Utah State University
DigitalCommons@USU

All Graduate Theses and Dissertations | Graduate Studies

8-2023

Developing Firmware for Space Weather Probes 2 Using HDL Coder

Nicholas L. Wallace
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/etd

Part of the Electrical and Computer Engineering Commons

Recommended Citation

Wallace, Nicholas L., "Developing Firmware for Space Weather Probes 2 Using HDL Coder" (2023). *All Graduate Theses and Dissertations*. 8800.
https://digitalcommons.usu.edu/etd/8800

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

DEVELOPING FIRMWARE FOR SPACE WEATHER PROBES 2 USING HDL CODER

by

Nicholas L. Wallace

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

Charles M. Swenson, Ph.D.
Major Professor

Jonathan Phillips, Ph.D.
Committee Member

Todd K. Moon, Ph.D.
Committee Member

D. Richard Cutler, Ph.D.
Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2023

Copyright © Nicholas L. Wallace 2023

All Rights Reserved

ABSTRACT

Developing Firmware for Space Weather Probes 2 Using HDL Coder

by

Nicholas L. Wallace, Master of Science

Utah State University, 2023

Major Professor: Charles M. Swenson, Ph.D.
Department: Electrical and Computer Engineering

This thesis will describe the process of designing, implementing, and testing the firmware for the Space Weather Probes 2 (SWP2). The Space Weather Probes (SWP) instrument suite provides measurements of Earth's ionospheric plasma, and was originally flown on the Scintillation Prediction Observation Task (SPORT) mission, completed jointly between the United States and Brazil. The firmware is developed using MATLAB/Simulink and deployed directly to an FPGA using HDL Coder. This thesis discusses data collection, CCSDS Space Packets creation, communication with the spacecraft computer, and the creation of Simulink blocks for reuse in other projects. This thesis also discusses problems that needed to be overcome for HDL Coder to be used to develop a full FPGA system.

(76 pages)

PUBLIC ABSTRACT

Developing Firmware for Space Weather Probes 2 Using HDL Coder

Nicholas L. Wallace

GPS and wireless communications are affected by interference from the ionosphere. Space weather affects plasma in the ionosphere, causing communication disruptions and reliability issues. To better understand how space weather affects the ionosphere, instruments are flown in space to collect data about the electrical characteristics of plasma in the ionosphere. Space systems require a lot of time and effort to develop and test. This thesis explores how a high level tool can be used to simplify the process and some obstacles that still exist with developing some space systems. To do this, the firmware architecture of a new version of the Space Weather Probes (SWP) was developed and documented in this thesis.

To my wife, Kinsey.

ACKNOWLEDGMENTS

I would like to thank my professor Dr. Swenson. His guidance in furthering the development of the Space Weather Probes instrument suite was invaluable. I would also like to thank those who worked on this project. Jason Powell, Benjamin Lewis, Rowan Antonuccio, and Justin Wellington worked hard to assist in developing the firmware, designing the analog components of SWP2, and developing the data analysis scripts for future work.

I would also like to thank my parents for teaching me to value knowledge and hard work. Lastly I would like to thank my wife Kinsey for her support and encouraging me to finish what I started.

Nicholas L. Wallace

CONTENTS

Page

ABSTRACT . . . iii

PUBLIC ABSTRACT . . . iv

ACKNOWLEDGMENTS . . . vi

LIST OF TABLES . . . ix

LIST OF FIGURES . . . x

ACRONYMS . . . xii

1 INTRODUCTION . . . 1
 1.1 Research Objectives . . . 2
  1.1.1 Can the entire Space Weather Probes instrument firmware be developed in the MathWorks MATLAB/Simulink environment and deployed to an FPGA? . . . 2
  1.1.2 Does the developed FPGA system perform the same in physical hardware as in simulation? . . . 2
  1.1.3 What issues need to be overcome for using MATLAB/Simulink to develop a complex system for a PolarFire FPGA? . . . 3
 1.2 SPORT Mission . . . 3
 1.3 Space Weather Probes . . . 3
 1.4 Space Weather Probes 2 . . . 5

2 OVERVIEW . . . 7
 2.1 System Overview . . . 7
 2.2 Methodology . . . 9
 2.3 Version Control . . . 9
 2.4 Literature Review . . . 11
 2.5 Thesis Outline . . . 12

3 COMMAND AND DATA HANDLING . . . 13
 3.1 CCSDS Space Packet Structure . . . 13
 3.2 Theory of Operation . . . 14
 3.3 Generic Telemetry Packet Structure . . . 16
  3.3.1 Telemetry Packet Creation . . . 17
 3.4 Packet Routing . . . 18
 3.5 Generic Telecommand Packet Structure . . . 20
  3.5.1 Telecommand Parser . . . 20

4 Interfaces and Drivers . . . 23
4.1 Theory of Operation . . . 23
4.1.1 Data Rates . . . 24
4.2 SPI . . . 24
4.2.1 Analog to Digital Converter . . . 25
4.2.2 Digital to Analog Converter . . . 25
4.2.3 Magnetometer . . . 26
4.2.4 Real Time Clock . . . 27
4.3 UART . . . 27

5 Data Processing Chain . . . 30
5.1 Theory of Operation . . . 30
5.2 Packet Structure . . . 30
5.3 Simulink Implementation . . . 31
5.3.1 FPP Processing . . . 32
5.3.2 FPP Granule . . . 33
5.4 Simulation Results . . . 34
5.5 FPGA Synthesis . . . 34
5.6 Resource Utilization . . . 36

6 Results . . . 38
6.1 Firmware development . . . 38
6.2 Physical Performance . . . 38
6.3 Issues with HDL Coder . . . 40
6.3.1 Resource usage . . . 40
6.3.2 Clock domains . . . 41
6.3.3 Multicycle Constraints . . . 43

7 Conclusion . . . 44
7.1 Further Work . . . 44

REFERENCES . . . 45

APPENDICES . . . 46
A Simulink Models . . . 47
B VHDL Modules . . . 57
C Data Processing Chains . . . 63
C.1 Command and Telemetry Dictionary . . . 63

LIST OF TABLES

Table | Page

3.1 List of Telemetry Packet APIDs and Sizes 14

3.2 List of Telecommand Packet APIDs and Sizes 16

3.3 Resource Utilization for Packet Router 19

4.1 Data rates by packet type 24

5.1 FPP Resource Utilization by component 36

5.2 FPP Total Resource Utilization 37

6.1 Filter Downsample resource utilization comparison 41

LIST OF FIGURES

Figure Page

1.1 SPORT Architecture overview 4
1.2 SWP Firmware Overview 5
1.3 SWP2 Firmware Overview 6
2.1 SWP2 Firmware Overview 8
2.2 Git directory structure 11
3.1 CCSDS Space Packet Structure [1] 13
3.2 CCSDS Packet Primary Header [1] 14
3.3 Command and Data Handling Theory of Operation 15
3.4 Generic Telemetry Packet 16
3.5 Telemetry Packetizer 17
3.6 Telemetry Packetizer Mask 18
3.7 Packet Router 18
3.8 Generic Telecommand Packet 20
3.9 Telecommand Parser 21
4.1 Spacecraft UART interface 23
4.2 UART state machine for spacecraft communication 24
4.3 ADC oversampling 26
4.4 DAC Operations 26
4.5 Magnetometer Interface 27
4.6 UART Transmit Model 28
4.7 UART Receive Model 28

5.1 FPP Granule 31
5.2 FPP data processing chain 31
5.3 FPP mask configuration 32
5.4 FPP Processing 32
5.5 FPP Configuration Block 33
5.6 FPP Granule Creation 33
5.7 FPP Processing Chain simulation 34
5.8 FPP ADC counts from test 36
6.1 Clock Domain Overview 39
6.2 Filter Downsample using Deserializer 41
6.3 Filter Downsample using in place accumulation 41
A.1 Packet Creation 48
A.2 Packet Router 49
A.3 Router FIFO Management 50
A.4 Router FIFO Management For Each Block 51
A.5 Router FIFO Management For Each Block 52
A.6 Telecommand Parser 53
A.7 UART Send 54
A.8 UART Receive 55
A.9 SPI Controller 56

ACRONYMS

ADC	Analog to Digital Converter
APID	Application Process Identifier
CCSDS	Consultative Committee for Space Data Systems
C&DH	Command and Data Handling
CRC	Cyclic Redundancy Check
CS	Chip Select
DAC	Digital to Analog Converter
EFP	Electric Field Probe
EFW	Electric Field Wave
FFT	Fast Fourier Transform
FIFO	First In First Out
FLP	Fixed Langmuir Probe
FPGA	Field Programmable Gate Array
FPP	Floating Potential Probe
GPS	Global Positioning System
HDL	hardware description language
INPE	Instituto Nacional de Pesquisas Espaciais
IP	Impedance Probe
ISS	International Space Station
ITA	Instituto Tecnologico de Aeronautica
LAPAN	Lembaga Penerbangan dan Antariksa Nasional
MAG	Magnetometer
MISO	Master In Slave Out
MOSI	Master Out Slave In
MSFC	Marshall Space Flight Center

PCB	Printed Circuit Board
QIP	Track-Q Impedance Probe
RAM	Random Access Memory
RC	Resistor Capacitor
RTC	Real Time Clock
RTL	Real Time Logic
RTR	Ready to Receive
SIP	Sweeping Impedance Probe
SLP	Sweeping Langmuir Probe
SoC	System on Chip
SoM	System on Module
SPI	Serial Peripheral Interface
SPORT	Scintillation Prediction Observation Research Task
SWP	Space Weather Probes version 1
SWP2	Space Weather Probes version 2
TIP	Tracking Impedance Probe
UART	Universal Asynchronous Receiver-Transmitter
USA	United States of America
USU	Utah State University
UTD	University of Texas at Dallas
VHDL	VHSIC Speed Integrated Circuit HDL

CHAPTER 1

INTRODUCTION

Field Programmable Gate Arrays (FPGA) are used for time critical applications, such as data acquisition and control systems. Application development for FPGAs requires testing and verification, and is more time consuming than software development on a computer due to not having descriptive error messages and needing to simulate all of the Real Time Logic (RTL) on a computer to retrieve debug information. Development time may be reduced through testing and verifying the behavior of individual design components before combining all of the components in a final design. Hardware Description Languages (HDL) are used to design the behavior of the system in the FPGA. VHSIC Hardware Description Languange (VHDL) and Verilog are two common HDLs which are used in research and industry to develop systems.

Both VHDL and Verilog allow the user to define modules of logic which can be reused throughout the design. Testbenches can verify behavior of these modules, either through test driven verification or the designer creating testbenches to check individual behaviors. High level tools may be used to facilitate rapid development by abstracting details away from the designer. This results in less efficient resource utilization without affecting the behavioral design. Using a high level tool also simplifies documentation, either through automatic documentation generation or graphical interfaces. Because details are abstracted away from the designer, some tasks require additional forethought for the designer to work with (e.g., sensitivity to rising and falling clock edges).

One high level tool which supports exporting a system representation to HDL is Mathworks Simulink combined with the HDL Coder toolbox provided by Mathworks. HDL Coder has been used to implement simple systems or specific functionality.

1.1 Research Objectives

The objective of this research thesis is to address the following questions during the development of the Space Weather Probes version 2 (SWP2) firmware:

1. Can the entire Space Weather Probes instrument firmware be developed in the MathWorks MATLAB/Simulink environment and deployed to an FPGA?
2. Does the developed FPGA system perform the same in physical hardware as in simulation?
3. What issues need to be overcome for using MATLAB/Simulink to develop a complex system for a PolarFire FPGA?

1.1.1 Can the entire Space Weather Probes instrument firmware be developed in the MathWorks MATLAB/Simulink environment and deployed to an FPGA?

Simulink has built in simulation tools and verification libraries which were leveraged to verify simulation behavior. These simulation tools also allowed wire values to be logged for analysis with various MATLAB scripts. The developed system contains multiple components, including:

- SPI drivers
- Signal processing blocks
- Command and data handling
- Spacecraft interface

1.1.2 Does the developed FPGA system perform the same in physical hardware as in simulation?

After verifying behavior in simulation, the system was placed on a PolarFire FPGA to verify behavior in hardware. The PolarFire FPGA was selected due to the flash-based firmware to be more resilient to radiation effects.

1.1.3 What issues need to be overcome for using MATLAB/Simulink to develop a complex system for a PolarFire FPGA?

Some known issues with developing HDL in Simulink is the higher resource utilization through abstraction. Methods for overcoming these limitations were explored. One limitation is Simulink is only sensitive to the rising edge of a clock, so the maximum speed of the SPI clock is half of the FPGA clock. Simulink also does not support bidirectional ports, but no chips on the SWP2 board used bidirectonal ports, so this limitation was not necessary to overcome.

1.2 SPORT Mission

The SPORT mission was a joint 6U CubeSat mission between the United States of America (USA) and Brazil [2]. The science goals of this space weather mission were to investigate the conditions that lead to the formation of plasma bubbles in the ionosphere. Instrumentation payloads were developed by organizations in the USA: Utah State University (USU), University of Texas - Dallas (UTD), Marshall Space Flight Center (MSFC), and Goddard Space Flight Center (GSFC). The Brazilian organizations, Instituto Tecnológico de Aeronáutica (ITA) and Instituto Nacional de Pesquisas Espaciais (INPE), provided the spacecraft, flight computer and ground station. The SPORT program was selected by NASA in December of 2016. USA partners received funding in the fall of 2017, and Brazil partners received funding in early 2018. The required USA-Brazil Framework Agreement allowing the two countries to work together was ratified in April 2018 and was signed in early 2019. Delivery of the completed Space Weather Probes from USU was completed in August 2020. Delivery of the spacecraft for launch occurred in July of 2022, with the launch occurring November 26, 2022 on a resupply mission to the International Space Station (ISS) followed by a release to orbit on December 29, 2022.

1.3 Space Weather Probes

The Space Weather Probes (SWP) were developed for the SPORT mission [2]. SWP was the first version of a collection of in-situ ionospheric diagnostic instruments and mea-

surement techniques developed by Utah State University (USU) for CubeSats. The instrument electronics were miniaturized and combined to a single 9 x 9 cm printed circuit board (PCB). The SWP was composed of the Sweeping Langmuir Probe (SLP) that produces both direct current (DC) measurements in the electron saturation region and I-V curves from a voltage sweep. The Floating Potential Probe (FPP), previously the Electric Field Probe (EFP), provided a monitor of the floating potential of the CubeSat during the voltage sweep of the Langmuir Probe and provided a measurement of the electric field using the double probe technique. The Wave Spectrometer was an on-board computation of both the high frequency electron density and electric field wave spectrum. The Sweeping Impedance Probe (SIP) provided observations of fundamental plasma resonances that occur at RF frequencies. The hardware architecture of SWP for the SPORT mission is shown in Figure 1.1 [2]. The hardware included the PCB, power regulation, and connectors to each probe.

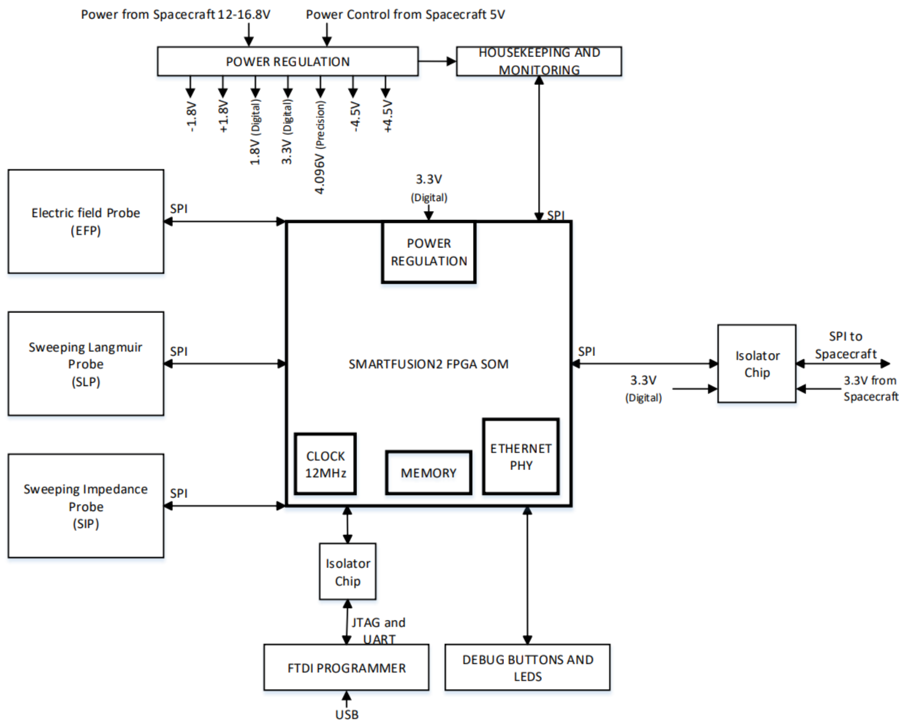


Fig. 1.1: SPORT Architecture overview

This first version of the SWP was controlled by a SoM, which included a SmartFusion2

FPGA and an integrated microcontroller. The FPGA interfaced with the physical probes to collect data and perform digital signal processing (DSP) and the microcontroller handled the creation and buffering of space packets before the packets were sent to the spacecraft over SPI (Figure 1.2). The firmware was developed using a mix of handwritten VHDL modules, open source VHDL modules, and code generated by Mathworks HDL Coder. The microcontroller was programmed using C. The SWP firmware buffered packets in a round-robin fashion until the spacecraft requested the next packet to be sent. Telecommands were sent from the ground configured the SWP firmware to operate in different science modes using each of the instruments.

