Flight Software Onboarding Documentation

Brandon Molyneaux, Grant Robertson, Houston Walley, Joseph Langan, Leah Lee, Theo Zinner

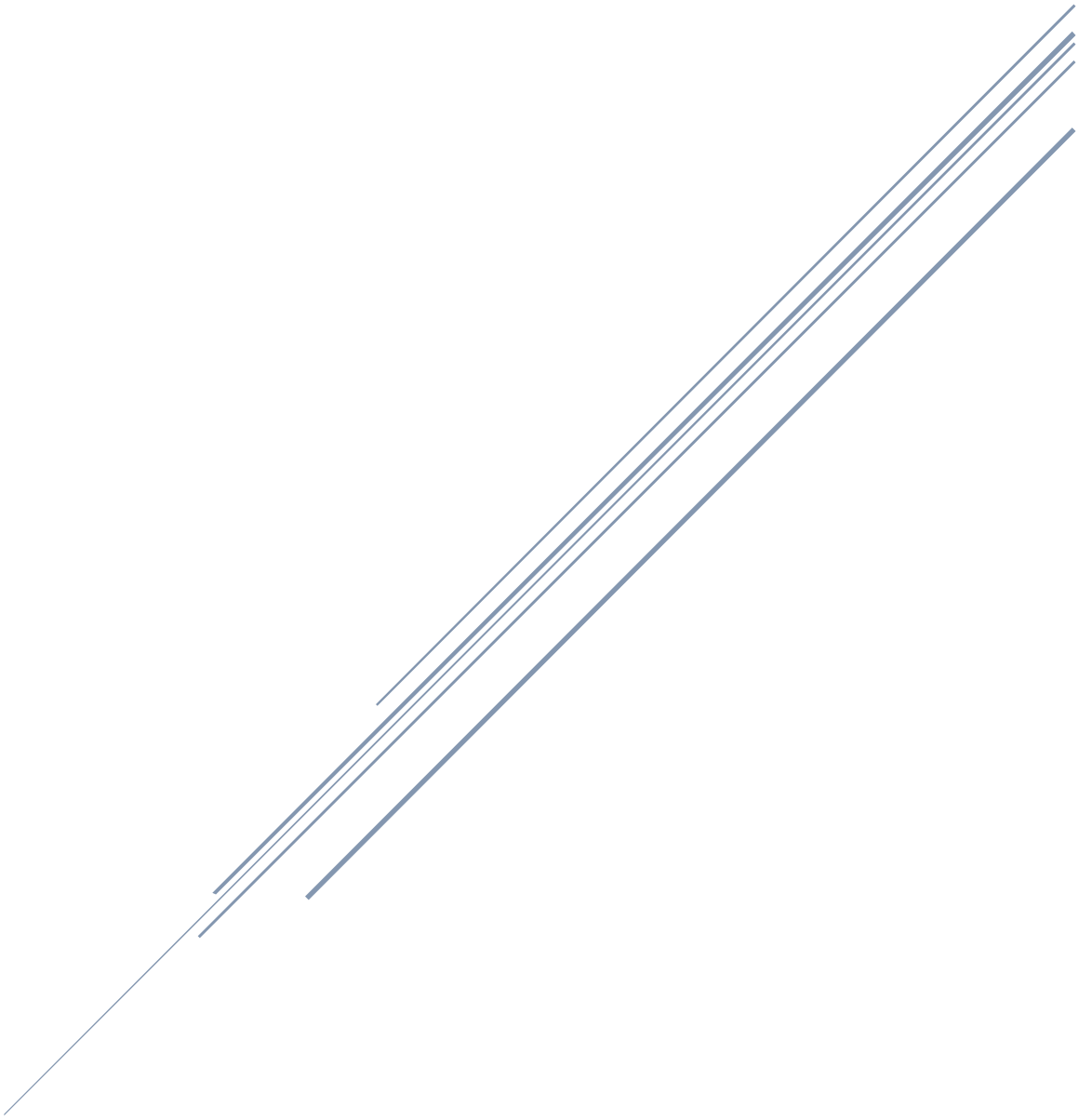


Table of Contents

1 Introduction *4*

1.1 Project Objectives **4**

1.2 Simulation Environment **4**

1.3 Engineering Test Unit Environment **4**

2 Design *5*

2.1 Considerations **5**

2.2 Satellite Design **5**

2.3 Subsystems **6**

2.4 Software States **7**

2.4.1 Introduction 7
2.4.2 Detailed Description 8
2.4.3 State Queue 9
2.4.4 Burn 9
2.4.5 Charge 10
2.4.6 Comms 10
2.4.7 Contingency 11
2.4.8 Deployment 11
2.4.9 Disposal 12
2.4.10 Safety 1 13
2.4.11 Safety 2 14
2.4.12 Pre-disposal 15
2.4.13 RW Desaturation 16
2.4.14 Science 16
2.4.15 Warmup 16

2.5 Future Considerations for Software States **17**

3 Software States Simulation Tool *17*

3.1.1 Software State Testing 17

4 Cameo Enterprise Architecture *18*

4.1 Overview **18**

4.2 Using Cameo **19**

4.3 Pre-defined ports **19**

4.4 Port Permission Problem **20**

4.5 Project File Permission Change **21**

4.6 Pre-defined Component Absence Problem **25**

4.7 Creating Topology Diagrams **26**

4.7.1 IBD Creating Port and Component Packages 26
4.7.2 Component Instance Stereotype and Tags 29
4.7.3 Displaying and Hiding ports 34

4.7.4 Connecting Ports ... 36
4.7.5 Parameters IBD ... 38
4.7.6 Rate Group IBD ... 38
4.7.7 REF Commanding IBD ... 42
4.7.8 REF Logging IBD ... 45
4.7.9 REF Telemery IBD ... 46
4.7.10 REF Time IBD ... 47
4.7.11 SUB Thermal Control IBD ... 48
4.8 Null Role Error ... 49
4.9 Opposite Type Error ... 51
4.10 Optional XML ... 53
4.11 Inertial Topology Diagrams ... 60
5 F Prime ... 66
5.1 Background ... 66
5.2 Usage ... 66
5.3 Example component design ... 67
5.4 Abstracted Topology Diagram Design for Subsystem Components ... 68
5.5 No Target Error while Generating C++ Files ... 68
5.6 Commands for Running Demo ... 70
5.7 Useful Links ... 71
6 Operating System ... 71
6.1 OS Overview ... 71
6.2 OS Survey ... 71
6.3 GPOS vs RTOS ... 72
6.4 Primary OS Candidate ... 73
6.5 Secondary OS Candidate ... 73
6.6 Bootloader Considerations ... 73
6.7 Yocto Vs. Traditional Kernel Development ... 74
6.8 Kernel Selection ... 74
6.9 What the OS Needs to Support ... 74
6.10 Operating Systems in Development ... 74
6.11 Drivers ... 75
6.12 Developing the OS ... 75
6.12.1 Yocto Development ... 75
6.12.2 76
6.12.3 Editing the Kernel ... 79
7 Peripherals and Communications ... 79
7.1 Serial Vs Parallel ... 79

7.1.1 RS 422 80
7.1.2 CAN bus 81
7.1.3 SPI 82
7.1.4 SpaceWire 82
7.1.5 I2C 83
8 Hardware and Hardware Specific Software 83
8.1 Overview 83
8.2 Development Board 84
8.3 Design Tools 85
7.5 Bootloader Information 95
7.6 Overview of bringing Linux Online for the Development Board 95
8.4 Overview 97
9 Reed-Solomon Codes 97
9.1.1 Introduction 97
9.1.2 Encoder 98
9.1.3 Interleaving 99
9.1.4 Decoding Interleaved Reed-Solomon Symbols 100
9.2 Convolutional Codes 100
9.2.1 Introduction 100
9.2.2 Encoder 100
9.2.3 Viterbi Decoding 101
9.3 Frame Synchronization 103
9.3.1 Attached Sync Markers 103
10 Glossary 104
11 References 106

1 Introduction

1.1 Project Objectives

The overall objective of this project is to understand radiation effects to make travel to the moon safer. A component of this project is to design and build a satellite system. Originally, this system was going to mimic and build off of what AEGIS has done due to the success of the project, but due to government restrictions and NDA's, this endeavor was not feasible. Thus, a new system had to be developed. This system consists of building a real-time operating system on a development board and building software components using F Prime, an open source framework for flight software and embedded systems. The software components that are built will be run on the development board with the OS. From here, the software and hardware components will be assembled into a NASA CubeSat, which will then be launched into deep space to collect information about the radiation environment surrounding the Moon in attempt to make space travel safer for humans.

1.2 Simulation Environment

The simulation environment is F Prime with all the driver components stubbed out. This can be run on most computers. No real pre-requisites exist to build this system. However, if there's time, some effort can be put into automated testing architecture. It should be fairly easy to script commands, but validation may be difficult. Some logic to analyze telemetry output may be needed. During primary development, the workflow will be largely similar. After analyzing the interface control document and replicating the device functions, the higher-level components will be updated in order to manage these devices. Multiple devices can be worked on at the same time, but it will take some coordination. If developers take ownership of components, the high-level component developers will have to work closely with the driver stub component developers. Alternatively, the responsibility of high level components can be shared, and the low level components are developed individually. While the components are being developed, high level tests need to be created. These tests include unit tests that only affect the new device and integration tests that affect previously developed devices. For example, a unit reaction wheel test could include the failure state where the 4th reaction wheel has to be used. An integration test for the reaction wheels would be the control loop between the star tracker and reaction wheel.

1.3 Engineering Test Unit Environment

The engineering test environment will be the development board running a Linux Operating system using PREEMPT_RT. See section 4 and 5 for the Operating system and development board. This testing environment will allow for the development of device drivers for each peripheral. The end goal of this is to integrate device drivers with the custom hardware created by the Command and Data Handling team.

2 Design

2.1 Considerations

There are two main considerations in designing the flight software (FSW) for this project: the effects of radiation on the On-Board Computer (OBC) and the probability of introducing faults as the complexity of the system and development process increases. As the satellite orbits the Moon, radiation may impact the OBC, causing data to be corrupted. These events may result in large-scale corruption that is easily detectable, as it will vary drastically from what is expected, or discretely corrupted data that appears to be nominal, but is not correct. Radiation-tolerant hardware and hardware-level error detection and correction methods will be used in order to make the system less susceptible to radiation-induced failure but may not be sufficient to detect and resolve all faults in the system. Many of these concerns will be addressed by the Command and Data Handling team.

The FSW must have additional monitoring of the health of software and hardware components to protect the system from failure. The health monitoring component will be able to evaluate the response from each component to determine if it is behaving as expected and if not, will report these failures to ground control and execute any necessary contingency operations. Some of these operations include, but are not limited to, rebooting the hardware and waiting for ground control instructions. In addition to the health monitoring system, other subsystems such as power management and attitude control, are built into the FSW to monitor and manage the respective system(s).

As functionality is added to the FSW, there may be considerable increases in the complexity of the system, giving a higher likelihood that bugs will be present and pose a risk to the integrity of the mission. Previous and following sections discuss the development process that has been created to mitigate that risk and uncover system defects through extensive and robust testing measures. While testing and code review may be effective in uncovering and repairing system defects, attempting to introduce fewer defects in the development process is even more effective. This may be achieved by prioritizing less complex implementations of functions and refraining from adding functionality to the system that is not strictly necessary.

2.2 Satellite Design

Satellite operations are grouped into several subsystems with corresponding hardware components and dedicated teams working on their development. Communications between the satellite and ground control will be facilitated by an on-board radio, with that communication being initiated through the FSW. Solar panels will be used to generate power for the satellite and must be deployed when separation from the launch vehicle is detected. Guidance and navigation (GN&C) encompasses several different components on the satellite. These include IMUs that measure velocity and orientation of the satellite and star trackers that take pictures of the stars to determine the satellite's position, sun sensors that provide additional measurements of the satellite's position. The FSW will be responsible for taking the output of these devices and reporting it to ground control as well as using that information to guide the satellite. Similarly, temperature monitors will be placed around the satellite whose data must be monitored by the FSW, reported to ground control

and used in autonomous operations, such as failure response, when necessary. The science components must also be monitored in a similar manner but pose the additional challenge to the FSW of storing data for extended periods of time. Due to limitations on communication, the satellite may only be reachable every few days, so the FSW must also handle storing collected science data and overwriting old data when all storage space has been used. The final hardware component that the FSW must interact with is the EPS board, which handles the power storage and distribution on-board the satellite.

Generally, the goal of the FSW architecture is to have a driver for every hardware component, allowing the software to interface with those components, and manager components that will drive the execution of operations relative to each component. In addition to hardware component managers, global components will be used to monitor the status of all components in the system in order to monitor component health, detect failures and log events. Process priority and proper execution timing will be implemented by the scheduler component.

2.3 Subsystems

There are 10 subsystems in the flight software, and each subsystem has each peripheral value. Each peripheral is listed, and a high-level explanation is included in each subsystem.

Subsystems	Definition	Peripherals
Altitude Control	Attitude control determines the orientation of a spacecraft in a particular direction.	5 Sun Sensors 2 IMU 1 Reaction Wheel DCE 1 Thruster 1 Star Tracker
Fault Protection	Fault Protection checks which processor goes wrong and what hardware problem is issued.	1 Watchdog Timer 1 FPGA
Health Monitor	Health monitor gets the temperature value from peripherals and checks the temperature value is within the threshold range.	Every peripheral

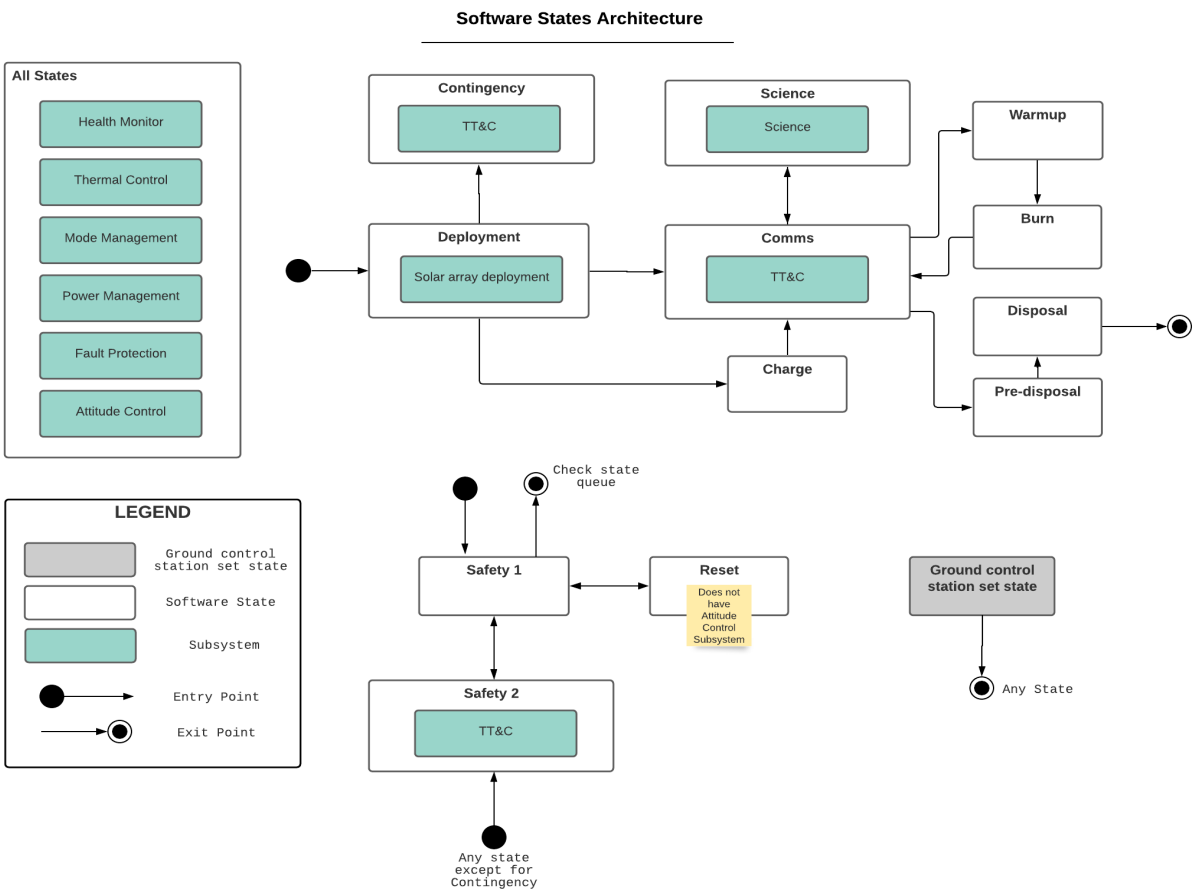
State Management	If a process gets interrupted, the state will be shifted to another state.	Every peripheral
Power Management	The power is generated by the electrical power system (EPS). There are solar panels and batteries in the EPS. 8 batteries are packed in one battery pack.	1 EPS 1 Battery Pack
Propulsion	Propulsion warms up propellent to get the thruster ready.	Thruster
Science	Science gets data and downlink the data.	
Solar Array Deployment	The solar array deployment will send the hold and release mechanism (H&RM) signal, check array release signal, and deploy the three solar arrays.	3 H&RM 1 EPS 1 IMU

2.4 Software States

2.4.1 Introduction

The satellite design consists of 11 software states. These software states allow developers to easily visualize the flow of the software in the satellite. A high-level diagram of these states and how the both states and subsystems are connected are shown in the below architecture diagram. For clarity, the states presented in this satellite design are (in alphabetical order): Burn, Charge, Comms, Deployment, Disposal, Pre-Disposal, RW Desaturation, Safety 1, Safety 2, Reset, and Warmup. A detailed description of each state can be found in the subsequent sections.

Further reference of these diagrams can be found in a document labeled “Software States Documentation”. The purpose of this document is to define abbreviations and expand upon various algorithms. This document is mainly for referential purposes and should be referenced when an aspect of these diagrams isn’t clear.



Architecture for Software States and Subsystems

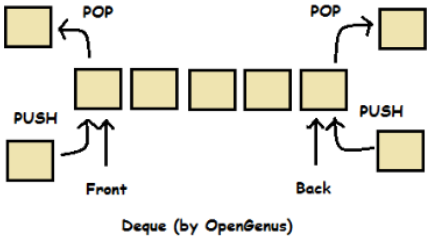
2.4.2 Detailed Description

This section will discuss each state in detail with assumptions with the associated state, as well as how the satellite transitions autonomously from state to state without having to communicate to ground control. Note that the diagram flow will not be explained in detail unless a diagram is not presented.

All entry points into the state are denoted by a solid black dot, whereas all exit points (denoted by a dot with a white outline) must check the state queue (except disposal) before exiting.

2.4.3 State Queue

The way that the satellite transitions from one state to the next autonomously is storing future states and reading a double-ended queue. When the satellite recognizes it needs to go to another state in the future, it will add (push) 1 or more states to the top (front) of the queue. In the event that the state queue is empty, the satellite will communicate with ground control to get the next command.

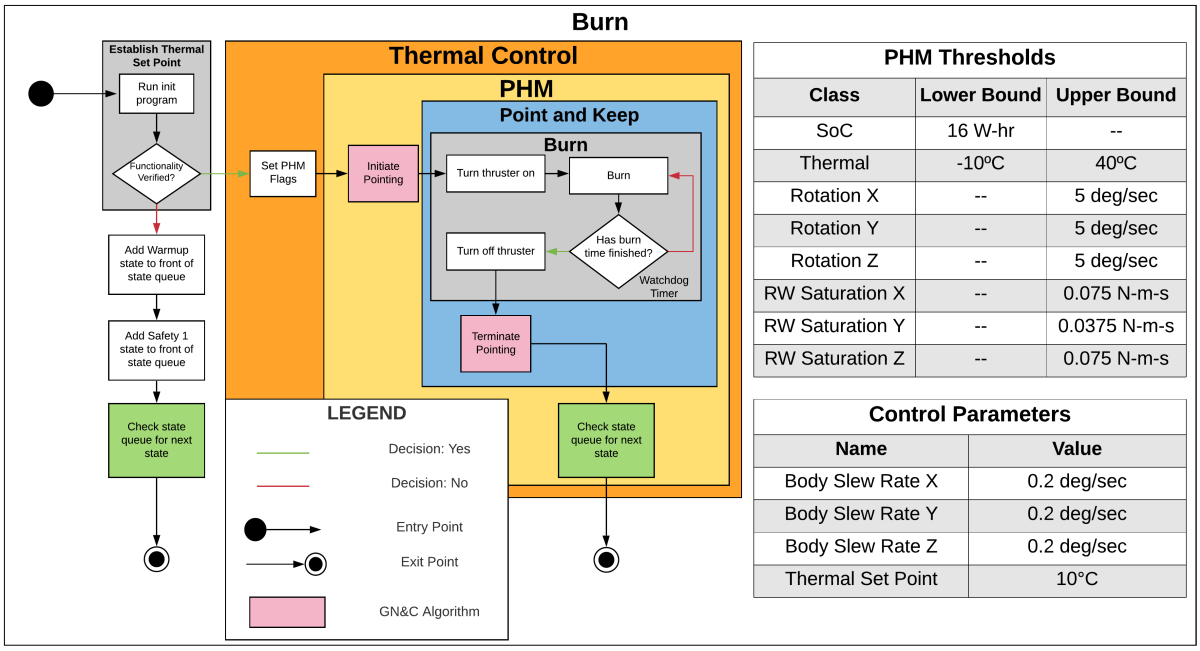


Double-ended queue structure

At the time of writing, the queue only utilizes one end, mimicking a stack data structure. However, future design choices could allow for the other end of the queue to be utilized, thus the choice of calling the data structure a queue was decided upon.

2.4.4 Burn

The burn state is a state in which the thruster is utilized to aid in either movement or detumbling



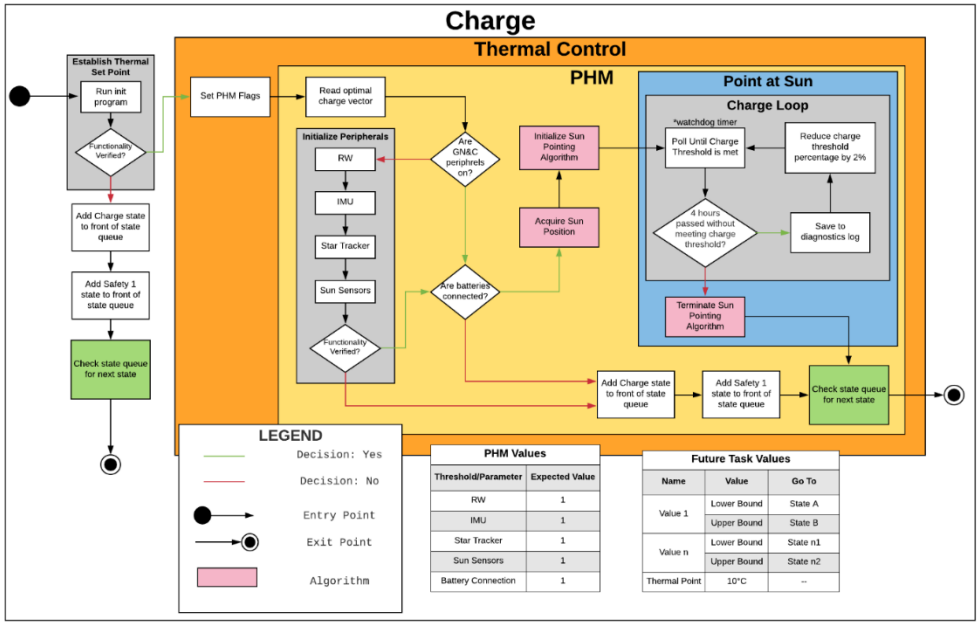
PHM Thresholds		
Class	**Lower Bound**	**Upper Bound**
SoC	16 W-hr	--
Thermal	-10ºC	40ºC
Rotation X	--	5 deg/sec
Rotation Y	--	5 deg/sec
Rotation Z	--	5 deg/sec
RW Saturation X	--	0.075 N-m-s
RW Saturation Y	--	0.0375 N-m-s
RW Saturation Z	--	0.075 N-m-s

Control Parameters	
Name	**Value**
Body Slew Rate X	0.2 deg/sec
Body Slew Rate Y	0.2 deg/sec
Body Slew Rate Z	0.2 deg/sec
Thermal Set Point	10°C

Burn state diagram

2.4.5 Charge

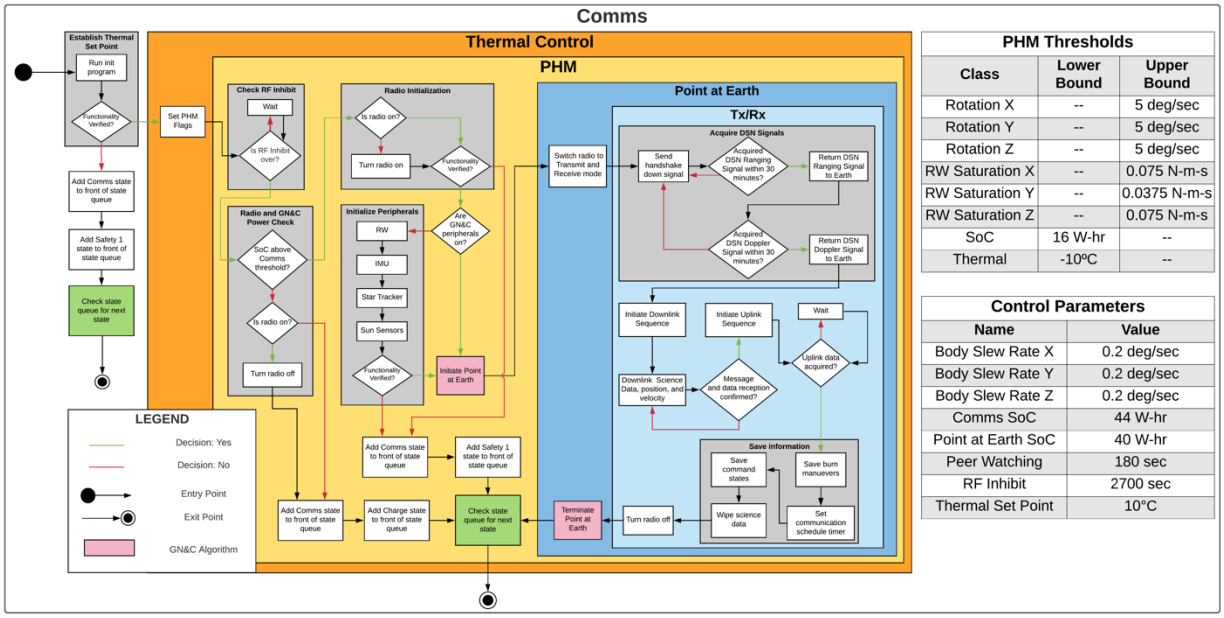
Charge utilizes the peripherals to charge the batteries on the satellite. The satellite will point the solar panels towards the sun and charge until a charging threshold is met.



Charge state diagram

2.4.6 Comms

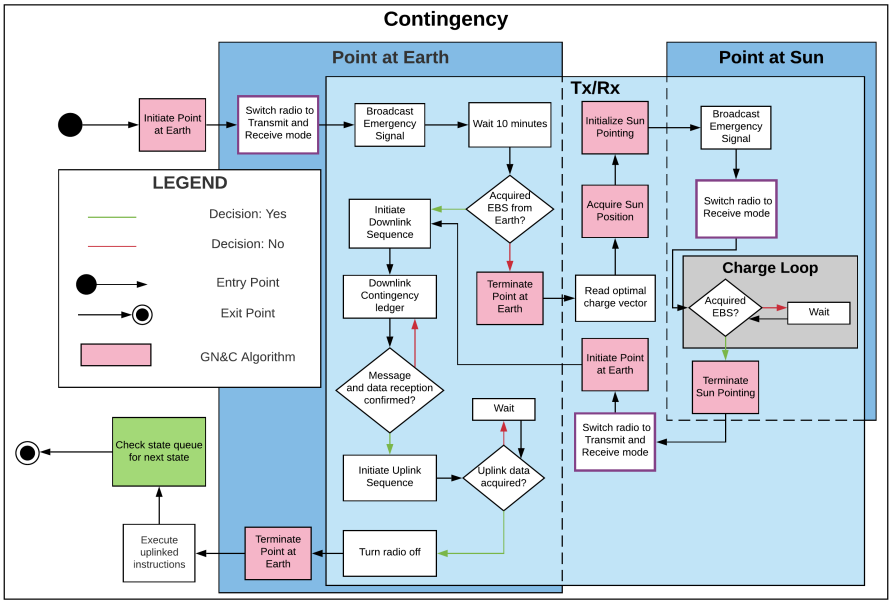
This state establishes communication with ground to uplink/downlink data. Note that work may be needed in the Tx/Rx block, as a handshake signal might not be used.



Comms state diagram

2.4.7 Contingency

Contingency is a unique safety state that tells us that something went wrong with deployment. It is similar to Safety except that it downlinks contingency logs instead of the logs that would be typically used. Note that no other state can access contingency except for deployment.

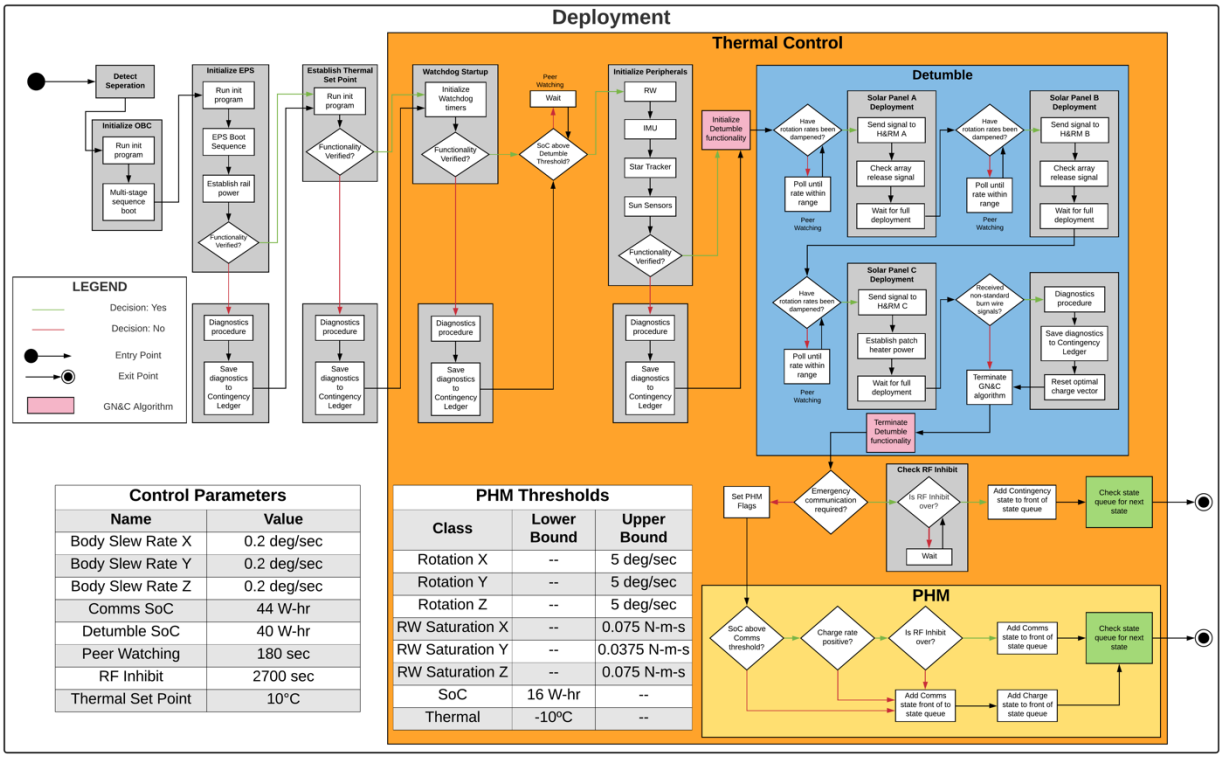


Contingency state diagram

2.4.8 Deployment

Deployment is only accessed once, and that is when the satellite is separated from the parent spacecraft. This state instantiates the hardware on the satellite and reduces any rotational velocities it may be experiencing and deploys the solar arrays.

Deployment can be the most confusing out of all of the software state diagrams, as it is different than the rest with respect to the PHM, as well as the safety state choice. Instead of going into Safety 2 in the event emergency communications are needed, it goes into Contingency. This is because in order to access Safety 2 the satellite must go through Safety 1. Given the nature of Safety 1, it simply doesn't make sense to go into Safety 1.



