Integrated Autonomous Flight Termination Unit (IAFTU) Wrapper

Software Design Specification

October 2, 2023

Sagrad Document: SG906-0016

Revision: - 1.4

Status: Pending Approval

Sagrad, Inc.

Corporate Headquarters
USA Sales and Service

202 West Drive
Melbourne, Fl 32904
Telephone: (321).726-9400

Sagrad.com

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

TABLE OF CODE BLOCKS

1 COPYRIGHT, TRADEMARKS, DISTRIBUTION NOTICES

1.1 COPYRIGHT

1.2 TRADEMARKS

1.3 NOTICE OF CONTROLLED UNCLASSIFIED INFORMATION

2 DOCUMENT CONTROL

2.1 REVISION HISTORY

REV LETTER	REVISION	DATE	AUTHOR	DOCUMENT CHANGES
	1.0	05/26/2019	Kleber Valencia	Initial Draft Release
	1.1	6/24/20241	Kleber Valencia	Addressing 30th SLD comments. Added Partition Details
	1.2	06/24/20241	Kleber Valencia	Addressing IV&V and 30th SLD comments.
	1.3	12/09/2022	Kleber Valencia	Addressing IV&V comments. Updated Class Diagrams.
	1.4	10/02/2023	Stephen Mitchell	Layout, Format, Editing. Content Review and Correction.

Table 1: Revision History

2.2 APPROVALS

NAME	TITLE	ROLE	SIGNATURE	DATE
John Rizzo	Sr. Vice President, Business/Programs	Program Manager		6/24/2024
Kleber Valencia	Vice President, Engineering	Design Architect		6/24/2024

Table 2: Approvals

2.3 POINTS OF CONTACT

NAME	TITLE	EMAIL	CONTACT NUMBER
John Rizzo	Sr. Vice President, Business/Programs	jrizzo@sagrad.com	321-726-9400
Kleber Valencia	Vice President, Engineering	kvalencia@sagrad.com	
Tony Crosthwaite	Vice President, Operations	tcrosthwaite@sagrad.com	
Martha Harriman	President	mharriman@sagrad.com	

Table 3: Points of Contact

2.4 SALES and SERVICE DESK

In the event of a question about the operation of the systems, users should contact the Sagrad Service Desk at +1 (321) 726-9400, or via email: sales@sagrad.com

3 INTRODUCTION

Intergrated Autonomous Flight Termination System (IAFTS) is an independent, self-contained system that performs the flight **termination** function onboard a launch vehicle using a set of rule-based decisions and sensor information. The AFTS is designed to augment or replace the function of the traditional ground-commanded safety system, the ground tracking system, and the real-time computation and display system utilized by range operators.

Multiple independent sensors are used to monitor the real-time position, velocity, acceleration, and attitude with respect to flight termination criteria. The system can compare data gathered from the sensors to preloaded safety rules provided by the appropriate Range Safety offices. It can autonomously terminate a flight by providing the necessary inputs to the **destruct** or **termination** systems onboard the launch vehicle.

The system is designed to incorporate subsystems that make it more than single-fault tolerant. The **AFTS** consists of one or more redundant **Autonomous Flight Safety System** (AFSS) strings. An **AFSS** string consists of integrated hardware and software, including Ground Support Equipment (GSE) interfaces, sensors, an **Autonomous Flight Termination Unit** (AFTU), and the other Flight Termination System (FTS) equipment necessary to perform the safety function for a launch vehicle or vehicle-subsystem. Two AFSS strings may use the same sensors. AFTUs provide limited communication over a connection between **AFSS** strings (cross-strap) when flight enabled. The **AFTU** in each AFSS string contains a **Watchdog** that outputs a signal indicating when it is healthy and checks for the existence of a healthy **AFTU** in another **AFSS** string. Each **AFTU** communicates with GSE via a serial connection to a GSE command console prelaunch and via telemetry throughout launch activities.

The **Sagrad Wrapper** is a derivative software product developed by the National Aeronautics and Space Administration (NASA) Kennedy Space Center (KSC) and NASA Wallops Flight Facility (WFF), based on the February 9th, 2021 Engineering Technology Transfer, which is designed to work with Core Autonomous Safety Software (CASS) Operational Release 1.1 for rule processing and safety recommendations. The **Wrapper** takes those recommendations and asserts the hardware to perform safety actions. Those actions include **termination** and **performing mission-safe and fail-safe actions**.

Launch vehicle applications include:

- Primary Flight Termination System (FTS) for unmanned Range Safety Operations
- Primary FTS or Crew advisory system for human space flight

Several advantages of implementing an **AFTS** are:

- Cost reduction due to decreased need for ground-based assets
- Global coverage (vehicle does not have to be launched from a range)
- Increased launch responsiveness
- Safe Fly Zone boundary limits increase due to a 3-5 second gain from not having safety officers.
- Can support multiple vehicles simultaneously (such as flyback boosters)

3.1 PURPOSE AND SCOPE

The purpose of this **Software Design Document** (SDD) is to define the software design of the **AFTU Wrapper** for **CASS** when used with the already qualified hardware design. The **Wrapper** has been spilt into two partitions; one will support safety-critical tasks and another for **support support-critical** tasks. Partitioning is a safety feature implemented by Sagrad to ensure code executing in one partition does not adversely affect the other. The safety critical partition includes software that processes navigation content received from sensors, processes flight safety rules, makes flight safety decisions based on **CASS** recommendations, monitors health and status, and performs critical tasks such as flight termination. The **support critical portion** processes telemetry messages, prepare navigational messages to be ingested for **CASS** processing, and performs other activities discussed later in the document.

The scope of this **SDD** document covers the following:

- Design of hardware, firmware, software, and the relationship between them.
- Operational scenarios.
- To minimize duplications, anything relevant to the end-users is in the SG906-0013-AFTU User Manual. For example, telemetry format, interface information, configuration parameters, operating instructions, and more are in the SG906-0013-AFTU User Manual.

3.2 DOCUMENT ORGANIZATION

The design of the **AFTU** is described in this document and other documents as listed in the following table:

DOCUMENT	DOCUMENT NUMBER	DESCRIPTION
Software Design Description (SDD)	SG906-0016	This document describes the design of hardware, firmware, and software.
Software Requirements Specification (SRS) and Traceability	SG906-0017	This document describes the requirements and the traceability of those requirements to parent documents and tests.
Software Configuration Management Plan	SG922-0070	This document describes the software's configuration management, maintenance, updating, and backup.
Software Quality Assurance Standard	QOP 8.3.6	This document describes Sagrad's process of peer-reviewing engineering-developed software.
Software Static Analysis	SG926-0024	This document describes the code's software metrics, including information such as code complexity.
Software User Manual	SG926-0033	This document describes the functions and proper use of the AFTU to the end user.
Hardware User Manual	SG926-0032	This document describes the hardware and contains the ICD needed for vehicle integration and end-to-end testing.
Software Version Description (SVD)	SG926-0034	This document describes the version of the code we used, the Git hash, and build procedures.

Table 4: Document Organization

3.3 ACRONYMS AND ABBREVIATIONS

ACRONYM	DEFINITION
A	Amps
ADC	Analog-to-Digital Converter
AFSS	Autonomous Flight Safety System
AFTS	Autonomous Flight Termination System
AFTU	Autonomous Flight Termination Unit
BIT	Built-In Test
CASS	Core Autonomous Safety System
CBIT	Continuous Built-In Test
CPU	Central Processing Unit
DDR	Double Data Rate
DDR3	Double Data Rate 3
DOP	Dilution of Precision
ECC	Error Correction Code
FPGA	Field-Programmable Gate Array
FSBL	First Stage Boot Loader
FTIB	Flight Termination Interface Board
FTS	Flight Termination System
GoF	Gang of Four
GPIO	General Purpose Input/Output
GPS	Global Positioning System
GSE	Ground Support Equipment
IDE	Integrated Development Environment
IF	Interface
IPC	Interprocess Communication
LOD	Lift-off Discrete
MDL	Mission Data Load
MMU	Memory Management Unit
nTERM	Negative Term
OCM	On-Chip Memory

ACRONYM	DEFINITION
PDOP	Position Dilution of Precision
POST	Power-On Self-Test
RAM	Random Access Memory
RTP	Real-Time Process
SoC	System on Chip
SPI	Serial Peripheral Interface
TERM	Termination
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Data Protocol
UTC	Coordinated Universal Time
VCC	The voltage at the Common Collector
XADC	Xilinx Analog-to-Digital Converter

Table 5: Acronyms and Abbreviations

3.4 REFERENCE DOCUMENTS

3.4.1 AUTONOMOUS FLIGHT TERMINATION UNIT (AFTU) REFERENCES:

- National Aeronautics and Space Administration, NASA-AFTU-Users Manual 1.9, August 2020
- NASA AFTU Wrapper Code Doc, NASA-AFTU-CassWrapper-CodeDoc, Git commit ID c2a903212db665955fdf383ffcdc9ff8f04fa743

3.4.2 CORE AUTONOMOUS SAFETY SOFTWARE (CASS) REFERENCES:

- Core Autonomous Safety Software (CASS) Distribution, REV 7, Operational Release 1, 21 April 2015, Update 01 version, CASS_OR1_Update01_Release.7z
- Core Autonomous Safety System (CASS) Autonomous Flight Termination System (AFTS) Requirements for the CASS Flight Software, Release Date: 21 April 2015, CASS_FlightCode_SRS_2015April21.xlsx
- Core Autonomous Safety System (CASS) Autonomous Flight Termination System (AFTS) Requirements Draft Release: System and Wrapper tabs, Release Date: 08 April 2015, CASS_SRS_DraftRel2015April14.xlsx

3.4.3 XILINX REFERENCES:

- UG480, 7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide, v1.10.1, July 23, 2018, https://www.xilinx.com/support/documentation/user_guides/ug480_7Series_XADC.pdf
- UG585, Zynq-7000 SoC Technical Reference Manual, v1.12.2, July 1, 2018, https://sceweb.sce.uhcl.edu/xiaokun/Doc/Teaching/CENG4265/ug585-Zynq-7000-TRM.pdf
- UG782, Xilinx Software Development Kit Help, https://www.xilinx.com/htmldocs/xilinx2019_1/SDK_Doc/index.html

- UG821, Zynq-7000 All Programmable SoC Software Developers Guide, v12.0, September 30, 2015, https://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf
- Micron NAND Flash Memory Data Sheet (MT29F8G08ABABA, MT29F8G08ABCBB), https://www.micron.com/-/media/client/global/documents/products/data-sheet/nand-flash/60-series/m61a_8gb_asyncsync_nand.pdf?rev=8b4da60b2f5e472ebb07441a2823560c
- Xilinx U-Boot, v2016.4, https://github.com/Xilinx/u-boot-xlnx/tree/xilinx-v2016.4

3.4.4 VXWORKS REFERENCES:

- VxWorks Kernel Programmer's Guide, 6.9, https://node1.123dok.com/dt01pdf/123dok_us/004/492/4492204.pdf.pdf?X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=7PKKQ3DUV8RG19BL%2F20230828%2F%2Fs3%2Faws4_request&X-Amz-Date=20230828T163922Z&X-Amz-SignedHeaders=host&X-Amz-Expires=600&X-Amz-Signature=3d75617b45f606152a51d1437921d04ffcedf69f93f0c944009496bb7a28821d

3.4.5 MISCELLANEOUS REFERENCES:

- Javad GNSS Receiver External Interface Specification (GREIS), https://download.javad.com/manuals/GREIS/GREIS_Reference_Guide.pdf
- MISRA C++:2008 Guidelines for the use of the C++ language in critical systems, June 2008, https://electrovolt.ir/wp-content/uploads/2022/09/MISRA-C_2012_-Guidelines-for-the-Use-of-the-C-Language-in-Critical-Systems-Motor-Industry-Research-Association-2013-2013.pdf
- The standard C++ programming language specified by MISRA C++:2008 is defined by ISO/ IEC 14882:2003, C++2003, http://lmu.web.psi.ch/docu/manuals/software_manuals/ISO_Cpp/ISO_Cpp.pdf
- MISRA C:2012 Guidelines for the use of the C language in critical systems, March 2013, http://www.misra-c.com
- The standard C programming language specified by MISRA C:2012 is defined by ISO/IEC 9899:1999, C99, https://www.dii.uchile.cl/~daespino/files/Iso_C_1999_definition.pdf

3.4.6 REQUIREMENTS

- Flight Termination Systems Commonality Standard tailored for the NASA AFTU, RCC 319-14- T- NASA AFTU
- RCC 319-19-T
- Sagrad Wrapper SRS Rev 1.4
- CASS OR1.1 Wrapper Requirements

3.5 NEW FEATURES

The program will take all the positive features the previously developed baseline and build on to it. Key Sagrad customers were requesting features left out of the baseline. The Sagrad **Wrapper** will split the baseline code into **Safety Critical** and **Support Critical** partitions to support these customers and make the code adaptable for further requests. By performing this **split**, the program will limit the amount of future IV&V needed when adding additional features, namely support of other navigational sources.

Sagrad will implement compatibility for the **NovAtel OEM6/OEM7** series navigation sensor. This and support for the **Javad JNS100** navigation sensor will give customers the greatest flexibility in choosing appropriate sensors to support the safety function.

Another additional feature will be the inclusion of **100 Mbit/s Ethernet** support for the AFTU. This interface will support the ingestion of navigation sensor messages and a bi-directional telemetry interface for GSE.

The final difference is the Wrapper will support pre-defined system configurations to support navigation sensor inputs. The Wrapper will support three configurations: all serial navigation sensor interfaces for only **Javad** navigation sensors, all serial navigation sensor interfaces for only **NovAtel** navigation sensors, or all navigation sensor messages are sent over **Ethernet** and accepted through predefined ports of **8910** and **8911** for **Javad** and **3002** and **3003** for **NovAtel**.

In summary, the new features are:

1. **Partitioning** of the code for Critical and Support Critical
2. Adding support for **NovAtel OEM6/OEM7** navigation sensor
3. Enabling **100 Mbit/s Ethernet** for both telemetry activities and navigation sensor ingestion
4. Pre-defined system configurations for navigation sensor message ingestion

4 OVERVIEW

The AFTU is a single unit consisting of hardware (See: Section 5, Hardware), firmware (See: Section 6, FPGA), and software (See: *Section* 7, Software). Two AFTUs can be used together for redundant purposes. Figure 1: AFTU Chassis shows the AFTU chassis.

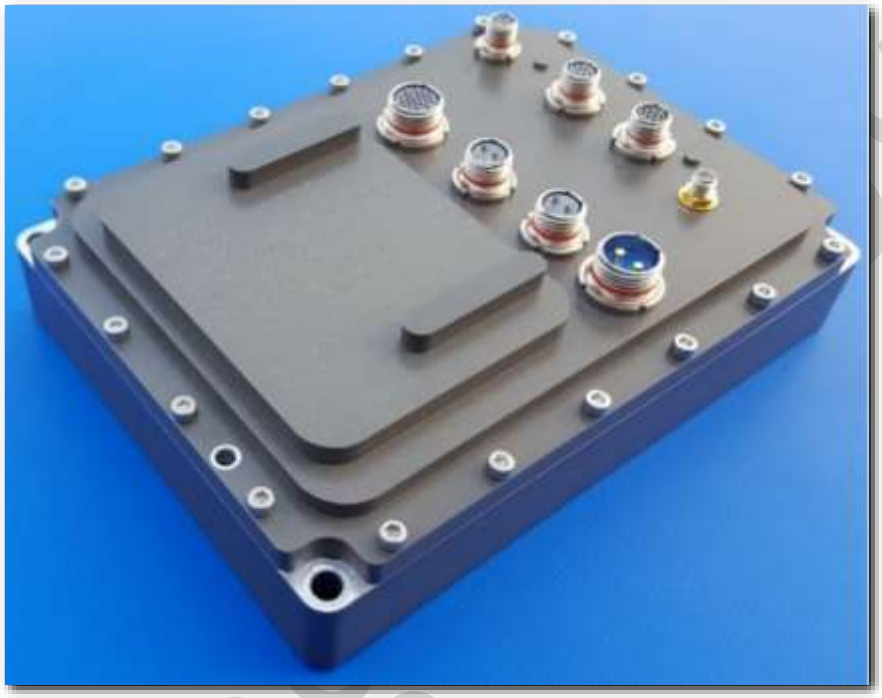


Figure 1: AFTU Chassis

The AFTU consists of a mainboard and an I/O board, as shown in Figure 2: Mainboard and I/O Board. The mainboard is based on a **Xilinx Zynq 7020 System on Chip** (SoC). The I/O board is a custom circuitry board for switching power to termination outputs. The firmware and software run on the mainboard.

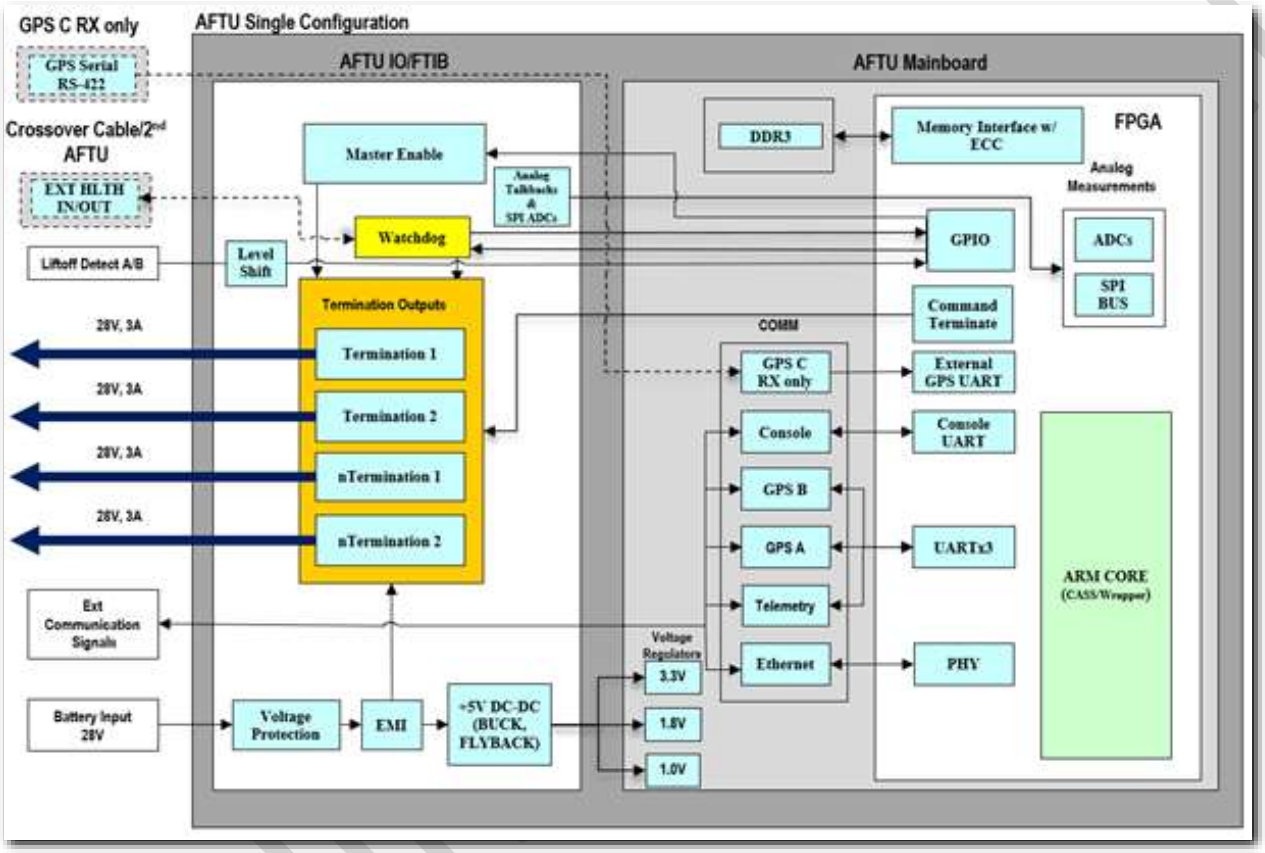


Figure 2: Mainboard and I/O Board

The external interfaces as shown in Figure 2: Mainboard and I/O Board are described in Table 6: Mainboard and I/O Board: External Interfaces below:

INTERFACE	DESCRIPTION
Power input	22 to 34 volts DC input to AFTU
Console	Transmission and receiver sides of the RS-422 line for console communication to GSE over umbilical. Console will be received only once the VxWorks application starts.
Ethernet	Supports up to 100 Megabit Ethernet, Used for vehicle and GSE communications. It also provides a two-way telemetry link to GSE to support user commands. Supports navigation sensor message ingestion.
Navigation	Transmission and receiver sides of the RS-422 lines to receive navigation data from navigation sensors.
Telemetry	Transmission and receiver sides of the RS-422 line to transmit AFTU telemetry data and receive user commands.
Liftoff A and B	0 to 34 volts and ground supplied by vehicle (Umbilical). Nominally 28V
TERM 1 and 2 outputs	Designed to deliver up to 3 amps continuously. Voltage drop for 1 amp is estimated to be 1 volt. Example use cases are for ordnance or normally open valves. Vin is AFTU supply voltage (nominally 28V), L is the length of cable in ft, A is desired current amps, and load is in ohms for the following formulas: Termination Voltage = Vin-((L/1000) + 0.64) * A) Load = 2(Vin-((L/1000) + 0.64) * A)/A)-0.385-(2(L(1.59/1000)))-L/1000 - 0.866 **NOTE** The word 'TERM' is an abbreviation for 'termination.'
nTERM 1 and 2 outputs	Designed to deliver up to 3amps continuous. Voltage drop is expected to be 1 volt. An example use case is for normally closed valves. Vin is AFTU supply voltage (nominally 28V), L is the length of cable in ft, A is desired current amps, and load is in ohms for the following formulas: Termination Voltage = Vin-((L/1000) + 0.64) * A) load = 2(Vin-((L/1000) + 0.64) * A)/A)-0.385-(2(L(4.35/1000)))-L/1000-0.866 **NOTE** The word 'nTERM' is an abbreviation for 'negative termination'
Health Monitor	Isolated 22 to 34 volts discrete indicates to the other chassis the AFTU health.

Table 6: Mainboard and I/O Board: External Interfaces

4.1 TWO AFTUS CONFIGURATION

An AFTS should consist of two AFTUs, each of which can independently activate the FTS to mitigate the following points of failure:

- Inability to terminate if commanded.
- Unrecoverable failure(s) in one **AFTU**.

Figure 3: Two AFTUs' Example Configuration, shows an example of how two AFTUs are setup for redundant purposes. In this configuration, both AFTUs are connected with each other through the Watchdog health monitor cable. This allows one AFTU to determine the health status of another AFTU. Each AFTU receives navigation data from two independent navigation sensors, and each sensor receives independent RF signals. Termination outputs from each AFTU go to an FTS.

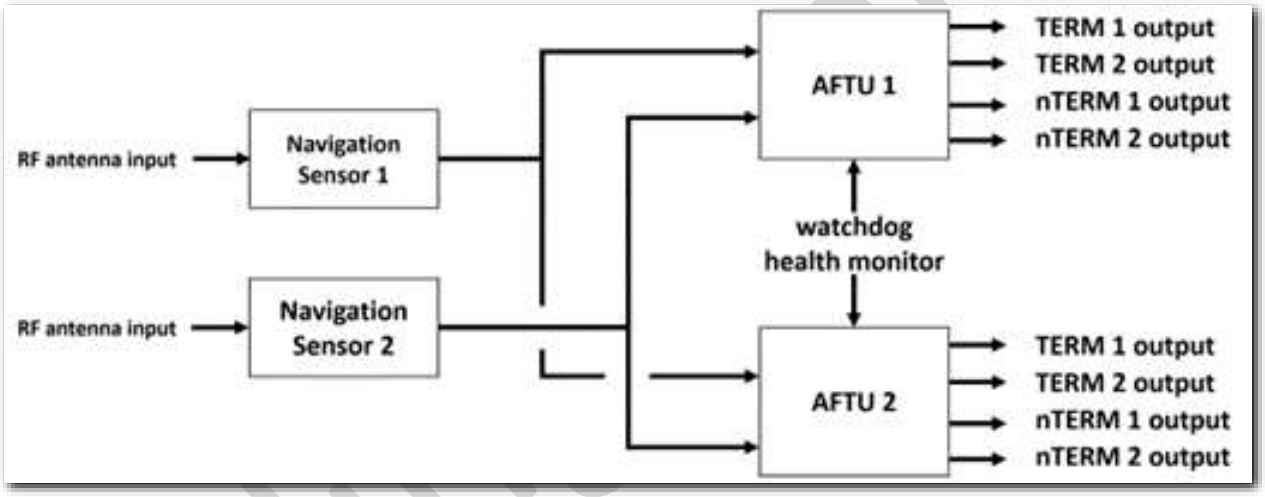


Figure 3: Two AFTUs' Example Configuration

The AFTU hardware supports redundancy against failure to terminate in the following ways:

1. Two independent AFTUs may be cross-strapped for health only; otherwise, they make independent safety decisions.
2. Each AFTU has two **Master Enable circuit input paths** with latches. Latching one latches both on, but unlatching one does not automatically unlatch the other. Should one latch break, the redundant latch may remain and can re-energize the first latch, protecting the unit's ability to terminate.
3. Each AFTU has two **Watchdog circuits** that both need to be unhealthy before a Watchdog can initiate a termination.
4. The Watchdog circuitry can initiate termination when the redundant AFTU is unhealthy and a critical failure has been detected on the remaining AFTU.

5. Each AFTU has two (**TERM1** and **TERM2**) positive termination outputs that operate redundantly. Both outputs go high when the AFTU commands terminate (nominally 28V).
6. Each **AFTU** has two (**nTERM1** and **nTERM2**) hold open outputs. Each **nTERM** output has one switch on the power side and one on the return side. If either switch is opened, power will not flow to the valves.
7. The **AFTU** supports multiple navigation sensor inputs, so a navigation sensor failure will still allow safety decisions. Should all sensors fail, **CASS** will default to applying minimum time to endanger rules (green time rules) from the **MDL**.

Table 7: Decision Logic in Single AFTU Configuration, and Table 8: Decision Logic in Two AFTU Configuration show the Watchdog decision logic in one and two AFTUs configuration, respectively. The definition for each AFTU state in both tables is as follows:

- **Healthy** – The AFTU is running with no detected critical failures, and its **Watchdog** circuit is not triggered to take an action.
- **Unhealthy** – The AFTU has detected a critical failure, and its **Watchdog** circuit is triggered to take an action. A critical failure results in the inability to reliably perform the safety function. Should the redundant AFTU also be detected as unhealthy, or a redundant AFTU not be detected, the local AFTU's Watchdog will invoke termination through direct control of the hardware.
- **Safe** – CASS recommends End-of-Mission based on mission rules configured in the **MDL**.
- **Terminate** – CASS recommends Terminate based on mission rules configured within the **MDL**.

AFTU	TERMINATE?
Unhealthy	Yes
Healthy	No
Safe	No
Terminate	Yes

Table 7: Decision Logic in Single AFTU Configuration

AFTU 1	AFTU 2	TERMINATE?
Unhealthy	Unhealthy	Yes
Unhealthy	Healthy	No
Unhealthy	Safe	No
Unhealthy	Terminate	Yes
Healthy	Unhealthy	No
Healthy	Healthy	No
Healthy	Safe	No
Healthy	Terminate	Yes
Safe	Unhealthy	No
Safe	Healthy	No
Safe	Safe	No
Safe	Terminate	Yes
Terminate	Unhealthy	Yes
Terminate	Healthy	Yes
Terminate	Safe	Yes
Terminate	Terminate	Yes

Table 8: Decision Logic in Two AFTU Configuration

4.2 CONCEPT OF OPERATIONS OVERVIEW

The operation for each AFTU, as shown in Figure 4: Operation Flow for Each AFTU, is broken into three groups:

1. The user updates a new software image in the AFTU. After powering on, the user would instruct the AFTU to boot into **U-Boot** by selecting maintenance mode over the **RS-422** console port. In **U-Boot**, the user would transfer a new binary image from the image repository to **DDR3** in the AFTU over either **Ethernet** or the **RS-422** console port. Then, the user would copy the binary image from **DDR3** to NAND. Lastly, the user would power cycle the AFTU to exit the **U-Boot**.
2. The user updates a new **MDL** in the AFTU. After powering on, the user would instruct the AFTU to boot into U-Boot by selecting maintenance mode over the **RS-422** console port. In U-Boot, the user would transfer a new **MDL** from the **MDL** repository to **DDR3** in the AFTU over either Ethernet or the RS-422 console port. Then, the user would copy the **MDL** from **DDR3** to **NAND**. Lastly, the user would power cycle the AFTU to exit the **U-Boot**.

3. The user verifies the software image and **MDL** in the AFTU. After powering on, the user would instruct the AFTU to boot into **U-Boot** by selecting maintenance mode over the **RS-422 console** port. In U-Boot, the user reads the **MDL**/**Software** image from **NAND** to **DDR3** in the AFTU. Then, the user would perform an **MD5sum** of the data. Lastly, the user would power cycle the AFTU to exit the **U-Boot**.
4. The user executes the **AFTU** software. The user would instruct or let the AFTU boot into **VxWorks** after powering on. In **VxWorks**, the Wrapper will be executed automatically, and at that time, the Wrapper will read **MDL** from the **NAND** from a specified memory location. Then, the user would send a command to enable the **CASS** rule processing, power up the termination circuit, and power up the Watchdog circuit over the **RS-422 telemetry** port. Unless the AFTU is power cycled, momentary removal of supply voltage, the Wrapper is kept running until either the end of the mission or a termination is executed.

Please refer to the SG906-0013-AFTU User Manual for port configuration, command interface, data transfer over **RS-422**/**Ethernet**, and **MDL** locations in **NAND** for detailed explanations.

The user would repeat any operation flow for each **AFTU** in the configuration of the two AFTUs.

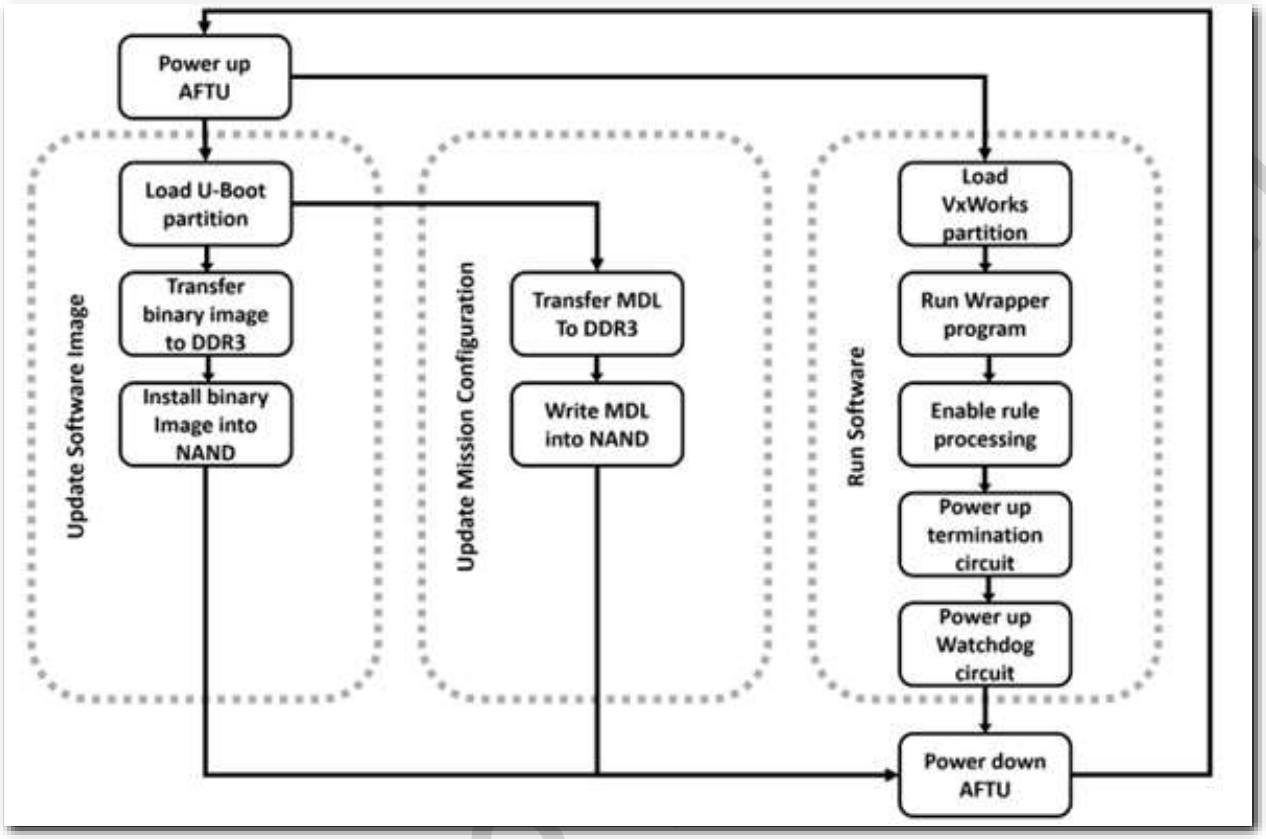


Figure 4: Operation Flow for Each AFTU

4.3 REUSE CONSTRAINTS

Some of the constraints that drove the **AFTU** hardware and software design were:

- Re-use already approved/IV&V **CASS Flight Software**
  - Use **CASS OR1** for the rule engine
  - Re-use **CASS** development process tailoring where applicable
  - Re-use/tailor **System** and **Wrapper** requirements from **CASS** development
    - Re-use/extend **CASS** error categorization
    - Re-use/extend **CASS OR1** input file (**MDL**) parser for configuration input
    - Re-use/extend **CASS_DDSim** error message formatting

> **NOTE:**
>
> DDSim itself was not part of IV&V but was part of CASS System Testing.