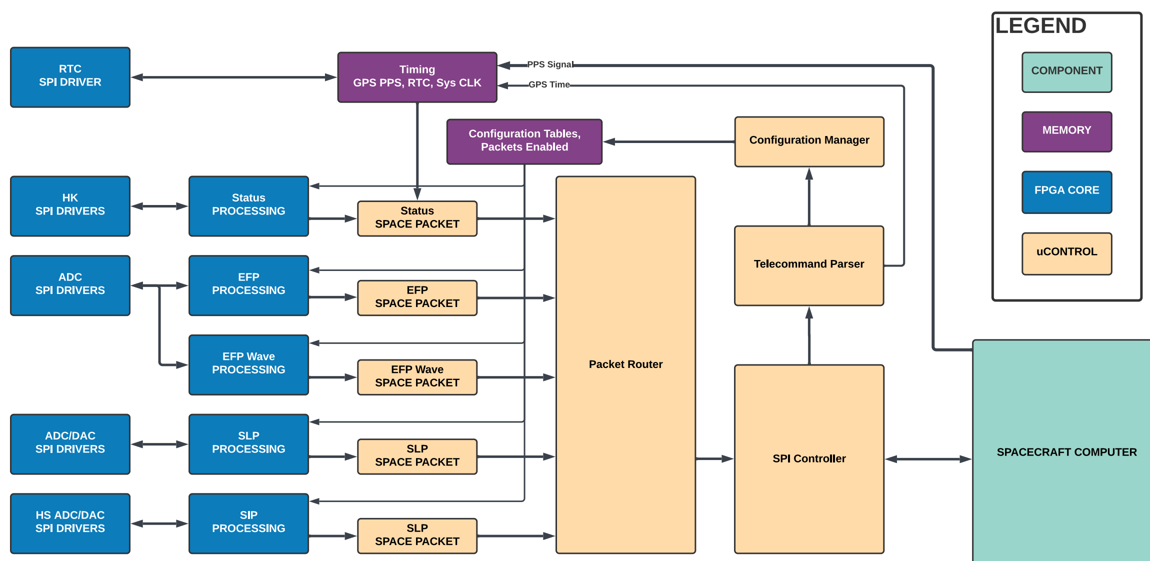


Fig. 1.2: SWP Firmware Overview

1.4 Space Weather Probes 2

Space Weather Probes 2 (SWP2) was the next generation of the Space Weather Probes instrument suite. This version was designed to be more modular to allow for better control of each science instrument. Packets were restructured to reduce overlap between science instruments and provide better time alignment during ground-based analysis. The connection with the spacecraft was changed from SPI to UART. SWP2 dropped the SoM in

favor of handling all of the SWP control within a single PolarFire FPGA to reduce interface complexity within the SWP2 firmware (Figure 1.3). The firmware was developed entirely within Simulink and HDL Coder to simplify simulation and documentation of the overall system.

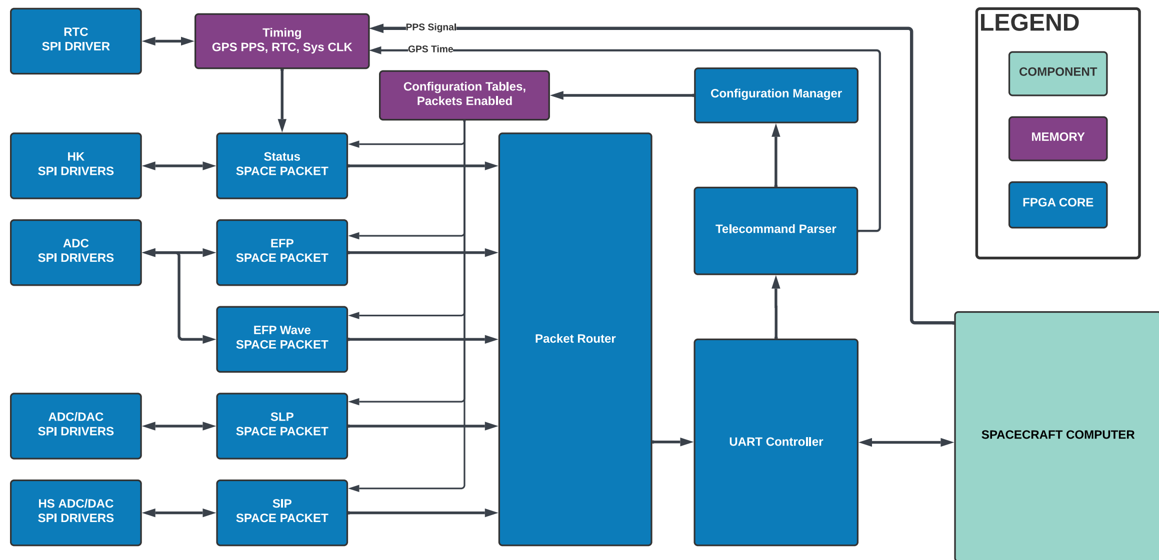


Fig. 1.3: SWP2 Firmware Overview

CHAPTER 2
OVERVIEW

2.1 System Overview

The SWP2 firmware includes several main components: device drivers, data processing, command and data handling, and configuration of each of these blocks. Each chip is controlled using SPI (see Section 4.2) and communication with the spacecraft is achieved using the Universal Asynchronous Receiver-Transmitter (UART) protocol.

To build the entire system, the researcher developed and tested generic framework blocks to simplify the process. These blocks were first tested in simulation to ensure functional correctness, then constraints from datasheets were added to the simulation, and finally the blocks were tested on hardware. The SPI drivers and Command & Telemetry Handling blocks were built first. Once these blocks were created and tested, data processing blocks were created for each instrument. A full data processing chain was included in Chapter 5 to illustrate how the full system would be built using these generic blocks.

The entire SWP2 firmware was developed in Simulink (Figure 2.1), exported to HDL using HDL Coder, and then imported into the Libero IDE for synthesis and bitstream generation. The firmware for each instrument involved collecting data from the SPI drivers and performing data processing, then being serialized into a granule, and finally being combined into a space packet to send to the ground. The Langmuir and impedance probes controlled DACs as part of the data collection and processing for the respective instruments. A status packet was created periodically to include housekeeping data about the state of the SWP2 suite and allow for time alignment of data packets with GPS time.

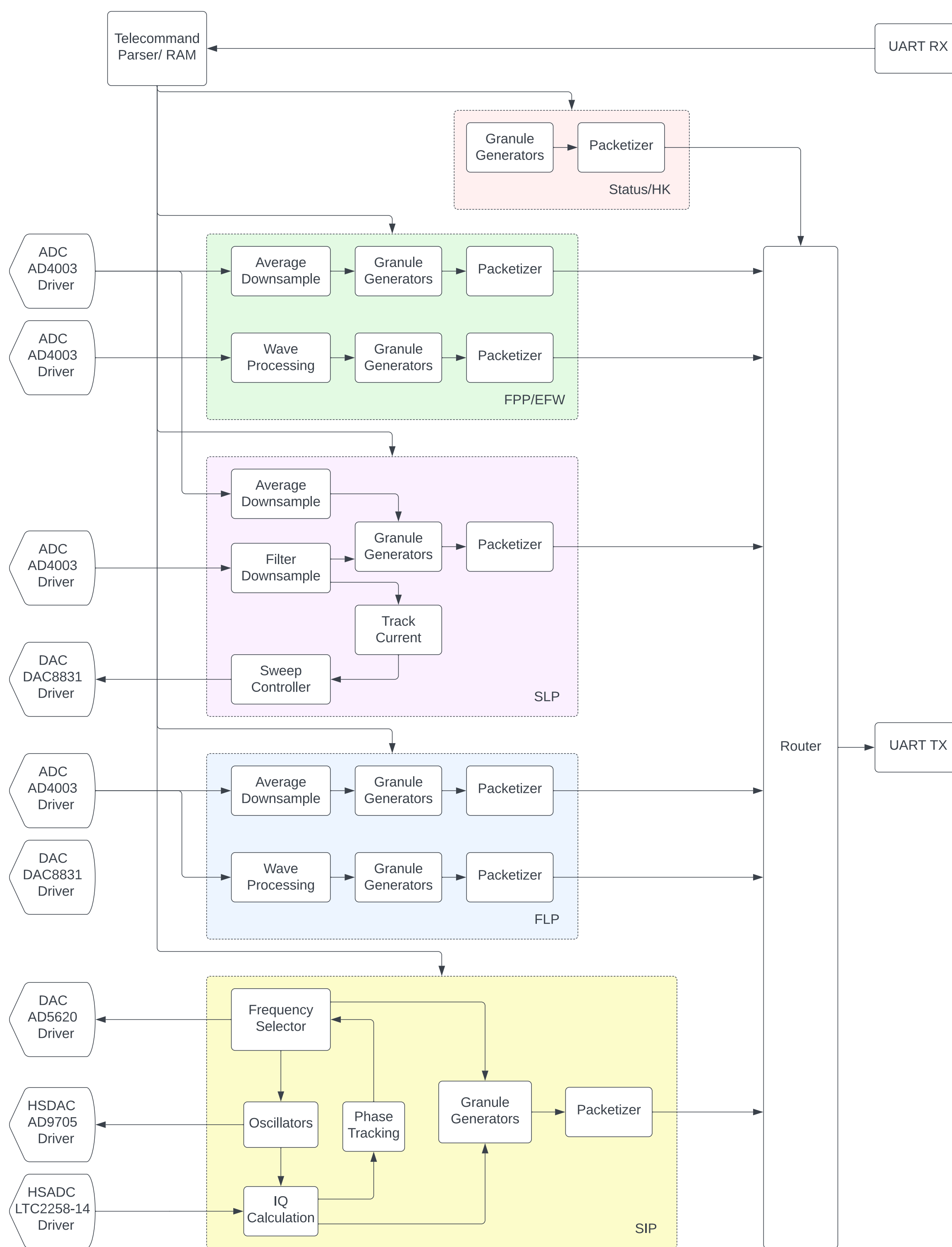


Fig. 2.1: SWP2 Firmware Overview

2.2 Methodology

To design the full SWP2 firmware, each individual part was built as a Referenced Subsystem block. These blocks allowed for easy component reuse and version control. Simulink could use Models instead of Referenced Subsystems to support design, but each individual Model was generated as an additional VHDL library. The additional VHDL library created from Model references made importing the generated HDL code into Libero more time consuming and error prone, so the researcher opted to use Referenced Subsystems instead.

Referenced Subsystems also allowed for easier simulation using Simulink's logic analyzer. The logic analyzer only allowed probing of lines within a model, so probing data lines between two different models to ensure functional correctness required multiple instances of the logic analyzer to be opened instead of only a single instance. Using subsystems in this way simplified integration testing during simulation.

Once each block worked in simulation, HDL Coder was used to export the full system to VHDL. The generated VHDL was imported into Libero as a single VHDL top module, with each subsystem building the hierarchy correctly. The constraints file was set in Libero, and testing on the PolarFire FPGA began at this point.

A Raspberry Pi was connected to the SWP2 test board to simulate the spacecraft and log the data sent over UART. This data was then processed and analyzed using MATLAB scripts to verify functionality. After the functionality was verified in hardware, the researcher continued working to develop the next blocks necessary for this project.

This process supported each research objective as the SWP2 firmware was entirely developed in Simulink, the firmware was verified to work in hardware, and individual issues with the workflow were discovered early in the development process instead of after full system integration.

2.3 Version Control

The SWP2 firmware was version controlled using git. A modified version of semantic versioning was used to track revisions for the data analysis scripts. The major number

was the Command and Telemetry Dictionary revision number, the minor number was a firmware change which would affect analysis such as digital gain, and the patch number was other firmware changes which would not affect the analysis or control scripts.

By matching the version number to be internally consistent with project documentation, the data analysis was simplified. Future work done on the project was also simplified by forcing the researcher to make specific comments about why changes were made in the commit messages.

The git repository was organized into several directories to manage different models and setup scripts (Figure 2.2). Models were stored in the corresponding directory under `model_library/`, with the corresponding test benches stored under `model_testing/`. Setup scripts were stored in `scripts/`. Simple analysis scripts were stored in the `analysis/` directory, but more robust analysis scripts were stored in a separate repository to support SWP data analysis along with SWP2 data analysis.

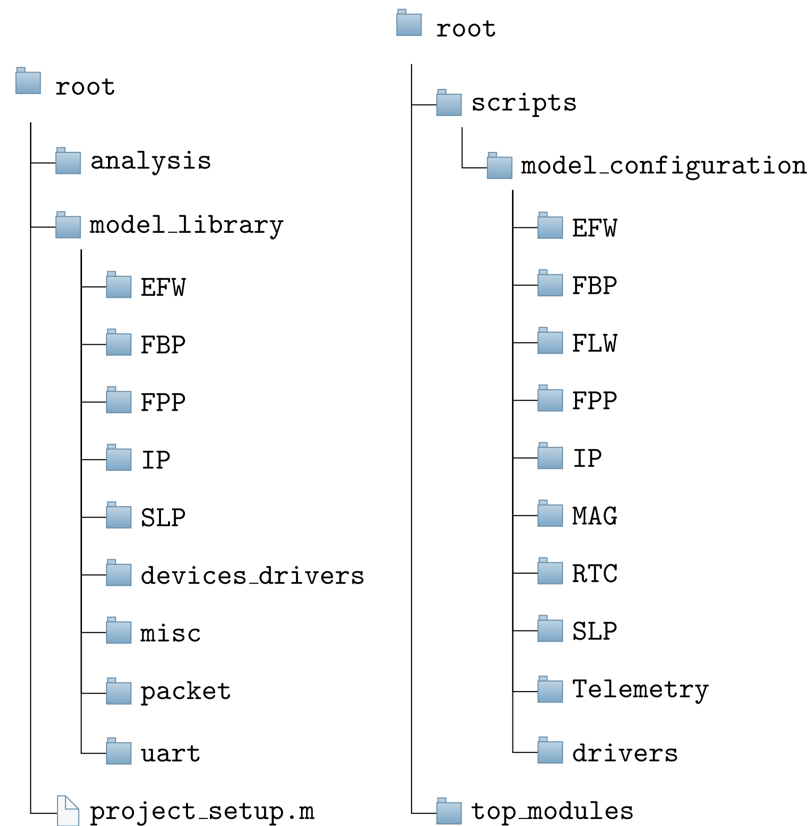


Fig. 2.2: Git directory structure

2.4 Literature Review

Various groups have used HDL Coder to develop firmware, but these groups have mostly implemented singular components or subsystems instead of the full FPGA system.

These projects include the CCSDS telecommand space packet decoder for the Indonesian LAPAN satellites [3], a comparison of HDL Coder generated HDL and hand written VHDL [4], and a comparison of digital filters generated by MATLAB/Simulink and Soft-Core [5]. HDL Coder was shown to use about 5% more Look Up Table (LUT) resources on the FPGA in most instances, but was more efficient in RAM block utilization.

A project by Hüner, Göker and Yeniçeri to develop a System on Chip (SoC) module for an ADCS system using an FPGA and a real-time application in an onboard processor was completed using HDL Coder [6]. The firmware generated by HDL Coder was shown to

meet project and timing requirements for this specific real-time application.

2.5 Thesis Outline

Chapter 2 includes an overview of the full SWP2 firmware and how it was developed. Chapter 3 discusses the Command and Data Handling (C&DH), including packet structures and overviews of the Simulink blocks. Chapter 4 discusses the physical interfaces and firmware drivers for each interface. Chapter 5 discusses developing a specific data processing chain and and testing on the FPGA. Chapter 6 discusses the results of the research objectives. Chapter 7 includes information on further work related to this research.