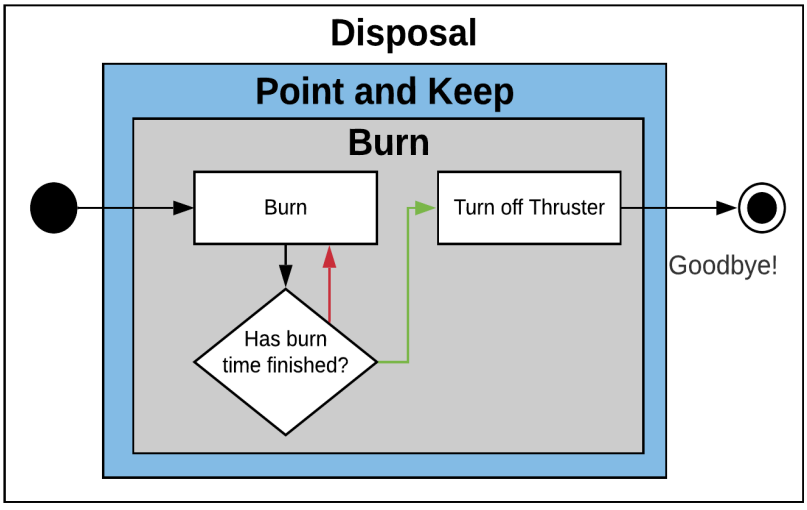
Control Parameters	
Name	**Value**
Body Slew Rate X	0.2 deg/sec
Body Slew Rate Y	0.2 deg/sec
Body Slew Rate Z	0.2 deg/sec
Comms SoC	44 W-hr
Detumble SoC	40 W-hr
Peer Watching	180 sec
RF Inhibit	2700 sec
Thermal Set Point	10°C

PHM Thresholds		
Class	**Lower Bound**	**Upper Bound**
Rotation X	--	5 deg/sec
Rotation Y	--	5 deg/sec
Rotation Z	--	5 deg/sec
RW Saturation X	--	0.075 N-m-s
RW Saturation Y	--	0.0375 N-m-s
RW Saturation Z	--	0.075 N-m-s
SoC	16 W-hr	--
Thermal	-10ºC	--

Deployment state diagram

2.4.9 Disposal

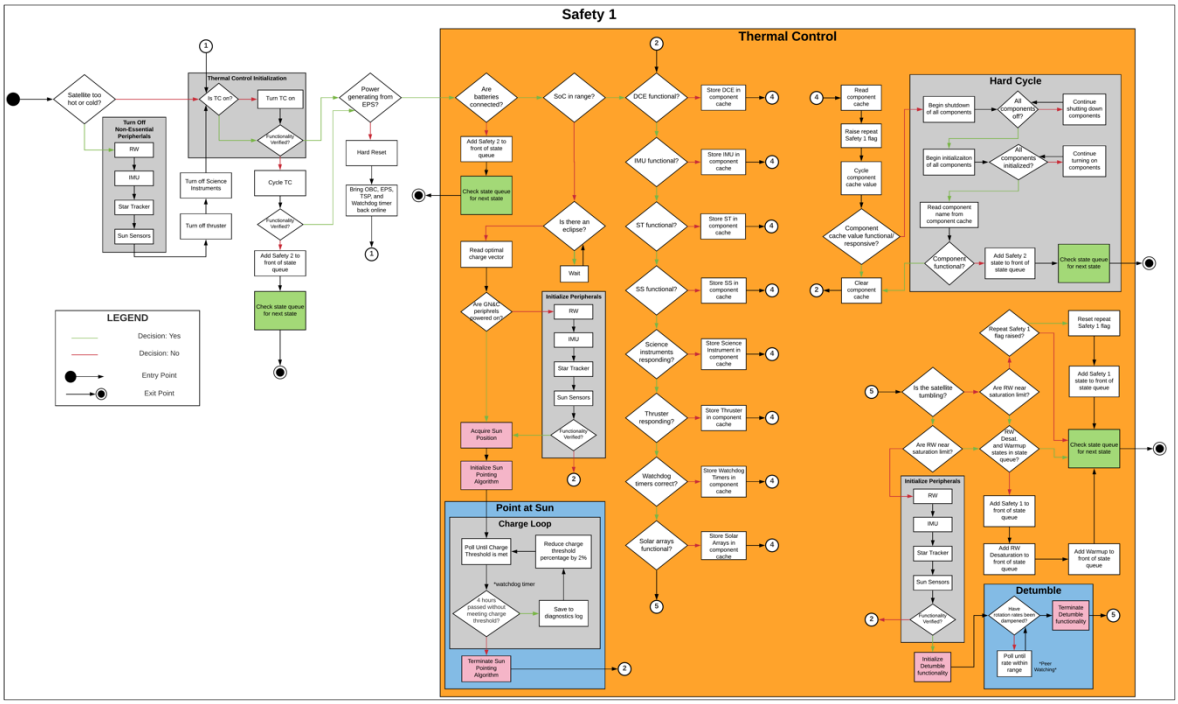
Disposal is when the satellite needs to be discarded. **THERE IS NO WAY TO EXIT OUT OF THIS STATE.** In this state, the satellite warms up the propellent, then burns and goes in a direction, completely discarding it.



Disposal State Diagram

2.4.10 Safety 1

Safety 1 is a fully autonomous self-diagnosing and self-fixing state. It runs through all of the hardware from a thermal and electrical perspective, and then determines the best course of action. Often times, the hardware just needs to be reset (cycled). In the event it can't fix itself, then it goes into Safety 2.



Safety 1 state diagram

It should be noted that everything is logged after each decision diamond. 2 example safety logs are shown below, one successfully passing every check and the other being sent into Safety 2 due to a failed post-hardware cycle check.

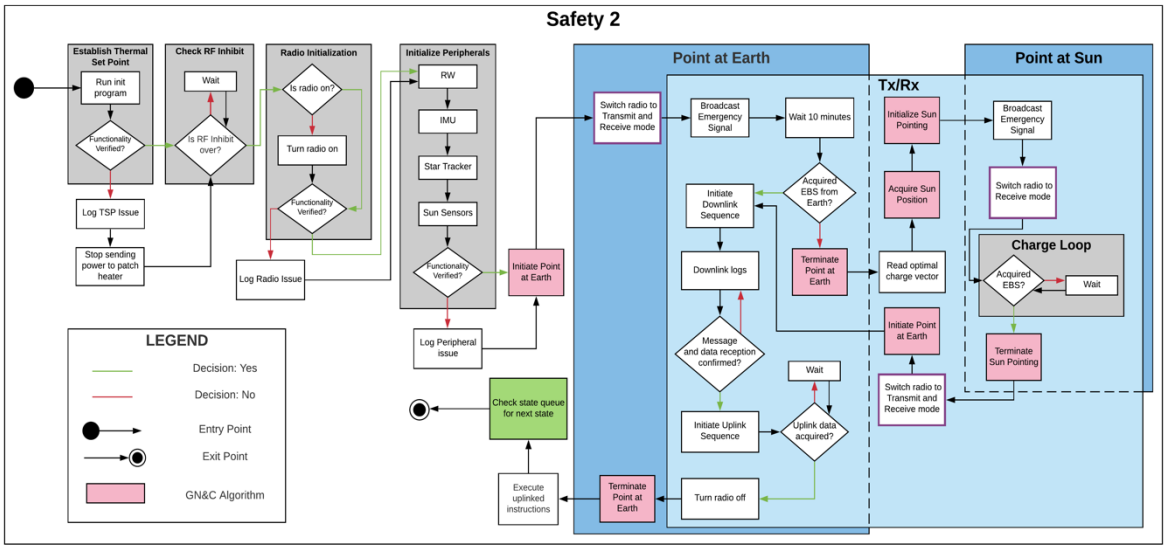
Safety 1 Verification: All OK	
Check	OK?
Satellite temperature check	OK
TC functionality check	OK
Power generation from EPS check	OK
Batteries Connected	OK
SoC in range	OK
DCE functionality check	OK
IMU functionality check	OK
ST functionality check	OK
SS functionality check	OK
Science instrument check	OK
Thruster Response	OK
Watchdog Timer Check	OK
Solar array functionality check	OK
Tumble threshold check	OK
RW saturation check	OK
Repeat Safety 1 flag check	OK
Exiting Safety 1, return nominal status.	

Safety 1 Verification: Safety 2 post hardware cycle.	
Check	OK?
Satellite Temperature	OK
TC functional?	OK
Power generation from EPS	OK
Batteries Connected	OK
SoC in range	OK
DCE functionality check	ERROR
Cycling DCE	
DCE functionality check post cycle	ERROR
Enter Safety 2, DCE functionality check post cycle failed.	

If Safety 2 is entered, this log will be sent down to ground control once communications are established with the ground. One aspect a future team can work on is what these logs will look like and how they are going to be stored in memory before transmission.

2.4.11 Safety 2

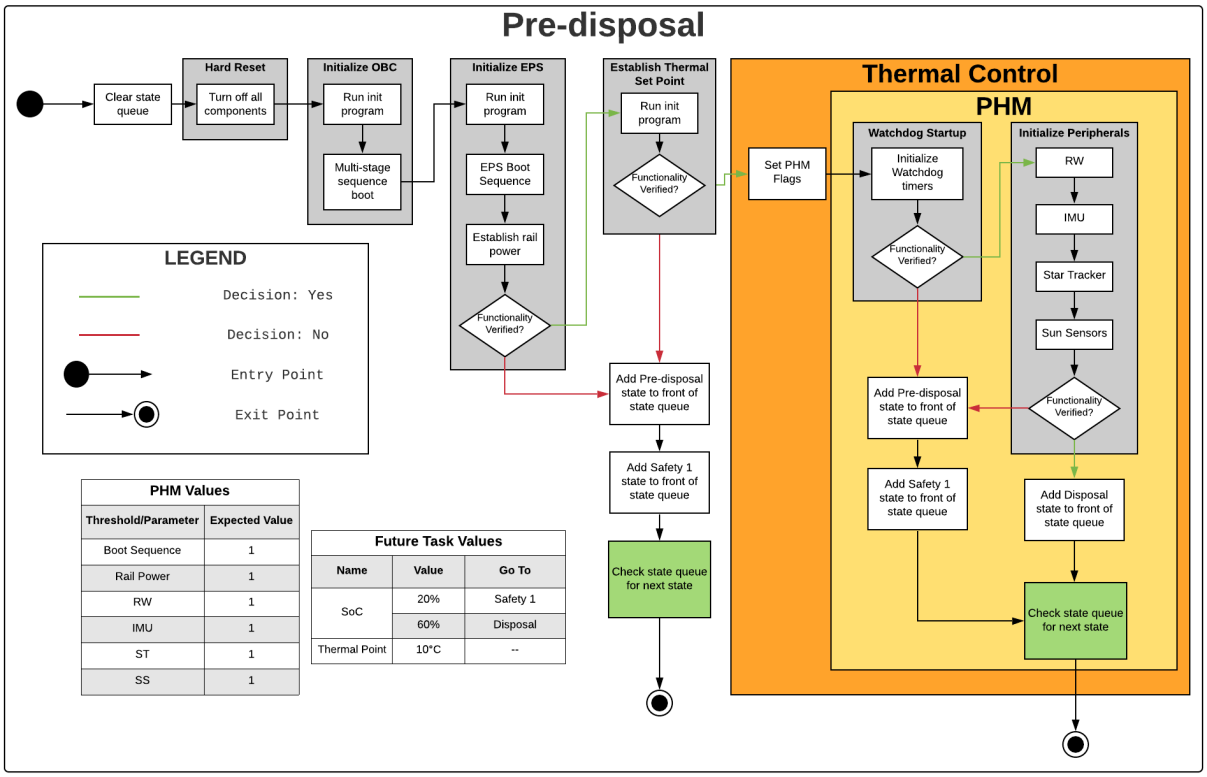
This state can only be accessed through Safety 1, and only occurs when something is truly wrong. It "panics" and send out a broadcast signal for human help. Some entity (whether if this would be ground, DSN, or another spacecraft/satellite) will hopefully pick up this signal and relay it to ground, which then will diagnose the error and fix it appropriately.



Safety 2 state diagram

2.4.12 Pre-disposal

This state is preparing for disposal. **IT IS IMPORTANT TO NOTE THAT THIS MEANS THE SATELLITE WILL BE DISCARDED.** It clears the state queue so it doesn't enter any other state other than what is specified.



Pre-disposal state diagram

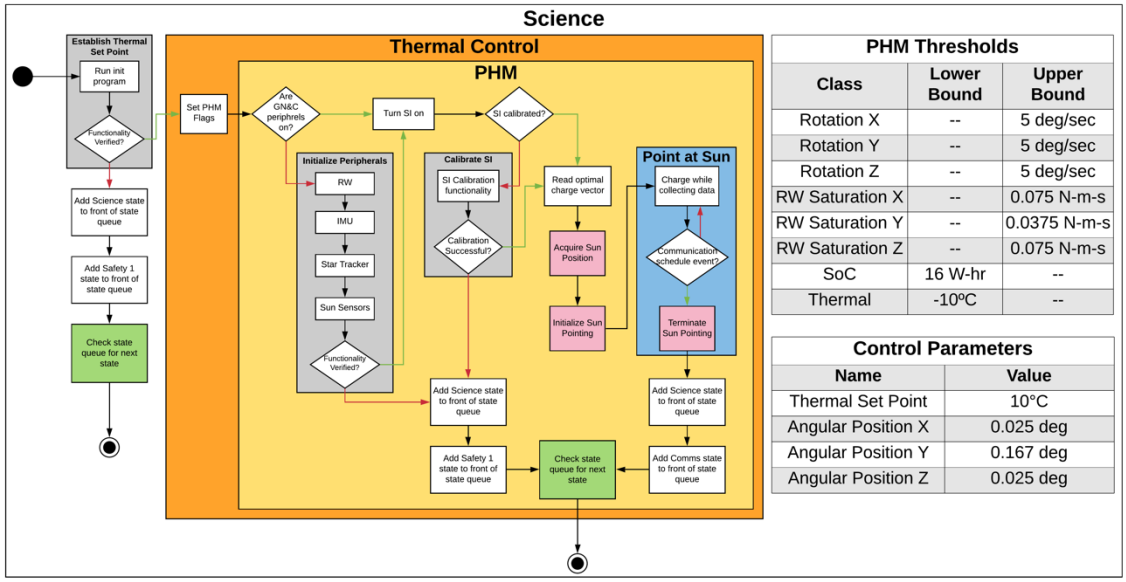
Future implementation: double check by ground control to ensure that the satellite really should be disposed of.

2.4.13 RW Desaturation

At the time of writing, there is no diagram for this state. RW Desaturation is a one of the more, if not most, complicated state, as it is a state that desaturates the reaction wheels. Not much information can be documented about this state as it has not been discussed amongst the team.

2.4.14 Science

At the time of writing, this state does not have a diagram. The Science state is the state that the satellite will be in for most of its lifetime. It collects data about the environment for the given mission, then saves it in memory and switches to Comms to relay that information back to ground so it doesn't get lost/corrupted while in space.



PHM Thresholds

Class	Lower Bound	Upper Bound
Rotation X	--	5 deg/sec
Rotation Y	--	5 deg/sec
Rotation Z	--	5 deg/sec
RW Saturation X	--	0.075 N-m-s
RW Saturation Y	--	0.0375 N-m-s
RW Saturation Z	--	0.075 N-m-s
SoC	16 W-hr	--
Thermal	-10ºC	--

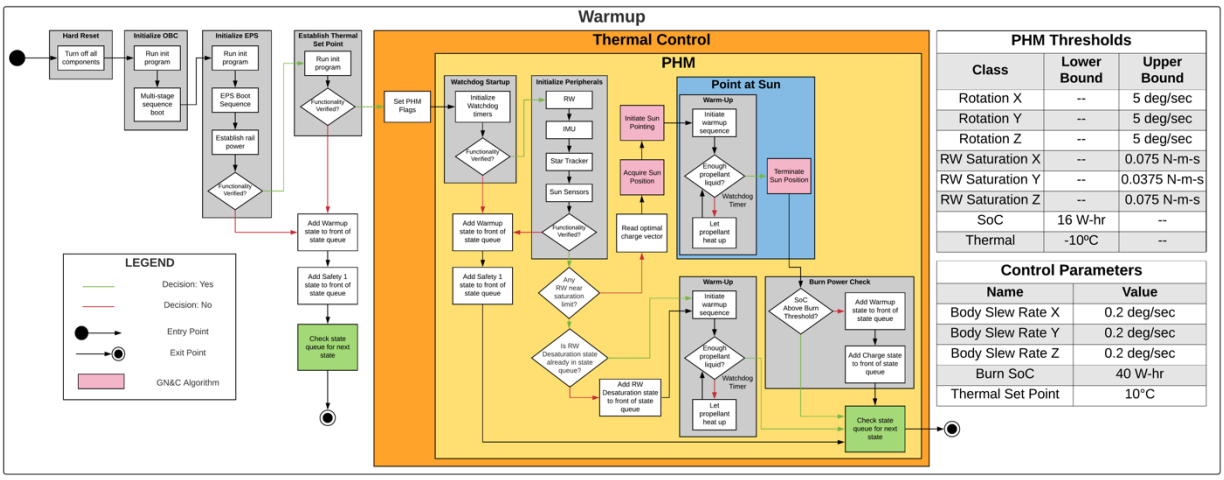
Control Parameters

Name	Value
Thermal Set Point	10°C
Angular Position X	0.025 deg
Angular Position Y	0.167 deg
Angular Position Z	0.025 deg

Science state diagram

2.4.15 Warmup

The final state in this section heats up the propellent to get ready to perform a burn maneuver. This state points towards the sun to warm up the propellent.



Warmup state diagram

2.5 Future Considerations for Software States

Future considerations for updating and maintaining these diagrams are discussed in this section. This was included to provide a starting point next group(s) to build upon.

- If the implementation of a stack continues, then the queue should be renamed to a stack across all diagrams. It should be pushed to keep a stack implementation, as the implementation from a coding perspective is easier than a double-ended queue (less human-made errors while developing the implementation).
- How are data errors handled?
- How can data flow be represented?
- The PHM and Control Parameters should be revisited and revised.

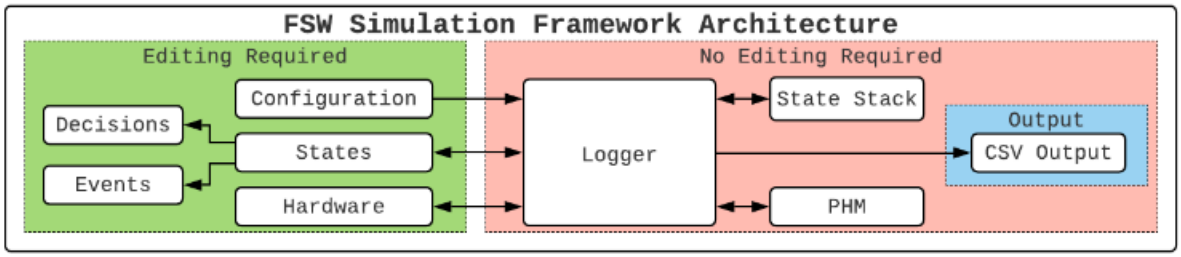
3 Software States Simulation Tool

During development of the software state diagrams, a framework for testing the logic flow of the diagrams have been developed. The purpose of this framework is to allow the user to create an environment where they can test the FSW design with ease. It is used to better understand their system, identify redundancies, and to see what could go wrong within the design when scaled.

For documentation regarding this, please see the FSW simulation document, as well as the associated GitHub.

3.1.1 Software State Testing

These software state systems will be tested by the state diagram simulation. This simulation is a copy of the logic from each diagram written in Python. The advantage to using this system is that it allows one to test the logic from the diagram before it is fully built into the final system.



Understanding this architecture is critical to understand how all of the components (next section) are connected. The above figure depicts the architecture of how the FSW simulation is structured with respect to what the user will edit when constructing their FSW design. There are editable components and non-editable components of the framework. Some of these components are built directly into the framework, whereas others the user may have to stub out using built-in scripts. The user shouldn't have to add on self-built components to the framework unless it's absolutely needed. A more detailed description of each component is described in the next section. In short, the user edits the states component, which encapsulates states; this is user defined. This is the driver code for the simulation with respect to the specific state behavior and should follow very closely to what the state diagrams represent. This component will call a decisions and events class, where the user will write the functionality of the decision or event(the boxes and diamonds found on the state diagrams). The hardware component of the framework is intuitive: it's the hardware that's found on the satellite and whether if it's on or off. Finally, there's a configuration file to set parameters before the simulation begins. The non-editing components include the logger, which is a super class to the entire system, a state stack to transition from state to state, a PHM (Prognostic Health Management), and the output functionality.

4 Cameo Enterprise Architecture

4.1 Overview

The cameo program allows us to use an auto generating system feature after installing the plug-in. Before using the autogenerating feature, it is required to create ports, components, and topology packages.

Port, component, and topology packages are needed to create each element in their packages. New Port Type Component can be created with value property element(s). in a port package(s). Active/Passive/Queued components can be created with Synchronous/Asynchronous/Guarded ports in a component package(s). Subsystems with the Internal Block Diagram (IBD) are in a topology package(s).

There are pre-defined F Prime components in the F Prime Source Tree. Own components have to be created depending on the design. If ports are correctly connected between pre-defined and own components in every Internal Block Diagram (IBD) in a topology package, xml files will be successfully generated. The xml specification will be used to generate C++ files.

4.2 Using Cameo

One way to use the Cameo Enterprise Architecture program is physically going to the lab because the program is a licensed program and installed in the school lab computer. Another way to use the program is using an application called Microsoft Remote Desktop to remotely access to the lab computer.

4.3 Pre-defined ports

Port Type	XML File	Description
Commands		
Command	Fw/Cmd/CmdPortAi.xml	A port that passes a serialized command to a component.
Command Response	Fw/Cmd/CmdResponsePortAi.xml	A port that passes the completion status of a command.
Command Registration	Fw/Cmd/CmdRegPortAi.xml	A port used to request registration of a command. Used during initialization to tell a command dispatcher where to send specific opcodes.
Telemetry		
Telemetry	Fw/Tlm/TlmPortAi.xml	A port that passes a serialized telemetry value.
Time	Fw/Time/TimePortAi.xml	A port that returns a time value for time stamping the telemetry.
Events		
Log	Fw/Log/LogPortAi.xml	A port that passes a serialized event.
LogText	Fw/Log/LogTextPortAi.xml	A port that passes the text form of an event. Can be disabled via configuration of the architecture.
Time	Fw/Time/TimePortAi.xml	A port that returns a time value for time stamping the telemetry.
Parameters		
Parameter	Fw/Prm/PrmPortAi.xml	A port that returns a serialize parameter value

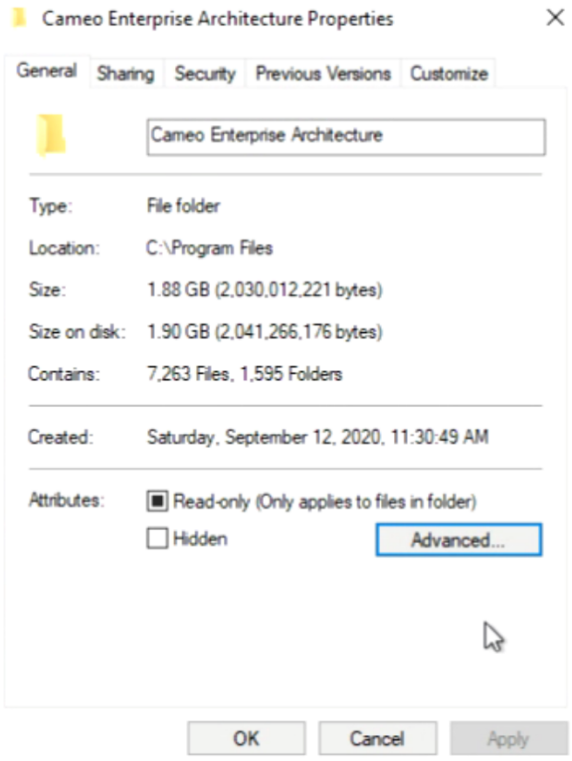
This table is in the F' User's Guide [4]. There are three command ports, 1 telemetry, 1 time, 2 log, and 1 parameter ports that are commonly used to connect a component(s) to pre-defined components. Besides the information of the pre-defined ports, the F' User's Guide also has information regarding pre-defined components.

4.4 Port Permission Problem

```
[2020.09.30::11:20:05] Writing new file: C:\Users\cubesat\fprime-aegis\Ref/AutoXML/BufferManagerComponentAi.xml
[2020.09.30::11:20:05] Writing new file: C:\Users\cubesat\fprime-aegis\Ref/AutoXML/HealthComponentAi.xml
[2020.09.30::11:20:05] Writing new file: C:\Users\cubesat\fprime-aegis\Ref/AutoXML/AltitudeReqPortPortAi.xml
[2020.09.30::11:20:05] ===> Problem with getting resource:Failed to initialize an instance of org.apache.velocity.runtime.log.Log4JLogChute with the current runtime configuration.
[2020.09.30::11:20:05] Writing new file: C:\Users\cubesat\fprime-aegis\Ref/AutoXML/IMUPortPortAi.xml
[2020.09.30::11:20:05] ===> Problem with getting resource:Failed to initialize an instance of org.apache.velocity.runtime.log.Log4JLogChute with the current runtime configuration.
[2020.09.30::11:20:05] Writing new file: C:\Users\cubesat\fprime-aegis\Ref/AutoXML/RWPortPortAi.xml
[2020.09.30::11:20:05] ===> Problem with getting resource:Failed to initialize an instance of org.apache.velocity.runtime.log.Log4JLogChute with the current runtime configuration.
[2020.09.30::11:20:05] Writing new file: C:\Users\cubesat\fprime-aegis\Ref/AutoXML/SSPortPortAi.xml
[2020.09.30::11:20:05] ===> Problem with getting resource:Failed to initialize an instance of org.apache.velocity.runtime.log.Log4JLogChute with the current runtime configuration.
```

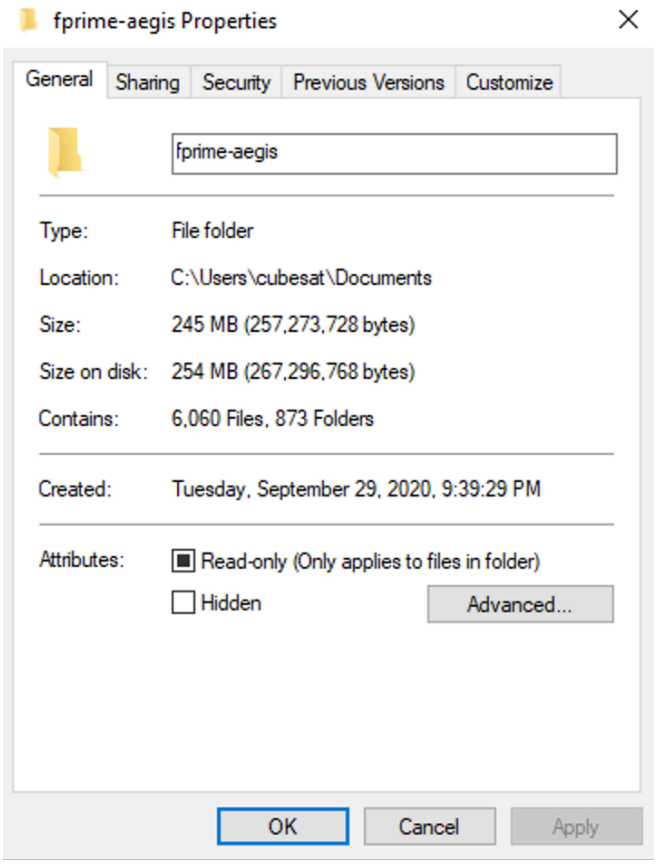
There was an unknown problem with "===> Problem with getting resource:

Failed to initialize an instance of org.apache.velopcity.runtime.log.Log4JLogChute with the current runtime configuration" message.

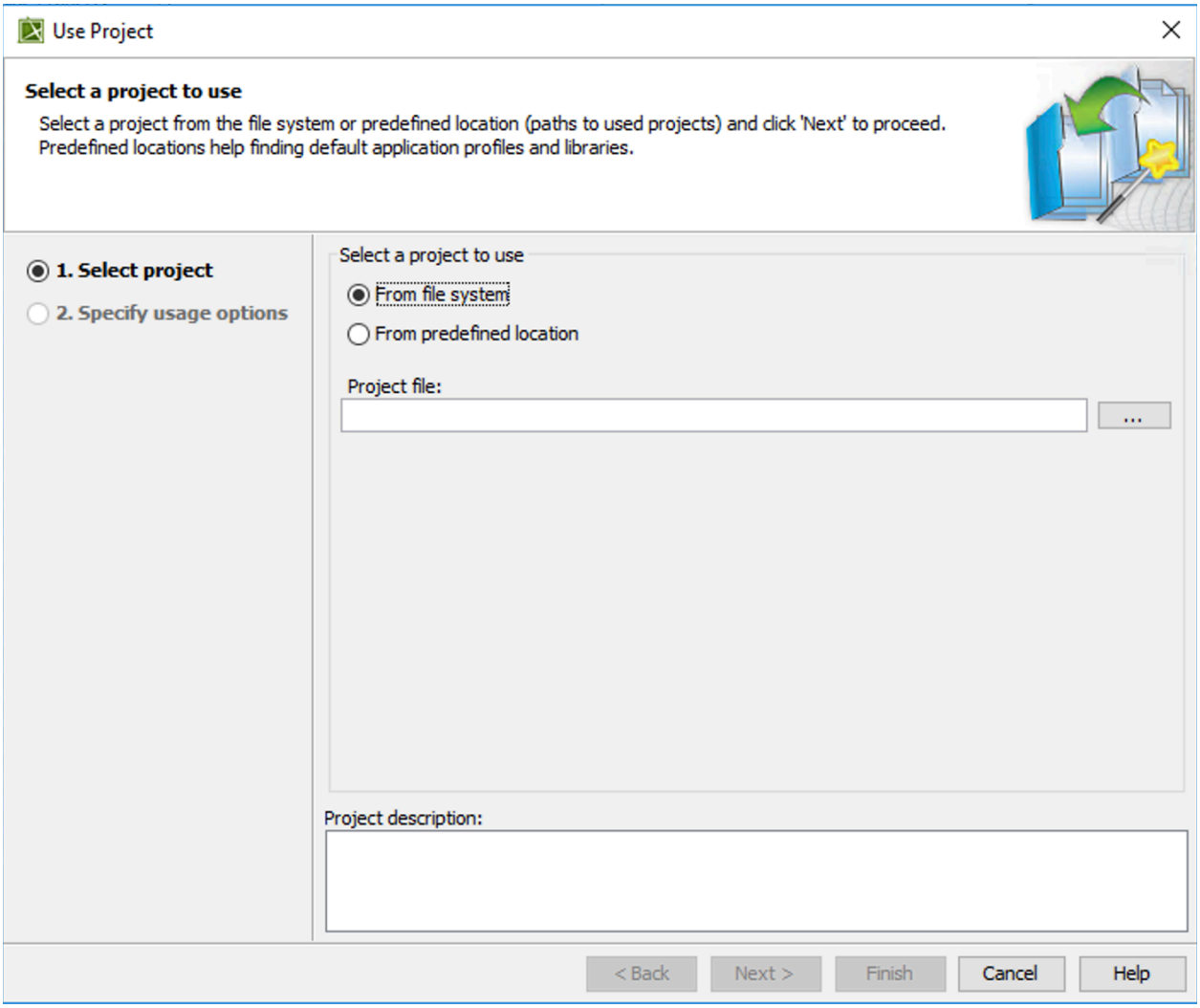


This problem was caused be the permission of the permission of the Cameo Enterprise Architecture program folder. The Read-only box of the Cameo folder and F Prime Source Tree folder should be unchecked from the admin account. Also, the F Prime folder needs to be in a directory where the directory overwrite stuff. Then, the port error was gone.

4.5 Project File Permission Change



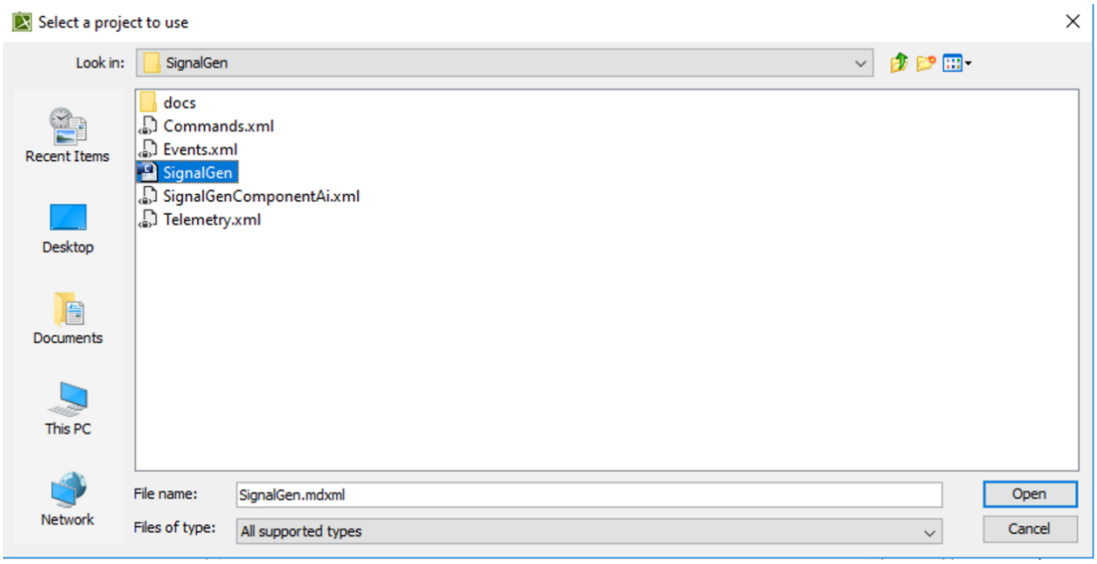
The black square box can still be existed when checking the properties of the folder. The black square means there could be some read-only files and none read-only files.



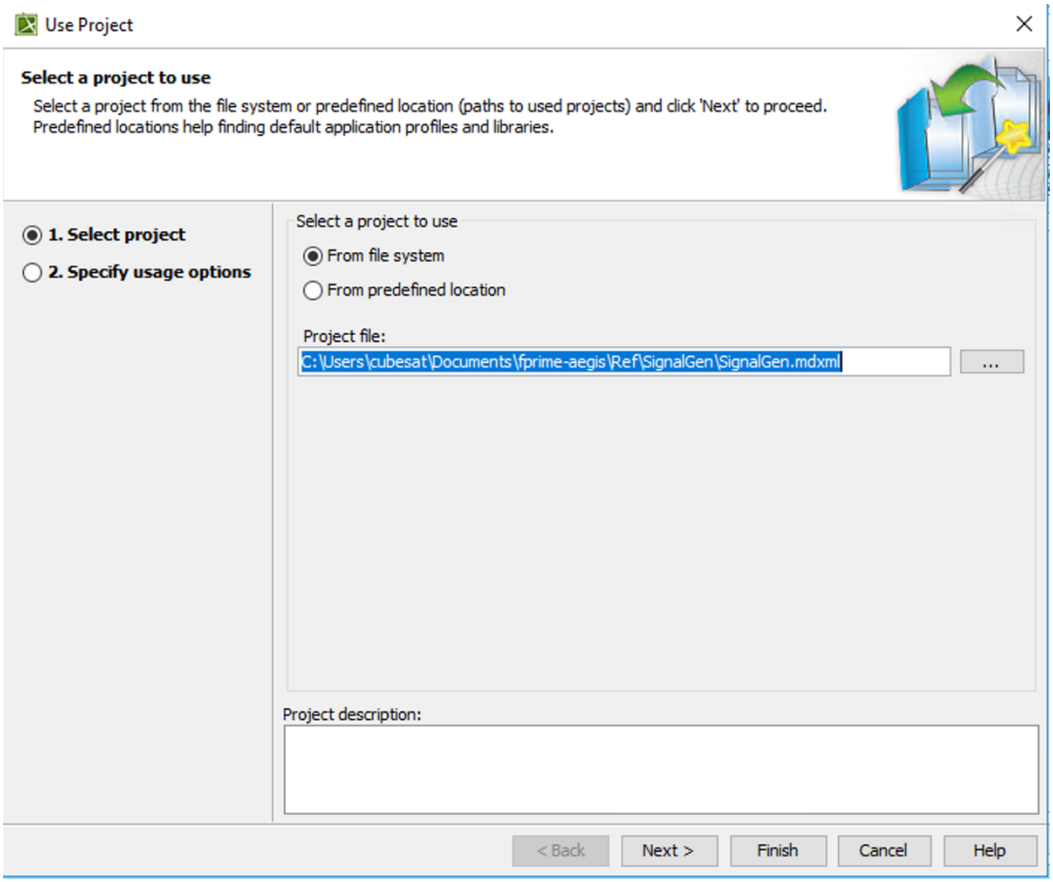
If a certain component needs to be imported or change the permission of the

file, the Use Project under Option menu can be clicked, and this window will pop up.

