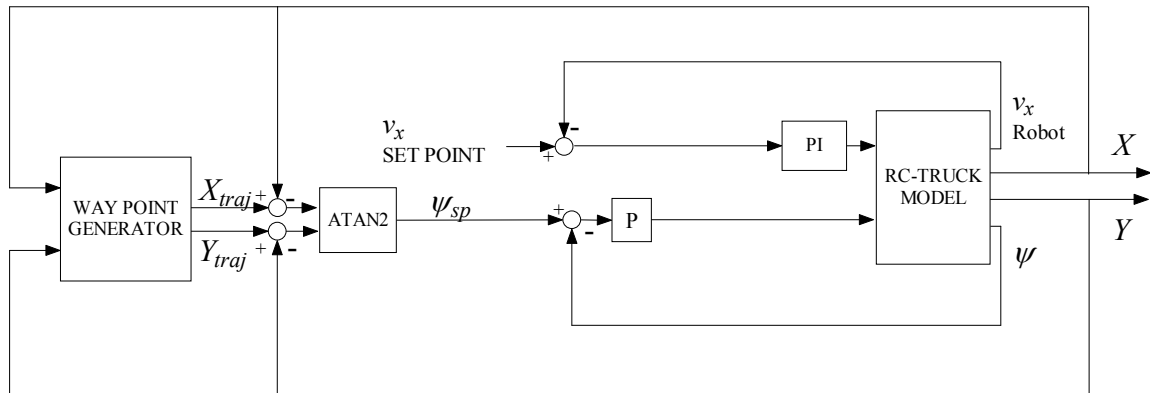
Figure 65: RC-Truck Control System

The hardware-in-the-loop verification, performed during Murthy's research, utilized various sensors. A 10 Hz IMU unit was used to provide the heading angle. An encoder was used to provide body reference velocity at 50 Hz. A 10Hz GPS was utilized for position. However, the conversion to latitude/longitude or ECEF reference frames was neglected, [48].

The sensor set utilized on the robot for this research did not include an encoder. The velocity was obtained at 5 Hz from the IMU in the North-East reference frame. There was an observed delay of 1 to 2 seconds. In addition, the IMU operated at a guaranteed minimum of 50 Hz. The Latitude/Longitude readings from the GPS unit arrived at 5 Hz. The *Simulink* implementation was modified slightly to incorporate the sampling rates. The sampling time and delay of the velocity readings was of particular concern. The modified RC-Truck control system is presented in Figure 66.

100

Figure 66: Modified RC-Truck Control System

The velocity PI controller was implemented with a *Simulink* PID controller block.
The proportional gain was set to 0.05, the integral gain to 0.001 and the derivative gain to
0. The proportional controller used for the heading control had a gain of one. This
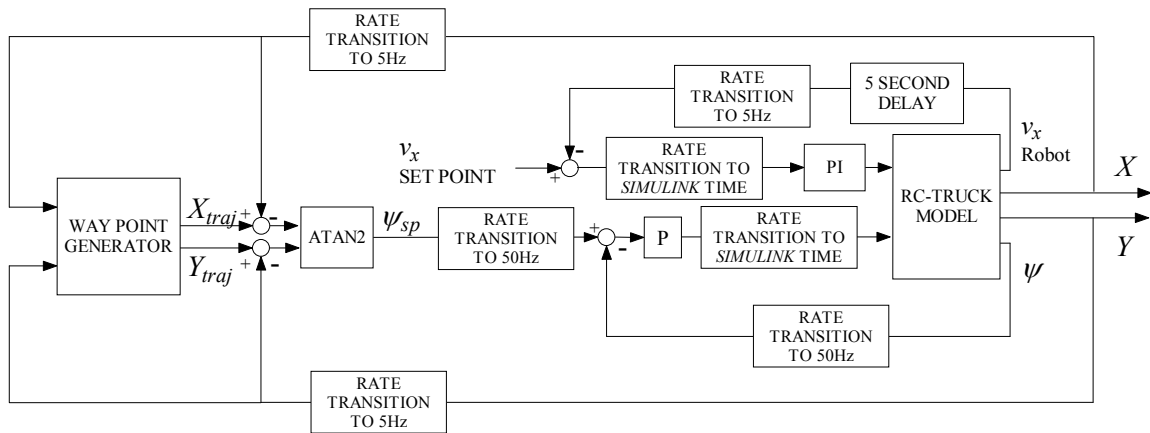caused the steering angle to equal the heading error. This angle was not limited in the
controller; rather, it was limited to +/- $\pi/6$ radians within the RC-Truck model. Including
the limitation within the RC-Truck model best reflected the behavior of the system. The
rate transition blocks were incorporated to reflect the sampling rates of the sensors. The
rate was transitioned back to the *Simulink* time step just before the PI controller and the
steering angle input to the RC-Truck model. *Simulink* required that the PI controller to be
run at the *Simulink* rate. A realistic mathematical representation of the system behavior
was obtained because the update of the error value entering PID block is limited to the
sensor rate.

The way point generator was contained within an m-file that assigned two vectors
of X and Y set points, calculated the Euclidean distance from the position set point in the
X and Y world reference frame and incremented the set points when the RC-Truck was
within one meter of the current set point. The code is presented in Figure 67.

101

Figure 67: *Simulink* Implementation of Way Point Generator

The velocity response was not ideal due to the slow sampling rate and delay. However, the controller remained stable for a set point of 1 m/s with the controller proportional value equal to 0.01 and the integral value equal to 10^{-5}. The velocity of the RC-Truck during simulation is graphed in Figure 68. The RC-Truck successfully reached each of the way points. The route established for the RC-Truck to traverse, with way points, is presented in Figure 69.
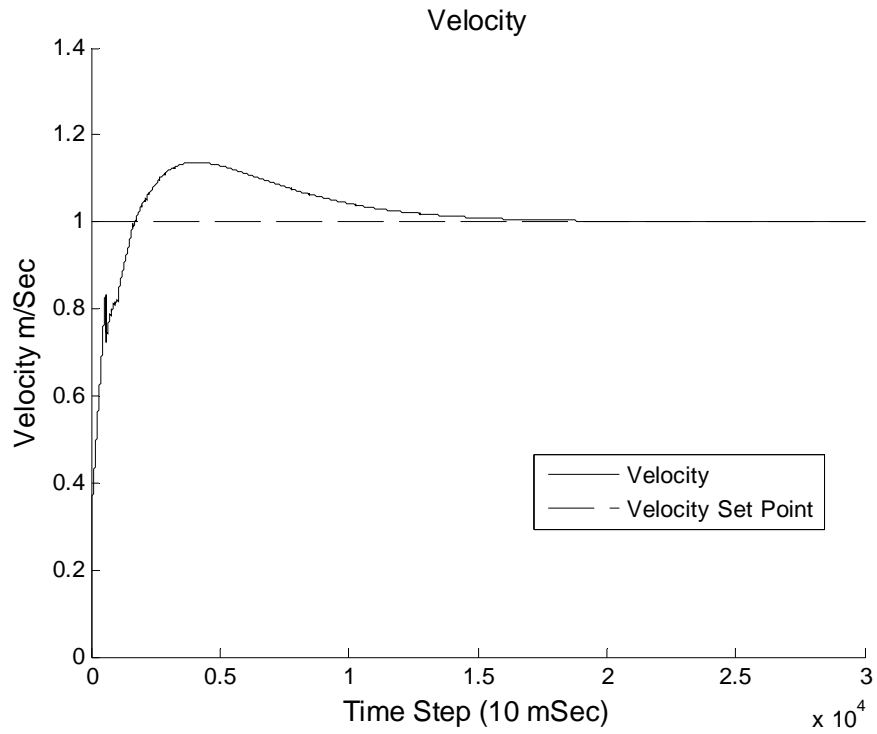
102

Figure 68: RC-Truck Simulation Velocity Output
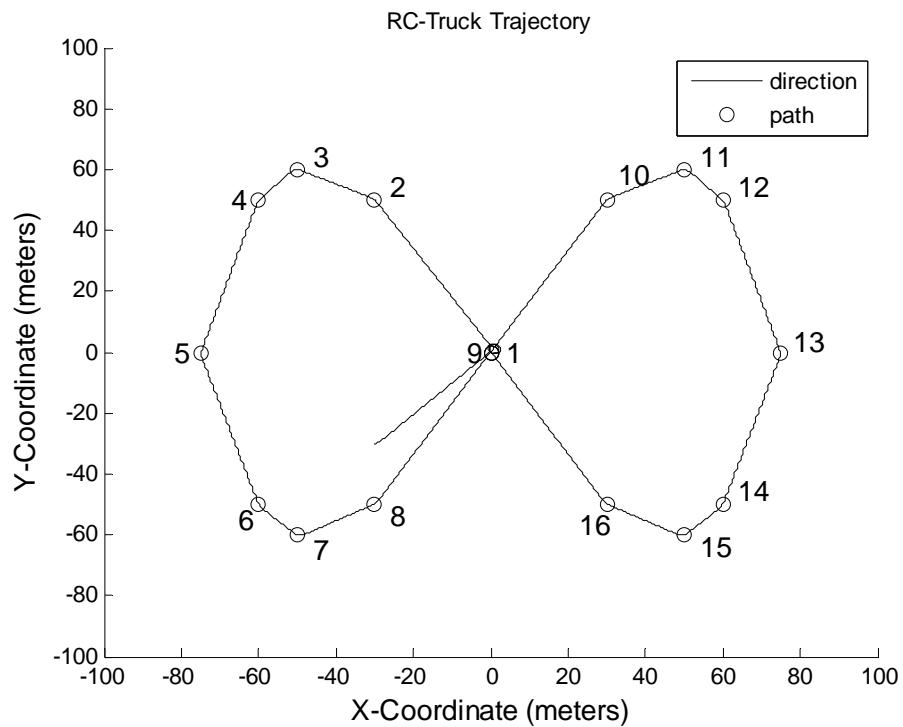
Figure 69: RC-Truck Simulation Heading and Position

103

7.1   RC-Truck Controller Design

The hardware implementation required further revision to incorporate the sensors, realistic operation of the hardware, and the math related to the latitude/longitude coordinate system. The Measurements from the GPS unit were received in 64-bit double representation for the latitude and longitude and 32-bit single representation for the North and East velocity measurements. The received values were converted to a binary representation with a fixed word length. Because the double and float precision reflects a much longer fixed word format, a limitation of the word length providing a sufficient resolution was incorporated into the logic design. Unlike with the simulated system, the velocity was rotated to the body reference in order to correctly implement the control of the RC-Truck motor. A block diagram of the hardware control system for the RC-Truck is presented in Figure 70.

The servo control was slightly more complex when implemented in hardware. The additional complexity was necessary to prevent damage to the servo controlling the steering angle. If the wheels were turned when the RC-Truck was stationary, there existed a potential for damage due to the additional force needed. In addition, the RC-Truck had to be stopped when the number of satellites used by the GPS unit dropped below five. This feature was also incorporated in the servo control.

The battery used to power the autopilot board, sensors and servos was Lithium Polymer. Therefore, full discharge could incur damage. Battery damage was prevented by monitoring the battery voltage with the FPAA and using an LED as a low voltage indicator. A 7.4V battery was selected. Since the voltage was approximately 8.5V at full

charge, the low battery flag was set for 7.4V, which provided for some additional time for the LED indicator to be observed visually by the operator.
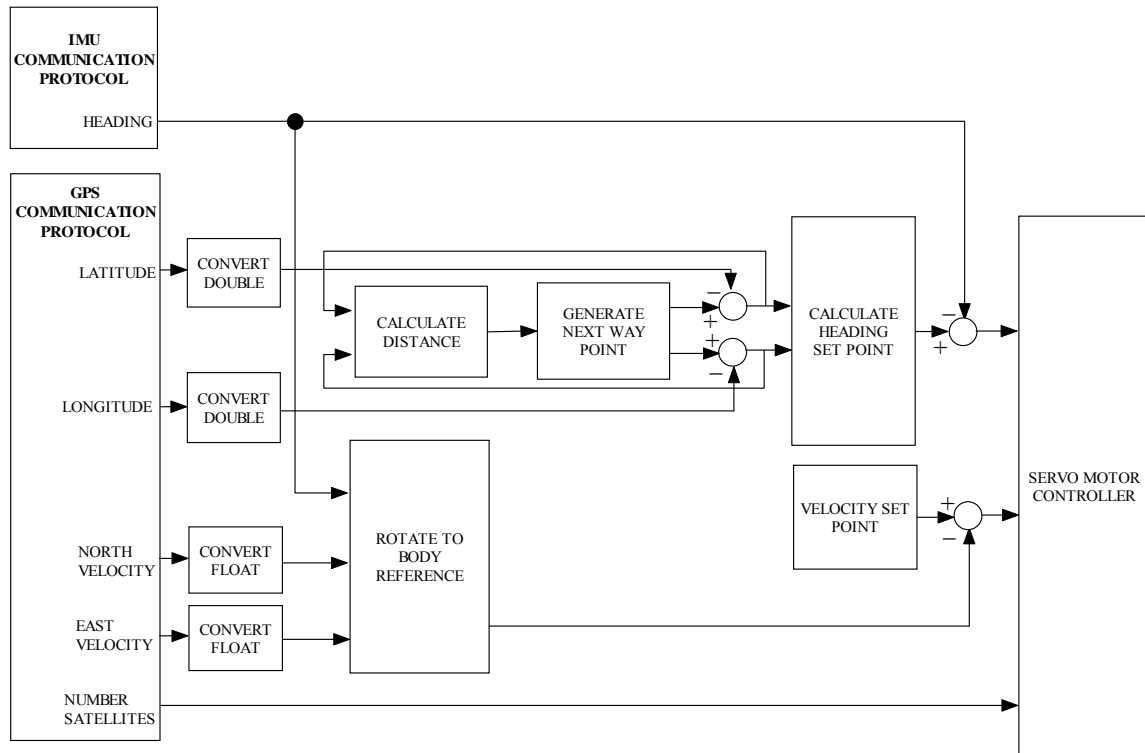
Figure 70: Hardware RC-Truck Control System

During the development of the control algorithms, it was necessary to collect both measurements and calculated values while the vehicle was in motion. This provided for the observation of these values with respect to the robot's behavior at that moment. The JTAG is limited in how much data can be received into *Simulink* for each time step. This was due to the limitations of the parallel port integration meeting the timing constraints of the *Simulink* program. In order to overcome this, a subsystem was built that converts the information into ASCII format and then sent the information to one of the autopilot's TTL output ports. In order to receive the information through the computer's USB port, Acroname's TTL to USB converter was utilized. The received information was observed

105

using Window's HyperTerminal program. The HyperTerminal program also allowed for the incoming data to be stored to a text file for further analysis.

RC-Truck platform was provided by the Army Research lab. This platform included the servos, motor speed controller and motor. A wood box was fabricated to house the autopilot platform and mounted to the metal framing towards the rear of the RC-Truck. The required Superstar II GPS receiver, antenna and power supplies were mounted on a wood platform attached to the top metal framing on the vehicle. The MicroStrain IMU was mounted directly to the metal frame on the back of the RC-Truck. The location of the IMU prevented the magnetic field generated by the drive train motor located at the center of the vehicle from corrupting the measurements.

The motor was powered from a single 7.4V Polymer Lithium battery. The same type of battery was also utilized to power the autopilot, servos and sensors. The autopilot and GPS required a 5V regulated input. In order to meet this requirement, a circuit containing two voltage regulators and heat sinks was included in the hardware implementation. The servos did not require a regulated voltage, but were limited to a maximum of six volts. For this reason, the servos were also powered from the regulated five volt supply. In order to guarantee enough current, the voltage supply to the electronics was divided between the two regulators. The IMU is powered directly from the same 7.4V battery. The RC-Truck is presented in Figure 71.
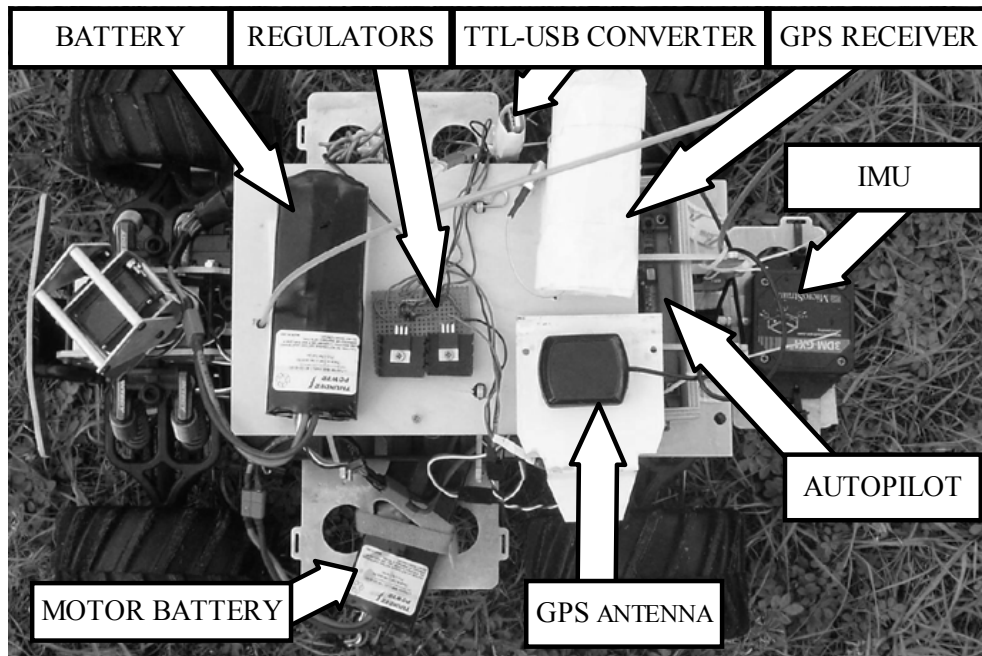
Figure 71: RC-Truck with Sensors and Power Supply

7.1.1 ASCII Data Collection

The speed and word length that can realistically be sent through the JTAG is limited by both the parallel port and *Simulink*'s integration with Windows. For this reason, a subsystem was developed to convert the binary values to decimal ASCII representation. The ASCII values were then sent utilizing RS232 protocol. This allowed a TTL-USB converter to be utilized to receive the data from the computer's USB port through Window's HyperTerminal program. The algorithms for converting binary to ASCII were designed for a specific representation. In order for sufficient resolution, the subsystem required a 40-bit word length, with the lower twenty-nine bits representing the fractional portion. The information was sent sequentially with coma inserted before each value. A line-feed character was the last character to be sent before the subsystem was reset for the next eight measurement values. The additional characters provided the

107

necessary delimitation when stored into a text file through the HyperTerminal program.
Each of the ASCII characters was then sent to the library subsystem, SENDRS232 in
order to rotate the bits out to a TTL port. The design is presented in Figure 72.
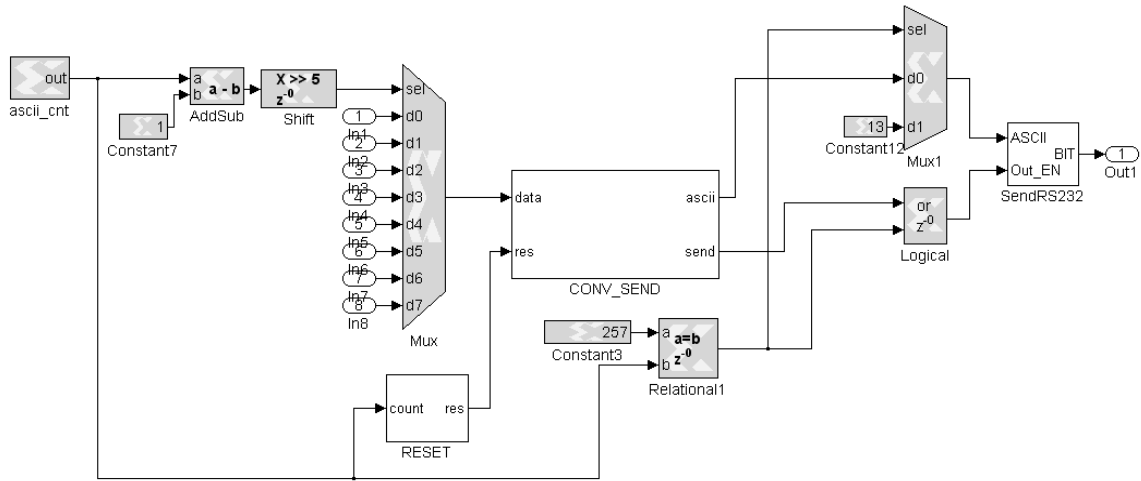
Figure 72: Send ASCII Subsystem

A counter was set increment at $1.042(10^{-3})$ seconds, which was the required
timing for sending each ASCII character for a baud rate of 9600 bits/sec. The count
value was utilized for synchronizing the sending of the required eight measurement
values and the delimitation characters. The final count value was based on the time it
took all of the ASCII characters to be sent. The eight measurement values were
multiplexed just before the subsystem which inserted the coma and converted the binary
value to ASCII characters, CONV_SEND. The select input to the multiplexer was
controlled by the count value. Equation (14)

$$sel = (count - 1) / 2^5 \tag{14}$$

108

was implemented to convert the count value to the required multiplexer select input of 0 to 6. After each of the measurement values were sent, the CONV_SEND subsystem was reset for the next character. The reset logic in equation (15)

$$(count = 32)OR(count = 64)OR(count = 96)OR(count = 128)OR...$$
$$(count = 160)OR(count = 192)OR(count = 224)OR(count = 255) \tag{15}$$

was contained in the RESET subsystem. When the count was equal to the value 257, the last value had been sent and a second multiplexer was utilized to send the value 13, which is equivalent to the line-feed character in ASCII.