CHAPTER 3

COMMAND AND DATA HANDLING

A major component of the SWP2 firmware architecture is correctly handling and packetizing data generated by the instruments and properly parsing and applying telecommands. This chapter discusses the packet structures used, the theory of operation, and the models built in Simulink to accomplish the Command and Data Handling (C&DH) for this research.

3.1 CCSDS Space Packet Structure

For this project, the CCSDS Space Packet Protocol standard CCSDS 133.0-B-2 was used [1]. This standard defines a header and framing information for each packet. A packet header consists of 6 octets, or 8-bit numbers, followed by a data field of up to 65536 octets (Figure 3.1). The Packet Primary Header includes information about the version of the space packet, its location in the data stream, and the data field structure and size (Figure 3.2).

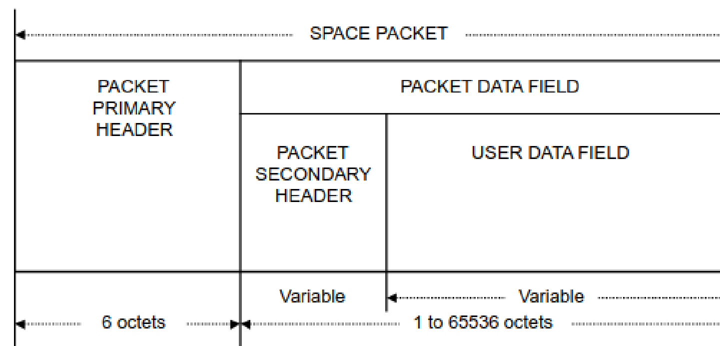


Fig. 3.1: CCSDS Space Packet Structure [1]

Each packet type is assigned a mission-specific Application Process Identifier (APID) to identify which packets correspond to each instrument. On missions with many instruments, such as the ISS, where a limited amount of APIDs are available, a secondary header is

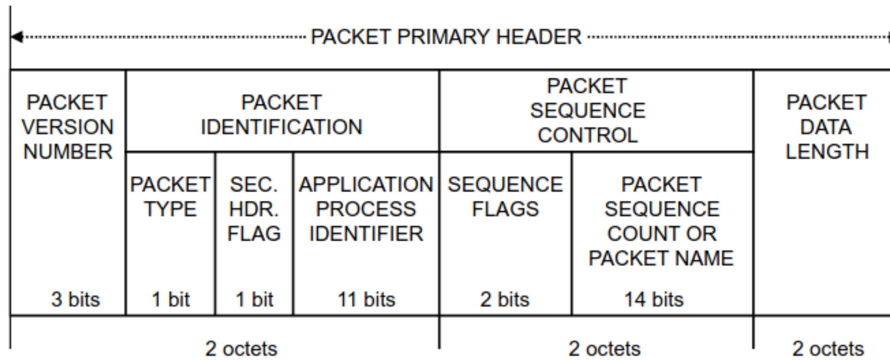


Fig. 3.2: CCSDS Packet Primary Header [1]

defined by the instrument. For smaller missions, like SPORT, each packet type is assigned its own APID to simplify routing and data overhead.

3.2 Theory of Operation

Every SWP2 packet type has a unique APID and packet length (Table 3.1). Instead of reusing the APID from the telemetry data packet and setting the Packet Type bit in the header, telecommand packets each have their own APID. It was decided to simplify ground-based analysis scripts to ensure the data structure is the same for all packets sharing an APID, and is not dependent on whether the packet is a telemetry or telecommand packet.

APID	Packet Name	Granule Count	Packet Size (bytes)
0x020	FPP 12	200	1015
0x021	FPP 34	200	1015
0x022	EFP Wave	10	495
0x024	SLP Full Sweep	32	271
0x025	SLP Fast Sweep	16	143
0x026	FLP	10	45
0x027	FLP Wave	10	495
0x028	SIP Sweep	512	2575
0x029	SIP Track	100	315
0x02A	SIP Track Q	200	3215
0x02B	Magnetometer	100	815
0x02C	Config Echo	1	5367
0x02D	Status	1	56

Table 3.1: List of Telemetry Packet APIDs and Sizes

In the SWP2 firmware, telemetry packets are generated from data granules from each science instrument (Figure 3.3). These packets are then streamed from the SWP2 instrument to the spacecraft, which are then buffered by the spacecraft computer. The SWP2 firmware includes a Packet Router, which multiplexes parallel data streams of packets into a serial stream of packets. This allows the packets to be sent to the spacecraft over UART, but any serial protocol could be used from the output of this block. SWP2 can also buffer data for several seconds to allow the spacecraft computer to handle other data streams as necessary.

In addition to generating telemetry packets, the SWP2 firmware also processed telecommand packets sent from the ground to modify operating parameters. Some of the telecommand packets contained tables to adjust the voltage sweep of the SLP and SIP (Table 3.2). To ensure proper operation of the SWP2 instrument suite, the telecommand parser ignored incomplete or corrupted packets.

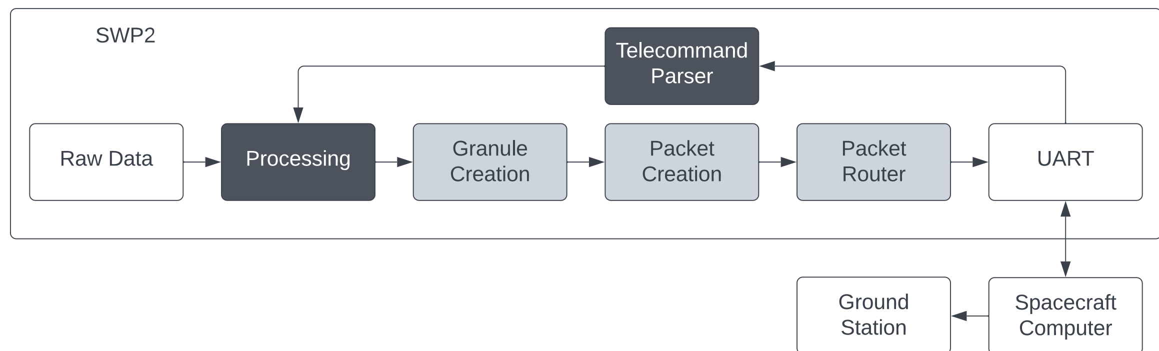


Fig. 3.3: Command and Data Handling Theory of Operation. *Gray boxes*: Telemetry data. *Black boxes*: Configuration commands

APID	Packet Name	Packet Size (bytes)
0x030	FPP	12
0x031	EFP Wave	72
0x032	SLP	2112
0x033	FLP	12
0x034	FLP Wave	72
0x035	SIP Sweep	1032
0x036	SIP Track	12
0x037	SIP Track Q	16

Table 3.2: List of Telecommand Packet APIDs and Sizes

3.3 Generic Telemetry Packet Structure

For each science instrument in SWP2, the firmware collected and buffered individual data granules until enough granules had been collected to form a packet. Each telemetry packet followed the generic telemetry packet structure (Figure 3.4), where only the APID, packet length, and packet sequence count fields change in the Space Packet Header. The system clock and Real Time Clock (RTC) values are latched when the first granule is completed. Immediately following the header and timing information, data granules are added sequentially to the packet. The checksum for each packet is calculated using the CRC-16 algorithm [7] for the entire packet, including the primary header, and is appended at the end of packet.

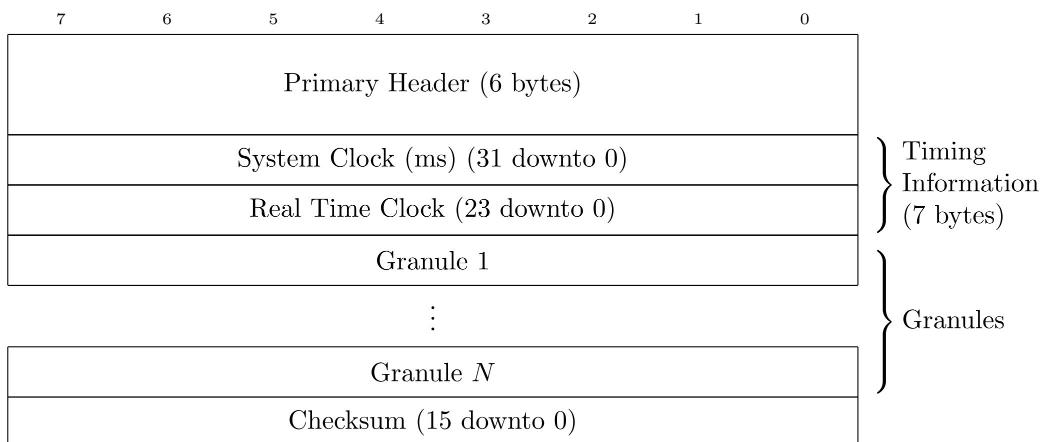


Fig. 3.4: Generic Telemetry Packet

After packets were received on the ground, they were ordered by the RTC time. This is due to the system clock, accurate to the millisecond, which rolls over approximately every 50 days. The RTC chip is not very accurate due to using the built-in RC oscillator instead of an external crystal oscillator, but the RTC chip is used for keeping packets in order instead of accurate timekeeping. Status packets are used to match the RTC and system clock to the GPS time during ground-based analysis.

3.3.1 Telemetry Packet Creation

To handle telemetry packets, a generic Simulink block was created (Appendix A.1). A simplified block diagram is included in this section (Figure 3.5). This block buffers all incoming granules. The block mask allowed the designer to specify the APID of the packet, granule size, and granule count of the packet (Figure 3.6). This block calculated the total packet length using the granule information from the mask and the structure of the generic telemetry packet to correctly generate the header and checksum. Once enough bytes were received, a full packet was output to the custom Packet Bus, which is discussed more in Section 3.4.

To reduce power draw in the FPGA, this packetizer block was held in reset when the corresponding science instrument was disabled, to prevent any gates or downstream flip-flops from changing states. This block was run at 2 MHz instead of the global 160 MHz clock required for some data processing, further reducing power requirements.

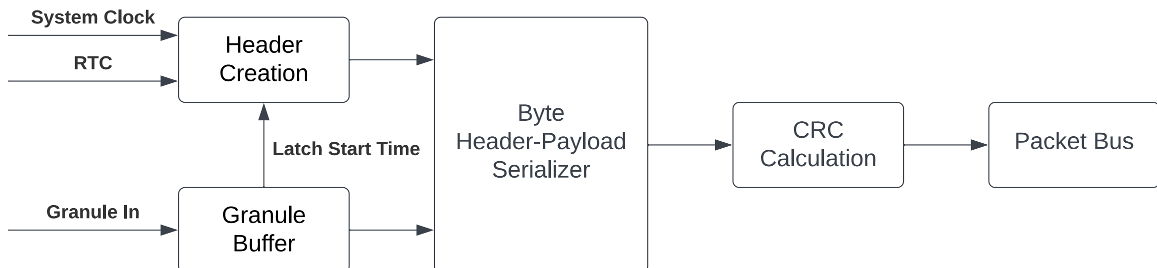


Fig. 3.5: Telemetry Packetizer

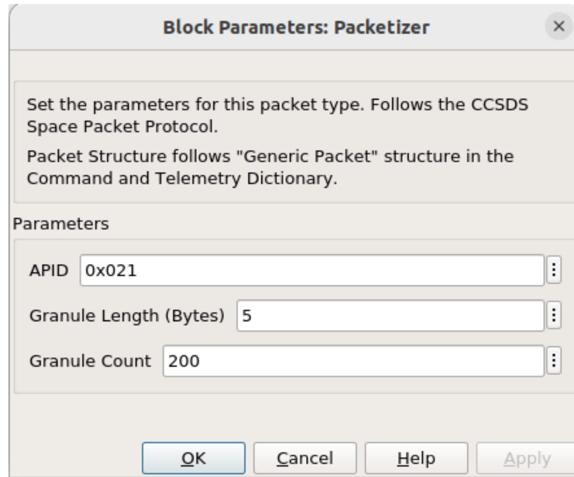


Fig. 3.6: Telemetry Packetizer Mask

3.4 Packet Routing

A generic packet router was created in Simulink (Appendix A.2) to handle buffering of each packet type, with a simplified block diagram included in this section (Figure 3.7). A buffer was created for each packet type to ensure each packet type could be handled correctly before being multiplexed. A state machine checked if any packets were ready to be buffered into RAM for transmission in a round robin fashion. If the transmission buffer became too full, the oldest packet was dropped to ensure no partial packets were transmitted or stored. No additional priority was given to which packet type was dropped from the buffer.

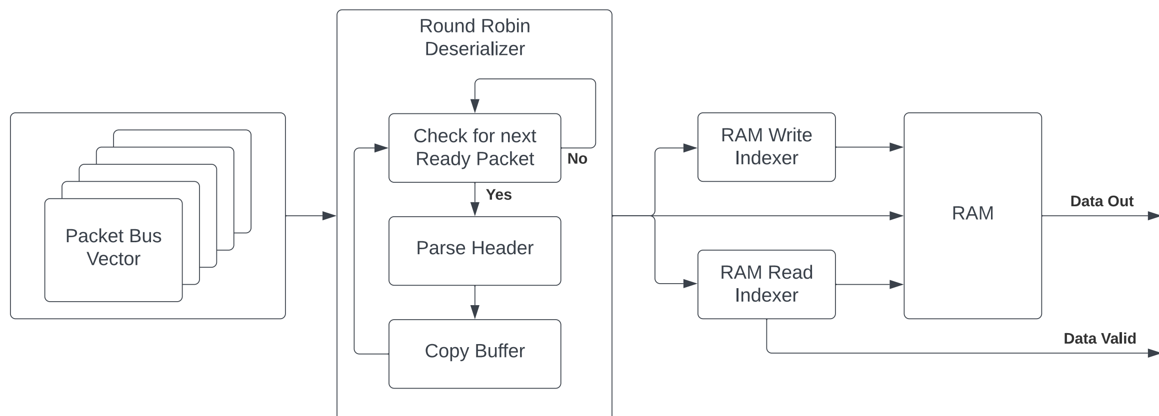


Fig. 3.7: Packet Router

To handle all of the various packet types and simplify routing between blocks, a Packet Bus was created. This bus included a data line, whether the data was a valid byte, whether the packet was completed on this clock cycle, and whether the packet was enabled. If the packet was not enabled, the state machine continued to check for whether any complete packets needed to be buffered to RAM. If only an incomplete packet existed, the buffer for that packet was flushed.

Utilizing a "For Each" Subsystem in Simulink, this generic packet router only required a vector of Packet Buses as input. Based on the size of this input vector, enough individual buffers were created. The state machine determined how many bytes to copy based on the packet header. This simplified the design of the rest of the firmware as any number of packet generators could be added to the vector and no changes needed to be made to this block to handle all packets that were sent to the spacecraft.

The resource utilization of the packet router was found during testing on the PolarFire FPGA (Table 3.3). Using this information, an estimate of resource utilization was created for however many packets would be needed so this packet could be used for other projects. The only change caused by increasing the number of packets was the replication of the For Each block, leading to additional input lines for the state machine and a First-In First-Out (FIFO) buffer for the incoming packet bytes. After several values were tested, a full design with 9 different packet types was synthesized and compared to the estimated usage. The interface resources used were higher than expected, but both the interface and fabric utilization were close to the estimated values.

Packet Count	Fabric LUT	Fabric DFF	Interface LUT	Interface DFF
2	892	459	1764	1764
3	1031	530	2052	2052
4	1164	601	2340	2340
5	1311	672	2628	2628
7	1584	814	3204	3204
9	1886	803	2916	2916

Table 3.3: Resource Utilization for Packet Router

3.5 Generic Telecommand Packet Structure

For the SWP2 instrument, various operating modes were required for each of the science instruments. Each science instrument could be enabled or disabled or have its specific configuration parameters changed while on orbit. To facilitate this, a generic telecommand payload structure was defined (Figure 3.8). This structure was nearly identical to the generic telemetry packet, but replaced the data granules with configuration parameters and did not include the timing information. The CRC-16 of the packet was still calculated and appended to the end of the packet.