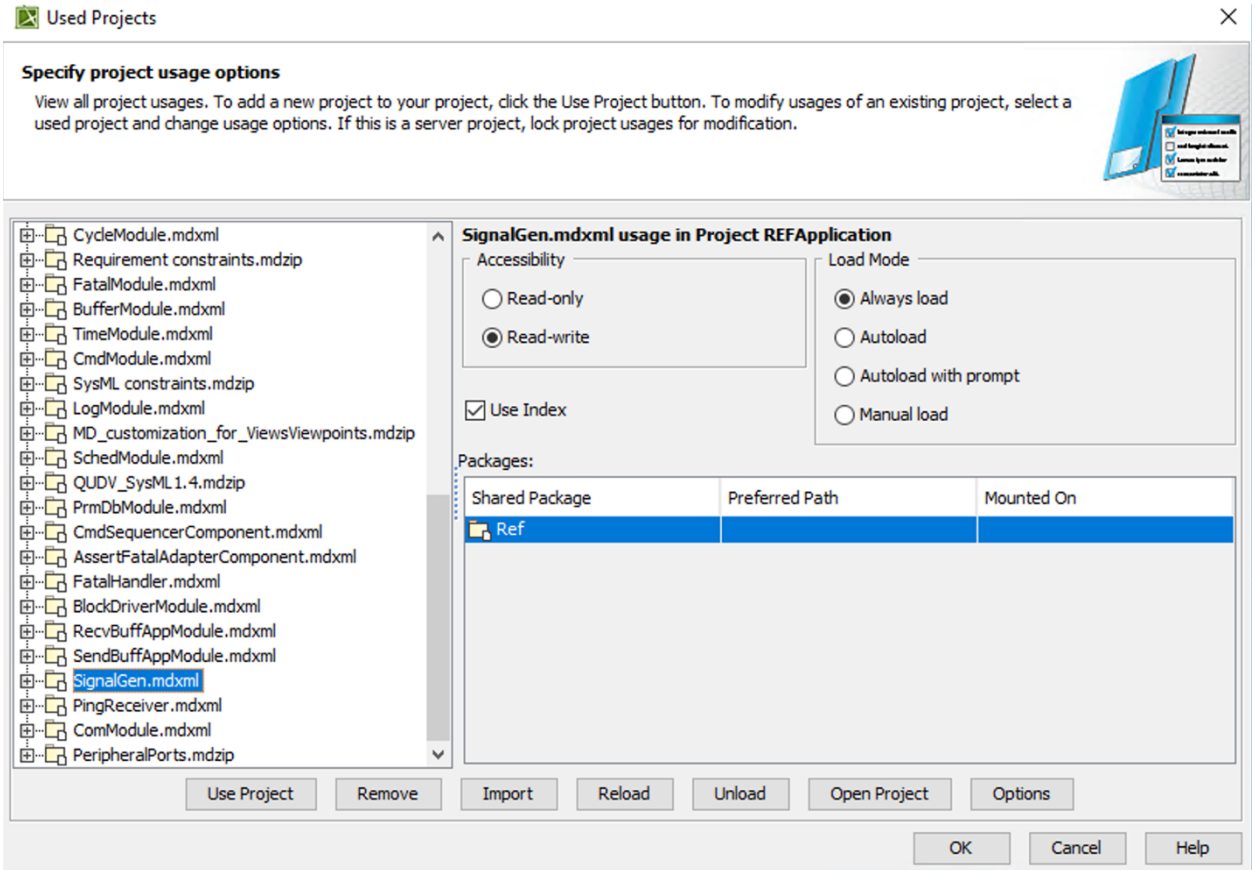
To import a project, first select the ‘From file system’ under ‘Select a project to use’ and click browse button (the box with three dots).



Second, select a specific project file that will be used and click Open.

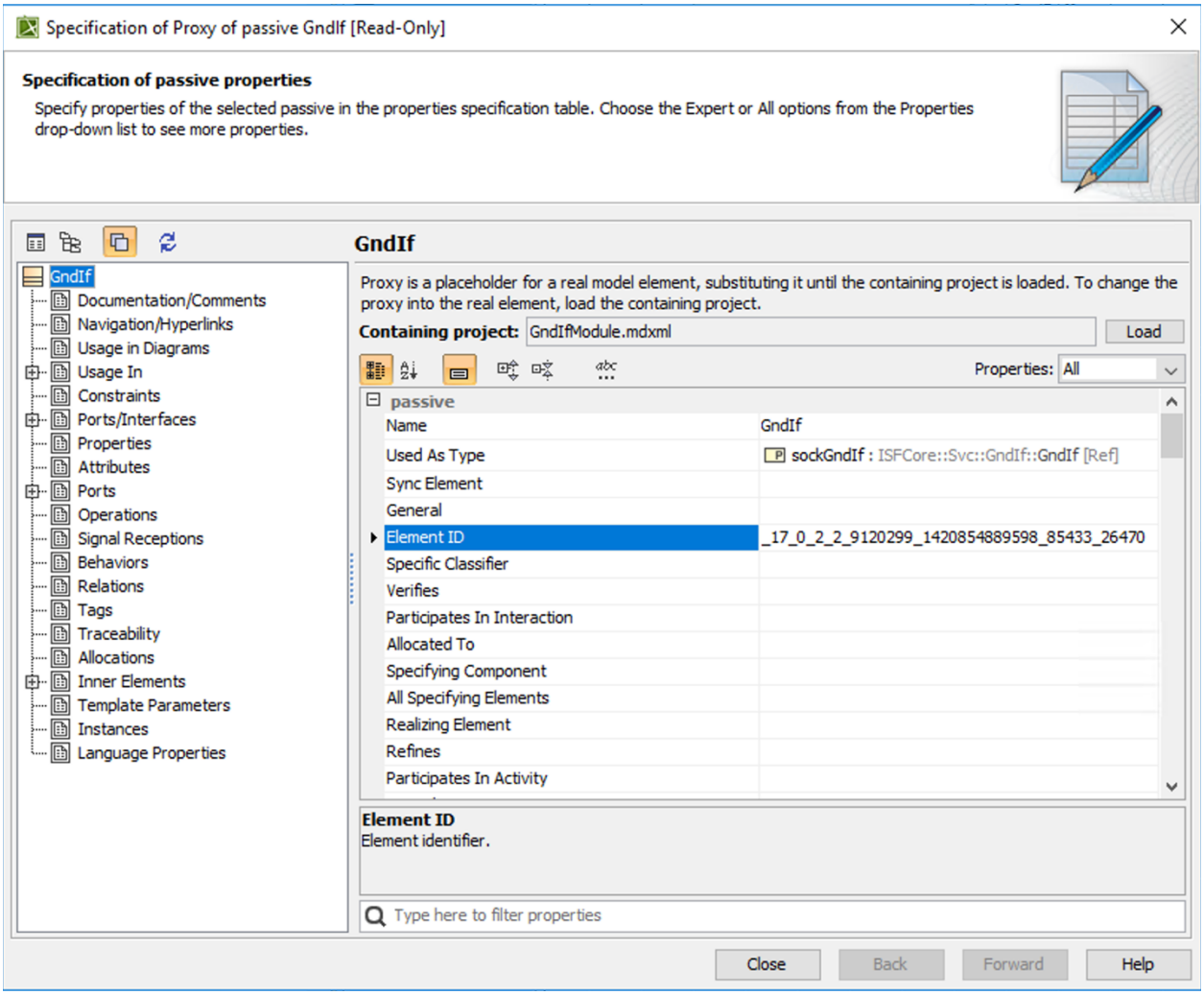


After clicking Open button, you will see this window with the current directory of the project file.



Then, the ‘Accessibility’ of project files. project files can be changed by selecting a different option. Some pre-defined project files can be changed by selecting Read-only box, but some files are not allowed to change the Accessibility.

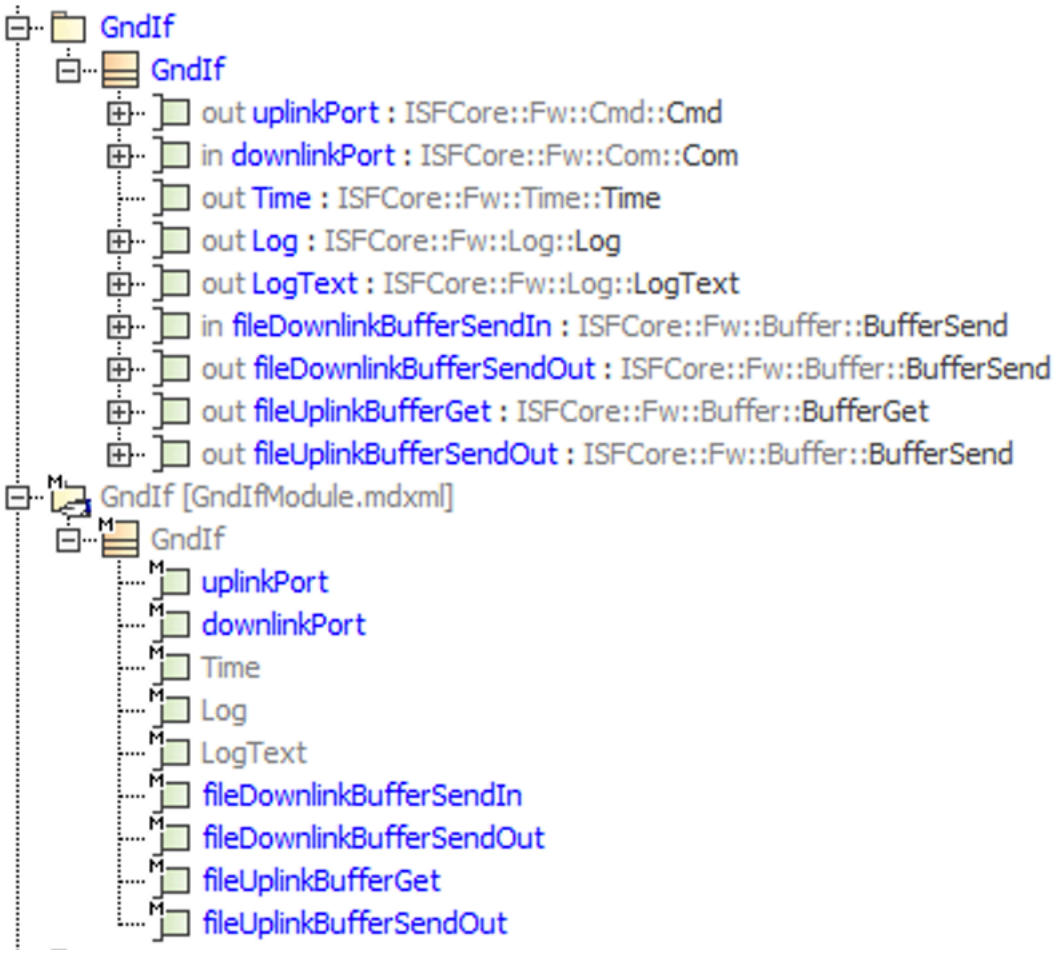
4.6 Pre-defined Component Absence Problem



The pre-defined GndIf project file was absent in the Svc directory, and the permission of this project file was not able to be changed.

```
[2020.10.10::20:23:48] Incomplete Installation More...
[2020.10.10::20:23:54] My Project Name = REFApplication
[2020.10.10::20:23:54] Parsing model for Components
[2020.10.10::20:23:54] ==> Warning no stereotype:Active Component
[2020.10.10::20:23:54] ==> Warning no stereotype:Passive Component
[2020.10.10::20:23:54] ===> FATAL Exception: Port uplinkPort of component GndIf has no data type
```

Therefore, when using the auto-generation feature of the Ref topology, there is an error message like the above. If this file needs to be used, it is required to recreate by looking at the connection in IBD diagrams in a Ref topology package.



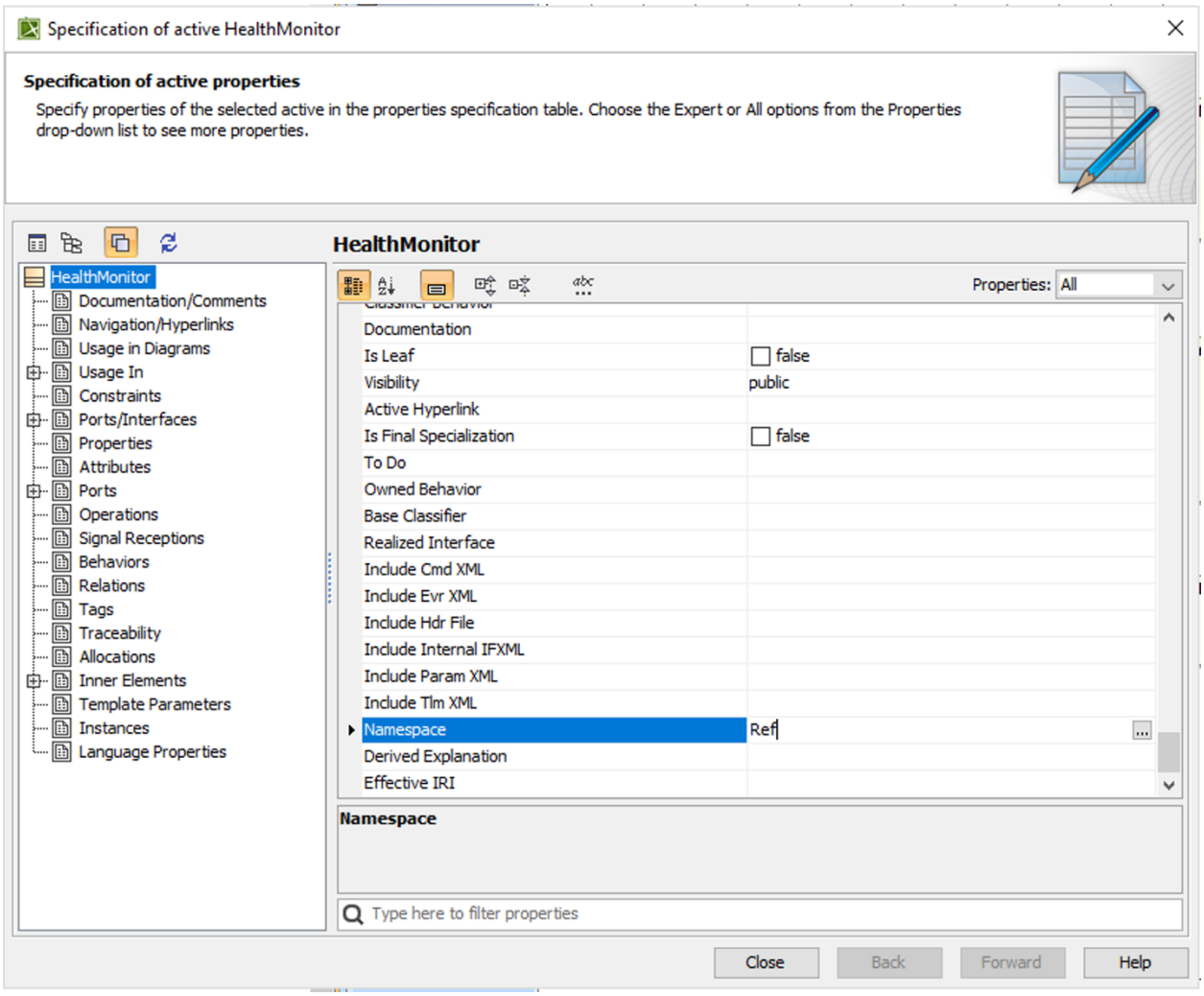
GndIf component and ports with data type are created in a GndIf package. The bottom GndIf package is without data types and the top GndIf package is with data type.

The new GndIf component must be reconnected with other components through the ports in IBDs in the Ref topology package. Then, the previous GndId component can be deleted in IBDs.

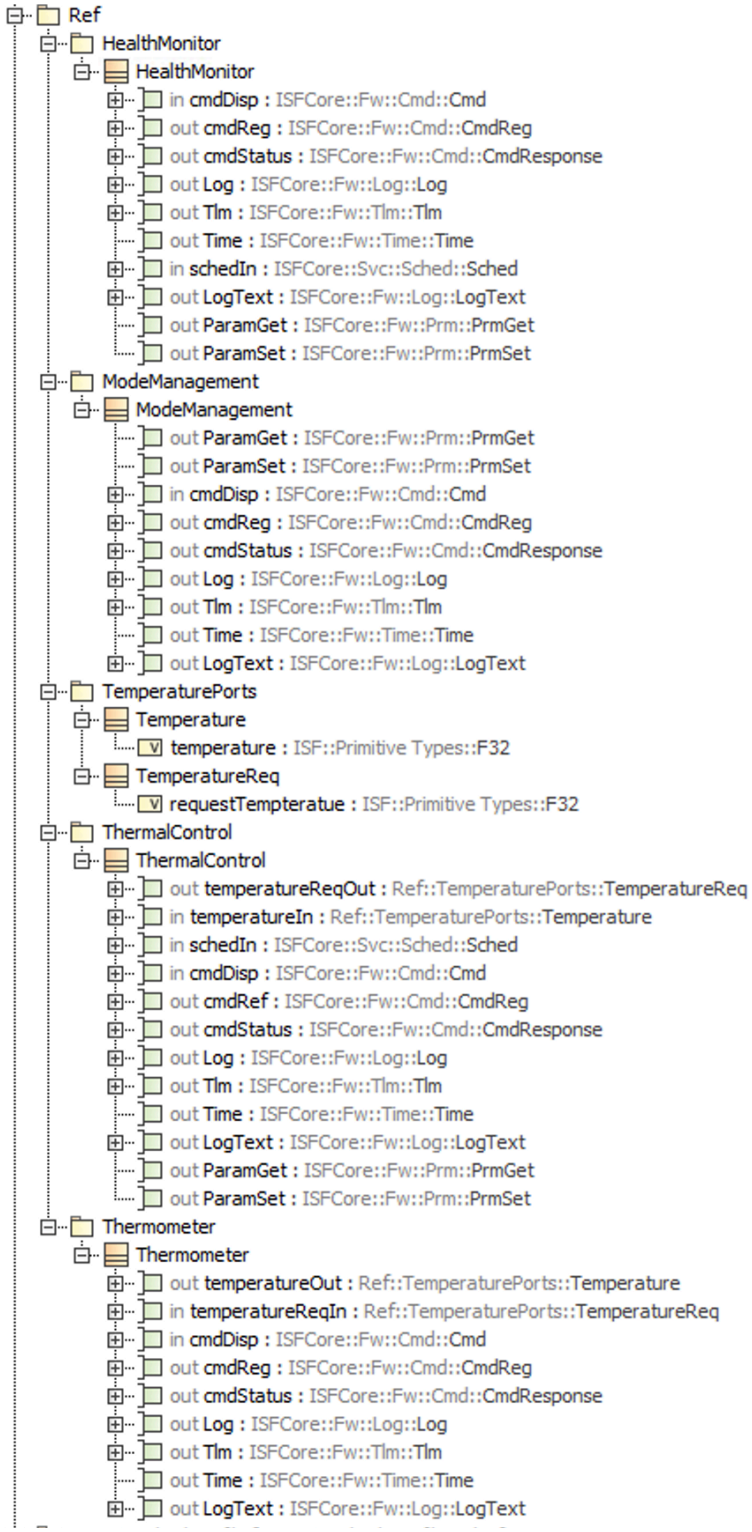
4.7 Creating Topology Diagrams

4.7.1 IBD Creating Port and Component Packages

To make topology diagrams by using the RefApplication project file, go to the Ref/Top directory in the F Prime Source Tree, and open the RefApplication project file.



When creating a new port type or component, it is required to enter ‘Namespace’ which is Ref in this case.

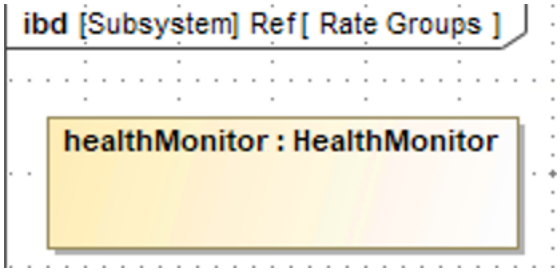


To create topology diagrams, component and port packages must be created. There are 4 component and 1 port packages under the Ref package. Under component packages, every port is pre-defined ports except for the ports with Temperature and TemperatureReq data type.

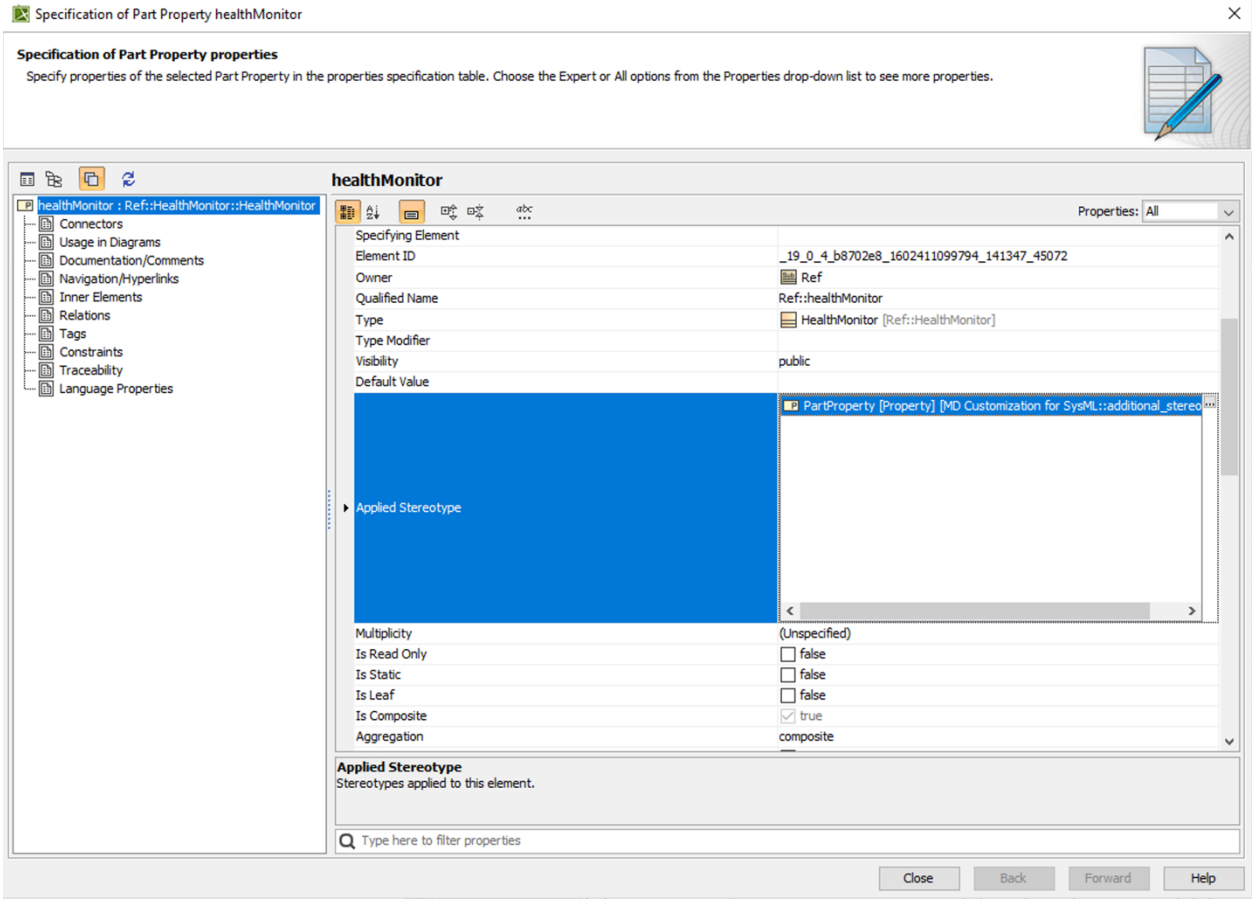
4.7.2 Component Instance Stereotype and Tags

Tag	Attribute	Description
deployment		The outermost tag the indicates that a topology is being defined
assembly		Alternate declaration for "deployment"
deployment	name	The name of the deployment
import_component_type		Imports the XML file that defines a component used in the topology.
instance		Defines a component instance.
instance	name	Name of the component instance. This instance name must match a component object declared in the C++ code.
instance	namespace	C++ Namespace of component implementation type.
instance	type	C++ type of implementation class
instance	base_id	The starting ID value for commands, events and telemetry for this instance of the component. Used to construct dictionary for ground system.
instance	base_id_window	A bookkeeping attribute that the modeler uses to space out the base_id values. It can be omitted if the base_ids are spaced enough to cover the id range in the component.
connection		Defines a connection between two component ports
connection	name	Name of connection
source		Defines the source of the connection
source	component	Defines source component. Must match an instance in instance section above.
source	port	Defines source port on component
source	type	Source port type
source	num	Source port number if multiple port instances
target	component	Defines target component. Must match an instance in instance section above.
target	port	Defines target port on component
target	type	Target port type
target	num	Target port number if multiple port instances

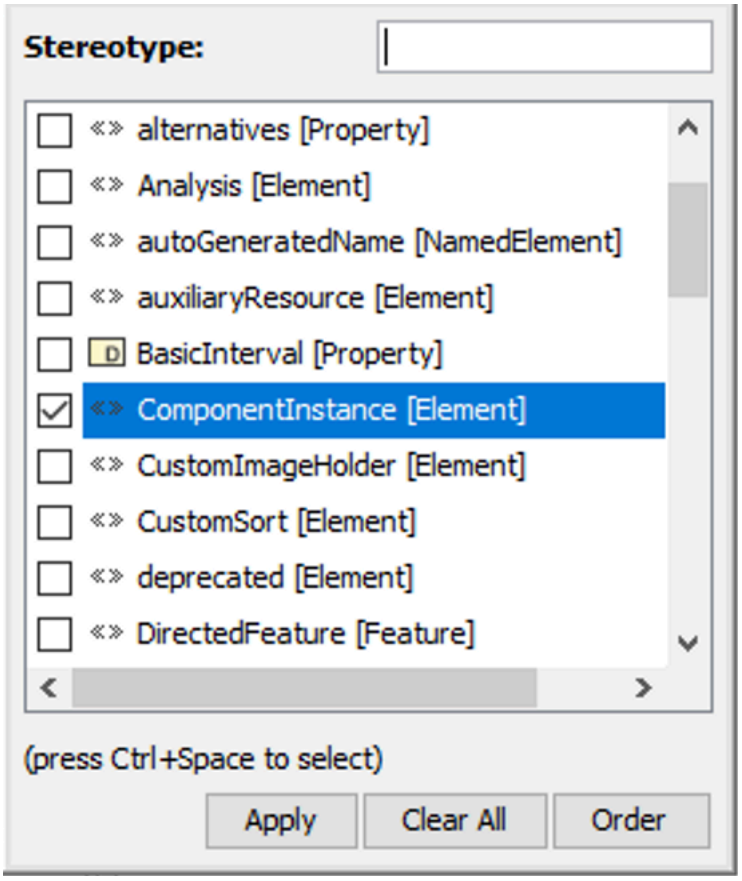
This table is in the F' User's Guide[4]. In this section, the following instructions will show how to create the base id for a component.



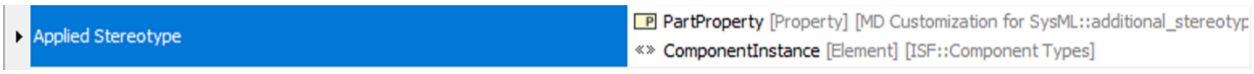
To connect the healthMonitor component to other components, the component needs to be dragged into one of the IBDs.



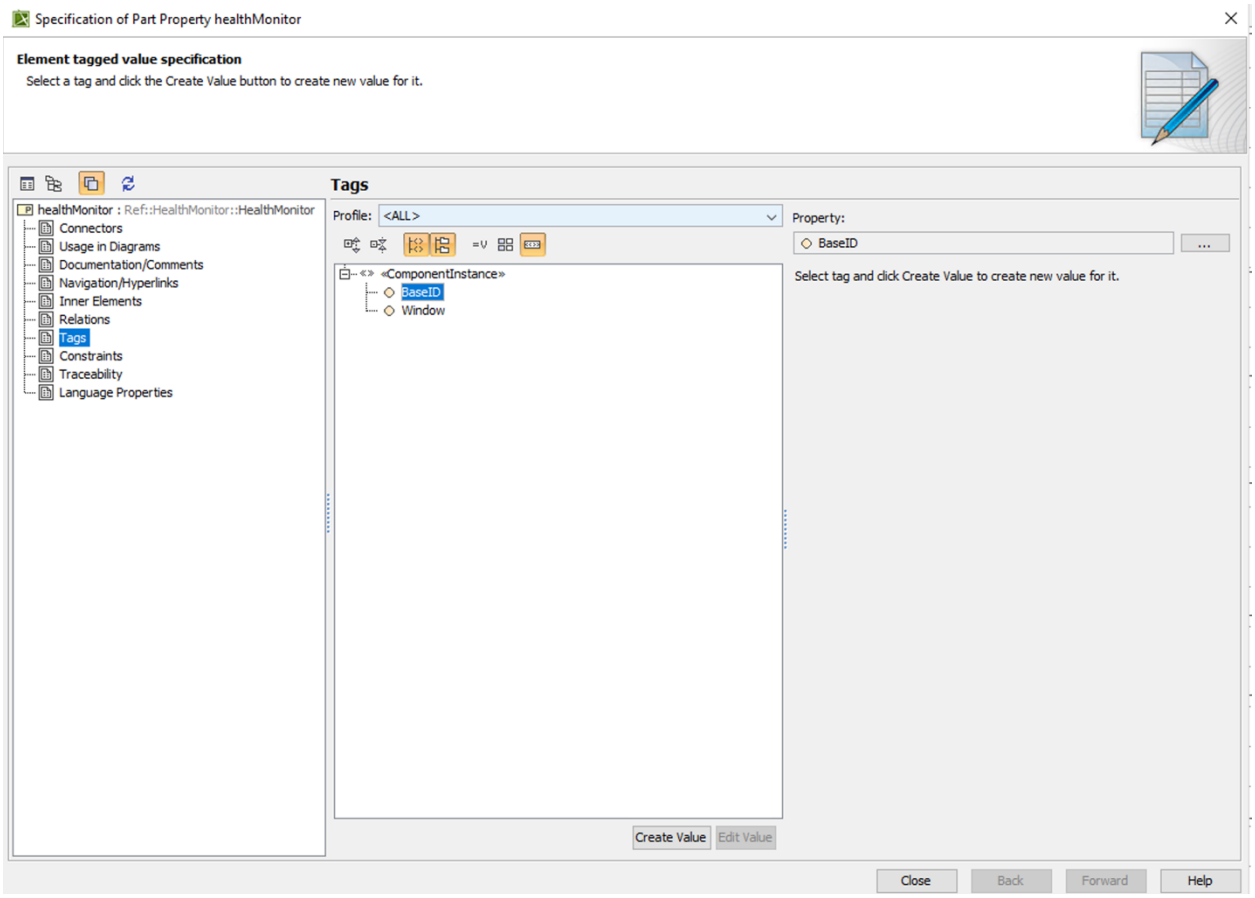
After double clicking a component, this window will pop up, and click browse button (the box with three dots) in the ‘Applied Streotype’ section.

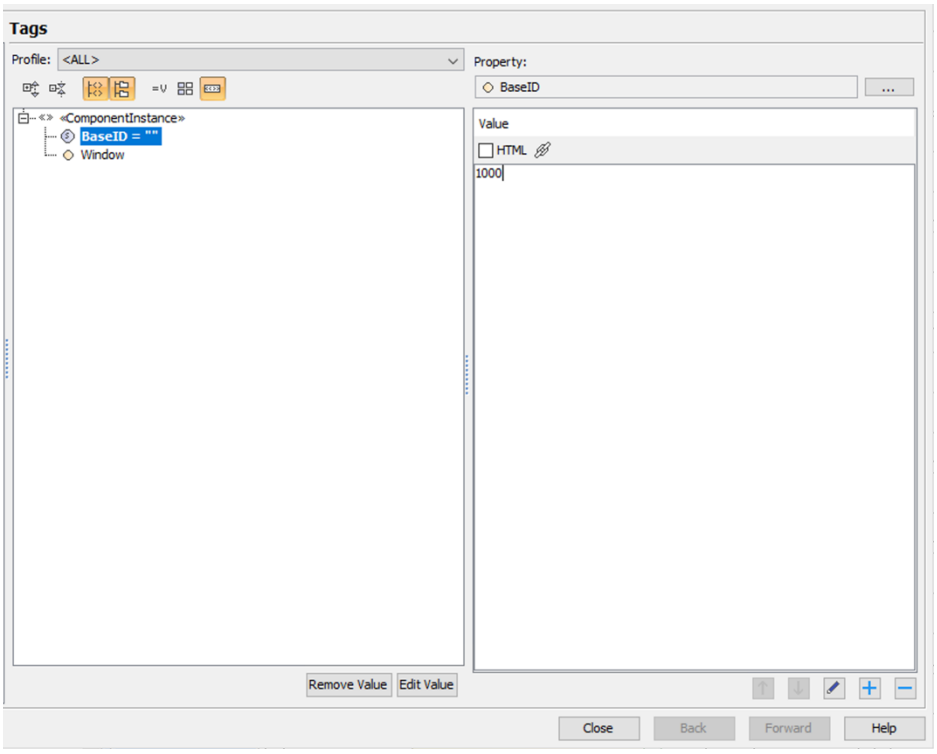


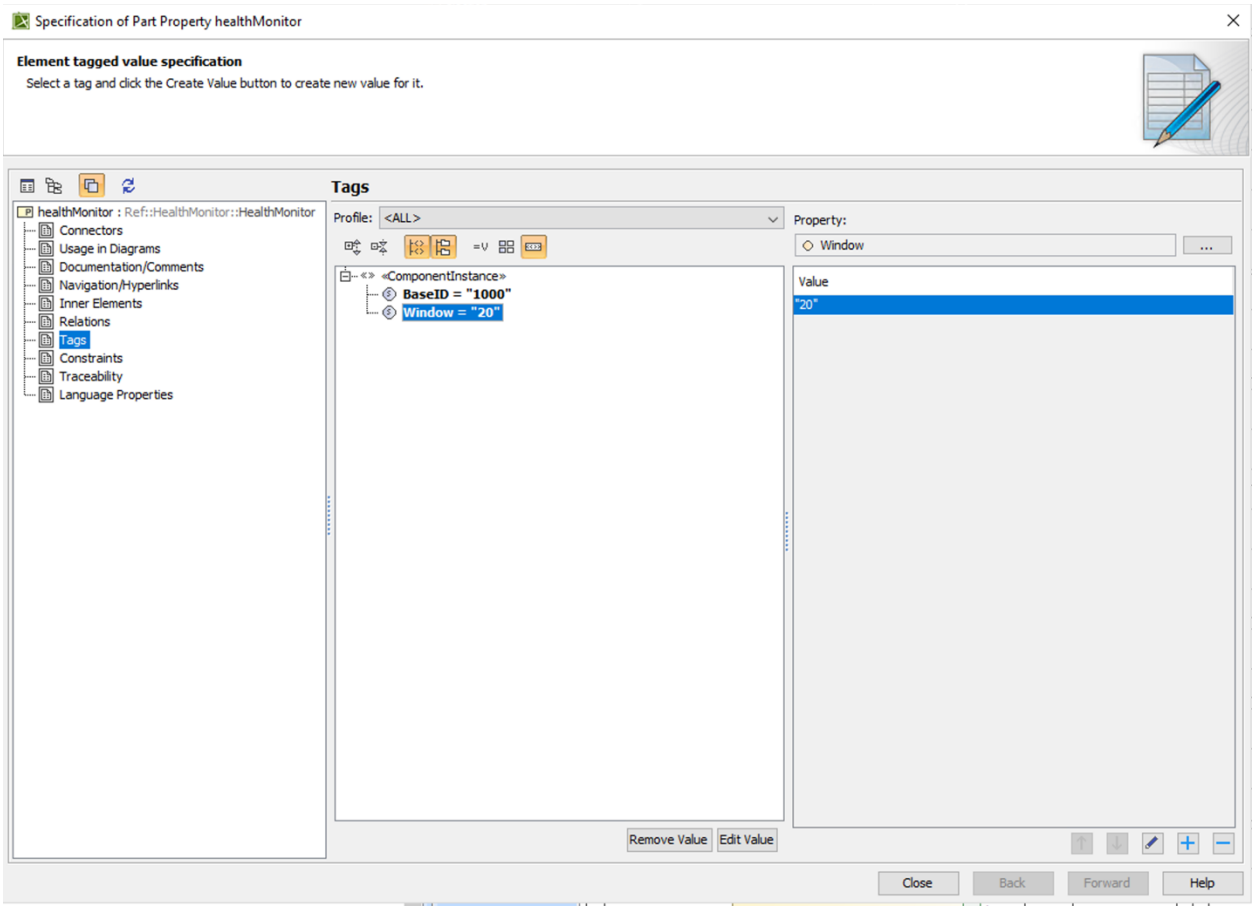
'Component Instance' need to be selected for the component to have base ID and Window.