The CONV_SEND subsystem contained a counter block utilized as a timer to synchronize the sending of the ASCII characters. The characters sent by this subsystem included a starting coma followed by a plus or minus sign, the ASCII characters representing the decimal form of binary value to be sent, with the decimal point character inserted before the fractional portion. The counter was reset by the external control counter after the last character had been sent. The first bit of the value received was the sign bit which controlled a multiplexer. The multiplexer selected between converting the binary value to the ASCII characters, for a positive number, or the negated binary value, for a negative number. The subsystem is presented in Figure 73.

109
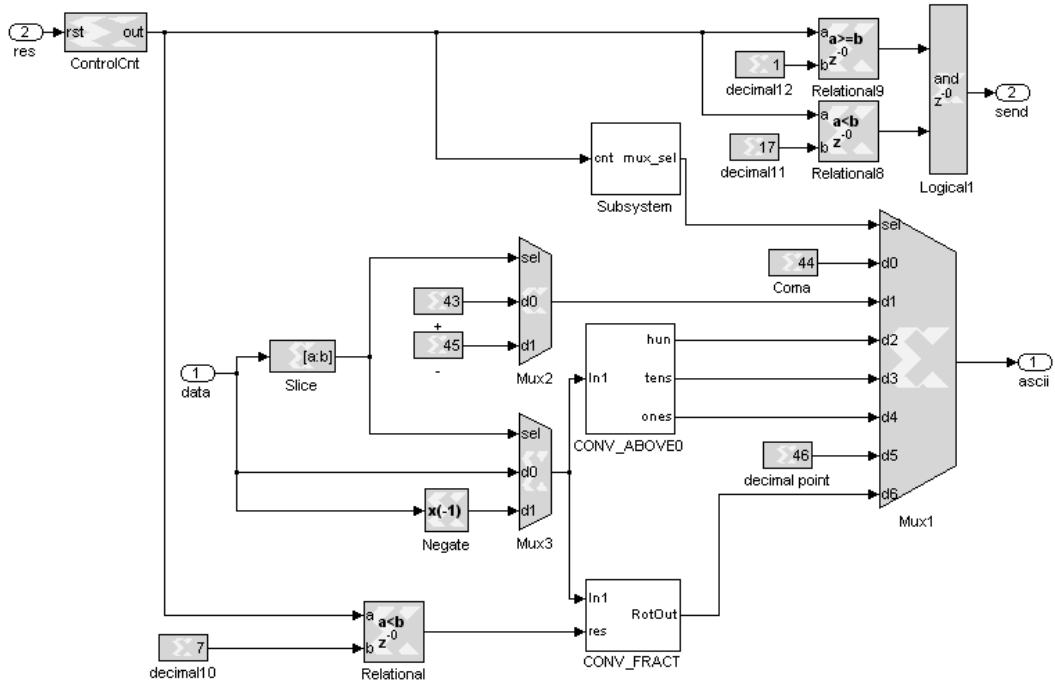
Figure 73: Convert to ASCII Subsystem

The CONV_ABOVE0 subsystem calculated the characters which represented the decimal hundreds value, the tens value and the ones value. The 8-bit values were calculated with three sequential mathematical operations. Equation (16)

$$hvalue = \text{CAST}(bin / 100) \qquad (16)$$

calculated the decimal hundreds placeholder, *hvalue*, directly. The variable *bin* is the highest 11-bits of the 40-bit binary value to be converted to ASCII. The CAST operator represents the system generator cast block which was set to define the output as an 8-bit value with the truncate option set. Equation (17)

$$tvalue = \text{CAST}((bin - hvalue) / 10) \qquad (17)$$

110

used the output from the first calculation, *hvalue*, and *bin* to calculate the decimal tens

placeholder. Equation (18)

$$ovalue = \text{CAST}(hvalue - tvalue) \tag{18}$$

was the final calculation that results in the decimal ones placeholder, *ovalue*. In order to

obtain the ASCII character, these values were then OR'd with the value of forty-eight.

This set the highest four bits to the required ASCII sequence of '0011'.

The CONV_FRACT subsystem utilized a feedback loop to calculate a sequential

series of division by ten. This sequential division was calculated at ten times the

communication baud rate, as required by the subsystem sending the ASCII characters.

Each division by ten resulted in shift of the fractional value by one decimal place holder

to the decimal ones placeholder.   The value resulting from this calculation was then

converted to an 8-bit value and OR'd with the value forty-eight to obtain the ASCII

character. After the last character was sent, a comparison to the value of the counter

controlling the CONV_SEND subsystem was used to reset the subsystem. The

CONV_FRACT subsystem is presented in Figure 74.
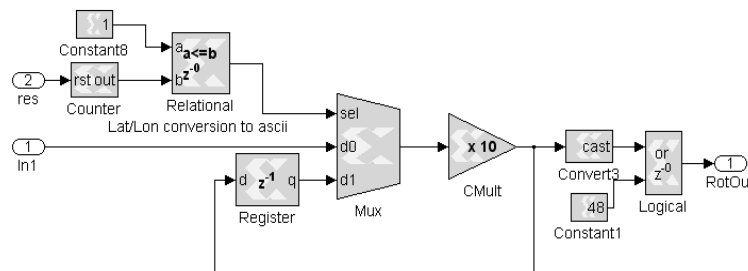
Figure 74: Convert Fraction to ASCII

111

7.1.2 Battery Monitoring Design

The FPAA IC was utilized for monitoring the battery voltage. The battery voltage was inputted directly to the FPAA large signal input. The FPAA was programmed for an internal gain of negative two. The A/D output was assigned to the FPGA port connected to *data1* in the autopilot template's FPAA_INPUT subsystem. A negative gain was utilized because the A/D utilized twos compliment formatting. By utilizing the negative gain, the output was directly inputted as 0 for an internal voltage equal to -1.5 and 255 for an internal voltage equal to +1.5. The AnadigmDesigner2 environment is presented in Figure 75.
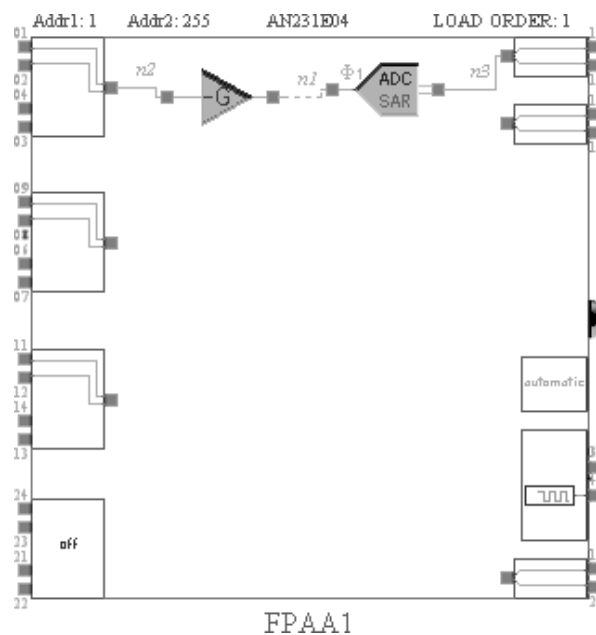
Figure 75: FPAA Program for Battery Monitoring

The FPAA configuration was saved as a binary file in the folder containing the *Simulink* autopilot program. The binary file was then converted to a variable and assigned within the mask of the autopilot template's PROGRAM_FPAA subsystem. The

112

output of the autopilot template's FPAA _INPUT block was connected to a variable for comparison to the A/D input equal to 7.4V which was equal to 198. Equation (19)

$$flag\_set = \left(\frac{7.4}{8.96} + 1.5 \right) \frac{2^8 - 1}{3} \tag{19}$$

provided the conversion from the 7.4V input to the binary value outputted by the FPAA A/D. The value 7.4 was the battery voltage that the flag was set for. The 8.96 was the conversion due to the voltage division ahead of the FPAA input. The value 1.5 was added to this value because of the voltage offset within the FPAA. The $(2^{8-1})/3$ was the conversion factor from the voltage seen at the input to the A/D converter to the 8-bit binary value entering into the FPGA. The logic used for the battery monitoring is presented in Figure 76.
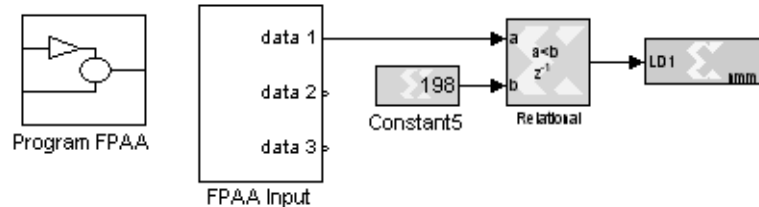
Figure 76: Program for Battery Monitoring

7.1.3   Double and Float Conversion to Binary

The formatting of the measurements received from the GPS unit were 64-bit double precision for latitude and longitude measurements and were 32-bit single point precision for North and East velocities. Both these formats required conversion into a fixed binary word length in order to be utilized within standard System Generator blocks.

113

The binary word representation for the double and single precision formats is presented in Figure 77. The only difference between the two representations is the number of bits contained the exponent and the fraction. The sign for both formats is represented with one bit, with a value equal to one representing a negative number. The exponent is eight bits for single precision and eleven bits for double. The fraction is twenty-three bits for single precision and fifty-two for double.
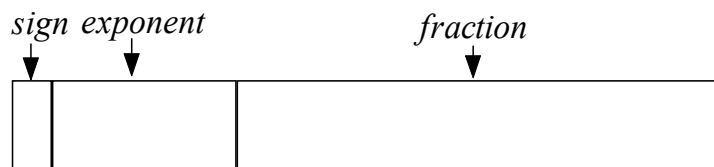
Figure 77: Single and Double Representation Word Format

The algorithm for conversion between the single and double precision formats and the binary fixed word length is obtained in two mathematical steps. First, equation (20)

$$e = exponent - bias \tag{20}$$

results in an intermediate variable, *e*, from the value contained in the received *exponent* bits. The variable, *bias*, is an offset utilized in the single and double precision formatting and is equal to 127 for single precision and 1023 for double precision. Second, equation (21)

$$magnatude = 2^e * 1.fraction \tag{21}$$

114

calculates the magnitude of the single and double precision value represented as a fixed binary word length, *magnatude*.

The single precision and double precision to binary conversions could not be implemented within a single subsystem. This was due to the difference in word length requiring different values within the logic design. The algorithms for each conversion were very similar. A shift block was utilized to perform the 2^e calculation. In order to obtain a reasonable word length, the output was limited to forty-five bits, with forty bits representing the fractional portion, for the latitude and longitude measurements and to thirty-two bits, with sixteen representing the fractional portion, for the velocities.

The value received from the GPS unit was broken up into three slices, the sign bit, the *exponent* and the *fraction* values. This separation provided for the mathematical calculation given in equation (20) and equation (21) to be performed.

After equation (20) is implemented the resulting value, *e*, may be either a negative or positive value, depending on the size of the numeric value received. A shift right was required for a positive result, while a shift left was required for a negative result. In order to accommodate the different logic requirements, a multiplexer block was utilized to determine the sign and allow the correct calculation to be passed to the next stage of logic.

The slice containing the *fraction* value required a one to be added to the value received. The most efficient implementation was to utilize a System Generator concate block in order to place a bit equal to 1 in the highest order of the output. The reinterpret block was then utilized to assign this additional bit as the value 1 with the rest of the bits assigned to the fractional portion. This provided the required format for the

115

multiplication given in equation (21). In order to limit the delay and number of gates utilized by the multiply block, a cast block was used to reduce the number of bits. For the latitude and longitude measurements, the value was limited to forty-one bits with the lowest forty representing the fractional portion. For the velocity measurements, the value is limited to nine bits, with the lowest eight representing the fractional portion.

Because the calculated magnitude is always positive, a multiplexer was utilized to select between a negative and a positive measurement being outputted by the subsystem. The calculated magnitude was directly connected to the multiplexer's output selected by the value 0, while the negated magnitude was connected the multiplexer's output selected by the value 1.   The selected output of the multiplexer was directly determined by the sign bit of the value received from the GPS.

The subsystem for converting from double precision to binary is presented in Figure 78. The single precision to binary design was identical with exception to the numeric values dependant on the word lengths.
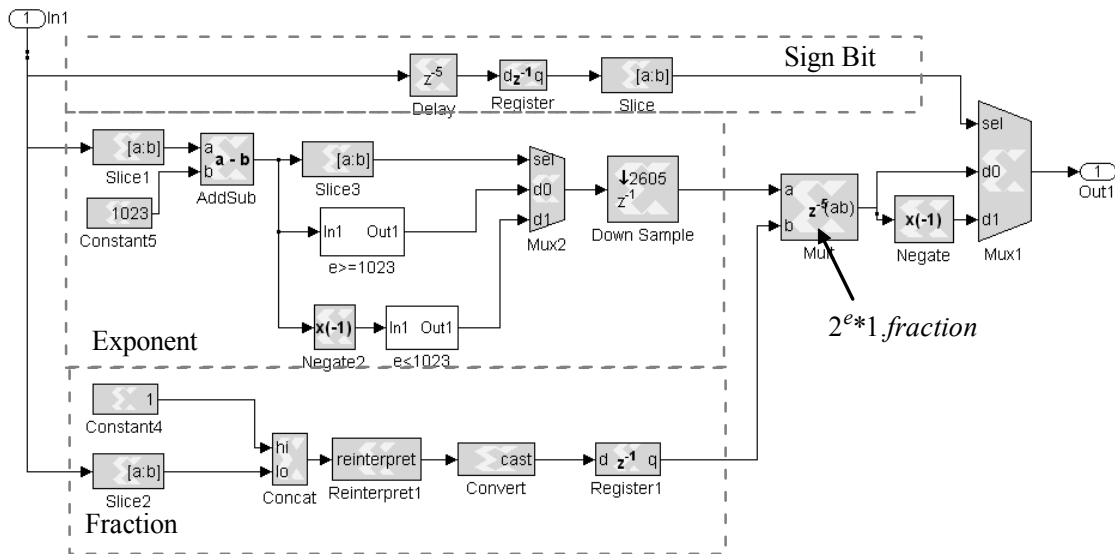
Figure 78: Double to Binary Conversion

116

7.1.4 Heading Set Point Control

The heading set point control included an approximation of the distance, an m-file to store and increment to the next way point, and a calculation of the heading set point. In addition to the code for incrementing the way points, a multiplexer was included with the user input switch utilized as a select. This allowed the m-file to be reset to the first way point by the operator. The heading control subsystem is presented in Figure 79.
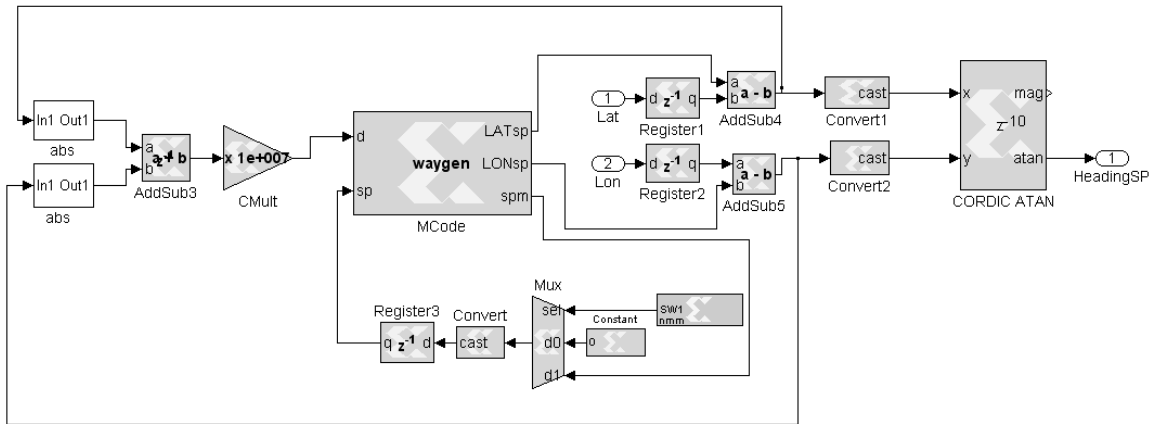
Figure 79: Heading Set Point Control Subsystem

The values for latitude and longitude received from the GPS unit were given in radians and a very small value represented the difference between waypoints. A large word length would have been required to accurately calculate the Euclidean distance. The calculation would, not only require a large amount of logic, but also create a significant latency within a feedback loop. For this reason, equation (22)

$$d = 10^7 \left(\left| lat_{sp} - lat \right| + \left| lon - lon_{sp} \right| \right)$$

(22)

117

calculated an approximation of the distance, *d*. The value of *d* that determined the advancement to the next way point was set to the value 1. The multiplier given in equation (22) was treated as a tuning parameter and determined through experimentation.