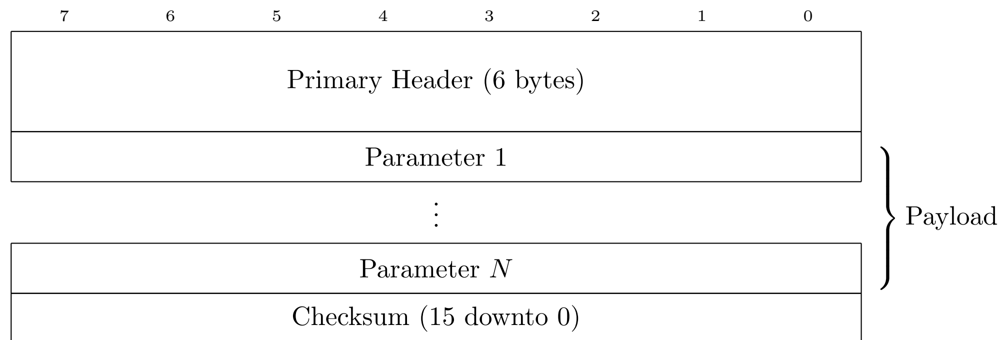


Fig. 3.8: Generic Telecommand Packet

These configuration parameters could not be modified while the corresponding science instrument was operating. A status packet was generated whenever a science instrument was turned on or off, changing the operating mode. The status packet included the configuration parameters for each instrument to ensure the parameters were correctly documented for data processing on the ground.

3.5.1 Telecommand Parser

The generic telecommand parser was created in Simulink using a large state machine and several combinatorial sections to calculate the checksum and header validity (Figure A.6). A simplified block diagram of the state machine was included in this section (Figure 3.9). This model had two inputs, data in and data valid, and a single output bus. All incoming data was stored in a circular buffer. The data was stored in a circular buffer

instead of a FIFO in case an invalid or incomplete packet was received to recover the following valid and complete packets.

Once 6 bytes were buffered, a check was performed to determine whether the 6 bytes formed a valid CCSDS Space Packet header. A header was defined as valid if all of the following values matched an entry in the telecommand dictionary: packet version number, packet type, sequence flags, APID, and packet length. If a valid header was found, the parser waited for the rest of the payload to be buffered.

Once the full packet was buffered, the CRC at the end of the packet was checked for correctness. If the CRC was correct, the payload bytes were sent out over the telecommand bus with the corresponding APID for other blocks to process the configuration. If the CRC was invalid, the parser searched forward through the buffer to find the next valid header. This accounted for incomplete packets being sent. If a corrupted packet was received, the packet was discarded and a resend request packet was sent to the ground.

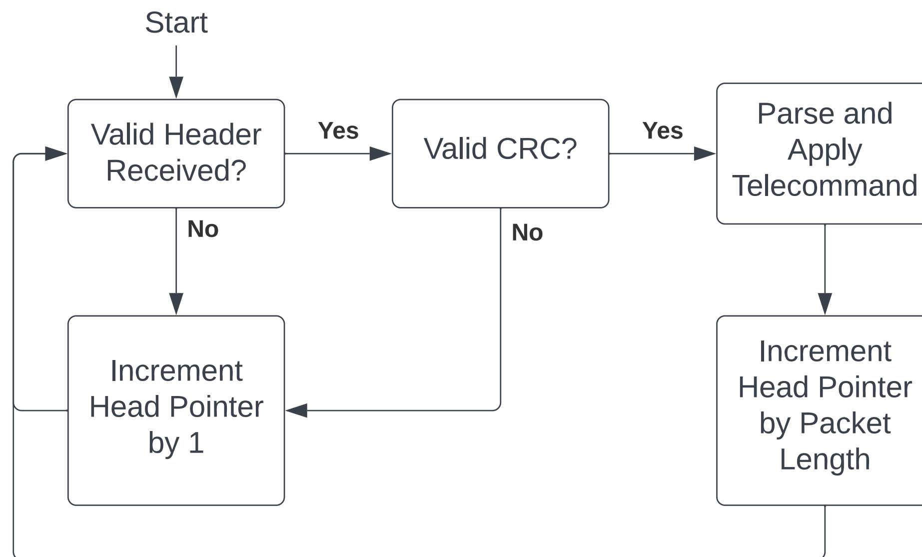


Fig. 3.9: Telecommand Parser

In the event a packet was not completely sent, a watchdog marked the header as invalid if no additional bytes were received within 5 seconds of waiting. This allowed the parser to recover from hanging indefinitely if a large packet transmission was not completed.

Analysis was completed to determine the minimum FPGA clock rate to handle the worst-case scenario of invalid packets being received and the parser recovering the rest of the data stream. The data stream could be recovered even using the maximum size packet by the Space Packet standard. If the 65,636-byte buffer was completely filled, the parser could still recover. If a valid header was found, the data would be processed before being overwritten by the incoming bytes. If an invalid header was found, the head pointer would update before a new byte could be read, preventing the buffer from overflowing.

CHAPTER 4

Interfaces and Drivers

This chapter discusses how the SWP2 firmware interfaces with the spacecraft and the science instruments. A high-level theory of operation is presented, then each interface and device is discussed in greater detail.

4.1 Theory of Operation

To collect data from the science instruments, the SWP2 firmware used the SPI protocol. All of the data brought in from the science instruments went through an ADC and was controlled through a DAC. Supplementary information was provided by the RTC over SPI. Once the raw data samples were collected and processed onboard, packets were sent over UART to the spacecraft. The spacecraft could also control the SWP2 instrument suite over UART by forwarding telecommand packets from the ground (see Section 3.5).

The spacecraft could control when data was streamed from SWP2 by controlling a Ready to Receive (RTR) line from the spacecraft to SWP2 (Figure 4.1). The SWP2 firmware was always ready to receive data from the spacecraft, so no RTR line existed from SWP2 to the spacecraft. The spacecraft could also request a packet to be resent from SWP2 if the calculated CRC did not match the CRC at the end of the packet (Figure 4.2). If the spacecraft did not signal to the SWP2 firmware that the packet needed to be resent, the next packet was sent and operation continued normally.

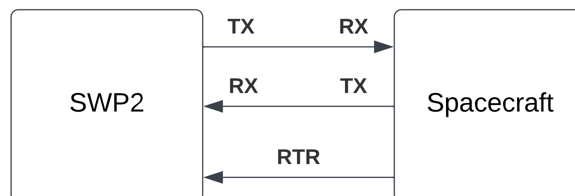


Fig. 4.1: Spacecraft UART interface

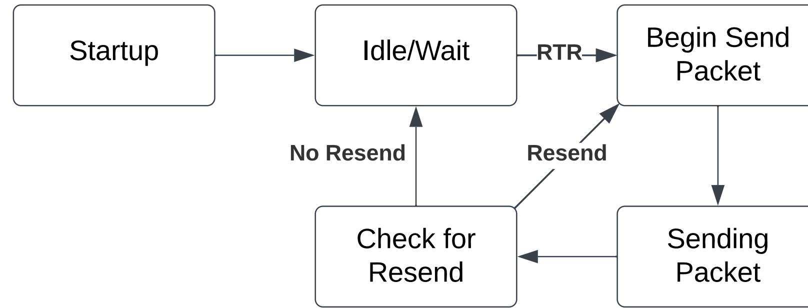


Fig. 4.2: UART state machine for spacecraft communication

4.1.1 Data Rates

The required throughput of data packets to the spacecraft was calculated using the packet size and generation frequency. Table 4.1 shows the minimum required throughput. Due to some packets being mutually exclusive, only the worst-case scenario of packets being enabled is shown to determine the minimum data rate. Assuming no packets required a resend, the minimum data rate required to get all packets to the spacecraft is 37,223 bits per second or about 4.6 kB per second.

Packet Name	bps
Status	9
FPP 12	4060
FPP 34	4060
EFP Wave	4830
SLP Full Sweep	18
FBP Wave	4830
SIP Track Q	12860
Magnetometer	6520
Config Echo	36

Table 4.1: Data rates by packet type

4.2 SPI

SWP2 communicated with all of the ADCs, DACs, magnetometers, and the RTC over SPI. Because some chips used SPI Phase 0 and others used SPI Phase 1, two different SPI drivers were written. Each of these SPI drivers could be configured to use either clock

polarity, and were configured using a mask. Initially, an attempt was made to create a generic SPI driver block that would work with all phase and polarity combinations, but this caused problems with Simulink Charts.

When the first version of the SPI driver was created, two different charts were used. One of these charts controlled the clock, and one controlled data handling. HDL Coder only allowed the use of the rising or falling edge of the clock, so the SPI driver was unable to read on one edge and write on the other as SPI expected. To account for this, the Simulink chart was run at twice the SPI clock rate, allowing one state to assert the clock line and the next to deassert it.

The second version combined both state machines into one to reduce the complexity of the SPI driver and rais the fmax to 160 MHz (discussed more in Section 6.3.2). The final version of the SPI drivers incorporated both clock polarities and 3 different models created. The three model configurations used: a preset word sent for every transaction (such as sampling an ADC), multiple words sent for various transactions (such as configuring the RTC), and accepting a word input (such as a DAC word).

4.2.1 Analog to Digital Converter

Several ADCs were used in SWP2. Most of the instruments used an AD4003, which was communicated with through SPI. This ADC did not need to receive data from SWP2, and performed a conversion whenever CS was deasserted. For each instrument, the ADC was oversampled and the raw data was then averaged to reduce the noise from individual readings before being put into granules at the correct data rate expected by the packet structure (Figure 4.3).

For the IP measurements, the high-speed LTC2258-14 ADC was used instead of the AD4003 due to required sample rate of 80 MHz. This high speed ADC allowed for more data throughput by using 8 data lines instead of a single data line.

4.2.2 Digital to Analog Converter

For the slower measurements of the SLP, the DAC8831 was used. This chip did not

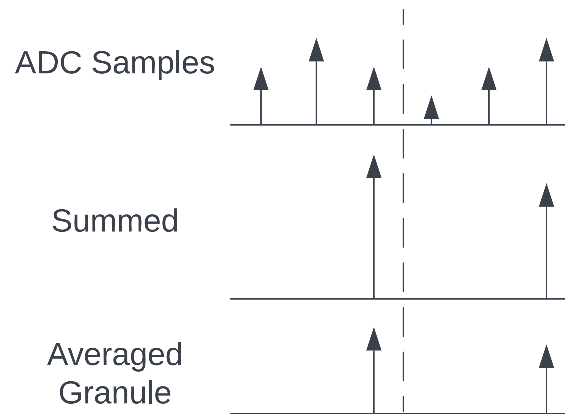


Fig. 4.3: ADC oversampling

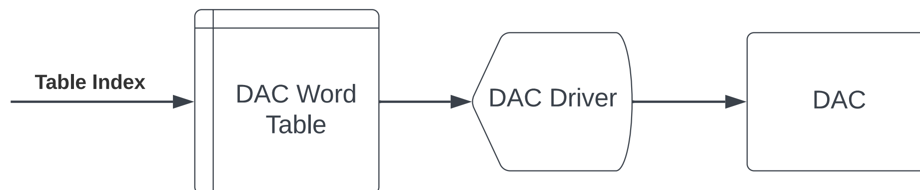


Fig. 4.4: DAC Operations

send any data back to the SWP2 firmware. The chip only needed a DAC word to control what analog output was needed and a convert line to start the transaction. The DAC words of the sweep were stored in a data table on the FPGA which was indexed through (Figure 4.4). Subsections of this table were used to achieve different sweep parameters, and the table could be updated using telecommand packets.

For the faster IP measurements, the AD9705 high-speed DAC was used. This DAC was driven by a separate table of DAC words for IP measurements.

4.2.3 Magnetometer

To provide orientation data, the SWP2 firmware included the ability to read from multiple magnetometers and correlate the data on the ground to reduce the noise of magnetometer measurements. Each magnetometer used a shared SPI bus, except for multiple MISO lines (Figure 4.5). This allowed the reuse of the CLK, CS, and MOSI lines for simpler routing. Each magnetometer was polled at the same time using this method.

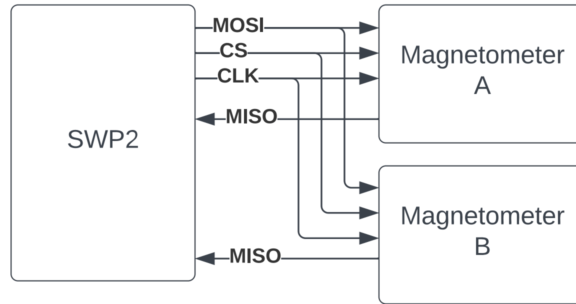


Fig. 4.5: Magnetometer Interface

4.2.4 Real Time Clock

SWP2 used a Real-Time Clock (RTC) chip to keep track of time when the FPGA was not powered. This RTC time was mainly used to ensure packets were ordered correctly during ground-based analysis. The system clock reset during a power cycle of the FPGA, so the RTC time was included with every packet to ensure coarse alignment, with the actual time stamping being performed using the system clock and GPS time from status packets. The RTC also stored some data from the FPGA in RAM to preserve the data between power down and power up events.

To achieve these goals, the AB0815 chip was selected [8]. This chip was controlled over SPI and contains a low-power internal RC oscillator. The RTC chip was polled every few seconds to determine the number of minutes since an epoch, and this value was latched inside the FPGA to reduce the power draw from polling the RTC continuously. The epoch used was the "launch date" of the instrument, but could be modified using the MATLAB configuration scripts.

4.3 UART

The decision to use UART to communicate with the spacecraft was due to the low number of wires and simple implementation. The UART communication between the spacecraft and the SWP firmware runs at 115,200 baud with a single stop bit and no parity bits, but the baud rate could be changed using the configuration scripts. This fulfilled the data rate requirements mentioned in Section 4.1.1 and allowed for packets to be resent if needed.

UART modules were created in Simulink instead of using Libero IP cores to facilitate the full SWP2 firmware being developed in Simulink. This led to some data type complications, but UART modules were successfully created and tested.

Two different Simulink models were created: UART Transmit and UART Receive. The UART Transmit model sent a single byte when signaled by a start line and asserted a `Data_Done` line once the transaction was completed (Figure 4.6). This model did not buffer additional incoming bytes, so the data source needed to buffer the bytes correctly. For SWP2, this buffering was already completed using the resend functionality of the Packet Router, so no additional buffering was needed for the UART Transmit model.

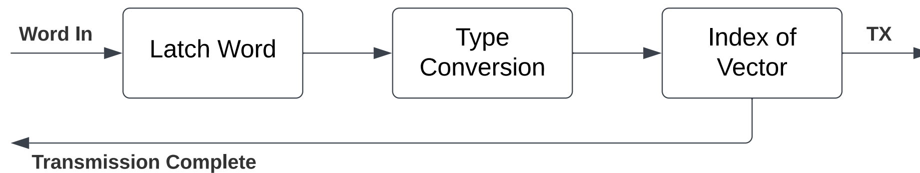


Fig. 4.6: UART Transmit Model

The UART Receive model oversampled the incoming data to reduce bit errors (Figure 4.7). The oversampling rate was determined by the FPGA clock rate, and the bit was determined to be asserted if more than half of the samples were asserted. A small settling delay was added to the model to ensure the correct byte was received. The only input for this model was the RX line; the outputs were the byte read in and the `Data_Valid` line was asserted for one clock cycle once the byte was received.

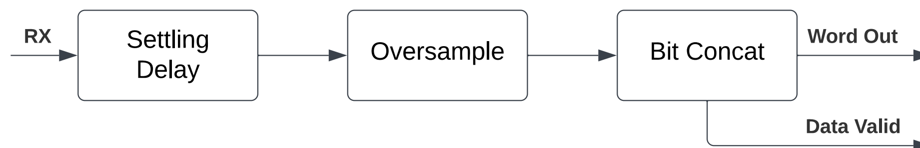


Fig. 4.7: UART Receive Model

Before adding the oversampling and settling time delay, byte errors occurred during benchtop testing. Before these two methods were implemented, there was a 41.9% chance

a 21-byte packet would be received incorrectly by the UART Receive module. Once the settling delay and oversampling were added, no byte errors occurred during a test of 670,865 packets using the same physical setup.

CHAPTER 5

Data Processing Chain

Several instruments are included as part of the SWP2 firmware. Each instrument has a specific data processing chain to convert the raw ADC samples into useful data on the ground. This chapter discusses the process of using the C&DH blocks and interface drivers discussed in Chapters 3 and 4 to build the data processing chain for the FPP. This chapter begins with Simulink and ends with data collected on the FPGA.