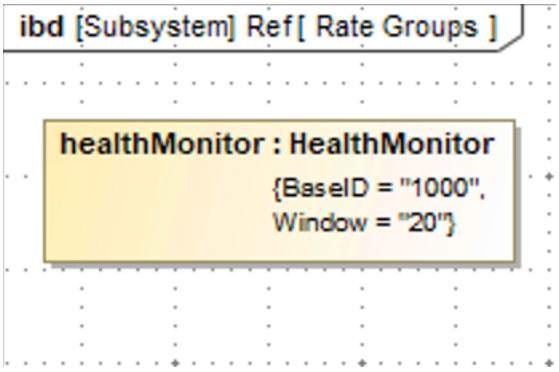
After clicking the 'Apply' button, there should be Component Instance stereotype in the 'Applied Stereotype'.


Specification of Part Property healthMonitor
Element tagged value specification
Select a tag and click the Create Value button to create new value for it.
Tags
healthMonitor : Ref::HealthMonitor::HealthMonitor
Connectors
Usage in Diagrams
Documentation/Comments
Navigation/Hyperlinks
Inner Elements
Relations
Tags
Constraints
Traceability
Language Properties
Profile: <ALL>
«ComponentInstance»
BaseID
Window
Property:
BaseID
Select tag and click Create Value to create new value for it.
Create Value
Edit Value
Close
Back
Forward
Help


Tags
Profile: <ALL>
«ComponentInstance»
BaseID = ""
Window
Property:
BaseID
Value
HTML
1000
Remove Value
Edit Value
Close
Back
Forward
Help

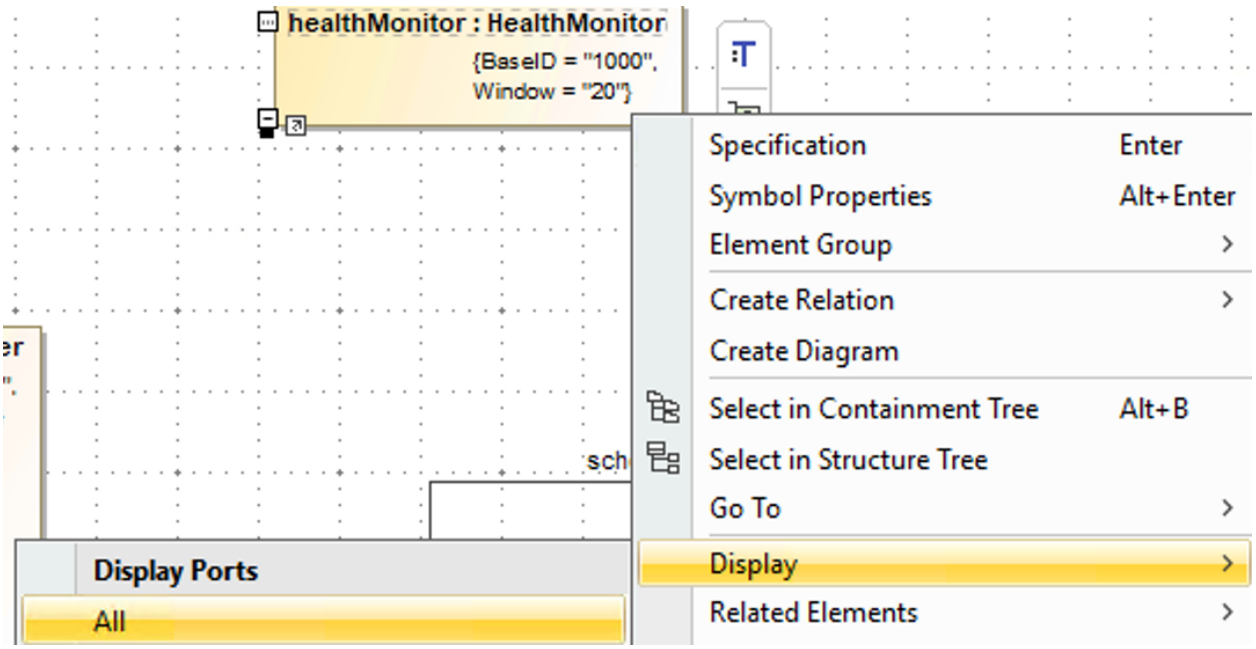


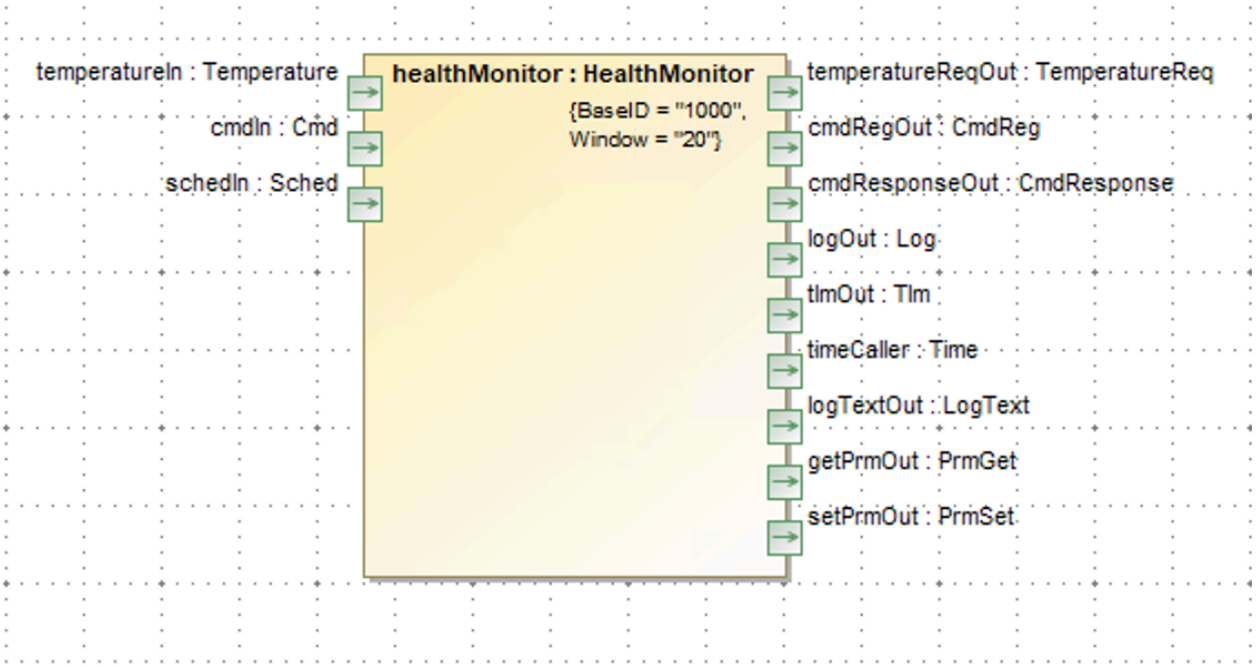
Then, go to ‘Tags’ and click the ‘Create Value’ button and assign a value for both BaseID and Window.



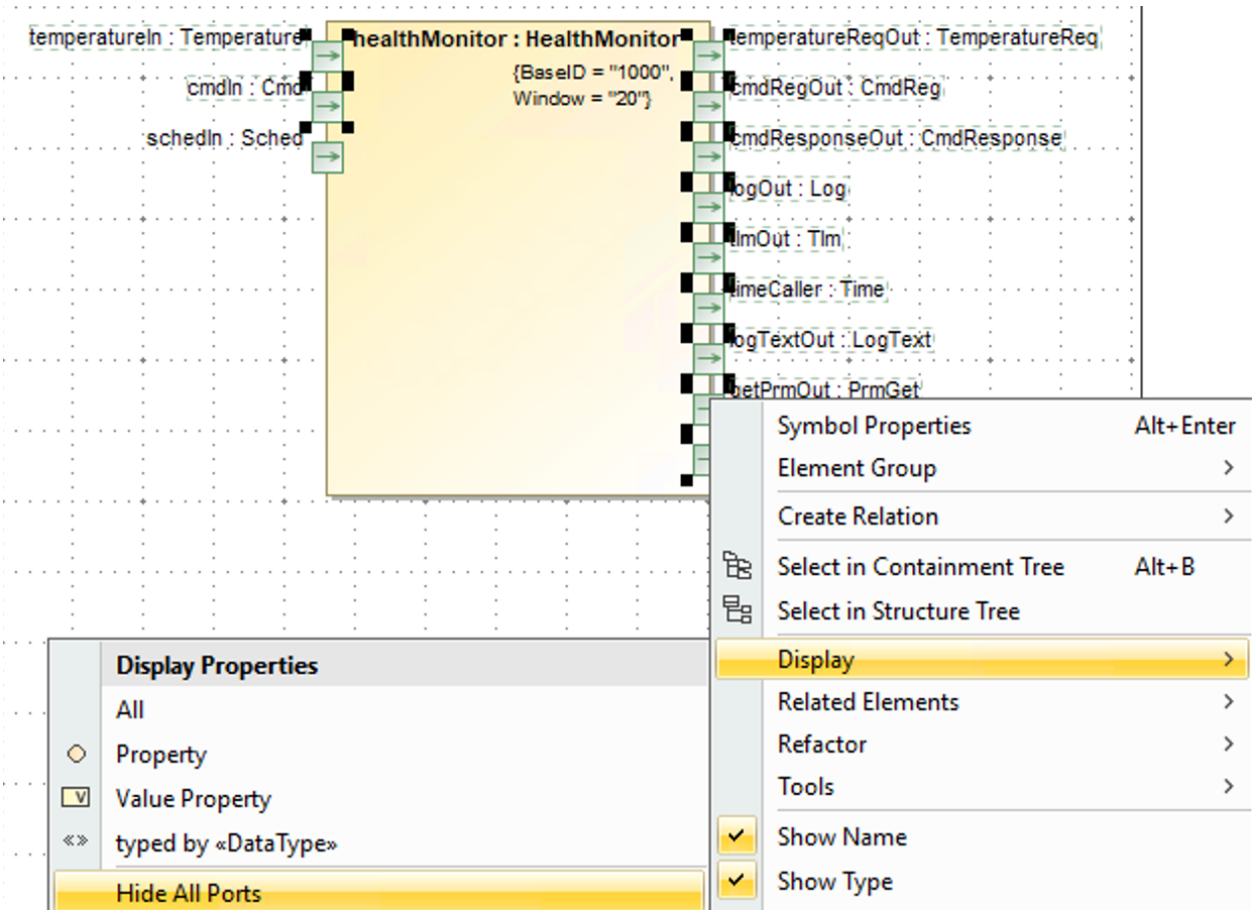
The base id and window should be appeared in the health monitor component.

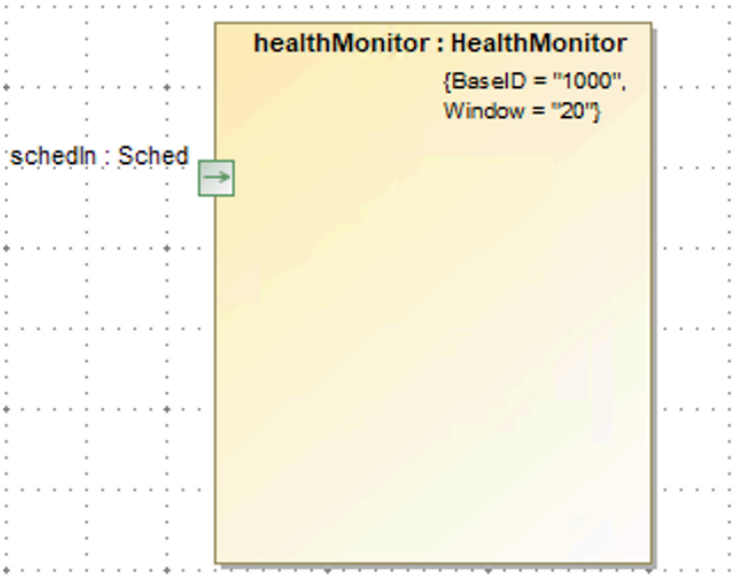
4.7.3 Displaying and Hiding ports





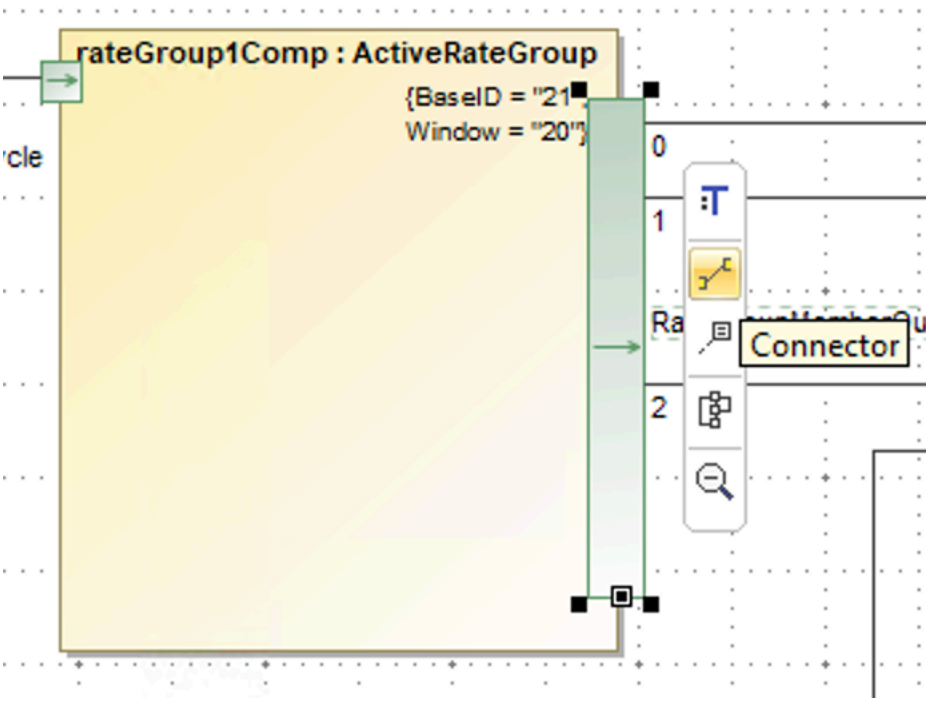
All ports in the health monitor component can be displayed when clicking the Display All.



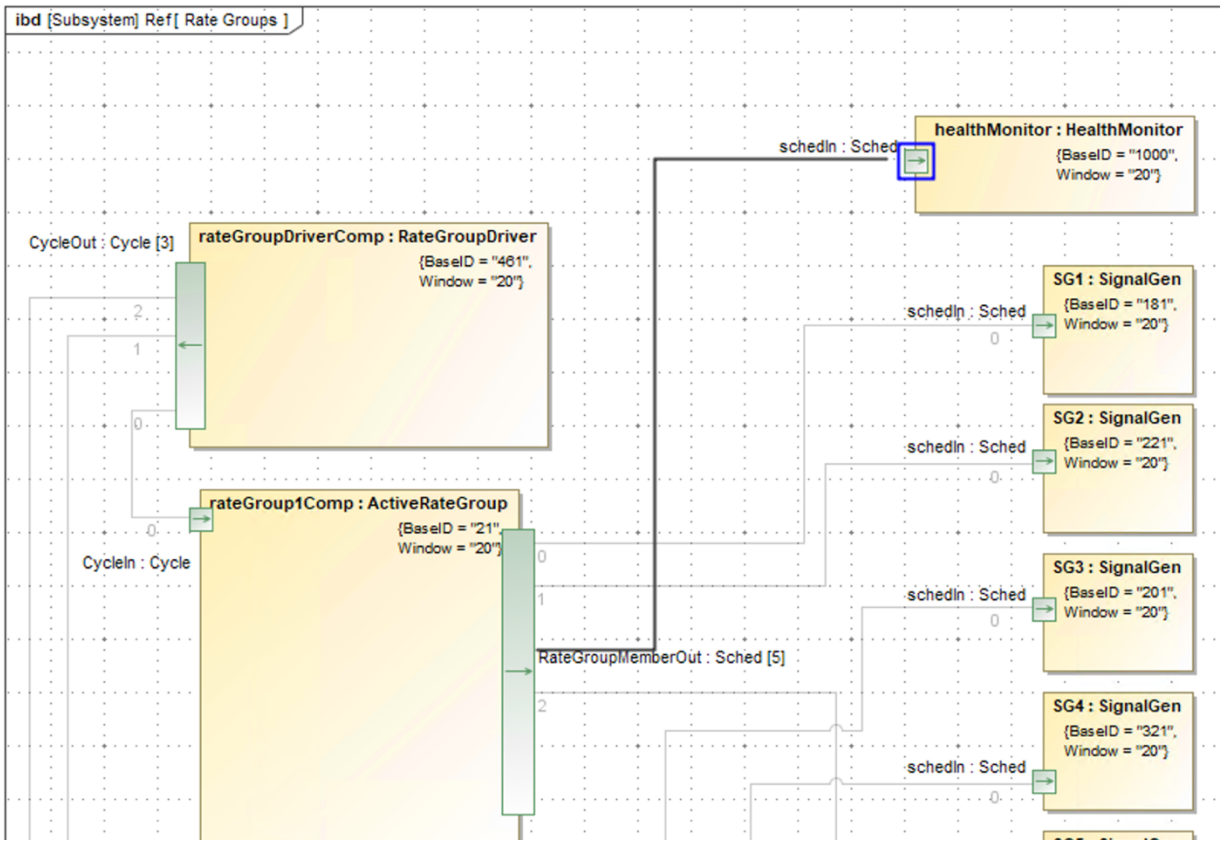


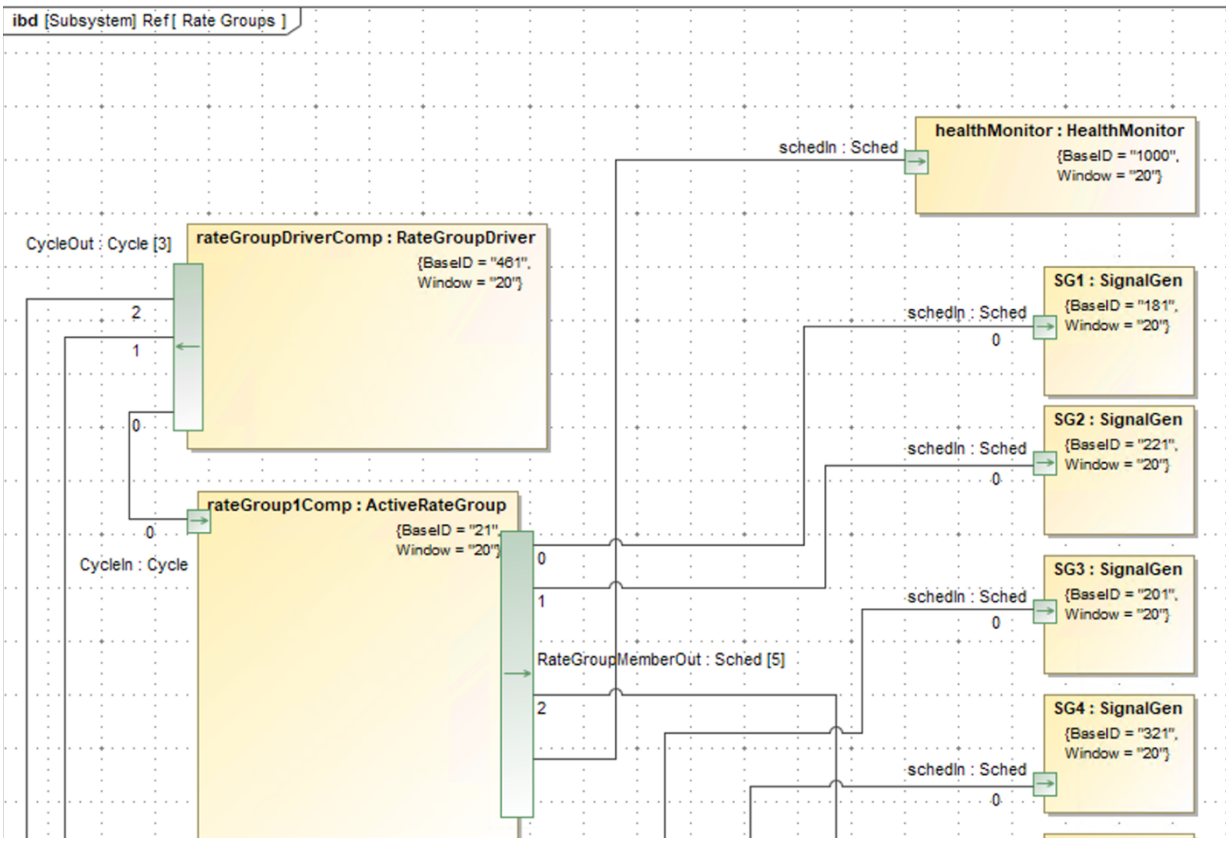
Unnecessary ports in an IBD can be hidden by clicking the Hide All Ports.

4.7.4 Connecting Ports

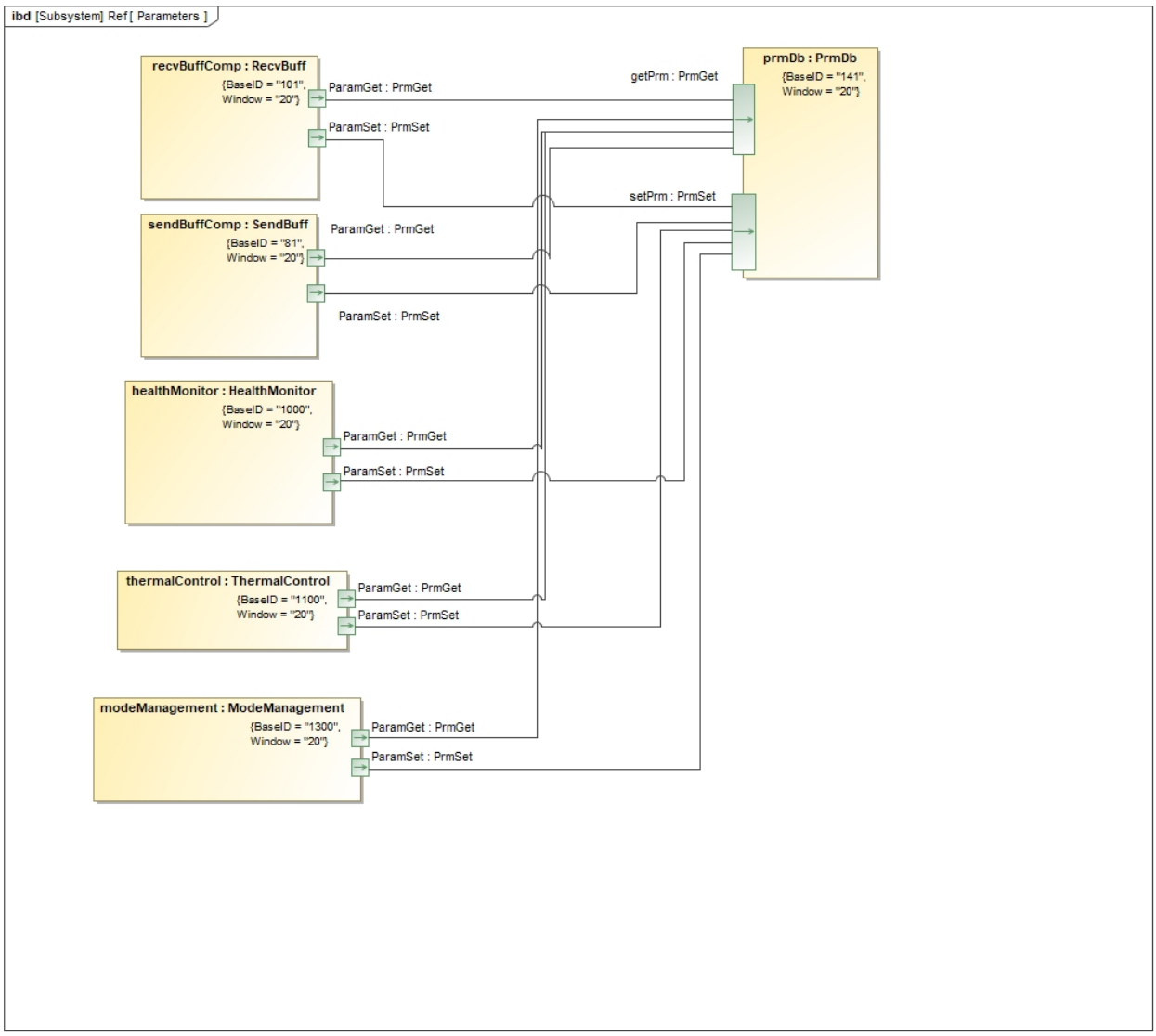


Then, click the connector button to connect a port to another port with the same port type.


ibd [Subsystem] Ref [Rate Groups]
schedIn : Sched
healthMonitor : HealthMonitor
{BaseID = "1000",
Window = "20"}
CycleOut : Cycle [3]
rateGroupDriverComp : RateGroupDriver
{BaseID = "461",
Window = "20"}
2
1
0
SG1 : SignalGen
{BaseID = "181",
Window = "20"}
schedIn : Sched
0
SG2 : SignalGen
{BaseID = "221",
Window = "20"}
schedIn : Sched
0
rateGroup1Comp : ActiveRateGroup
{BaseID = "21",
Window = "20"}
0
CycleIn : Cycle
0
1
SG3 : SignalGen
{BaseID = "201",
Window = "20"}
schedIn : Sched
0
RateGroupMemberOut : Sched [5]
2
SG4 : SignalGen
{BaseID = "321",
Window = "20"}
schedIn : Sched
0


ibd [Subsystem] Ref [Rate Groups]
schedIn : Sched
healthMonitor : HealthMonitor
{BaseID = "1000",
Window = "20"}
CycleOut : Cycle [3]
rateGroupDriverComp : RateGroupDriver
{BaseID = "461",
Window = "20"}
2
1
0
SG1 : SignalGen
{BaseID = "181",
Window = "20"}
schedIn : Sched
0
SG2 : SignalGen
{BaseID = "221",
Window = "20"}
schedIn : Sched
0
rateGroup1Comp : ActiveRateGroup
{BaseID = "21",
Window = "20"}
0
CycleIn : Cycle
0
1
SG3 : SignalGen
{BaseID = "201",
Window = "20"}
schedIn : Sched
0
RateGroupMemberOut : Sched [5]
2
SG4 : SignalGen
{BaseID = "321",
Window = "20"}
schedIn : Sched
0

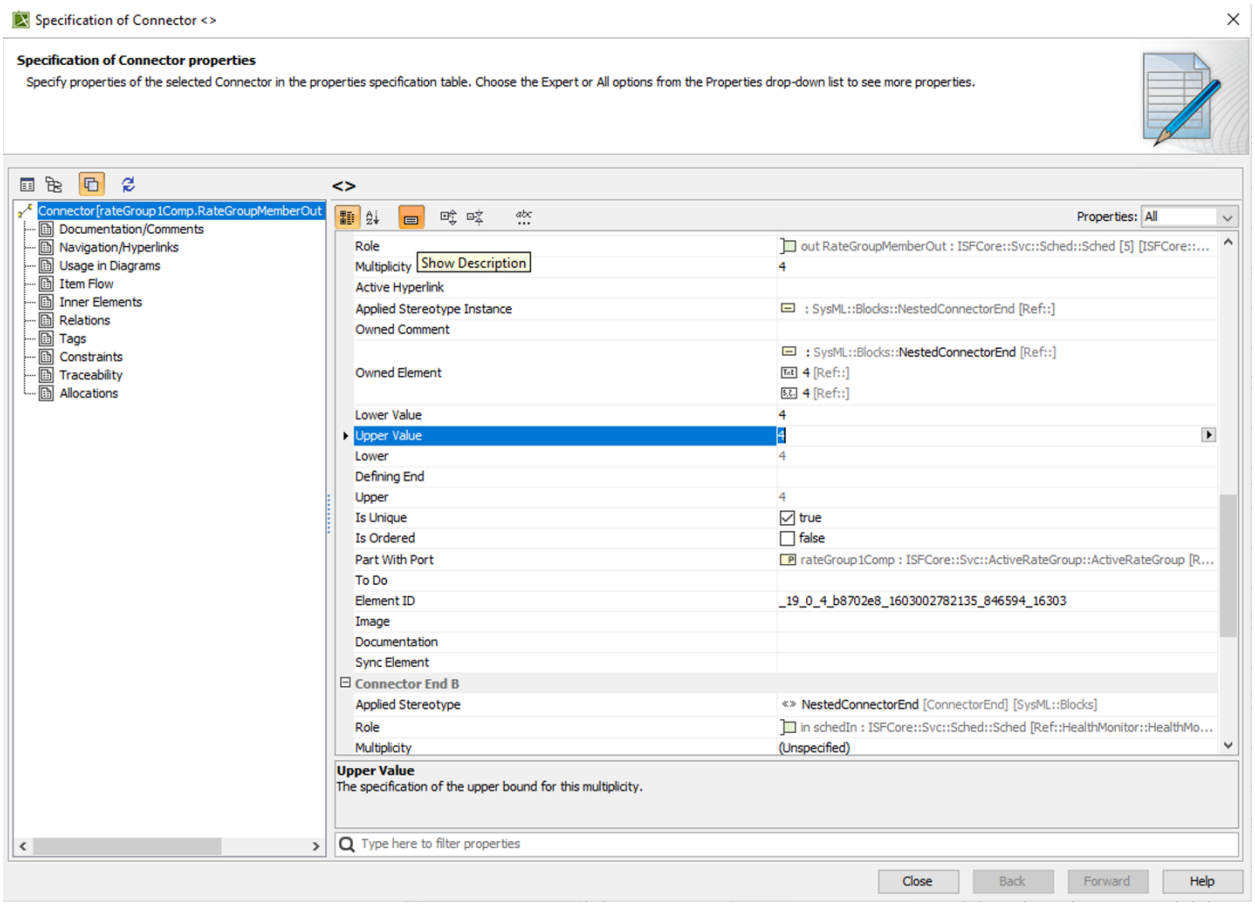
4.7.5 Parameters IBD

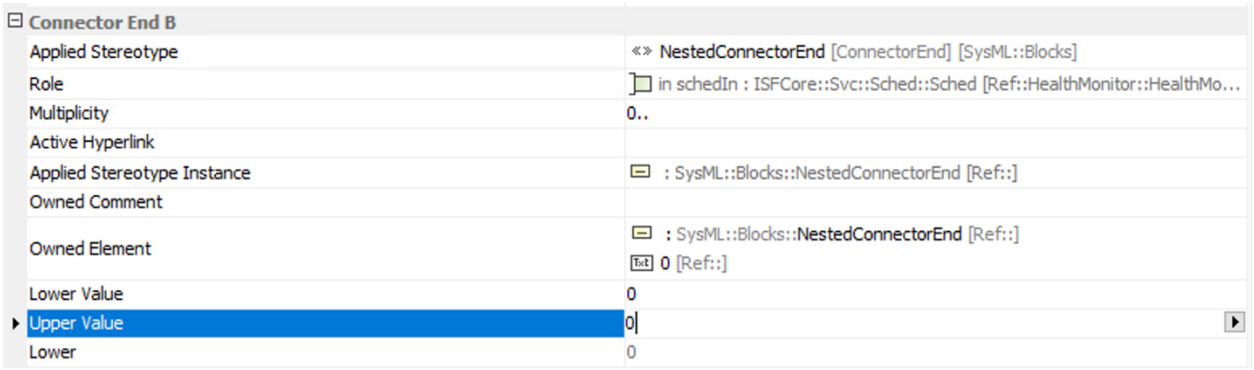


The mode management, health monitor, thermal control components are connected to the pre-defined PrmDB component through ports with the PrmGet and PrmSet port types.