- Re-use **NASA ORS/TA3 Wrapper** design/code where compatible with requirements
  - Use of **COTS VxWorks** operating system to provide computer hardware abstraction
  - **AFTU** prototype telemetry format
- Re-use **NASA** & **COTS** computer implementations where applicable
  - Use **COTS Xilinx Zynq boot code and low-level drivers**
  - **Xilinx FPGA** libraries
- Since the **AFTU** does not contain **ECC** memory **RAM** parts, **ECC RAM** memory shall be implemented using the **FPGA** as described in the **Xilinx** article: Zynq-7000 AP SoC - 32 Bit DDR Access with ECC Tech Tip and example project:

  (https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842219/Zynq-7000+AP+SoC+-+32+Bit+DDR+Access+with+ECC+Tech+Tip)
- Navigation Sensors
  - JAVAD GREIS Integrated Messages-JAVAD chosen because of flight heritage.
  - NovAtel OEM6/7 - Chosen because of the end user's desire to use it.

5 HARDWARE

This section describes the characteristics of the **AFTU** hardware that Sagrad manufactures. Details about the hardware can be found in the **mechanical drawing** and **electrical schematic** documents. Figure 5: AFTU System Interface illustrates the AFTU chassis with cables.

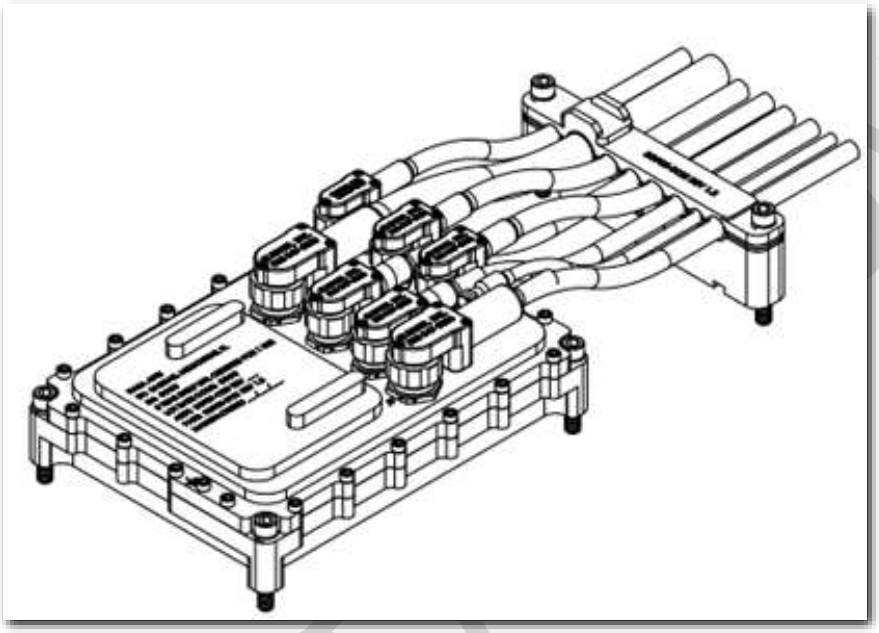


Figure 5: AFTU System Interface

5.1 TERMINATION AND WATCHDOG CIRCUITS

TERMINATION CIRCUITS – The power to support the **termination** circuit is controlled by the user from a **GSE** over **telemetry RS-422** managed by Wrapper software. The user must send commands as hex numbers (**0x22** and **0x33**) to enable or commands as hex numbers (**0x44** and **0x55**) to disable the termination circuit, See: SG901-0013-AFTU User Manual, section 7.0 Telemetry.

WATCHDOG CIRCUITS – The power to the Watchdog circuitry is controlled by the user from a **GSE** over **telemetry RS-422** managed by Wrapper software. The user must send commands as hex numbers (0x66 and 0x77) to enable the Watchdog circuit; see SG901-0013-AFTU User Manual, , section 7.0 Telemetry. After the Watchdog circuits are powered, they cannot be unpowered, except by power cycling the AFTU.

Figure 6: Electrical Logic Design on Generating TERM1 and nTERM1 Output describes the electrical logical design of generating **TERM1** and **nTERM1** outputs.

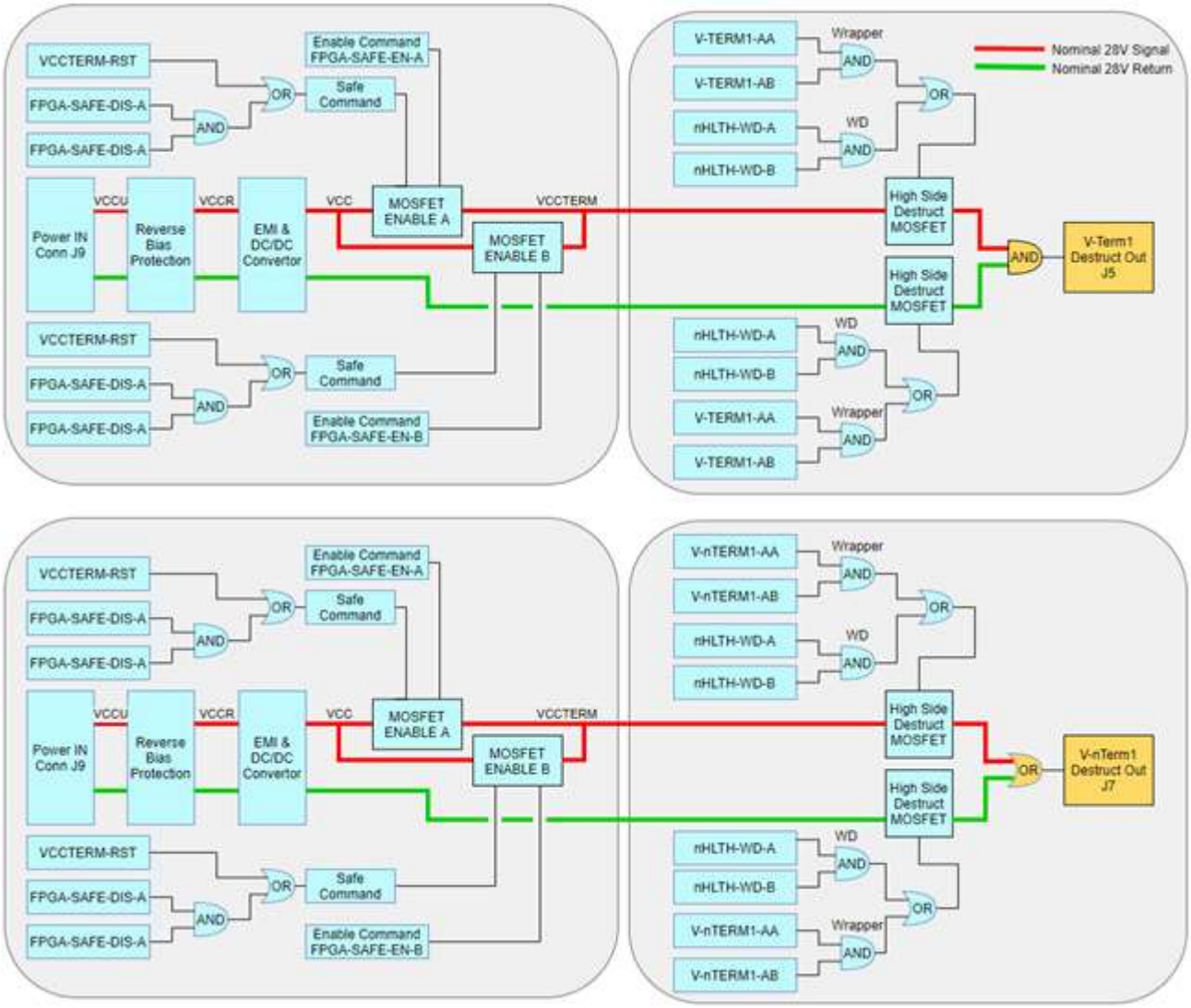


Figure 6: Electrical Logic Design on Generating TERM1 and nTERM1 Output

If the software fails to command a **Watchdog** reset in **300ms** or less, the Watchdog circuitry will perform the following sequence:

- Health status to other AFTU will not be sent (**bad health**)
- If the other AFTU reports healthily, this Watchdog will command **VCCTERM-RST** and turn off power to support termination circuits.
  - No current will flow to **nTERM** outputs regardless of **CASS**recommendations.

- No current will flow to **TERM** outputs, regardless of **CASS**recommendations.

- If there is no health signal from the other **AFTU** (or in a single **AFTU** configuration), this **Watchdog** will command terminate:
  - The **nTERM** source switches will be opened (nTERM 1/2 voltage/current stopped)
  - The **TERM** source switches will be closed (TERM1/2 voltage/current is sent but cannot be pulsed)
- Should the Wrapper's failure to reset the **Watchdog** circuit allow the **AFTU** software to continue running, it will:
  - Set health status to **bad** (if not already)
  - Continue ingesting sensors (if possible)
  - Continue calling **CASS** Update — having CASS cycles be performed and status sent to the wrapper.
  - NOT act on any **CASS** recommendations
  - Continue sending **Telemetry** data

5.2 HARDWARE REDUNDANCY

The AFTU hardware redundancy was designed to guarantee the termination of an unfit launch vehicle. The redundancy of two units mitigates the occurrence of a watchdog-commanded termination, also known as a **fail-safe termination**, and supports the primary mission of terminating an unsafe vehicle. The following section covers how redundancy was implemented in the hardware.

The Watchdog circuit consists of two resets that must be received within **~300** milliseconds before the failsafe action occurs. A reset is a way to tell the Watchdog that the AFTU is healthy. When an AFTU fails to reset the timer in the Watchdog within the timeframe, the Watchdog will think that the AFTU is unhealthy. Failsafe actions that can occur include removing supporting voltage for termination outputs or **fail-safe termination**. The redundant AFTU sends a health signal to the current AFTU as an input into its Watchdog circuitry for decision logic to either safe the unit or failsafe terminate.

Software-commanded termination happens when **CASS**recommends termination and then **Wrapper** instructs the hardware to do so. The instruction occurs by commanding four switches to the appropriate voltage levels to allow current to flow from the power side switch to the return side switch. The **AFTU** supports two forms of termination: **positive termination** or **TERM** and **negative termination** or **nTERM**. **TERM** requires both the power side switch and the return side switch to be on to perform termination; however, **nTERM** only requires the power side or the return side switch to be off to perform termination.

Each **AFTU** provides two **TERM** outputs and two **nTERM** outputs. Implementing **TERM** outputs provides straight forward redundancy by having two independent outputs, with separate cabling and driving circuitry, to guarantee a termination will occur. Redundancy with **nTERM** is less straightforward due to **nTERM** design having an **OR-driven** output; however, two outputs are provided for **nTERMS**. The previously mentioned outputs are on the same cable.

Other redundancies were implemented into the AFTU to support its safety mission. Two independent discrete inputs were included that are typically used as **liftoff detectors**. The AFTU named these discrete inputs because that is there intended purpose; however, a user can define the function of these inputs within the **MDL** to support their particular launch program.

- Critical variables are stored in memory equipped with an **error correction co**de (ECC) to detect and correct corrupt bits.
  - All AFTU Flight Software code and data reside in memory with ECC single-bit correction and multi-bit detection.
  - Since the AFTU does not contain **ECC** memory **RAM** parts, **ECC RAM** memory management shall be implemented using the **FPGA** as described in the **Xilinx** article: Zynq-7000 AP SoC - 32 Bit DDR Access with ECC Tech Tip and example project:

    (https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842219/Zynq-7000+AP+SoC+-+32+Bit+DDR+Access+with+ECC+Tech+Tip)
- Critical communications between software and hardware use unique patterns where corrupt bits cannot change state.
- **NAND** flash shall be used and verified through checksum.

6 FPGA DESIGN

This section describes the interfaces between the **FPGA**, the software, and the mainboard. More specifically, this section will explain how the software will interact with this **FPGA** design, the functions of the **FPGA** that will occur as a result of the software commands, and the outputs of the **FPGA** at the main board level. The reverse flow will also be described.

Sagrad shares the same **FPGA** design as was baselined in the February 9th, 2021 technology release provided by the WFF technology transfer office.

6.1 FPGA OVERVIEW

This **FPGA** design is safety-critical.

The **FPGA** design is primarily responsible for enabling flight software to control four major circuits at the board level: **master enable**, **watchdog**, **termination**, and **nterminaion**—two custom cores and one **Xilinx GPIO** core control these four circuits. The **Master** enable, and **Watchdog** enable signals are controlled through the **pattern_ip** core. The **Watchdog** is monitored and petted through the GPIO core. The **TERM** and **nTERM** circuits are controlled through the **pattern16_ip** core.

The **FPGA** also provides access to several key interfaces and peripherals to software: **DDR** memory, **NAND** flash, seven **UART**s, one **SPI**, one **Ethernet**, one **XADC**, and one **GPIO**.

Finally, the **FPGA** is responsible for **DDR ECC** (SECDED). **SECDED** is performed at the byte level. This is done via **ECC** proxy. Two counters are provided to count the number of **single-bit** errors corrected and double-bit errors detected with one caveat. The counters will always increment on error, but the number of errors corrected may be inaccurate. See: the ECC section 6.11 for more information.

This **FPGA** has a mix of **Xilinx** and custom cores. The cores were written in both VHDL and Verilog. The **FPGA** design targets the **Xilinx XC7Z020-1CLG484I**. The **FPGA** design was synthesized and implemented with **Vivado 2019.1**. The custom cores were simulated using **VUNIT** and **Modelsim**.

Due to the nature of the **Zynq FPGA**, the **FPGA** design naturally splits into two parts: The **Zynq Processing System** (PS) and the **Programmable Logic** (PL). Additional logic and features cannot be added to the PS. PS features can only be enabled, disabled, or configured. The **PS** contains the interfaces to **DDR** memory, **NAND** flash, the console **UART**, **SPI**, and **Ethernet**. The PL is the custom area of the **FPGA**.

The **PL** contains the additional **UARTS**, **ECC** counters, **Pattern_IP**, **Pattern16_IP**, **GPIO**, **XADC**, and routing logic. The **PL** also contains the **AXI Interconnect** and the **Processor System Reset** cores.

These are off-the-shelf cores released by **Xilinx**. The **AXI** Interconnect connects the **PS** to the other cores on the bus that it services. In this case, it connects all the **GPIO**, **XADC**, **UARTs**, and **Pattern** cores to the **Zynq** processor. The **Processor System Reset** is responsible for resetting all the **PL** cores.

Once configured, all functions of the **FPGA** may occur in parallel.

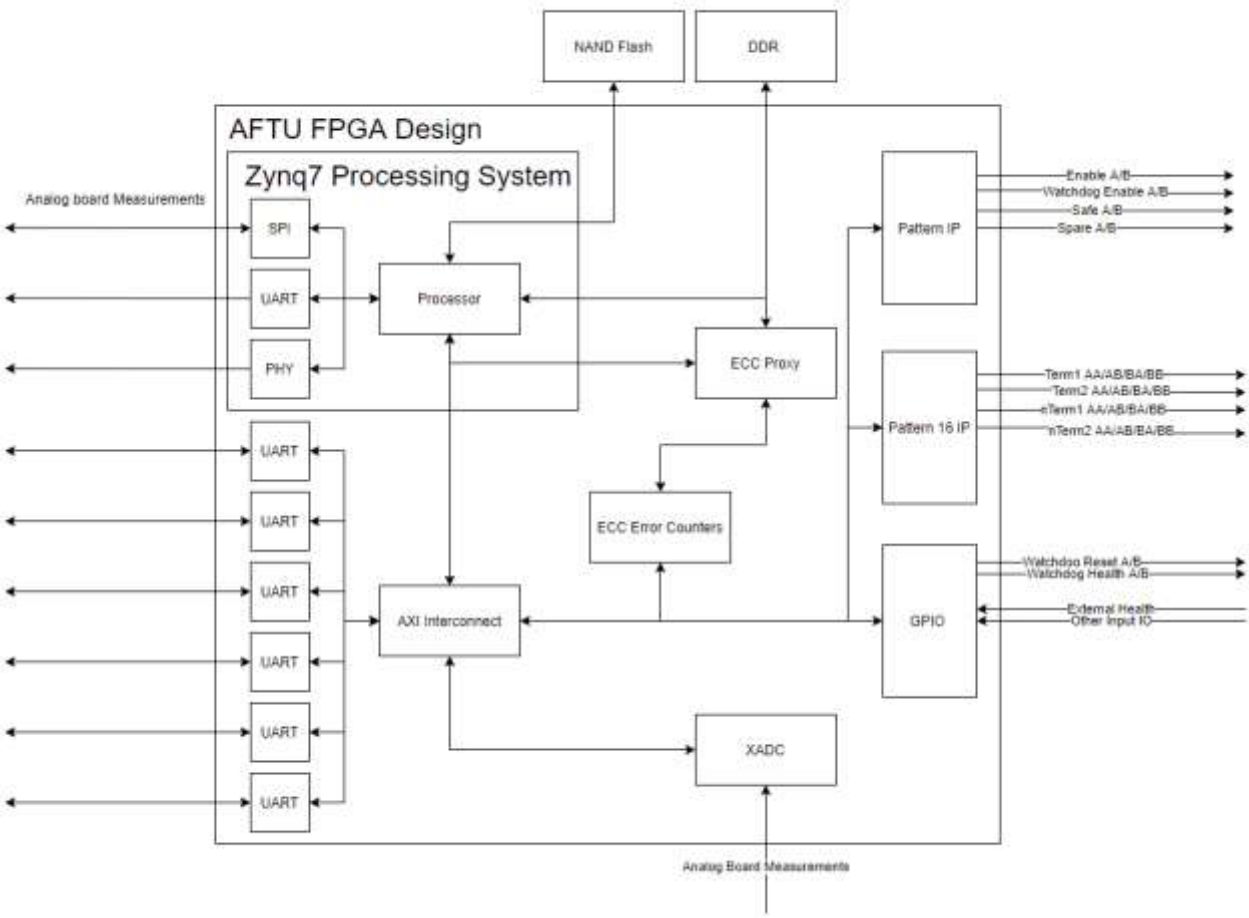


Figure 7: Top-Level FPGA Design

6.2 ZYNQ PROCESSING SYSTEM

The **AFTU** Flight Software uses the **Zynq** hardware depicted in Figure 8: Zynq XC7Z020 Processing System below. Used **I/O Peripherals** and **Flash Memory Interfaces** are checked.

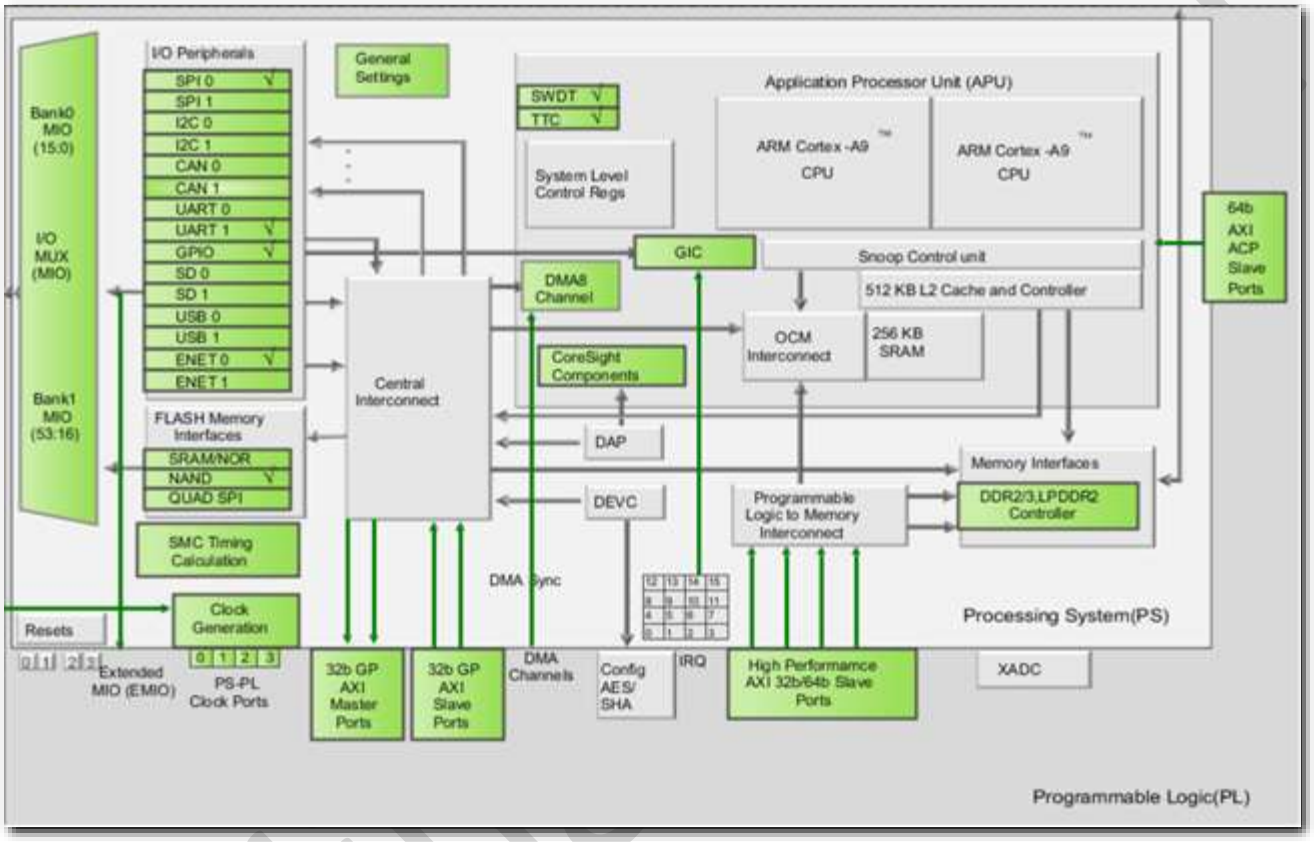


Figure 8: Zynq XC7Z020 Processing System

The **Zynq Processing System** contains the following interfaces:

MAJOR ZYNQ INTERFACES	LABEL	DESCRIPTION
UART 1	Zynq UART	16550 Compatible, Console/GSE Connection
ENET 0	Ethernet	Supports up to 1000 Mbps
SPI 0	SPI	Used to access on board ADCs
NAND	NAND Flash	Access to on board NAND Flash
DDR	DDR Memory	Access to on board DDR Memory

Table 9: Zynq System Major Interfaces

LINE	LENGTH IN NM
CLK0_P	57.3934
CLK0_N	42.1288
DQS0	23.4242
DQS1	21.1307
DQS2	19.35095
DQS3	13.9801
D0-D7	26.86541
D8-D15	25.05443
D16-D23	22.31624
D24-D31	18.24296

Table 10: DDR Training Lengths

6.3 MEMORY MAP

The following figures: Figure 10: Base Zynq Memory Map, Figure 11: VxWorks Memory Map Without ECC, and Figure 12: VxWorks Zynq Memory Map Using ECC are the memory maps of different systems. These are the default **Zynq** systems: **VxWorks** without **ECC** and **VxWorks Zynq** with **ECC**. The memory maps of these systems are better explained below.

Figure 9: Base Zynq Memory Map, and Figure 10: Base Zynq Memory Map These are the default memory maps for two scenarios of a base **Zynq** system. The first is the base **Zynq Memory** map with 512MB of **DDR3** (AFTU and Zedboard). The second is the base **Zynq Memory** map with **1024MB** (ZC702 and microZed) and an unmodified **VxWorks BSP** (Rev 13 and on) partition into the memory.

Figure 11: VxWorks Memory Map Without ECC. This is the **VxWorks** memory map with **AFTU** modifications to be able to allocate memory to the BSP:

- The first 250MB are restricted use for **VxWorks**' allocated code and data.
- All addresses above **0x10000000** are reserved for the AFTU configuration data loaded into memory. (Prior to adding the **NAND** driver, the **AFTU configuration data** was added to the boot image. This was done via a **bootgen** partition also loaded into this region.)

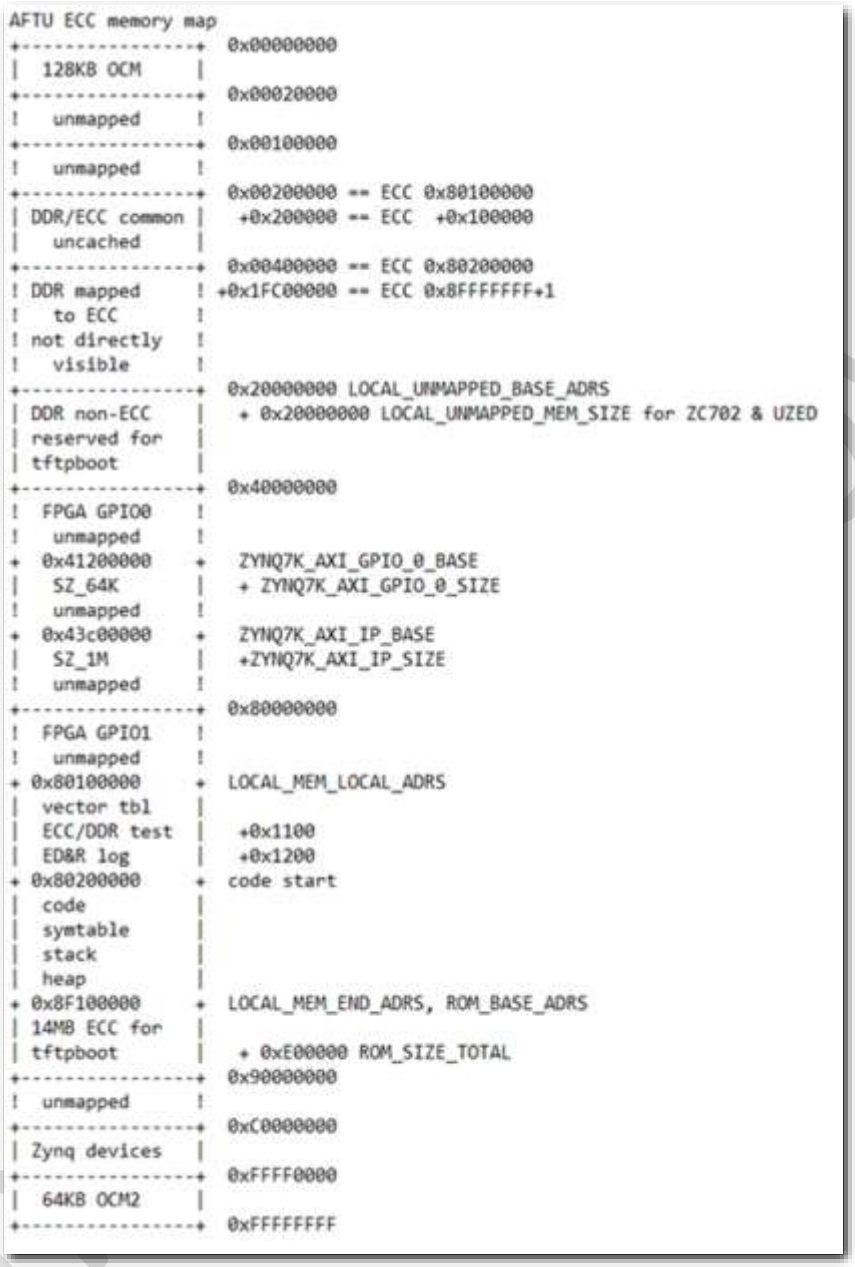


Figure 9: Base Zynq Memory Map

The **VxWorks Zynq Memory Map** is the current baseline that is in use. The AFTU configuration data can still be loaded into memory via bootgen. But the address range for this bootgen has been changed from **0x0FC00000** to **0x0FEFFFFF**.