An m-file was utilized to store two vectors of way points, where one vector represented latitude and the other longitude. As the vehicle neared the current set point, the index assigning the values for the latitude and longitude set points was advanced. An additional comparison to the set point index was also included to hold the index at the final way point value. The m-file code is given in Figure 80.


```
1      function [LATsp,LONsp,spm]=waygen(d,sp)
2
3      % persistent defines a variable that holds its value between time steps
4      % this variable must be defined by type (xlSigned = signed), bits and
5      % binary point (type, bits, binary)
6
7      % Vectors containing way points
8  -   persistent LONspv, LONspv=xl_state([-1.438430416697306,...
9        -1.438431045015837,-1.438432196933144,-1.438433663009715,...
10       -1.438434046982151,-1.438434780020437,-1.438435792311403,...
11       -1.438437101308342,-1.438437921613090,-1.438437834346627,...
12       -1.438436856962246,-1.438435600325185,-1.438434622940804,...
13       -1.438433942262396,-1.438433174317525,-1.438431970040341,...
14       -1.438430940296082,-1.438430416697306,...
15       -1.438430416697306],{xlSigned,45,40});
16 -   persistent LATspv, LATspv=xl_state([0.489695615251969,0.489694882213683,...
17       0.489694637867588,0.489694986933438,0.489696330836962,...
18       0.489697168595003,0.489697692193779,0.489697552567439,...
19       0.489696819529153,0.489695510532214,0.489694986933438,...
20       0.489695178919656,0.489695562892091,0.489696138850744,...
21       0.489696671889030,0.489697168595003,0.489696959155493,...
22       0.489696627542935,0.489695615251969],{xlSigned,45,40});
23
24 -   if d<1 && sp<18
25 -       spm=sp+1;
26 -   else
27 -       spm=sp;
28 -   end
29
30 -   LONsp=LONspv(spm);
31 -   LATsp=LATspv(spm);
```

Figure 80: Hardware Way Point Generator M-File

118

Equation (23)

$$E_{error} = longitude - longitude_{sp} \qquad (23)$$

calculated the error in the East direction, E_{error}. The sign reversal was required because

the longitude values were negative and decreasing for movement in the East direction.

Equation (24)

$$N_{error} = latitude_{sp} - latitude \qquad (24)$$

calculated the error in the North direction. Equation (25)

$$Heading_{sp} = \tan^{-1}\left(\frac{E_{error}}{N_{error}}\right) \qquad (25)$$

calculated the heading set point, $Heading_{sp}$, from the North and East errors; where a

CORDIC inverse tangent block was used to implement the inverse tangent function. The

block provided the quadrant with the output given from $-\pi/2$ to $\pi/2$. The formatting of

the value received from the IMU was the same. When the vehicle was aligned South the

measurement changed from $-pi/2$ to the $pi/2$ value for a small change in heading. This

sign reversal created errors in the controller. For this reason, an additional subsystem

was inserted just before the heading controller to modify these values to a 0 to 2π

representation. A multiplexer and comparator was used to implement an if-then

119

statement. If the input, *AngleIn*, was negative then 2π was added to the value, if the input was positive, then the output was not adjusted. The subsystem is presented in Figure 81.
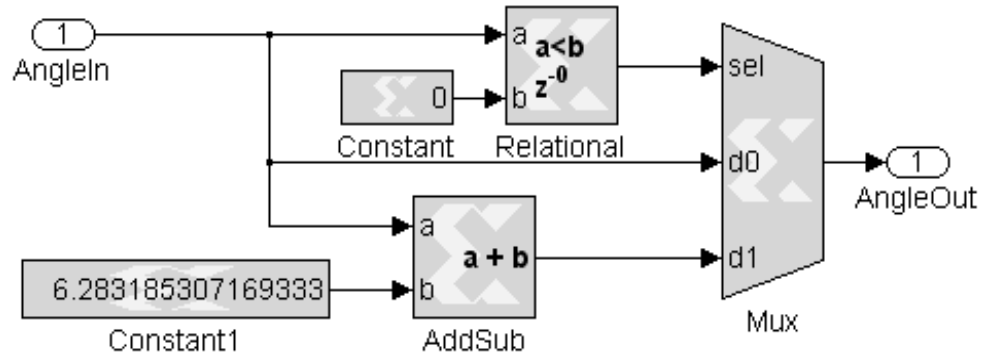
Figure 81: Heading Correction Subsystem

7.1.5 Velocity Set Point Control

As with the simulation, the velocity set point was a fixed 1m/s, but with additional control to hold the set point to 0 when the RC-Truck was under the control of the handheld radio. If a set point of 1m/s was allowed when the RC-Truck was held stationary by the radio, an error was present. This created an increasing control effort at the output of the PI controller. When the RC-Truck was finally allowed to enter the autonomous mode, this control effort created a sudden increase in motor RPMs. By synchronizing the set point to change to the required 1m/s to the switch to autonomous mode, the velocity control system presented the required step response. The set point design is presented in Figure 82. The SS_IN block was an FPGA input port connected to an output port in the Safety Switch. This port was logic high for manual control and controlled the multiplexer that selected between the two set points.

120
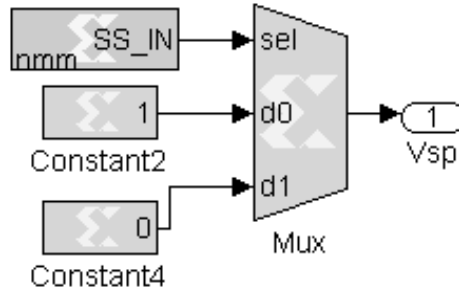
Figure 82: Velocity Set Point Subsystem

7.1.6 Servo Control

The servo control incorporated the velocity controller, the heading controller, and two m-files. The m-files were implemented directly before the duty cycle inputs to the PWM generator for the steering and the drive train servos for additional control. The m-file that provided additional control to the steering servo prevented a change in the steering angle when the RC-Truck is stationary and when the number of satellites has dropped below five. This was necessary to prevent potential damage to the servo caused by the additional torque generated when the RC-Truck is at a standstill. The m-file providing additional control to the drive train servo set the duty cycle to neutral when the number of satellites is below five. This m-file also included a logic output to an onboard LED to inform the operator when the vehicle had stopped due a lost GPS lock. Since it was always possible that the control effort would exceed the limits of the servos, both m-files included code to limit the duty cycle to an acceptable range. A block diagram of the servo control design is presented in Figure 83.

121
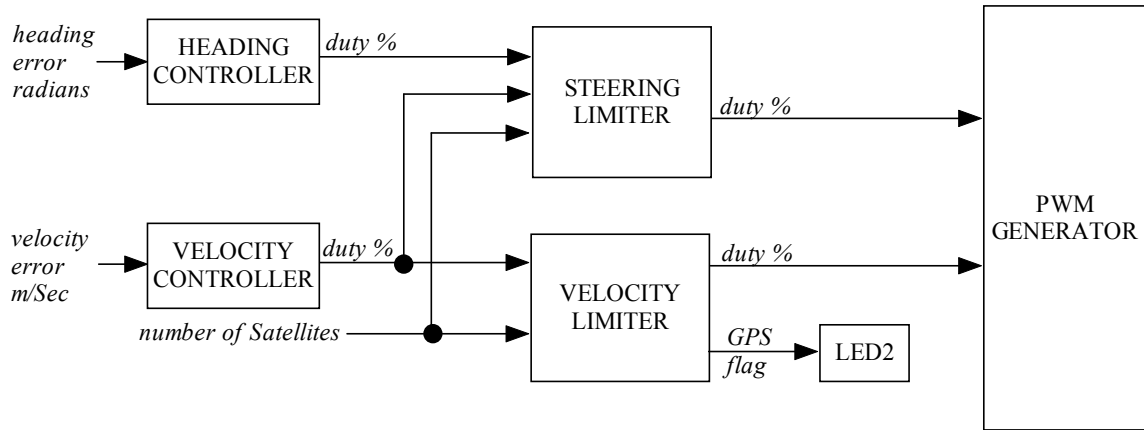
Figure 83: Servo Control Block Diagram

The heading controller utilized a proportional controller with a gain equal to one. The multiplier block was not implemented, since the output would simply equal the input. The heading error required a conversion to the 0 to 100% duty cycle. The duty cycle required for a steering angle of $-\pi/6$ radians was approximately equal to 5.5%. For a steering angle of $+\pi/6$ radians the duty cycle was approximately 9.5%.   These values were determined experimentally by slowly increasing and decreasing the duty cycle while observing the resulting angle of the wheels. Based on this mathematical relationship, equation (26)

$$duty = \frac{angle*12}{\pi} + 7.5 \tag{26}$$

implemented the conversion from radians to duty cycle was implemented through.

As with the simulated system the velocity control was implemented with a PI controller, but slightly different gain values. The gains were modified because, unlike the simulation model, the drive train motor was being driven from the PWM signal, rather

122

than the motor voltage. In addition, the characteristics of the motor were not exactly known, which created a difference in the behavior of the simulated motor and the RC-Truck motor. Because of the inherent stability of the system, the gain values were easily found by experimentation. The proportional gain was equal to 0.035 and the integral gain equal to $3.5(10^{-5})$. Because neutral, where the RC-Truck is not was motion, was equal to a duty cycle input of 7.5%, the output from the controller was subtracted from this value. The calculation implemented subtraction rather than addition due to the inverse relationship of duty cycle to motor control. A duty cycle input of less than 7.5% resulted in forward motion, while a duty cycle of greater than 7.5% resulted in reverse motion.

The m-file providing additional control to the velocity servo, VelocityLimiter.m, was designed with two if-then-else statements in order to adjust the percent duty cycle output, *PWMout*. *PWMout* was set to neutral when the *SV* input, which provided the number of satellites used by the GPS unit, dropped below five. The first if-then-else statement adjusted the output to the lower and upper bounds if the control effort exceeded the limits of the servos. The second if-then-else statement set a logic output, *SVout*, to true when the number of satellites dropped below five. The output was connected to a user LED as a flag so the operator was able to observe if the vehicle was stopped due to a loss of GPS. The m-file code is given in Figure 84.

The steering limiting m-file, SteeringLimiter.m, was designed to prevent the duty cycle controlling the steering servo from updating when both the velocity duty is set to neutral and when there are less than five satellites in view. In addition, the duty cycle input, *PWMin*, was limited to the maximum or minimum allowed values. In order to check each of these conditions, an if-then-else statement is utilized. If the number of

123

satellites was adequate and the drive train duty cycle was set for forward motion, but the maximum or minimum value was exceeded then the duty output, *PWMout*, was re-assigned the maximum or minimum value, respectively. The m-file containing this code is given in Figure 85.

```matlab
function [PWM,SVout] = PWMcontrol2(PWMin,SV)
Pin=xfix({xlUnsigned,14,7},PWMin);
SVin=xfix({xlUnsigned,4,0},SV);
SVout=xfix({xlBoolean},1);
% prevents reverse
if Pin > xfix({xlUnsigned,14,7},7.6)
    PWMout=xfix({xlUnsigned,14,7},7.6);
% Sets upper speed limit to a duty of 5.6%
elseif (Pin < xfix({xlUnsigned,14,7},5.6)) && ...
                (SV > xfix({xlUnsigned,4,0},4))
    PWMout=xfix({xlUnsigned,14,7},5.6);
% acceptable range
elseif (xfix({xlUnsigned,14,7},5.60) < Pin) && ...
        (Pin < xfix({xlUnsigned,14,7},7.6)) && ...
        (SV > xfix({xlUnsigned,4,0},4))
    PWMout=Pin;
% all others set to nuetral
else
    PWMout=xfix({xlUnsigned,14,7,xlRound},7.60);
end
PWM=PWMout;
% set flag
if SV > xfix({xlUnsigned,4,0},4)
    SVout=0;
else
    SVout=1;
end
```

Figure 84: Velocity Limiting M-File

Figure 85: Steering Limiting M-File

7.2 RC-Truck Results

Five trials were run with the RC-Truck following the same trajectory each time. As with the simulation, an approximate figure eight trajectory was assigned. The position in latitude and longitude and the velocity in the vehicle's body reference frame were stored utilizing a laptop's USB port and the HyperTerminal program. A text file

125

containing the information was created by the HyperTerminal. The information was then

read into MATLAB and the trajectories and velocities plotted for each trial. The

velocities for trial one, trial two, trial three, trial four and trial five are presented in

Figure 86, Figure 88, Figure 90, Figure 92, and Figure 94, respectively. The trajectories

for trial one, trial two, trial three, trial four and trial five are presented in Figure 87,

Figure 89, Figure 91, Figure 93, and Figure 95, respectively.
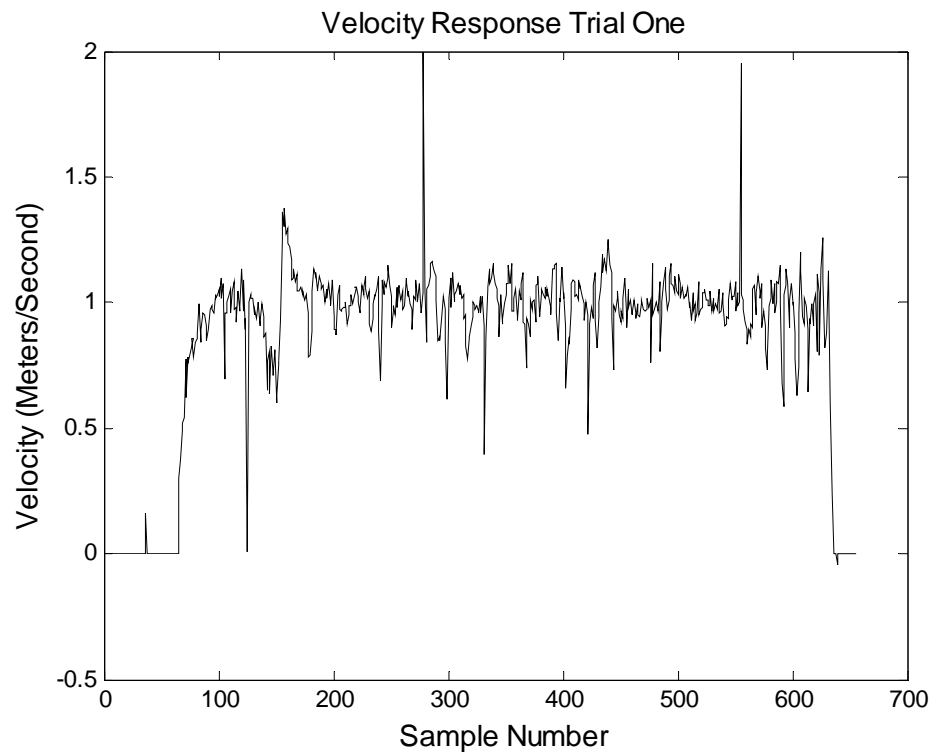
Figure 86: Velocity Response for Trial One

126
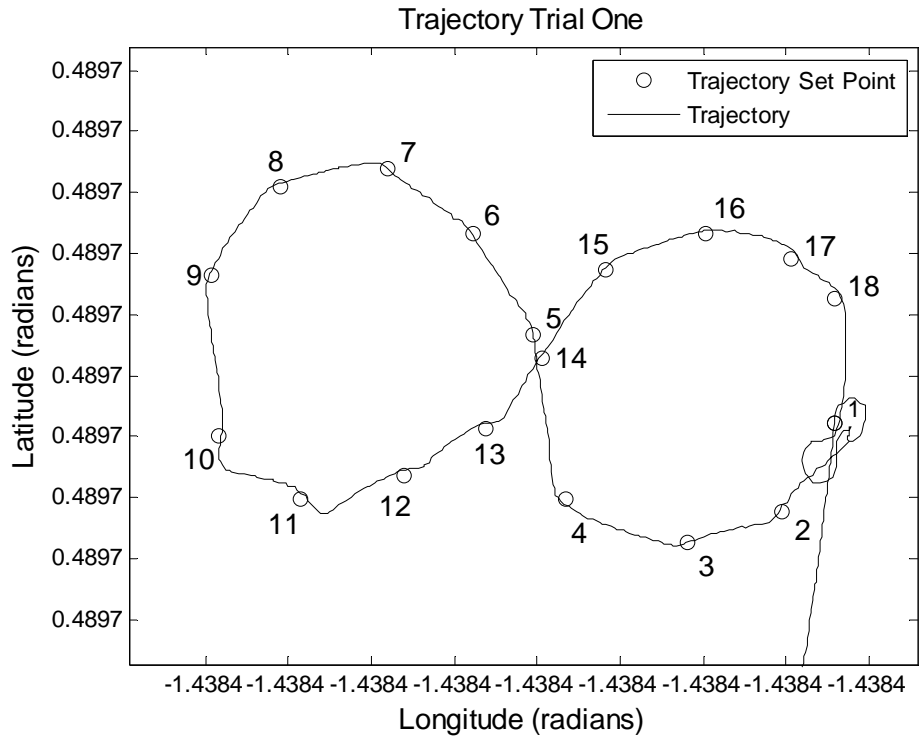
Figure 87: Trajectory for Trial One
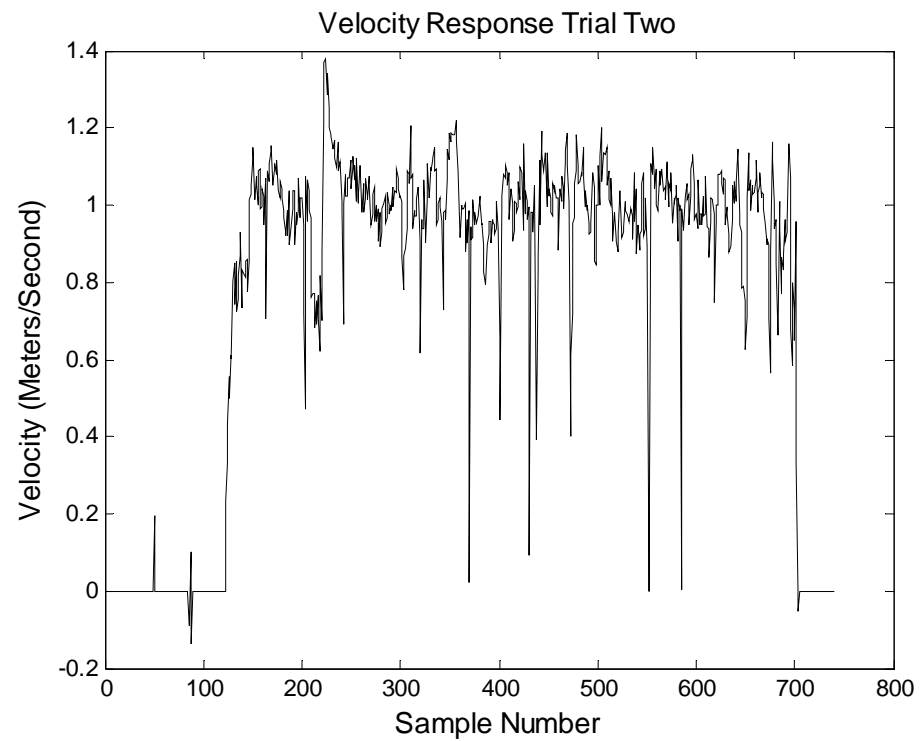
Figure 88: Velocity Response for Trial Two