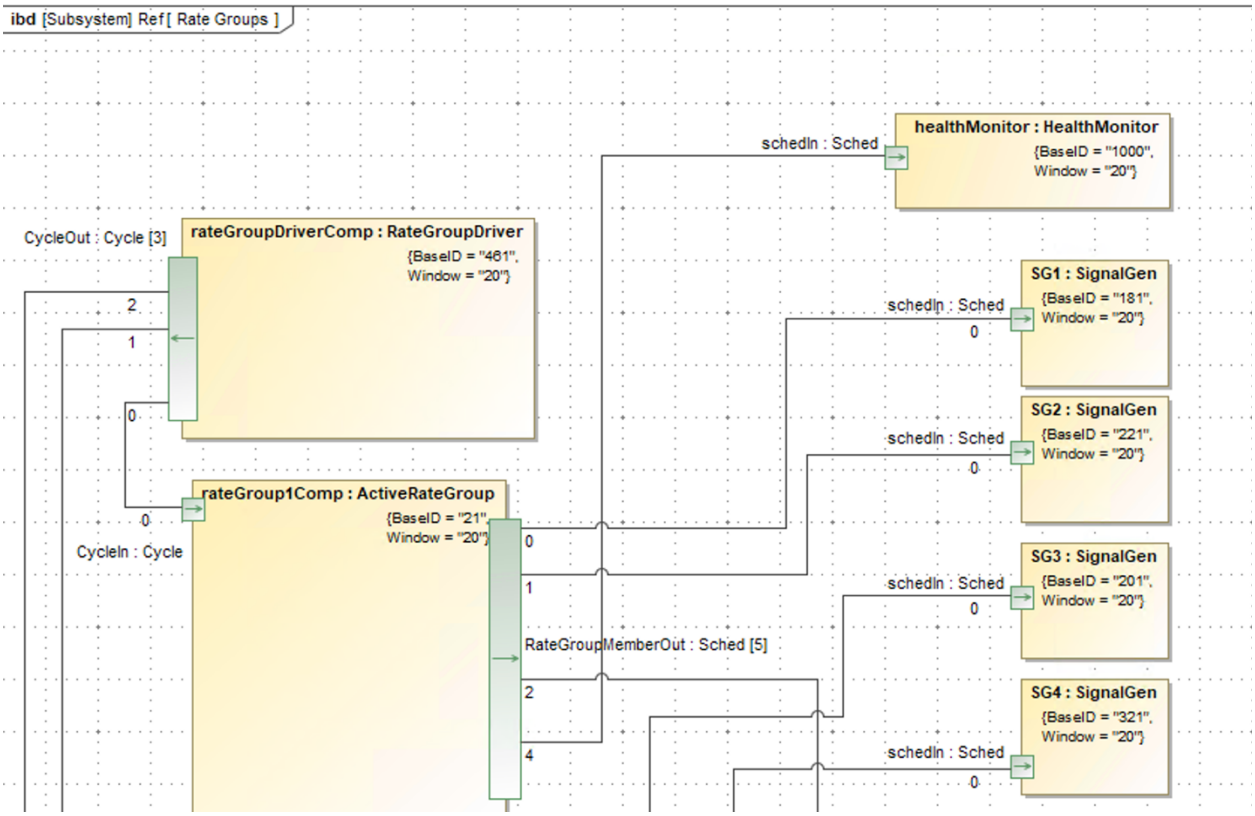
4.7.6 Rate Group IBD

The health monitor component is connected to the rate group1 component through a port with the Sched port type.

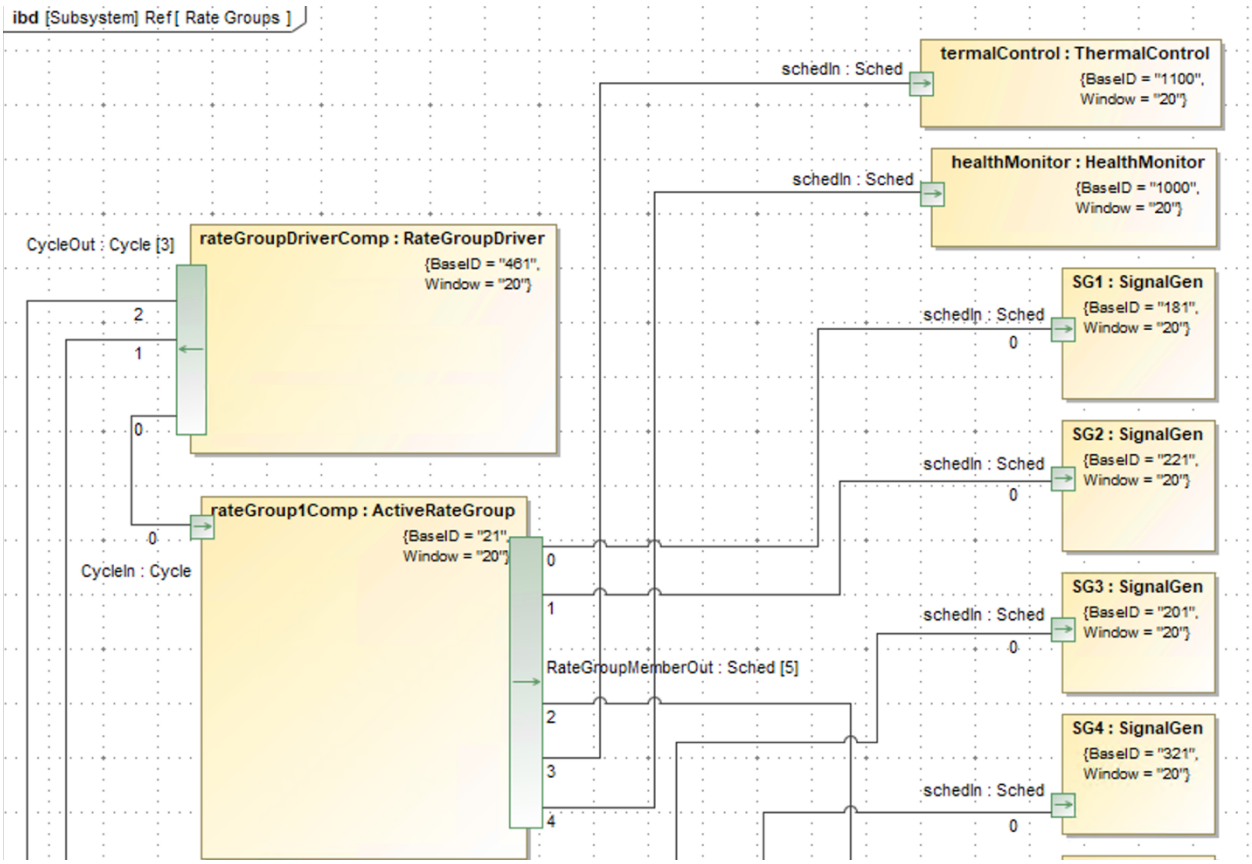




Since the rate group member out port can be connected up to 5 different components, each port needs to be assigned Upper and Lower value.

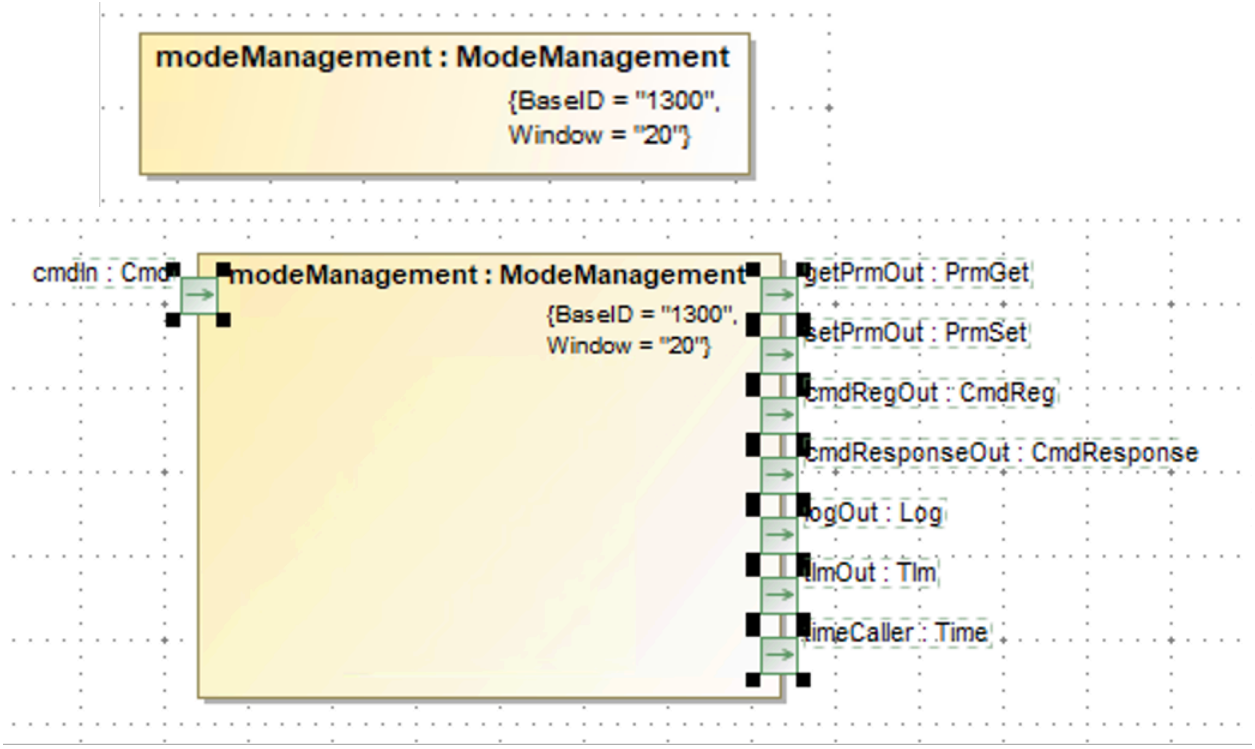


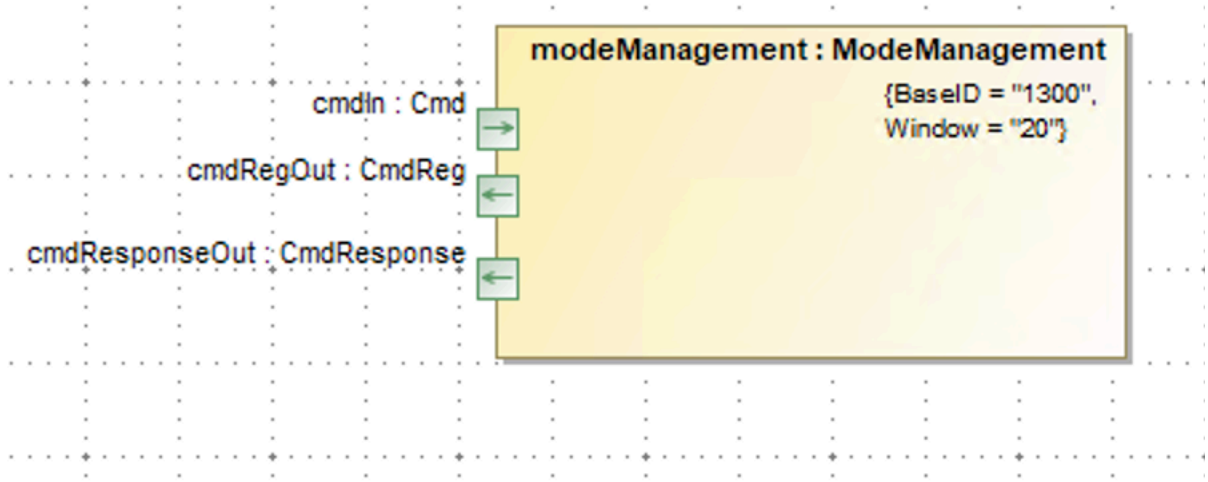
The value will be appeared after assigning the Lower and Upper Value.



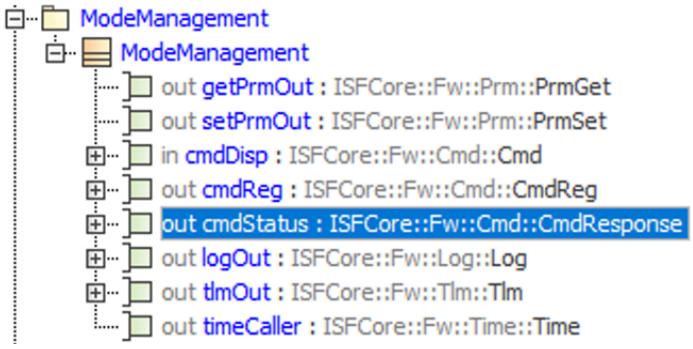
In the same way, in the rate group IBD, the thermal control component is also connected to the rate group 1 component.

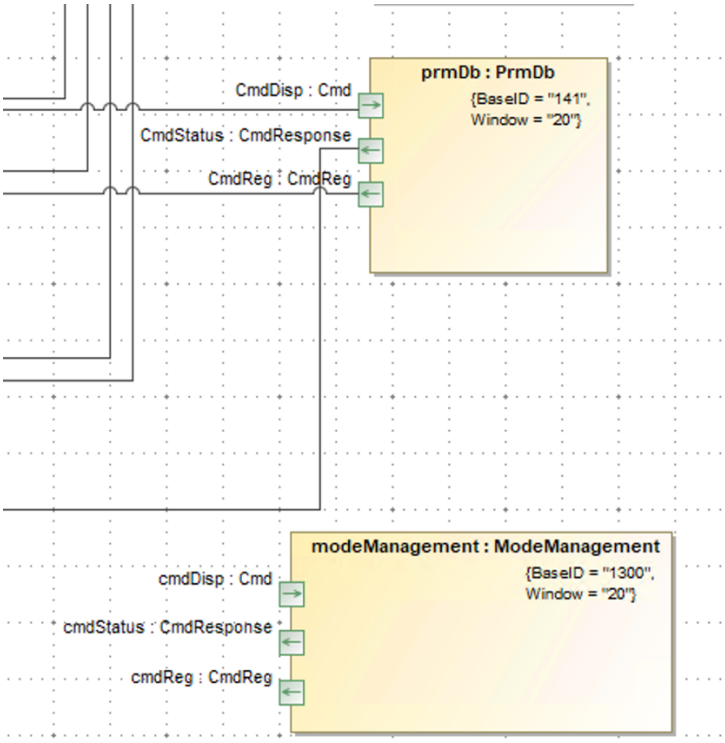
4.7.7 REF Commanding IBD



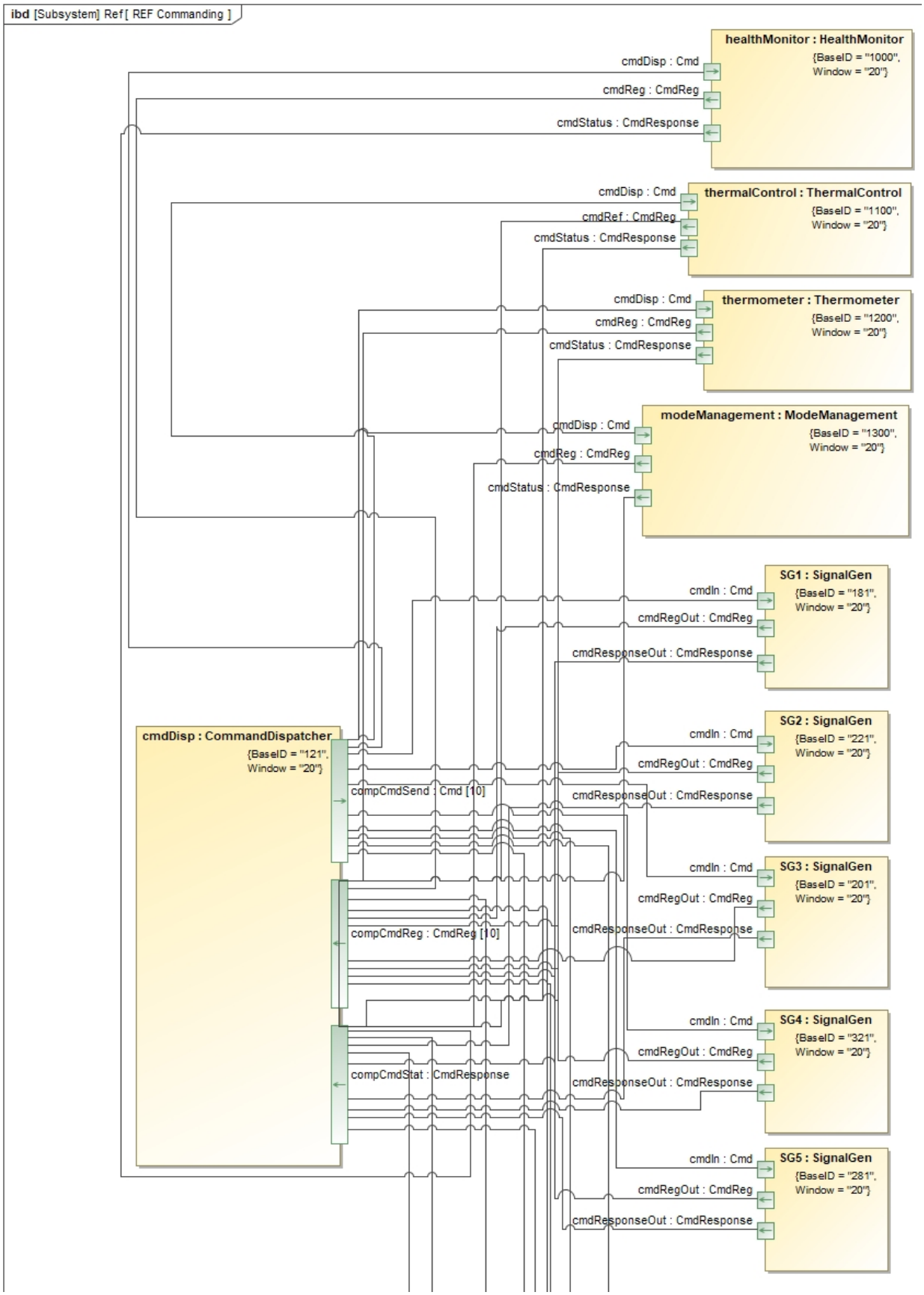


In the REF Commanding IBD, the mode management component is dragged and displayed necessary ports.



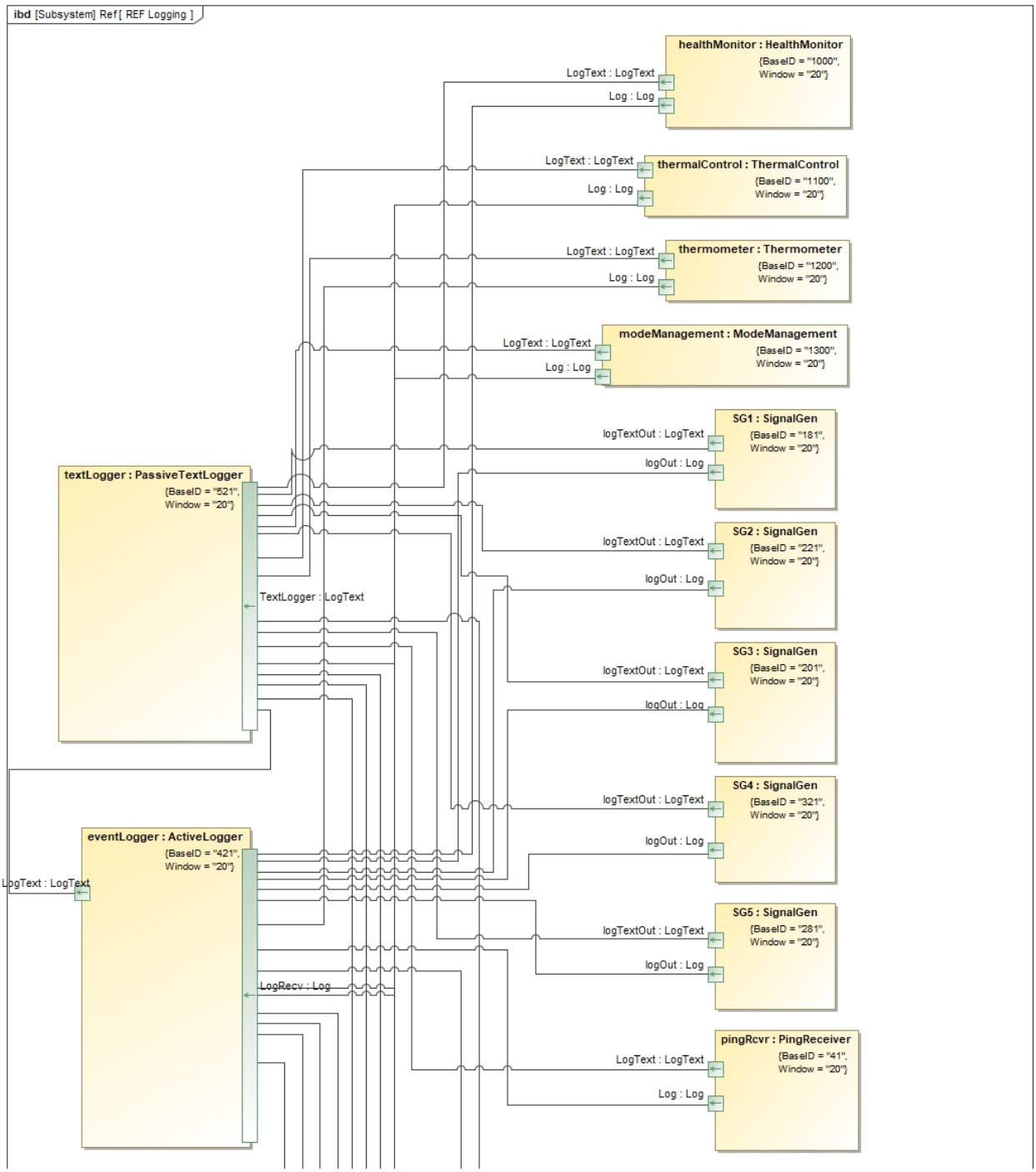


The port name can also be changed in the mode management component.


ibd [Subsystem] Ref [REF Commanding]
healthMonitor : HealthMonitor
{BaseID = "1000", Window = "20"}
cmdDisp : Cmd
cmdReg : CmdReg
cmdStatus : CmdResponse
thermalControl : ThermalControl
{BaseID = "1100", Window = "20"}
cmdDisp : Cmd
cmdRef : CmdReg
cmdStatus : CmdResponse
thermometer : Thermometer
{BaseID = "1200", Window = "20"}
cmdDisp : Cmd
cmdReg : CmdReg
cmdStatus : CmdResponse
modeManagement : ModeManagement
{BaseID = "1300", Window = "20"}
cmdDisp : Cmd
cmdReg : CmdReg
cmdStatus : CmdResponse
SG1 : SignalGen
{BaseID = "181", Window = "20"}
cmdIn : Cmd
cmdRegOut : CmdReg
cmdResponseOut : CmdResponse
SG2 : SignalGen
{BaseID = "221", Window = "20"}
cmdIn : Cmd
cmdRegOut : CmdReg
cmdResponseOut : CmdResponse
SG3 : SignalGen
{BaseID = "201", Window = "20"}
cmdIn : Cmd
cmdRegOut : CmdReg
cmdResponseOut : CmdResponse
SG4 : SignalGen
{BaseID = "321", Window = "20"}
cmdIn : Cmd
cmdRegOut : CmdReg
cmdResponseOut : CmdResponse
SG5 : SignalGen
{BaseID = "281", Window = "20"}
cmdIn : Cmd
cmdRegOut : CmdReg
cmdResponseOut : CmdResponse
cmdDisp : CommandDispatcher
{BaseID = "121", Window = "20"}
compCmdSend : Cmd [10]
compCmdReg : CmdReg [10]
compCmdStat : CmdResponse

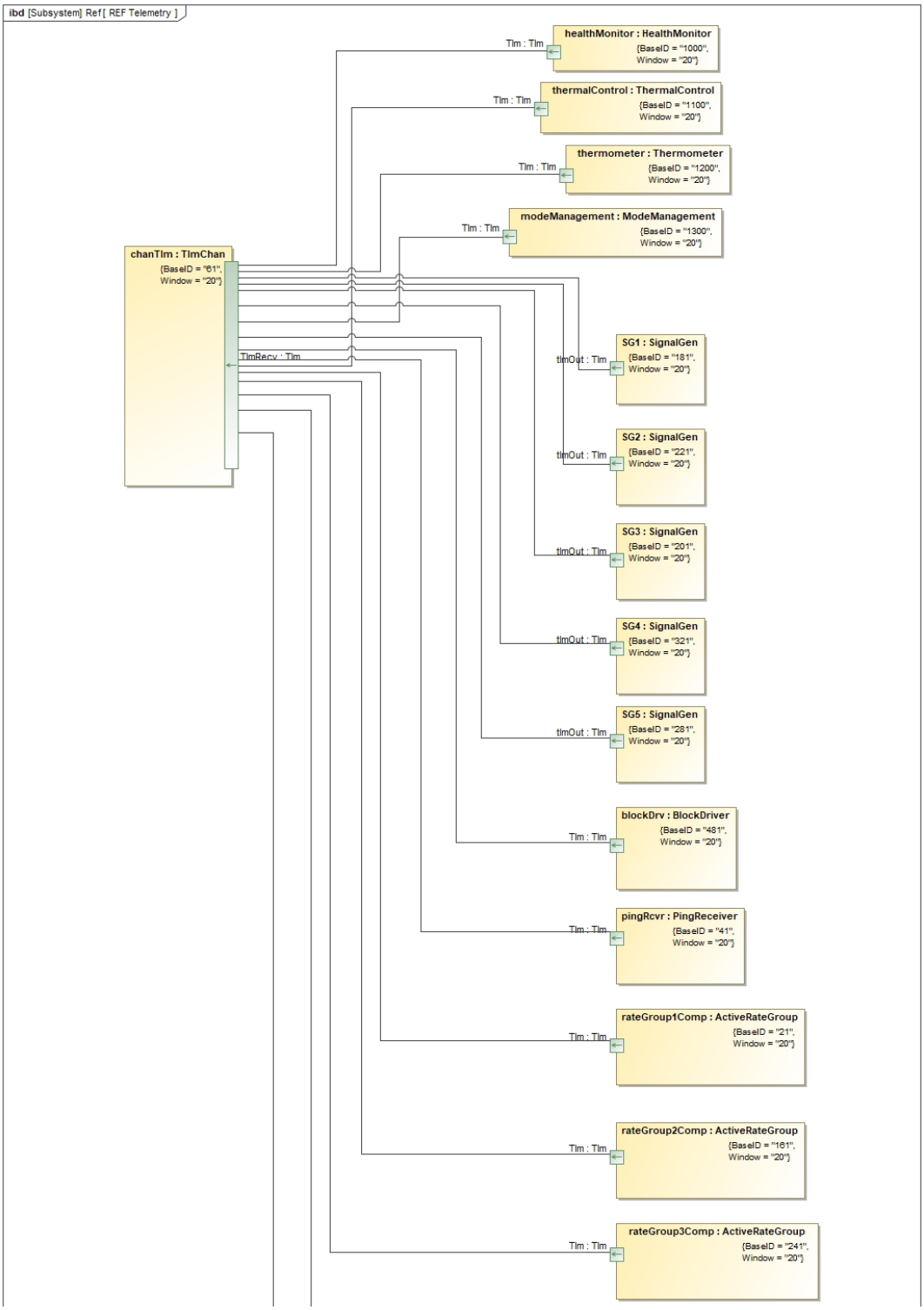
The mode management, health monitor, thermal control, and thermometer components are connected to the pre-defined command dispatcher component through ports with Cmd, CmdResponse, and Cmd Reg port type.

4.7.8 REF Logging IBD

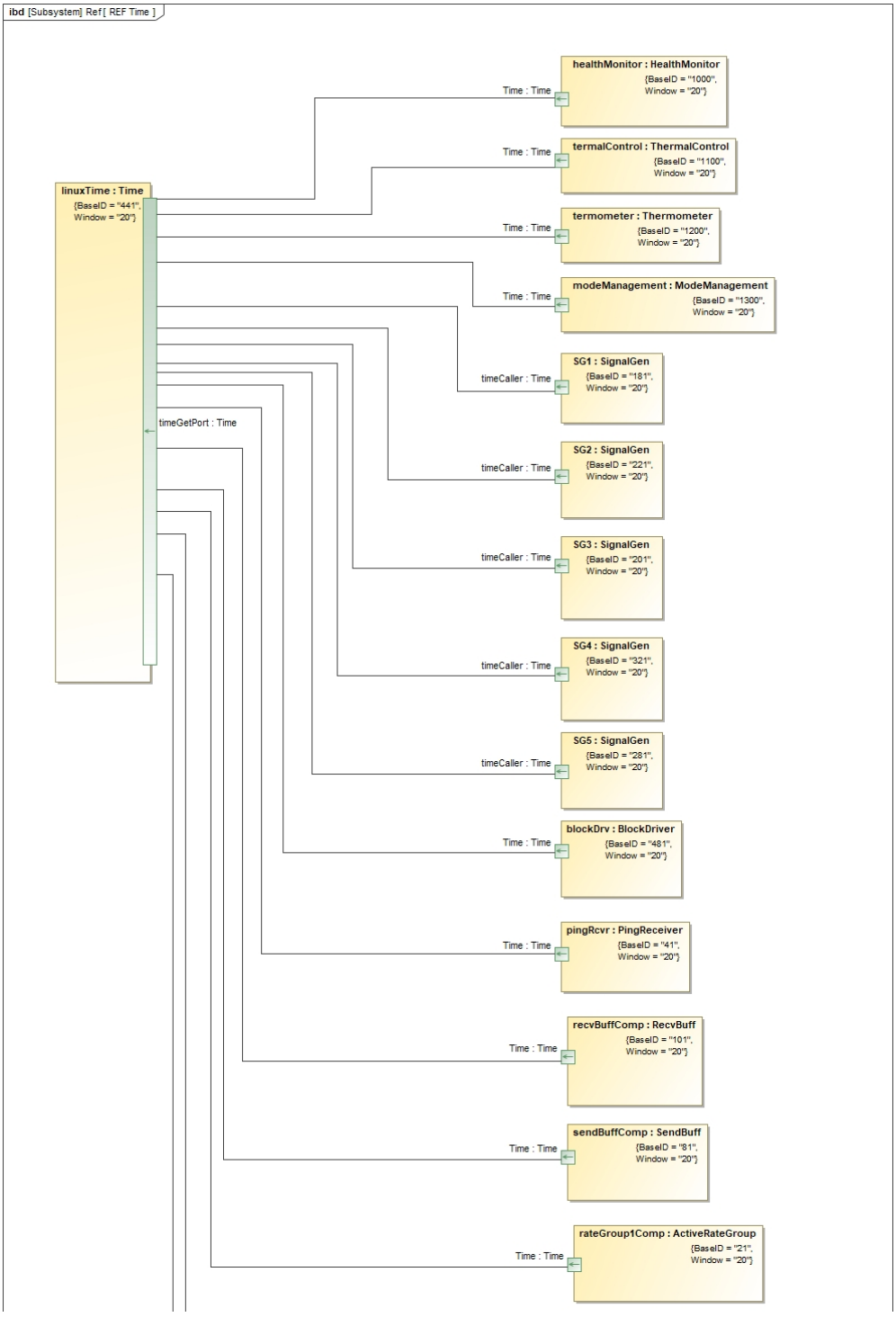


The mode management, health monitor, thermal control, and thermometer components are connected to the pre-defined Active logger and Passive Text Logger components through ports with Log and LogText port types.

4.7.9 REF Telemery IBD

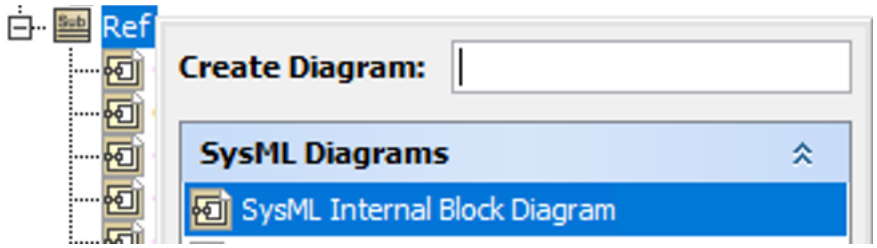


4.7.10 REF Time IBD

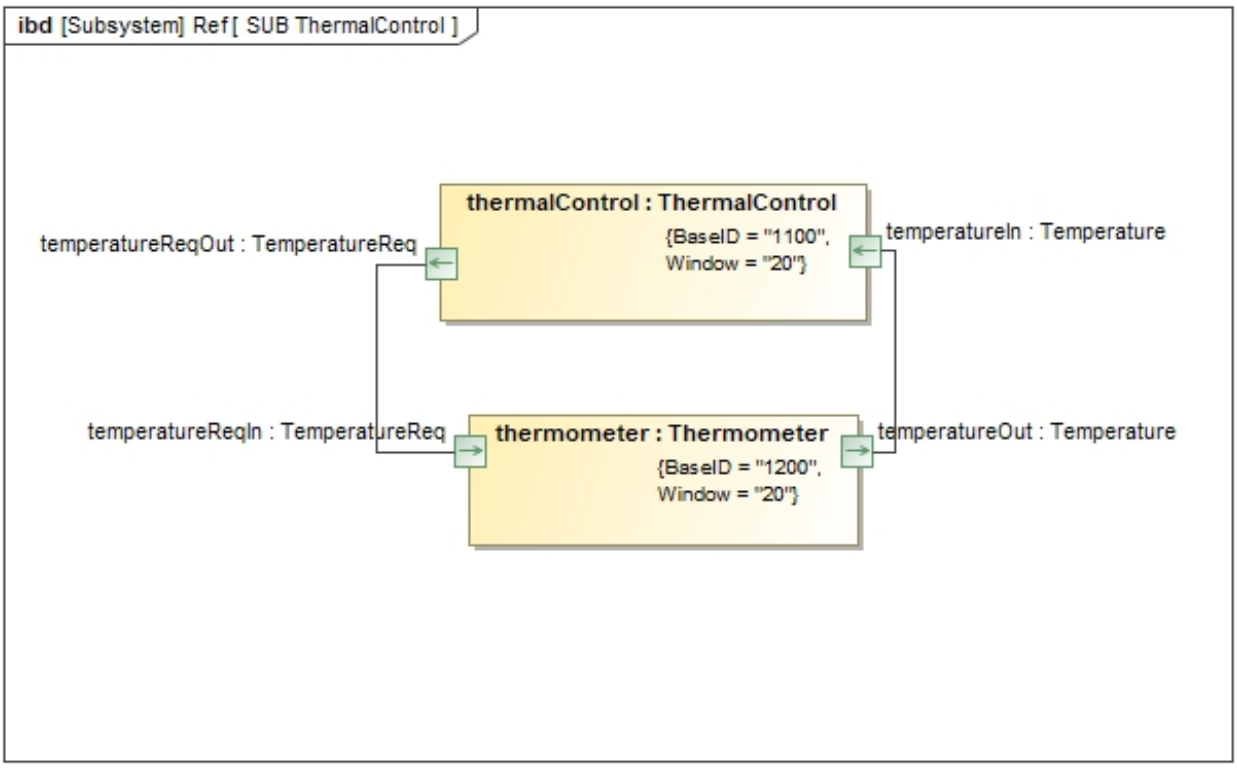


The mode management, health monitor, thermal control, and thermometer components are connected to the pre-defined Time component through a port with the Time port type.

4.7.11 SUB Thermal Control IBD

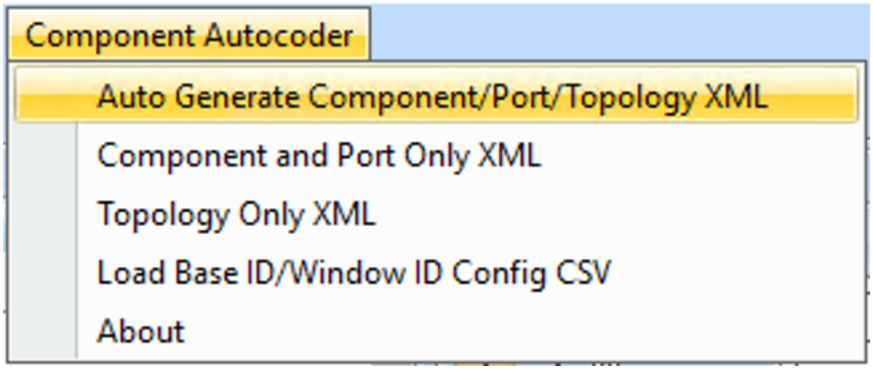


To create own IBD, select the create diagram after right clicking the Ref subsystem. Then, select the 'SysML Internal Block Diagram'.