5.1 Theory of Operation

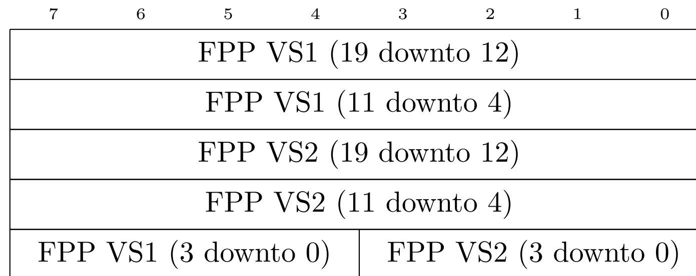
The FPP reads data from two probes through two separate ADCs. The ADCs are oversampled and averaged to reduce the noise of the measurement. These processed measurements are put into a granule, and these granules are combined into a telemetry packet. This packet is then sent over UART to the spacecraft, and eventually sent to the ground for processing.

MATLAB scripts were created to configure each Simulink block and verify parameter validity. As part of these scripts, the AD4003 documentation was converted to a struct so other scripts could reference these values (Listing B.2). Some errors were found in the documentation during testing, and the affected parameters were corrected in this struct. The operating conditions for the FPP SPI driver was configured from a MATLAB script (Listing B.3). This script included assertions to ensure conversion times and clock rates did not exceed the AD4003 specifications. Finally, a MATLAB script was created to configure the FPP data processing (Listing B.4).

5.2 Packet Structure

The FPP Packet contained 200 granules of floating potential data to determine the floating potential of the spacecraft. The data is sampled from the ADC at 10 kHz and is

averaged down to 100 Hz. Each granule consisted of 2 20-bit numbers packed into 5 bytes (Figure 5.1).



7	6	5	4	3	2	1	0
FPP VS1 (19 downto 12)							
FPP VS1 (11 downto 4)							
FPP VS2 (19 downto 12)							
FPP VS2 (11 downto 4)							
FPP VS1 (3 downto 0)				FPP VS2 (3 downto 0)			

Fig. 5.1: FPP Granule

5.3 Simulink Implementation

The full system for this data processing chain included the SPI drivers, the telecommand parser, the FPP processer and packetizer, the packet router, and the UART Send module (Figure 5.2). This system did not include a status packet. The SPI driver mask was configured using variables generated by the MATLAB scripts (Figure 5.3a). The packetizer block was also configured using variables from the MATLAB scripts (Figure 5.3b).

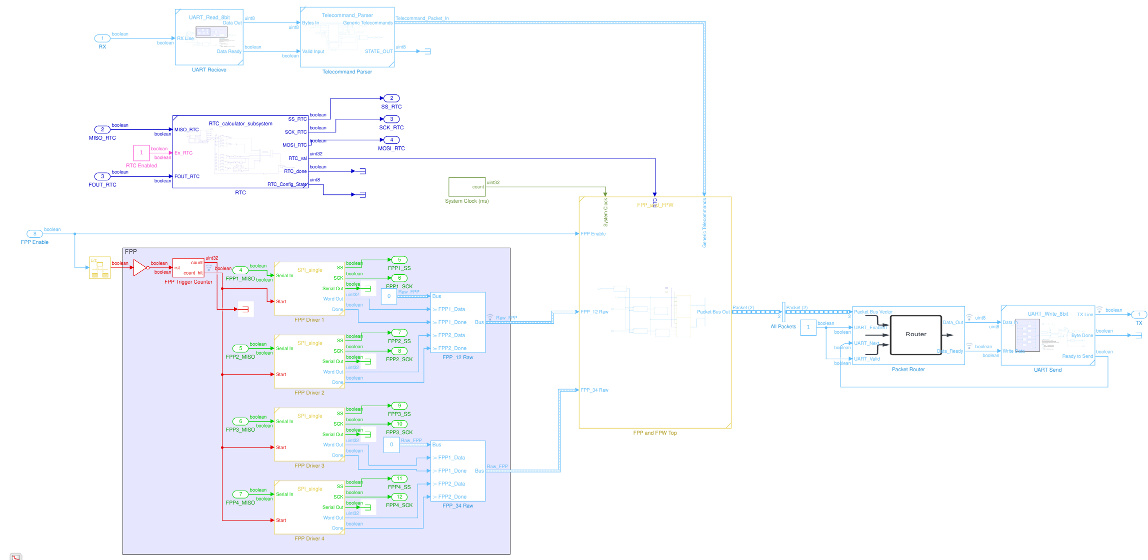
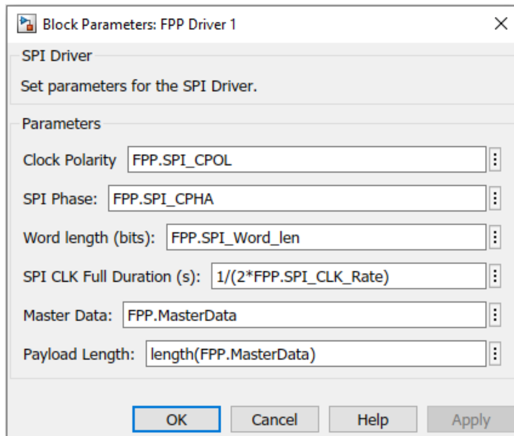
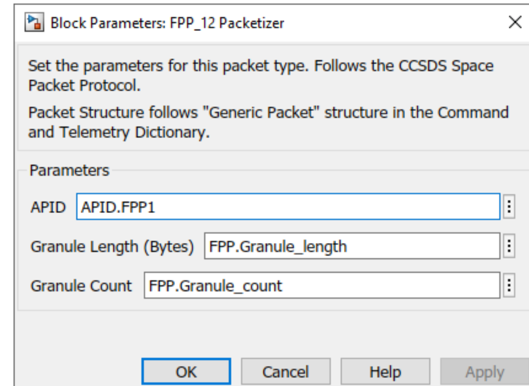


Fig. 5.2: FPP data processing chain



(a) SPI Driver mask



(b) Packetizer mask

Fig. 5.3: FPP mask configuration

5.3.1 FPP Processing

The FPP processing block summed a number of ADC samples together to downsample the signal (Figure 5.4). In this case, 100 samples were summed together to downsample 10 kHz down to 100 Hz. Once 100 samples had been taken, a digital gain was applied. By default, the FPP processing used the gain specified by the MATLAB script to shift the most significant bit of the summed value to match the MSB of the 20 bit granule value. The granule sliced out the top 20 bits, so the digital gain defaulted to 5 to ensure the sign bit was not accidentally overwritten.

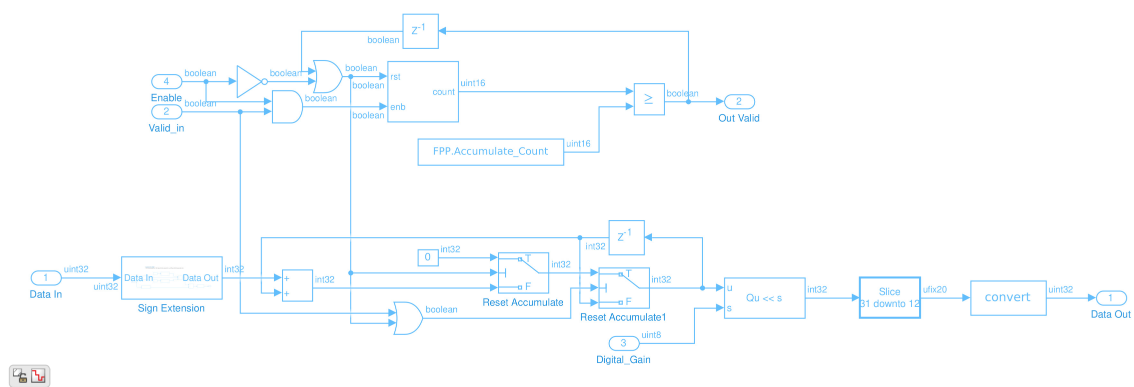


Fig. 5.4: FPP Processing

The digital gain of the FPP could be configured using a telecommand. The configuration block (Figure 5.5) was designed to allow for various parameters to be configured for different processing chains and more examples are given in Appendix C.

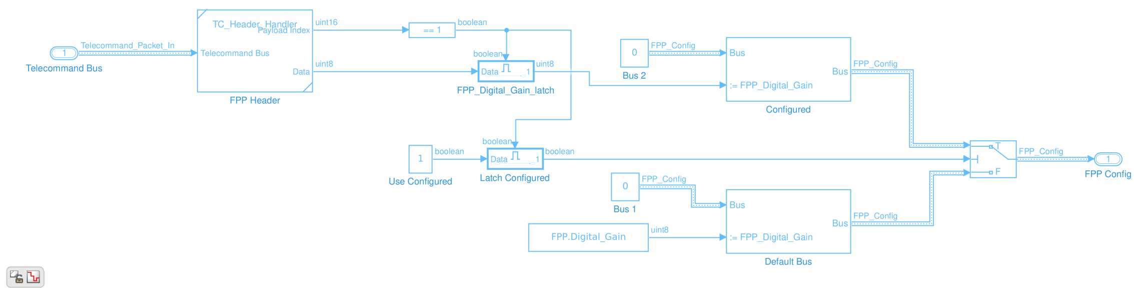


Fig. 5.5: FPP Configuration Block

5.3.2 FPP Granule

Once a granule was created by the FPP Processor, the FPP granule block serialized the 5 bytes (Figure 5.6). These bytes were fed into the packetizer. The granule block was designed to allow for changing the deserialization of any processing chain to be completed with minimal changes to the SWP2 architecture. To change the structure of a granule, the researcher only needs to add more ports to the multiport switch and connect the correct bytes to each byte from the granule. A state machine in the Command subsystem controls the output of this Referenced Subsystem.

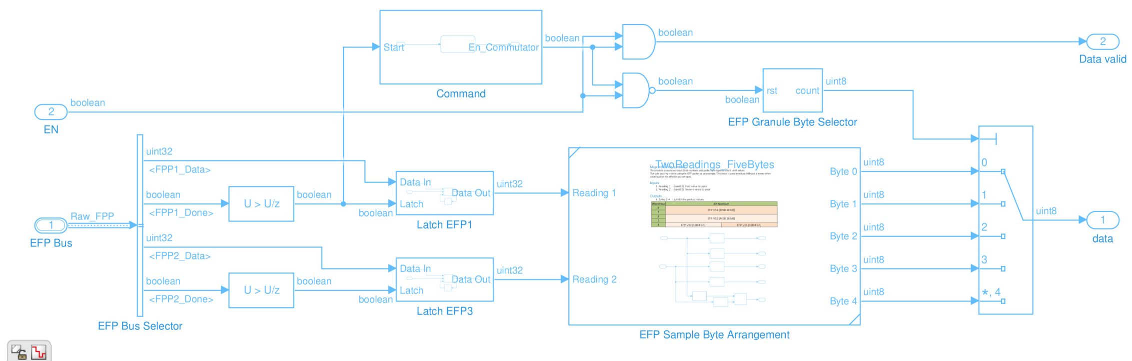


Fig. 5.6: FPP Granule Creation

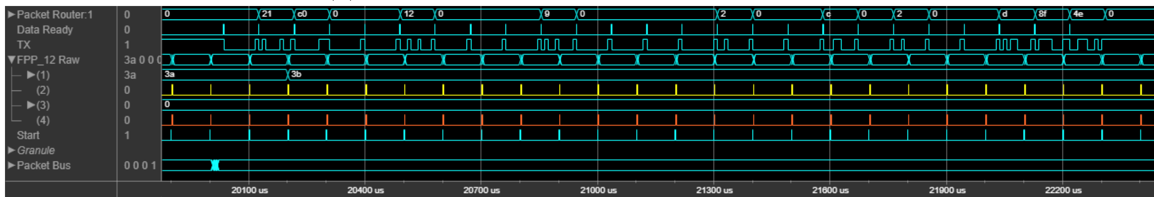
5.4 Simulation Results

This design was then simulated using the Logic Analyzer from the Simulink DSP Toolbox. For this simulation, the granule count of a packet was changed from 200 to 2 to reduce the simulation time from 2.005 seconds to 2.5 milliseconds. A SPI device was simulated to write a constant value for the first granule and then a different constant value for the second granule.

The expected data stream (Figure 5.7a) consists of 6 components: the packet header, the system clock time, the RTC time, two 5-byte granules, and the 2-byte checksum. The expected data stream was compared to the first line of the simulation output (Figure 5.7b). The bytes transmitted over the simulated TX line matched what was expected for this simulation.

```
0x00  0x21  0xc0  0x00  0x00  0x12
0x00  0x00  0x00  0x09
0x00  0x00  0x00
0x00  0x02  0x00  0x00  0x0c
0x00  0x02  0x00  0x00  0x0d
0x8f  0x4e
```

(a) Expected data stream, from left to right



(b) FPP Processing Chain simulation waveform

Fig. 5.7: FPP Processing Chain simulation

5.5 FPGA Synthesis

The design was then exported using HDL Coder and placed on the FPGA board. The test board only had one physical ADC connected at the time of the test, so only that specific

granule was considered for this test. The other ADC SPI lines were constrained to PMOD pins in Libero, but no physical device was connected to the FPGA. This was done to prevent the synthesis tools from optimizing away the extra SPI drivers and processing blocks so the resource utilization of the design could be analyzed. Libero provides a resource monitor, which was used to ensure the firmware was held in reset until power had stabilized and the FPGA had finished booting.

Because multiple clock rates were present, multicycle constraints were set in Libero (Appendix B.5). Additional information about multicycle constraints can be found in Section 6.3.3. Timing analysis was performed in Libero and the worst negative slack was found to be 0.143 ns.

Once the PolarFire had been programmed with the bitstream, data was collected by connecting an external voltage source to the input of the FPP probe. The voltage was initially set to 0.6 volts, then manually swept around within the 0.2 V to 1.9 V range by the researcher to illustrate various voltages being read by the probe. The voltage settled at 1.55 V at the end of this test.

FPP packets were correctly sent over the UART to the Pi acting as a spacecraft emulator. The Pi logged this data to a file so the packets could be unpacked and plotted on a computer using analysis scripts (Figure 5.8). The voltage could be found from the ADC count using Equation 5.1.

$$V = 9.744 \cdot 10^{-5} ADC + 0.015385 \tag{5.1}$$

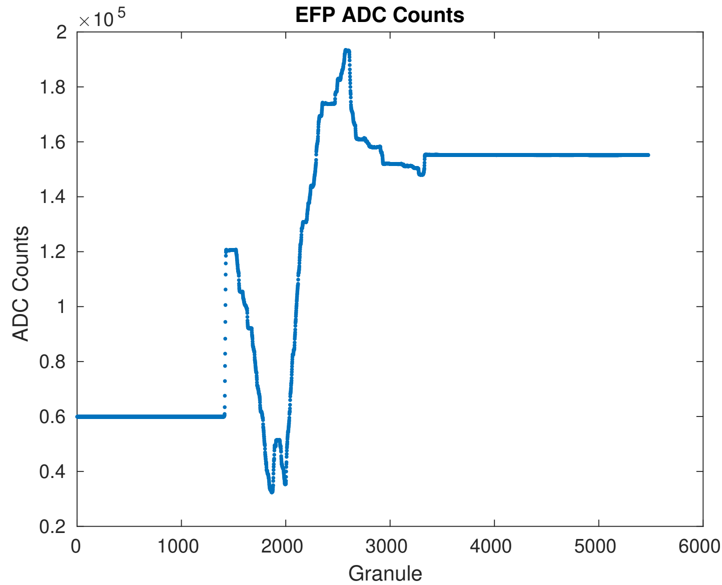


Fig. 5.8: FPP ADC counts from test

5.6 Resource Utilization

After synthesizing the design in Libero and verifying timing constraints were met, the resource utilization of each component was found (Table 5.1). The utilization is only shown for one instance of each module. For example, the utilization for the SPI driver is only for one of the SPI drivers instead of all 4 FPP drivers.

Component	4LUT Used	4LUT Percentage	DFF Used	DFF Percentage
SPI Driver	212	0.20%	98	0.09%
RTC Driver	1429	1.32%	733	0.67%
UART Receive	168	0.15%	86	0.08%
Telecommand Parser	3863	3.56%	2876	2.65%
FPP Configuration	18	0.02%	9	0.01%
FPP Processing	346	0.32%	69	0.06%
Granule Creation	118	0.11%	69	0.06%
Packetizer	413	0.38%	260	0.24%
Packet Router	2760	2.54%	2260	2.08%
UART Send	90	0.08%	55	0.05%

Table 5.1: FPP Resource Utilization by component

The full resource utilization of this design on the PolarFire FPGA is about 11% of the

4LUT and 7% of the DFF (Table 5.2). From the resource utilization and previous chapters, the framework components (SPI Drivers, UART, Telecommand Parser, and Packet Router) take up approximately 8.3% of the PolarFire, leaving the rest of the fabric available for data collection and processing of other packet types.