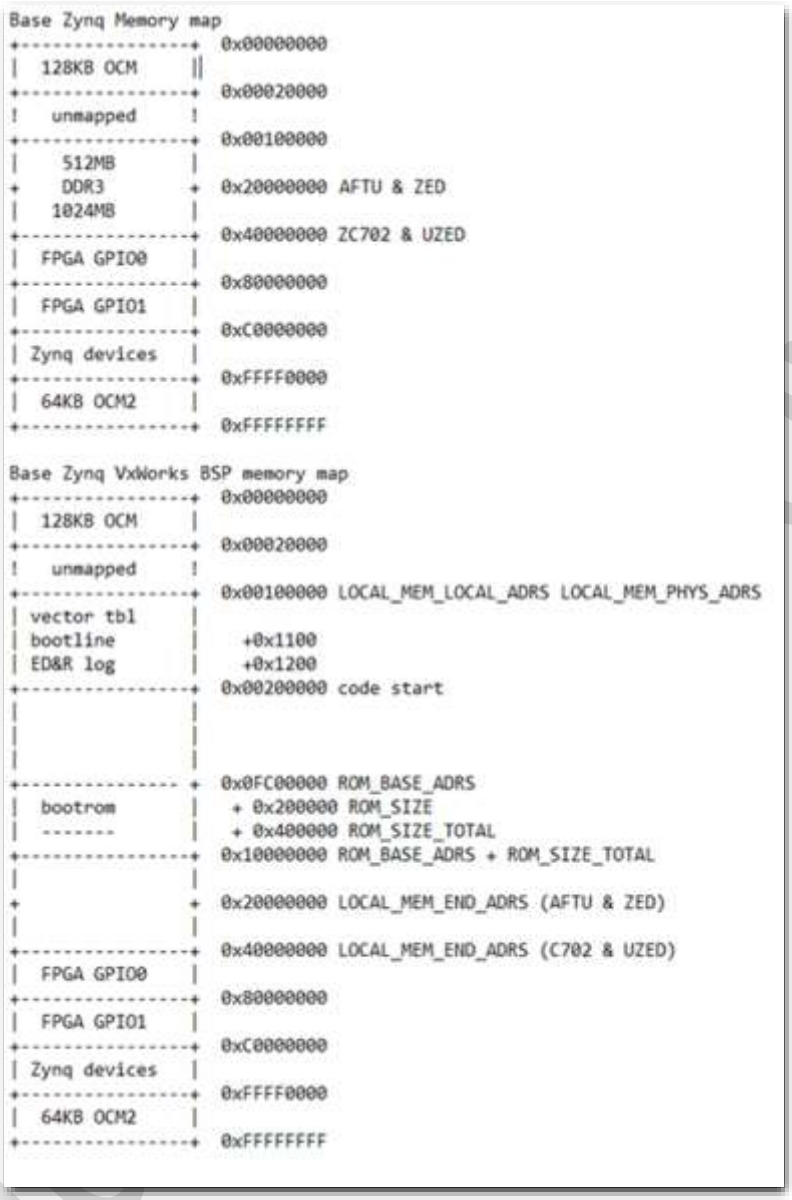


Figure 10: Base Zynq Memory Map

```
AFTU non-ECC memory map
+----------------+  0x00000000
|  128KB OCM     |
+----------------+  0x00020000
!   unmapped     !
+----------------+  0x00100000 LOCAL_MEM_LOCAL_ADRS LOCAL_MEM_PHYS_ADRS
| vector tbl     |
| reserved       |    +0x1100
| ED&R log       |    +0x1200
+----------------+  0x00200000 code start
| code           |
| symtable       |
| stack          |
| heap           |
+--------------- +  0x0FC00000 LOCAL_MEM_END_ADRS
|  reserved      |   + 0x200000 ROM_SIZE
|                |   + 0x400000 ROM_SIZE_TOTAL
+----------------+  0x10000000 LOCAL_UNMAPPED_BASE_ADRS
| DDR reserved   |   + 0x10000000 LOCAL_UNMAPPED_MEM_SIZE NAFTU & ZED
+                +  0x20000000
| for tftpboot   |   + 0x30000000 LOCAL_UNMAPPED_MEM_SIZE ZC702 & UZED
+----------------+  0x40000000
|  FPGA GPIO0    |
+----------------+  0x80000000
|  FPGA GPIO1    |
+----------------+  0xC0000000
| Zynq devices   |
+----------------+  0xFFFF0000
|  64KB OCM2     |
+----------------+  0xFFFFFFFF
```

Figure 11: VxWorks Memory Map Without ECC

```
AFTU ECC memory map
+----------------+  0x00000000
|  128KB OCM     |
+----------------+  0x00020000
!   unmapped     !
+----------------+  0x00100000
!   unmapped     !
+----------------+  0x00200000 == ECC 0x80100000
| DDR/ECC common |   +0x200000 == ECC  +0x100000
|   uncached     |
+----------------+  0x00400000 == ECC 0x80200000
! DDR mapped     ! +0x1FC00000 == ECC 0x8FFFFFFF+1
!   to ECC       !
! not directly   !
!   visible      !
+----------------+  0x20000000 LOCAL_UNMAPPED_BASE_ADRS
| DDR non-ECC    |   + 0x20000000 LOCAL_UNMAPPED_MEM_SIZE for ZC702 & UZED
| reserved for   |
| tftpboot       |
+----------------+  0x40000000
!  FPGA GPIO0    !
!   unmapped     !
+  0x41200000    +   ZYNQ7K_AXI_GPIO_0_BASE
|   SZ_64K       |   + ZYNQ7K_AXI_GPIO_0_SIZE
!   unmapped     !
+  0x43c00000    +   ZYNQ7K_AXI_IP_BASE
|   SZ_1M        |   +ZYNQ7K_AXI_IP_SIZE
!   unmapped     !
+----------------+  0x80000000
!  FPGA GPIO1    !
!   unmapped     !
+ 0x80100000     +  LOCAL_MEM_LOCAL_ADRS
|  vector tbl    |
|  ECC/DDR test  |   +0x1100
|  ED&R log      |   +0x1200
+ 0x80200000     +  code start
|  code          |
|  symtable      |
|  stack         |
|  heap          |
+ 0x8F100000     +  LOCAL_MEM_END_ADRS, ROM_BASE_ADRS
| 14MB ECC for   |
| tftpboot       |   + 0xE00000 ROM_SIZE_TOTAL
+----------------+  0x90000000
!  unmapped     !
+----------------+  0xC0000000
| Zynq devices   |
+----------------+  0xFFFF0000
|  64KB OCM2     |
+----------------+  0xFFFFFFFF
```

Figure 12: VxWorks Zynq Memory Map Using ECC

6.4 FPGA PHYSICAL MEMORY MAP

The software interacts with the **FPGA** via this physical address space. The **Zynq Processor System** (PS) contains its own set of registers and memories as denoted in **ps7_init**. A software read of any address in the below memory map will initiate a read request on the appropriate bus. The **FPGA** core, memory, or register will respond with the appropriate information. The same process will occur for a write.

Keep in mind that one can always directly access **DDR** memory between **0x0010_0000** to **0x1FFF_FFFF**. The **ECC** block maps to **DDR** memory by shifting address bits to the left by **1 bit**, effectively multiplying the address by **2**. Once **DDR** memory is initialized with **ECC-protected** words, a direct write to **DDR** memory may corrupt the **ECC-protected** words.

CELL	OFFSET ADDRESS	RANGE	HIGH ADDRESS
PS (See: ps7_init.html) The **ps7_init.html** stores a summary report of the register configuration. The Zynq IP output products can generate the **ps7_init.html** file and opened in an internet browser.	0x0000_0000		0x000F_FFFF
DDR memory	0x0010_0000		0x1FFF_FFFF
axi_gpio_0	0x4120_0000	64K	0x4120_FFFF
uarts/axi_uart16550_0	0x43C0_0000	64K	0x43C0_FFFF
uarts/axi_uart16550_1	0x43C1_0000	64K	0x43C1_FFFF
uarts/axi_uart16550_2	0x43C2_0000	64K	0x43C2_FFFF
uarts/axi_uart16550_3	0x43C3_0000	64K	0x43C3_FFFF
uarts/axi_uart16550_4	0x43C4_0000	64K	0x43C4_FFFF
uarts/axi_uart16550_5	0x43C5_0000	64K	0x43C5_FFFF
pattern16_ip_0	0x43C6_0000	64K	0x43C6_FFFF
pattern_ip_0	0x43C7_0000	64K	0x43C7_FFFF
xadc_wiz_0	0x43C8_0000	64K	0x43C8_FFFF
ECC_BLOCK/Custom_IP/customip_v1_0_0	0x8000_0000	256M	0x8FFF_FFFF
PS (see ps7_init.html)	0xE000_0000		0xFFFF_FFFF

Table 11: FPGA interface Table

ADDRESS	BYTE 3	BYTE 2	BYTE 1	BYTE 0
0x0010_0000	Word 0 ECC Byte 1	Word 0 ECC Byte 1	Word 0 ECC Byte 0	Word 0 ECC Byte 0
0x0010_0004	Word 0 ECC Byte 3	Word 0 ECC Byte 3	Word 0 ECC Byte 2	Word 0 ECC Byte 2
0x0010_0008	Word 0 ECC Byte 1	Word 0 ECC Byte 0	Word 0 ECC Byte 1	Word 0 ECC Byte 0
0x0010_000C	Word 0 ECC Byte 3	Word 0 ECC Byte 3	Word 0 ECC Byte 2	Word 0 ECC Byte 2

Table 12: DDR Memory Organization after ECC Initialization

6.5 AXI_GPIO_0

AXI_GPIO_0 is **Xilinx** Core. See: PG144, AXI GPIO v2.2, LogiCORE IP Product Guide for more information on this core.

This core is configured for 1 channel, readable and writable by software. The **1** channel has a mix of input and output-only signals mapped to a **20-bit** register. For a description of each bit, see Table 12: DDR Memory Organization after ECC Initialization. This **GPIO** core initializes to input only. The software writes the **tri-state** register to make **WD-RESET-A** and **WD-RESET-B** output signals. This **GPIO** core allows the software to monitor the status of 2 **Lift Off Detect**, 2 **Watchdog** health, **external** health, and various **talkback** signals.

The function of the core is simple. If the input to the **FPGA** meets the **LVCMOS33** standard of a logic '1', then the software will read a '**1**' from the corresponding bit in this register. If the input to the **FPGA** meets the **LVCMOS33** standard of logic '**0**', then the software will read a '**0**' from the corresponding bit in this register. **Zynq** switching characteristics can be found here:

https://www.xilinx.com/support/documentation/data_sheets/ds187-XC7Z010- XC7Z020-Data-Sheet.pdf

6.5.1 CHANNEL 0

NAME	DIRECTION	LSB OFFSET	DEFAULT	DESCRIPTION
WD-RESET-A	Out	00	None	Reset for Watchdog A.
WD-RESET-B	Out	01	None	Reset for Watchdog B.
WD-A-nHLTH	In	02	Pullup	Watchdog A, not health status.
WD-B-nHLTH	In	03	Pullup	Watchdog B, not health status.
EXT-HLTH	In	04	Pulldown	Cross-strap Watchdog health.
VCCWD-TB	In	05	Pulldown	Watchdog power talkback signal.
VCCWDXCVR-TB	In	06	Pulldown	Watchdog XCVR power talkback.
Reserved	In	07	Pullup	Reserved.
LODL-TTLA	In	08	Pulldown	Lift-off detector A signal.
LODL-TTLB	In	09	Pulldown	Lift-off detector B signal.
UMB-SAFE-DIS-A-TB	In	10	Pulldown	Reserved. Umbilical safe disabled A talkback.
UMB-SAFE-DIS-B-TB	In	11	Pulldown	Reserved. Umbilical safe disabled B talkback.
GPS-ERROR	In	12	Pulldown	Unused. Internal GPS error signal.
GPS-PVALID	In	13	Pulldown	Unused. Internal GPS position valid signal.
GPS-RTS2 (KEY STATUS1)	In	14	Pulldown	Unused. SAASM key status 1.
GPS-RTS1 (KEY STATUS2)	In	15	Pulldown	Unused. SAASM key status 2.
Mainboard-Config-1	In	16	None	Mainboard version bit 1.
Mainboard-Config-2	In	17	None	Mainboard version bit 2.
Mainboard-Config-3	In	18	None	Mainboard version bit 3.
Mainboard-Config-4	In	19	None	Mainboard version bit 4.

Table 13: GPIO Channel 1

6.6 AXI_GPIO_1

AXI_GPIO_1 is Xilinx Core. See: PG144, AXI GPIO v2.2, LogiCORE IP Product Guide for more information on this core.

This core is configured for **2** channels, read-only. This core exists to make **ECC** counters readable by software. Channel **0** is set to read the counter responsible for **double-bit** errors. Channel 1 is set to read the counter responsible for **single-bit** errors.

When reading each channel, **AXI_GPIO_1** will only return the current value of the counters.

6.7 UART 16550

These are five instances of the **Xilinx UART_16550 IP** in the **Processor Logic**. For more information on this IP, see PG143-axi-uart16550.pdf dated October 5, 2016.

This **UART** option was chosen because of **VxWorks** driver compatibility. One instance was created for each **FPGA**-implemented **serial** port. The **UART 16550** core supports **115200 BAUD**. Interrupts are mapped to the **IRQ_F2P** line that the **VxWorks xlnk7k BSP** otherwise used.

UARTs are labeled in the **FPGA** design, as seen below. The **/tyCo/#** is purely a software representation.

UART	SERIAL	VXWORKS TTY
Zynq UART	Console.	/tyCo/0
axi_uart16550_0	Internal GPS (unused).	/tyCo/1
axi_uart16550_1	External GPS, input only.	/tyCo/2
axi_uart16550_2	Telemetry.	/tyCo/3
axi_uart16550_5	RS-232 (may not be connected) IMU_TX	/tyCo/4
axi_uart16550_4	Spare.	/tyCo/5
axi_uart16550_3	INS.	/tyCo/6

Table 14: UART Assignments

6.8 PATTERN16_IP

The **pattern16_ip** custom core controls four termination (term) control lines per **TERM** circuit and **nTERM** circuit on the board for a total of **16 output** signals.

There are two **TERM** circuits labeled **Term1** and **Term2**. These two **TERM** circuits are identical. Do not confuse **Term1** and **Term2** with **nTERM** circuits. The control lines are notated as **AA**, **AB**, **BA**, and **BB**. See: The truth table in SG906-0015-NAFTU Internal ICD REV-1.7. Each control line has its independent register. On reset, the control lines initialize to a nominal state; the nominal state is (**0**, **1**, **0**, **1**); these correspond directly to (**AA**, **AB**, **BA**, **BB**) and will do so for the rest of this document. This is considered the nominal state because the negation of all of these values (**1**, **0**, **1**, **0**) is launch vehicle commanded termination.

There are two **nTERM** circuits labeled **nTerm1** and **nTerm2**. These two **TERM** circuits are identical. Do not confuse **nTerm1** and **nTerm2** with **TERM** circuits. The control lines are notated as **AA**, **AB**, **BA**, and **BB**. See: The truth table in SG906-0015-NAFTU Internal ICD REV-1.7. Each control line has its independent register. On reset, the control lines initialize to a nominal state; the nominal state is (**0**, **1**, **0**, **1**); these correspond directly to (AA, AB, BA, BB) and will do so for the rest of this document. (**0**, **1**, **0**, **1**) is considered the nominal state. **AA** and **AB** combine to create the high side. –**BA** and –**BB** combine together to create the low side. (**1**,**0**) in either of these two sides will result in a launch vehicle termination. Seven potential **FPGA** output combinations will assert ntermination. See: Table 4 in the SG906-0015-NAFTU Internal ICD REV-1.7.

Each **TERM** and **nTERM** control line has its individual register inside the **pattern16 ip** core. Each register has a unique pattern to which only that specific register will respond. When the correct pattern is written to the corresponding control line register, then that control line will assert and stay asserted. Assert in this scenario means that the control line will change to the launch vehicle commanded termination state. If the **AA** pattern were written to the control line **AA** register, the **AA** control line would transition from **0** to **1**. If the **AB** pattern were written to the control **AB** register, then the **AB** register would transition from **1** to **0**.

If the bit-wise **NOT** of a correct pattern is written to the corresponding control line register, then that control line will de-assert and stay de-asserted. If a correct pattern is written to the wrong register, the state of the output signals will not change. If random data is written to any register, the state of the output signals will not change.

The allowable delay between switching for constant current loads is **1ms** on the board side. The circuitry can support up to a **3**A constant current draw from the termination outputs. If pulsing

conditions occur, **Termination 1** and **Termination 2** should be performed **10**ms from each other with a 33% duty cycle. I.e., **Termination 1** on for **10ms**, **Termination 2** on for the next **10ms**, and both outputs are off for **10ms**. The **AFTU ICD** also states that it is important to execute **Termination** only after **nTermination** has been executed.

Flight software handles the control line timing and the order of **nTermination** and **Termination** asserting and de-asserting. Reads on these registers may occur anytime and will not change the output. To check the output values on the **FPGA** level, the software must read back on the patterns below in the corresponding register. If software reads back a high pattern, then the output is asserted. The output is de-asserted if the software reads back a low pattern. For all other values, the output is unknown unless the output was previously set or the core was previously reset. On the board level, the output of the **TERM** signals can be read back through talk-back signals through the ioboard **ADC** via **PS SPI** interface. Other talkbacks can be used to determine the state of the **TERM** circuits readable through the **XADC**.

Table 15: 16-bit TERM/nTERM Register contains the outputs to the main board from the **pattern16_ip** core. For all first-bit patterns, a high pattern will pull the designated output line to high. A **high pattern** will pull the designated output line to low for all second-bit patterns. The result of bringing the first bit high and the second bit low of the same **TERM** signal will bring the equivalent Table 15: 16-bit TERM/nTERM Register signal high. The reverse is true for bringing both patterns low.

NAME	MAP	HIGH PATTERN	LOW PATTERN	DEFAULT OUTPUT	UNPROGRAMMED	DESCRIPTION
V-TERM1-AA	0x43C60002	0xf42d	0x0BD2	'0'	Pulldown	TERM1 high side, the first bit
V-TERM1-AB	0x43C60000	0x8ccf	0x7330	'1'	Pullup	TERM1 high side, the second bit
V-TERM1-BA	0x43C60006	0xf646	0x09B9	'0'	Pulldown	TERM1 low side, the first bit
V-TERM1-BB	0x43C60004	0x5b04	0xA4FB	'1'	Pullup	TERM1 low side, the second bit
V-TERM2-AA	0x43C6000A	0x7f78	0x8087	'0'	Pulldown	TERM2 high side, the first bit
V-TERM2-AB	0x43C60008	0x4868	0xB797	'1'	Pullup	TERM2 high side, the second bit
V-TERM2-BA	0x43C6000E	0x56a7	0xA958	'0'	Pulldown	TERM2 low side, the first bit

NAME	MAP	HIGH PATTERN	LOW PATTERN	DEFAULT OUTPUT	UNPROGRAMMED	DESCRIPTION
V-TERM2-BB	0x43C6000C	0xb7ff	0x4800	'1'	Pullup	TERM2 low side, the second bit
V-nTERM1-AA	0x43C60012	0xbaf1	0x450E	'0'	Pulldown	TERM1 NOT high side, the first bit
V-nTERM1-AB	0x43C60010	0x0796	0xF869	'1'	Pullup	TERM1 NOT high side, the second bit
V-nTERM1-BA	0x43C60016	0x2ca5	0xD35A	'0'	Pulldown	TERM1 NOT low side, the first bit
V-nTERM1-BB	0x43C60014	0x4157	0xBEA8	'1'	Pullup	TERM1 NOT low side, the second bit
V-nTERM2-AA	0x43C6001A	0xb115	0x4EEA	'0'	Pulldown	TERM2 NOT high side, the first bit
V-nTERM2-AB	0x43C60018	0x8628	0x79D7	'1'	Pullup	TERM2 NOT high side, the second bit
V-nTERM2-BA	0x43C6001E	0xed43	0x12BC	'0'	Pulldown	TERM2 NOT low side, the first bit
V-nTERM2-BB	0x43C6001C	0x9589	0x6A76	'1'	Pullup	TERM2 NOT low side, the second bit

Table 15: 16-bit TERM/nTERM Register

6.9 PATTERN_IP

The **pattern16_ip** custom core controls four termination (term) control lines per **TERM** circuit and **nTERM** circuit on the board for a total of **16** output signals.

There are two **TERM** circuits labeled **Term1** and **Term2**. These two **TERM** circuits are identical. Do not confuse **Term1** and **Term2** with **nTERM** circuits. The control lines are notated as **AA**, **AB**, **BA**, and **BB**. See: The truth table in SG906-0015-NAFTU Internal ICD REV-1.7. Each control line has its independent register. On reset, the control lines initialize to a nominal state; the nominal state is (0, 1, 0, 1); these correspond directly to (AA, AB, BA, BB) and will do so for the rest of this document. This is considered the nominal state because the negation of all of these values (1, 0, 1, 0) is launch vehicle commanded termination.

There are two **nTERM** circuits labeled **nTerm1** and **nTerm2**. These two **TERM** circuits are identical. Do not confuse **nTerm1** and **nTerm2** with **TERM** circuits. The control lines are notated as **AA**, **AB**, **BA**, and **BB**. See: The truth table in SG906-0015-NAFTU Internal ICD REV-1.7. Each

control line has its independent register. On reset, the control lines initialize to a nominal state; the nominal state is (0, 1, 0, 1); these correspond directly to (AA, AB, BA, BB) and will do so for the rest of this document. (0, 1, 0, 1) is considered the nominal state. **AA** and **AB** combine to create the high side. –**BA** and –**BB** combine together to create the low side. (1,0) in either of these two sides will result in a launch vehicle termination. Seven potential **FPGA** output combinations will assert ntermination. See: Table 4 in the SG906-0015-NAFTU Internal ICD REV-1.7.

Each **TERM** and **nTERM** control line has its individual register inside the **pattern16 ip** core. Each register has a unique pattern to which only that specific register will respond. When the correct pattern is written to the corresponding control line register, then that control line will assert and stay asserted. Assert in this scenario means that the control line will change to the launch vehicle commanded termination state. If the **AA** pattern were written to the control line **AA** register, the **AA** control line would transition from **0** to **1**. If the **AB** pattern were written to the control **AB** register, then the **AB** register would transition from **1** to **0**.

If the bit-wise **NOT** of a correct pattern is written to the corresponding control line register, then that control line will de-assert and stay de-asserted. If a correct pattern is written to the wrong register, the state of the output signals will not change. If random data is written to any register, the state of the output signals will not change.

The allowable delay between switching for constant current loads is 1ms on the board side. The circuitry can support up to a **3**A constant current draw from the termination outputs. If pulsing conditions occur, **Termination 1** and **Termination 2** should be performed **10**ms from each other with a 33% duty cycle. I.e., **Termination 1** on for **10**ms, **Termination 2** on for the next **10**ms, and both outputs are off for **10**ms. The **AFTU ICD** also states that it is important to execute **Termination** only after **nTermination** has been executed.

Flight software handles the control line timing and the order of **nTermination** and **Termination** asserting and de-asserting. Reads on these registers may occur anytime and will not change the output. To check the output values on the **FPGA** level, the software must read back on the patterns below in the corresponding register. If software reads back a **high pattern**, then the output is asserted. The output is de-asserted if the software reads back a **low pattern**. For all other values, the output is unknown unless the output was previously set or the core was previously reset. On the board level, the output of the **TERM** signals can be read back through talk-back signals through the ioboard **ADC** via **PS SPI** interface. Other talkbacks can be used to determine the state of the **TERM** circuits readable through the **XADC**.

Table 15: 16-bit TERM/nTERM Register contains the outputs to the main board from the **pattern16_ip** core. For all first-bit patterns, a **high pattern** will pull the designated output line to high. A **high pattern** will pull the designated output line to low for all **second-bit** patterns. The result of bringing the first bit high and the second bit low of the same **TERM** signal will bring the equivalent Table 15: 16-bit TERM/nTERM Register signal high. The reverse is true for bringing both patterns low.

NAME	MAP	HIGH PATTERN	LOW PATTERN	DEFAULT OUTPUT	DEFAULT	DESCRIPTION
FPGA-SAFE-EN-A	0x43C70000	0x589de07e	0xA7621F81	'0'	Pulldown	Enable ON A; The FPGA design designates this trace as ENABLE_A
FPGA-SAFE-EN-B	0x43C70004	0x22981c3a	0xDD67E3C5	'0'	Pulldown	Enable ON B; The FPGA design designates this trace as ENABLE_B
WD-EN-A	0x43C70008	0x0fc10023	0xF03EFFDC	'0'	Pulldown	Enable for Watchdog A
WD-EN-B	0x43C7000C	0x71f2556b	0x8E0DAA94	'0'	Pulldown	Enable for Watchdog B
FPGA-SAFE-DIS-A	0x43C70010	0xe283abc6	0x1D7C5439	'1'	Pullup	Enable UNLATCH A; The FPGA design designates this trace as SAFE_A
FPGA-SAFE-DIS-B	0x43C70014	0xd2ec8d75	0x2D13728A	'1'	Pullup	Enable UNLATCH B; The FPGA design designates this trace as SAFE_B
GPS-RESETIN	0x43C70018	0xd2ec8d75	0x2D13728A	'0'	Pullup	Internal GPS reset signal
XFACTOR	0x43C7000C	0xb84c6e0a	0x47B391F5	'0'	Pullup	Enable SAASM

Table 16: Enable Registers

SPECIAL RULES:

- Either **FPGA-SAFE-EN-A** or **FPGA-SAFE-EN-B** high will latch the **masterEnable VCCTERM** circuit. The **VCCTERM** latch will remain in the prior state if both are low. If either register is still being held high, it is not possible to unlatch **VCCTERM** either via command or **FAILSAFE**.
- Both **FPGA-SAFE-DIS-A** and **FPGA-SAFE-DIS-B** are required to be high simultaneously to unlatch **masterEnable VCCTERM**. However, if either **FPGA-SAFE-EN-A** or **FPGA-SAFE-EN-B** are still high, **VCCTERM** remains enabled until both **FPGA-SAFE-DIS-A** and **FPGA-SAFE-DIS-B** are low and both **FPGA-SAFE-DIS-A** and **FPGA-SAFE-DIS-B** are low.
- Either **WD-EN-A** or **WD-EN-B** high will latch the failsafe **VCCWD** circuit. There is no unlatch other than power cycling.
- **GPS-RESETIN** and **XFACTOR** are not currently connected in hardware and have no effect.

Reads on these registers may occur anytime and will not change the output. Currently, the only way for software to check the output values of the core to the board is to set a register to one of the patterns below. If software reads back a high pattern, then the output is asserted. The output is de-asserted if the software reads back a low pattern. For all other values, the output is unknown unless the output was previously set or the core was previously reset.

6.10 XADC

The **XADC** is block design level, instantiated **Zynq** primitive, much like the **Zynq** Processing System. The **XADC** has specific pins and can monitor a few internal voltage rails. More information can be found in XADC Wizard PG091.

The **XADC** has been set up as a channel sequencer running at **1 MSPS** in continuous mode for this design. This means the **XADC** will continue to sample and convert the selected channels. All of the channels are selected. Averaging has been enabled for channels **0** to **15** and is set at **16** samples.

This **XADC** samples and monitors **16** external channels and **10** internal voltages.

6.10.1 CHANNELS

The following are the internal voltages monitored by the **XADC**.

NAME	DESCRIPTION
Temperature	On-Chip Temperature
VCCINT	VCCINT
VCCAUX	VCCAUX
VCCBRAM	VBRAM - 7 Series and Zynq
VCCPINT	VPINT
VCCPAUX	VREFN
VCCDDRO	VDDRO
VP/VN	Mainboard temp
VREFP	VREFP
VREFN	VREFN

Table 17: XADC Internal Monitored Voltages

The following are the channels of external voltages monitored by the **XADC**. The mainboard schematic labels these signals as **AD#P**, where # is the channel number. These signals are on **Bank 35**. The **FPGA** design labels these signals as **VAUX[15:0]**, where the number in brackets is the channel number.

CHANNEL #	NAME	DESCRIPTION
0	VCCTERM-TB	Term enable voltage
1	VCC1V35_VS	1.35V regulator
2	TERM1_I-TB	TERM1 current
3	nTERM1_I-TB	nTERM1 current
4	TERM1_HI-TB	TERM1 HI side switch voltage
5	TERM2_HI-TB	TERM2 HI side switch voltage
6	nTERM1_HI-TB	nTERM1 HI side switch voltage
7	nTERM2_HI-TB	nTERM2 HI side switch voltage
8	TEMPB	I/O board temperature sensor
9	VCC5V0_VS	DC-DC converter 5V output
10	TERM2_I-TB	TERM2 current
11	nTERM2_I-TB	nTERM2 current
12	TERM1_LOW-TB	TERM1 LOW side switch voltage
13	TERM2_LOW-TB	TERM1 LOW side switch voltage
14	nTERM1_LOW-TB	nTERM1 LOW side switch voltage
15	nTERM2_LOW-TB	nTERM1 LOW side switch voltage

Table 18: ADC Channels

6.11 ERROR CORRECTION CODE (ECC)

The **ECC IP** was adapted from the **Xilinx ZYNQ7000AP_Soc_32bitDDRAcccesWithECC** Example project. It was converted from an earlier version to **Vivado 2015.4**, and the conversion was tested on the **Xilinx ZC702** board using the memory tests in the **SDK**. The whole **FPGA** design has been converted from **Vivado 2015.4** to **Vivado 2019.1**. Other changes include:

- Bundling into sub-blocks for cleaner diagrams.
- Changing the input clock to use the **Sagrad AFTU 100Mhz OSC2** input.

> **NOTE:**
>
> The Zedboard and microZed do not have additional oscillators. The clock may be derived from a CPU clock source for those boards.

- Changing the **ECC** output clock from **350 MHz** to **250 MHz**. Testing at high ambient temperatures found a subtle timing issue with the exact part revisions of **Zynq** and **DDR3** parts used on the **Qual Sagrad** boards, resulting in lockups. The part revisions used on pre-**Qual Sagrad** boards did not seem to have the same issue. Several setup and hold violations were found. The clock was reduced from **250** to **200 MHz** to close timing.

ECC, **single error** correct, **double error** detect (SECDED) occurs on the byte level using a **Hamming** encoder/decoder set for a data width of **8 bits** and a calculated syndrome of **5 bits**. There are two counters attached to the **ECC** block. The first counter increments only if there is a single-bit error in any byte in a **32-bit word**; therefore, if a **32-bit word** has a single-bit error in multiple bytes, the counter will only increment once. The second counter increments only if there is a **double-bit** error in any byte in a **32-bit word**; therefore, if a **32-bit** word has double-bit errors in multiple bytes, the counter will only increment once. These two counters are sampled by **axi_gpio_1**. Either counter may increment more than once before the software reads the counters.

The custom part of this **ECC** is called **customip_v1_0_0**. This custom hdl is the glue between the four interfaces: **slave axi**, **master axi**, **encode ECC**, and **decode ECC**.

6.11.1 ECC_BLOCK/CUSTOM_IP/CUSTOMIP_V1_0_0

On the **AFTU**, the **256MB** region of IO space from **0x8008_0000** to **0x8FFF_FFFF** is mapped to DDR3 region **0x00100000** to **0x0FFFFFFF** using the **ECC Proxy IP**. The following description of how that works comes from the Xilinx document ZYNQ7000AP_SoC_ProxyDesign.pdf:

> *"ECC Proxy block adds byte level ECC to each AXI transaction. With Byte level ECC, 1 ECC byte is used for each data byte, which doubles the amount of read or write data. Transactions protected by the ECC proxy will have increased latency because effective memory bandwidth is reduced. System level performance will be less affected since the primary usage is to load DDR data to L2 cache. Outside of the increased memory latency to protected regions of DDR, the use of ECC protection should be transparent to the system. Application Processer unit (APU) of Zynq-7000 SoC normally accesses external DDR memory via L2 cache. All coherent access from APU goes through Snoop control*

unit (SCU) which in turn connected to L2-cache controller. There is dedicated port from L2- controller to DDR controller for reducing latency in accessing external DDR memory. When the ECC proxy is added to the Programmable Logic (PL) as shown below, APU transactions to General purpose Master AXI interface (M_AXI_GP1) are redirected to DDR through the ECC proxy, which adds ECC protection bytes to each data bytes. Data movement to/from DDR will be through High Performance Slave (HP) port."