127
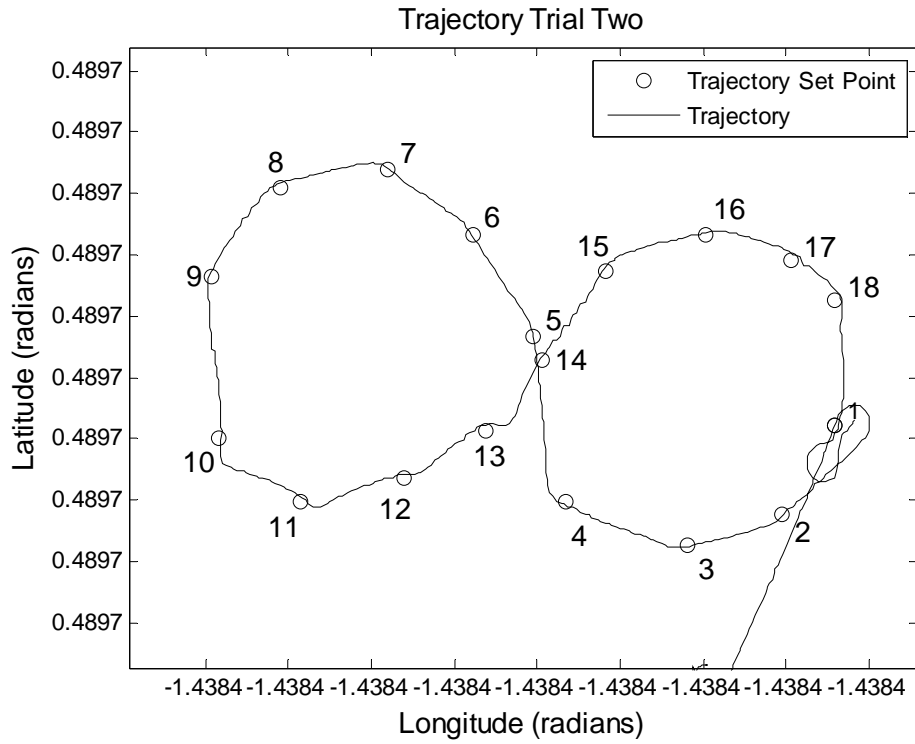
Figure 89: Trajectory for Trial Two
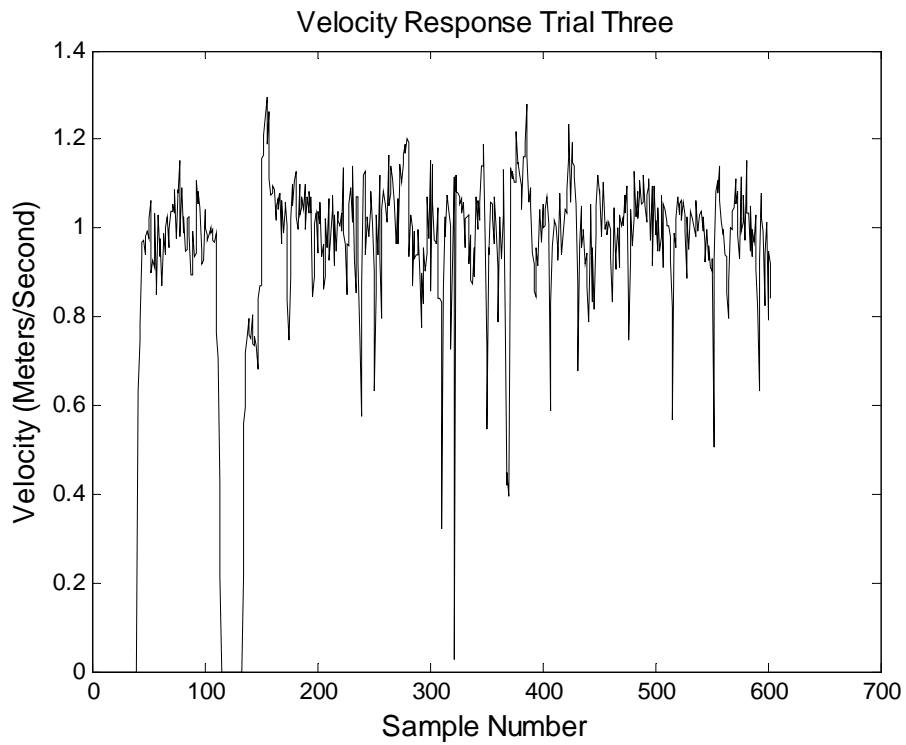
Figure 90: Velocity Response for Trial Three

128
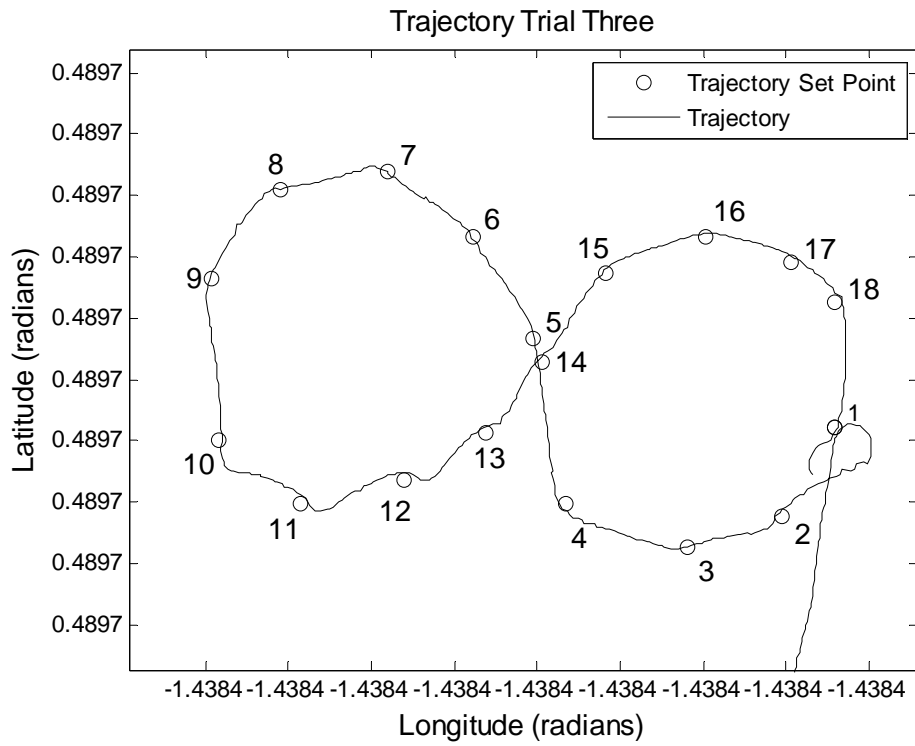
Figure 91: Trajectory for Trial Three
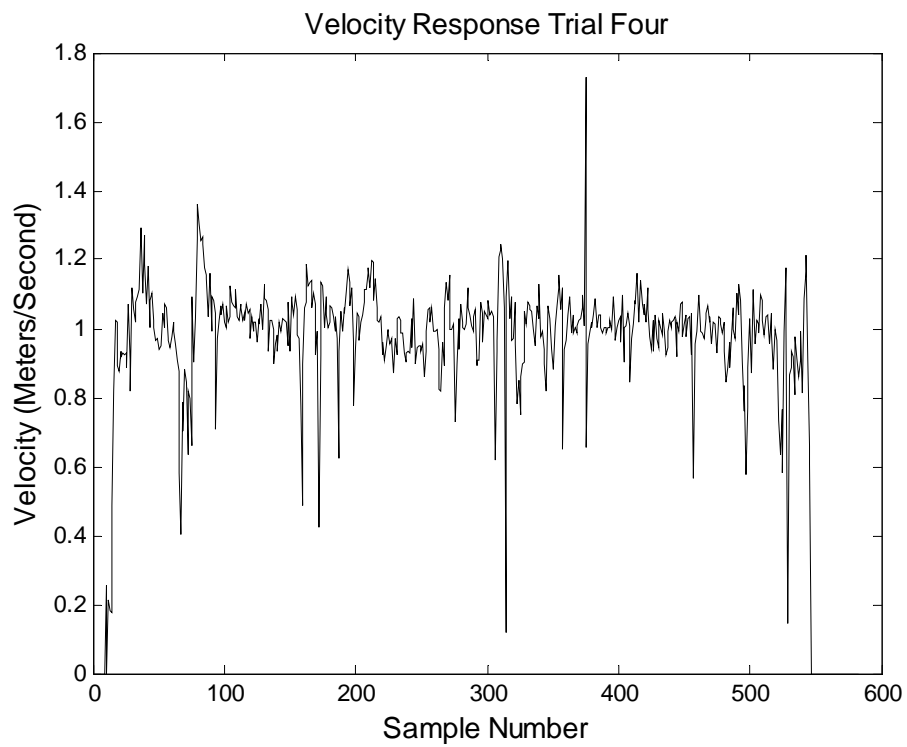
Figure 92: Velocity Response for Trial Four

129
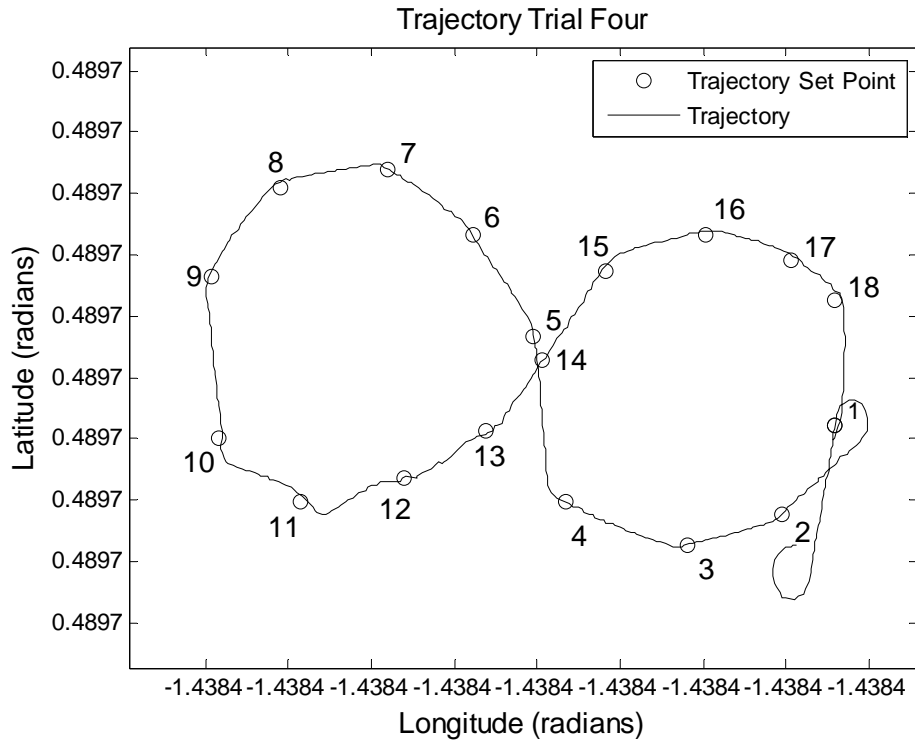
Figure 93: Trajectory for Trial Four
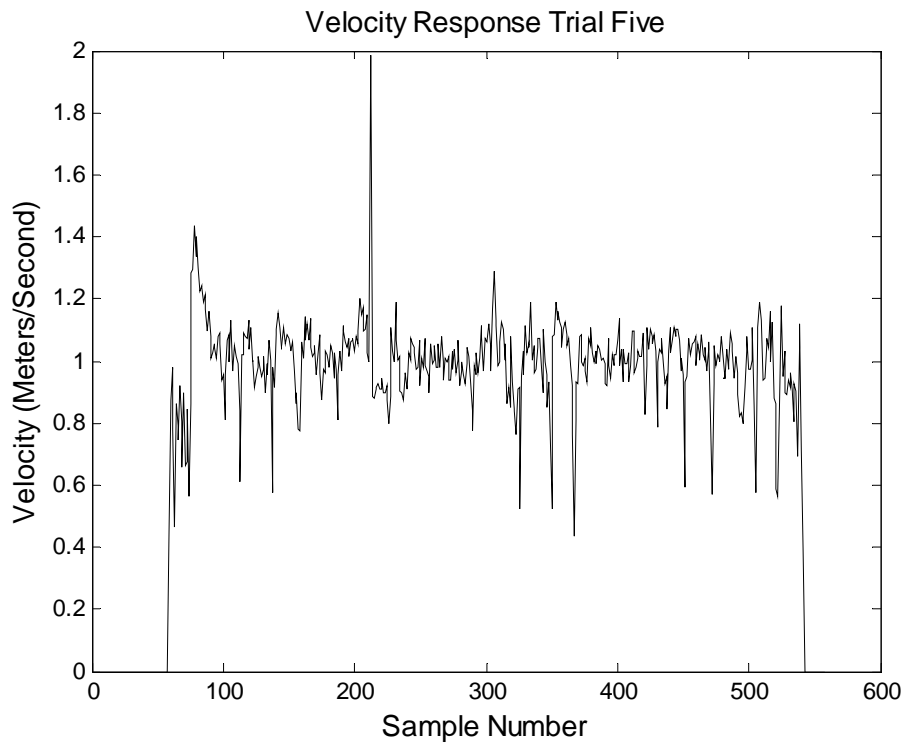
Figure 94: Velocity Response for Trial Five

130
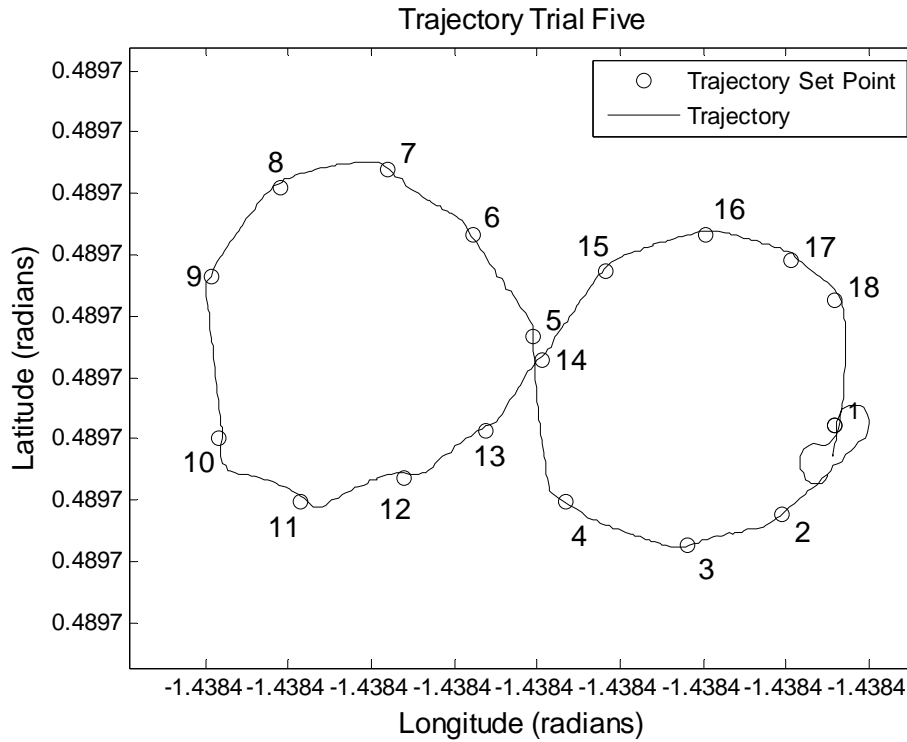
Figure 95: Trajectory for Trial Five

The velocity responses of the RC-Truck were similar for all trials. The velocities were held close to one meter, but with some slight oscillation. These oscillations occurred because the GPS unit contains some error in the measurements. There were also points in along the path where the robot slowed down. It was visually observed from the LED indicator that the number of satellites used by the GPS had dropped below four. Because of the PWM control algorithm, the motors were turned off and the RC-Truck began to coast to a stop until a GPS lock was re-established. During the time period where there was no GPS lock, a strong velocity control effort was present due to the velocity error that was present. Once the GPS lock was regained, this control effort created a slight increase the duty cycle controlling the motor. This response was acceptable, as the loss in GPS rarely lasted more than a second. Had the GPS been less

131

reliable, code could have been added to force a velocity set point of zero when the number satellites used by the GPS unit dropped below five.

The RC-Truck followed very similar paths for each of the trials. It was observed that for each trial the RC-Truck made a sharp turn just past the 10th, 11th, and 12th way points. Because the robot was operating close to a building, it was quite likely that there was interference with the heading measurement due to underground power lines, or some other external contributing factor. Despite this slight wavering from the desired figure eight trajectory, the robot did successfully reach all the way points along the path.

Although this implementation was very simplistic in nature, it demonstrated the effectiveness of the autopilot platform as a method rapid system prototyping. In addition, it demonstrated the flexibility across sensors and platforms. All of the sensors and the RC-Truck platform were available within the Unmanned System Lab at the University of South Florida. The autopilot accommodated each piece of hardware without requiring any circuitry modification or custom sensors to be ordered.

CHAPTER 8

CONCLUSIONS

This design of the autopilot produced during this research included all of the best features of various autopilot platforms such as integration with *Simulink*, open source to allow any modification required and full FPGA implementation. In addition, the design demonstrated its contribution by including additional features, which are unique as far as the author is aware.

Many of the designs implementing FPGAs such as the Microbot and the GTSpy still utilize a separate DSP/microcontroller processor for the majority of the processing. Therefore, these designs do not allow for the benefits of parallel processing. Two of the full FPGA designs require the programmer to implement the majority of the programming in a PowerPC, [9, 36]. This restriction requires implementing some or all of the processing utilizing a real-time operating system, without taking full advantage of parallel processing capabilities. The research being performed by Wolter et. al., on a design for the control of a Satellite is still in the initial stages. However, the analysis performed indicated good timing and showed parallel communication could be maintained by utilizing the full parallel processing capabilities of the FPGA, [40]. The only design found that provided for programming directly through *Simulink* is the Piccolo autopilot. The Piccolo utilized a DSP processor without parallel processing capabilities

133

and, in addition, required a CAN interface in order to implement the hardware-in-the-loop verification.

By implementing full FPGA processing design and full *Simulink* integration, the benefits of rapid system prototyping, tight timing control and flexibility across platforms and sensors are realized. The integration with *Simulink* provides programming and hardware-in-the-loop capabilities in an environment familiar to researchers in many areas of engineering. Design within this environment will provide for rapid prototyping of new ideas. In addition, to hardware-in-the-loop capabilities of *Simulink*, the software design capabilities provide for directly integrating the hardware ports and required communication protocol, which further improves upon the time required to implement a new design. The autopilot design, produced by this research in this environment, provides an unrivaled flexibility due the programmable analog and TTL interfaces and the inherent flexibility of the FPGA processor. There are many systems in use, which incorporate the mini-ITX or PC-104. The autopilot design produced by this research, instead of intending to be a replacement, complements these systems. The complement arises by providing dedicated hardware for real-time controls of the system dynamics while giving the off board computer the role of a "master" computer when necessary. The combined functionality and flexibility of the design has produced a novel and well-needed processing platform for the unmanned systems community.