Type	Used	Total	Percentage
4LUT	11789	108900	10.86
DFF	7430	108900	6.84

Table 5.2: FPP Total Resource Utilization

CHAPTER 6

Results

This chapter discusses the results of building the SWP2 firmware in Simulink and how well each research objective was met.

6.1 Firmware development

This section addresses research objective 1.1.1, "Can the entire Space Weather Probes instrument firmware be developed in the MathWorks MATLAB/Simulink environment and deployed to an FPGA?"

Each individual subsystem of the SWP2 firmware was successfully created in MATLAB/Simulink and was able to be deployed to a PolarFire FPGA. This included the SPI drivers, the data processing chains, the command and data handling, and the UART interface. Multiple clock domains were also correctly created in Simulink and successfully deployed to the FPGA.

A specific interfacing task was unable to be completed using only Simulink and HDL Coder. Several op amps on the physical board had a shut-off pin to reduce power draw of the circuit when the instrument was not actively taking measurements. The shut-off pin is active low, and if not being driven to ground, the pin should be left floating for the internal pull-up resistor. Simulink can only output active high or active low signals, not a high impedance signal. To fix this, a small VHDL module was created to convert active low and active high to active low and high impedance, respectively (Appendix B.1).

6.2 Physical Performance

This section addresses research objective 1.1.2, "Does the developed FPGA system perform the same in physical hardware as in simulation?"

The developed SWP2 firmware performed the same in physical hardware as in simulation. Each clock domain interacted with the other clock domains correctly. Several different clock rates were used in the SWP2 firmware (Figure 6.1). The minimum clock rate of the FPGA was 160 MHz due to the impedance probe SPI driver. The FPP, EFW, and SLP did not require this faster data rate and were run at 16 MHz or 2 MHz to relax timing constraints. Some of the DSP for these probes was sufficiently complicated that the 160 MHz clock could not be used.

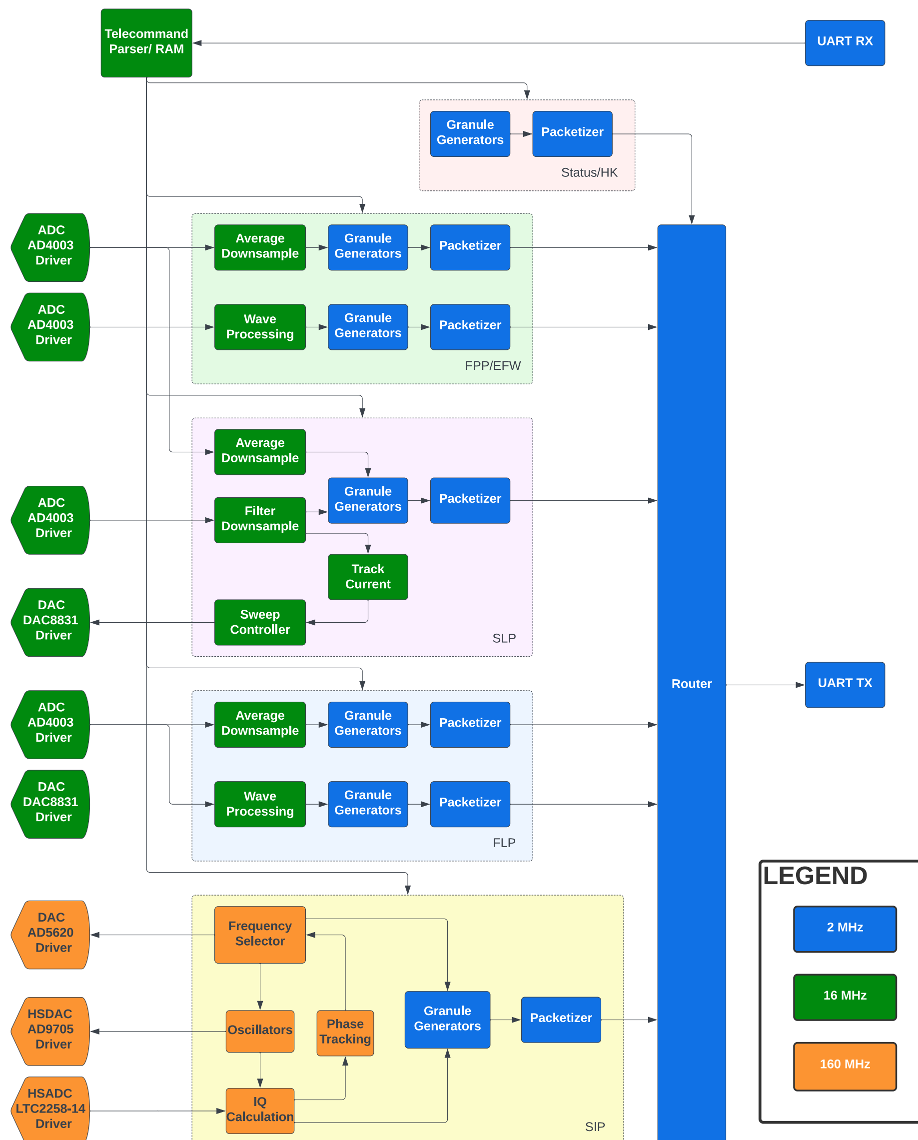


Fig. 6.1: Clock Domain Overview

HDL Coder used the same reference clock for each clock domain, and was unable to generate a multicycle constraints file for Libero. HDL Coder is able to generate a multicycle constraints file for Quartus, Vivado, and ISE. To ensure timing constraints were met in Libero, the researcher manually created a multicycle constraints file, discussed more in Section 6.3.3.

6.3 Issues with HDL Coder

This section addresses research objective 1.1.3, "What issues need to be overcome for using MATLAB/Simulink to develop a complex system for a PolarFire FPGA?"

Simulink and HDL Coder abstracted many aspects away from the designer so the researcher only needed to focus on functional correctness. Some of these abstractions lead to unexpected HDL generation and made optimizations difficult, but the researcher was able to overcome these issues. The following subsections discuss issues related to resource utilization, multicycle constraints, and timing encountered by the researcher.

6.3.1 Resource usage

Specific Simulink blocks caused excessive resource utilization when converted to HDL. One example of this is an early version of how data was downsampled for the IQ calculation (Figure 6.2). This used the built in Deserializer block to parallelize 128 samples, add the samples together, and then bit shift to remove the less significant bits. This solution worked great in simulation, but did not generate efficient HDL code. The Deserializer caused timing constraints to not be met, as shown by the blue clock domain, and caused two sets of 128 numbers to be added together instead of accumulating only two numbers in place (Figure 6.3).

In addition to a large number of bits being sent in parallel, 8 sequential additions needed to be completed to add all 128 numbers together before performing the bit shift. While the in place accumulation used more Simulink blocks, the resource utilization of the IQ calculation on the FPGA is much more efficient (Table 6.1) and only requires one addition to be completed before the bit shift at the end of the calculation. This model was

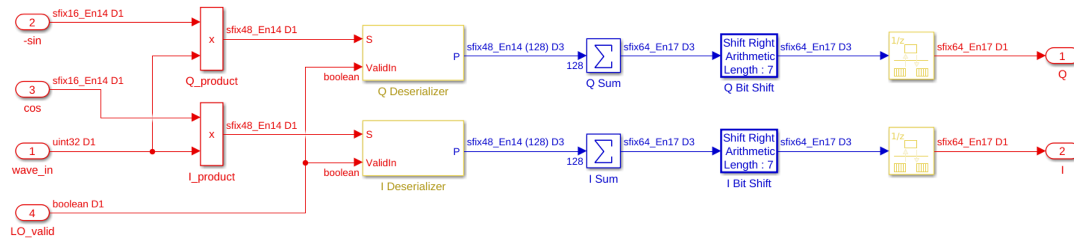


Fig. 6.2: Filter Downsample using Deserializer

stored as a Referenced Subsystem and reused in multiple places to avoid this unexpected issue.

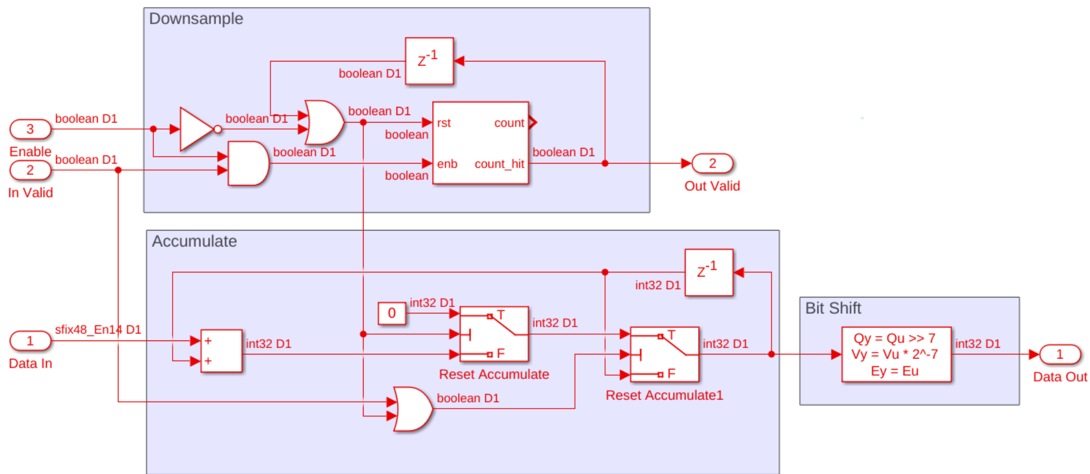


Fig. 6.3: Filter Downsample using in place accumulation

Method	4 LUT		DFF	
	Count	FPGA Percentage	Count	FPGA Percentage
Deserializer	21282	19.597 %	37058	34.123 %
In Place	292	0.2689 %	220	0.2026 %

Table 6.1: Filter Downsample resource utilization comparison

6.3.2 Clock domains

HDL Coder allowed the generated HDL model to be driven by a single clock or multiple

clocks. By default, HDL Coder used a single clock and created a clock control block to drive each of the clock domains within the generated HDL. Rate transition blocks were used to easily transfer data between clock domains, as seen in the Deserializer example in Section 6.3.1 to move the data back into the faster clock domain after the bitshift had been completed. This allowed Simulink to perform clock rate pipeline balancing to correctly adjust the model when certain DSP blocks took more than one clock cycle to complete.

HDL Coder was unable to perform clock rate pipeline balancing if the researcher wanted to use multiple external clocks. For SWP2, `atan2` was required for the phase angle computation for the impedance probe (IP) calculation, and this block required multiple clock cycles to compute the angle. Because of this, only a single input clock could be used for the SWP2 firmware. This limitation could be overcome by moving the IP subsystem to an external model, using a single clock, and interface the SIP model to the rest of the SWP2 model using multiple clocks. For this thesis, the researcher continued using a single model to simplify HDL importing to Libero, but further work would allow for better clock domain control in the generated HDL.

Because the slower clock domains were still driven by a faster clock, timing constraints were not met initially. Most of the SWP2 firmware did not require the 160 MHz clock the impedance probe required. The researcher added multicycle constraints to the Libero project to allow the slower clock domains to meet timing, discussed more in the next section. Once multicycle constraints were added to the Libero project, several sections were found to still not meet timing constraints.

The SPI driver state machines made use of the `after` keyword to control transitions between states based on FPGA clock cycles. For example, the `after(3, ‘tick’)` transition would hold the state machine in the same state for 3 FPGA clock cycles before transitioning. This transition caused the SPI driver blocks to have an fmax of 131 MHz. Even though the SPI drivers would not be run at 160 MHz for the SWP2 firmware, the generic blocks would be reused for other projects. The fmax of the SPI blocks was brought up to 180 MHz by adjusting the clock domain of the state machine instead of using the `after` keyword.

6.3.3 Multicycle Constraints

Multicycle constraints were used to improve the performance of slower sections of the SWP2 firmware by allowing critical paths to take multiple clock cycles to propagate from the source register to the destination register. Synthesis tools were unable to detect this on their own, so a constraints file needed to be created by the researcher. A multicycle constraint consisted of 4 parts: the source register, the destination register, the hold delay, and the setup delay. For the SWP2 firmware, the source register was the FPGA clock. A wildcard match was used for the destination register to reduce the number of constraints set. The setup delay was the integer multiple between the source clock and the destination clock rate. The hold delay was one less than the setup delay. For the Telecommand Parser, the destination register was `*/u_Telecommand_Parser*`, the setup delay was 10, and the hold delay was 9.

Multicycle constraints were not set for every block. Multicycle constraints were only used when the entire block used a single clock domain to simplify the wildcard matching. This reduced resource utilization by relaxing timing constraints for the synthesis and place and route tools. This also ensured the data processing blocks met timing constraints. For example, the FFT calculation in the EFW processing block had an fmax of 23 MHz, which could not meet timing for the 160 MHz FPGA clock.

Two main complications occurred during the implementation of multicycle constraints in Libero. The first was the researcher not recognizing the term multicycle constraint when searching through Libero documentation. Once the term was known, the researcher was able to find documentation and create the multicycle constraints file. The documentation for Libero also varies between Libero versions, so the researcher had to follow several documents to piece together how multicycle constraints worked in Libero 12 as opposed to Libero 11.

CHAPTER 7

Conclusion

The firmware architecture for SWP2 was successfully created in Simulink and exported using HDL Coder. This firmware was successfully tested to show the behavior on a PolarFire FPGA was consistent with simulated behavior. As part of this testing, timing constraints were met through the use of multicycle constraints. All of the signal processing was not finalized at the time this thesis research was completed, but mirrors the DSP of the first version of SWP. The generic blocks were created successfully for the SWP2 firmware architecture, and could be reused for other projects. Each instrument interface was successfully created, packets were generated, and the SWP2 firmware was correctly controlled using telecommand packets sent from a Raspberry Pi emulating a spacecraft.

7.1 Further Work

Further work for this project would include adding the ability to upload new firmware images to the SWP2 instrument on orbit. This would allow future researchers to adapt the signal processing on other missions in case the science instruments operate in an unexpected plasma environment. Power consumption analysis of each instrument and the FPGA would also be completed, as the analog board with all of the instruments was not completed during this thesis research.

Updates from Simulink 2022a to 2023a would be utilized, including signal conversion. Library blocks would be used instead of just using Reference Subsystem blocks once support for custom library blocks is improved for HDL Coder.

REFERENCES

[1] C. Secretariat, *Space Packet Protocol*, The Consultative Committee for Space Data Systems, Washington, DC, USA, 2020.

[2] N. Tipton, "Design of miniaturized sweeping langmuir probe and electric field probe for the sport mission," Master's thesis, Utah State University, Logan, UT, 2021. [Online]. Available: https://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=9214&context=etd

[3] M. Taufik, D. E. Amin, and M. A. Saifuddin, "Design and implementation of packet telecommand decoder based on ccsds using fpga," in *Proc. AIP/International Seminar on Aerospace Science and Technology*, 2021. [Online]. Available: https://aip.scitation.org/doi/abs/10.1063/5.0060011

[4] V. Y. Sarge, "Evaluating simulink hdl coder as a framework for flexible and modular hardware description," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2018. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/119717/1078637048-MIT.pdf?sequence=1

[5] K. Thesni, K. Praveen, and L. Srivani, "Implementation and performance comparison of digital filter in fpga," in *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020, pp. 589–594.

[6] Y. Hüner, M. G. Gayretli, and R. Yeniçeri, "Hw/sw design space exploration of a complementary filter on zynq soc," in *2021 8th International Conference on Electrical and Electronics Engineering (ICEEE)*, 2021, pp. 1–5.

[7] K. A. Jackson Pang and J. L. Torgerson, "Clarification for ccsds crc-16 computation algorithm," 2006. [Online]. Available: https://cwe.ccsds.org/sls/docs/SLS-CandS/Meeting%20Public%20Materials/2006/200606.Rome.Meetings/CRC%20Discussion/crc_clarify2.pdf

[8] *AB0815 Application Manual*. [Online]. Available: https://abracon.com/Support/AppsManuals/Precisiontiming/AB08XX-Application-Manual.pdf

APPENDICES

APPENDIX A

Simulink Models

This appendix contains Simulink models discussed in the thesis.