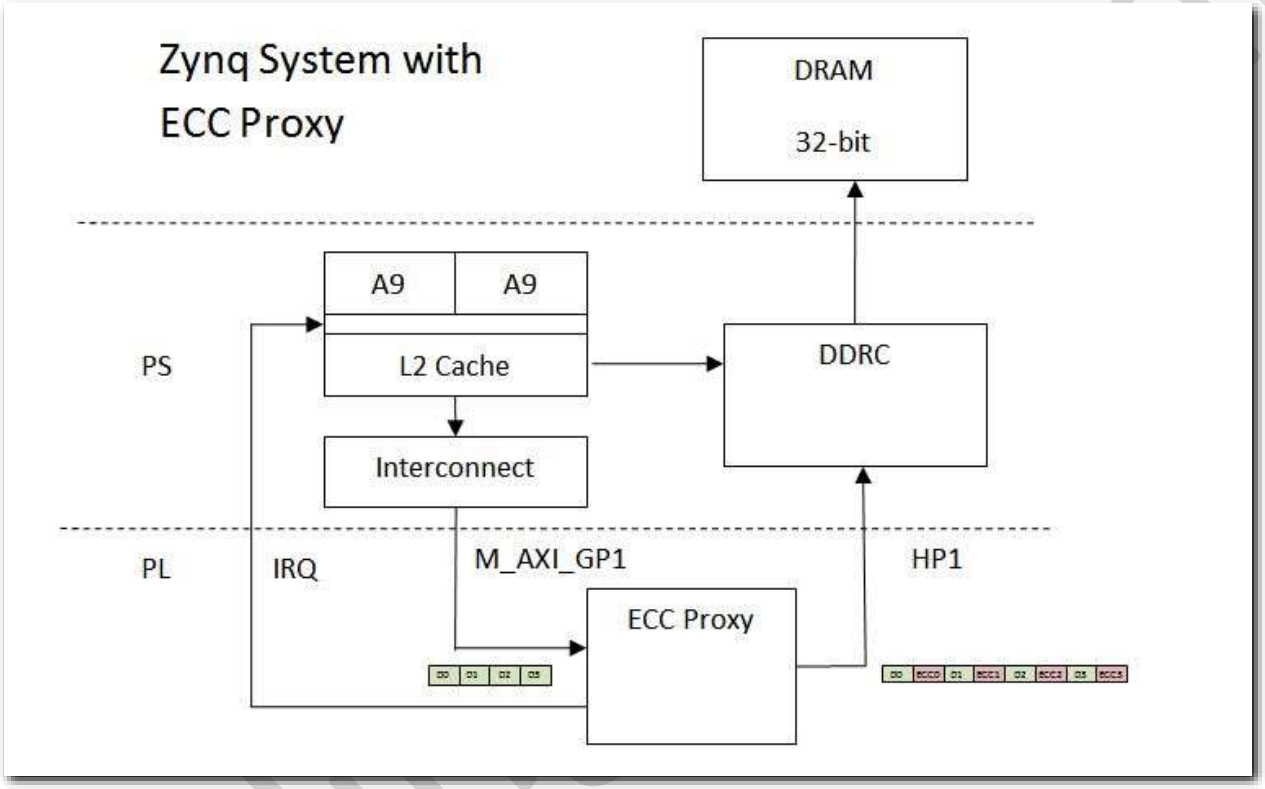


Figure 13: Zynq System with ECC Proxy

6.12 FPGA PIN ASSIGNMENTS AND UTILIZATION

The **FPGA pin assignments** map the logical outputs defined in the design to **FPGA** pins from the **Sagrad AFTU hardware layout**.

```
# internal GPS
set_property PACKAGE_PIN Y4 [get_ports uart16550_0_rx]
set_property IOSTANDARD LVCMOS33 [get_ports uart16550_0_rx]
set_property PACKAGE_PIN T6 [get_ports uart16550_0_tx]
set_property IOSTANDARD LVCMOS33 [get_ports uart16550_0_tx]

# EXT GPS RX  (tx is not connected)
set_property PACKAGE_PIN AA4 [get_ports uart16550_1_rx]
set_property IOSTANDARD LVCMOS33 [get_ports uart16550_1_rx]

# Telemetry
set_property PACKAGE_PIN W6 [get_ports uart16550_2_rx]
set_property IOSTANDARD LVCMOS33 [get_ports uart16550_2_rx]
set_property PACKAGE_PIN W5 [get_ports uart16550_2_tx]
set_property IOSTANDARD LVCMOS33 [get_ports uart16550_2_tx]

# INS
set_property PACKAGE_PIN U4 [get_ports uart16550_3_rx]
set_property IOSTANDARD LVCMOS33 [get_ports uart16550_3_rx]
set_property PACKAGE_PIN T4 [get_ports uart16550_3_tx]
set_property IOSTANDARD LVCMOS33 [get_ports uart16550_3_tx]

# SPARE
set_property PACKAGE_PIN U6 [get_ports uart16550_4_rx]
set_property IOSTANDARD LVCMOS33 [get_ports uart16550_4_rx]
set_property PACKAGE_PIN U5 [get_ports uart16550_4_tx]
set_property IOSTANDARD LVCMOS33 [get_ports uart16550_4_tx]

# IMU TX
set_property PACKAGE_PIN V9 [get_ports uart16550_5_tx]
set_property IOSTANDARD LVCMOS33 [get_ports uart16550_5_tx]
set_property PACKAGE_PIN V10 [get_ports uart16550_5_rx]
set_property IOSTANDARD LVCMOS33 [get_ports uart16550_5_rx]


# ENABLE_A
set_property PACKAGE_PIN R19 [get_ports {enable[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[0]}]
set_property PULLDOWN true [get_ports {enable[0]}]
# ENABLE_B
set_property PACKAGE_PIN P17 [get_ports {enable[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[1]}]
set_property PULLDOWN true [get_ports {enable[1]}]
# WD_ENABLE_A
set_property PACKAGE_PIN AB1 [get_ports {enable[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[2]}]
set_property PULLDOWN true [get_ports {enable[2]}]
# WD_ENABLE_B
set_property PACKAGE_PIN V5 [get_ports {enable[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[3]}]
```

```
set_property PULLDOWN true [get_ports {enable[3]}]
# SAFE_A
set_property PACKAGE_PIN T16 [get_ports {enable[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[4]}]
set_property PULLDOWN true [get_ports {enable[4]}]
# SAFE_B
set_property PACKAGE_PIN N15 [get_ports {enable[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[5]}]
set_property PULLDOWN true [get_ports {enable[5]}]
# GPS reset
set_property PACKAGE_PIN AA8 [get_ports {enable[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[6]}]
set_property PULLUP true [get_ports {enable[6]}]
# SAASM enable
set_property PACKAGE_PIN J15 [get_ports {enable[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[7]}]
set_property PULLUP true [get_ports {enable[7]}]

# V-TERM1-AA
set_property PACKAGE_PIN AB21 [get_ports {term[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[0]}]
set_property PULLDOWN true [get_ports {term[0]}]
# V-TERM1-AB
set_property PACKAGE_PIN P16 [get_ports {term[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[1]}]
set_property PULLUP true [get_ports {term[1]}]
# V-TERM1-BA
set_property PACKAGE_PIN AB22 [get_ports {term[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[2]}]
set_property PULLDOWN true [get_ports {term[2]}]
# V-TERM1-BB
set_property PACKAGE_PIN R18 [get_ports {term[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[3]}]
set_property PULLUP true [get_ports {term[3]}]
# VTERM2-AA
set_property PACKAGE_PIN AA16 [get_ports {term[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[4]}]
set_property PULLDOWN true [get_ports {term[4]}]
# V-TERM2-AB
set_property PACKAGE_PIN AB20 [get_ports {term[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[5]}]
set_property PULLUP true [get_ports {term[5]}]
# V-TERM2-BA
set_property PACKAGE_PIN Y18 [get_ports {term[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[6]}]
set_property PULLDOWN true [get_ports {term[6]}]
# V-TERM2-BB
set_property PACKAGE_PIN Y21 [get_ports {term[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[7]}]
set_property PULLDOWN true [get_ports {term[7]}]
# V-nTERM1-AA
set_property PACKAGE_PIN J20 [get_ports {term[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[8]}]
set_property PULLDOWN true [get_ports {term[8]}]
# V-nTERM1-AB
set_property PACKAGE_PIN M15 [get_ports {term[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[9]}]
```

```
set_property PULLUP true [get_ports {term[9]}]
# V-nTERM1-BA
set_property PACKAGE_PIN K18 [get_ports {term[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[10]}]
set_property PULLDOWN true [get_ports {term[10]}]
# V-nTERM1-BB
set_property PACKAGE_PIN N17 [get_ports {term[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[11]}]
set_property PULLUP true [get_ports {term[11]}]
# V-nTERM2-AA
set_property PACKAGE_PIN P20 [get_ports {term[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[12]}]
set_property PULLDOWN true [get_ports {term[12]}]
# V-nTERM2-AB
set_property PACKAGE_PIN N22 [get_ports {term[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[13]}]
set_property PULLUP true [get_ports {term[13]}]
# V-nTERM2-BA
set_property PACKAGE_PIN R20 [get_ports {term[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[14]}]
set_property PULLDOWN true [get_ports {term[14]}]
# V-nTERM2-BB
set_property PACKAGE_PIN L21 [get_ports {term[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {term[15]}]
set_property PULLDOWN true [get_ports {term[15]}]
# GPS_PPS binary counter
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets GPS_PPS_IBUF]
set_property PACKAGE_PIN AB11 [get_ports GPS_PPS]
set_property IOSTANDARD LVCMOS33 [get_ports GPS_PPS]

# OSC1
set_property PACKAGE_PIN Y9 [get_ports GCLK]
set_property IOSTANDARD LVCMOS33 [get_ports GCLK]


# WD-RESET-A
set_property PACKAGE_PIN U7 [get_ports {gpio_tri_io[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[0]}]
# WD-RESET-B
set_property PACKAGE_PIN AB2 [get_ports {gpio_tri_io[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[1]}]
# WD-HEALTH-A
set_property PACKAGE_PIN AA7 [get_ports {gpio_tri_io[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[2]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[2]}]
# WD-HEALTH-B
set_property PACKAGE_PIN AA6 [get_ports {gpio_tri_io[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[3]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[3]}]
# EXT_HEALTH
set_property PACKAGE_PIN R6 [get_ports {gpio_tri_io[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[4]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[4]}]
# VCCWD-TB
set_property PACKAGE_PIN AB6 [get_ports {gpio_tri_io[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[5]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[5]}]
```

```
# VCCWDXCVR-TB
set_property PACKAGE_PIN AB7 [get_ports {gpio_tri_io[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[6]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[6]}]
# dummy
set_property PACKAGE_PIN AB4 [get_ports {gpio_tri_io[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[7]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[7]}]
# LOD-A
set_property PACKAGE_PIN U20 [get_ports {gpio_tri_io[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[8]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[8]}]
# LOD-B
set_property PACKAGE_PIN U22 [get_ports {gpio_tri_io[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[9]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[9]}]
# UMB-SAFE-DIS-A-TB
set_property PACKAGE_PIN M19 [get_ports {gpio_tri_io[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[10]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[10]}]
# UMB-SAFE-DIS-B-TB
set_property PACKAGE_PIN M20 [get_ports {gpio_tri_io[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[11]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[11]}]
# GPS-ERROR
set_property PACKAGE_PIN AB10 [get_ports {gpio_tri_io[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[12]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[12]}]
# GPS-PVaLID
set_property PACKAGE_PIN AB9 [get_ports {gpio_tri_io[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[13]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[13]}]
# KEY_STATUS1
set_property PACKAGE_PIN AA9 [get_ports {gpio_tri_io[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[14]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[14]}]
# KEY_STATUS2
set_property PACKAGE_PIN Y10 [get_ports {gpio_tri_io[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[15]}]
set_property PULLDOWN true [get_ports {gpio_tri_io[15]}]


#Mainboard-Config1
set_property PACKAGE_PIN AA12 [get_ports {gpio_tri_io[16]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[16]}]
#Mainboard-Config2
set_property PACKAGE_PIN AB12 [get_ports {gpio_tri_io[17]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[17]}]
#Mainboard-Config3
set_property PACKAGE_PIN AA11 [get_ports {gpio_tri_io[18]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[18]}]
#Mainboard-Config4
set_property PACKAGE_PIN U21 [get_ports {gpio_tri_io[19]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_tri_io[19]}]


set_property IOSTANDARD LVCMOS25 [get_ports Vaux0_v_n]
```

```
set_property IOSTANDARD LVCMOS25 [get_ports Vaux0_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux1_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux1_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux2_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux2_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux3_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux3_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux4_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux4_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux5_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux5_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux6_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux15_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux15_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux14_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux14_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux13_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux13_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux12_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux12_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux11_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux11_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux10_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux10_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux9_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux9_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux8_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux8_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux7_v_p]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux7_v_n]
set_property IOSTANDARD LVCMOS25 [get_ports Vaux6_v_p]
```

Code Block 1: FPGA Pin Assignments Map

Resource	Utilization	Available	Utilization %
LUT	5756	53200	10.82
LUTRAM	506	17400	2.91
FF	8620	106400	8.10
IO	88	200	44.00
MMCM	1	4	25.00

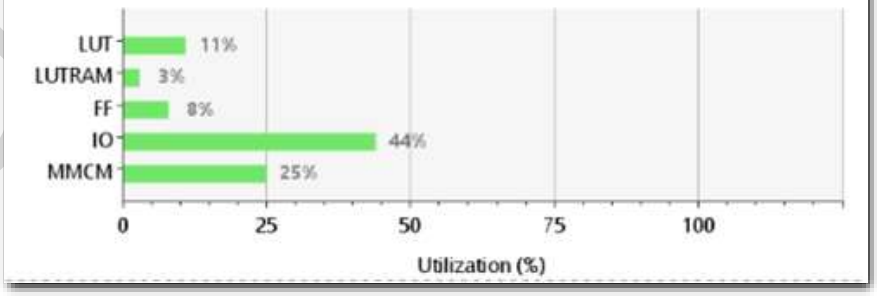


Figure 14: FPGA Utilization

6.13 INTERRUPTS

There are **eight interrupts** generated from the PL and sent to the PS. **Xilinx** cores generate all these interrupts. In the PL, they are mapped as follows:

INTERRUPT #	FPGA INTERRUPT SOURCE BLOCKS
0	UART 4
1	UART 5
2	UART 0
3	UART 1
4	UART 2
5	UART 3
6	AXI GPIO 0
7	XADC

Table 19: Interrupt Source Map

6.14 NAND FLASH

All software uses the **NAND** flash to store code and configuration data permanently. The **AFTU** organizes its **NAND** by predefined byte/block offsets. See: SG906-0013-AFTU User Manual, section 10.0 Aftu Memory CBIT Operation, for the **NAND** software structure. These offsets have been calculated to allow for **50% bad blocks** vs. the expected maximum size. Only the maintenance image and the **FSBL** will have the library code to write/erase arbitrary flash blocks.

The **NAND** flash has been made available through the **Zynq Processing System**. Please see the **static memory controller** (SMC) for the available features.

6.15 SPI

The **Wrapper** accesses the **SPI** bus through the built-in **Zynq SPI** interface using the Xilinx **xspips driver** ported to **VxWorks**. For more information on the **SPI** bus. See: **SPI Controller** in UG585-Zynq-7000-TRM. Outside of the **FPGA**, each **SPI ADC** is programmed to use the **Auto-2 register program** on the board. All 32 channels are read in sequence as **8-bit** values.

CHANNEL	HARDWARE NAME	DESCRIPTION
CH0	V-TERM2-D-HI-TB	TERM2 high-side gate voltage diagnostic
CH1	V-TERM1-D-LOW-TB	TERM1 low-side gate voltage diagnostic
CH2	V-nTERM1-D-LOW-TB	nTERM1 low-side gate voltage diagnostic
CH3	V-nTERM2-D-HI-TB	nTERM2 high-side gate voltage diagnostic
CH4	VCCTERM-TB	Voltage to termination circuits
CH5	V-TERM1-I-TB	Current through TERM1
CH6	V-nTERM1-I-TB	Current through nTERM1
CH7	VCC-TB	Input VCC
CH8	V-nTERM1-D-HI-TB	nTERM1 high-side gate voltage diagnostic
CH9	V-TERM1-D-HI-TB	TERM1 high-side gate voltage diagnostic
CH10	V-TERM2-D-LOW-TB	TERM2 low-side gate voltage diagnostic
CH11	V-nTERM2-D-LOW-TB	nTERM2 low-side gate voltage diagnostic
CH12	VCCWD-TB	Watchdog circuit voltage
CH13	VCCWDXCVR-TB	Watchdog external health transceiver voltage
CH14	UMB-SAFE-DIS-B-TB	Not used
CH15	UMB-SAFE-DIS-A-TB	Not used

Table 20: SPI ADC0 channels

CHANNEL	HARDWARE NAME	DESCRIPTION
CH0	V-nTERM1-LOW-TB	nTERM1 LO side switch voltage
CH1	V-TERM1-HI-TB	TERM1 HI side switch voltage
CH2	VCC-TB	Input VCC
CH3	VCCLODXCVR-TB	Liftoff detector transceiver voltage
CH4	V-nTERM2-I-TB	Current through nTERM2
CH5	V-TERM1-LOW-TB	TERM1 LO side switch voltage
CH6	V-TERM2-LOW-TB	TERM2 LO side switch voltage
CH7	V-TERM2-I-TB	Current through TERM2
CH8	Hardware-3	I/O Board version bit
CH9	Hardware-2	I/O Board version bit
CH10	Hardware-1	I/O Board version bit
CH11	Hardware-0	I/O Board version bit
CH12	V-nTERM2-LOW-TB	nTERM2 LO side switch voltage
CH13	V-nTERM2-HI-TB	nTERM2 HI side switch voltage
CH14	V-nTERM1-HI-TB	nTERM1 HI side switch voltage
CH15	V-TERM2-HI-TB	TERM2 HI side switch voltage

Table 21: SPI ADC1 channels

7 SOFTWARE

The **AFTU** software components are:

SOFTWARE COMPONENT	DESCRIPTION
Base First Stage Boot Loader (FSBL)	FSBL initializes, transfers images from NAND to RAM, and then transfers the control to U-Boot.
Enhanced First Stage Boot Loader (FSBL)	FSBL initializes, performs self-test, transfers images from NAND to RAM, initializes FPGA, and then transfers the control to U-Boot or VxWorks. The FSBL is based on Xilinx with customizations.
Field-Programmable Gate Array (FPGA) Bitstream	FPGA provides the interfaces between hardware and software. See: section Error: Reference source not found.
U-Boot (Maintenance Mode)	U-Boot allows the user to transfer an image or mission configuration from a GSE to RAM and then transfer it from RAM to NAND. The U-Boot is based on Xilinx with customizations.
VxWorks (Flight Mode)	VxWorks is a real-time operating system that runs the Wrapper as an application.
Wrapper	Wrapper manages hardware and passes parameters to CASS.
CASS Rule Engine	CASS Rule Engine is a library the Wrapper uses for processing mission rules based on MDL.

Table 22: Software Components

FSBL and **FPGA** bitstream are requirements of the **Zynq** architecture, which contains both **ARM** architecture processing cores and a programmable **FPGA** fabric. The division of executable software into separate maintenance and flight modules is to allow the separation of flight software, which contains safety-critical functions, from the support-critical functions not needed for flight. For example, the end user will not be able to change images or configurations while it is running the mission rules.

The version description for each component and off-the-shelf supporting software can be found in the SG901-0013-AFTU User Manual, "Image Build Instruction" section.

These software components are packaged in the form of an image. The user would transfer an image from **GSE** to the **NAND**, which will be verified during end-to-end testing. Once verified, a power cycle will allow the unit to boot automatically with the selected **MDL** for the mission. There are **flight and recovery** images, each of which is made of the following software components:

	FLIGHT IMAGE	RECOVERY IMAGE
Software Components	▪ Enhanced FSBL ▪ FPGA Bitstream ▪ U-Boot ▪ VxWorks ▪ Wrapper ▪ CASS Rule Engine	▪ Base FSBL ▪ FPGBitstream ▪ U-Boot
Description	This image is the default image and comes with the Wrapper software for an actual mission operation.	This image is used when the flight image is not bootable. This image contains only the necessary functions for the users to update a new flight image. This does not include the Wrapper software.

Table 23: Software Images

Instructions on how to build the images can be found in the SG906-0013-AFTU User Manual, section "Image Build Instruction." Instructions on transferring the images to the **NAND** can be found in the SG906-0013-AFTU User Manual, section "Binary Load Instruction."

The following diagram describes the logic flow among these software components and images.

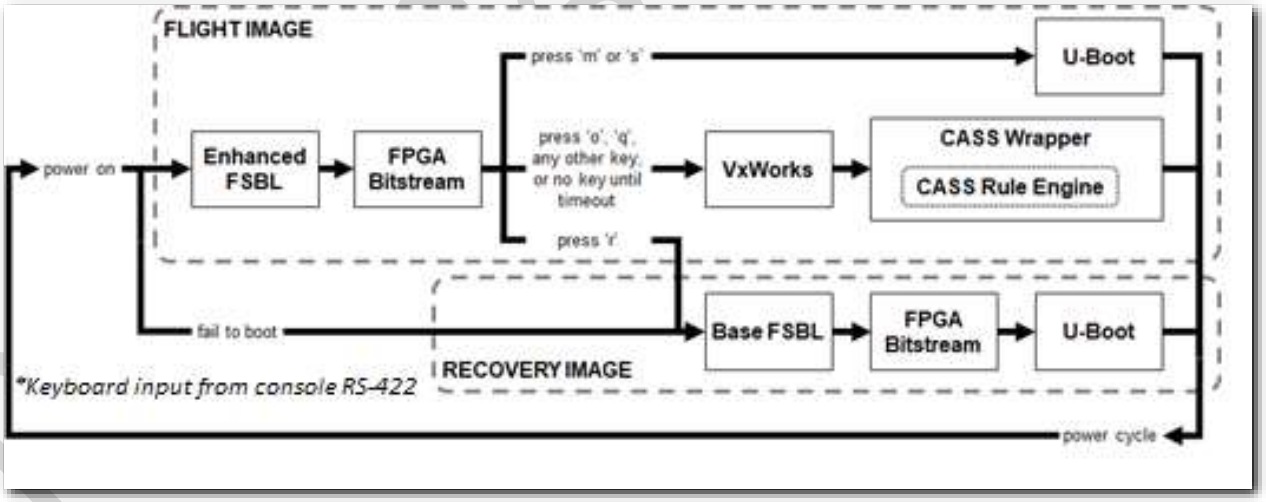


Figure 15: Software Components Logic Flow

The following subsections provide additional information on each software component.

7.1 BASE FSBL

Base First Stage Boot Loader (FSBL) allows users to boot the AFTU using **Xilinx**-provided **FSBL** code without customizations. This is used for recovery purposes. Information about the **FSBL** can be found in the Zynq-7000 All Programmable SoC Software Developers Guide, chapter 3, "Boot and Configuration."

7.2 ENHANCED FSBL

Enhanced First Stage Boot Loader (FSBL) is the same as the **Base FSBL** but with customizations to include functions needed to operate the AFTU. This is written in C programming language. The following lists the changes from the **Xilinx**-generated **FSBL** code.

NEW OR MODIFIED FILE	MODIFICATION DESCRIPTION
Bitstream-FSBL/AFTU.sdk/enhanced_fsbl/src/boot_info.h	1. Define parameters that are common to FSBL, VxWorks, and Wrapper.
Bitstream-FSBL/AFTU.sdk/enhanced_fsbl/src/main.c	1. Output FSBL version information to the console. 2. Prompt for boot mode with a timeout. 3. Output boot progress to console. 4. Test RAM using all five built-in memory test functions.
Bitstream-FSBL/AFTU.sdk/enhanced_fsbl/src/fsbl.h	1. Enable to output of timing information to the console. 2. Enable population of boot info structure. 3. Output Xilinx release version. 4. Define region for ECC memory test and initialization.
Bitstream-FSBL/AFTU.sdk/enhanced_fsbl/src/fsbl_debug.h	1. Enable verbose message output.
Bitstream-FSBL/AFTU.sdk/enhanced_fsbl/src/pcap.c	1. Output loading bitstream status to console. 2. Configure the "processing system" in FPGA.
Bitstream-FSBL/AFTU.sdk/enhanced_fsbl/src/fsbl_hooks.c	After downloading the bitstream into FPGA:

NEW OR MODIFIED FILE	MODIFICATION DESCRIPTION
	1. Set off to all registers for TERM and nTERM. 2. Test and initialize ECC memory with 0xDEADBEEF 32-bit pattern. Before transferring the control to the VxWorks: 1. Read the first 512 bytes of the U-Boot environment table from NAND. 2. Scan the environment for strings the AFTU uses and save that data in the boot config area.
Bitstream-FSBL/AFTU.sdk/enhanced_fsbl/src/image_mover.[c|h]	1. Output partition loading and MD5 verification information to the console. 2. Load recovery image if requested. 3. Skip the U-Boot partition if requested to boot to the flight mode. 4. Stop continuing loading partitions if requested to boot to the maintenance mode. 5. Save the MD5 results to boot info.
Bitstream-FSBL/AFTU.sdk/enhanced_fsbl/src/nand.c	1. Output information on NAND part and configuration to console. 2. Output information on NAND Bad Block Table to console.
Bitstream-FSBL/AFTU.sdk/enhanced_fsbl_bsp/ps7_cortexa9_0/libsrc/nandps_v2_2/src/xnandps_bbm.c	1. Output information about any bad blocks found in NAND to the console. 2. Add an option to force the Bad Block Table to be recreated by scanning.
Bitstream-FSBL/AFTU.sdk/enhanced_fsbl_bsp/ps7_cortexa9_0/libsrc/nandps_v2_2/src/xnandps_onfi.c	1. Add a new function (NandInfo) to query and print verbose information on the NAND part and configuration. 2. Add a new function (NandBbtInfo) to print verbose information on the Bad Block Table.

Table 24: List of modified items in FSBL

Once powered on, the Processing System samples the bootstrap pins to determine which memory to access (only NAND in AFTU design). On the AFTU, resistors shall be tied to ground to tell the SoC to load from the **NAND** flash. The resistors are tied schematically from **DXP_0 and DXN_0**, **pin N11** and **12**, respectively, from the **SoC** to ground. From there, the **Stage 0 Bootloader** is loaded into an internal nonvolatile memory, which loads the boot image onto volatile memory.

The **Zynq-7000 AP System on Chip** (SoC) devices support the following boot processes, as described in Chapter 6 of UG585:

- **Stage-0 Boot** (on-chip BootROM)
- **First Stage Bootloader** (FSBL)

The AFTU uses the following **Zynq** boot options:

- **OCM ROM CRC** check (need to enable via eFuse)
- **Boot from NAND** (non-volatile programmable flash memory)
- **MD5 checksum verification** of **FPGA** bitstream and combined VxWorks+Wrapper Image before use
- Finally, the **FSBL transfers control to the Flight Software** (or maintenance mode) image for execution.

After **power-good is asserted**, firmware in the Zynq does an internal **cyclic redundancy check** (CRC) of the **on-chip ROM** and then searches the first **128MB NAND** for the signature of a **Zynq BootROM** header with a valid checksum. At this point, it copies the **FSBL** code/data into the first 192K of on-chip **RAM**.

Since the **Zynq** cannot map **NAND** memory into a linear address space, all programs and data must be first copied from **NAND** into temporary working memory before use. The integrity verification (MD5) is always performed on the **RAM** copy.

The following describes the overall control/data flow of the **FSBL**, starting while the call to main in "main.c":

- Initializes **MIO**, **PLL**, **CLK**, and **DDR** devices
- Offers the user a boot option by pressing one of the following keys through the console **RS-422** port:
  - '**o**' for flight mode
  - '**q**' for flight mode without memory test
  - '**m**' for maintenance mode
  - '**s**' for maintenance mode without memory test
  - '**r**' for recovery mode
  - any other or no key is the same as '**o**'
- Performs a series of **RAM** tests on the raw **DDR**. The results of these tests are printed on the console and stored in **On-Chip Memory** (OCM).
- Initializes the interface to the **FPGA**
- Initializes the **FSBL NAND** driver
  - Initialize **NAND** device
  - Load/initialize **NAND** bad block table
- Process the bitstream partition:
  - Copy the **FPGA** bitstream from **NAND** to raw **DDR3**
  - Verify the **MD5** checksum
  - Program the **FPGA** from the **RAM** image. It is after this point **ECC RAM** is available.
- Initializes **ECC DDR3** to the fill pattern (AFTU addition).
- If maintenance mode has been selected (the default is to skip it), the **FSBL** ignores all other partitions, loads the **U-Boot** image into memory, performs the integrity check (MD5), and then

passes control to it. Otherwise, the **FSBL** does not load the **U-Boot** image or perform its integrity check.

- Loads the Flight Software into ECC memory - the location is specified in the ELF header at link time. The **FSBL** will compute and compare an **MD5SUM** of the image.
- Loads any other images into the memory address specified when the images were created.
- Reads system configuration data from **NAND** to **OCM**.
- Finally, the **FSBL** transfers control to the **Flight Software** (or maintenance mode) image for execution.

Any detected failure will abort boot into **Flight Software** (failure will not abort boot into maintenance mode software).

7.2.1 POWER-ON SELF-TEST (POST)

When the **AFTU** is powered up, it performs the **Power-On Self-Test** (POST) that checks the hardware status. Error severity is not categorized within this processing state, and errors are stored for later action. Any failure will halt the **AFTU**.

Power-on self-tests in the **Zynq** system:

- **OCM ROM** checksum
- **Boot ROM** header checksum

Power-on self-test in the **FSBL**:

- Raw **DDR3** memory tests - Incrementing Value Test
  - Walking Ones test
  - Walking Zeros test
  - Inverse Address test
  - Fixed Pattern test
- Test ICache and DDache
- **NAND** volume checksums
- **ECC** fill/verify test
  - **Fixed Pattern** test in **ECC** memory range

7.2.2 BOOT INFORMATION AND CONFIGURATION

The **FSBL** and **Wrapper** share the boot information by defining the '**boot_info.h**' header file. The **FSBL** populates the boot info region from self-test results and reads the image from **NAND**. It is located in **Zynq On-Chip Memory** (OCM) at absolute address **0xFFFFD000** with a size of 4096 bytes. **Zynq OCM** is used as it is available before **DDR3** has been initialized and is not otherwise used after the **FSBL** transfers control.

ADDRESS OFFSET	BYTE SIZE	TYPE	DESCRIPTION
0xFFFFD000	9	string	Magic string "AFTUBOOT" to identify boot info.
0xFFFFD009	1	8-bit unsigned integer	Incrementing value memory test result. 0 = failed, 1 = passed, 255 = not tested, otherwise = unknown.
0xFFFFD00A		8-bit unsigned integer	Walking ones memory test result. 0 = failed, 1 = passed, 255 = not tested, otherwise = unknown.
0xFFFFD00B	1	8-bit unsigned integer	Incrementing value memory test result. 0 = failed, 1 = passed, 255 = not tested, otherwise = unknown.
0xFFFFD00C	1	8-bit unsigned integer	Walking zero memory test result. 0 = failed, 1 = passed, 255 = not tested, otherwise = unknown.
0xFFFFD00D	1	8-bit unsigned integer	Inverse address memory test result. 0 = failed, 1 = passed, 255 = not tested, otherwise = unknown.
0xFFFFD00E	1	8-bit unsigned integer	Fixed pattern memory test result. 0 = failed, 1 = passed, 255 = not tested, otherwise = unknown.
0xFFFFD00F	8	64-bit unsigned integer	Partition 0 MD5SUM (not used)
0xFFFFD017	8	64-bit unsigned integer	Partition 1 MD5SUM (FPGA Bitstream)
0xFFFFD01F	8	64-bit unsigned integer	Partition 2 MD5SUM (U-Boot - if loaded)
0xFFFFD027	8	64-bit unsigned integer	Partition 3 MD5SUM (VxWorks + Wrapper + CASS code)
0xFFFFD02F	8	64-bit unsigned integer	Partition 4 MD5SUM (VxWorks + Wrapper + CASS symbols)
0xFFFFD037	8	64-bit unsigned integer	Partition 5 MD5SUM (not used)
0xFFFFD03F	8	64-bit unsigned integer	Partition 6 MD5SUM (not used)
0xFFFFD047	8	64-bit unsigned integer	Partition 7 MD5SUM (not used)
0xFFFFD04F	4017	8-bit unsigned integer	Reserved for future use.

Table 25: Boot info Layout

The boot configuration is located at the absolute address **0xFFFFD200** with a size of **256** bytes. It holds configuration information about the **NAFTU**.

ADDRESS OFFSET	BYTE SIZE	TYPE	DESCRIPTION
0xFFFFD200	8	string	AFTU serial number.
0xFFFFD208	8	string	Mainboard serial number.
0xFFFFD210	17	string	Ethernet MAC address (e.g., 01:23:45:67:89:ab).
0xFFFFD221	223	n/a	Reserved for future use.

Table 26: Layout of the Boot Configuration Information Region.

The boot line argument is located at the absolute address **0xFFFFD300** with a size of **3328** bytes.

ADDRESS OFFSET	BYTE SIZE	TYPE	DESCRIPTION
0xFFFFD300	256	string	Boot line arguments for VxWorks.
0xFFFFD400	3072	n/a	Reserved for future use.

Table 27: Layout of the Boot Line Argument Information Region.