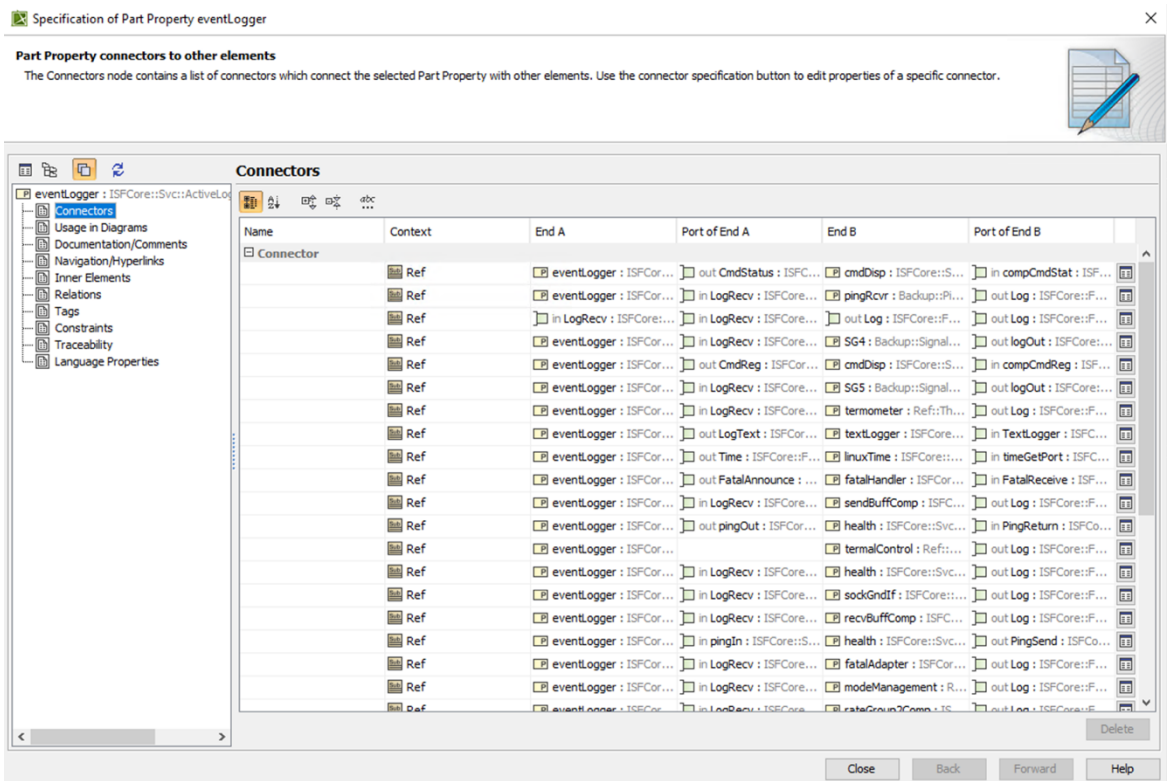
The Thermometer and Thermal Control components are connected through the ports with the Temperature and TemperatureReq port types.

4.8 Null Role Error

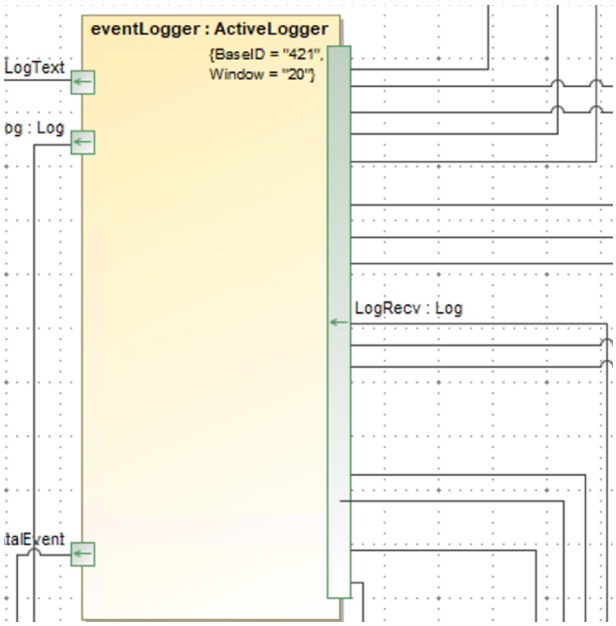



```
[2020.10.24::18:05:27] Writing new file: C:\Users\cubesat\Documents\fprime-aegis\Ref\Top/AutoXML/LogTextPortAi.xml
[2020.10.24::18:05:27] Writing new file: C:\Users\cubesat\Documents\fprime-aegis\Ref\Top/AutoXML/LogPortAi.xml
[2020.10.24::18:05:27] Writing new file: C:\Users\cubesat\Documents\fprime-aegis\Ref\Top/AutoXML/SchedPortAi.xml
[2020.10.24::18:05:27] Writing new file: C:\Users\cubesat\Documents\fprime-aegis\Ref\Top/AutoXML/ComPortAi.xml
[2020.10.24::18:05:27] Writing new file: C:\Users\cubesat\Documents\fprime-aegis\Ref\Top/AutoXML/TImPortAi.xml
[2020.10.24::18:05:27] Project Name = REFApplication-Sub
[2020.10.24::18:05:27] *** Subsystem = Subsystem Ref
[2020.10.24::18:05:27] ===> FATAL Exception: Connector in Subsystem Ref has an end with a null role.
```

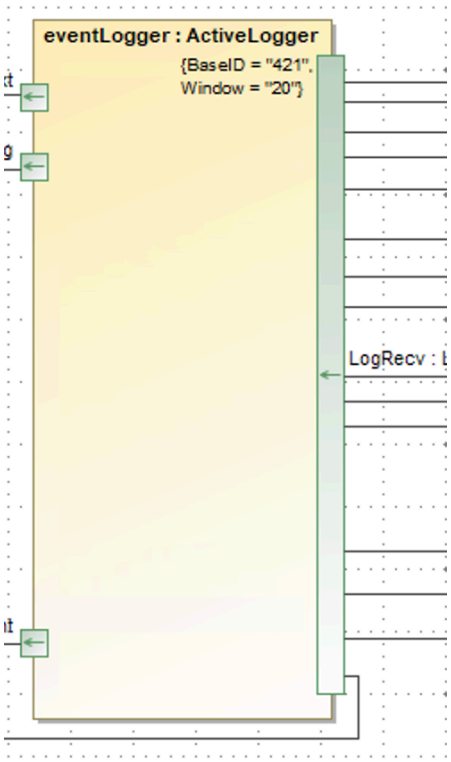
When clicking 'Auto Gnerate Component/Port/Topology XML,' the "===> FATAL Exception: Connector in Subsystem Ref has an end with a null role." error can be thrown like the above.



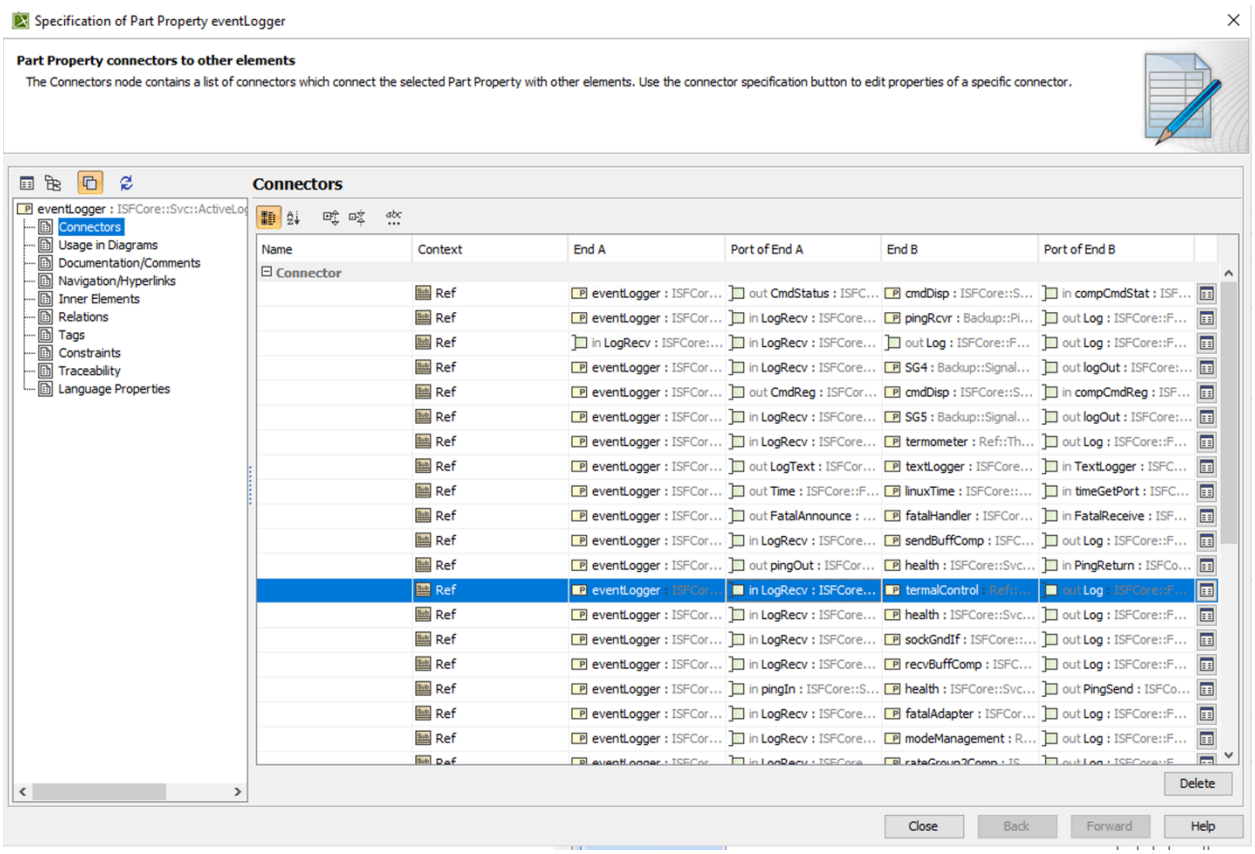
To fix this error, double click components to see the ‘Connectors’ if there is any blank Port of End A or B. There is a blank in Port of End A in the eventLogger component.



There is a line slightly off of the LogRecv port. That means this line is not properly connected.



Make sure this line does not go off from the other ports.

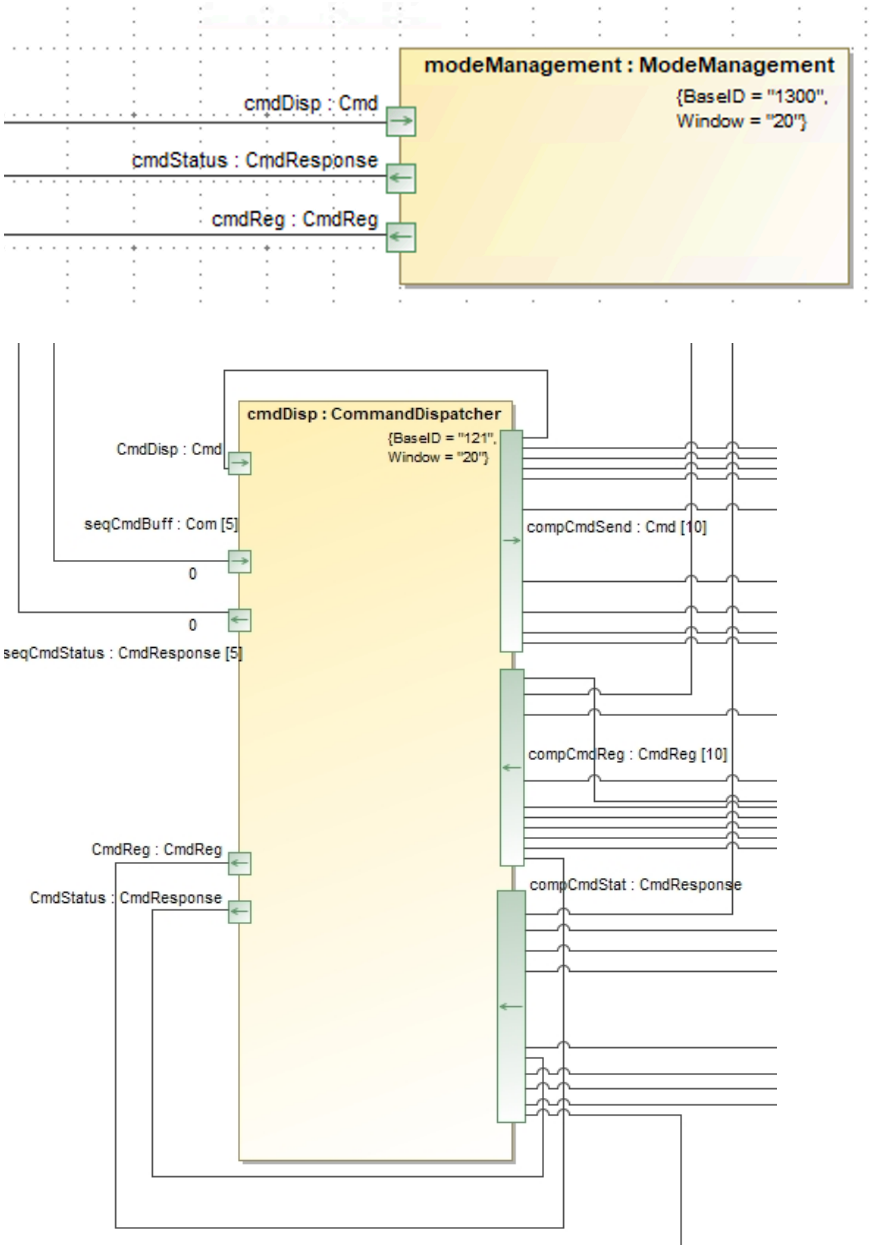


After moving the line and double clicking the eventLogger component again, there is no blank in the Port of End A list.

4.9 Opposite Type Error

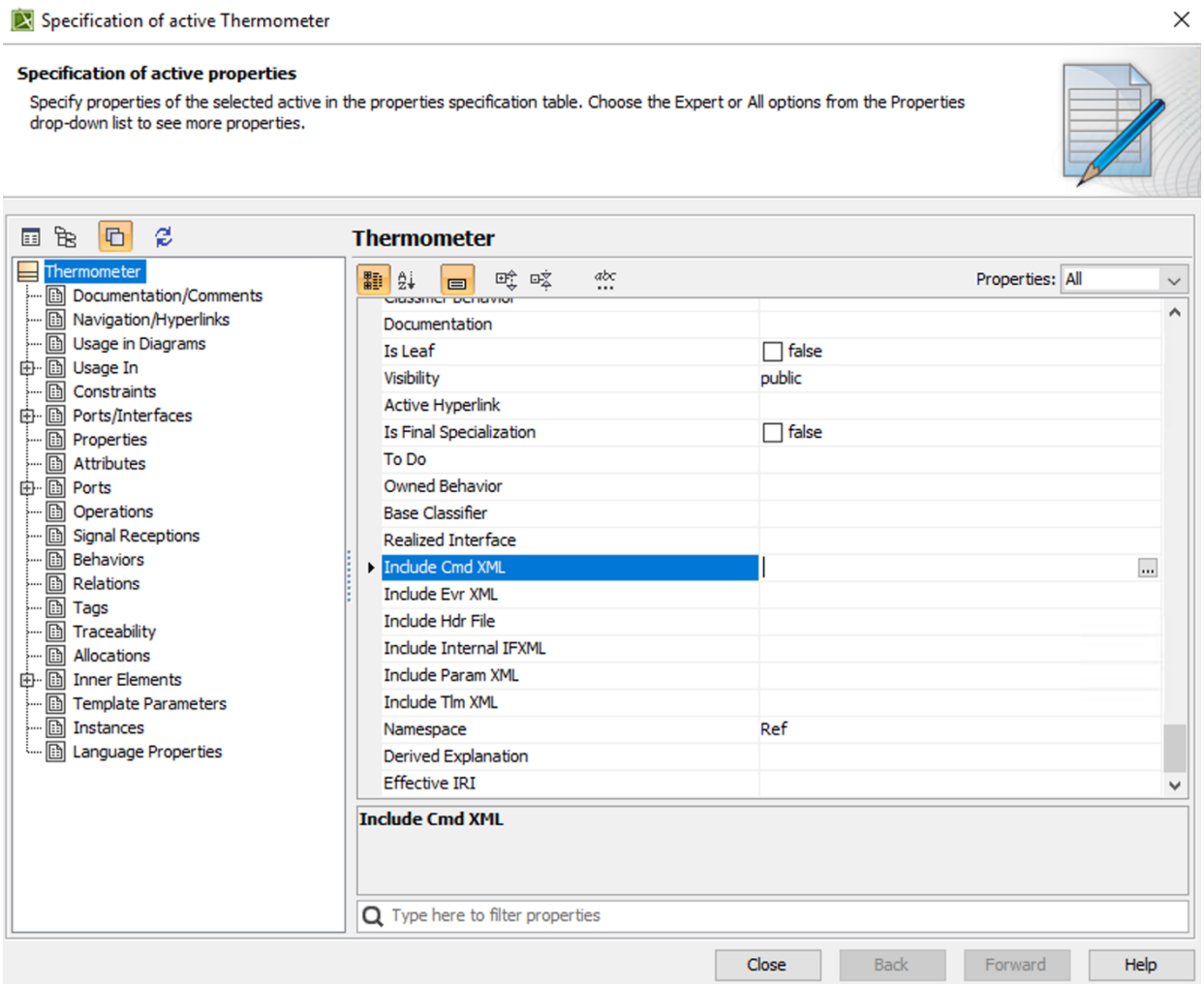
[2020.10.24::18:22:48] ===> FATAL Exception: The connection from cmdDisp in subsystem Subsystem Ref to modeManagement in subsystem Subsystem Ref of type Cmd does not have an connection going from the same objects of the opposite type. Auto-assigning indexes cannot be continued.

If there is "===> FATAL Exception: The connection from cmdDisp in subsystem Subsystem Ref to modeManagement in subsystem Subsystem Ref of type Cmd does not have a connection going from the same objects of the opposite type. Auto-assigning indexes cannot be continued." error, it means that the connected port types between components does not match.



Check the port type goes to the same port type. For example, the Cmd port type in mode management component goes to the same port type in command dispatcher component as well as the CmdReg and CmdResponse port types.

4.10 Optional XML

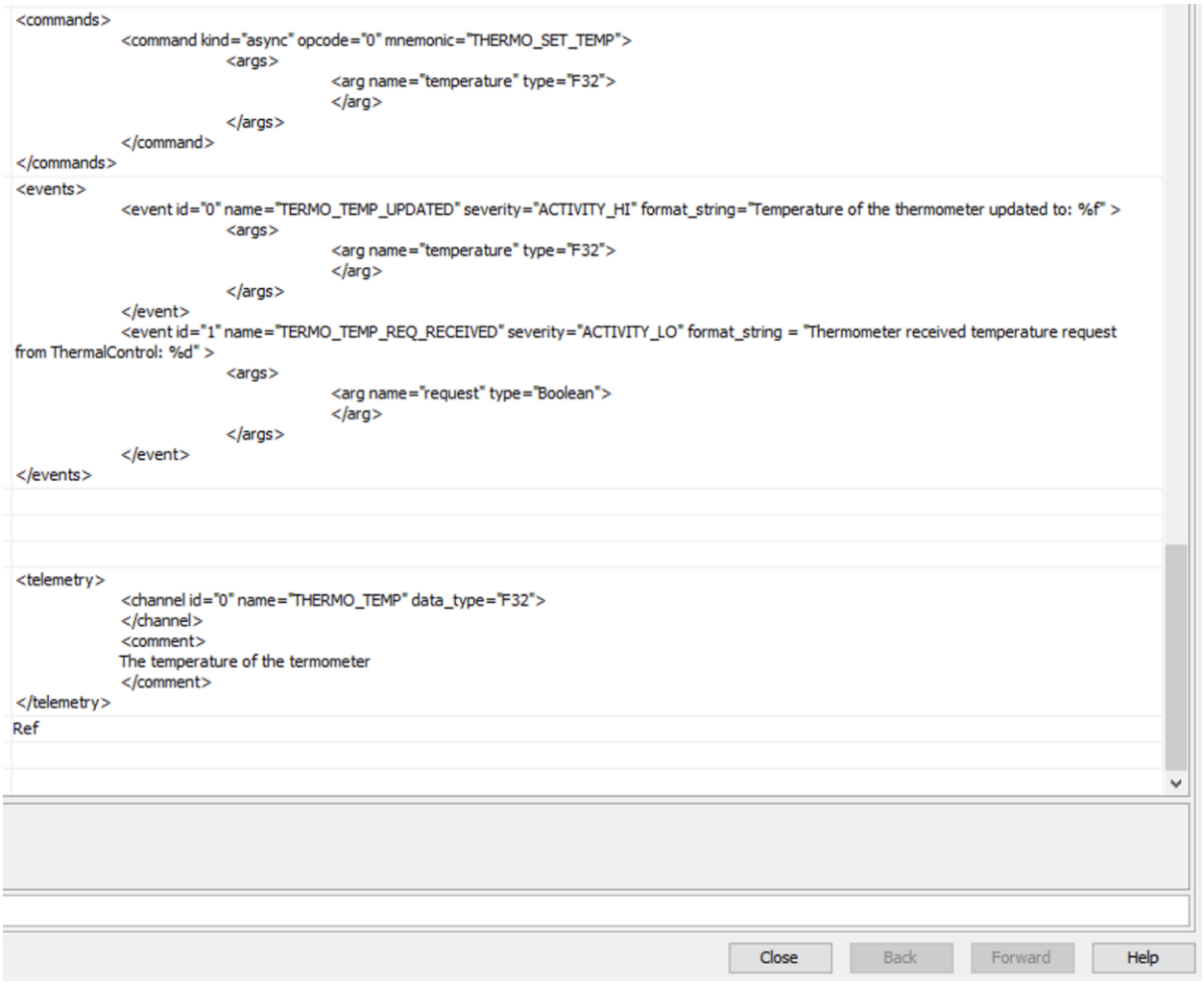


This window will pop up by double-clicking a component. Besides the port and component, there are other optional XML sections that can be added. Cmd is for commands, Evr is for events, Param is for parameters, and Tlm is for channels. Each section can be updated by clicking the browse button (the box with three dots) and writing each xml lines.

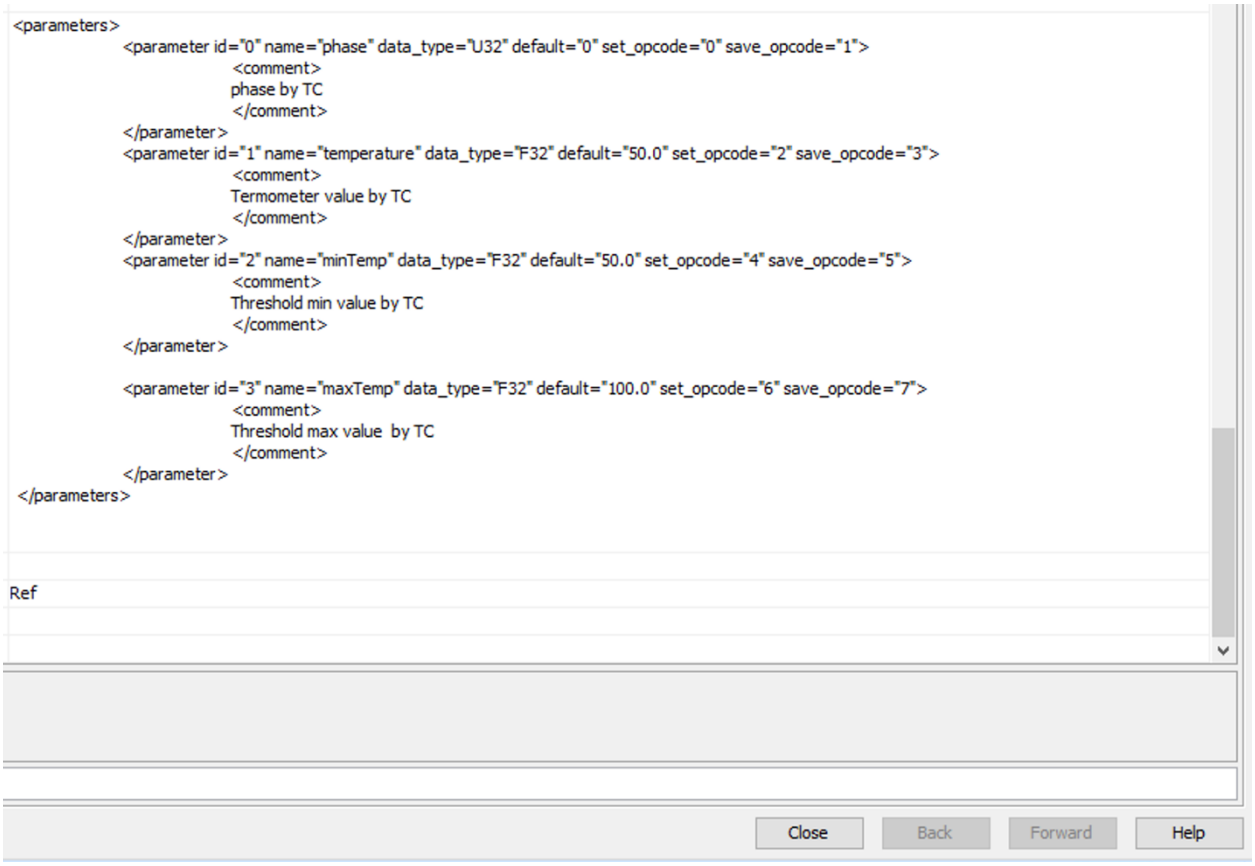
These sections can be modified depending on the design.

<table>
<tr><th>Tag</th><th>Attribute</th><th>Description</th></tr>
<tr><td>parameters</td><td></td><td>Optional. Specifies the section that defines parameters for the component.</td></tr>
<tr><td>parameters</td><td>parameter_base</td><td>Defines the base value for the parameter IDs. If this is specified, all parameter IDs will be added to this value. If it is missing, parameers IDs will be absolute. This tag can also have a variable of the form “$variable” referring to values in Fw/Cfg/AcContstants.ini.</td></tr>
<tr><td>parameters</td><td>opcode_base</td><td>Defines the base value for the opcodes in the parameter set and save commands. If this is specified, all opcodes will be added to this value. If it is missing, opcodes will be absolute. This tag can also have a variable of the form “$variable” referring to values in Fw/Cfg/AcContstants.ini.</td></tr>
<tr><td>parameter</td><td></td><td>Starts the definition for a parameter</td></tr>
<tr><td>parameter</td><td>id</td><td>Specifies a numeric value that represents the parameter</td></tr>
<tr><td>parameter</td><td>name</td><td>Specifies the name of the parameter</td></tr>
<tr><td>parameter</td><td>data_type</td><td>Specifies the type of the parameter. Should be one of the types defined in Table 1, ENUM, “string”, or an XML specified serializable. A “string” type should be used if a text string is the argument.</td></tr>
<tr><td>parameter</td><td>size</td><td>Specifies the size of the parameter if it is of type “string”</td></tr>
<tr><td>parameter</td><td>default</td><td>Specifies a default value for the parameter if the parameter is unable to be retrieved from non-volatile storage. Only for built-in types.</td></tr>
<tr><td>parameter</td><td>comment</td><td>A comment describing the parameter.</td></tr>
<tr><td>parameter</td><td>set_opcode</td><td>Command opcode used to set parameter</td></tr>
<tr><td>parameter</td><td>save_opcode</td><td>Command opcode used to save parameter</td></tr>
<tr><td>event</td><td>severity</td><td>Specifies the severity of the event. The values can be:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>DIAGNOSTIC</td><td>Software debugging information. Meant for development.</td></tr>
<tr><td>ACTVITY_LO</td><td>Low priority events related to software execution.</td></tr>
<tr><td>ACTVITY_HI</td><td>Higher priority events related to software execution.</td></tr>
<tr><td>COMMAND</td><td>Events related to command execution. Should be reserved for command dispatcher and sequencer.</td></tr>
<tr><td>WARNING_LO</td><td>Error conditions that are of low importance.</td></tr>
<tr><td>WARNING_LO</td><td>Error conditions that are of critical importance.</td></tr>
<tr><td>FATAL</td><td>An error condition was encountered that the software cannot recover from.</td></tr>
</table></td></tr>
</table>

This table is in the F’ User’s Guide[4]. There attributes and description for the optional parameter xml in this table. There more severity values and meanings for the optional event xml in this table.



The ‘Include Cmd XML,’ ‘Include Evr XML,’ and ‘Include Tlm XML’ sections are updated in the Thermometer component, and these optional XML sections will be included when auto-generating the Thermometer component xml file.



These optional xml sections are updated for the Thermal Control component.

```
</commands>
<events>
        <event id="0" name="HM_TEMP_THRESHOLD_UPDATED" severity="ACTIVITY_HI" format_string = "Phase updated to: %d, Min temp updated
to: %f,  Max temp updated to: %f" >
                <args>
                        <arg name="phase" type="U32">
                        </arg>
                        <arg name="minTemp" type="F32">
                        </arg>
                        <arg name="maxTemp" type="F32">
                        </arg>
                </args>
        </event>
        <event id="1" name="HM_TEMP_LO_FATAL" severity="FATAL" format_string = "Fatal: Temperature is less than the min temperature: %f" >
                <args>
                        <arg name="temperature" type="F32">
                        </arg>
                </args>
        </event>
        <event id="2" name="HM_TEMP_HI_FATAL" severity="FATAL" format_string = "Fatal: Temperature is higher than the max temperature: %f" >
                <args>
                        <arg name="temperature" type="F32">
                        </arg>
                </args>
        </event>
  </events>


<parameters>
        <parameter id="0" name="phase" data_type="U32" default="0" set_opcode="16" save_opcode="17">
                <comment>
                phase by HM
                </comment>
        </parameter>
        <parameter id="1" name="temperature" data_type="F32" default="50.0" set_opcode="18" save_opcode="19">
                <comment>
                Termometer value by HM
                </comment>
        </parameter>

        <parameter id="2" name="minTemp" data_type="U32" default="50.0" set_opcode="20" save_opcode="21">
                <comment>
                Threshold min value by HM
                </comment>
        </parameter>

        <parameter id="3" name="maxTemp" data_type="U32" default="100.0" set_opcode="22" save_opcode="23">
                <comment>
                Threshold max value by HM
                </comment>
        </parameter>
</parameters>
```

Close | Back | Forward | Help

```
                                        <arg name="temperature" type="F32">
                                        </arg>
                        </args>
        </event>
        <event id="2" name="HM_TEMP_HI_FATAL" severity="FATAL" format_string = "Fatal: Temperature is higher than the max temperature: %f" >
                        <args>
                                        <arg name="temperature" type="F32">
                                        </arg>
                        </args>
        </event>
</events>


<parameters>
        <parameter id="0" name="phase" data_type="U32" default="0" set_opcode="16" save_opcode="17">
                        <comment>
                        phase by HM
                        </comment>
        </parameter>
        <parameter id="1" name="temperature" data_type="F32" default="50.0" set_opcode="18" save_opcode="19">
                        <comment>
                        Termometer value by HM
                        </comment>
        </parameter>

        <parameter id="2" name="minTemp" data_type="U32" default="50.0" set_opcode="20" save_opcode="21">
                        <comment>
                        Threshold min value by HM
                        </comment>
        </parameter>

        <parameter id="3" name="maxTemp" data_type="U32" default="100.0" set_opcode="22" save_opcode="23">
                        <comment>
                        Threshold max value by HM
                        </comment>
        </parameter>
</parameters>


<telemetry>
        <channel id="0" name="HM_PHASE" data_type="U32">
        </channel>
        <channel id="0" name="HM_MIN_TEMP" data_type="F32">
        </channel>
        <channel id="0" name="HM_MAX_TEMP" data_type="F32">
        </channel>
</telemetry>
Ref
```

Close Back Forward Help

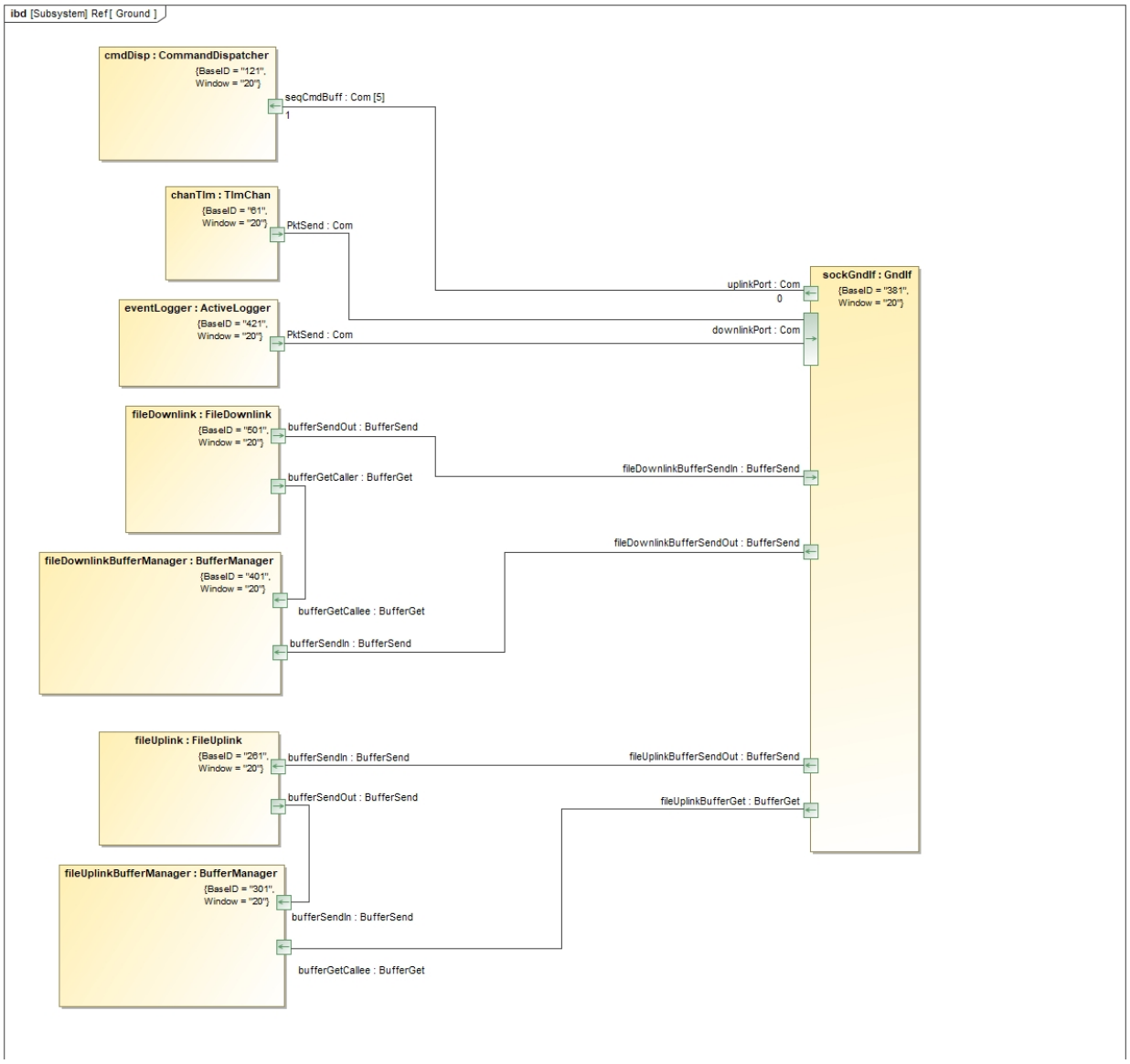
These optional xml sections are updated for the Health Monitor component.