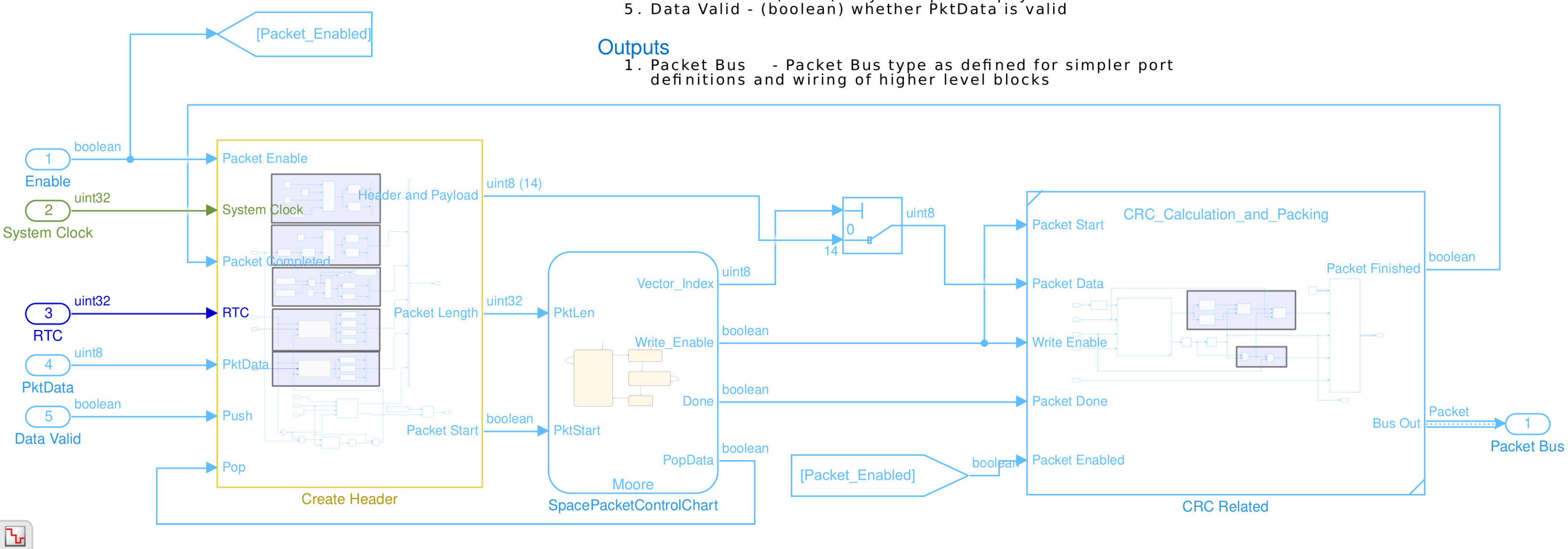


Fig. A.1: Packet Creation

Packet Bus Vector
Packet (2)
Packet FIFO Management
Data Out
RAM_Wr_Enable
Packet_Size
Header Done
uint8
boolean
uint16
Byte Indexer
enb
count
UART_Enabled
UART_Next
UART_Valid
RAM Indexer
RAM Wr Enable
Packet Start Index
Packet Length
Header Done
Transmission_Enabled
Send_Next_Byte
Valid
Address
Data_Ready
Packet RAM
wr_din
wr_addr
wr_en
rd_addr
rd_dout
Data_Out
Data_Ready

Main FIFO

This module handles a FIFO for each packet type to store all of the data, and resending packets when they are not recieved correctly by the satellite.

Inputs

1. Packet Bus In (x7)
   a. Data Byte - uint8 of data for packet
   b. Data Valid - data byte is able to be written to FIFO
   c. Packet Enabled - Packet type is enabled
1. UART Enabled - (bool) Begin transmission over UART
2. UART Next - (bool) Can send next byte, needed for clock division
3. UART Valid - (bool) Packet sent to satellite was well formed