7.3 U-BOOT

U-Boot allows the user to boot the **AFTU** into the maintenance mode using **Xilinx**-provided **FSBL** code with customizations to include needed functions. The maintenance mode is used to update the images or mission configurations. Information about the **U-Boot** can be found in the **Xilinx U-Boot** codebase repository. This is written in **C** programming language. The following lists the changes from the **Xilinx U-Boot** code.

NEW OR MODIFIED FILE	DESCRIPTION
Maintenance-Mode/arch/arm/dts/Makefile	1. Add reference to device definition.
Maintenance-Mode/arch/arm/dts/zynq-aftu.dts	1. Board definition file.
Maintenance-Mode/configs/zynq_aftu_defconfig	1. Include/exclude U-Boot options and commands. 2. Enable CMD_TFTPPUT, NAND_ZYNQ, CMD_NAND, CMD_CRC32, MD5, CMD_MD5SUM, ENV_IS_IN_FLASH. 3. Set ENV_ADDR.

NEW OR MODIFIED FILE	DESCRIPTION
Maintenance-Mode/drivers/mtd/nand/nand_base.c	1. Define ECC layout.
Maintenance-Mode/include/configs/zynq-common.h	1. Set environment options. 2. Set program options.
Maintenance-Mode/include/version.h	1. Set software version.
Maintenance-Mode/.gitignore	1. Remove from ignoring "*.S" files. 2. Remove from ignoring the "u-boot" file.

Table 28: List of Modified Items in U-Boot

When the Maintenance Mode option is selected shortly after power-up, only the executable code in **Partition 2** is loaded. The **FSBL** transfers control to the **U-Boot** program for the console operator or **GSE** to issue commands to transfer, read, write, and verify **AFTU Configuration** and program images and to optionally run external diagnostic programs from **GSE**, as described in Figure 16: U-Boot Flow Diagram. The **AFTU** expects the **U-Boot** environment to be stored in **NAND** offset **0x20000000**; see SG906-0013-AFTU User Manual, section 11.0, SysConfigData.

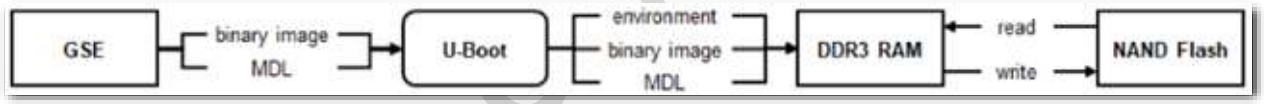


Figure 16: U-Boot Flow Diagram

7.4 VXWORKS

VxWorks is a commercial off-the-shelf (COTS) operating system that runs the **Wrapper** program. This section describes areas in the **VxWorks** modified to work with the **Wrapper** and **AFTU**. **VxWorks** is written in **C** programming language. The **VxWorks** system image is created with the following options:

- **VxWorks** System Image
- bsp xlnx_zynq7k
- 32-bit kernel
- gnu toolset
- smp -debug_opt

- PROFILE_STANDALONE_DEVELOPMENT

NEW OR MODIFIED FILE	DESCRIPTION
VxWorks/xlnx_zynq7k/boot_info.h	1. Define parameters that are common to FSBL, VxWorks, and Wrapper.
VxWorks/xlnx_zynq7k/config.h	1. Define default boot line macro. 2. Define and edit memory addresses and size macros for ECC memory. 3. Define mapping address macros for FPGA such as GPIO 0, GPIO 1, and IP. 4. Define macros for additional serial ports. 5. Define macros for NAND.
VxWorks/xlnx_zynq7k/hwconf.c	1. Add driver configuration data for the NS16550 serial ports. The first few settings are generic to adding new ports. The hcfResource settings match the FPGA details on the register and interrupt mappings. 2. Set the physical address.
VxWorks/xlnx_zynq7k/Makefile	3. Edit ROM addresses and size.
VxWorks/xlnx_zynq7k/sysLib.c	1. Enable debug macro. 2. Set a region in RAM to serve as ECC memory space. 3. Set a memory map in MMU for FPGA, such as GPIO 0, GPIO 1, and IP. 4. Set memory map in MMNU for SPI and NAND.
VxWorks/xlnx_zynq7k/sysNet.c	1. Enable debug macro. 2. Read the MAC address from RAM placed by FSBL.
VxWorks/xlnx_zynq7k/target.ref	1. Edit interface names. 2. Edit macro name.
VxWorks/xlnx_zynq7k/xlnx_zynq7k.h	1. Add definitions for INT_F2P user interrupt. 2. Set the system clock rate to half.

NEW OR MODIFIED FILE	DESCRIPTION
	3. Set the tcc clock rate to half.
VxWorks/usrAppInit.c	1. Add the user application startup hook to examine the VxWorks BOOT_LINE parsed boot arguments. If the "o" field contains valid **startAfss** options, the flight software's main task will automatically start with those options. If no “o=” options exist, boot to the VxWorks command prompt, and **startAfss** is run manually.

Table 29: List of Modified Items in VxWorks

7.4.1 PROJECT SETTINGS

1. Set the following parameters to configure **AFTU** clocks, serial ports, and console message buffers.

```
SYS_CLK_RATE =
AUX_CLK_RATE_MAX = ()
DEFAULT_BOOT_LINE = "gem(,)host:AFTU_flight.elf f=x tn=AFTU"
MAX_LOG_MSGS =
NUM_TTY = (N_SIO_CHANNELS + N_NS)
```

Code Block 2: AFTU Clocks, Serial Ports, and Console Message Buffers

2. Set the following parameters to configure to use the **AFTU** boot info for the **VxWorks** boot line.

```
NV_RAM_SIZE = NONE
BOOT_LINE_ADRS = ((char *) (xffffd))
```

Code Block 3: AFTU Boot

3. Change the following kernel parameters if using the **AFTU 256MB** memory map with **ECC**.

```
LOCAL_MEM_LOCAL_ADRS = x
LOCAL_MEM_SIZE = (LOCAL_MEM_END_ADRS - LOCAL_MEM_LOCAL_ADRS)
LOCAL_UNMAPPED_BASE_ADRS = x
RAM_HIGH_ADRS = xfc
RAM_LOW_ADRS = x
```

Code Block 4: Change Kernel Parameters

4. Change the following values in **xlnx_zynq7k/Makefile**. If using the **AFTU 256MB** memory map with **ECC**.

```
ROM_TEXT_ADRS  = fc # ROM entry address
ROM_SIZE       =  # number of bytes of ROM space
RAM_LOW_ADRS   =  # RAM text/data address
RAM_HIGH_ADRS  = fc # RAM text/data address
```

Code Block 5: Change Values in xlnx_zynq7k/Makefile

7.5 WRAPPER AND CASS RULE ENGINE

The **Wrapper** wraps the **Core Autonomous Safety Software** (CASS) Rule Engine, providing navigation sensors, liftoff detectors, termination outputs, **GSE**, and **telemetry interface** interfaces. This runs on the **VxWorks** operating system as an application. This is written in **C++** programming language. Information about the **CASSRule Engine** can be found in the **CASS** distribution package. See: Core Autonomous Safety Software (Cass) References:

The design structure in the **Wrapper** can be broken into four parts: 1) initialization, 2) inputs, 3) processing, and 4) outputs. Before the **Wrapper** begins with its main activities, it performs the initialization activities. The initialization activities include reading an **MDL** from the **NAND** flash, setting up several objects, and starting several threading tasks. These activities start when the **startAfss** function is called automatically. Any errors that occur during the initialization will result in the **Wrapper** transitioning to the **Stop Processing** state, which would require a power cycle to exit from. The error shall be transmitted in the console **RS-422** and may be seen from the telemetry **RS-422**.

Once the initialization activities are executed without detected errors, the **Wrapper** will proceed with the input, processing, and output activities until the **AFTU** is power cycled. On the input side, sensor navigation messages, liftoff detections, and digitized analog measurements are gathered for processing. User's commands are accepted in changing states, including rule processing, master enable, **Watchdog** enable, and end of mission.

In the processing component of the design structure, the mission rules contained within the **MDL** are executed within the **CASS** rule engine using the navigation data and liftoff statuses received from the input side. **Continuous built-in tests** (CBIT) will monitor select data items, select voltage levels within measurements within the **AFTU** and the state of the **termination** circuitry, and test for critical failures. An unhealthy status would indicate stopping the **Watchdog** reset and performing a fail-safe function. The **Watchdog** timer is reset only when the **Wrapper** is healthy on the output side. Lock and key patterns are sent to the **FPGA** only if a recommendation is made from the **CASS**. The **lock**

and key pattern is a **32-bit** number that the **FPGA** uses to check to confirm the **Wrapper** correctly commands the hardware. Various statuses and information are sent to a **GSE** via telemetry and console.

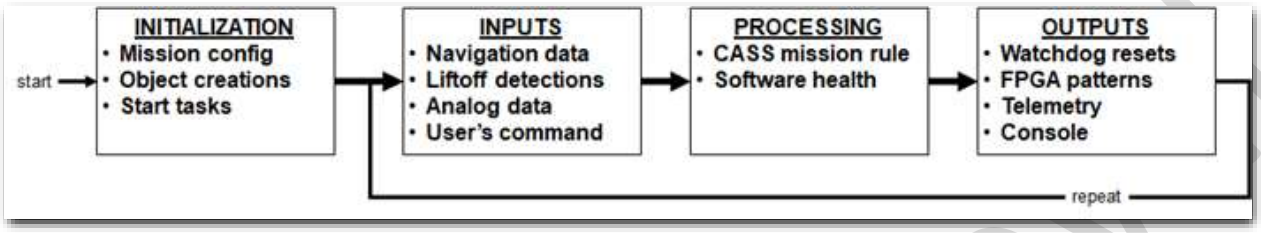


Figure 17: Design Structure in Wrapper

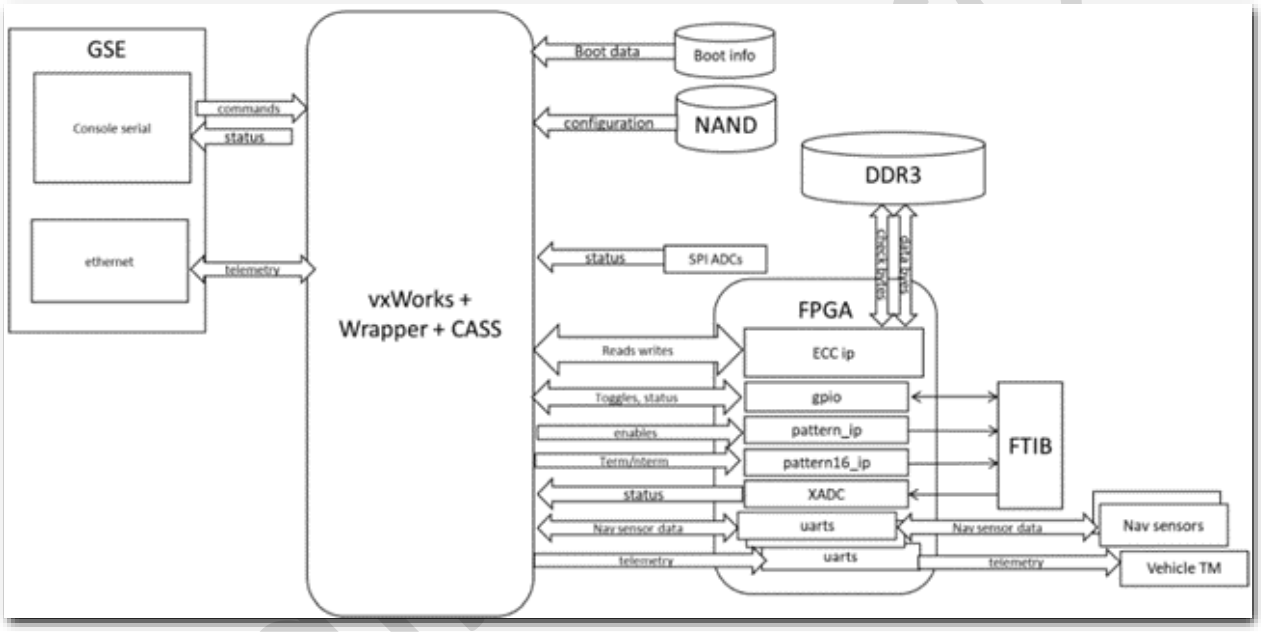


Figure 18: Flight Software Context Diagram

7.5.1 WRAPPER PROCESSING STATES

Several points in the **Wrapper**'s processing allow an event to change the **Wrapper**'s state, which represents how the **Wrapper** executes certain activities. The relationship between events and states is illustrated in the following diagram:

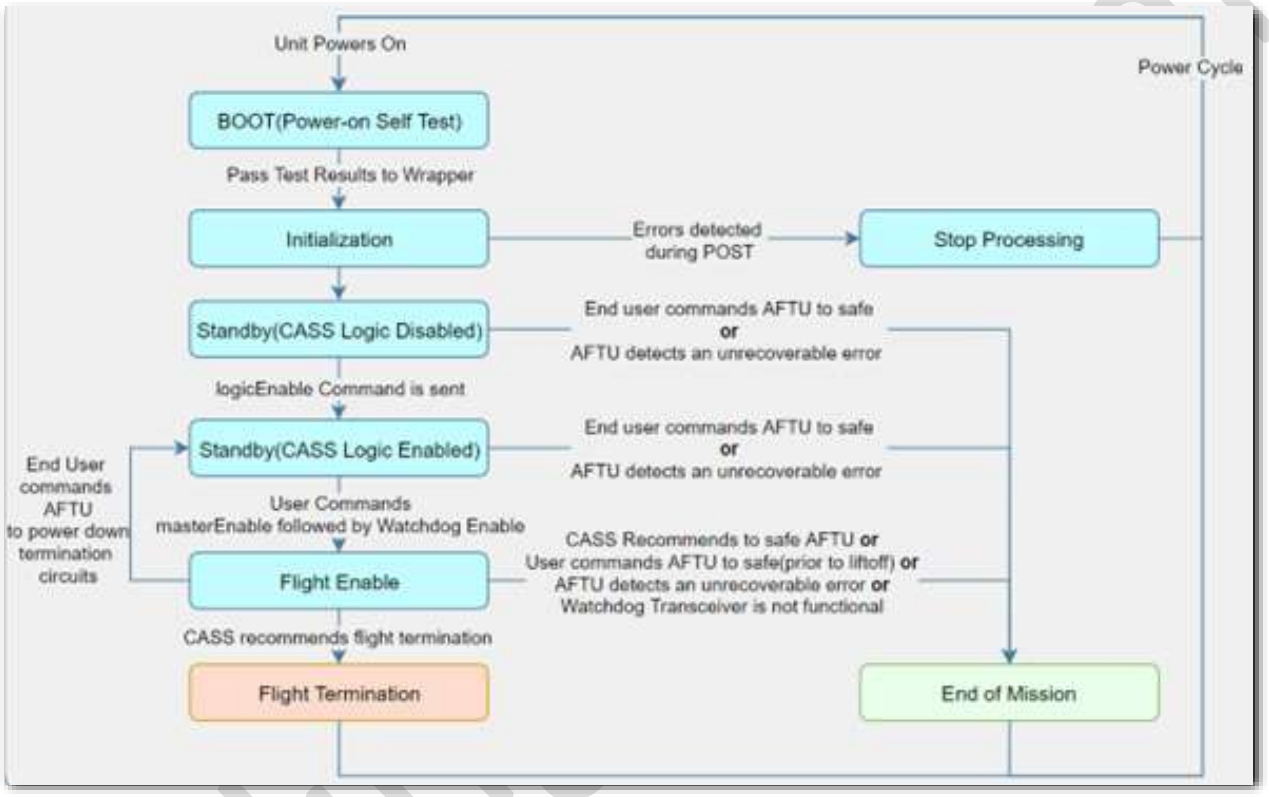


Figure 19: Wrapper Processing State

Each of the following subsections describes each state and its entry and exit conditions:

7.5.1.1 INITIALIZATION STATE

When the **Wrapper** is started by calling the **Wrapper** function **startAfss**, it begins with the **INITIALIZATION** state. In this state, it configures hardware and software for operational use. This is the only state that allows configuration changes. Recommendations from the **CASS Rule Engine** are not processed in this state. After the completion of the initialization process, it will change the state to **STANDBY**. If any issues are found during the initialization process, it will change the state to **STOP PROCESSING**.

ENTRY CONDITIONS:

- The boot-up process is completed.
- Console command **startAfss** is issued.

EXIT CONDITIONS:

- Transition to **STANDBY** (CASS disabled):
  - No issue was found.
  - System time is synchronized to a navigation sensor.
- Transition to **STOP PROCESSING**:
  - An issue is found.

7.5.1.2 STOP PROCESSING STATE

The **Wrapper** transitions to the **STOP PROCESSING** state from the **INITIALIZATION** state if the **INITIALIZATION** state encounters an error. The **Wrapper** provides details of the issue via the console since the downlink telemetry may not be initialized. At this point, the **Wrapper** is no longer running, and a power cycle is required to change the state. Recommendations from the **CASS Rule Engine** are not processed in this state.

ENTRY CONDITIONS:

- Issue found during initialization.

EXIT CONDITIONS:

- Power cycle.

7.5.1.3 STANDBY (CASS DISABLED) STATE

The **Wrapper** transitions to the **STANDBY** (CASS disabled) state from the **INITIALIZATION** state when it meets the below entry conditions. In this state, the **Wrapper** has acquired valid sensor data and processes sensor data. The **CASS Rule Engine** is not executing any rules. The **Wrapper** listens to, processes user commands from the console, and uplinks telemetry. The **Wrapper** will reject the **Watchdog Enable Command** and send an error reply. The **Wrapper** will reject the **Master Enable Command** and send an error reply.

ENTRY CONDITIONS:

- System time is synchronized to a navigation sensor.
- The **CASS Rule Engine** is in the **READY FOR RULE PROCESSING** state.
- **Termination** circuits are disabled.
- **AFTU** is healthy.

EXIT CONDITIONS:

- Transition to **STANDBY** (CASS enabled):
  - The user issues the command (0x11) to enable the **CASS** flight rule processing.
- Transition to **END OF MISSION**:
  - The user issues the command (0x88) to put the **AFTU** in safe mode.

OR

  - A **SEVERE** or **FATAL** issue occurs.

7.5.1.4 STANDBY (CASS ENABLED) STATE

The **Wrapper** transitions to the **STANDBY** (CASS enabled) state from the **STANDBY** (CASS disabled) or **FLIGHT ENABLED** state when it meets the below-mentioned entry conditions. The **Wrapper** passes the processed sensor data to the **CASS Rule Engine** in this state. The **CASS Rule Engine** executes rules. Liftoff detections are not processed in this state. Recommendations from the **CASS Rule Engine** are not processed in this state. The **Wrapper** listens to, processes user commands from the console, and uplinks telemetry. The **Wrapper** will reject the **Watchdog** Enable Command and send an error reply.

ENTRY CONDITIONS:

- The **CASS Rule Engine** is in the **RULE PROCESSING** state.
- **Termination** circuits are disabled.
- **AFTU** is healthy.

EXIT CONDITIONS:

- Transition to **FLIGHT ENABLED**:
  - The user issues a command (0x22 or 0x33) to enable the **termination** circuit.
- Transition to **END OF MISSION**:
  - The user issues a command (0x88) to put the **NAFTU** in safe mode.

OR

  - The **Watchdog** transceiver does not have the correct voltage value of 18V after the **Watchdog** circuit is powered.

OR

  - A SEVERE or FATAL issue occurs.

7.5.1.5 FLIGHT ENABLED STATE

The **Wrapper** transitions to the FLIGHT ENABLED state from the STANDBY (CASS enabled) state when it meets the entry conditions listed below. In this state, the **termination** circuits are enabled and powered. This is the only processing state which performs the safety function. This state must be entered before the first motion of the vehicle. Liftoff detections are processed in this state. Recommendations from the **CASS Rule Engine** are processed in this state. The **Wrapper** will only act on a destruct recommendation, not an arm recommendation. **Arm** recommendations are for when a vehicle is close to the flight corridor's edge but has not crossed the boundary. The **AFTU** does not have hardware that would make use of this recommendation. The **Wrapper** listens to, and processes user commands from the uplink telemetry.

ENTRY CONDITIONS:

- The **CASS Rule Engine** is in the RULE PROCESSING state.
- **CASS Rule Engine** does not recommend destruct.
- The **termination** circuit is enabled.
- **nTERM 1** and **2** outputs in the **termination** circuit are set to **ON**.
- **TERM 1** and **2** outputs in the **termination** circuit are set to **OFF**.
- **AFTU** is healthy.

EXIT CONDITIONS:

- Transition to **STANDBY** (CASS enabled):
  - The user issues a command (0x44 or 0x55) to disable the termination circuit.
- Transition to FLIGHT TERMINATION:
  - **CASS Rule Engine** recommends destruct.
- Transition to **END OF MISSION**:
  - **CASS Rule Engine** recommends safe.

OR

  - The user issues a **command** (0x88) to put the **AFTU** in safe mode.

OR

  - A **SEVERE** or **FATAL** issue occurs.

7.5.1.6 FLIGHT TERMINATION STATE

The **Wrapper** transitions to the **FLIGHT TERMINATION** state from the **FLIGHT ENABLED** state when it meets the below entry conditions. In this state, the powered flight must end. This terminal state repeatedly sends commands to end a powered flight. Some example flight configurations include firing ordinances or removing power from valves.

ENTRY CONDITIONS:

- The **CASS Rule Engine** is in the **RULE PROCESSING** state.
- **CASS Rule Engine** recommends destruct.
- The **termination** circuit is enabled.
- **nTERM 1** and **2** outputs in the **termination** circuit are set to **OFF**.
- **TERM 1** and **2** outputs in the **termination** circuit are set to **ON**.

EXIT CONDITIONS:

- Power cycle.

Upon transition from the **FLIGHT ENABLED** state to the **FLIGHT TERMINATION** state, the order of changing the **termination** circuit state and sending telemetry messages is swapped only for

the first time but back to normal order afterward. As described in Figure 20: Termination Circuit State and Telemetry Messages in the Flight Termination State, the flow in terms of **termination** circuit state and telemetry messages is as follows:

1. During the **FLIGHT ENABLED** state, the **CASS** recommends destruct.
2. The **Wrapper** changes the state from **FLIGHT ENABLED** to FLIGHT TERMINATION.
3. The **Wrapper** generates and sends telemetry messages (#16, 17, 50).
4. The **Wrapper** waits for all telemetry messages to be sent but only up to **0.05** seconds. There may be more messages in the queue, especially the navigation solution messages (#20, 21, 22).
5. The **Wrapper** changes the **termination** circuit state by turning on the **TERM1/2** and then turning off the **nTERM1/2** outputs.
6. The **Wrapper** waits for the next cycle, approximately **0.1** seconds.
7. The **Wrapper** re-applies the **termination** circuit state by turning off the **nTERM1/2** outputs and turning on the **TERM1/2**.
8. The **Wrapper** generates and sends telemetry messages (#16, 17, 50).
9. The **Wrapper** repeats steps #7 and 8.

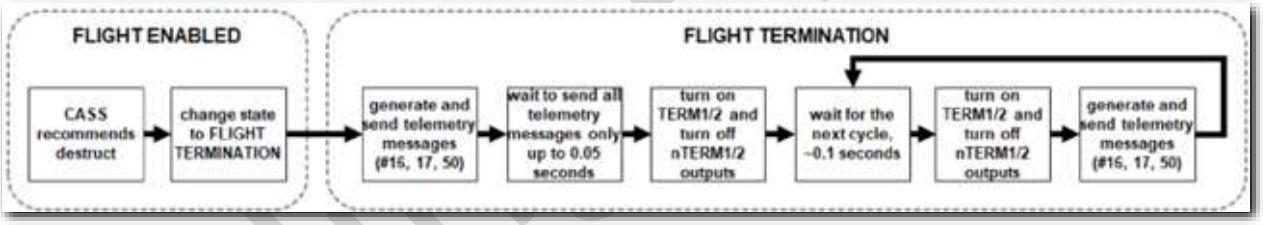


Figure 20: Termination Circuit State and Telemetry Messages in the Flight Termination State

NOTE:

The maximum wait time of 0.05 seconds is mostly arbitrary but based on:is mostly arbitrary but based on:

1. One AFTU status message (42 bytes),
2. One (up to 256 bytes),
3. Two navigation solution messages from each two sensors (42 bytes each, 168 bytes total),
4. and one analog housekeeping message (87 bytes).

That is **553** bytes total = **4424 bits** = **4424 bits** / **115200** bits per second baud rate = **~0.04** seconds. The purpose of the delay is to allow the **termination** recommendation to be sent down the telemetry stream and off the vehicle for after-action reporting. The end user must account for the **0.04**-second delay when considering its actions following the output of the **AFTU**.

7.5.1.7 END OF MISSION STATE

The **Wrapper** transitions to the **END OF MISSION** state from **STANDBY** (CASS disabled), **STANDBY** (CASS enabled), or **FLIGHT ENABLED** state when it meets the below-mentioned entry conditions. This terminal state is entered when the **NAFTU** function is no longer required. Recommendations from the **CASS Rule Engine** are not processed in this state. The **CASS Rule Engine** is disabled in this state. Whenever a critical error is found during the FLIGHT ENABLED state, the **Wrapper** will stop updating the **Watchdog** reset.

ENTRY CONDITIONS:

- **CASS Rule Engine** recommends safe.

OR

- The user issues a command (0x88) to put the **NAFTU** in safe mode.

OR

- An issue is found during **STANDBY** or **FLIGHT ENABLED**.

EXIT CONDITIONS:

- Power cycle.

7.5.2 LOOP

After the **Wrapper** completes its initialization process, but before waiting for time synchronization, it runs in the main loop for as long as the program runs and until the **NAFTU** is power cycled. The **Wrapper** states that the main loop runs in include **INITIALIZATION** (partial due to not including POST), **STANDBY**, **FLIGHT ENABLED**, **FLIGHT TERMINATION**, and **END OF MISSION**.

The main loop contains several activities designated to run in a specific cycle slot. Each cycle runs at approximately **0.0100**-second intervals. The interval is approximate because it depends on how long an activity will take. If an activity takes longer than **0.010** seconds in a cycle, that cycle will run slightly longer than **0.010** seconds. But it will not skip an activity on the next cycle because the cycle number is counter-based and will always be incremented by one.

The following tables list the activities and designated cycle slots:

Every one cycle (approximately every 0.010 seconds):

ACTIVITY	CYCLE SLOT	EXAMPLE
Reset the Watchdog timer if healthy or change the state if unhealthy.	0	6000, 6001, 6002, …

Table 30: Activities in the 1-Cycle

Should a blocking or starvation occur in other activities in the main loop, it will prevent the **Wrapper** from resetting the **Watchdog** timer in a timely manner. Hence, the **AFTU** will be considered unhealthy, and the **Watchdog** will act appropriately.

Every **ten** cycles (approximately every 0.100 seconds):

ACTIVITY	CYCLE SLOT	EXAMPLE
Process the user's command, if any.	1	6001, 6011, 6021, …
Poll analog measurements from TI ADCs.	2	6002, 6012, 6022, …
Change state between STANDBY (CASS enabled) and FLIGHT ENABLED if needed.	3	6003, 6013, 6023, …
Run CASS update. Process CASS recommendation. Generate and send status, CASS, and analog housekeeping telemetry messages.	4	6004, 6014, 6024, …
Unused.	other slots	Unused

Table 31: Activities in the 10-Cycle

Every **100** cycles (approximately every 1.000 seconds):

ACTIVITY	CYCLE SLOT	EXAMPLE
Check whether we need to change the state from INITIALIZATION to STANDBY (CASS disabled) upon time synchronization.	7	6007, 6107, 6207, …

Run the built-in test.	8	6008, 6108, 6208, …
Unused.	other slots	Unused.

Table 32: Activities in the 100-Cycle

Every **200** cycles (approximately every 2.000 seconds):

ACTIVITY	CYCLE SLOT	EXAMPLE
Automatically unlatch termination enables and disables if needed.	5	6005, 6205, 6405, …
Unused	other slots	Unused

Table 33: Activities in the 100-Cycle

Every **3000** cycles (approximately every 30.000 seconds):

ACTIVITY	CYCLE SLOT	EXAMPLE
Report AFTU/CASS/FTIB statuses.	9	6009, 9009, 12009, …
Report navigation sensor stats.	209	6209, 9209, 12209, …
Report telemetry stats.	409	6409, 9409, 12409, …
Report user command stats.	609	6409, 9609, 12609, …
Report TI ADC stats.	809	6809, 9809, 12809, …
Report memory stats.	1009	7009, 10009, 13009, …
Report voltages.	1209	7209, 10209, 13209, …
Report CPU stats and internal temperatures.	1409	7409, 10409, 13409, …
Report time sync stats.	1609	7609, 10609, 13609, …
Report log stats.	1809	7809, 10809, 13809, …
Unused.	other slots	Unused.

Table 34: Activities in the 3000-Cycle

EXAMPLE:

CYCLE	ACTIVITIES
6000	Reset the Watchdog timer.
6001	Reset the Watchdog timer. Process the user's command.

CYCLE	ACTIVITIES
6606	Reset the Watchdog timer.
6607	Reset the Watchdog timer. Change state due to time sync.

CYCLE	ACTIVITIES
6002	Reset the Watchdog timer. Poll analog measurements.
6003	Reset the Watchdog timer. Change state due to termination state.
6004	Reset the Watchdog timer. Run CASS update. Process CASS recommendation. Send telemetry messages.
6005	Reset the Watchdog timer. Unlatch circuit commands.
6006	Reset the Watchdog timer.
6007	Reset the Watchdog timer. Change state due to time sync.
6008	Reset the Watchdog timer. Run the built-in test.
6009	Reset the Watchdog timer. Report AFTU/CASS/FTIB statuses.
6010	Reset the Watchdog timer.
6011	Reset the Watchdog timer. Process the user's command.
6012	Reset the Watchdog timer. Poll analog measurements.
6013	Reset the Watchdog timer. Change state due to termination state.
6014	Reset the Watchdog timer. Run CASS update. Process CASS recommendation.

CYCLE	ACTIVITIES
6608	Reset the Watchdog timer. Run the built-in test.
6609	Reset the Watchdog timer. Report command stats.
6800	Reset the Watchdog timer.
6801	Reset the Watchdog timer. Process the user's command.
6605	Reset the Watchdog timer. Unlatch circuit commands.
6802	Reset the Watchdog timer. Poll analog measurements.
6803	Reset the Watchdog timer. Change state due to termination state.
6804	Reset the Watchdog timer. Run CASS update. Process CASS recommendation. Send telemetry messages.
6805	Reset the Watchdog timer. Unlatch circuit commands.
6806	Reset the Watchdog timer.
6807	Reset the Watchdog timer. Change state due to time sync.
6808	Reset the Watchdog timer. Run the built-in test.
6809	Reset the Watchdog timer. Report memory stats.

CYCLE	ACTIVITIES
	Send telemetry messages.
6015	Reset the Watchdog timer.
6016	Reset the Watchdog timer.
6017	Reset the Watchdog timer.
6018	Reset the Watchdog timer.
6019	Reset the Watchdog timer.
6100	Reset the Watchdog timer.
6101	Reset the Watchdog timer. Process the user's command.
6102	Reset the Watchdog timer. Poll analog measurements.
6103	Reset the Watchdog timer. Change state due to termination state.
6104	Reset the Watchdog timer. Run CASS update. Process CASS recommendation. Send telemetry messages.
6105	Reset the Watchdog timer.
6106	Reset the Watchdog timer.
6107	Reset the Watchdog timer. Change state due to time sync.
6108	Reset the Watchdog timer. Run the built-in test.
6109	Reset the Watchdog timer.