REFERENCES

[1]   "Unmanned Aircraft Systems Roadmap 2005-2030",
        http://www.acq.osd.mil/usd/Roadmap%20Final2.pdf, 2005

[2]   http://www.baiaerosystems.com

[3]   D. N. Borys and R. Colgren, "Advances in Intellegent Autopilot Systems for
        Unmanned Aerial Vehicles", AIAA Guidance, Navigation, and Control
        Conference and Exhibit, San Francisco, California, 2005

[4]   http://www.rotomotion.com

[5]   http://www.procerusuav.com

[6]   http://www.micropilot.com

[7]   http://www.ezi-nav.com

[8]   http://www.o-navi.com

[9]   R. H. Klenke, W. C. S. IV and M. A. Motter, "A High-Throughput Processor for
        Flight Control Research Using Small UAVs", 25th AIAA Aerodynamic
        Measurement Technology and Ground Testing Conference, San Francisco,
        California, 2006

[10]   http://www.cloudcaptech.com

[11]   http://www.microboticsinc.com

[12]   D. Jung, E. J. Levy, D. Zhou, R. Fink, J. Moshe, A. Earl and P. Tsiortras, "Design
        and Development of a Low-Cost Test-Bed for Undergraduate Education in
        UAVs", *44th IEEE Conference on Decision and Control, and the European
        Control Conference*, pp. 2739-2744, 2005

[13] D. Kingston, R. Beard, T. McLain, M. Larsen and W. Ren, "Autonomous Vehicle Technologies for Small Fixed Wing UAVs", 2nd AIAA "Unmanned Unlimited" Systems, Technologies, and Operations, San Diego, California, 2003

[14] A. D. Kahn and J. C. Kellogg, "Low Complexity, Low Cost, Altitude Heading Hold Flight Control System", *IEEE AESS Systems Magazine*, pp. 14-18, 2003

[15] A. Sagahyroon, M. A. Jarah, A. Al-Ali and M. Hadi, "Design and Imeplemention of a Low Cost UAV Controller", *IEEE International Conference on Industrial Technology* pp. 1394-1397, 2004

[16] P. Y. Oh and W. E. Green, "CQAR: Closed Quarter Aerial RobotDesign for Reconnaissance, Surveillance and Target Acquisition Tasks in Urban Areas", *International Journal of Computational Intelligence*, vol. 1, pp. 353-360, 2004

[17] R. J. Wood, S. Avadhanula, E. Steltz, M. Seeman, J. Entwistle, A. Bachrach, G. Barrows, S. Sanders and R. S. Fearing, "Design Fabrication and Initial Results of a 2g Autonomous Glider", *IEEE*, pp. 1870-1877, 2005

[18] S. Bouabdallah, A. Noth and R. Siegwart, "PID vs LQ Control Techniques Applied to an Indoor Micro Quadrotor", *RSJ International Conference on Intelligent Robots and Systems*, pp. 2451-2456, 2004

[19] S. Todorovic and M. C. Nechyba, "A Vision System for Intelligent Mission Profiles of Micro Air Vehicles", *IEEE Transactions on Vehicular Technology*, vol. 53, pp. 1713-1725, 2004

[20] Y.-J. Yang, J.-P. Chen, J.-S. Cheng, C. Zhang and Y.-L. Xiao, "Autonomous Micro-Helicopter Control Based on Reinforcement Learning with Replacing Eligibility Traces", *Proceedings of the First International Conference on Machine Learning and Cybernetics*, pp. 860-864, 2002

[21] M. D. Bugajska and A. C. Schultz, "Coevolution Form and Function in the Design of Micro Air Vehicles", *IEEE Proceedings, NASA/DOD Conference on Evolvable Hardware*, 2002

[22] S. H. McIntosh, S. K. Agrawal and Z. Khan, "Design of a Mechanism for Biaxial Rotation of a Wing for a Hovering Vehicle", *IEEE/ASME Transactions on Mechatronics*, vol. 11, pp. 145-153, 2006

[23] S. E. Lyshevski, "Distributed Control of MEMS-Based Smart Flight Surfaces", *Proceedings of the American Control Conference*, pp. 2351-2356, 2001

[24]   A. S. Wu, A. C. Schultz and A. Agah, "Evolving Control for Distributed Micro Air Vehicles", *IEEE*, pp. 174-179, 1999

[25]   J. M. Pflimlim, P. Soueres and T. Hamel, "Hovering Flight Stabilization in Wind Gusts for Ducted Fan UAV", *43rd IEEE Conference on Decision and Control*, pp. 3491-3496, 2005

[26]   D. Sun, H. Wu, R. Zhu and L. C. Hung, "Development of Micro Air Vehicles Based on Aerodynamic Modeling Analysis in Tunnel Tests", *Proceedings, IEEE International Conference on Robotics and Automation*, pp. 2235-2240, 2005

[27]   H.-y. Wu, D. Sun, Z.-y. Zhou, S.-s. Xiong and X.-h. Wang, "Micro Air Vehicle: Architecture and Implementation", *Proceedings, International Conference on Robotics & Automation*, pp. 534-539, 2003

[28]   F. Ruffier, S. Viollet, S. Amic and N. Franceschini, "Bio-Inspired Optical Flow Circuits for the Visual Guidance of Micro-Air Vehicles", *IEEE*, pp. III-846-III-849, 2003

[29]   S. Taamallah, A. J. C. d. Reus and J.-F. Boer, "Development of a Rotorcraft Mini-UAV System Demonstrator", *IEEE*, vol. 2005, pp. 11.A.2-1-11.A.2-15, 2005

[30]   J. Evans, G. Inalhan, J. S. Jang, R. Teo and C. J. Tomlin, "Dragonfly: A Versitile UAV Platform for the Advancement of Aircraft Navigation and Control", *IEEE*, pp. 1.C.3-1-1.C.3-12, 2001

[31]   J. L. Campbell and J. T. Kresge, "Brumby Uninhabited Aerial Vehicle Flight Dynamics-Instrumentation and Flight Test Results", *IEEE*, 2003

[32]   G. Cai, K. Peng, B. M. Chen and T. H. Lee, "Design and Assembling of a UAV Helicopter System", *IEEE International Conference on Control and Automation*, pp. 697-702, 2005

[33]   S.-J. Lee, S.-P. Kim, T.-S. Kim, H.-K. Kim and H.-C. Lee, "Development of Autonomous Flight Control System for 50m Unmanned Airship", *IEEE*, pp. 457-462, 2004

[34]   E. N. Johnson, S. G. Fontaine and A. D. Kahn, "Minimum Complexity Unnhabited Air Vehicle Guidance and Flight Control System", *AIAA Digital Avionic Conference*, pp. 1-9, 2001

[35]   W. E. Hong, J. S. Lee, L. Rai and S. J. Kang, "RT-Linux based Hard Real-Time Software ARchitecture for Unmanned Autonomous Helicopters", *11th IEEE Conference on Embedded and Real-Time Compute Systems an Applications*, 2005

[36]   T. Brotherton, R. Luppold, P. Padykula and S. L. Richard Wade, "Generic Integrated PHM / Controller System", *IEEE*, 2005

[37]   H. B. Christopherson, W. J. Pickell, A. A. Koller, S. K. Kannan and E. N. Johnson, "Small Adaptive Flight Control Systems for UAVs using FPGA/DSP Technology", *American Institute of Aeronautics and Astronautics*

[38]   A. A. Proctor, B. Gwin, S. K. Kannan and A. A. Koller, "Ongoing Development of an Autonomous Aerial Reconnaissance System at Georgia Tech."

[39]   R. H. Klenke, "A UAV-Based Computer Engineering Capstone Senior Design Project", *IEEE International Conference on Microelectronic Systems Education*, 2005

[40]   G. Grillmayer, M. Hirth, F. Huber and V. Wolter, "Development of an FPGA Based Attitude Control System for a Micro-Satallite", AIAA/AAS Astrodynamics Specialist Conference and Exhibit, Keystone, Colorado, 2006

[41]   F. Krach, B. Frackelton, J. Carletta and R. Veillette, "FPGA-Based Implementation of Digital Control for a Magnetic Bearing", Proceedings of the American Control Conference, Denver, Colorado, 2003

[42]   Z. Fang, J. E. Carletta and R. J. Veillete, "A Methodology for FPGA-Based Control Implementation", *IEEE Transactions on Control Systems Technology*, vol. Vol 13, pp. 977-987, 2005

[43]   T. S. Hall, C. M. Twigg, P. Hasler and D. V. Anderson, "Developing Large-scale Field-programmable Analog Arrays for Rapid Prototyping", *International Journal of Embedded Systems*, vol. 1, 2005

[44]   "www.maxim-ic.com", *Application Note 3803*

[45]   http://www.wilkepedia.com

[46]   www.xilinx.com, "Spartan-3A/AN Starter Kit Board Schematic", 2007

[47]   www.national.com, "Flexible Power Management Units for Low-Power Xilinx FPGAs", 2007

138

[48]   S. N. Murthy, "Implementation of Unmanned Vehicle Control on FPGA Based
        Platform Using System Generator", 2007

[49]   R. Andraka, "A Survey of CORDIC Algorithms for FPGAs", Proceedings,
        ACM/SIGDA Conference, Sixth International Symposium on Field
        Programmable Gate Arrays, 1998

[50]   "SD Specifications Part 1 Physical Layer Simplified Specification," *SD Group*,
        2006

[51]   I. Nov Atel, "Superstar II Firmware Reference Manual," 2005

APPENDICES

140

Appendix A Details of Commercial Autopilots

Table 7: Kestral by Procerus

Processing Hardware:	29 MHz, 8-bit Rabbit 3000 processor
Onboard Sensors:	IMU unit onboard, does not specify brand
	Pressure sensors for altitude and air speed
I/O Ports:	4 RS232 ports for off-shelf components such as GPS
	3 12-bit analog inputs provided
	Built in support for 2 axis with zoom camera gimbal
Outputs:	4 onboard servo ports, 8 external servo ports
Programming:	developers kit and dynamic C
Hardware-in-the-Loop:	proprietary software used in conjunction with Aviones simulator

Table 8: MP2028 by Micropilot

Processing Hardware:	Motorola's 68332 processor 20MHz 32-bit processor
Onboard Sensors:	Trimble Lassen SQ GPS receiver
	Motorola onboard pressure sensors for air speed and altitude
	iMEMS ADXL202 accelerometer
	iMEMS ASXRS150 Gyro
I/O Ports:	Additional ADC board for 32 analog inputs and compass
	Additional AGL board for ultrasonic altimeter and modem
Actuator Outputs:	24 Servos or relays
Programming:	XTENDER software can be purchased that allows for custom programming.
Hardware-in-the-Loop:	With proprietary software (Horizon) only

Table 9: Ezi-Nav by Autonomous Unmanned Air Vehicles, (AUAV)

Processing Hardware:	8 micro-processors
Onboard Sensors:	Connections for handheld type GPS units only
	IMU provided, details not given
I/O Ports:	Not disclosed
Outputs:	Not disclosed
Programming:	Not disclosed
Hardware-in-the-Loop:	Not designed for this capability
Additional Functions:	Off-board wireless transceiver capable of 900MHz or 2.4 GHz provided

Appendix A (Continued)

Table 10: Phoenix by O-Navi

Processing Hardware:	32 MHz Motorola MMC-2114 processor
Onboard Sensors:	Unspecified MEMS accelerometers and gyros
	On-board pressure sensors for air speed and altitude.
	On-board Trimble GPS receiver
I/O Ports:	Additional sensors can be connected, but details not specified
Outputs:	6 PWM servo
Programming:	Flash programming kit available
Hardware-in-the-Loop:	Not designed for hardware-in-the-loop

Table 11: Piccolo II by Cloudcap

Processing Hardware:	Motorola's MPC555 40MHz 32-bit processor
Onboard Sensors:	3 ADXRS300 rate gyros
	2 two-axis ASXL21e
	Accelerometers
	uBlox TIM LP 4Hz GPS
	input port for sonic altimeter
	Honeywell HMR-2300 magnetometer,
	onboard pressure sensors to provide air speed and altitude
I/O Ports:	Additional daughter board provides analog, SPI, serial, CAN
Outputs:	10 servos
Programming:	*Simulink* using the Real Time Workshop
Hardware-in-the-Loop:	*Simulink* running on a PC equipped with a CAN interface card
Additional Functions:	Wireless capabilities supplied on a daughter board containing MHX-910/2400

Appendix A (Continued)

Table 12: Microbot by Microbotics

Processing Hardware:	FPGA for I/O operations
	M-Core MMC211 microprocessor for system programming
Onboard Sensors:	Expansion board provides temperature sensor and mounting for Midge series IMU/GPS
I/O Ports:	32 FPGA ports can be configured for various sensors
	Expansion board provides 2 asynchronous serial ports and 12 analog ports
Outputs:	FPGA lines used with pulse width generator to provide up to 16 PWM outputs
Programming:	Fully reprogrammable, details on required compiler not given
Hardware-in-the-Loop:	Not designed specifically for this
Additional Functions:	External board for Aerocomm AC4490 modem available
	Expansion board provides mounting for flash memory

Appendix B Port Connections to the FPGA
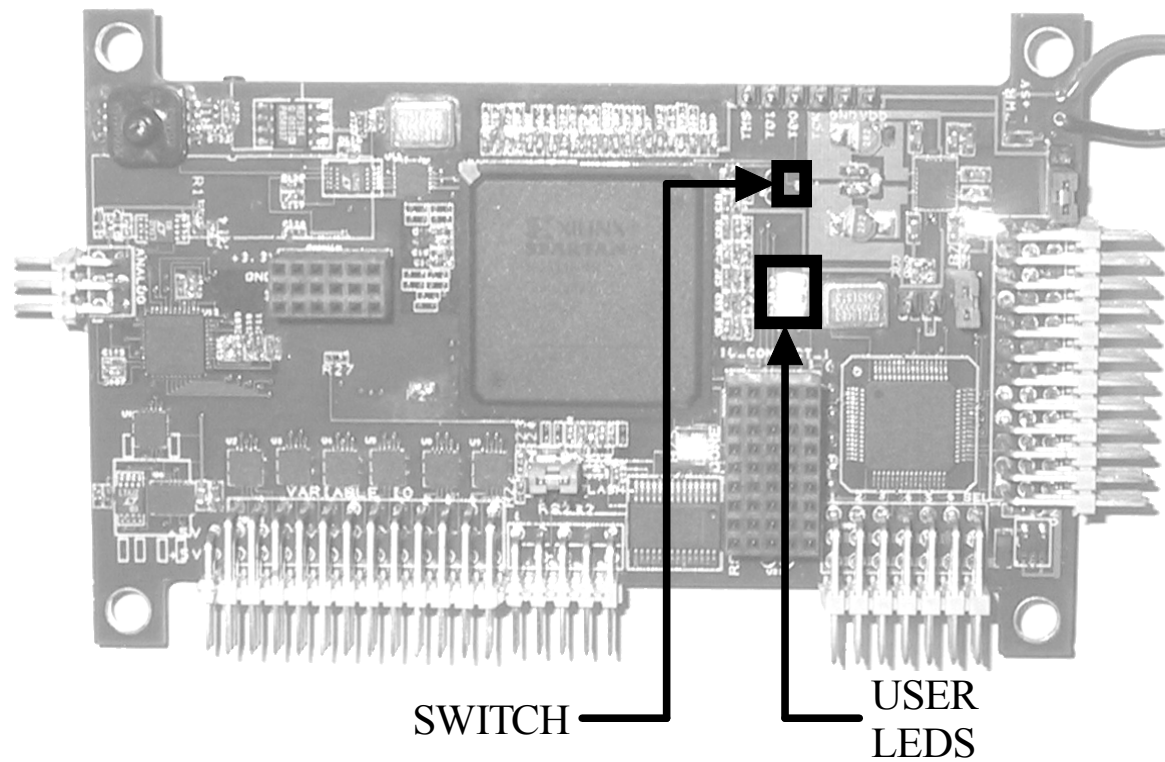
Figure 96: User LEDs and Switch Locations

Table 13: LED and Switch Port Assignments

PORT DESCRIPTION	FPGA PORT	PORT NAME
User LED 1	B21	LD1
User LED 2	B23	LD2
User LED 3	A22	LD3
User Switch	A20	SW1

144

Appendix B (Continued)
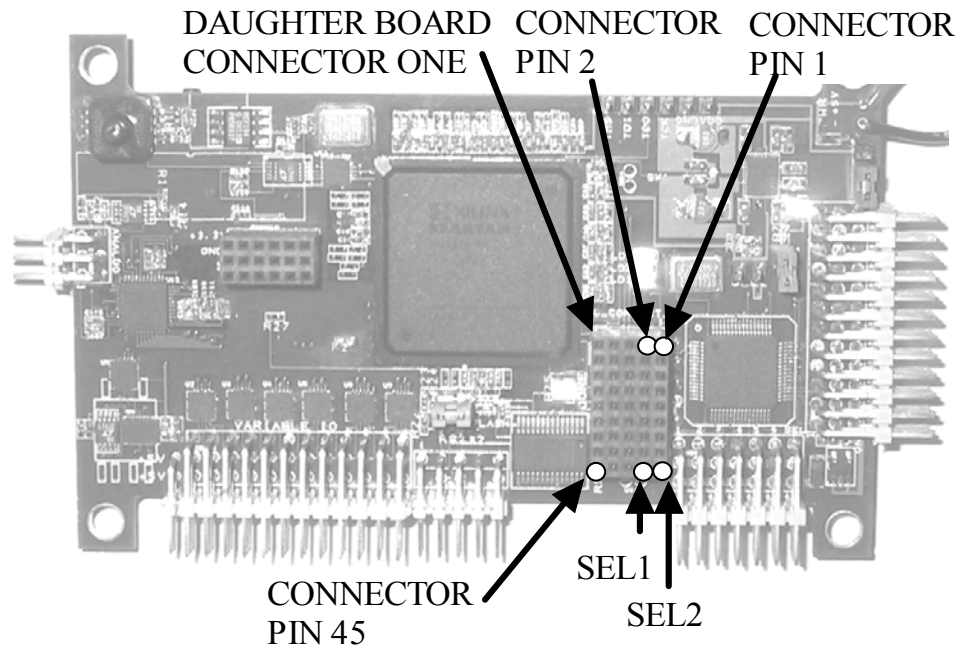
Figure 97: Daughter Board Connector One

Table 14: Daughter Board Connector One Safety Switch Connectors

PORT	FPGA PORT	CONNECTOR PIN
SEL1	--	41
SEL2	--	42
PWM1	--	43
PWM2	--	36
PWM3	--	37
PWM4	--	38
PWM5	--	31
PWM6	--	32
PWM7	--	33
PWM8	--	26
PWM9	--	27
PWM10	--	28
PWM11	--	21
PWM12	--	22

145

Appendix B (Continued)

Table 15: Daughter Board Connector One

PORT	FPGA PORT	CONNECTOR PIN
IO1	K26	1
IO2	K25	6
IO3	K23	2
IO4	K22	7
IO5	K21	3
IO6	V24	12
IO7	AD26	17
IO8	K20	8
IO9	G22	13
IO10	AC25	18
IO11	AF25	23
IO12	Y22	4
IO13	K18	9
IO14	G23	14
IO15	V18	19
IO16	AC21	24
IO17	AF23	29
IO18	V16	34
IO19	AE23	39
IO20	AE21	44
IO21	K19	5
IO22	L18	10
IO23	G24	15
IO24	V17	20
IO25	V19	25
IO26	V18	30
IO27	AE25	35
IO28	AD22	40
IO29	AE20	45
IO30	V19	11
IO31	AC26	16