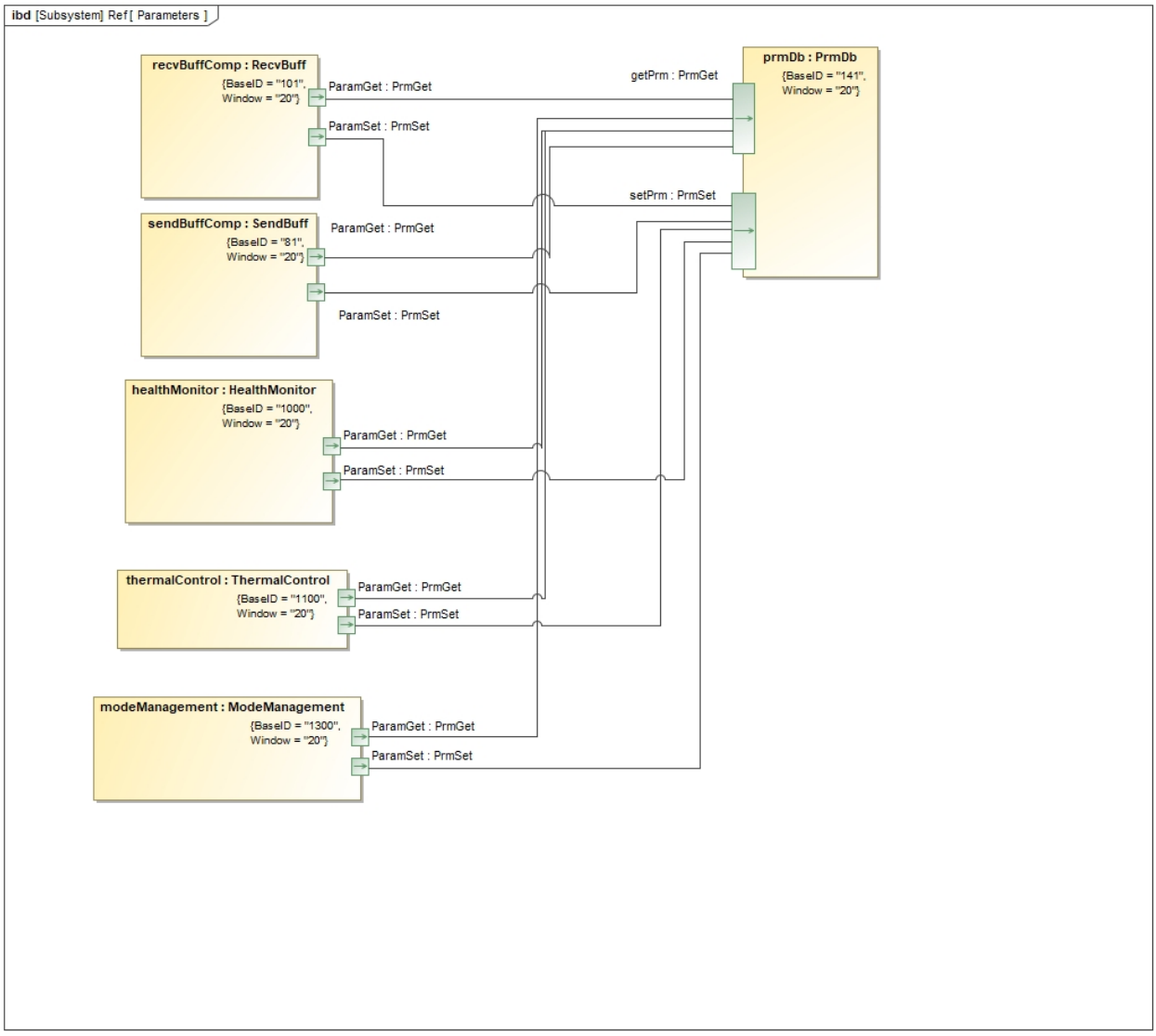
```
        <command kind="async" opcode="0" mnemonic="MM_UPDATE_PHASE">
                <args>
                        <arg name="phase" type="U32">
                        </arg>
                </args>
        </command>
</commands>
<events>
        <event id="0" name="MM_PHASE_UPDATED" severity="ACTIVITY_HI" format_string = "Phase updated to: %d" >
                <args>
                        <arg name="phase" type="U32">
                        </arg>
                </args>
        </event>
        <event id="1" name="MM_SAFETY_MODE_ENTERED" severity="WARNING_HI" format_string = "Warning: Enter the safety mode: %d" >
                <args>
                        <arg name="phase" type="U32">
                        </arg>
                </args>
        </event>
</events>


<parameters>
        <parameter id="0" name="phase" data_type="U32" default="0" set_opcode="8" save_opcode="9">
                <comment>
                phase by MM
                </comment>
        </parameter>
        <parameter id="1" name="temperature" data_type="F32" default="50.0" set_opcode="10" save_opcode="11">
                <comment>
                Termometer value by MM
                </comment>
        </parameter>
        <parameter id="2" name="minTemp" data_type="F32" default="50.0" set_opcode="12" save_opcode="13">
                <comment>
                Threshold min value by MM
                </comment>
        </parameter>

        <parameter id="3" name="maxTemp" data_type="U32" default="100.0" set_opcode="14" save_opcode="15">
                <comment>
                TThreshold max value by MM
                </comment>
        </parameter>
</parameters>


<telemetry>
        <channel id="0" name="MM_PHASE" data_type="U32">
        </channel>
</telemetry>
```

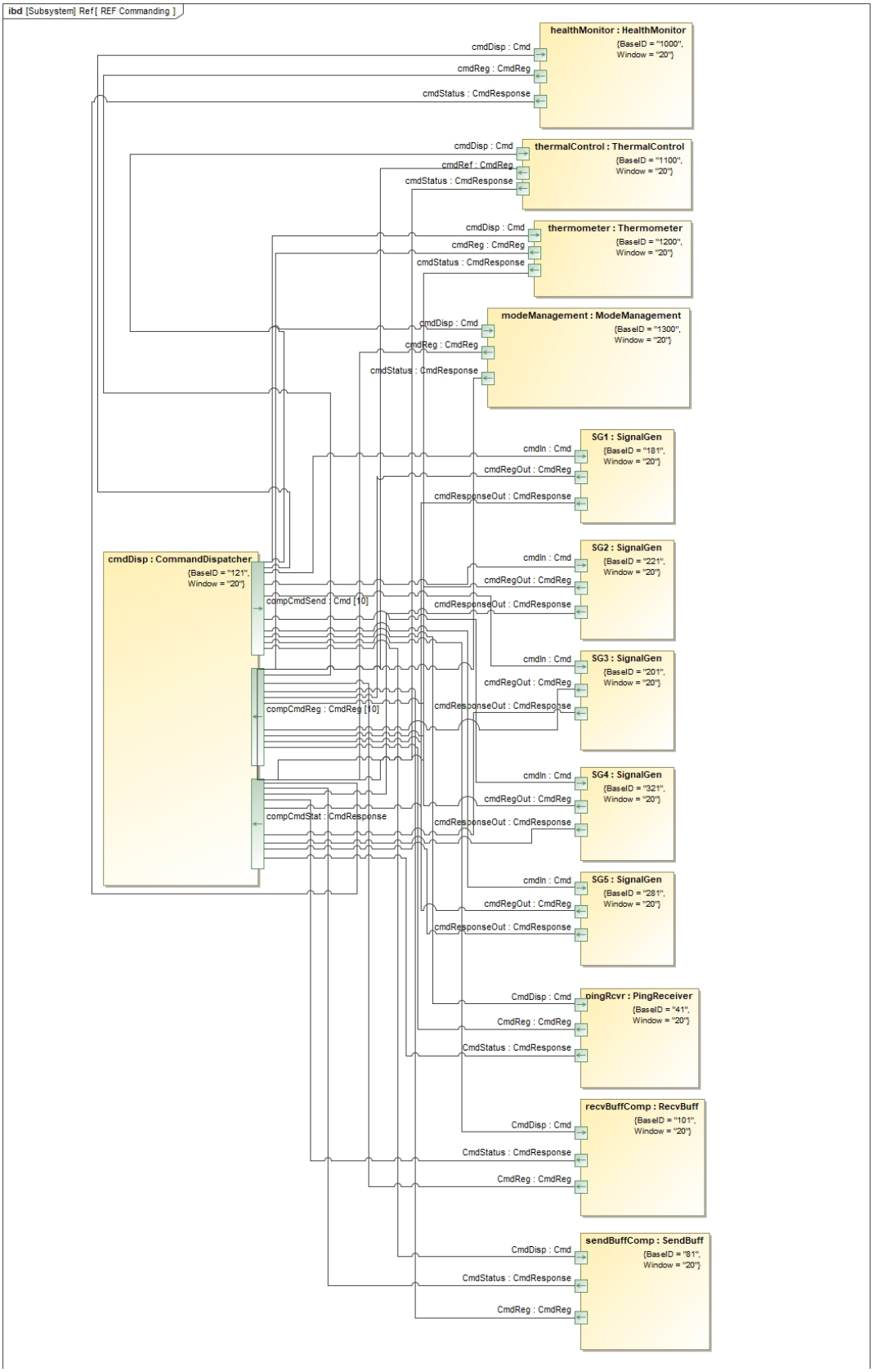
Close Back Forward Help

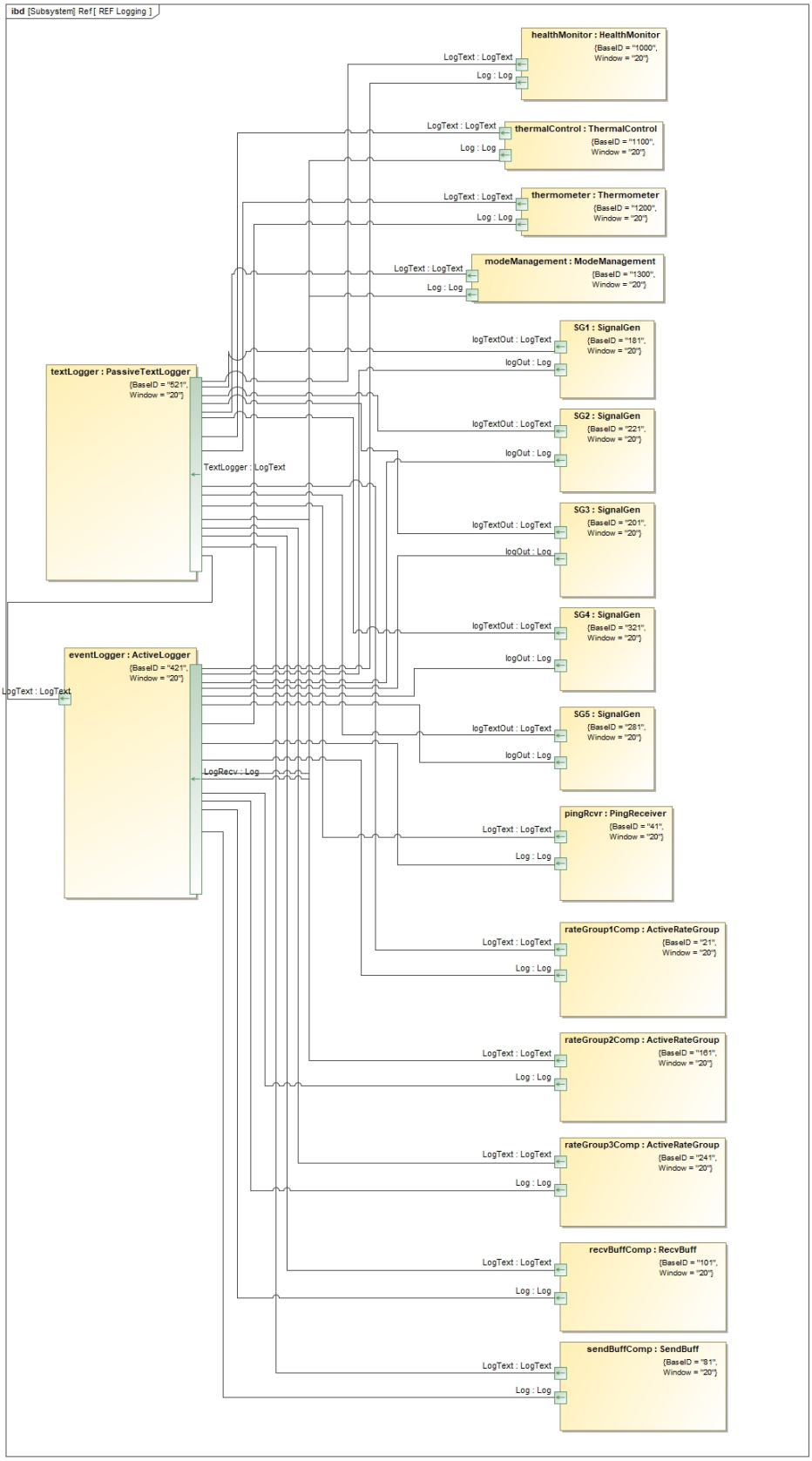
These optional xml sections are updated for the Mode Management component.

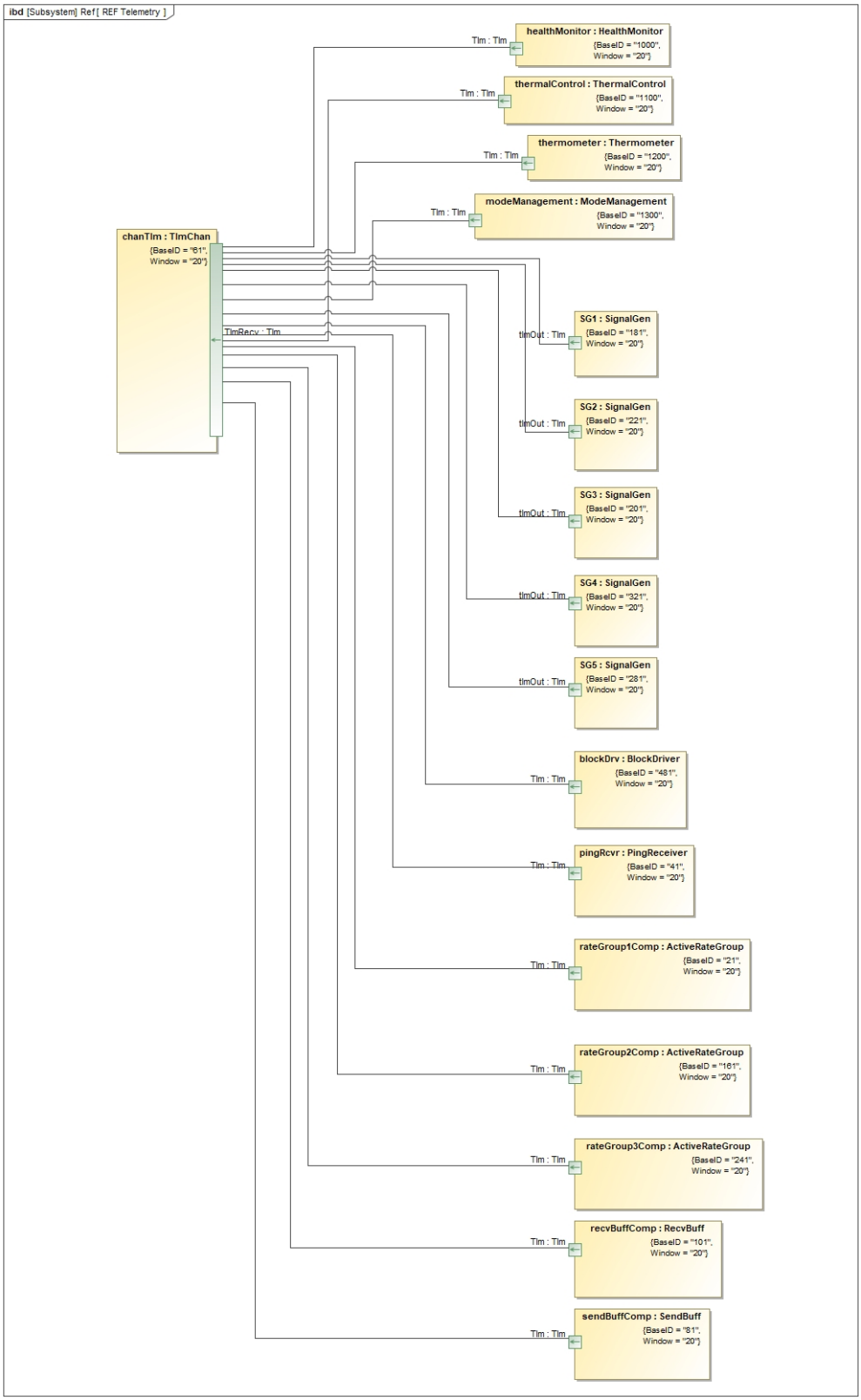
4.11 Inertial Topology Diagrams

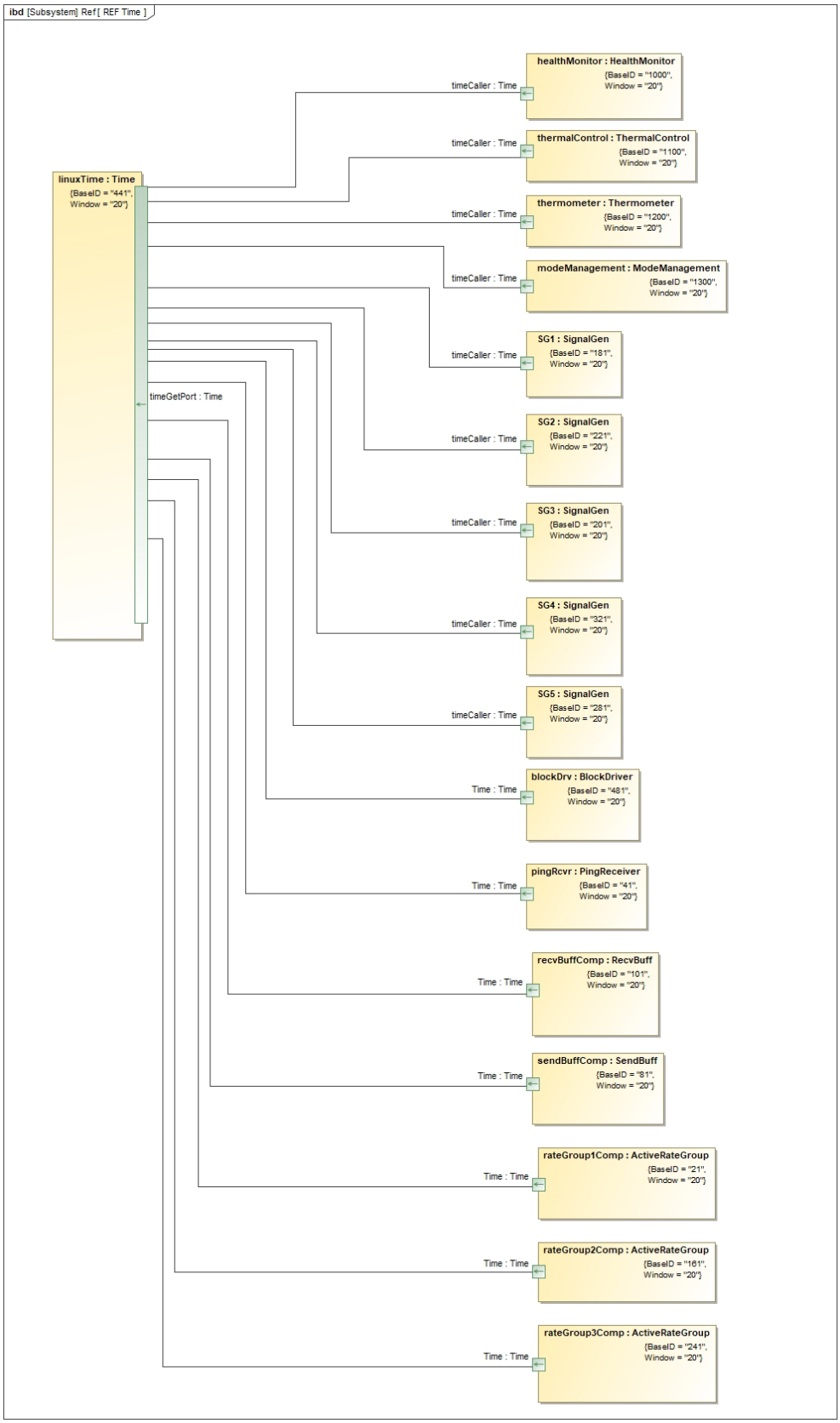


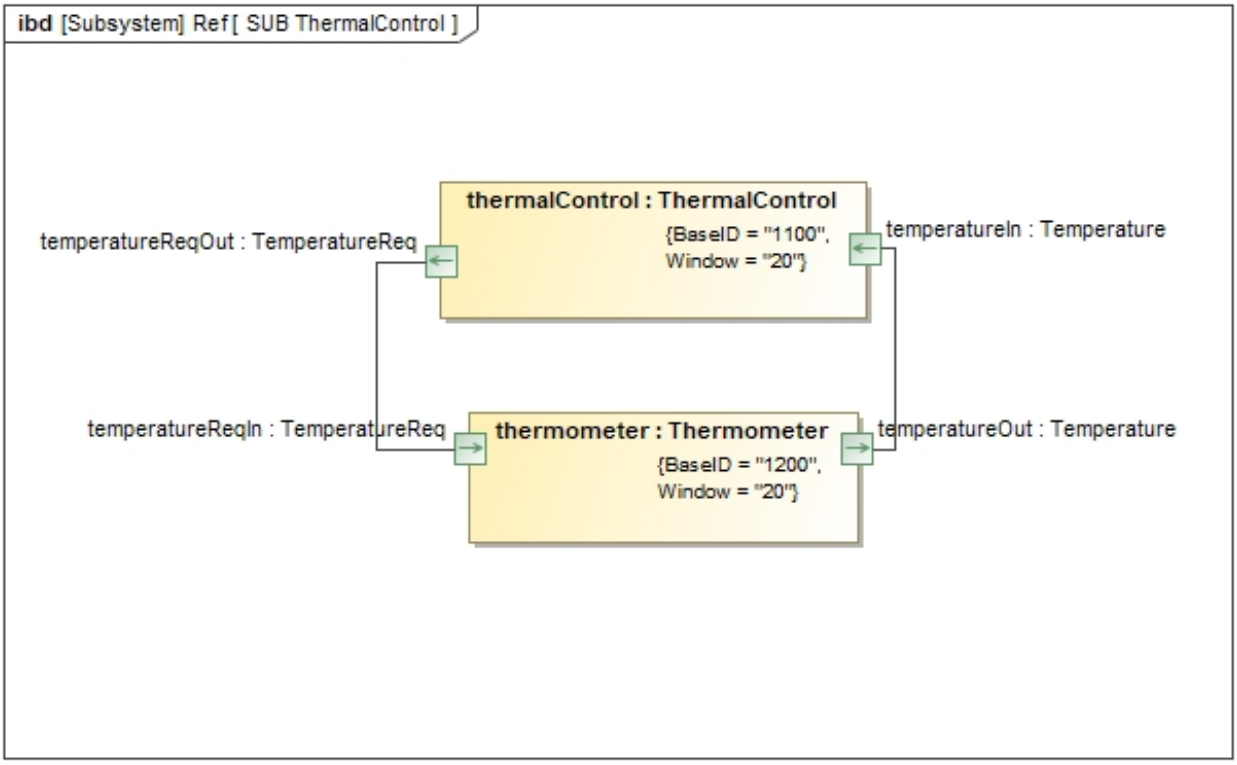

ibd [Subsystem] Ref [Parameters]
recvBuffComp : RecvBuff
{BaseID = "101",
Window = "20"}
ParamGet : PrmGet
ParamSet : PrmSet
sendBuffComp : SendBuff
{BaseID = "81",
Window = "20"}
ParamGet : PrmGet
ParamSet : PrmSet
healthMonitor : HealthMonitor
{BaseID = "1000",
Window = "20"}
ParamGet : PrmGet
ParamSet : PrmSet
thermalControl : ThermalControl
{BaseID = "1100",
Window = "20"}
ParamGet : PrmGet
ParamSet : PrmSet
modeManagement : ModeManagement
{BaseID = "1300",
Window = "20"}
ParamGet : PrmGet
ParamSet : PrmSet
prmDb : PrmDb
{BaseID = "141",
Window = "20"}
getPrm : PrmGet
setPrm : PrmSet


ibd [Subsystem] Ref [REF Commanding]
healthMonitor : HealthMonitor
{BaseID = "1000", Window = "20"}
cmdDisp : Cmd
cmdReg : CmdReg
cmdStatus : CmdResponse
thermalControl : ThermalControl
{BaseID = "1100", Window = "20"}
cmdRef : CmdReg
thermometer : Thermometer
{BaseID = "1200", Window = "20"}
modeManagement : ModeManagement
{BaseID = "1300", Window = "20"}
SG1 : SignalGen
{BaseID = "181", Window = "20"}
cmdIn : Cmd
cmdRegOut : CmdReg
cmdResponseOut : CmdResponse
SG2 : SignalGen
{BaseID = "221", Window = "20"}
SG3 : SignalGen
{BaseID = "201", Window = "20"}
SG4 : SignalGen
{BaseID = "321", Window = "20"}
SG5 : SignalGen
{BaseID = "281", Window = "20"}
cmdDisp : CommandDispatcher
{BaseID = "121", Window = "20"}
compCmdSend : Cmd [10]
compCmdReg : CmdReg [10]
compCmdStat : CmdResponse
pingRcvr : PingReceiver
{BaseID = "41", Window = "20"}
CmdDisp : Cmd
CmdReg : CmdReg
CmdStatus : CmdResponse
recvBuffComp : RecvBuff
{BaseID = "101", Window = "20"}
sendBuffComp : SendBuff
{BaseID = "81", Window = "20"}


ibd [Subsystem] Ref [REF Logging]
healthMonitor : HealthMonitor
{BaseID = "1000", Window = "20"}
LogText : LogText
Log : Log
thermalControl : ThermalControl
{BaseID = "1100", Window = "20"}
thermometer : Thermometer
{BaseID = "1200", Window = "20"}
modeManagement : ModeManagement
{BaseID = "1300", Window = "20"}
SG1 : SignalGen
{BaseID = "181", Window = "20"}
logTextOut : LogText
logOut : Log
SG2 : SignalGen
{BaseID = "221", Window = "20"}
SG3 : SignalGen
{BaseID = "201", Window = "20"}
SG4 : SignalGen
{BaseID = "321", Window = "20"}
SG5 : SignalGen
{BaseID = "281", Window = "20"}
textLogger : PassiveTextLogger
{BaseID = "521", Window = "20"}
TextLogger : LogText
eventLogger : ActiveLogger
{BaseID = "421", Window = "20"}
LogRecv : Log
pingRcvr : PingReceiver
{BaseID = "41", Window = "20"}
rateGroup1Comp : ActiveRateGroup
{BaseID = "21", Window = "20"}
rateGroup2Comp : ActiveRateGroup
{BaseID = "161", Window = "20"}
rateGroup3Comp : ActiveRateGroup
{BaseID = "241", Window = "20"}
recvBuffComp : RecvBuff
{BaseID = "101", Window = "20"}
sendBuffComp : SendBuff
{BaseID = "81", Window = "20"}


ibd [Subsystem] Ref [REF Telemetry]
healthMonitor : HealthMonitor
{BaseID = "1000",
Window = "20"}
thermalControl : ThermalControl
{BaseID = "1100",
Window = "20"}
thermometer : Thermometer
{BaseID = "1200",
Window = "20"}
modeManagement : ModeManagement
{BaseID = "1300",
Window = "20"}
chanTlm : TlmChan
{BaseID = "61",
Window = "20"}
TlmRecv : Tlm
SG1 : SignalGen
{BaseID = "181",
Window = "20"}
SG2 : SignalGen
{BaseID = "221",
Window = "20"}
SG3 : SignalGen
{BaseID = "201",
Window = "20"}
SG4 : SignalGen
{BaseID = "321",
Window = "20"}
SG5 : SignalGen
{BaseID = "281",
Window = "20"}
blockDrv : BlockDriver
{BaseID = "481",
Window = "20"}
pingRcvr : PingReceiver
{BaseID = "41",
Window = "20"}
rateGroup1Comp : ActiveRateGroup
{BaseID = "21",
Window = "20"}
rateGroup2Comp : ActiveRateGroup
{BaseID = "161",
Window = "20"}
rateGroup3Comp : ActiveRateGroup
{BaseID = "241",
Window = "20"}
recvBuffComp : RecvBuff
{BaseID = "101",
Window = "20"}
sendBuffComp : SendBuff
{BaseID = "81",
Window = "20"}
Tlm : Tlm
tlmOut : Tlm


ibd [Subsystem] Ref [REF Time]
linuxTime : Time
{BaseID = "441",
Window = "20"}
timeGetPort : Time
healthMonitor : HealthMonitor
{BaseID = "1000",
Window = "20"}
thermalControl : ThermalControl
{BaseID = "1100",
Window = "20"}
thermometer : Thermometer
{BaseID = "1200",
Window = "20"}
modeManagement : ModeManagement
{BaseID = "1300",
Window = "20"}
SG1 : SignalGen
{BaseID = "181",
Window = "20"}
SG2 : SignalGen
{BaseID = "221",
Window = "20"}
SG3 : SignalGen
{BaseID = "201",
Window = "20"}
SG4 : SignalGen
{BaseID = "321",
Window = "20"}
SG5 : SignalGen
{BaseID = "281",
Window = "20"}
blockDrv : BlockDriver
{BaseID = "481",
Window = "20"}
pingRcvr : PingReceiver
{BaseID = "41",
Window = "20"}
recvBuffComp : RecvBuff
{BaseID = "101",
Window = "20"}
sendBuffComp : SendBuff
{BaseID = "81",
Window = "20"}
rateGroup1Comp : ActiveRateGroup
{BaseID = "21",
Window = "20"}
rateGroup2Comp : ActiveRateGroup
{BaseID = "161",
Window = "20"}
rateGroup3Comp : ActiveRateGroup
{BaseID = "241",
Window = "20"}
timeCaller : Time
Time : Time



5 F Prime

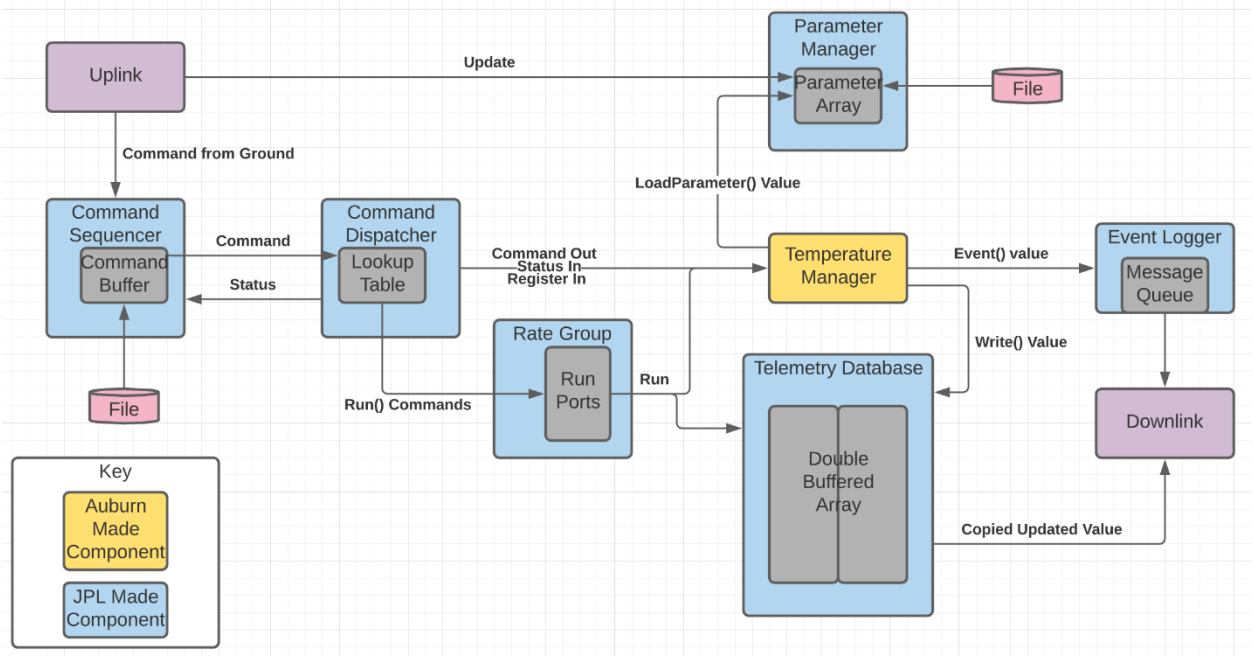
5.1 Background

F Prime is a FSW framework created by engineers at NASA's Jet Propulsion Lab and provides a simplified way to implement FSW with a component-based architecture. This framework handles the conversion of high-level XML design documents to C++ base classes, with communication between components and task scheduling implemented automatically. Developers are left only to implement the specific behaviors of each component, alleviating some of the more time-consuming and low-level work as well as ensuring those generated features meet a high standard of quality.

5.2 Usage

F Prime allows for the encapsulation of FSW functionality into components that communicate with each other through ports. Ports can be classified as either input or output ports with several sub-groups and characteristics that further specify their functionality. Input ports collect data to be used within the component it is attached to, while output ports collect data that will be sent to the input port of some other component in the system. Developers are tasked with defining these components and their ports as well as specifying the links between ports of separate components, but F Prime implements the actual communication between those ports.