CYCLE	ACTIVITIES
7000	Reset the Watchdog timer.
7001	Reset the Watchdog timer. Process the user's command.
7002	Reset the Watchdog timer. Poll analog measurements.
7003	Reset the Watchdog timer. Change state due to termination state.
7004	Reset the Watchdog timer. Run CASS update. Process CASS recommendation. Send telemetry messages.
7005	Reset the Watchdog timer. Unlatch circuit commands.
7006	Reset the Watchdog timer.
7007	Reset the Watchdog timer. Change state due to time sync.
7008	Reset the Watchdog timer. Run the built-in test.
7009	Reset the Watchdog timer. Report voltages.
7200	Reset the Watchdog timer.
7201	Reset the Watchdog timer. Process the user's command.
7202	Reset the Watchdog timer. Poll analog measurements.
7203	Reset the Watchdog timer.

CYCLE	ACTIVITIES
6200	Reset the Watchdog timer.
6201	Reset the Watchdog timer. Process the user's command.
6202	Reset the Watchdog timer. Poll analog measurements.
6203	Reset the Watchdog timer. Change state due to termination state.
6204	Reset the Watchdog timer. Run CASS update. Process CASS recommendation. Send telemetry messages.
6205	Reset the Watchdog timer. Unlatch circuit commands
6206	Reset the Watchdog timer.
6207	Reset the Watchdog timer. Change state due to time sync.
6208	Reset the Watchdog timer. Run the built-in test.
6209	Reset the Watchdog timer. Report navigation sensor stats.
6400	Reset the Watchdog timer.
6401	Reset the Watchdog timer.

CYCLE	ACTIVITIES
	Change state due to termination state.
7204	Reset the Watchdog timer. Run CASS update. Process CASS recommendation. Send telemetry messages.
7205	Reset the Watchdog timer. Unlatch circuit commands.
7206	Reset the Watchdog timer.
7207	Reset the Watchdog timer. Change state due to time sync.
7208	Reset the Watchdog timer. Run the built-in test.
7209	Reset the Watchdog timer. Report CPU stats and temperatures.
7400	Reset the Watchdog timer.
7401	Reset the Watchdog timer. Process the user's command.
7402	Reset the Watchdog timer. Poll analog measurements.
7403	Reset the Watchdog timer. Change state due to termination state.
7404	Reset the Watchdog timer. Run CASS update. Process CASS recommendation. Send telemetry messages.
7405	Reset the Watchdog timer. Unlatch circuit commands.
7406	Reset the Watchdog timer.

CYCLE	ACTIVITIES
	Process the user's command.
6402	Reset the Watchdog timer. Poll analog measurements.
6403	Reset the Watchdog timer. Change state due to termination state.
6404	Reset the Watchdog timer. Run CASS update. Process CASS recommendation. Send telemetry messages.
6405	Reset the Watchdog timer. Unlatch circuit commands
6406	Reset the Watchdog timer.
6407	Reset the Watchdog timer. Change state due to time sync.
6408	Reset the Watchdog timer. Run the built-in test.
6409	Reset the Watchdog timer. Report telemetry stats.
6600	Reset the Watchdog timer.
6601	Reset the Watchdog timer. Process the user's command.
6602	Reset the Watchdog timer. Poll analog measurements.
6603	Reset the Watchdog timer. Change state due to termination state.
6604	Reset the Watchdog timer. Run CASS update. Process CASS recommendation.

CYCLE	ACTIVITIES
7407	Reset the Watchdog timer. Change state due to time sync.
7408	Reset the Watchdog timer. Run the built-in test.
7409	Reset the Watchdog timer. Report time stats.
7600	Reset the Watchdog timer.
7601	Reset the Watchdog timer. Process the user's command.
7602	Reset the Watchdog timer. Poll analog measurements.
7603	Reset the Watchdog timer. Change state due to termination state.
7604	Reset the Watchdog timer. Run CASS update. Process CASS recommendation. Send telemetry messages.
7605	Reset the Watchdog timer. Unlatch circuit commands.
7606	Reset the Watchdog timer.
7607	Reset the Watchdog timer. Change state due to time sync.
7608	Reset the Watchdog timer. Run the built-in test.
7609	Reset the Watchdog timer. Report log stats.

CYCLE	ACTIVITIES
	Send telemetry messages.

CYCLE	ACTIVITIES

Table 35: Example of Cycle Activities

7.5.3 NAVIGATION SENSOR DATA PROCESSING

This section describes how the **Wrapper** processes data from a navigation sensor and populates the data parameters for the **CASS Rule Engine** for rule processing.

7.5.3.1 JAVAD SENSOR MESSAGE DATA PROCESSING

The **Wrapper** is expecting to receive at least [**rE**] **Reference Epoch** and [**rV**] **Receiver's Position** and **Velocity** messages when the **Wrapper** has been configured to use **JAVAD GNSS**; see the SG901-0013-AFTU User Manual, section 8.0 Navigation Sensors for format references and details. These types of messages are typically to be generated and provided by a **JAVAD GNSS GPS** receiver.

When the [**rE**] message is received, the following data fields are extracted and used:

- **Sample** number – This is used to pair the time information between the [**rE**] and [**rV**] messages.
- **Time scale ID** – This is used to check whether the time information in this message is **GPS** time. This message won't be processed further if the time is not **GPS** time.
- **GPS** week number – This is used to get the current **GPS** week number.
- **GPS** milliseconds – This is used to get the current **GPS** milliseconds.

After extracting the above data fields, the **GPS** week and milliseconds are adjusted to include the **GPS** week offset. The adjusted **GPS** time and sample number are stored as data variables for use the next time the [**rV**] message is received.

When the [**rV**] message is received, the following data fields are extracted and used:

- Sample number – This is used to pair the time information between the [**rE**] and [**rV**] messages. If the sample numbers are not matched, then the time information in this message will not be processed, and the **gps_health.is_valid_GPS_time CASS** parameter is set to **FALSE**.
- Time delta in **5** milliseconds – This is used to add to the **GPS** time from the [**rE**] message to account for the time difference between the time when the [**rE**] and [**rV**] messages are received.

After this, the **GPS** time from the [**rE**] message with the time delta is set to the **measurement_time CASS** parameter. **gps_health.is_valid_GPS_time CASS** parameter is set to **TRUE**.

- Number of satellites – This is used to populate the **gps_health.n_satellites CASS** parameter.
- Position validity – This flag assists in determining whether the **GPS** data is valid.
- Velocity validity – This flag assists in determining whether the **GPS** data is valid.
- PDOP – This is used to populate the **gps_health**. **PDOP CASS** parameter.
- Position coordinate system – This checks whether the position and velocity information is in the **ECEF** coordinate system. If the coordinate system is not **ECEF**, then the position and velocity information will not be processed, and the **gps_health.is_valid_GPS_data CASS** is set to **FALSE**. Otherwise, if both position and velocity validities are **TRUE**, then the **gps_health.is_valid_GPS_data CASS** parameter is set to **TRUE**.
- Position X component – This is used to populate the **pos_ECEF_X CASS** parameter.
- Position Y component – This is used to populate the **pos_ECEF_Y CASS** parameter.
- Position Z component – This is used to populate the **pos_ECEF_Z CASS** parameter.
- Velocity X component – This is used to populate the **vel_ECEF_X CASS** parameter.
- Velocity Y component – This is used to populate the **vel_ECEF_Y CASS** parameter.
- Velocity Z component – This is used to populate the **vel_ECEF_Z CASS** parameter.

After processing the above data fields, if both **gps_health.is_valid_GPS_data** and **gps_health.is_valid_GPS_time CASS** parameters are true; then the **is_valid_nav_data CASS** parameter is set to **TRUE**.

The s**rc_ID CASS** parameter is set with the navigation number in the sensor index order defined in the mission configuration.

After the [**rV**] message is processed, the **CASS** parameters are ready for use by the **CASS Rule Engine**.

The following lists all parameters found in the **CASS** input data structure and describes whether each parameter is populated:

CASS INPUT PARAMETER	JAVAD JNSS MESSAGE ITEM
src_ID	Navigation number. 0, 1, 2, etc.
measurement_time	GPS time.
pos_ECEF_X	Position X.

CASS INPUT PARAMETER	JAVAD JNSS MESSAGE ITEM
pos_ECEF_Y	Position Y.
pos_ECEF_Z	Position Z.
vel_ECEF_X	Velocity X.
vel_ECEF_Y	Velocity Y.
vel_ECEF_Z	Velocity Z.
heading	Not populated. Set to floating point 0.0.
pitch	Not populated. Set to floating point 0.0.
roll	Not populated. Set to floating point 0.0.
omega_IMU_X	Not populated. Set to floating point 0.0.
omega_IMU_Y	Not populated. Set to floating point 0.0.
omega_IMU_Z	Not populated. Set to floating point 0.0.
accel_IMU_X	Not populated. Set to floating point 0.0.
accel_IMU_Y	Not populated. Set to floating point 0.0.
accel_IMU_Z	Not populated. Set to floating point 0.0.
gps_health.n_satellites	Satellite count.
gps_health.PDOP	PDOP.
gps_health.HDOP	Not populated. Set to floating point 0.0.
gps_health.VDOP	Not populated. Set to floating point 0.0.
gps_health.TDOP	Not populated. Set to floating point 0.0.
gps_health.clock_drift	Not populated. Set to floating point 0.0.
gps_health.is_valid_GPS_time	If GPS time is valid.
gps_health.is_valid_GPS_data	If position and velocity are valid.
ins_health.mode	Not populated. Set to integer 0.
ins_health.accel_bias	Not populated. Set to floating point 0.0.
ins_health.rate_bias	Not populated. Set to floating point 0.0.
ins_health.accel_SF	Not populated. Set to floating point 0.0.

CASS INPUT PARAMETER	JAVAD JNSS MESSAGE ITEM
ins_health.rate_SF	Not populated. Set to floating point 0.0.
ins_health.pos_var	Not populated. Set to floating point 0.0.
ins_health.vel_var	Not populated. Set to floating point 0.0.
ins_health.att_var	Not populated. Set to floating point 0.0.
ins_health.is_valid_FOM	Not populated. Set to Boolean false.
is_valid_nav_data	If gps_health.is_valid_GPS_time is true and gps_health.is_valid_GPS_data is true and gps_health.PDOP is not zero.
is_valid_IMU_data	Not populated. Set to Boolean false.

Table 36: CASS Input Data Structure

The data values populated in the **CASS** input data structure are also inserted in the telemetry message for sending through the downlink telemetry stream.

7.5.3.2 NOVATEL SENSOR MESSAGE DATA PROCESSING

The **Wrapper** is expected to receive at least **BESTXYZ** and **PSRDOP2** when the **Wrapper** has been configured to use **NovAtel** sensor messages.

When **BESTXYZ** is received, the following data fields are extracted. **BESTXYZ** can only send position and velocity data that is **ECEF** coordinates.

- **P-Sol Status** - This flag assists in determining whether the **GPS** data is valid.
- **V-Sol Status** - This flag assists in determining whether the **GPS** data is valid.
- **P-X** – This is used to populate the **pos_ECEF_X CASS** parameter.
- **P-Y** – This is used to populate the **pos_ECEF_Y CASS** parameter.
- **P-Z** – This is used to populate the **pos_ECEF_Z CASS** parameter.
- **V-X** – This is used to populate the **vel_ECEF_X CASS** parameter.
- **V-Y** – This is used to populate the **vel_ECEF_Y CASS** parameter.
- **V-Z** – This is used to populate the **vel_ECEF_Z CASS** parameter
- **#SVs** – This is used to populate the **gps_health.n_satellites CASS** parameter.
- **BESTXYZ Header** – Contains **GPS** reference week number, seconds from the beginning of the reference week, accurate to milliseconds, and time status used for the time validity.

When **PSRDOP2** is received, the following data fields are extracted.

- **PDOP** – This is used to populate the **gps_health**. **PDOP CASS** parameter.

CASS INPUT PARAMETER	NOVATEL MESSAGE ITEM
src_ID	Navigation number. 0, 1, 2, etc.
measurement_time	GPS time.
pos_ECEF_X	P-X.
pos_ECEF_Y	P-Y.
pos_ECEF_Z	P-Z.
vel_ECEF_X	V-X.
vel_ECEF_Y	V-Y.
vel_ECEF_Z	V-Z.
heading	Not populated. Set to floating point 0.0.
pitch	Not populated. Set to floating point 0.0.
roll	Not populated. Set to floating point 0.0.
omega_IMU_X	Not populated. Set to floating point 0.0.
omega_IMU_Y	Not populated. Set to floating point 0.0.
omega_IMU_Z	Not populated. Set to floating point 0.0.
accel_IMU_X	Not populated. Set to floating point 0.0.
accel_IMU_Y	Not populated. Set to floating point 0.0.
accel_IMU_Z	Not populated. Set to floating point 0.0.
gps_health.n_satellites	#SV
gps_health.PDOP	PDOP.
gps_health.HDOP	Not populated. Set to floating point 0.0.
gps_health.VDOP	Not populated. Set to floating point 0.0.
gps_health.TDOP	Not populated. Set to floating point 0.0.
gps_health.clock_drift	Not populated. Set to floating point 0.0.
gps_health.is_valid_GPS_time	If GPS time is valid.
gps_health.is_valid_GPS_data	If position and velocity are valid.
ins_health.mode	Not populated. Set to integer 0.
ins_health.accel_bias	Not populated. Set to floating point 0.0.
ins_health.rate_bias	Not populated. Set to floating point 0.0.
ins_health.accel_SF	Not populated. Set to floating point 0.0.

CASS INPUT PARAMETER	NOVATEL MESSAGE ITEM
ins_health.rate_SF	Not populated. Set to floating point 0.0.
ins_health.pos_var	Not populated. Set to floating point 0.0.
ins_health.vel_var	Not populated. Set to floating point 0.0.
ins_health.att_var	Not populated. Set to floating point 0.0.
ins_health.is_valid_FOM	Not populated. Set to Boolean false.
is_valid_nav_data	If gps_health.is_valid_GPS_time is true and gps_health.is_valid_GPS_data is true and gps_health.PDOP is not zero.
is_valid_IMU_data	Not populated. Set to Boolean false.

Table 37: NovAtel Sensor Messages

7.5.3.3 TRACKING DATA INGESTION

It is important to note that the **AFTU** does not differentiate between simulated **GPS** tracking, recorded **GPS** playback, or live **GPS** messages. Simulated **GPS** tracking means that a real **GPS** sensor is being used along with a **GPS** simulator for **SIL** or **E2E** testing. Recorded **GPS** playback is a capture file of output messages from a compatible **GPS** – either a **Javad** or **NovAtel** receiver – and played into an **AFTU** for use in ATP testing. Live **GPS** message means an **AFTU** receiving **GPS** sensor data from a sensor using a real-time source (open sky).

An end user will select from one of four stored configurations in the **Wrapper** to select what type of **GPS** sensor data ingestion it will use for flight. The supported configurations are serial ingestion of **Javad**, serial ingestion of **NovAtel**, **Ethernet** ingestion of **Javad**, or **Ethernet** ingestion of **NovAtel**. This flexibility will allow an end user to have a different configuration for E2E testing and flight to ensure that simulated or recorded **GPS** playback is not used for flight. The user manual will cover how an end user must configure the unit to ensure proper safety use. For system testing of the **AFTU**, Sagrad shall perform a test using simulated trajectory data from recorded **GPS** messages.

7.5.4 WRAPPER CONFIGURATION

The **Wrapper** can be configured through boot arguments read in from the **VxWorks Boot Line**. More information regarding the **VxWorks Boot Line** can be found in the SG901-0013-AFTU User ManualRev 1.2. The **Wrapper** will receive from the **VxWorks Boot Line** four variables:

- AFSS ID
- Mission Data Load (MDL) Memory Location
- GPS Config variable
- Telemetry Config variable

The **AFSS ID** is used to set the id of the **AFTU** inside the **Wrapper** code. The **MDL Memory Location** is used to designate the memory location in which the **Wrapper** is to read in the **MDL** necessary for **CASS**.

The **GPS Config** variable is a number between **0** to **3**, inclusively, that determines the **GPS** being used and the connection to it as well. Each value represents a combination between the **GPS Javad** or **NovAtel**, with the connection being **Serial** or **Ethernet**. The **Wrapper** reads in the **GPS Config** variable and creates a **Navigation Task** corresponding to its representation. The following values represent the **GPS** combinations:

0 – Javad with Serial

1 – Javad with Ethernet

2 – Novatel with Serial

3 – Novatel with Ethernet

The **Telemetry Config** variable is one of two numbers: **0** or **1**. Each value represents the connection the Telemetry stream shall output to. When creating the **Telemetry Manager** object, the **Wrapper** will establish a **Serial** or **Ethernet** Telemetry connection. The following values represent the defined **Telemetry** connection type:

0 – Ethernet Telemetry output

1 – Serial Telemetry output

7.5.5 USER COMMANDS

The user commands allow the user to change the state of the **AFTU** and enable **CASS** rule processing. The commands can be issued via the uplink telemetry message; see the SG901-0013-AFTU User Manual for details.

The flow of a user command through the **AFTU** is as follows:

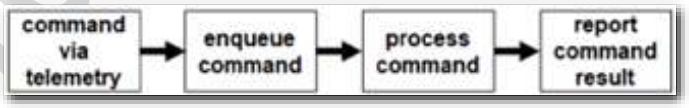


Figure 21: User Command Logic Flow

1. The user sends a command via the uplink telemetry message.
    a. The command sent via the uplink telemetry will be passed through only if the **processor ID** found in the telemetry message header matches what is entered in the bootargs. For example, if the **ID** in the header is **0** and the processor **ID** entered as an argument in the

startAfss is also **0**, then that command is accepted. If rejected, the message is noted in the console and ignored.

2. The new command is enqueued.
   a. If the queue, which can hold only one command at a time, is empty, the new command is enqueued.
   b. But if the queue is not empty, an error is reported via the downlink telemetry and console output for the command sent via the uplink telemetry.
3. Whenever the main loop gets into the designated time slot where it can process the user command, the command in the queue is processed.
   a. The command will not be processed if the liftoff has occurred. The **CASS Rule Engine** uses the '**haveLiftOff**' **CASS** database variable to determine the liftoff status.
   b. The command will not be processed if the command is not valid syntactically or the command is not recognized.
   c. Unprocessed commands will be cleared from the queue.
4. The result of the command is reported with either the message **Processed command** [num] or **Fail to process command** [num].

7.5.6 CASS-BASED TERMINATION DECISION

The Wrapper uses the **CASS Rule Engine** for termination decisions.

Whenever the **CASS Rule Engine** recommends destruct, the Wrapper will execute the following sequence to turn the **nTERM** outputs **OFF** and the **TERM** outputs **OFF**:

1. Open the nTERM 1 High switch, which immediately stops the current to that output.
2. 1 millisecond later.
3. Open the nTERM 2 High switch, which immediately stops the current to that output.
4. 1 millisecond later.
5. Close the TERM 1 Low switch, which arms the TERM 1 output (no current flows yet).
6. 1 millisecond later.
7. Close the TERM 2 Low switch, which arms the TERM 2 output (no current flows yet).
8. 1 millisecond later.

9. Close the TERM 1 High switch, enabling current for TERM 1 output.
10. 1 millisecond later.
11. Close the TERM 2 High switch, enabling current for TERM 2 output.
12. 1 millisecond later.
13. Open the redundant nTERM 1 Low switch, which will only stop the current if the High switch has failed.
14. 1 millisecond later.
15. Open the redundant nTERM 2 Low switch, which will only stop the current if the High switch has failed.

Whenever the **CASS Rule Engine** recommends safe, the **Wrapper** will execute the following sequence to disable the **termination** circuits:

1. Turn off the software latch of the enabled signal to the **termination circuit A**.
2. Turn off the software latch of the enabled signal to the **termination circuit B**.
3. Disable the **termination circuit A**.
4. Disable the **termination circuit B**.

7.5.7 SYSTEM TIME

Wrapper maintains and uses the **System Time** as a common time basis. There are three parts in the **System Time**: 1) Process **GPS Time** information from **navigation** sensors. 2) Synchronize the **System Time**. 3) Get and use the **System Time**. The first two parts are constantly repeated for as long as the **Wrapper** is running. The third one is used on an as-need-basis (i.e., a caller needs the time information to insert in a report log). The following diagram illustrates the overview flow of the **System Time**.

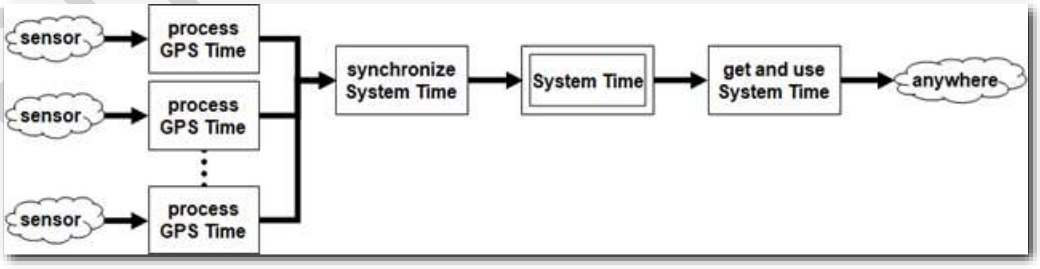


Figure 22: System Dime Synchronization Logic Flow

7.5.7.1 PROCESS GPS TIME

In the first part of the **System Time**, the **Wrapper** uses the **GPS Time** information from a **JAVAD GNSS navigation** sensor to serve as a basis in the **System Time**. Specifically, the **GPS Time** information is extracted from **GPS** week number and time in milliseconds data fields in the **rE** message (see SG901-0013-AFTU User Manual, section 8.0 Navigation Sensors). After extraction, the **GPS Time** is adjusted to include the **GPS** week rollover by adding the 2048 **GPS** week. The adjusted **GPS Time** is inserted into the common data type – **CASS Time** – before synchronizing the **System Time** in the second part of updating the **System Time**.

CASS Time is the data type defined in the **CASS Rule Engine** library. It contains 1) week offset from **6 January 1980** at **00**:**00**:**00** and 2) the number of seconds since the beginning of the current week within a range of [**0**, … ***604800***]. Each week starts on Sunday at **00**:**00**.

While still in the first part of the **System Time**, the **Wrapper** processes and adjusts the **GPS Time** information for each navigation sensor individually. It does not cross-check, cross-correlate, or cross-reference with **GPS Time** information from other **navigation** sensor(s). A validity check, however, is performed individually during this part. The following diagram illustrates the flow in processing **GPS Time** for each sensor.

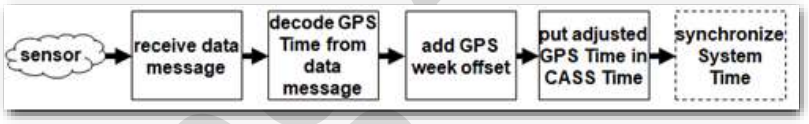


Figure 23: Processing GPS Time Logic Flow

7.5.7.2 SYNCHRONIZE SYSTEM TIME

In the second part of the **System Time**, whenever a **GPS Time**, which is in the form of **CASS Time**, is received and processed, the **System Time**, also in the form of **CASS Time**, is updated and aligned with the **GPS Time** only if one of the following two conditions are met:

1. The absolute difference between the new **GPS Time** and the current **System Time** is between **0.001** seconds (inclusive) and **0.015** seconds (inclusive). The **0.015** seconds are based on 15% of the **CASS** update period, which is 0.100 seconds.
2. Otherwise, the system time is updated if the absolute difference between them is outside the thresholds for five or more consecutive times.

Since there will be several **navigation** sensors, the **System Time** information is locked for exclusive access at the beginning of a synchronous request, and then the lock is released at the end. Whenever

the **System Time** is locked already when another synchronous request is made at that moment, that request will be skipped to avoid delay or incorrectly updating the time. The following diagram illustrates the flow in synchronizing **System Time**.

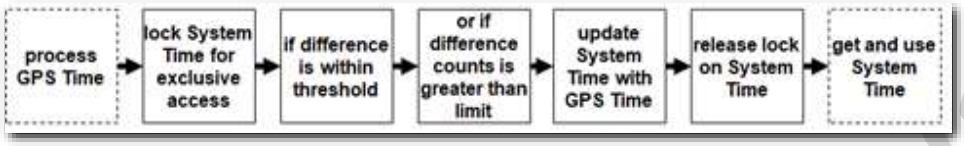


Figure 24: Synchronizing System Time Logic Flow

7.5.7.3 USE SYSTEM TIME

In the third part of the **System Time**, anywhere within the **Wrapper** can get and use the current **System Time**. Before the **System Time** is provided for use, the **Clock Time** is added to the **System Time** to account for the time between the time when the **System Time** is synchronized with the **GPS Time** and the time when a requester asks for the **System Time** information.

The **Clock Time** is the time since the beginning of the application start-up. Like in the second part, the **System Time** is locked before and released after providing the information to a requester. The following diagram illustrates the flow of getting and using the **System Time**.

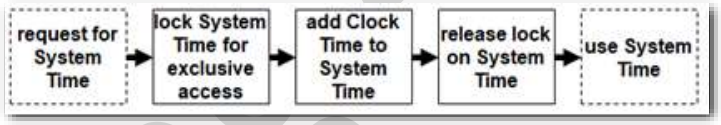


Figure 25: Using System Time Logic Flow

7.5.7.4 EXAMPLE SCENARIO

The following table illustrates an example scenario of how the **GPS Time** (in seconds) is processed, **System Time** (in seconds) is synchronized, and **System Time** is requested. This only shows two navigation sensors, but the same principles apply to more sensors.

EXAMPLE EVENT	SENSOR A GPS TIME	SENSOR B GPS TIME	SYSTEM TIME	SKIPPED SYNC COUNT	REQUESTOR
AFTU is powered up, and Wrapper is initiated.			0.000	0	
About 0.100 seconds later…					

EXAMPLE EVENT	SENSOR A GPS TIME	SENSOR B GPS TIME	SYSTEM TIME	SKIPPED SYNC COUNT	REQUESTOR
Requestor asks for System Time. At this time, the time is the Clock Time since the beginning.			0.100	0	0.100
About 0.100 seconds later…					
Requestor asks for System Time.			0.200	0	0.200
About 0.100 seconds later…					
New GPS Time is received and processed from Sensor A, including GPS week offset. System Time is synchronized to it.	381600.000		381600.000	0	
About 0.100 seconds later…					
Requestor asks for System Time. At this time, the time is the GPS Time plus Click Time since the last synchronous time.			381600.100	0	381600.100
About 0.100 seconds later…					
New GPS Time is received and processed from Sensor A. Synchronized with System Time. The requestor also asks for System Time, but it is locked for synchronizing; hence, it waits.	381600.200		381600.200	0	wait
Synchronizing is completed. The requestor gets System Time.			381600.210	0	381600.210
About 0.090 seconds later…					

EXAMPLE EVENT	SENSOR A GPS TIME	SENSOR B GPS TIME	SYSTEM TIME	SKIPPED SYNC COUNT	REQUESTOR
New GPS Time is received and processed from Sensor B, including GPS week offset. System Time is synchronized to it.		381600.300	381600.300	0	
About 0.100 seconds later…					
New GPS Time is received and processed from Sensor A and B. Time from A got to time synchronizing first; hence, it has the lock. Time from B is ignored.	381600.400	381600.400 (do nothing)	381600.400	0	
About 0.100 seconds later…					
New GPS Time is received and processed from Sensor B. But the time difference is greater than 0.015 seconds; hence, it is skipped.		390000.000		1	
About 0.100 seconds later…					
New GPS Time is received and processed from Sensor A. But the time difference is greater than 0.015 seconds; hence, it is skipped.	381500.000			2	
New GPS Time is received and processed from Sensor B. But the time difference is greater than 0.015 seconds; hence, it is skipped.		390000.100		3	
About 0.050 seconds later…					
Requestor asks for System Time.			381600.650	3	381600.650

EXAMPLE EVENT	SENSOR A GPS TIME	SENSOR B GPS TIME	SYSTEM TIME	SKIPPED SYNC COUNT	REQUESTOR
About 0.050 seconds later…					
New GPS Time is received and processed from Sensor B. But the time difference is greater than 0.015 seconds; hence, it is skipped.		390000.200		4	
About 0.100 seconds later…					
New GPS Time is received and processed from Sensor B. But the time difference is greater than 0.015 seconds; hence, it is skipped.		390000.300		5	
About 0.100 seconds later…					
New GPS Time is received and processed from Sensor B. But the time difference is greater than 0.015 seconds. But if the number of consecutive skipped synchronous is more than the limit, then force System Time to jump.		390000.400	390000.400	0	
About 0.100 seconds later…					
Requestor asks for System Time.			390000.500	0	390000.500
About 0.100 seconds later…					
New GPS Time is received and processed from Sensor A. But the time difference is greater than 0.015 seconds; hence, it is skipped.	381500.800			1	

EXAMPLE EVENT	SENSOR A GPS TIME	SENSOR B GPS TIME	SYSTEM TIME	SKIPPED SYNC COUNT	REQUESTOR
About 0.100 seconds later…					
New GPS Time is received and processed from Sensor B. System Time is synchronized to it.		390000.700	390000.700	0	
About 0.100 seconds later…					
New GPS Time is received and processed from Sensor A. But the time difference is greater than 0.015 seconds; hence, it is skipped.	400100.000			1	
About 0.100 seconds later…					
New GPS Time is received and processed from Sensor B. System Time is synchronized to it.		390000.900	390000.900	0	
About 0.100 seconds later…					
Requestor asks for System Time.			390001.000	0	390001.000

Table 38: Example of Time Synchronization Scenario

7.5.8 HEALTH AND STATUS

The **Wrapper** monitors the health of its process and as well as the **AFTU** for as long as the **Wrapper** is running. How the **Wrapper** checks the health and handles error events varies depending on its state.

When the **Wrapper** is in the **INITIALIZATION** state, the **C++ throw/exception/try/catch** handling handles error events and then transitions to the **STOP PROCESSING** state. The initialization processing starts in the try block in the **startAfss** function and ends in the catch block in the same function. Examples of error events during the initialization processing include but are not limited to failing to load mission configuration, failing to create **C++** objects, failing to initialize interfaces, hardware issues, and hardware not being operational.

As the last action of the **INITIALIZATION** state, the **Wrapper** waits for a time synchronization with the main loop running. When the synchronization occurs, the **Wrapper** transitions from the **INITIALIZATION** state to the **STANDBY** (CASS disabled) state while the main loop continues.

The **Wrapper** main loop generates and sends health and status messages through the console and downlink telemetry for every **Wrapper** state except **STOP PROCESSING**.

When the **Wrapper** is in either the **STANDBY** or **FLIGHT ENABLED** state, it will transition to the **END OF MISSION** state if a critical error is detected. Examples of errors during these states include but are not limited to **C++** objects no longer valid, mismatched hardware/software states, unknown exceptions, and errors in the **CASS Rule Engine**.

Any errors during the **FLIGHT TERMINATION** and **END OF MISSION** states do not change the **Wrapper** state.

The relationship between the types of errors (using the report categories) and the states is shown in the following diagram:

Current Wrapper States →		**Initialization**	**Stop Processing**	**Standby**	**Flight Enabled**	**End of Mission**	**Flight Termination**
CASS Report Categories	**Information**	No change	No change	No change	No change	No change	No change
	Low	No change	No change	No change	No change	No change	No change
	Medium	No change	No change	No change	No change	No change	No change
	Serious	Change to **Stop Processing**	No change	Change to **End of Mission**	No change	No change	No change
	Severe	Change to **Stop Processing**	No change	Change to **End of Mission**	Change to **End of Mission**	No change	No change
Wrapper Report Categories	**Event**	No change	No change	No change	No change	No change	No change
	Initialization	No change	No change	No change	No change	No change	No change
	Error	No change	No change	No change	No change	No change	No change
	Fatal	Change to **Stop Processing**	No change	Change to **End of Mission**	Change to **End of Mission**	No change	No change

Table 39: Relationship Between the Types of Errors and the States

7.5.9 CONTINUOUS BUILT-IN TEST

- In the **Continuous Built-In Test** (CBIT), the **Wrapper** performs the software and hardware health checks every second in all states except the **STOP PROCESSING** state.
- It also does this during the **INITIALIZATION** state while waiting for time synchronization. Whereas a check that involves an **ADC** measurement (e.g., VCC, power to **termination** circuit, power to **Watchdog** circuit, **TERM** outputs, etc.), the range limits are defined in the SG906-0015 NAFTU-Internal-ICD. The **Wrapper** performs the following:
- Check whether the power to the system is good by checking whether the **VCC** is within the operational limits; see section Error: Reference source not found.
- Check whether the power to the **liftoff detection** transceiver is good by checking whether the **VCCLODXCVR** is within the operational limits.
- Check whether the power to the **Watchdog** circuit is good by checking whether the **VCCWD** is within the operational limits.
- Check whether the power to the **Watchdog** transceiver is good by checking whether the **VCCWDXCVR** is within the operational limits.
- Check for any uncorrectable **ECC** errors.
- Ensure that the system clock rate is not changed.
- Ensure that the states of **Wrapper**, **CASS**, and **termination** circuitry are in agreement with each other; see the following table.