146

Appendix B (Continued)
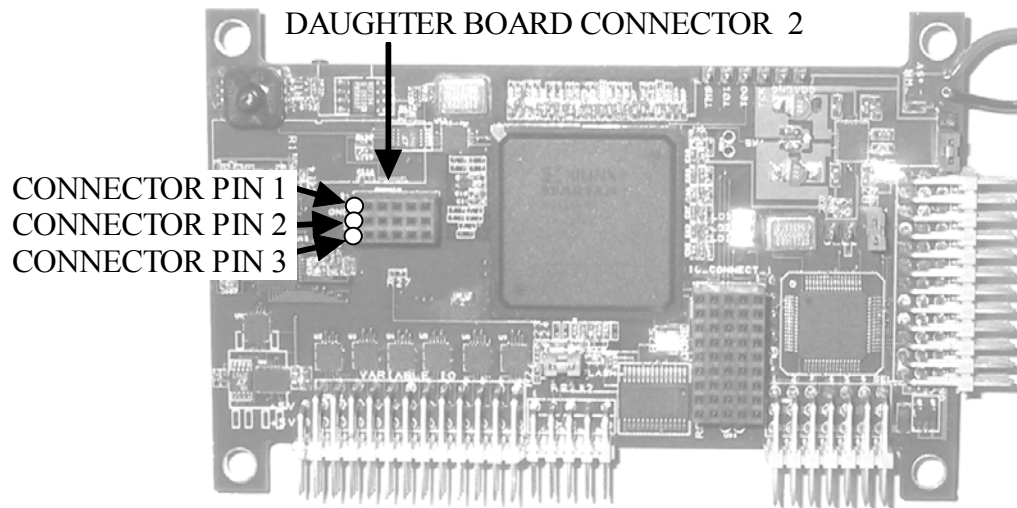
Figure 98: Daughter Board Connector Two

Table 16: Daughter Board Connector Two

CONNECTOR	FPGA PORT	PORT NAME
1	--	+3.3 Vcc
2	--	Cmn
3	--	+5Vcc
4	A3	IO32
5	F23	IO33
6	G20	IO34
7	B3	IO35
8	F25	IO36
9	F24	IO37
10	A4	IO38
11	E7	IO39
12	C8	IO40
13	B4	IO41
14	B6	IO42
15	D6	IO43
16	C6	IO44
17	B7	IO45
18	A8	IO46

147

Appendix B (Continued)

ANALOG INPUT CONNECTOR

BOTTOM ROW: PIN 5   TOP ROW: PIN 6
BOTTOM ROW: PIN 3   TOP ROW: PIN 4
BOTTOM ROW: PIN 1   TOP ROW: PIN 2
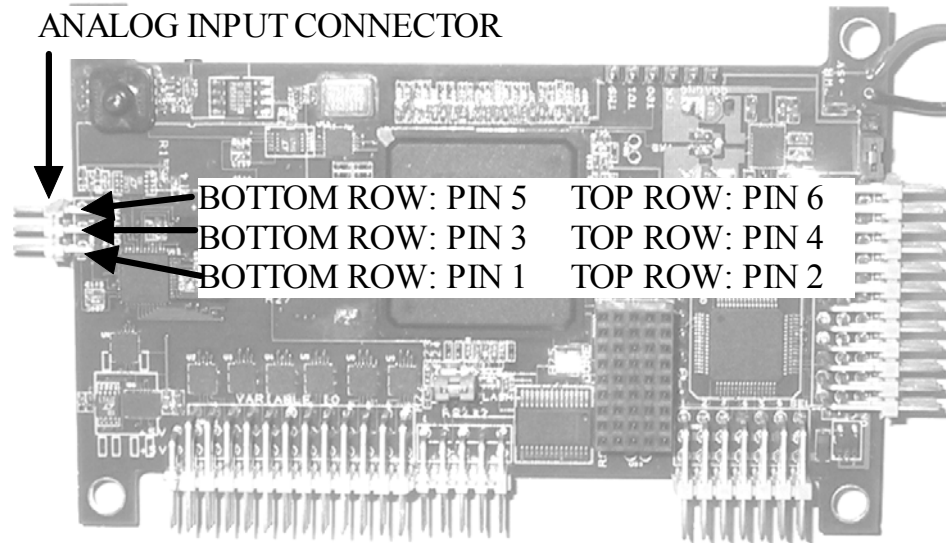
Figure 99: Analog Input Connectors

Table 17: FPAA Connections

CONNECTOR	FPGA PORT	PORT NAME
--	H17	FERRB
--	G9	FACT
--	F12	FRES
--	H10	FCS2B
--	J16	FSI
--	H12	FSCLK
--	H15	FACLK
--	F7	FCLK
--	K12	FDATA1
--	K11	FSYNCH1
--	J11	FDATA2
--	K16	FSYNCH2
--	J12	FDATA3
--	H9	FSYNCH3
1	--	CMN
2	--	+ SM SIGNAL
3	--	CMN
4	--	+ LRG SIGNAL 1
4	--	CMN
5	--	+ LRG SIGNAL 2

148

Appendix B (Continued)
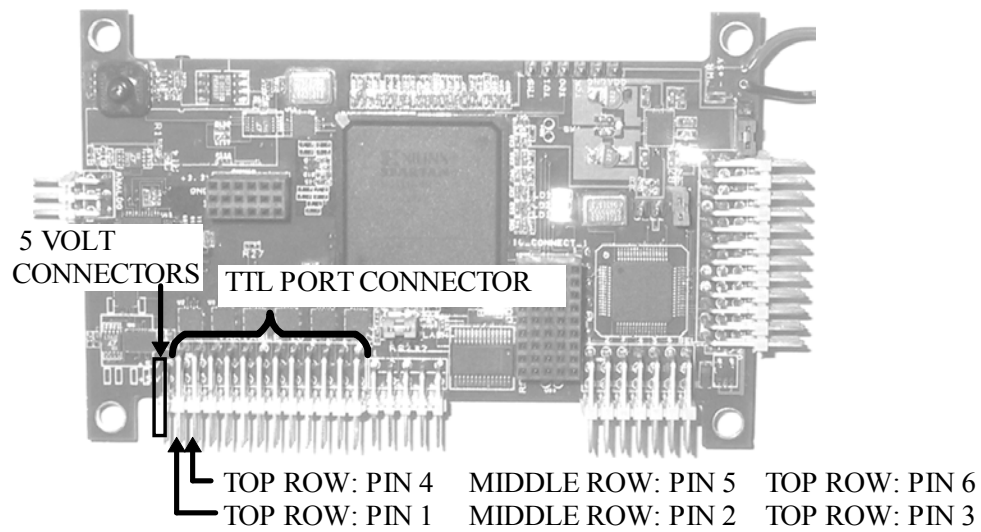
Figure 100: TTL I/O Connector

Table 18: TTL I/O Ports One to Three Connections

CONNECTOR	FPGA PORT	PORT NAME
1	V1	IO1_4
2	U1	IO1_3
3	--	CMN
4	Y5	IO1_2
5	AD1	IO1_1
6	--	CMN
--	Y2	IO1_EN
7	AD2	IO2_4
8	AC3	IO2_3
9	--	CMN
10	R3	IO2_2
11	T3	IO2_1
12	--	CMN
--	Y6	IO2_EN
13	T5	IO3_4
14	AA3	IO3_3
15	--	CMN
16	AA2	IO3_2
17	W3	IO3_1
18	--	CMN
--	T4	IO3_EN

149

Appendix B (Continued)

Table 19: TTL I/O Ports Four to Six Connections

CONNECTOR	FPGA PORT	PORT NAME
19	V2	IO4_4
20	U2	IO4_3
21	--	CMN
22	V5	IO4_2
23	U4	IO4_1
24	---	CMN
--	W4	IO4_EN
25	V6	IO5_4
26	W7	IO5_3
27	--	CMN
28	V7	IO5_2
29	U6	IO5_1
30	--	CMN
--	W6	IO5_EN
31	V8	IO6_4
32	U7	IO6_3
33	--	CMN
34	U8	IO6_2
35	U9	IO6_1
36	--	CMN
--	U5	IO6_EN
--	AC2	IO_SET_CS
--	AB1	IO_SET_SDI
--	Y1	IO_SET_CLK
--	AC1	IO_SET_EN

Table 20: Flash Memory

PORT NAME	FPGA PORT
uSD_CS	W10
uSD_DI	W9
uSD_CLK	AB7
uSD_DO	W12

150

Table 21: Pressure Sensor Connections

FPGA PORT	PORT NAME
B2	PS_CONV
B1	PS_SCK
D3	PS_SDO
E1	PS_SDI

Table 22: FPGA PWM Connections

FPGA PORT	PORT NAME
W23	PWM1
W21	PWM2
W20	PWM3
Y25	PWM4
Y24	PWM5
Y23	PWM6
AA25	PWM7
AA24	PWM8
AA23	PWM9
AB26	PWM10
AB23	PWM11
AC20	PWM12
U23	SS_IN
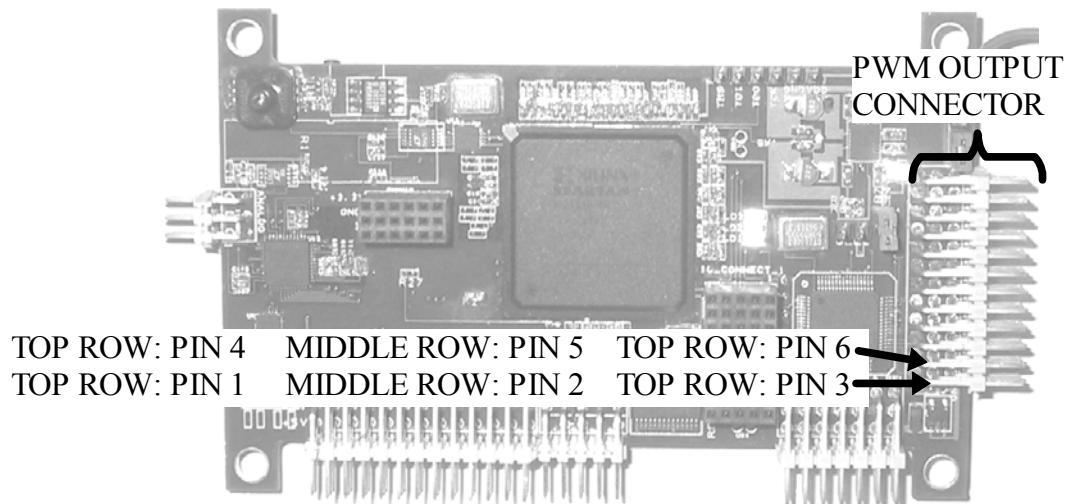
Figure 101: PWM Port Connections

151

Table 23: PWM Output Port Connections

CONNECTOR	PORT NAME
1	SERVO1
2	+6 VCC
3	CMN
4	SERVO2
5	+6VCC
6	CMN
7	SERVO3
8	+6VCC
9	CMN
10	SERVO4
11	+6VCC
12	CMN
13	SERVO5
14	+6VCC
15	CMN
16	SERVO6
17	+6VCC
18	CMN
19	SERVO7
20	+6VCC
21	CMN
22	SERVO8
23	+6VCC
24	CMN
25	SERVO9
26	+6VCC
27	CMN
28	SERVO10
29	+6VCC
30	CMN
31	SERVO11
32	+6VCC
33	CMN
34	SERVO12
35	+6VCC
36	CMN

152

TOP ROW: PIN 1   MIDDLE ROW: PIN 2   TOP ROW: PIN 3
TOP ROW: PIN 4   MIDDLE ROW: PIN 5   TOP ROW: PIN 6
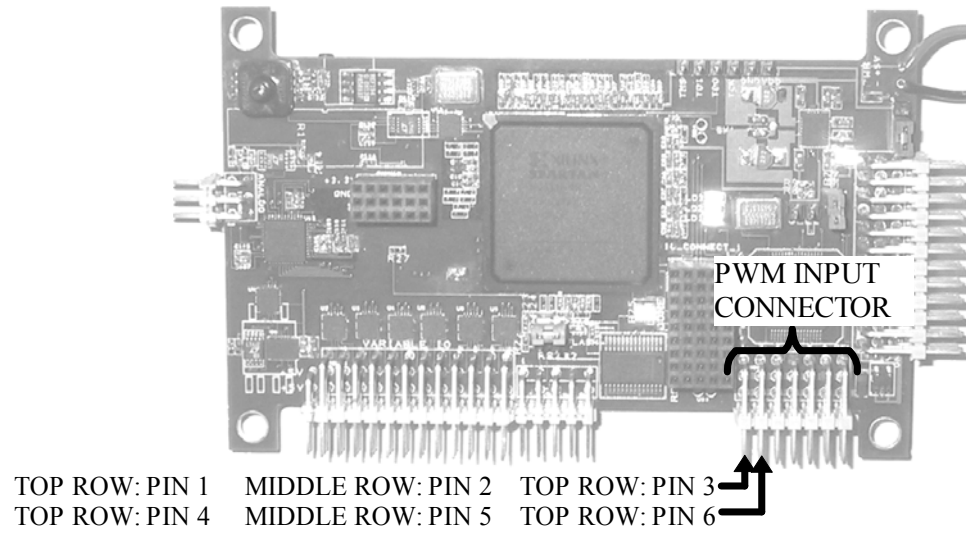
Figure 102: PWM Pilot Input Connector

Table 24: Pilot Input Connections

CONNECTOR	PORT NAME
1	PWM1
2	+6VCC
3	CMN
4	PWM2
5	+6VCC
6	CMN
7	PWM 3
8	+6VCC
9	CMN
10	PWM4
11	+6VCC
12	CMN
13	PWM5
14	+6VCC
15	CMN
16	PWM6
17	+6VCC
18	CMN
19	PILOT SELECT
20	+6VCC
21	CMN

153

TOP ROW: PIN 3   BOTTOM ROW: PIN 4
TOP ROW: PIN 1   BOTTOM ROW: PIN 2
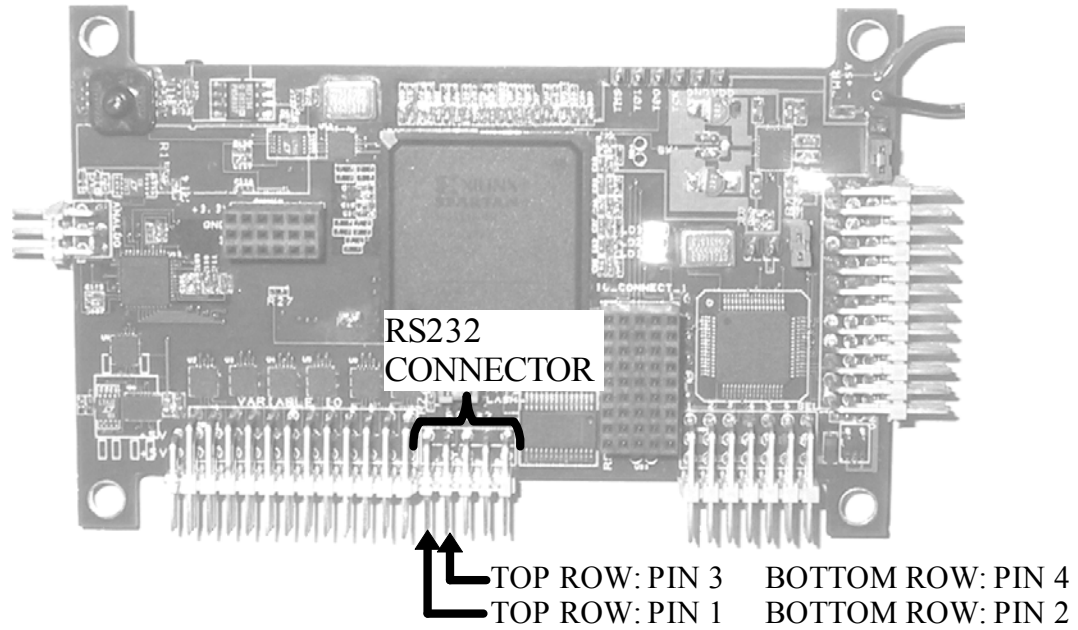
Figure 103: RS232 Connector

Table 25: RS232 Connections

CONNECTOR	FPGA	PORT NAME
1	AA17	TX1
2	AC19	RX1
3	Y17	TX2
4	AD19	RX2
5	AE17	TX3
6	AF20	RX3
7	AA18	TX4
8	AB18	RX4
9	AD17	RX5
10	--	CMN
--	AE19	RS232EN
--	AF19	RS232SD

154

Appendix C Detailed Schematics
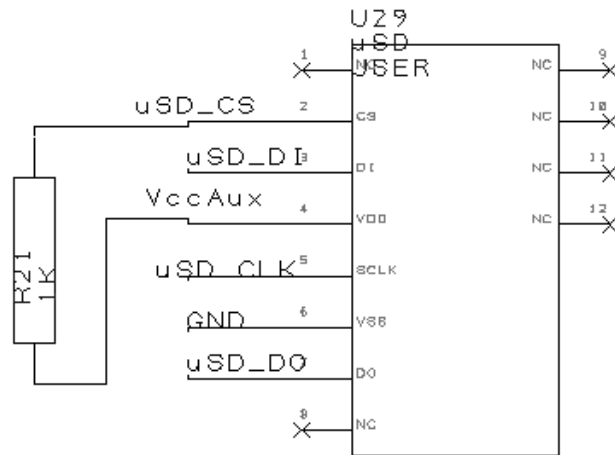

Figure 104: Flash Memory Circuit
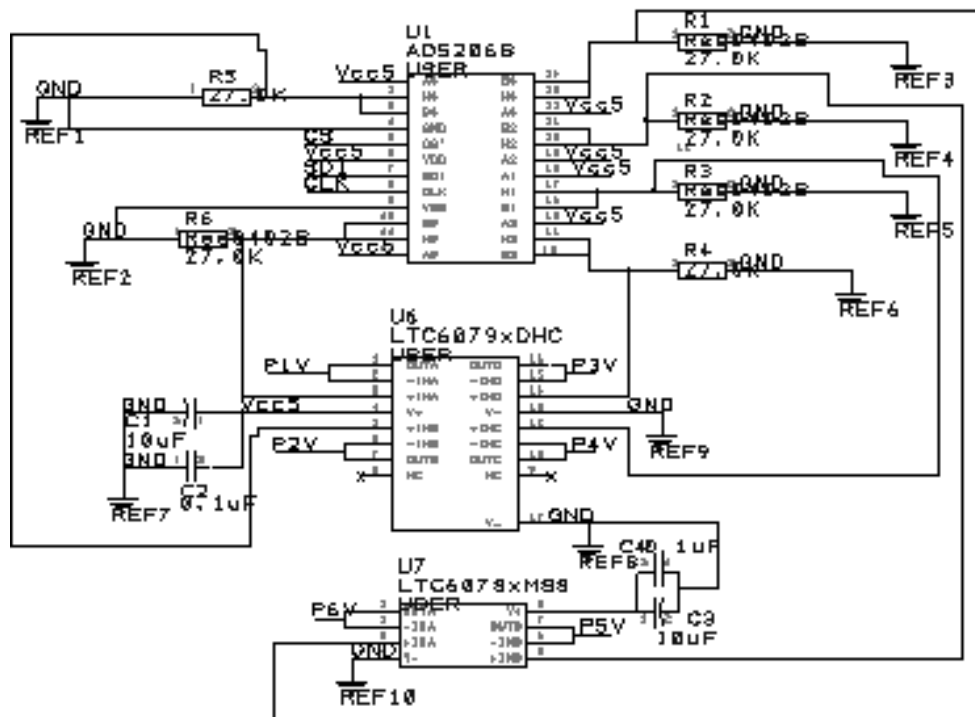

Figure 105: Variable I/O Port Potentiometer Circuit
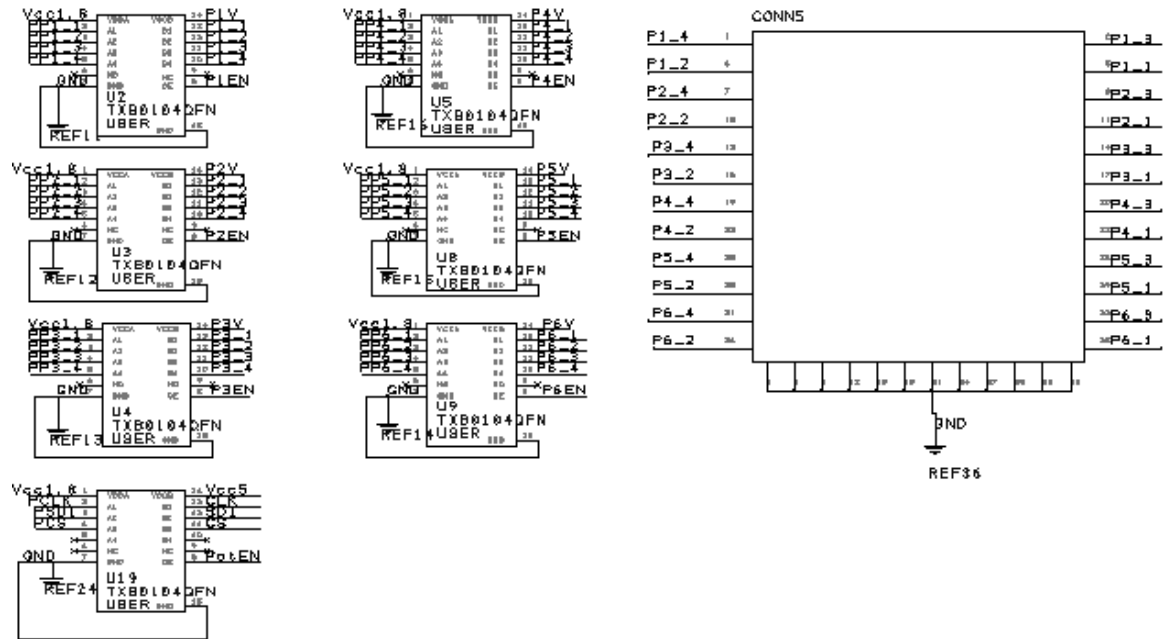
155