Outputs

1. Data Out - (uint8) data from RAM to UART
2. Data Ready - (bool) RAM data out is ready

Fig. A.2: Packet Router

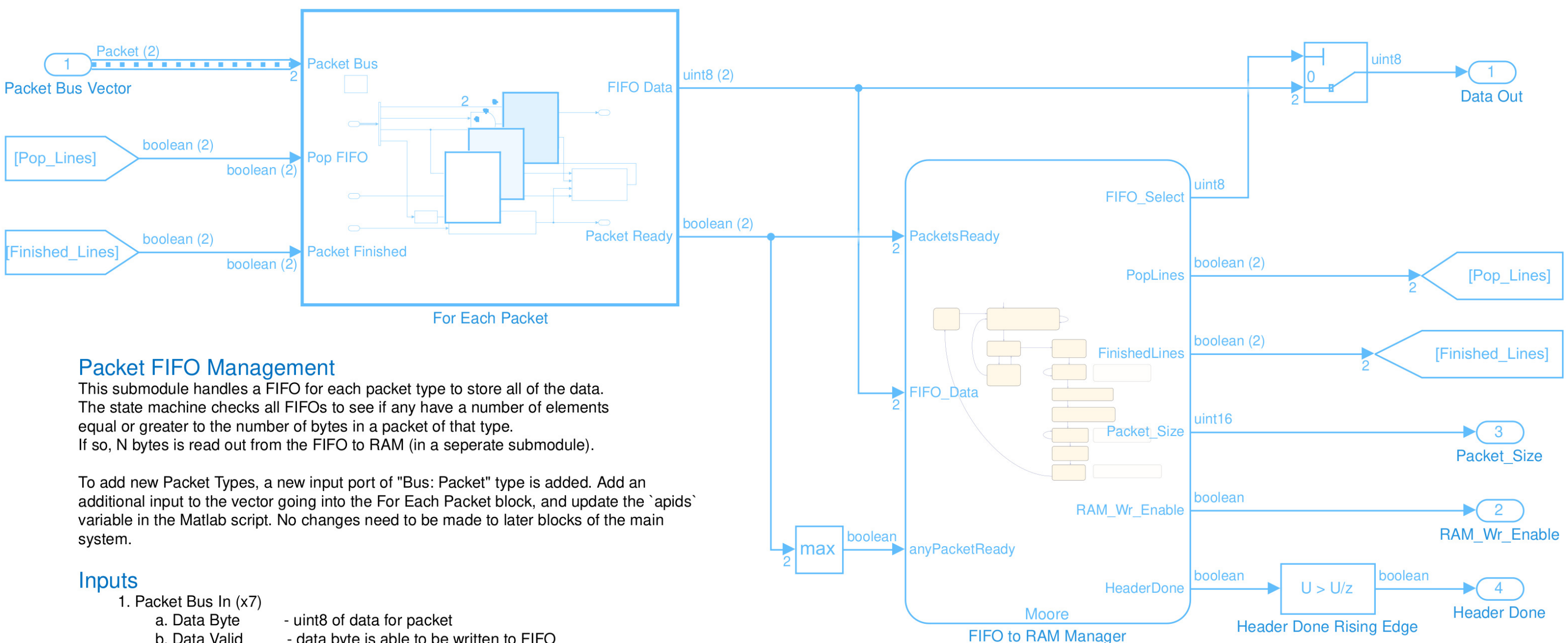


Fig. A.3: Router FIFO Management

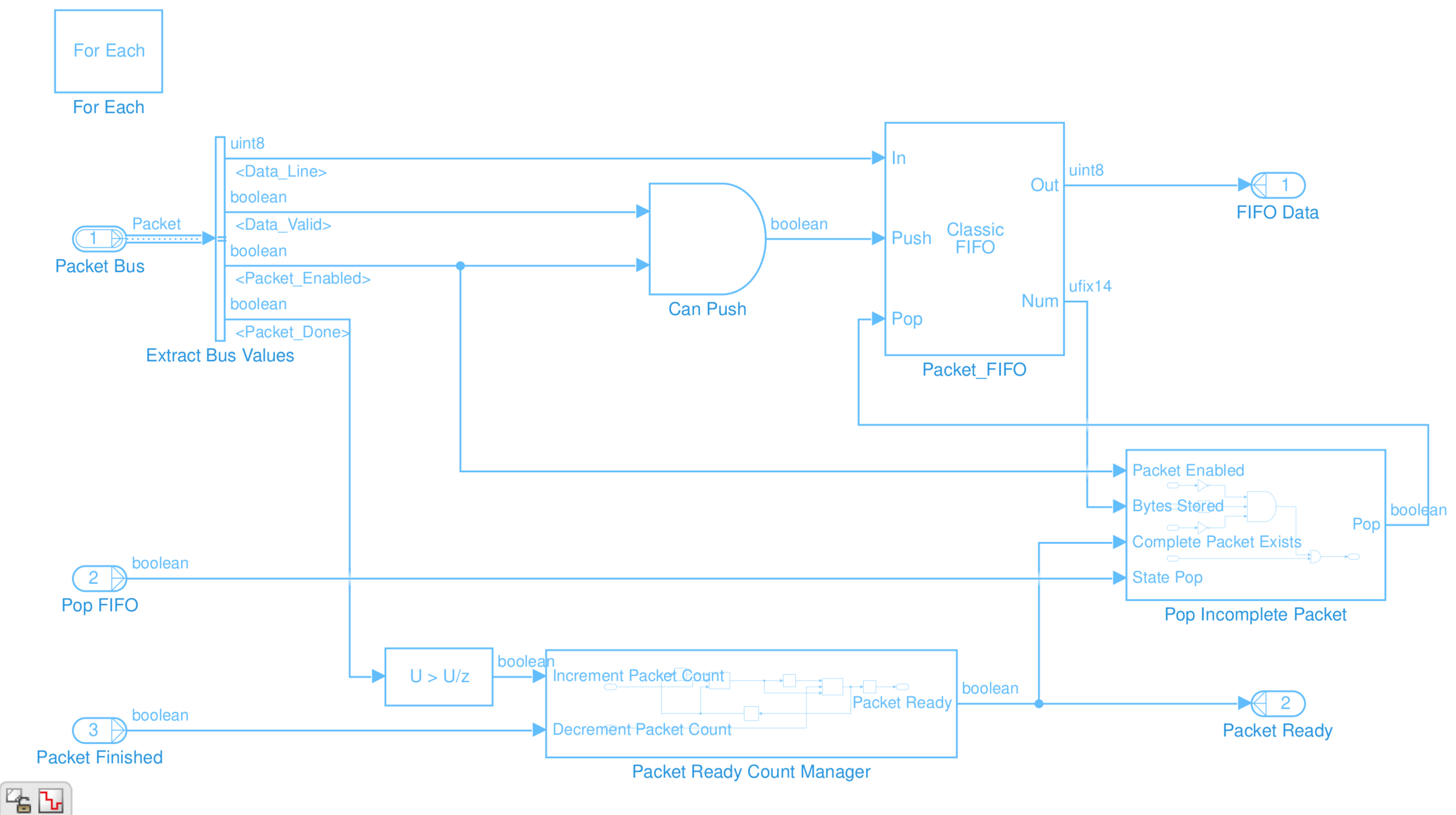


Fig. A.4: Router FIFO Management For Each Block

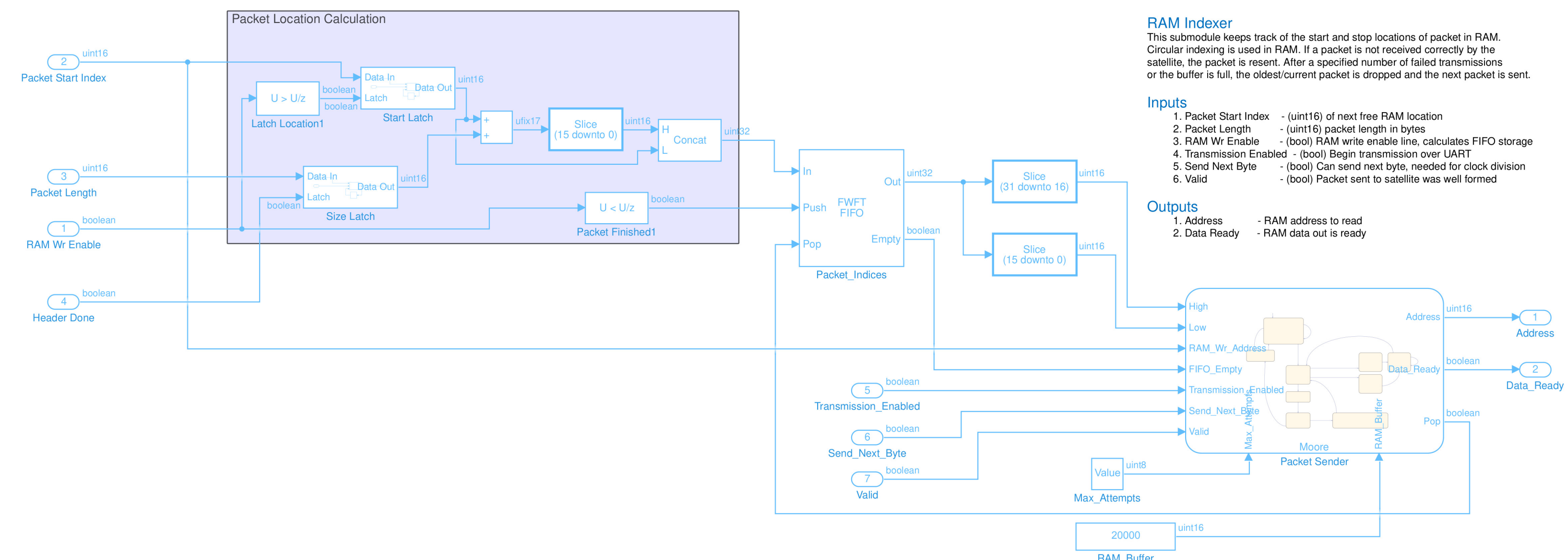


Fig. A.5: Router FIFO Management For Each Block

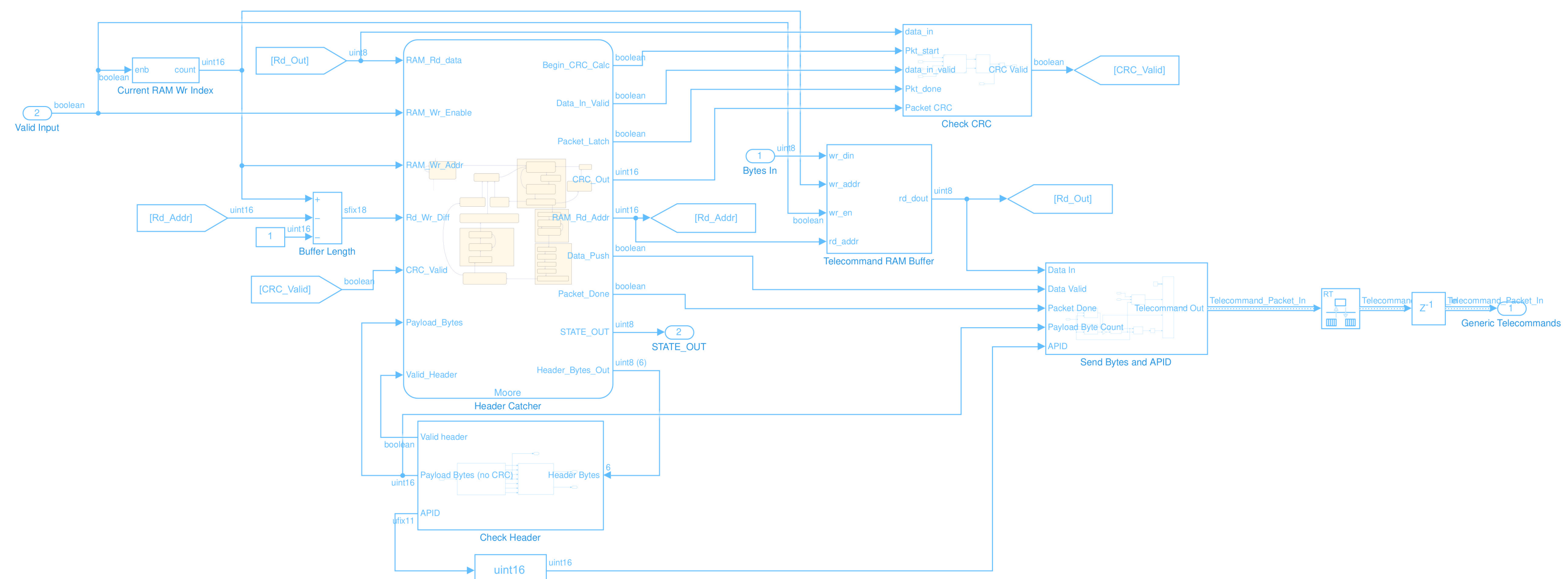


Fig. A.6: Telecommand Parser

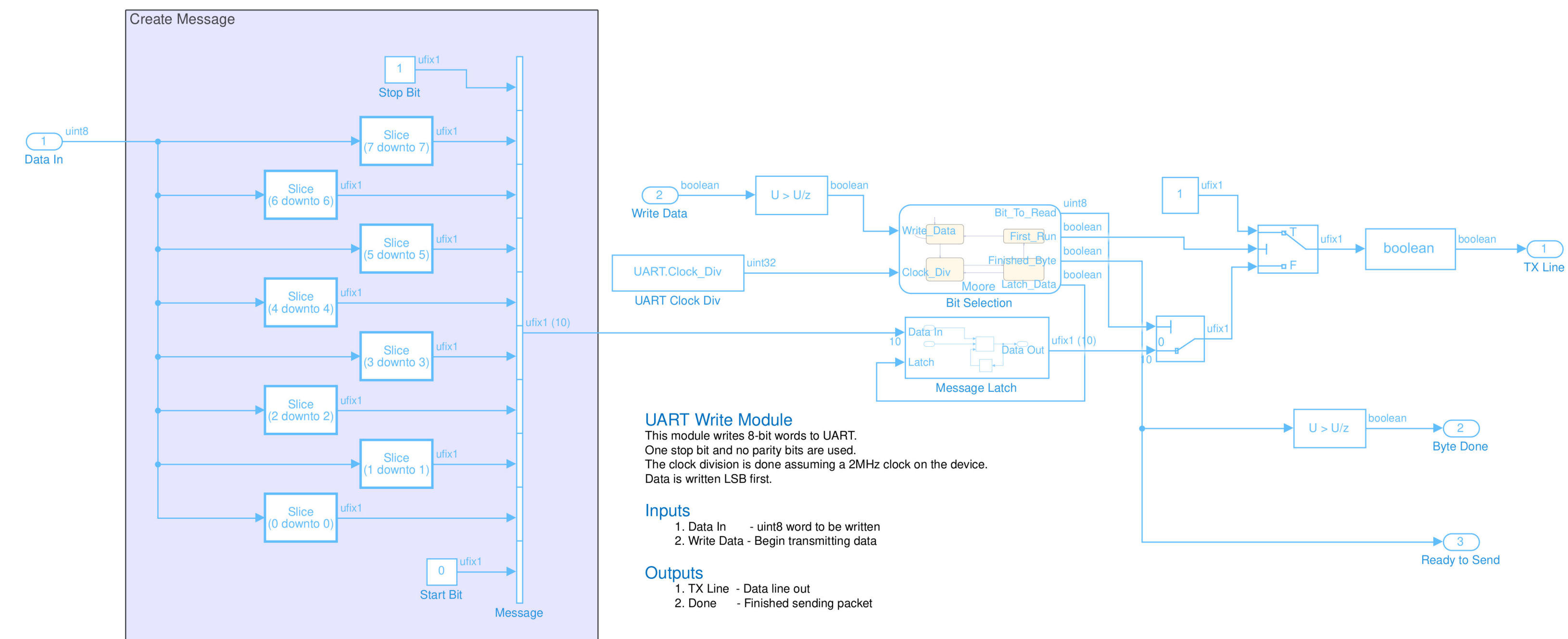


Fig. A.7: UART Send

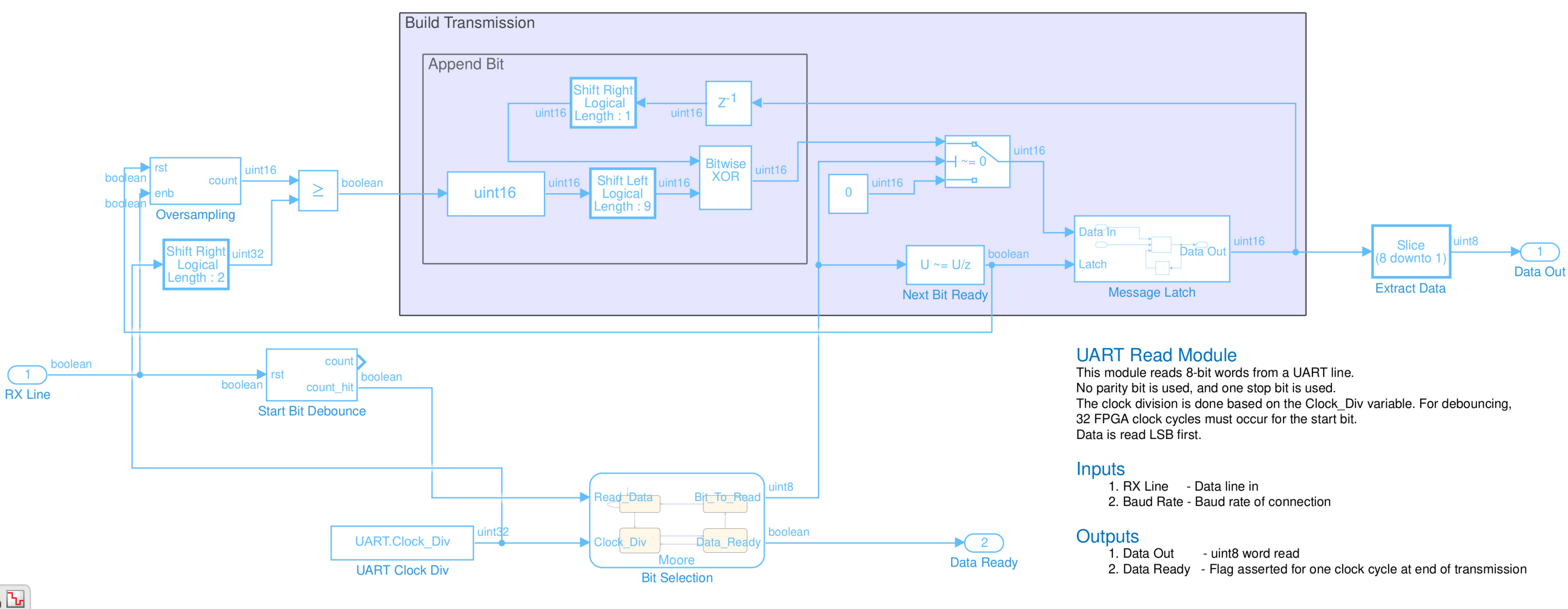


Fig. A.8: UART Receive

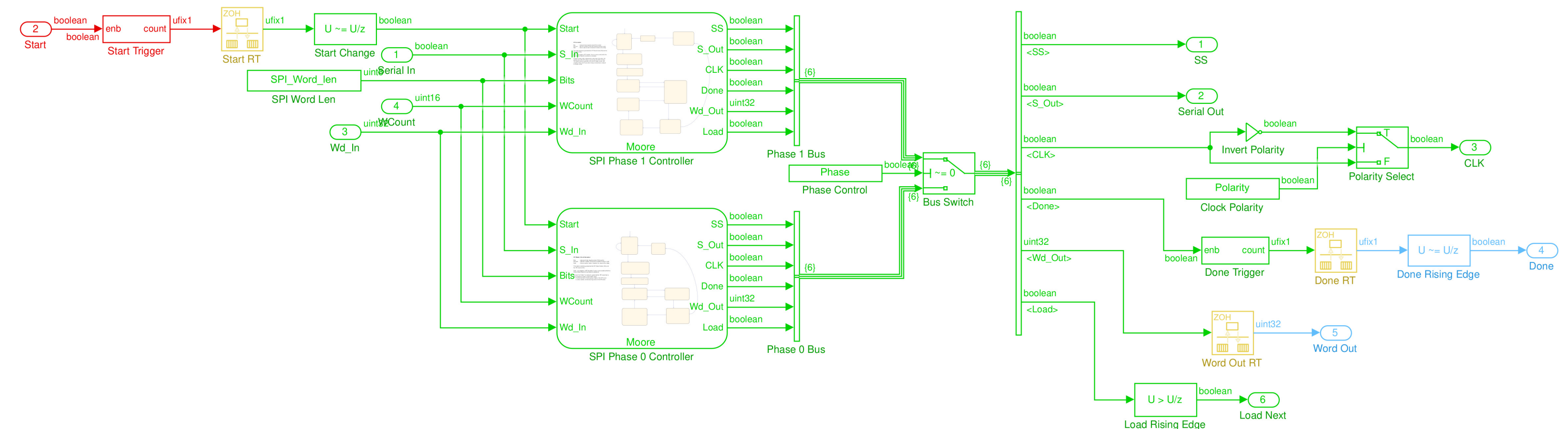


Fig. A.9: SPI Controller

APPENDIX B

VHDL Modules

Listing B.1: Simulink to high-impedance conversion

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


entity highz_convert is
port (
    pin_in  : IN  std_logic;
    pin_out : OUT std_logic
);
end highz_convert;
architecture architecture_highz_convert of highz_convert is
begin
    pin_out <= '0' WHEN pin_in = '0' ELSE 'Z';
end architecture_highz_convert;
```

Listing B.2: AD4003 values from datasheet

```
%% ==================================================
% This file configures the AD4003 struct to verify
% all values are valid with the datasheet and to
% configure various parameters within the Simulink
% model for HDL generation.
%
% These parameters are for CS mode, 3-wire without
% busy indicator. Some errors in the datasheet
% were found during our testing of the physical
% chip and are corrected here.

%% AD 4003 parameters from data sheet (ADC)
AD4003.SPI_Wait_Time       = 13e-9;   % Minimum time required
                                      % between SCK falling edge
                                      % and transmission start
AD4003.SPI_CLK_low_time    = 3e-9;    % Minimum time for SCK to
                                      % be low
AD4003.SPI_CLK_high_time = 3e-9;      % Minimum time for SCK to
                                      % be high
AD4003.time_between_conversions = 500e-9;
AD4003.SCK_period          = 12.3e-9; % Minimum period of SCK
AD4003.Conversion_time     = 290e-9;  % Time to complete conversion
```

Listing B.3: FPP SPI Configuration

```
%% ================================================
% This file configures the SPI variables for the
% FPP struct and verifies all values are valid
% with the datasheet for HDL generation.
%
% Sections are as follows:
%    SPI Configuration
%    Assertions to ensure correctness when synthesized

%% FPP SPI parameters
FPP.SPI_CLK_Rate  = 16e6; % Hz   : The SPI clock frequency
FPP.SPI_Word_Size = 32;   % bits : The size of the fi
                          %        structure used to contain
                          %        the SPI word (32-bits)
FPP.SPI_Word_len  = 18;   % bits : actual size of SPI word
FPP.SPI_Samp = 10e3;      % Hz   : The rate of sending words
                          %        across the system

FPP.clock_scale = FPGA_CLK_RATE / FPP.SPI_CLK_Rate;

FPP.SPI_Transfer_WDs   = 1; %    : Consecutive words transferred

% SPI definitions
FPP.SPI_CPOL    = false; % The Clock Polarity 0=false, 1=true
FPP.SPI_CPHA    = true;  % The Clock Phase 0=false, 1=true

% FPGA constants below here
FPP.SPI_trigger_ticks = uint32(floor( ...
             FPGA_CLK_RATE*FPP.SPI_Transfer_WDs/FPP.SPI_Samp));

FPP.SPI_delay = 1 + FPP.SPI_Wait_ticks + ...
```

```
                (FPP.SPI_CLK_half_ticks * 2)*FPP.SPI_Word_len;

FPP.MasterData = uint32([0,0]);

%% Asserts to make sure parameters are valid

% Check if wait after SS fall is long enough
assert(FPP.SPI_Wait_ticks * (1/FPGA_CLK_RATE) > AD4003.SPI_Wait_Time, ...
    ['FPP SPI Wait ticks lasts for too little time! Current time is ' ...
    num2str(FPP.SPI_Wait_ticks * (1/FPGA_CLK_RATE))])

% Check if SPI CLK is slow enough
assert(FPP.SPI_CLK_half_ticks*(1/FPGA_CLK_RATE) > AD4003.SPI_CLK_low_time, ...
    ['FPP SPI low period lasts for too little time! Current time is ' ...
    num2str(FPP.SPI_CLK_half_ticks * (1/FPGA_CLK_RATE))])

% Check that FPGA clock cycles required is less than FPGA speed1
clks_per_sample = FPP.SPI_Word_len*(FPP.SPI_CLK_half_ticks*2) + ...
                    FPP.SPI_Wait_ticks;
FPGA_clk_ticks_per_second = clks_per_sample * FPP.SPI_Samp;
assert(FPGA_clk_ticks_per_second < FPGA_CLK_RATE, ...
            ['FPP Sample rate too high! Max rate is ' ...
             num2str(FPP.SPI_CLK_Rate / clks_per_sample)])

% Check clock scale is a nonzero integer
assert(FPP.clock_scale == floor(FPP.clock_scale), ...
    "Clock scale is not an integer");

% Check that time between SS raise and next transaction is long enough
conversion_ticks = double(FPP.SPI_trigger_ticks - clks_per_sample);
assert(conversion_ticks*(1/FPGA_CLK_RATE) > AD4003.Conversion_time, ...
    ['FPP needs more time for conversion'])
```

Listing B.4: FPP Processing Configuration

```
%% ========================================================
% This file configures the granule and packet
% variables for the FPP struct and verifies all
% values are valid for HDL generation.
%
% Sections are as follows:
%    Granule/packet configuration
%    Processing Parameters



%% -- Granule Generator Setup --
FPP.Granule_generation_freq = 100; % Hz
FPP.Granule_count           = 200;
FPP.Granule_length          = 5;   % Bytes

%% -- Packet Configuration
APID.FPP1 = 0x21;
APID.FPP2 = 0x22;

%% -- Processing Parameters --

FPP.Accumulate_Count = FPP.SPI_Samp / FPP.Granule_generation_freq;
FPP.Digital_Gain     = floor(log2(FPP.Accumulate_Count));

FPP.Accumulate_Count = uint16(FPP.Accumulate_Count);
FPP.Digital_Gain     = uint8(FPP.Digital_Gain);
```

Listing B.5: FPP Processing Multicycle Constraints

```
set_multicycle_path -setup 40 -from [ get_clocks { PF_OSC_C0_0/
    PF_OSC_C0_0/I_OSC_160/CLK } ] -through [ get_nets { */u_RTC/* } ]
set_multicycle_path -hold 39 -from [ get_clocks { PF_OSC_C0_0/
    PF_OSC_C0_0/I_OSC_160/CLK } ] -through [ get_nets { */u_RTC/* } ]
set_multicycle_path -setup 40 -from [ get_clocks { PF_OSC_C0_0/
    PF_OSC_C0_0/I_OSC_160/CLK } ] -through [ get_nets { */
    u_Telecommand_Parser/* } ]
set_multicycle_path -hold 39 -from [ get_clocks { PF_OSC_C0_0/
    PF_OSC_C0_0/I_OSC_160/CLK } ] -through [ get_nets { */
    u_Telecommand_Parser/* } ]
set_multicycle_path -setup 40 -from [ get_clocks { PF_OSC_C0_0/
    PF_OSC_C0_0/I_OSC_160/CLK } ] -through [ get_nets { */
    u_Packet_Router/* } ]
set_multicycle_path -hold 39 -from [ get_clocks { PF_OSC_C0_0/
    PF_OSC_C0_0/I_OSC_160/CLK } ] -through [ get_nets { */
    u_Packet_Router/* } ]
set_multicycle_path -setup 40 -from [ get_clocks { PF_OSC_C0_0/
    PF_OSC_C0_0/I_OSC_160/CLK } ] -through [ get_nets { */u_FPP_and* } ]
set_multicycle_path -hold 39 -from [ get_clocks { PF_OSC_C0_0/
    PF_OSC_C0_0/I_OSC_160/CLK } ] -through [ get_nets { */u_FPP_and* } ]
set_multicycle_path -setup 80 -from [ get_clocks { PF_OSC_C0_0/
    PF_OSC_C0_0/I_OSC_160/CLK } ] -through [ get_nets { */u_UART* } ]
set_multicycle_path -hold 79 -from [ get_clocks { PF_OSC_C0_0/
    PF_OSC_C0_0/I_OSC_160/CLK } ] -through [ get_nets { */u_UART* } ]
```

APPENDIX C

Data Processing Chains

C.1 Command and Telemetry Dictionary

Each instrument and packet type has the parameters defined in the Command and Telemetry Dictionary, included with this thesis. The sheets within this document explain the granule composition, how many granules are in the packet, and at what rate the packet should be configured. This dictionary also explains what fields are configurable for each instrument through telecommands and how the bytes should be packed.