WRAPPER	INITIALIZATION	END OF MISSION	STANDBY	FLIGHT ENABLED	FLIGHT TERMINATION	OTHERWISE (2)
CASS Logic	Disabled	Disabled	Any	Enabled	Enabled	otherwise
CASS Arm	Off	Off	Off	Any	On	otherwise
CASS Destruct	Off	Off	Off	Off	On	otherwise
Term Circuit (3)	Disabled	Disabled	Disabled	Enabled	Enabled	otherwise
Term 1 (4)	Off	Off	Off	Off	On (1)	otherwise
Term 2 (5)	Off	Off	Off	Off	On (1)	otherwise
Not Term 1 (6)	Off	Off	Off	On	Off	otherwise

WRAPPER	INITIALIZATION	END OF MISSION	STANDBY	FLIGHT ENABLED	FLIGHT TERMINATION	OTHERWISE (2)
Not Term 2 (7)	Off	Off	Off	On	Off	otherwise
Is Healthy?	Yes	Yes	Yes	Yes	Yes	No

Table 40: AFTU, CASS, and Circuit States.

1. Only at least one of **TERM 1** or **TERM 2** must be **ON** to be considered healthy. If only one is **ON**, generate a warning message.
2. "otherwise" includes an unrecognized state.
3. The condition to determine the state of the **TERM** circuit is based on the operational limits as specified in the section Error: Reference source not found. The **Term Circuit** is associated with the **VCCTERM** measurement.
4. The condition to determine the state of **TERM 1** is based on the operational limits as specified in the section 0. **TERM 1** is associated with **TERM1 High Side Gate**, **TERM1 Low Side Gate**, **TERM1 High Side**, and **TERM1 Low Side** measurements.
5. The condition to determine the state of **TERM 2** is based on the operational limits as specified in the section Error: Reference source not found. **TERM 2** is associated with **TERM2 High Side Gate**, **TERM2 Low Side Gate**, **TERM2 High Side**, and **TERM2 Low Side** measurements.
6. The condition to determine the state of the Not **TERM 1** is based on the operational limits as specified in the section 0Error: Reference source not found. Not **TERM 1** is associated with **nTERM1 High Side Gate**, **nTERM1 Low Side Gate**, **nTERM1 High Side**, and **nTERM1 Low Side** measurements.
7. The condition to determine the state of the Not **TERM 2** is based on the operational limits as specified in the section 0. Not **TERM 2** is associated with **nTERM2 High Side Gate**, **nTERM2 Low Side Gate**, **nTERM2 High Side**, and **nTERM2 Low Side** measurements.

7.5.10 TASKS

The following table lists all **VxWorks** tasks created by the **Wrapper**. All tasks are created using the **VxWorks taskSpawn** primitives and scheduled by the **VxWorks SMP** task scheduler. The lower the number, the higher the priority.

TASK NAME	CYCLIC RATE	PRIORITY	STACK SIZE
MainTask	100 Hz	100	20480
MainIf	100 Hz	130	20480
MainManagerIfTask	100 Hz	130	8192
NavigationTask	I/O driven	140	8192
NavigationManagerIfTask	100 Hz	140	8192
NavigationManagerIfProxyTask	100 Hz	140	8192
TelemetryTask	100 Hz or faster	180	8192
TelemetryManagerIfTask	100 Hz	180	8192
UserCommandTask	I/O driven	200	8192

Table 41: List of Tasks

7.5.10.1 ATOMIC VALUES

VxWorks atomic (vxAtomicLib) is used to serialize accesses to a single variable.

ATOMIC32_T	32-BIT SIGNED INTEGER VARIABLE.
atomic32_t	32-bit signed integer variable.

Table 42: List of Used Atomic Data Types

VXWORKS OPERATION	DESCRIPTION
vxAtomic32Get	Get a value from a 32-bit signed integer variable.
vxAtomic32Set	Set a value to a 32-bit signed integer variable.
vxAtomic32Inc	Increment a 32-bit signed integer variable by one.

Table 43: List of Used Atomic Operations

7.5.10.2 SEMAPHORES

VxWorks semaphore (semLib) is used to serialize accesses to a set of variables or functions. All the semaphores use the **VxWorks** options listed in Table 44: List of Used Semaphore Options below.

VXWORKS OPTION	DESCRIPTION
SEM_Q_PRIORITY	Priority sorted queue.
SEM_INVERSION_SAFE	No priority inversion (mutex opt.).

Table 44: List of Used Semaphore Options

NAME	TYPE	USAGE
NavigationTask: m_data_semaphore_id (for each navigation sensor)	read/write	Get or set CASS data in a Navigation Task.
TimeManager: m_time_semaphore_id	read/write	Get or set system time.
TelemetryManager: m_print_semaphore_id	mutual exclusion	Serialize prints to console.
TelemetryManager: m_enqueue_semaphore_id	mutual exclusion	Serialize adding messages to queue for downlink telemetry.
UserCommand: m_queue_semaphore_id	mutual exclusion	Serialize adding user's commands to queue for processing.

Table 45: List of Semaphore Variables

7.5.11 PRE-FLIGHT TEST MODE

The test mode allows the user to perform certain activities that are unavailable or not allowed in the normal mode. The test mode is enabled by issuing the command (0xFF), see SG901-0013-AFTU User Manual, Table 14 - Table 8-2: List of user commands, only when the **Wrapper** state is in **STANDBY** (CASS disabled) or **STANDBY** (CASS enabled). The test mode cannot be turned off, and the power must be cycled to restore the system back to the normal mode. Whenever the test mode is enabled, the status on the test mode can be found in the **AFTU** status message; see SG901-0013-AFTU User Manual, section 7.0 Telemetry. Currently, only one function is available in the test mode described in the following subsection.

7.5.11.1 STOP WATCHDOG RESET

In the test mode, the user can stop the **Wrapper** from resetting the **Watchdog** by issuing the command (0xEE); see SG901-0013-AFTU User Manual, Table 14 - Table 8-2: List of user commands. Normally, the **Wrapper** will keep resetting the **Watchdog** when it is healthy and stop resetting when it is unhealthy. The purpose of this function is to either simulate the **Wrapper** being unhealthy or to test the watchdog.

7.5.12 CODE DESIGN

This section describes the code structure in the **Wrapper**. It shows the relationship among the class objects used in the main logic flow in the **Wrapper** with the details for each object described in Table Table 46: Inputs and Outputs in the Main Logic Flow in the Wrapper and Table Table 47: Class Objects Used in Main Logic Flow in the Wrapper.

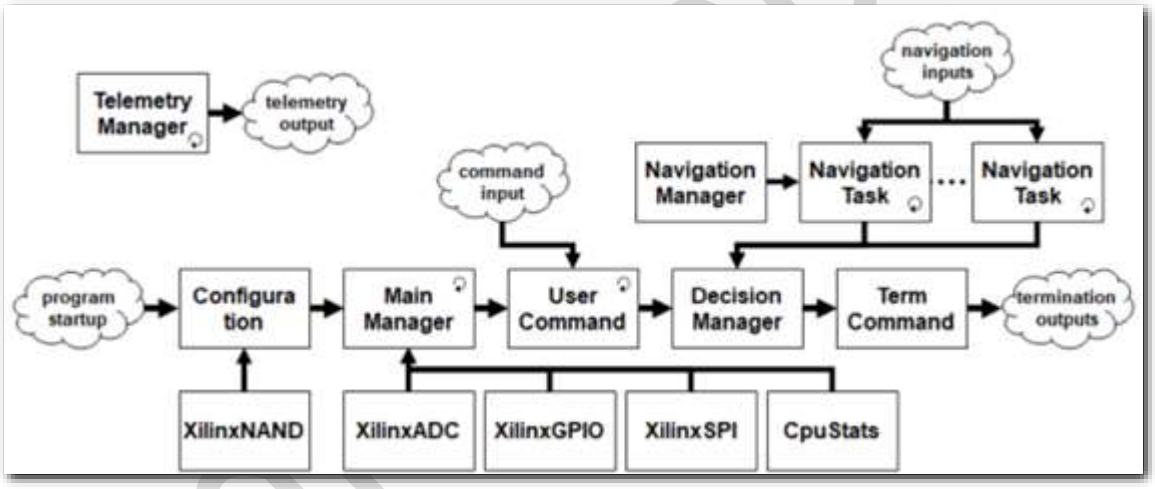


Figure 26: Main Logic Flow in the Wrapper

The code design inside the **Wrapper** is structured in the following list of parts:

OBJECT	DESCRIPTION
program startup	The Wrapper software is started by calling the **startAfss** function from the "usrAppInit" function in VxWorks.
telemetry output	The Wrapper generates and transmits telemetry messages.
command input	The Wrapper receives user commands.
navigation inputs	The Wrapper acquires navigation data from navigation sensors.
termination outputs	The Wrapper sends patterns to the FPGA to change the state of the termination and Watchdog circuits.

Table 46: Inputs and Outputs in the Main Logic Flow in the Wrapper

OBJECT	DESCRIPTION
Configuration	Configuration: is a class that reads the MDL configurations and processes them. The MDL_Tree class from the CASS Rule Engine is used to read and parse the MDL configuration.
XilinxNAND	XilinxNAND: is a class the Configuration class uses to read data from the NAND flash.
MainManager	MainManager is a class that runs the main loop, an infinite loop that will not exit until the NAFTU is power cycled.
XilinxADC	XilinxADC: is a class that reads and collects analog measurements from the Xilinx ADC.
XilinxGPIO	XilinxGPIO: is a class that reads and writes data from and to the Xilinx GPIO.
XilinxSPI	XilinxSPI: is a class that reads analog measurements from the TI ADCs through the SPI interface.
CpuStats	CpuStats: a class that collects and determines the CPU utilization stats.
UserCommand	UserCommand: is a class that acquires and processes commands from the user via telemetry uplink. It has an infinite loop and will not exit until the NAFTU is power-cycled.
DecisionManager	DecisionManager: is a class that gets navigation data from the NavigationTask classes, passes to the CASS Rule Engine, and processes CASS recommendations.
TermCommand	TermCommand: is a class that sends a pattern to the FPGA to change the state of the termination and Watchdog circuits.
NavigationManager	NavigationManager: is a class that creates and manages two to three NavigationTask objects.
NavigationTask	NavigationTask: is a class that acquires and processes data from a navigation sensor. It has an infinite loop and will not exit until the NAFTU is power-cycled.

OBJECT	DESCRIPTION
TelemetryManager	TelemetryManager: is a class that compiles and transmits telemetry messages. It has an infinite loop and will not exit until the NAFTU is power-cycled.

Table 47: Class Objects Used in Main Logic Flow in the Wrapper

Figure 27: Supporting Class Objects in the Wrapper shows the supporting class objects used with the details for each object described in Table 48: Supporting Class Objects in the Wrapper.

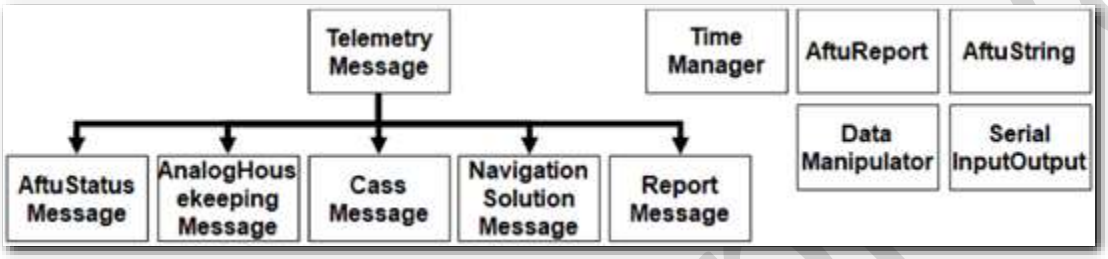


Figure 27: Supporting Class Objects in the Wrapper

OBJECT	DESCRIPTION
TelemetryMessage	**TelemetryMessage** is a generic class that encodes generic data fields into the telemetry message.
AftuStatusMessage	**AftuStatusMessage** is a subclass of the TelemetryMessage class. It encodes data about NAFTU status into the telemetry message.
AnalogHousekeeping Message	**AnalogHousekeepingMessage** is a subclass of the TelemetryMessage class. It encodes data about analog measurements into the telemetry message.
CassMessage	**CassMessage** is a subclass of the TelemetryMessage class. It encodes CASS stream data from the CASS Rule Engine into the telemetry message.
NavigationSolutionMessage	**NavigationSolutionMessage** is a subclass of the TelemetryMessage class. It encodes data from a navigation sensor into the telemetry message.
ReportMessage	**ReportMessage** is a subclass of the TelemetryMessage class. It encodes the reporting message into the telemetry message.
TimeManager	**TimeManager** is a class that manages time synchronization and provides system time information.
AftuReport	**AftuReport** is a class that holds a reporting message and associate information.
AftuString	**AftuString** is a utility class that holds and manipulates a string of characters.
DataManipulator	**DataManipulator** is a utility class that decodes or encodes an array of data bytes.
SerialInputOutput	**SerialInputOutput** is a utility class that receives and sends data from and to a serial port.

Table 48: Supporting Class Objects in the Wrapper

7.5.13 SOFTWARE PARTITIONING STRATEGY

To help facilitate software partitioning through time and space, Sagrad will separate the critical code from the support critical code. The **CASS** engine, along with the critical code, will be in its own **Real Time Process** (RTP), which is a **VxWorks** construct that enforces hardware **Memory Management Unit** (MMU) protections along with protecting the kernel level address space from being corrupted and confined to its processor core.

The support critical code shall also be on its own **RTP** and confined to its own processor core. The critical and support-critical code will run as two separate applications in one **VxWorks** instance. The two **RTP**s shall communicate through inter-process communication (IPC) using **VxWorks**-supplied named message queues.

Interface and proxy classes will be on different sides of the partition to send, receive, and handle all required processing activities. The project will continue to use the existing time partition infrastructure in the code base. Currently, based on the clock tick (which occurs 1000 times a second), certain actions are performed every **10**th cycle, **100**th cycle, **200**th cycle, or **3000**th cycle, such as checking the health status or updating **CASS**.

7.5.13.1 VXWORKS FEATURES AND FUNCTIONALITIES OVERVIEW

7.5.13.1.1 VXWORKS KERNEL ARCHITECTURE

The **VxWorks Real-Time** operating system before version 6.0 provided only a single memory model space for the user application and the operating system to use[1]. There was no segregation of the user application and the operating system[1]. This model provided performance and flexibility but also risk due to the fact that only skilled developers could ensure that the kernel and applications did not interfere with each other in the same memory space[1].

VxWorks 6.0 and all future releases provided support for real-time processes (RTPs)[1]. This allowed the execution of applications in user mode and other features that are common to other operating systems and provide a crisp boundary between kernel and application[1]. This architecture is often referred to as the process model[2]. This model was adopted by **VxWorks** specifically aimed at meeting the requirements of determinism and speed that are needed for hard real-time systems[2].

7.5.13.1.2 VXWORKS REAL-TIME PROCESS (RTP) MODEL

VxWorks real-time processes (RTPs) provide the means to execute code in user mode as opposed to kernel mode[3]. The **NASA WFF** engineering tech transfer code utilizes a downloadable kernel module or **DKM**. Therefore, the application runs in kernel mode and does not have any of the memory protections that are provided in user mode.

Each application in an **RTP** has its own address space, which contains the executable program, the program's data, stack for each task, heap, and all of the resources associated with the management of the process itself[3]. **RTP**s can contain one or more tasks[3]. Other operating systems will sometimes call these threads[3].

VxWorks processes are called real-time processes because they are designed to support the determinism that is required of real-time systems[4]. **VxWorks** provides determinism by allowing the real-time scheduler to schedule tasks in **RTP**s globally[4]. Processes are not scheduled, only tasks are scheduled[5]. Scheduling is based on the priority preemptive policy[6]. This allows the highest priority task in the system that is ready to be run, to execute regardless of if it is part of the kernel or any other process[6]. **VxWorks** has a priority range of **0** to **255**[7]. It provides **256** total priority levels, with 0 being the highest and **255** being the lowest[7]. All application tasks should be given a priority of **100 - 255** in order to not interfere with the kernel and any hardware or network drivers[7]. All kernel-related tasks are given a priority of **0 - 99**[7].

7.5.13.1.3 RTP MEMORY PROTECTION

RTPs are protected from other **RTP**s running on the system when the **MMU** has been configured to be used in the **VxWorks** kernel[8]. **RTP**s cannot call functions, access data, or memory of other processes[8]. Kernel functions that are not exported for public use[8] — even if the **MMU** is not enabled — are not accessible to the **RTP**. With the **MMU** enabled, the **RTP**s cannot read or write to memory external to its own address space[8]. The **MMU** shall be utilized with our design and configuration of **VxWorks**.

7.5.13.1.4 VXWORKS SYMMETRIC MULTIPROCESSING (SMP)

VxWorks symmetric multiprocessing (SMP) is designed for two or more processor cores and provides the same **RTOS** characteristics of performance and determinism as uniprocessor (UP) deployments of **VxWorks**[9]. Scheduling is priority-based preemptive. However, due to two processors, tasks on different CPUs can be executed concurrently.

Any task in an **SMP** deployment of **VxWorks** can run on any of the CPUs available in the system. There are instances where it may be useful to assign specific tasks to a specific logical CPU. **VxWorks SMP** provides this capability, which is referred to as **CPU** affinity[10].

7.5.13.1.5 VXWORKS CPU AFFINITY

VxWorks SMP provides the mechanism for assigning tasks to a specific processor or CPU[11]. This will ensure that the tasks are only executed on that CPU[11]. **CPU** affinity will only allow the task to run on the CPU that it has been assigned to. Having a task assigned to a CPU, i.e., having task **CPU** affinity, can improve performance due to the **L1** cache being filled with code that is required for that task[11]. Otherwise, the task would have to be moved to another CPU, and the **L1** cache must be refilled with new instructions, which could affect performance.

Sagrad will use **CPU** affinity to partition the critical and safety-critical tasks into their own **CPU** core. This will improve performance by less context switching and keeping the **L1** cache filled.

7.5.13.1.6 SMP VXWORKS REAL-TIME PRIORITY PREEMPTIVE SCHEDULING POLICY

SMP VxWorks utilizes a real-time priority preemptive scheduler. This scheduler will schedule the **N** (N is the number of CPUs) highest-priority tasks that are in the ready state, to run[12]. The priority-based preemptive scheduler preempts the CPU when a task has a higher priority than the current task running[13]. The kernel will always ensure that the CPU is allocated to the task with the highest priority that is ready to run[13].

There is not a task that does the scheduling; instead, the schedule is invoked when a **task** or **interrupt service** routine (ISR) exits a kernel critical section[12]. The scheduling decision is made when the ready queues are updated[12].

A disadvantage of this scheduling policy is that if multiple tasks of equal priority share the processor if the task that is running never blocks, it will prevent all other tasks from being run. Sagrad is fully aware of these shortcomings and has taken steps to ensure the tasks have balanced priorities within the system. No task has the same priority and will be blocked when appropriate to facilitate balanced task scheduling.

7.5.13.1.7 MESSAGE QUEUE IPC

In **RTP**s, the address space of each process is invisible and inaccessible to other tasks running in other processes (RTPs); there are still means to communicate with **inter-process communication**

(IPC) mechanisms such as **Public** message queues, which we will be using in our design. **Message** queues provide a higher-level mechanism for direct communication of messages between tasks.

Message queues are the primary intertask communication mechanism within **VxWorks**[14]. **Message** queues can be private or public objects. **Private** message queue objects are not accessible across the **RTP** boundary. **Public** message queues are accessible across the **RTP** boundary because the namespace of the **Message** queues is at the kernel level. **Message** queues allow the user to send variable-length messages[15]. In order to have full duplex communication between two tasks, two **Message** queues need to be utilized[14].

VxWorks M**essage** queues have a timeout parameter. Since **Message** queues will be utilized on both the critical and support critical **RTP**s, timeouts will be used to ensure that a task in the critical **RTP** is not artificially blocked waiting for a response if an off-nominal condition has occurred in a task in the support critical **RTP**.

REFERENCES:

1. (VxWorks Application Programmer's Guide, 6.9 1)
2. (VxWorks Application Programmer's Guide, 6.9 2)
3. (VxWorks Application Programmer's Guide, 6.9 5)
4. (VxWorks Application Programmer's Guide, 6.9 6)
5. (VxWorks Application Programmer's Guide, 6.9 6)
6. (VxWorks Application Programmer's Guide, 6.9 7)
7. (VxWorks Application Programmer's Guide, 6.9 93)
8. (VxWorks Application Programmer's Guide, 6.9 9)
9. (VxWorks Application Programmer's Guide, 6.9 518)
10. (VxWorks Application Programmer's Guide, 6.9 520)
11. (VxWorks Application Programmer's Guide, 6.9 540)
12. (VxWorks Application Programmer's Guide, 6.9 544)
13. (VxWorks Application Programmer's Guide, 6.9 94)
14. (VxWorks Application Programmer's Guide, 6.9 137)
15. (VxWorks Application Programmer's Guide, 6.9 138)

7.5.13.1.8 COMMON VULNERABILITY EXPOSURE (CVE) AND DEFECTS VIA WIND RIVER'S ONLINE DB RESOURCES

Research and analysis have been performed on the common vulnerability exposure (CVE) and defects available from Wind River for **VxWorks 6.9**. In the CVE, all **Ethernet**-related vulnerabilities are related to the **Intel Ethernet I210 Controller** series of network adapters. The **AFTU Wrapper** program uses a Microchip Technology model KSZ9031RNXVA. We believe the **Ethernet** adapter issue is unrelated to the **AFTU Wrapper** software. Two defects have not been fixed, but neither affects our OS use, **VXW6-87249** and **VXW6-87184**. Defect **VXW6-87249** pertains to an issue with a failure to reboot after several **VME bus** resets. The workaround is a power cycle. The second defect, **VXW6-87284**, pertains to an issue where the user needs to have a **/tmp** directory in Windows for the tac command to have data piped to it in the **VxWorks** shell. The program plans on having the shell disabled.

7.5.13.2 SOFTWARE PARTITIONING METHODS AND DESIGN

7.5.13.2.1 PROXY DESIGN PATTERN

The **Proxy** software engineering design pattern was the most efficient way to partition the code and change as little code as possible. The **Proxy** design pattern is one of the twenty-three well-known **Gang of Four Design Patterns** (GoF) that describe how to solve recurring design problems to design flexible and reusable object-oriented software- objects that are easier to implement, change, test, and reuse.

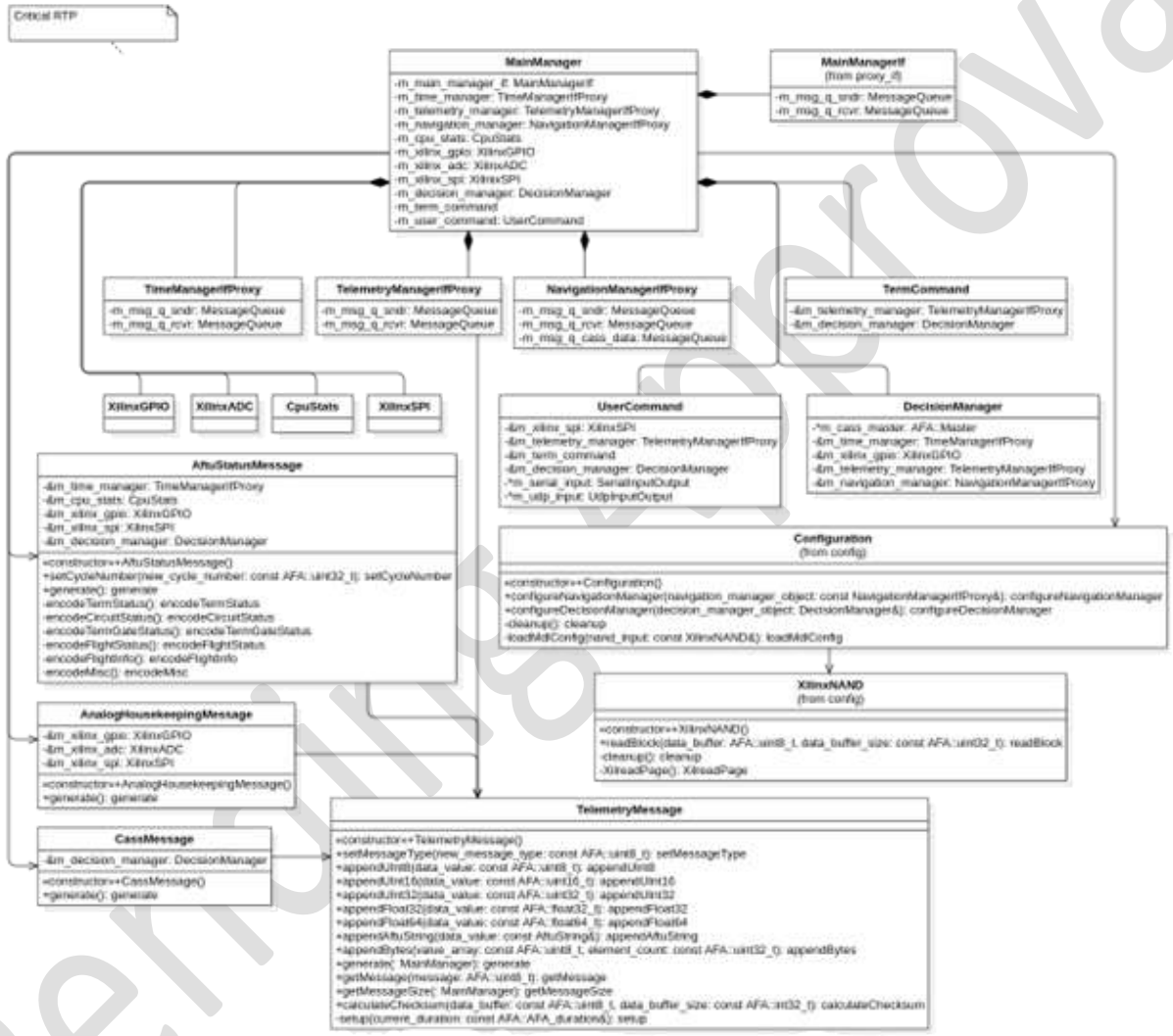
This, more specifically, is called a **Remote Proxy**. The use of the proxy can simply be forwarding to the real object, in our case, across **RTP** process boundaries. For the client, using a proxy object is similar to using the real object because both implement the same interface. This helps to avoid changing and rewriting large portions of the code. In Figure 28: RTP, Proxy, and Utilities Class Diagrams, one can see that there will be two **RTP**s. Classes deemed support critical are removed from the main code running in the **Critical RTP**.

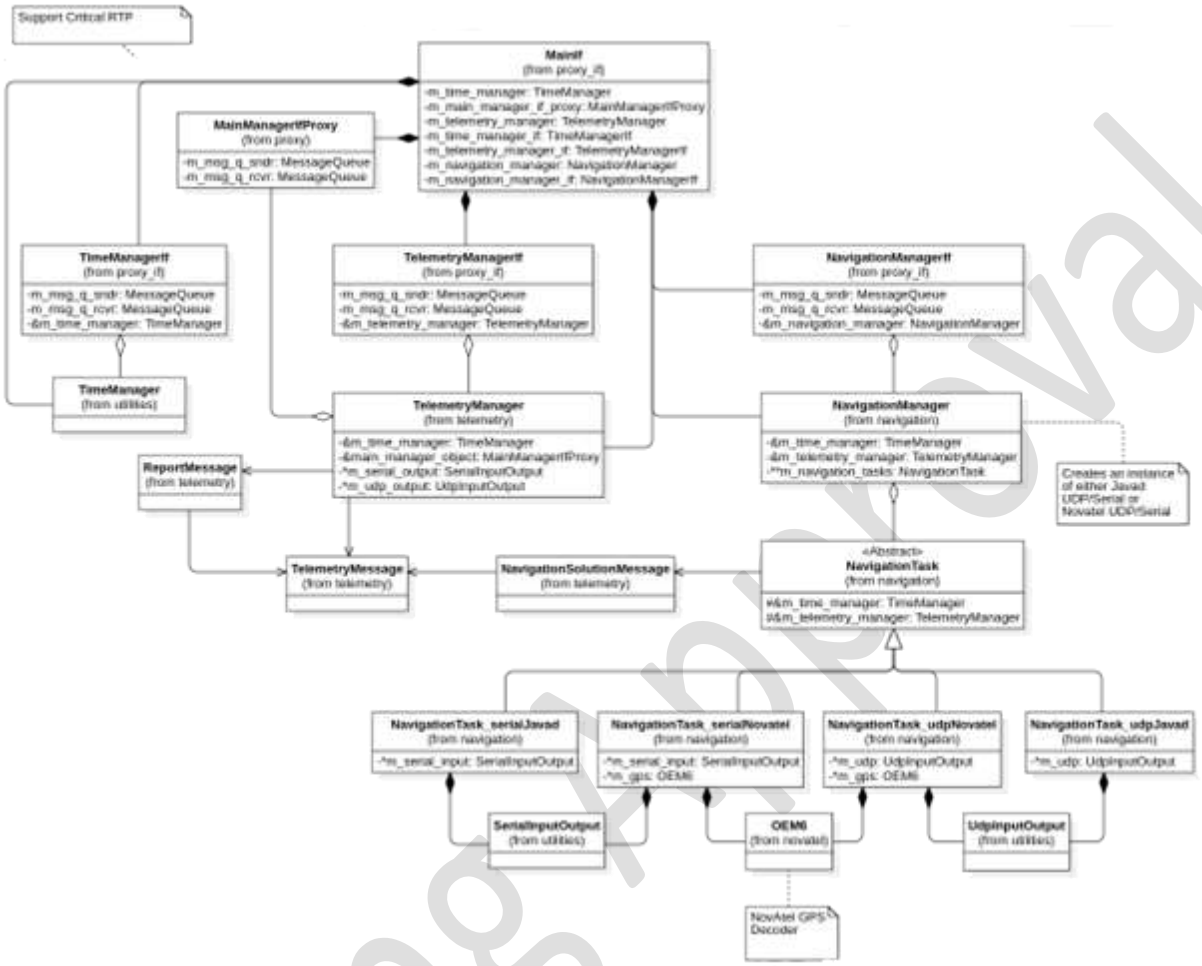
The **TelemetryManager** has been removed and replaced with the **TelemetryManagerIfProxy**. The MainManager will use the proxy class to communicate across the **RTP** boundary using **VxWorks Message** queues to the actual **TelemetryManager** in the **Support-Critical RTP**. The same pattern is repeated for the **Time** and **Navigation Manager** classes.

When a function is called from one of the proxy classes, the proxy class itself will populate a payload struct and place it on its message queue. Its respective interface class on the **Support Critical** side

will receive the payload struct, perform the function call, and return the results to the proxy class in the **Critical RTP**.

Using the proxy design pattern significantly reduces the amount of change to the current code, as very little logic will change. This also facilitates the project's objective of maintaining the current code module organization intact.