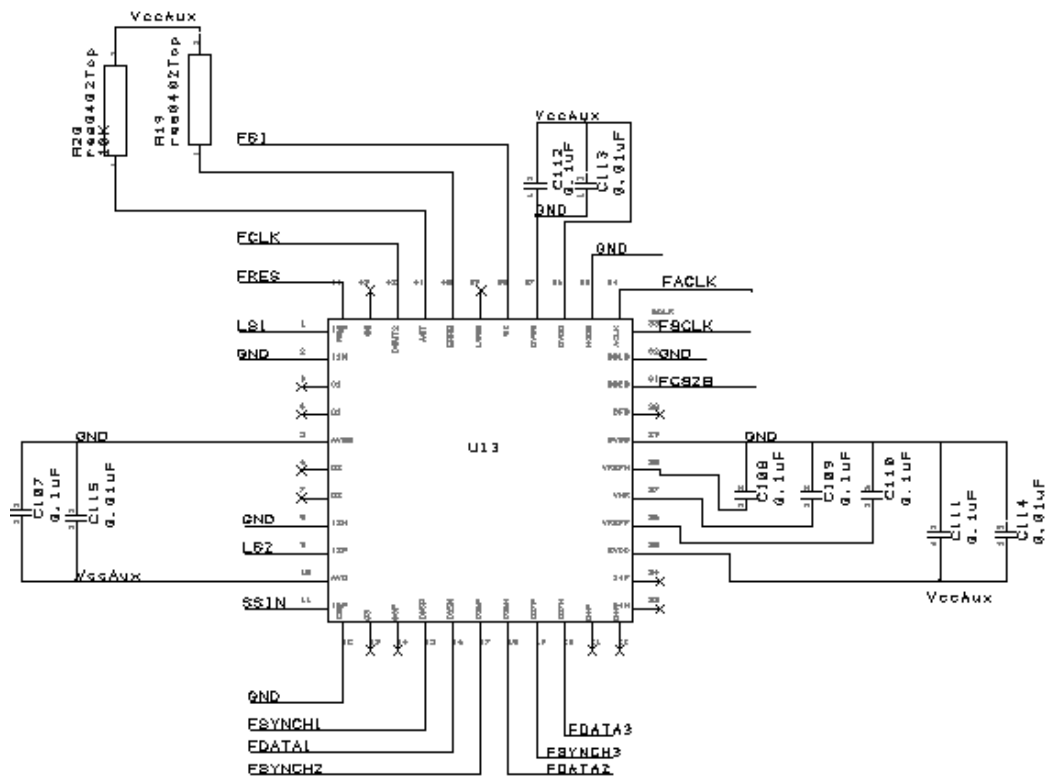Figure 106: Variable I/O Port Translator and Connector Circuitry

Figure 107: FPAA Circuit

156

Appendix C   (Continued)
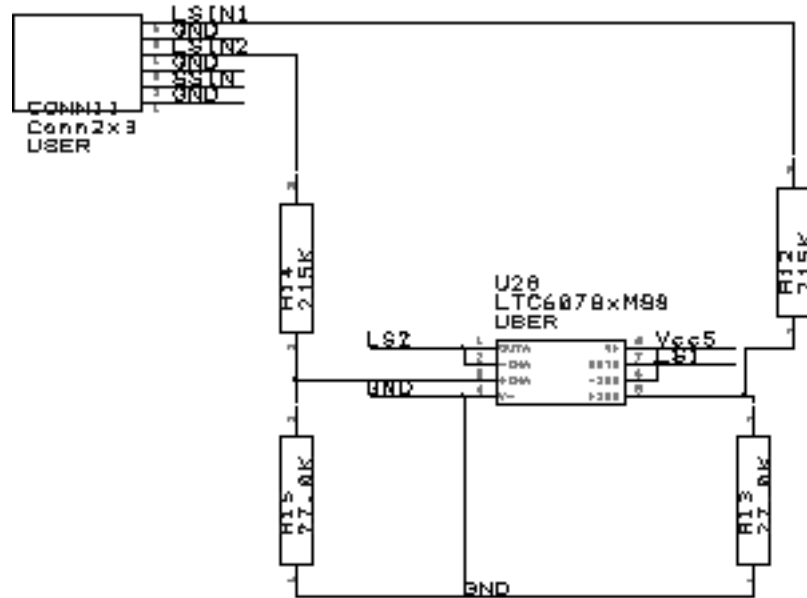
Figure 108: FPAA Input Circuit
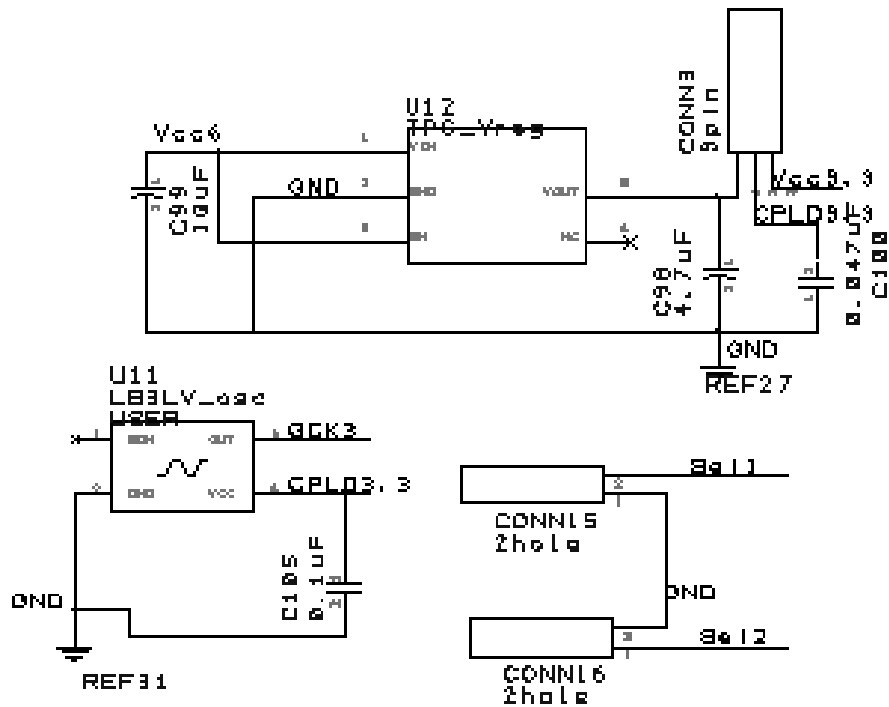
Figure 109: Safety Switch Power and Clock Circuit

157

Figure 110: Safety Switch CPLD and Connector Circuit
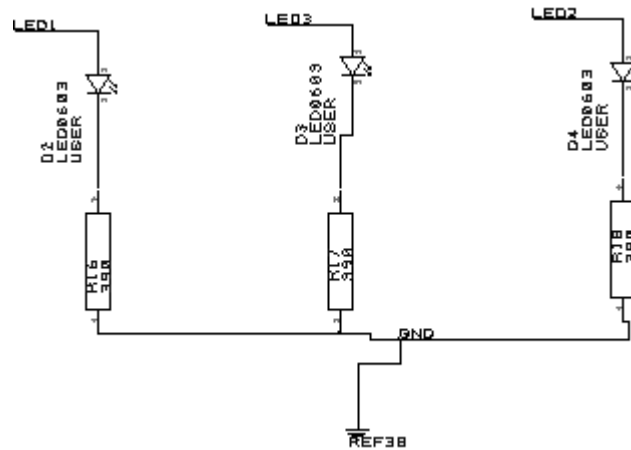
Figure 111: User LED Circuitry

158
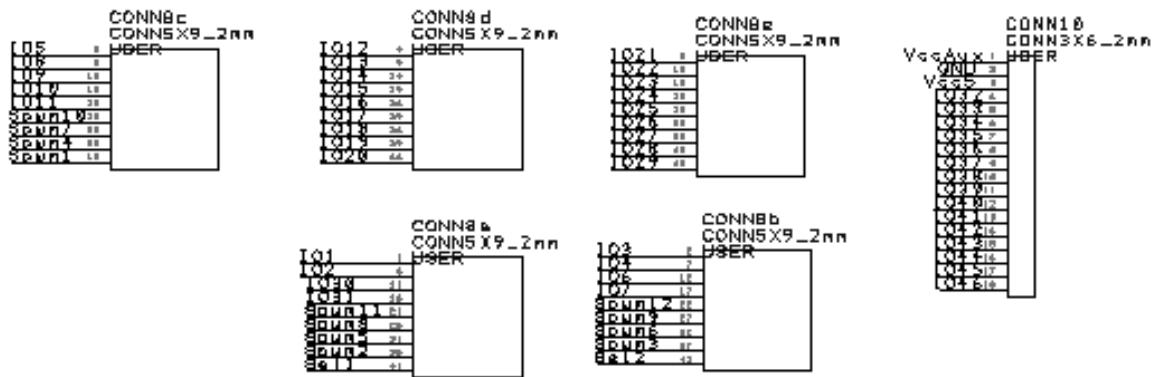
Figure 112: Daughter Board Connection Circuit
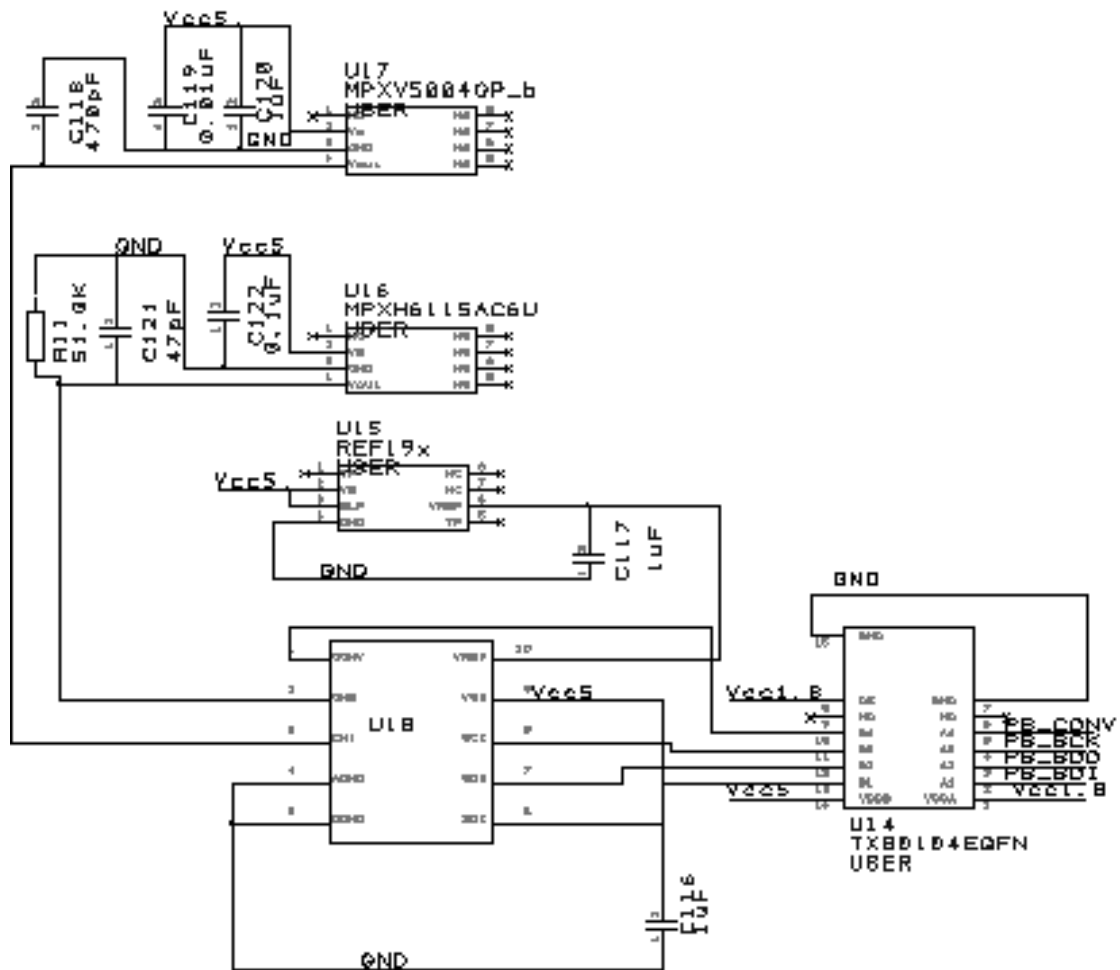
Figure 113: Pressure Sensor Circuitry

159

Appendix C   (Continued)

Figure 114: Power Supply Circuitry

Figure 115: RS232 Circuit

160
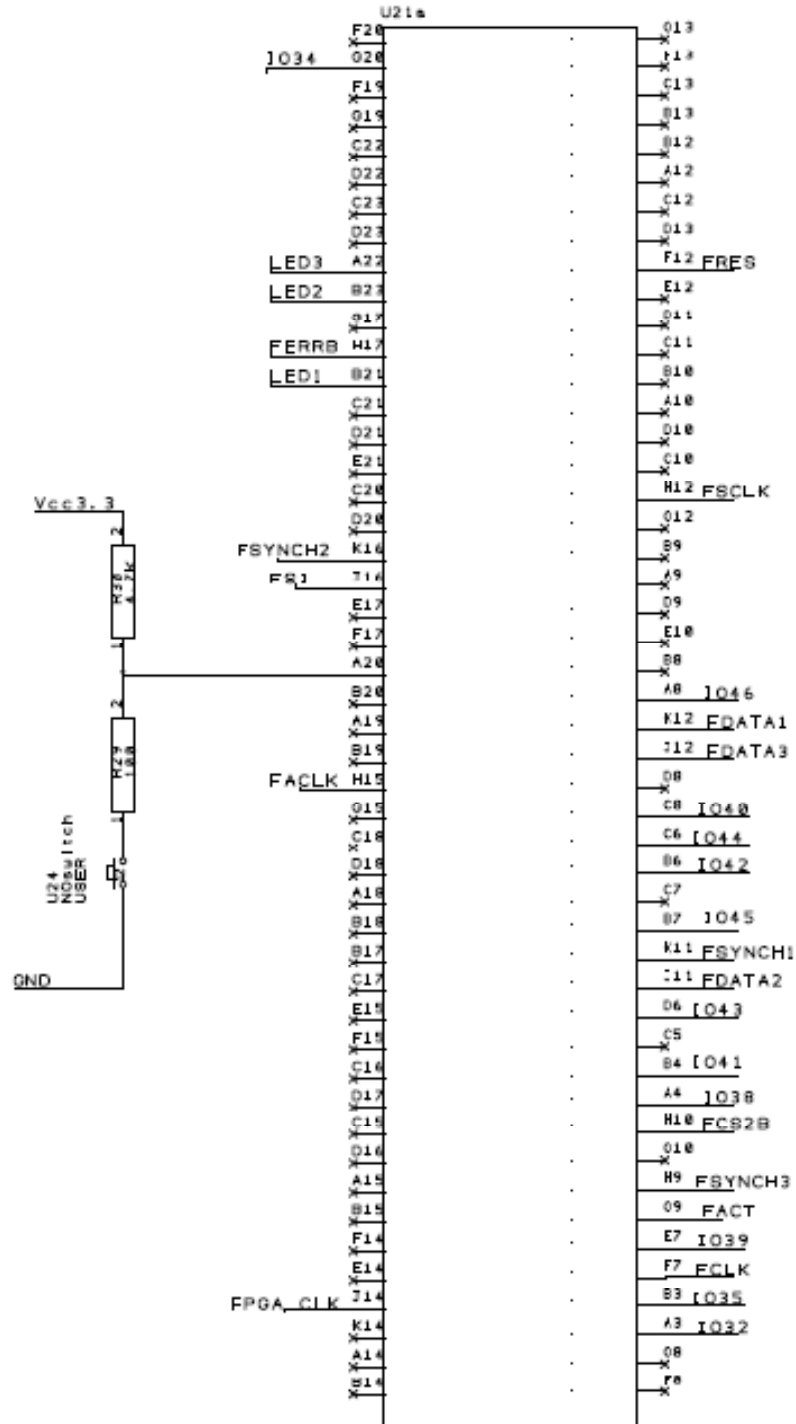
Figure 116: FPGA Bank0 Connections

161

Appendix C   (Continued)
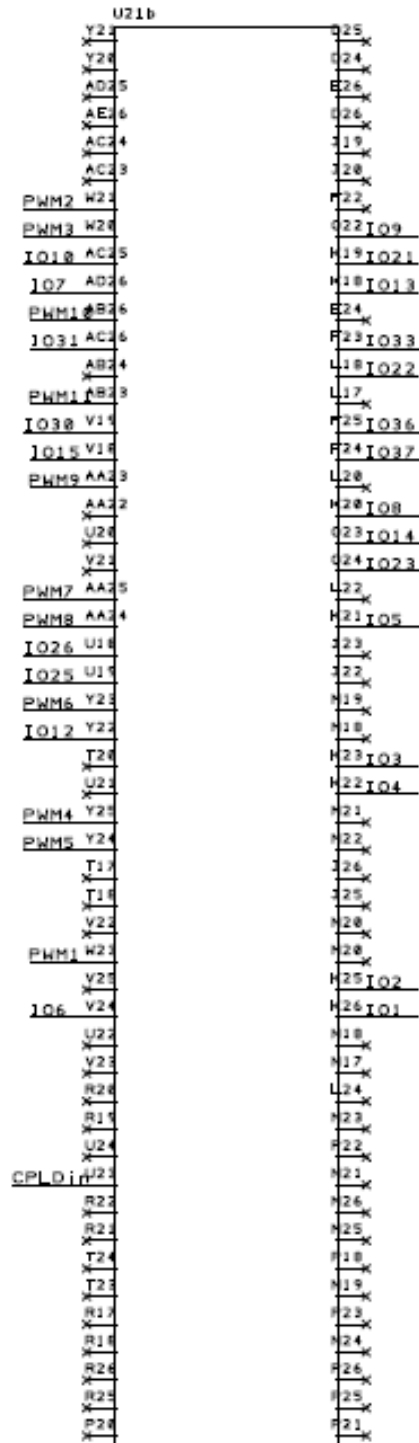
Figure 117: FPGA Bank1 Connections