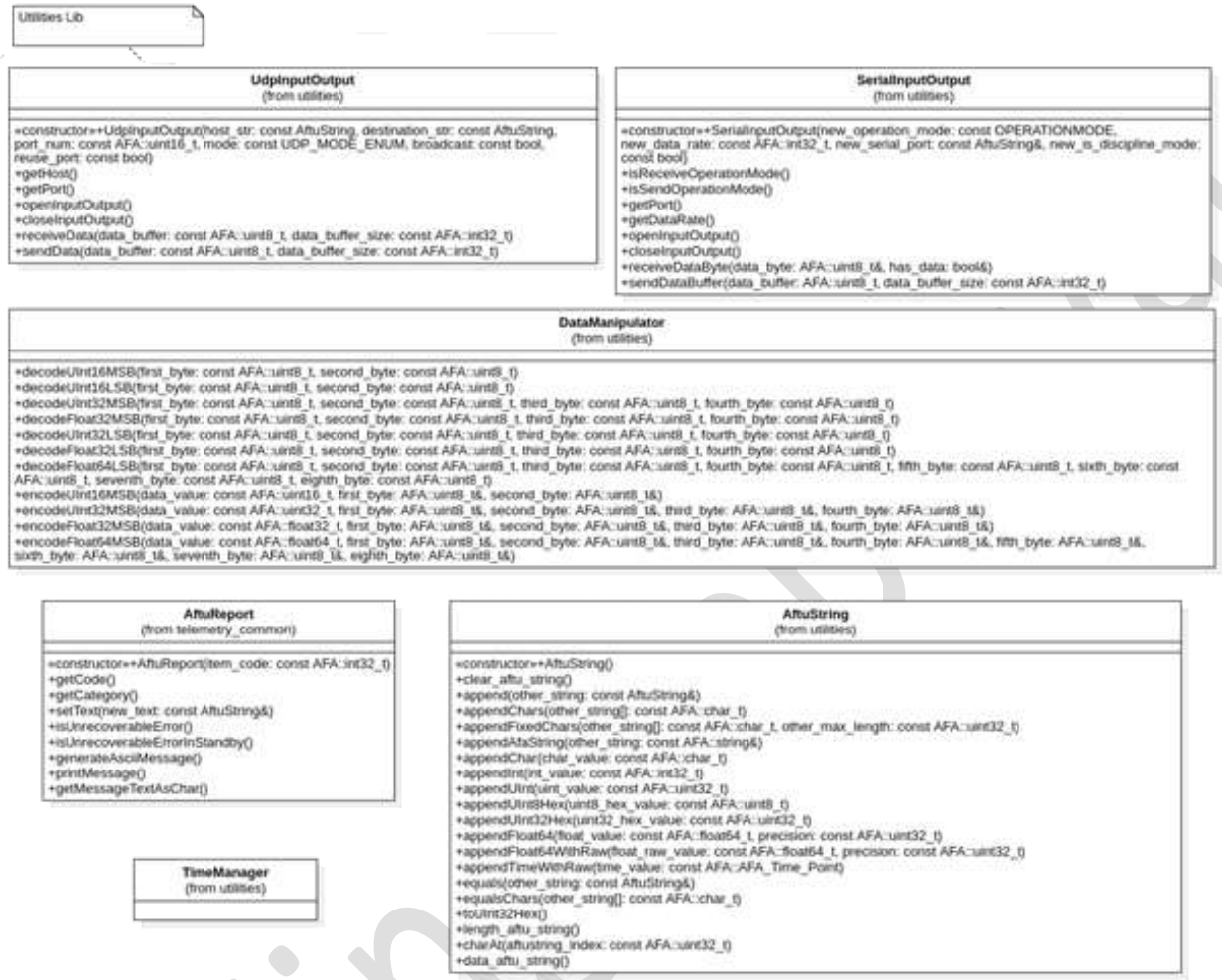


Figure 28: RTP, Proxy, and Utilities Class Diagrams

7.5.13.2.2 CRITICAL RTP

The MainManager will instantiate the proxy classes. The proxy classes will contain a send-and-receive public message queue. Each proxy class will have a payload struct that will be used to marshal the data across the RTP boundary. See: Figure 29: Critical RTP Proxy Object Class Diagram below.

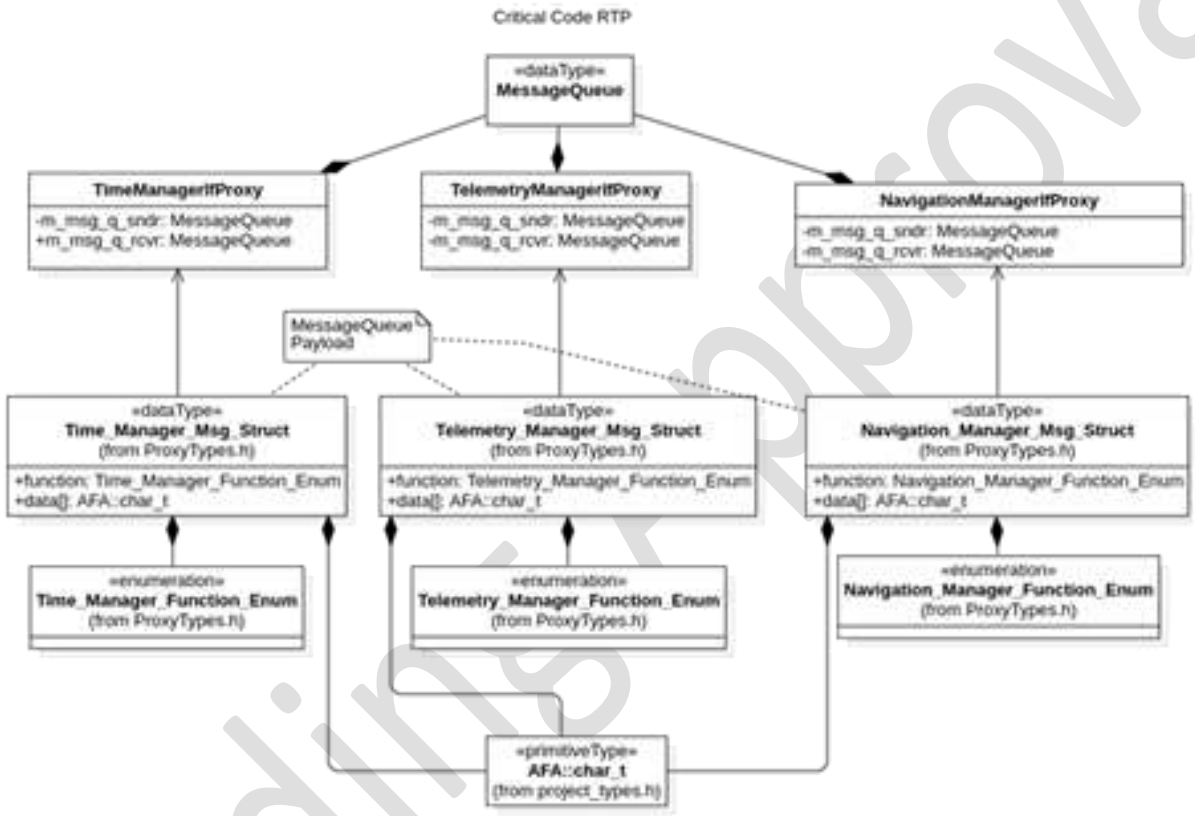


Figure 29: Critical RTP Proxy Object Class Diagram

7.5.13.2.3 SUPPORT CRITICAL RTP

The support critical RTP will contain the actual **Time**, **Telemetry**, and **Navigation** managers. The **Critical RTP** will communicate with the interface class of their respective manager. For example, **TimeManagerIf** will instantiate the real **TimeManager**. The interface class will forward all messages received on its message queue from the **Critical RTP** to the real **TimeManager**. Then, the results will be sent back through the **interface** class' **message** queue to the **Critical RTP**. Each **If** class will have a **message** payload containing all the data and information needed to perform the requested operation. See: Figure 30: Critical Support RTP Interface (If) Class Diagram below.

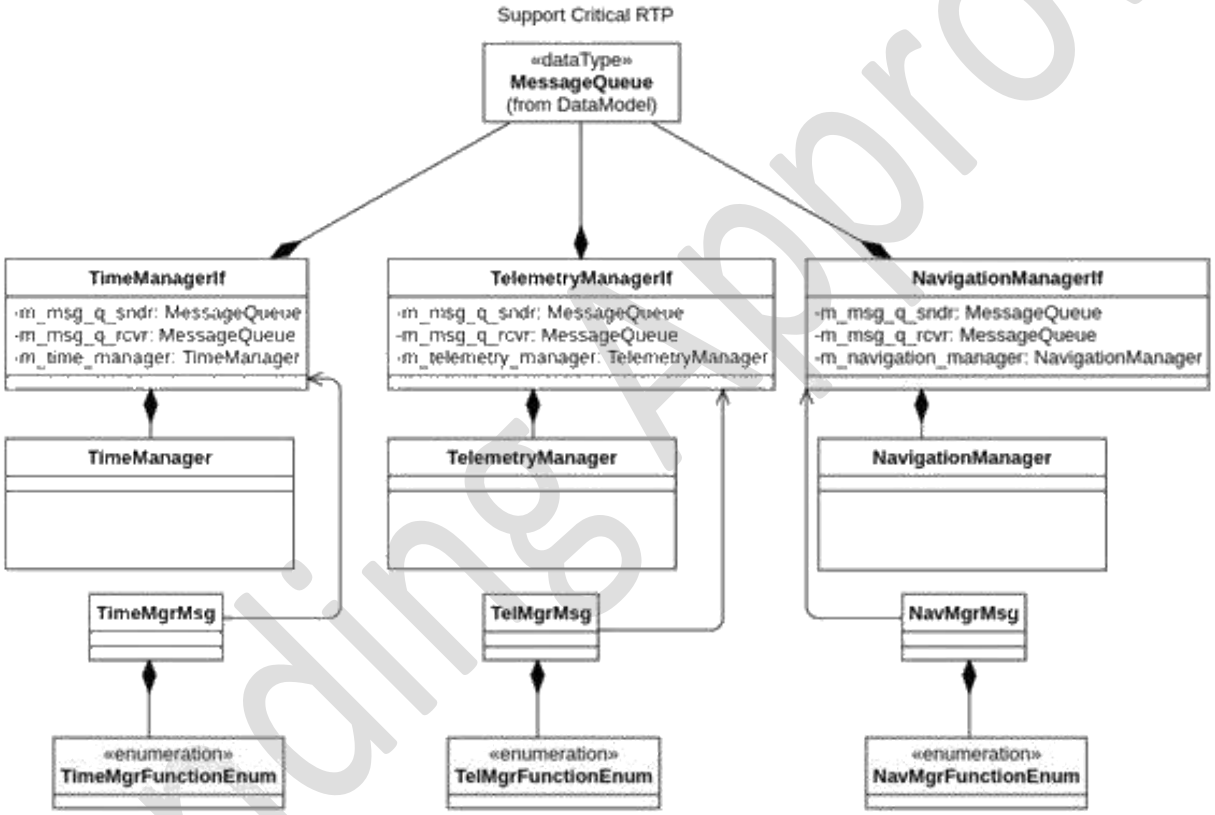


Figure 30: Critical Support RTP Interface (If) Class Diagram

7.5.13.2.4 SOURCE CODE PACKAGE REORGANIZATION

Inside of the **Workbench IDE**, files are grouped by packages. These packages or subsystems will be moved into their respective **RTP**s. This will facilitate the separation of critical and support critical code. See: Figure 31: Critical and Support Critical Package Diagrams below.

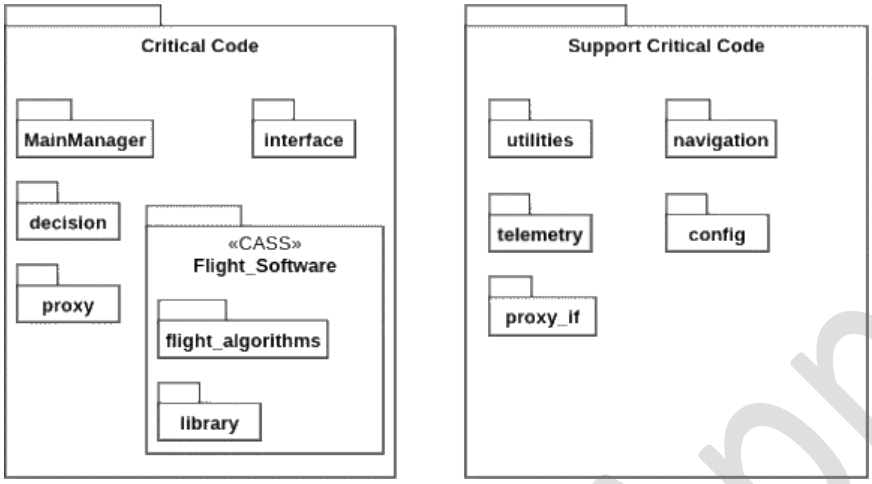


Figure 31: Critical and Support Critical Package Diagrams

7.5.13.2.5 CRITICAL AND SUPPORT CRITICAL DEPLOYMENT

Deployment diagrams help communicate the actual layout of a running node or task in a system. The critical and support critical **RTP**s will utilize **VxWorks** tasks as was implemented before. Figure 32: Critical and Support Critical Deployment Diagrams represent a high-level depiction of the two RTPs, their running tasks, the priority of the running tasks, and the **CPU** core they will reside on. **VxWorks** tasks are from **0 - 255**, with **0** being the highest priority and **255** being the lowest. Priorities **0 - 99** are reserved for the kernel.

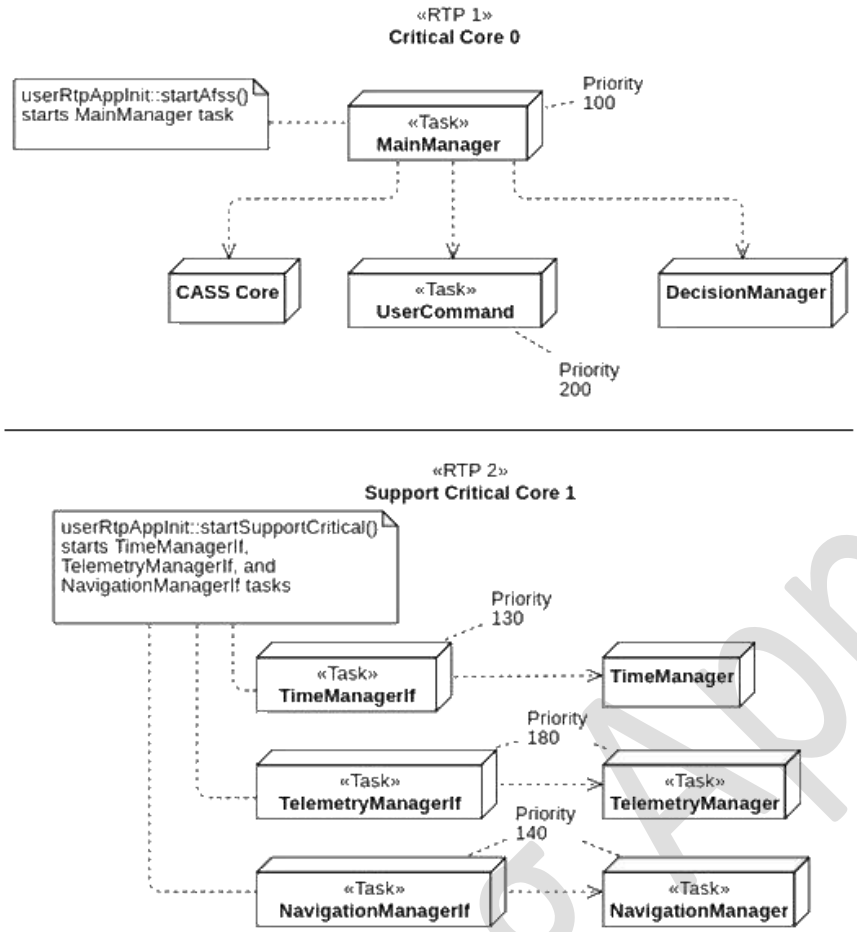


Figure 32: Critical and Support Critical Deployment Diagrams

7.5.13.2.6 CRITICAL TO SUPPORT CRITICAL SEQUENCE DIAGRAM

The sequence diagram below shows how messages will be sent and received across the **RTP** boundary. This pattern will be repeated for all messages for the **TimeManager**, **NavigationManager**, and **TelemetryManager**.

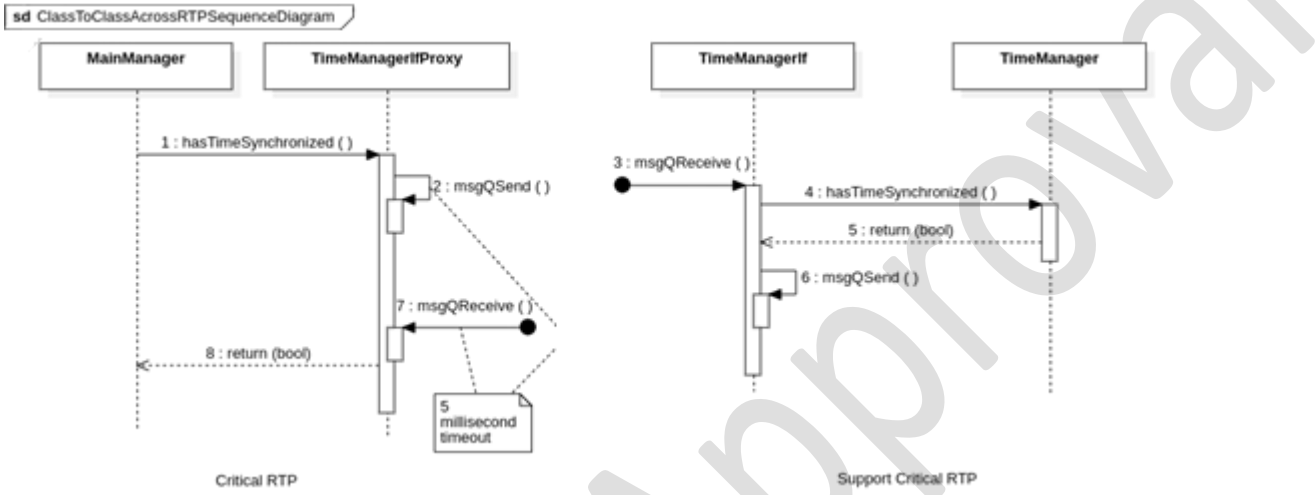


Figure 33: Critical and Support Critical Sequence Diagrams

7.5.13.2.7 TIMEMANAGER CLASS DIAGRAM

Below is the class diagram for the **TimeManager** class. This shows the interfaces contained within.

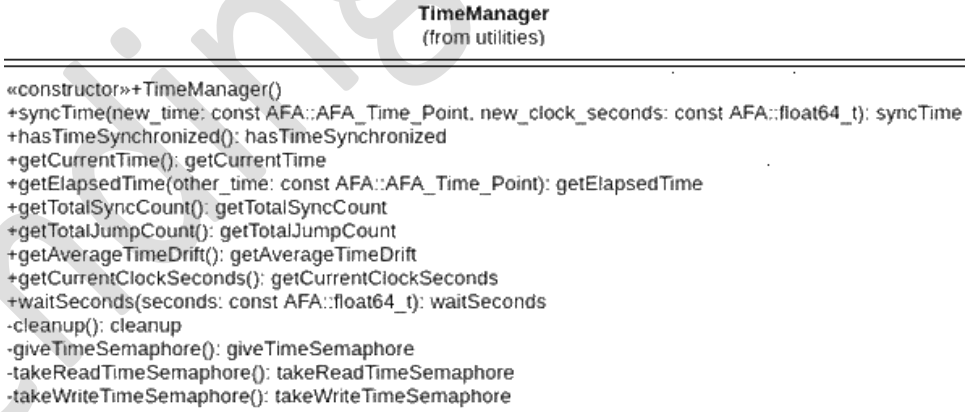


Figure 34: TimeManager Class Diagram

7.5.13.2.8 TELEMETRYMANAGER CLASS DIAGRAM

Below is the **TelemetryManager** class diagram. This shows the dependency between the **TelemetryManager** and the **TimeManager**.

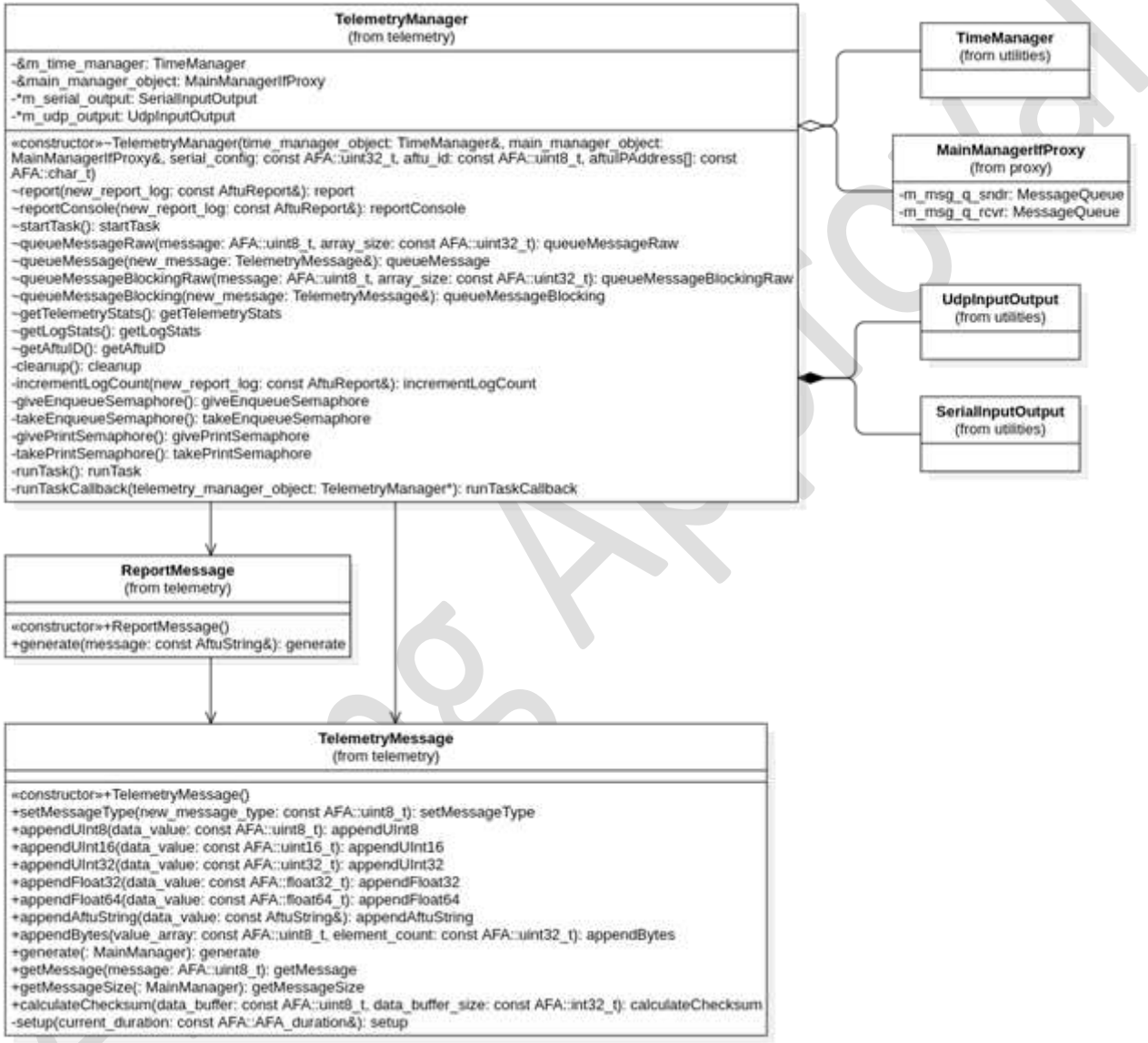


Figure 35: TelemetryManager Class Diagram

7.5.13.2.9 NAVIGATIONMANAGER CLASS DIAGRAM

Below is the **NavigationManager** class diagram and some of its dependencies.

Figure 36: NavigationManager Class Diagram

7.5.13.2.10 USERCOMMANDS CLASS DIAGRAM

Below is the **UserCommand** class diagram and some of its dependencies.

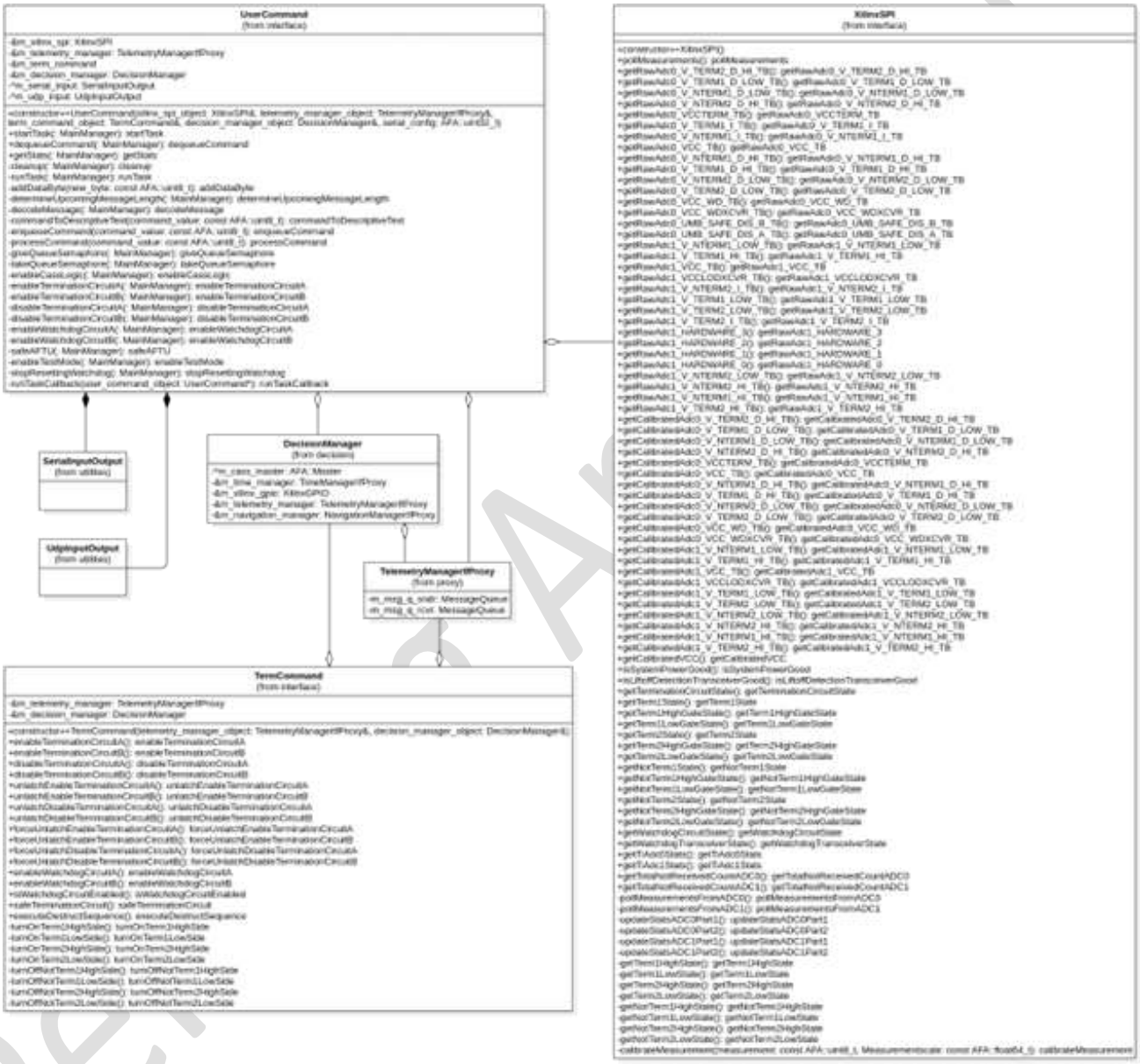


Figure 37: UserCommand Class Diagram

7.5.13.2.11 DECISIONMANAGER CLASS DIAGRAM

Below is the **DecisionManager** class diagram and some of its dependencies.

Figure 38: DecisionManager Class Diagram

7.5.13.2.12 MAINMANAGER CLASS DIAGRAM

Below is the **MainManager** class diagram and some of its dependencies

Figure 39: MainManager Class Diagram

7.5.13.2.13 MAINIF CLASS DIAGRAM

Below is the **MainIf** class diagram and some of its dependencies.

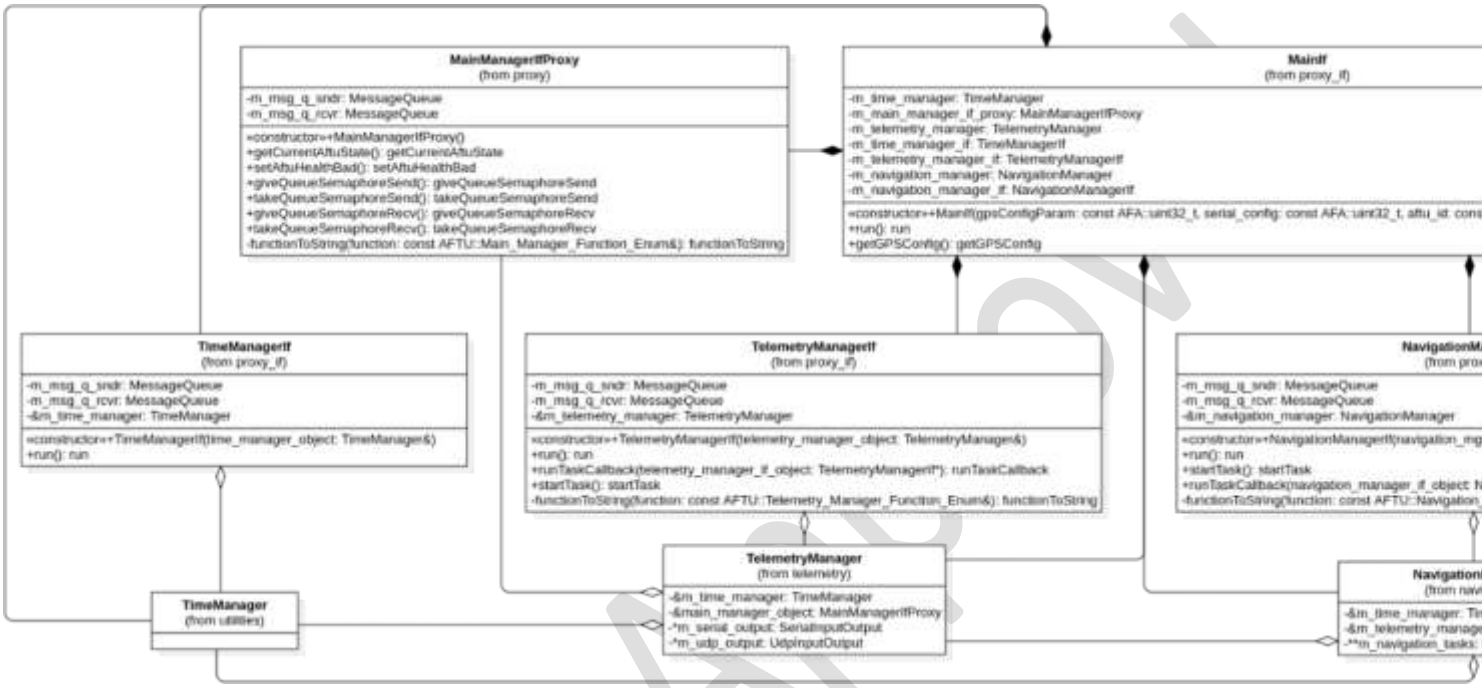


Figure 40: MainIf Class Diagram