162

Figure 118: FPGA Bank2 Connections

163
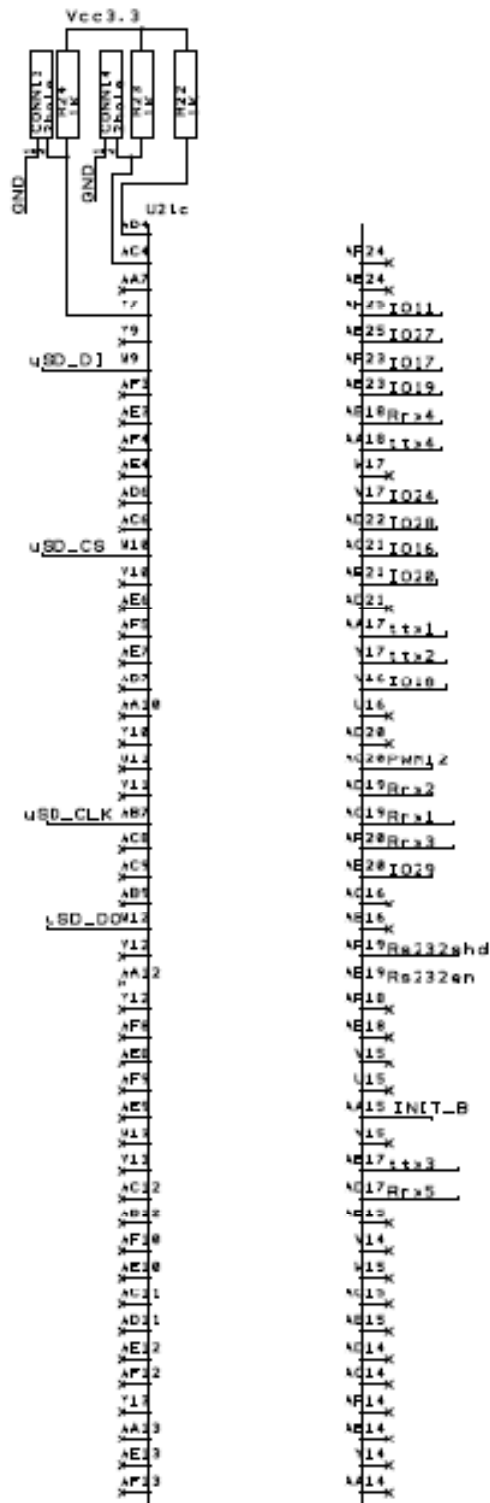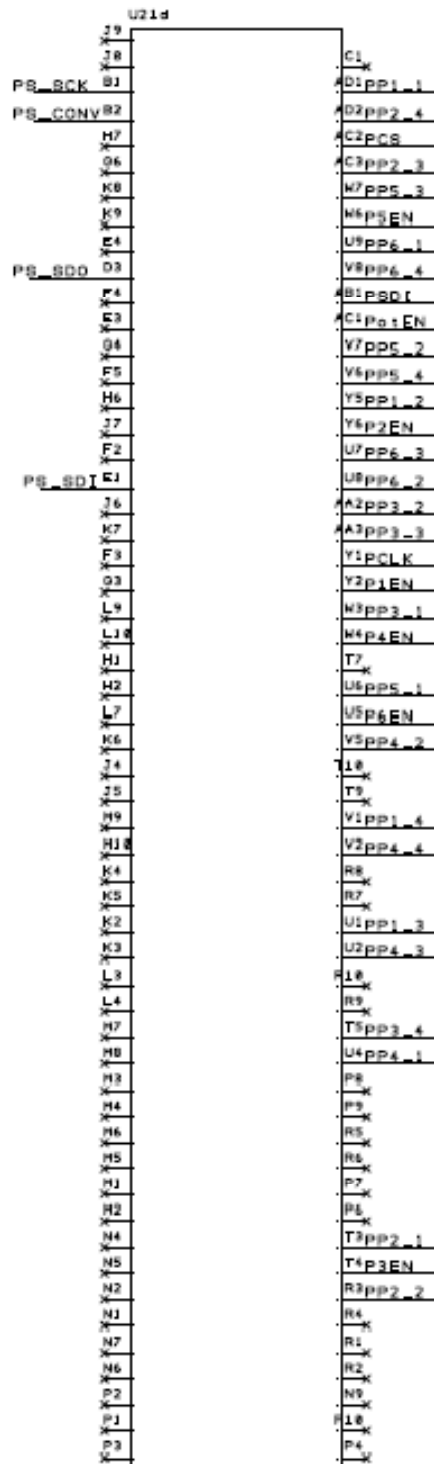
Figure 119: FPGA Bank3 Connections

164
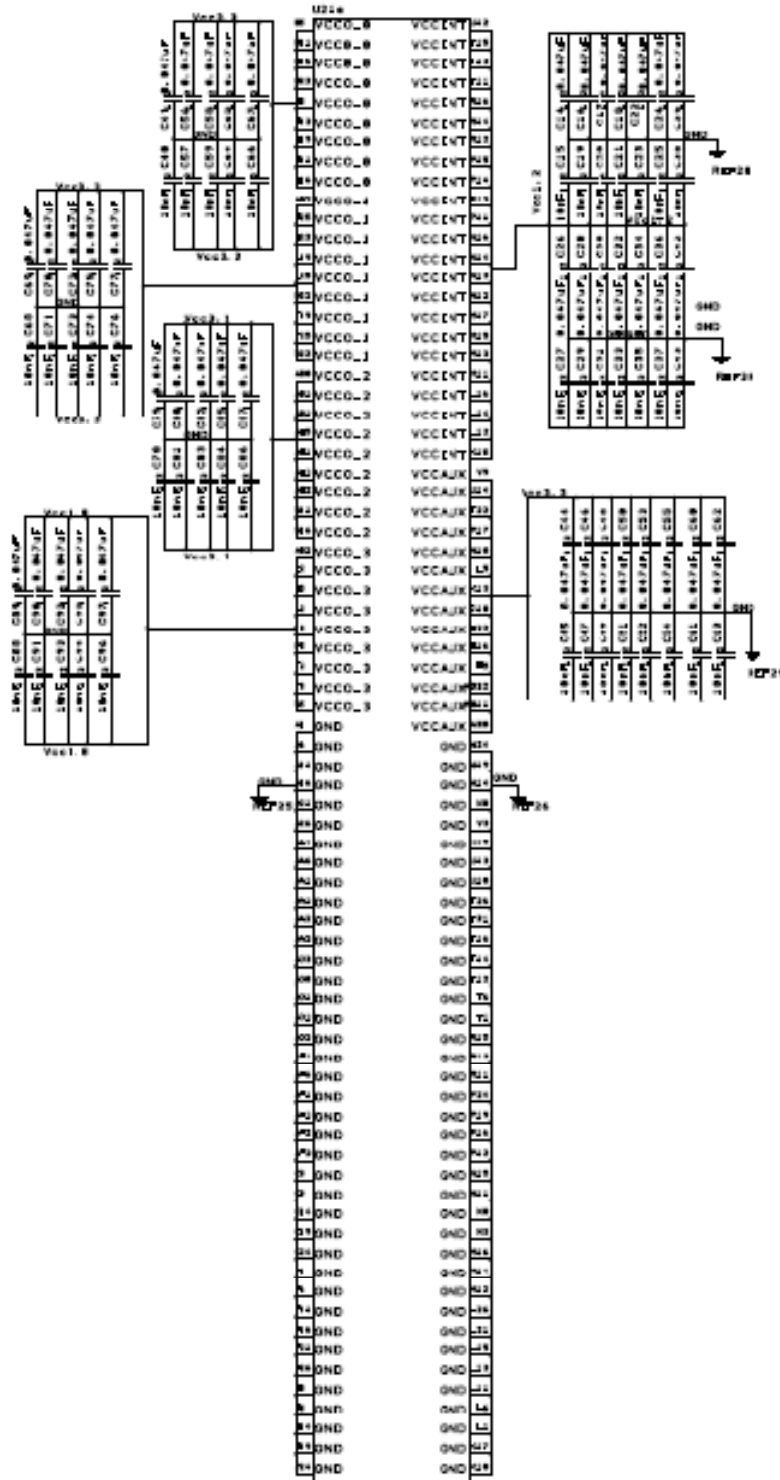
Figure 120: FPGA VCC Connections

165

Appendix C   (Continued)
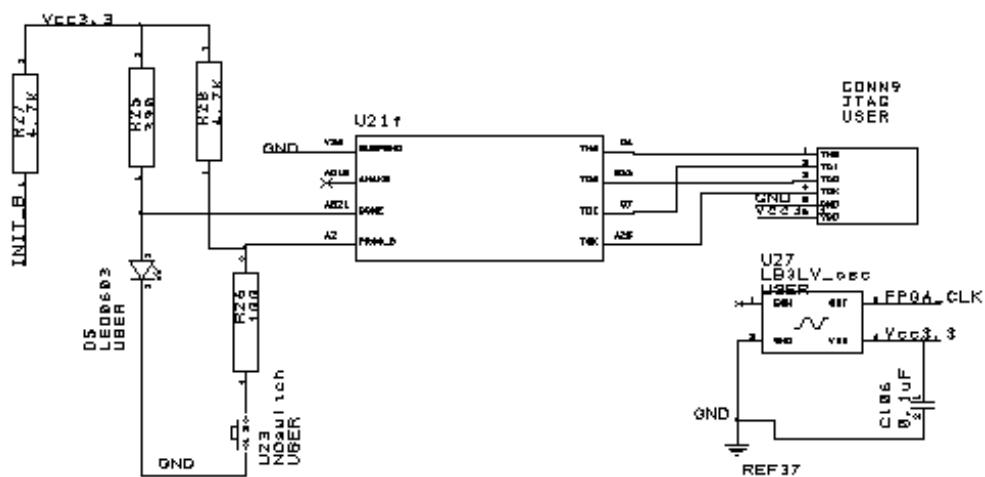
Figure 121: FPGA JTAG and Clock Circuit

Appendix D Safety Switch Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity SafetySwitch is
     Port(--CONTROL LOGIC INPUTS
               clk : in STD_LOGIC;
               pSel : in STD_LOGIC;
               dsp_sel1 : in STD_LOGIC:='0';
               dsp_sel2 : in STD_LOGIC:='0';
          --SERVO INPUTS
               pilot1 : in STD_LOGIC:='0';
               fpga1 : in STD_LOGIC;
               dsp1 : in STD_LOGIC:='0';
               pilot2 : in STD_LOGIC:='0';
               fpga2 : in STD_LOGIC;
               dsp2 : in STD_LOGIC:='0';
               pilot3 : in STD_LOGIC:='0';
               fpga3 : in STD_LOGIC;
               dsp3 : in STD_LOGIC:='0';
               pilot4 : in STD_LOGIC:='0';
               fpga4 : in STD_LOGIC;
               dsp4 : in STD_LOGIC:='0';
               pilot5 : in STD_LOGIC:='0';
               fpga5 : in STD_LOGIC;
               dsp5 : in STD_LOGIC:='0';
               pilot6 : in STD_LOGIC:='0';
               fpga6 : in STD_LOGIC;
               dsp6 : in STD_LOGIC:='0';
               fpga7 : in STD_LOGIC;
               dsp7 : in STD_LOGIC:='0';
               fpga8 : in STD_LOGIC;
               dsp8 : in STD_LOGIC:='0';
               fpga9 : in STD_LOGIC;
               dsp9 : in STD_LOGIC:='0';
               fpga10 : in STD_LOGIC;
               dsp10 : in STD_LOGIC:='0';
               fpga11 : in STD_LOGIC;
               dsp11 : in STD_LOGIC:='0';
               fpga12 : in STD_LOGIC;
               dsp12 : in STD_LOGIC:='0';
```

167

```vhdl
            --SERVO OUTPUTS
                servo1 : out STD_LOGIC;
                servo2 : out STD_LOGIC;
                servo3 : out STD_LOGIC:='0';
                servo4 : out STD_LOGIC:='0';
                servo5 : out STD_LOGIC;
                servo6 : out STD_LOGIC;
                servo7 : out STD_LOGIC:='0';
                servo8 : out STD_LOGIC:='0';
                servo9 : out STD_LOGIC;
                servo10 : out STD_LOGIC;
                servo11 : out STD_LOGIC;
                servo12 : out STD_LOGIC;
                SSout : out STD_LOGIC);
end entity SafetySwitch;

architecture Structural of SafetySwitch is
     signal pps,s : STD_LOGIC:='0';
     component single_switch is
        Port(pilot : in STD_LOGIC;
                    fpga : in STD_LOGIC;
                    dsp : in STD_LOGIC;
                    pilot_select : in STD_LOGIC;
                    dsp_select : in STD_LOGIC;
                    servo : out STD_LOGIC);
     end component single_switch;
     component freq_conv is
        Port(f : in STD_LOGIC:='0';
                 c: in STD_LOGIC:='0';
                 control_bit: out STD_LOGIC);
     end component freq_conv;
begin
     fc1: component freq_conv port map
                    (f=>pilot6, c=>clk, control_bit=>pps);
     --SERVOS CONTROLLING ROBOT DYNAMICS
     s1: component single_switch port map
                 (pilot=>pilot1, fpga=>fpga1,
                        dsp=>dsp1, pilot_select=>pps,
                        dsp_select=>dsp_sel1,
                        servo=>servo1);
```

168

```
     s2: component single_switch port map
                (pilot=>pilot2, fpga=>fpga2, dsp=>dsp2,
                        pilot_select=>pps,
                        dsp_select=>dsp_sel1,
                        servo=>servo2);
     s3: component single_switch port map
                (pilot=>pilot3, fpga=>fpga3, dsp=>dsp3,
                         pilot_select=>pps,
                         dsp_select=>dsp_sel1,
                         servo=>servo3);
     s4: component single_switch port map
                (pilot=>pilot4, fpga=>fpga4, dsp=>dsp4,
                        pilot_select=>pps,
                        dsp_select=>dsp_sel1,
                        servo=>servo4);
     s5: component single_switch port map
                (pilot=>pilot5, fpga=>fpga5, dsp=>dsp5,
                        pilot_select=>pps,
                        dsp_select=>dsp_sel1,
                        servo=>servo5);
     s6: component single_switch port map
                (pilot=>'0', fpga=>fpga6, dsp=>dsp6,
                        pilot_select=>pps,
                        dsp_select=>dsp_sel1,
                        servo=>servo6);
     --SERVOS CONTROLLING ACCESSORIES SUCH AS CAMERAS
     s7:component single_switch port map
                (pilot=>'0', fpga=>fpga7, dsp=>dsp7,
                        pilot_select=>'0',
                        dsp_select=>dsp_sel2,
                        servo=>servo7);
     s8:component single_switch port map
                (pilot=>'0', fpga=>fpga8, dsp=>dsp8,
                        pilot_select=>'0',
                        dsp_select=>dsp_sel2,
                        servo=>servo8);
     s9:component single_switch port map
                (pilot=>'0',fpga=>fpga9,dsp=>dsp9,
                        pilot_select=>'0',
                        dsp_select=>dsp_sel1,
                        servo=>servo9);
```

169

```
     s10:component single_switch port map
                   (pilot=>'0', fpga=>fpga10, dsp=>dsp10,
                          pilot_select=>'0',
                          dsp_select=>dsp_sel2,
                          servo=>servo10);
     s11:component single_switch port map
                   (pilot=>'0', fpga=>fpga11, dsp=>dsp11,
                          pilot_select=>'0',
                          dsp_select=>dsp_sel2,
                          servo=>servo11);
     s12:component single_switch port map
                   (pilot=>'0', fpga=>fpga12, dsp=>dsp12,
                          pilot_select=>'0',
                          dsp_select=>dsp_sel2,
                          servo=>servo12);
     SSout<=pps;
end architecture Structural;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity freq_conv is
     Port(f : in STD_LOGIC:='0';
          c: in STD_LOGIC:='0';
          control_bit: out STD_LOGIC);
end entity freq_conv;

architecture Behavioral of freq_conv is
     signal count: integer:=0;
     signal rst: STD_LOGIC:='0';
begin
     process (c) is
     begin
          if (c'event and c='1' and f='1') then
               count<=count+1;
          end if;
          if (f='0' and rst='1') then
               count<=0;
          end if;
     end process;
```

170

```vhdl
      Process (f) is
      begin
            if (f'event and f='0' and count>40000) then
                  if (count>65000) then
                        control_bit<='0';
                  else
                        control_bit<='1';
                  end if;
                  rst<='1';
            end if;
            if (f='0' and count=0) then
                  rst<='0';
            end if;
      end process;
end architecture Behavioral;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity single_switch is
      Port(pilot : in STD_LOGIC;
            fpga : in STD_LOGIC;
            dsp : in STD_LOGIC;
            pilot_select : in STD_LOGIC;
            dsp_select : in STD_LOGIC;
            servo : out STD_LOGIC);
end entity single_switch;

architecture Architectural of single_switch is
begin
      servo<=(not(pilot_select) and not(dsp_select)
            and fpga) or (not(pilot_select) and
            dsp_select and dsp) or (pilot_select
            and pilot);
end architecture Architectural;
```

171

ABOUT THE AUTHOR

Wendy received her Bachelor's degree from the University of South Florida in 1999. She worked part time in the area of embedded systems design for the signal conditioning industry while completing her master's degree at the University of South Florida. Wendy's master's thesis involved utilizing a second order Sliding Mode controller with DC/DC converters. As a teaching assistant in the Department of Electrical Engineering, she taught the controls laboratory and lectured in the undergraduate controls and microprocessor classes. While working on her PhD she received a fellowship from the Army Research Lab. In addition to embedded design, her interests include the design of controls for both ground and aerial vehicles and sensor integration.