5.3 Example component design



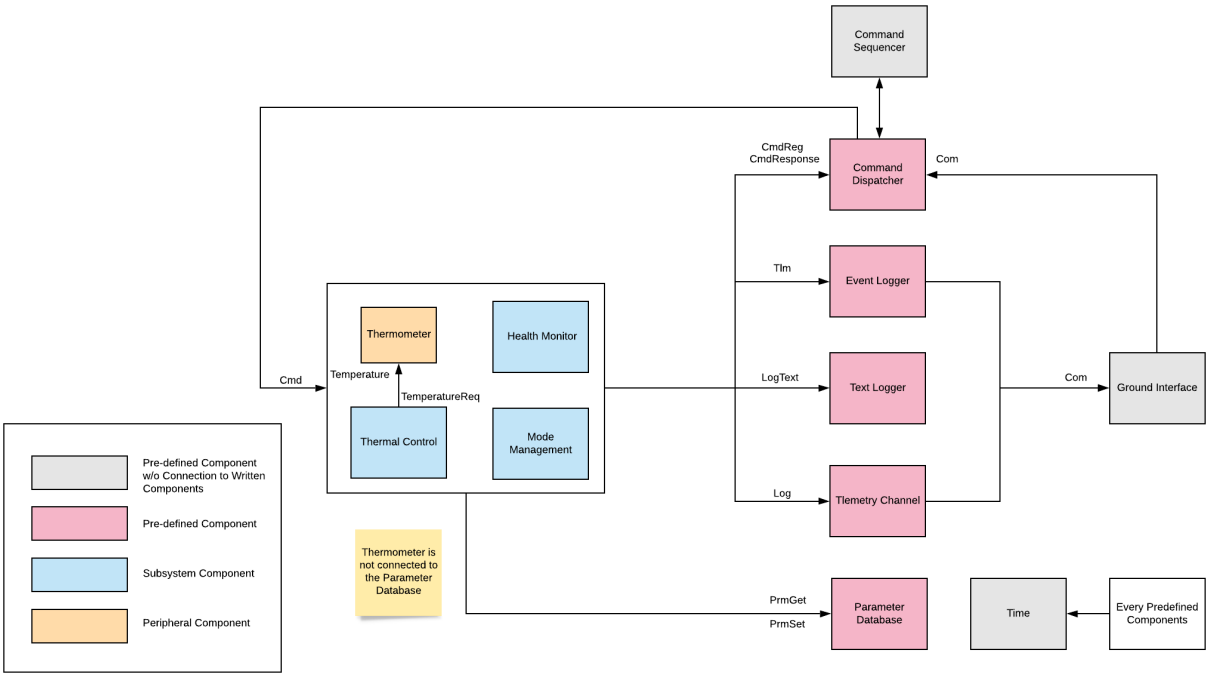
This is a diagram showing how a component will interact with the premade JPL backbone.

The above figure depicts how the FSW will interact with the JPL backbone. Any component will be able to substitute in where the temperature component is. The components use the backbone that the temperature component is connected to in order to communicate with all other components on the satellite. All of the blue boxes in the above diagram represent a component that was made by JPL and is provided via the FPrime github. The yellow box represents a component made in house at Auburn University.

For this particular example, a command would come into the system either from the uplink or from a different component that would then move to the command sequencer. From the command sequencer, the command would wait until it is at the front of the buffer before being passed over to the command dispatcher. This command's opcode would then be looked up via the look up table and then sent to the component. The component chosen is a temperature manager. The temperature manager is in charge of reading a thermometer. The temperature manager would take data from the thermometer and compare it to values stored in the parameter manager. If the temperature was not within the range specified, the temperature manager would generate an event. This event would say whether the temperature has either exceeded or subceeded the requirment. Once this event has been generated, a new command would be added to the sequencer which would make its way to another component following the same path listed above and would eventually generate an event which launches another command. This process continues as the satellite runs itself.

Note: All components will be connected to the JPL backbone. Some components will be connected to one another directly if they are part of the same subsytem. However, most of the interaction between the components made should be through this predefined backbone. This information can be referenced in the FPrime User Guide [4] and FPrime Software Framework [5].

5.4 Abstracted Topology Diagram Design for Subsystem Components



There are 4 written components and 8 pre-defined components in this abstracted subsystem topology diagram. Out of 4 written components, 3 components are the pre-defined components, and 1 is periperal component. The thermomal control and thermometer are connected through two written ports that one of them are Temperature Type, and the other is TemperatureReq type. The pre-defined components are connected through pre-defined ports. CmdReg port type is to register commands to the command dispatcher component. Cmd port type is to send the commands to other components from the command dispatcher component. CmdResponse port type is to send command responses to the command dispatcher coponent. Tlm port type is to send measurable values to the telemetry chanel, and the values will be shown in the telemetry tab on the ground control interface. Log is to send event logs to the Event Logger component. LogText is to send and see the logs in text by sending the text to the Text Logger component.

5.5 No Target Error while Generating C++ Files

```
make: *** No rule to make target 'Ref_ModeManagement_impl'.  Stop.
[ERROR] CMake erred with return code 2.
```

If thie error message is shown when using the “fprime-util impl” command in the component directory, that means there is no Makefile or mod.mk files in the component.

```
# Makefile to run global make.
DEPLOYMENT := Ref
BUILD_ROOT ?= $(subst /$(DEPLOYMENT),,$(CURDIR))

export BUILD_ROOT

default_build: all dict_install

include $(BUILD_ROOT)/mk/makefiles/deployment_makefile.mk

```

In the namespace directory, Makefile with this format should be used. The ‘Ref’ can be changed to another namespace if it is not ‘Ref’.

```
MODULE_DIR = Ref/Thermometer
MODULE = $(subst /,,$(MODULE_DIR))

BUILD_ROOT ?= $(subst /$(MODULE_DIR),,$(CURDIR))
export BUILD_ROOT

include $(BUILD_ROOT)/mk/makefiles/module_targets.mk
```

In a component ‘Thermometer’ under the namespace ‘Ref’, Makefile with this format should be used.

```
SRC = ThermometerComponentAi.xml
```

In the same directory, mod.mk file should be created in this format like SRC = component xml file name.

```
REF_MODULES := \
        Ref/Top \
        Ref/RecvBuffApp \
        Ref/SendBuffApp \
        Ref/SignalGen \
        Ref/PingReceiver\
        Ref/Thermometer
```

In the fprime/mk/configs/modules directory, click the modules.mk file to add the component under REF_MODULES. The format is Namespace/component.

```
make gen_make
```

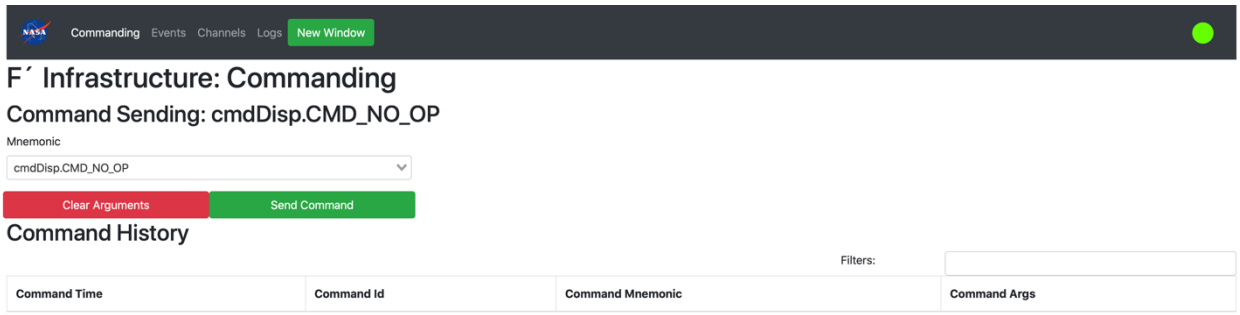
```
make rebuild
```

These two commands “make gen_make” and “make rebuild” should be typed one after the other in the component directory. Then, it will generate ____ComponentAc.cpp and ____ComponentAc.hpp files.

```
make impl
```

To generate the ____ComponentImpl.cpp and ____ComponentImpl.hpp files, the “make impl” command needs to be typed.

5.6 Commands for Running Demo



- Address: https://github.com/ljyjl/fprime/tree/devel-leah
- To install F Prime and run the demo, follow these commands:

  python3 -m venv ./<venv_folder_name>

  . ./ <venv_folder_name>/bin/activate

  git clone https://github.com/ljyjl/fprime <fprime_folder_name>

  cd <fprime_folder_name>

  git checkout devel-leah

  pip install ./Fw/Python

  pip install ./Gds

  cd Ref

  fprime-util generate

  fprime-util build-all

  fprime-util install

  fprime-gds -d .

5.7 Useful Links

- fprime-util.md: https://github.com/nasa/fprime/blob/devel/docs/UsersGuide/user/fprime-util.md
- Math Tutorial: https://github.com/nasa/fprime/blob/master/docs/Tutorials/MathComponent/Tutorial.md
- GPS Tutorial: https://github.com/nasa/fprime/blob/master/docs/Tutorials/GpsTutorial/Tutorial.md
- Previous team's tutorial: https://github.com/Aleha-C/fprime/tree/AEGIS_Assignment
- RPI README.md: https://github.com/nasa/fprime/blob/master/RPI/README.md

6 Operating System

6.1 OS Overview

The operating system is the glue between F Prime and the satellite hardware. A well-designed OS should utilize drivers to effectively access all hardware components connected to the computer and provide a high-level toolbox to the application developers.

The F Prime software framework can run with or without an OS. An OS provides three main features: thread management, resource management, and a hardware abstraction layer. These are extremely useful tools to manage concurrency and structure within a software system.

The current state of the system is a Linux Kernel 3.18 operating system running PREEMP_RT.

6.2 OS Survey

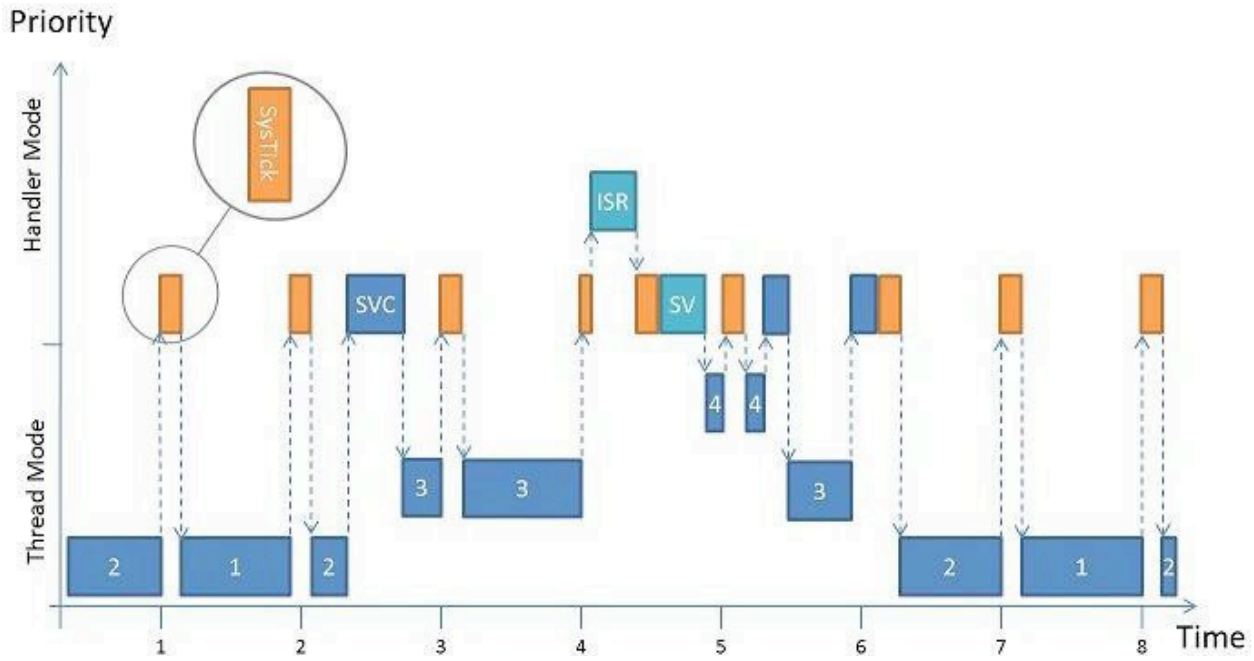
A survey of potential OS choices has been conducted, in which several factors were taken into consideration. First, compatibility. F Prime can operate off a POSIX API. Linux, and most embedded systems support the POSIX API. At the time of writing this document, the satellite will be using a Space Micro CubeSat Processor OBC. Space Micro, the computer company, has built toolchains for the board. Finally, community support is a large factor in OS selection. A strong community and good documentation will help out greatly when problems arise. The below depicts the results of the survey. The results of this survey were gathered under the assumption that a RISC-V processor would be used, but the results still apply.

Traits:	RTEMS	FreeRTOS	Linux
Compatibility	Compatible with 32 bit ARM	Compatible with 32 bit ARM	Compatible with 32 bit ARM
Support and Documentation	Reasonable amount of documentation	Less than Linux but still decent	Extensive documentation
Features	Good: Includes assorted libraries and device driver frameworks.	OK: Easy to learn, some assmebly required.	Excellent: Extensive libraries, drivers, and real time options
Foorprint	Small	Tiny	Larger than the others. Can be reduced
Real Time Capabilities	Comes pre-packaged	Comes pre-packaged	Has to be patched in with PREEMPT-RT
Cost	Free, but learning materials and training is expensive.	Free, however some tools are locked behind a paywall	Completely free
Developer Familiarity	None	None	Most students have extensively used Linux

List of possible operating systems.

6.3 GPOS vs RTOS

While researching Operating Systems for satellite use, the term Real-Time Operating system was a recurring suggestion. Many embedded operating systems are designed to operate in “real-time”. This means that the OS is guaranteed to meet “hard” deadlines. A hard deadline is a task that the computer has to do in a certain amount of time, otherwise the mission might be compromised. The way these OS’s meet hard deadlines is typically through a deterministic, preemptive, priority-based scheduler. This type of scheduler assigns priorities to threads and always executes the highest priority thread. In order to make sure that the highest priority thread is running, the OS is interrupted at a certain interval and the highest priority thread is run. Assuming no resource starvation, the system can respond within one of these interrupts or “ticks”.



Graphical representation of RTOS scheduling.

6.4 Primary OS Candidate

Linux is the most straightforward choice for an OS. It has been used in embedded systems, including space systems. Support for major Linux distros would be greater than RTEMS. F Prime natively compiles to Linux. It is also suspected that most device drivers will be written for Linux, reducing the number of drivers that must be created specifically for this project. Linux also is capable of real time operation with the PREEMPT_RT patch and several choices of real time capable scheduling classes. However, this choice would potentially be outperformed by a more traditional RTOS, as evidenced by the ASTERIA mission. It is possible that these issues could be mitigated through further development to the kernel that has occurred, more abundant system resources, or through scheduler choice. More evaluation on issues they faced with this configuration is required in this area.

6.5 Secondary OS Candidate

RTEMS is the main RTOS candidate as of now. It is a flight proven RTOS that has decent community and device support. RTEMS is open-source and documentation can be found at the RTEMS Documentation Project. F Prime has been ported to RTEMS in the past, and work is currently being done by members of the community to integrate compatibility with RTEMS 5 to F Prime.

Some headway has been made in integrating RTEMS into the development environment, but this is yet to be completed. Thus far, the RTEMS toolchain has been compiled and some sample applications are working. Initially, a demo was created showing that RTEMS can run on RISC-V which may prove useful in understanding the RTEMS toolchain going forward, but it is no longer necessary for development due to the change in processor architecture. The next major step if RTEMS is selected will be to compile an F Prime application using the RTEMS toolchain.

6.6 Bootloader Considerations

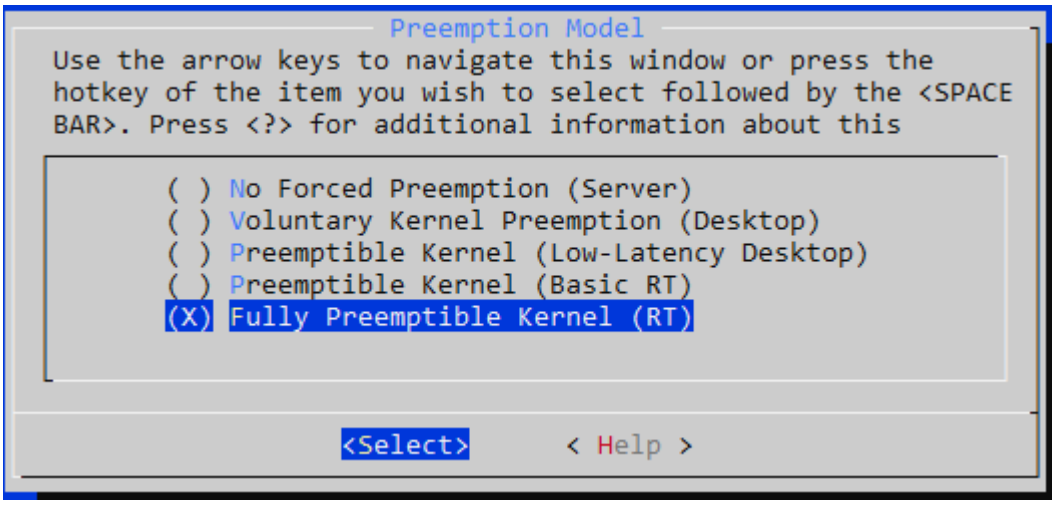
The bootloader is the software loaded by the firmware with the purpose of starting the operating system. Due to system constraints at boot time, they often have two stages. In this configuration, the first stage loads the second stage, which then loads the operating system. For a more common computer system any number of bootloaders may be available, but for the types of single board computers we are evaluating a working bootloader is usually supplied with the documentation and software for the board. In the case of Linux systems, the bootloader is responsible for setting kernel parameters. The bootloader is an important component of the system, and as such most bootloaders have already been extensively tested and documented so their operations should be transparent to the user.

6.7 Yocto Vs. Traditional Kernel Development

The Yocto Project is a tool to create Linux distributions for development of IoT and embedded system software. It allows for the developer to create custom distributions ranging from tiny to full images. It includes documentation and support for testing images under QEMU. The advantages to using the Yocto Project is the ease of which packages can be included in the initial install, especially when they have already been created before. This project will not greatly benefit from such capability as it is not a complicated Operating System. The biggest drawback to using the Yocto Project is the learning curve. For this reason, using traditional kernel development should be easier for someone who is new to creating an embedded Linux OS and drivers. For more info on this see section 4.12.

6.8 Kernel Selection

The main considerations when selecting a Kernel are stability and real time capabilities. For this project, the kernel does not need to have a lot of features outside of these. PREEMPT-RT was chosen because it is what Space-X and JPL use which speaks to its reliability. It is free and somewhat easy to patch in. For more on this decision see section 4.5



Patching in real time capabilities.

6.9 What the OS Needs to Support

The OS's only function is to support the device drivers and to support F Prime.

6.10 Operating Systems in Development

Umbo is a lightweight operating system based on Linux kernel 4.19 that uses Debian packages. To develop umbo the first step that was taken was to create a list of all the system calls and remove anything unnecessary. Next PREEMPT_RT was patched in to give the OS real-time capabilities.

This OS has been successfully simulated on QEMU. Another option was to use the kernel that came with the development board. This is an attractive option for this phase of development because there is not a custom board yet and all the drivers will already be there. The development board came with kernel 3.15. However, this Kernel does not have a patch to make it real-time. To deal with this issue the kernel that came with the development board was patched first to 3.18 and then it was possible to patch in PREEMP_RT.

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
# uname -r
4.19.87-rt30-yocto-preempt-rt
```

Umbo Linux distribution

6.11 Drivers

The OS will need a way to talk with its peripherals. To accomplish this, driver will need to be developed. The current contenders for communication protocols are CAN bus, SPI and RS422 serial. The primary difference between CAN bus and SPI is that CAN bus defines the protocol. While SPI only defines how bytes will be transmitted over the serial connection. The advantage to CAN bus is that it is already well defined, it can detect collisions, and some errors. The main drawbacks are that it requires some sort of addressing mechanism and may be more complicated to set up. RS422 allows for any type of packet to be sent in a bit stream. It is cheaper and will require less development time.

6.12 Developing the OS.

When developing the OS it is important for our purposes to consider stability and simplicity. A CubeSat does not generally need anything complicated as far as an operating system goes. There are multiple ways to go about building an OS. Usually simplicity is the best as it leaves the least room for something to go wrong. One can develop and OS with the Yocto Project however for this purpose relying on traditional kernel development may be simpler as the OS does not need to be very complicated.

6.12.1 Yocto Development

To get started with the yocto project one will need a linux os. Ubuntu is an easy to use and generally well supported os for this purpose. There are a few basic packages one will need. These packages can be found listed on the Yocto Project website. After these basic requirements have been procured one can start building. Once a linux OS has been setup with these packages on can clone the Yocto Project with the following command:

$ git clone git://git.yoctoproject.org/poky

Once this had been completed one needs to source the build environment. This file is found in the poky directory that was just cloned:

$ source oe-init-build-env

Finally after doing tese two tasks one can run bitbake to create an operating system:

$ bitbake core-image-minimal

These are the basic steps to using the Yocto Project. Once this process has been completed one delve further into the Yocto Documentation which can be found on the Yocto Project website. Furthermore the next section contains debugging notes for using the Yocto Project.

6.12.2

The purpose of these notes is to bridge the gap between the simple tutorials online and the documentation along with general debugging tips. This is not a comprehensive guide these are the things I figured out on my own that seemed hard to find online. To learn you should first do the quick-start tutorial here: https://www.yoctoproject.org/docs/3.1.2/brief-yoctoprojectqs/brief-yoctoprojectqs.html There are many other basic tutorials online to get started with.

Then skim the following documentation: https://www.yoctoproject.org/docs/3.1.2/overview-manual/overview-manual.html

Finally this page while lengthy should contain everything you will need: https://www.yoctoproject.org/docs/3.1.2/ref-manual/ref-manual.html#ref-devtool-reference

Important Files:

Poky/meta-debian/conf/include

umbo.conf

umbo.inc

-These specifies the distro

Poky/meta-debian/conf/include/Debian-preferred-provider.inc

-This is where one decides kernel version

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto-rt"
PREFERRED_VERSION_linux-yocto ?= "4.19"
```

Poky/build/conf/local.conf

-This is where I set my distro to

```
# The distribution setting controls which policy settings are used as defaults.
# The default value is fine for general Yocto project use, at least initially.
# Ultimately when creating custom policy, people will likely end up subclassing
# these defaults.
#
DISTRO ?= "umbo"
```

-To set a package manager I added this line

```
PACKAGE_CLASSES ?= "package_ipk"
```

-In this area one can decide between different package managers (ie aptitude) I chose IPK because it is lightweight.

-I also added “package-management” to EXTRA_IMAGE FEATURES

```
EXTRA_IMAGE_FEATURES = "debug-tweaks package-management"
```

-To allow packaging I added the following two lines to my conf file

```
WKS_FILE="mkefidisk.wks"
IMAGE_FSTYPES += "wic wic.bmap"
```

Known issue:

When creating a .wic file bitbake will need the image file. You will get a an error that looks like this

```
ERROR: Task (/home/theo/poky/meta/recipes-rt/images/core-image-rt.bb:do_image_wic) failed with exit code '1'
```

If you look further up the stack trace you will see something like this:

```
output: install: cannot stat '/home/theo/poky/build/tmp/deploy/images/qemuarm/bzImage': No such file or director
```

It cannot find an image called bzImage in machine directory. However, there is a file called “zImage”

```
theo@debian:~/poky/build/tmp/deploy/images/qemuarm$ ls
modules--4.19.87+git0+4f5d761316_d3fb163023-r0-qemuarm-20200919155037.tgz
modules-qemuarm.tgz
versatile-pb--4.19.87+git0+4f5d761316_d3fb163023-r0-qemuarm-20200919155037.dtb
versatile-pb.dtb
versatile-pb-qemuarm.dtb
zImage
zImage--4.19.87+git0+4f5d761316_d3fb163023-r0-qemuarm-20200919155037.bin
zImage-qemuarm.bin
```

First run

```
$ bitbake -c do_cleansstate core-image-rt
```

Then rename the image file zImage to bzImage You can then rerun bitbake and the error will be gone.

How to create an SDK to develop the kernel using devtools:

For this you will need to run the normal bitbake build command with the extension

"-c populate_sdk_ext" This will often cause issues with bitbake see general debugging notes for this.

If you are creating a .wic file you also may need to change zImage to bzImage see above in local.conf to do this.

Once this is done you will go to the sdk directory and source the build environment

Finally you can run devtool commands to modify the kernel (ie $devtool modify kernel-name)

To get more info on devtool type in: $ devtool -h

For more on kernel development use: https://www.yoctoproject.org/docs/3.1.2/kernel-dev/kernel-dev.html

General Debugging Notes:

If there are errors with bitbake the easiest solution is to save your conf files in the build directory and then delete the whole things and replace the conf files. This will take some time to run bitbake again but will ensure a clean build.

There are other options but most of them take longer. For example:

$bitbake -c cleanall <package name>

This will clean the package you selected but if there are multiple issues it will be hard to track them all down. I only recommend doing this approach if you have a slow internet connection or a slow CPU without a lot of threads. Often times when creating an sdk for your image cleaning the build files is needed.

Finally, always start with the simplest build. You can add additional things as you go. You need to be sure everything is compatible, which will save a lot of headache. Runing Yocto on the recommended OS will save a lot of headache. Preferably native if possible. I tried Manjaro, Ubuntu and Debian. Debian had the least issues and performed the best. If this is not possible use virtual box. When using virtual box consider the following: Make sure to get the extension pack which can be found here: https://www.virtualbox.org/wiki/Downloads this will add USB 3 support along with multiple monitors. To enable copy and paste one needs to insert the VBoxLinuxAddtions CD. It does have an autorun, but it may not work on some systems. I recommend going to the mounting

point and running the script manually. There also are some dependencies that can be installed with: $ *sudo dnf install -y dkms kernel-devel kernel-devel-$(uname -r)*. With the addition of these tools Virtual box is almost as good as a traditional native environment.

6.12.3 Editing the Kernel

To edit the kernel one will first need to download and extract the kernel. The kernel from the previous semester will be provided to you. You can tar the file for example:

```
$ tar xvfj linux-xilinx.tar.bz
```

This will give you the source code that you need to edit. For example to add text to display on start up one can add to the file “init/calibrate.c” Bellow is a screenshot of added code.

```
void calibrate_delay void

    unsigned long lpj
    static bool printed
    int this_cpu    smp_processor_id

    printk "Welcome to the NASA cubesat flight software team!\n"
```

After the needed code has been added one can compile the kernel. This will require a cross compiler for the desired hardware. To see more on this review section 5 on setting up the development board.

7 Peripherals and Communications

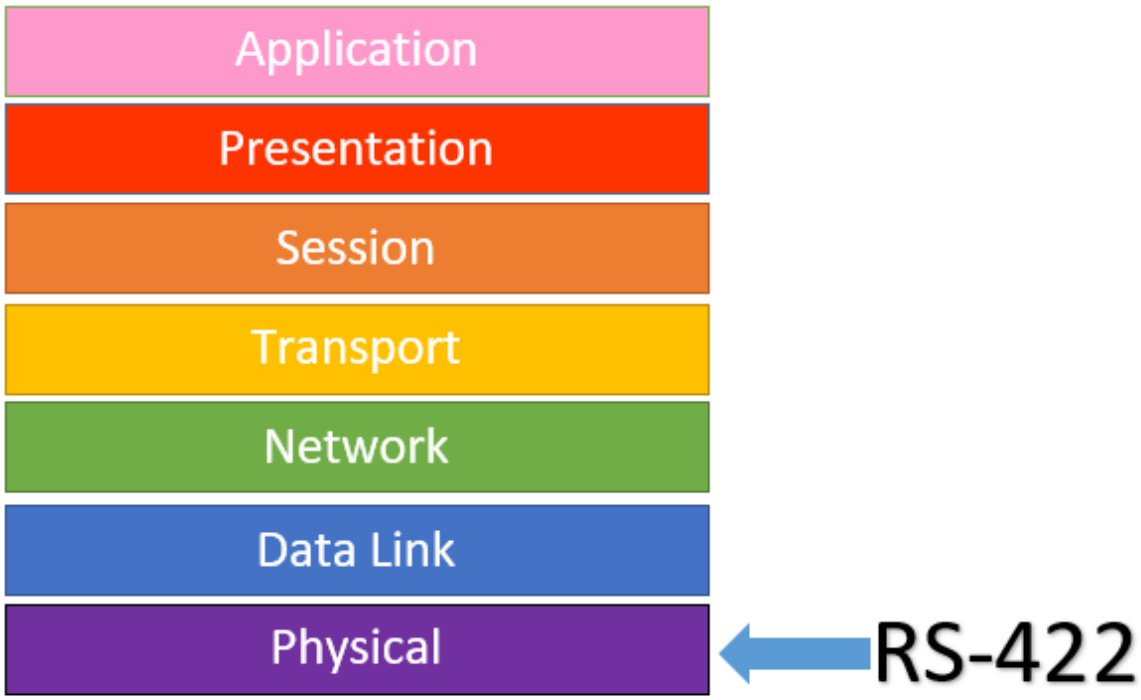
7.1 Serial Vs Parallel

Serial only transmits one bit at a time while parallel can transfer multiple bits at a time. At first it would seem that parallel should be much faster than serial. However, this is not the case. Parallel connections suffer from noise known as inter-symbol interference caused by the induction between neighboring wires. This allows the data being transported to be easily corrupted. This noise can be alleviated by putting in breaks in the signal. This will slow down the connection countering the advantage of having multiple signals. Another issue with parallel communication is clock skew. Clock skew can be caused by physical features of the wire being temperature, differing length, or other slight physical differences in the wire. Clock skew is when signals sent at the same time arrive at the receiver at different times. Clock skew reduces the reliability of the signal. As bit rate and

travel distance increase these issues grow. For these reasons parallel is less reliable and slower than a serial connection in most applications.

7.1.1 RS 422

RS422 uses a twisted pair for receiving and transmitting data. It is capable of full duplexing. This mean it is capable of sending and receiving signals. RS422 protocol is similar to RS232. RS422 supports a cable distance of up to 500 ft while RS232 only is recommended to go up to 50 ft. RS422 also supports Multi-Drop which allows it to have up to 32 devices connected to a port. RS232 does not support Multi-Drop. RS422 is also more noise resistant that RS232 since it uses a separate transmit and receive pair as opposed to a shared lines in RS232. RS422 is solely on the physical layer of the OSI model. It only defines how the signal is physically sent over wires. The advantage to RS422 is that it is inexpensive and allows



Here is the pinout of RS422:

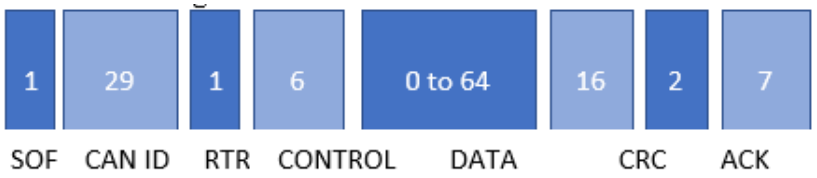
Pin Number	Name	Direction
1	TXD-	Negative Semder
2	TXD+	Positive Sender
3	RTS-	Negative Handshake Send
4	RTS+	Positive Handshake Send

5	GND	Common reference voltage
6	RXD-	Negative Receiver
7	RXD+	Positive Receiver
8	CTS	Negative Handshake Receive
9	CTS+	Positive Handshake Receive

The Advantage to using RS422 is that it is just a bitsream. A packet can be defined anyway that one wants to.

7.1.2 CAN bus

CAN signals are robust signals based on priority ID's. They are comprised of 8 parts, the number on each segment refers to the number of bits



1) SOF: This signifies the start of a frame to alert the system that a message is arriving.

2) CAN ID: This contains the priority and the address of the message.

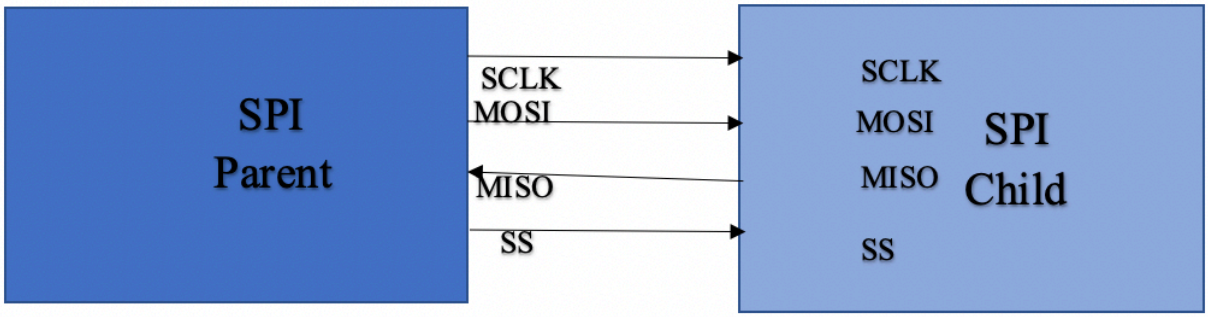
3) RTR: This allows the ECU to make a request from another ECU

4) CONTROL: This specifies the length of the DATA in the next section

5) DATA: This contains the data values in the message.

6) CRC: This checks the data integrity.

7) ACK: This indicates if the data integrity process (CRC) passes.



8) EOF: This signifies that it is the end of the message.

7.1.3 SPI

Serial Peripheral Interface or SPI is made up of four signals. SPI can use single parent device and one or multiple child device.

1) SCLK: Serial Clock
2) MOSI: Parent Out Child In
3) MISO: Parent In Child Out
4) SS: Child Select

Overall, SPI will allow more flexibility but will require more work in defining the protocol. Given that it can properly be set up to handle error correction and that the peripherals are not overly complicated SPI is probably the better choice than CAN bus for this purpose.

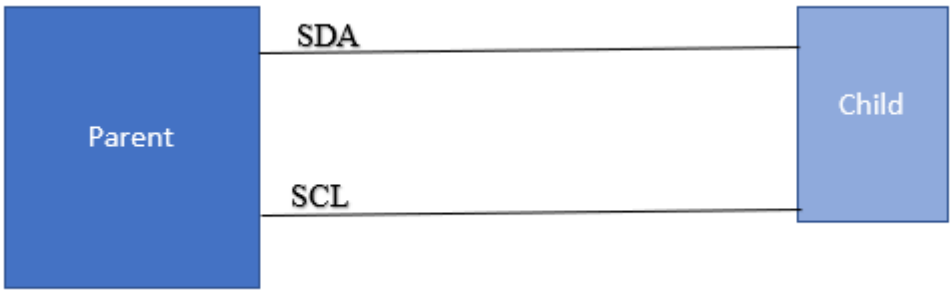
7.1.4 SpaceWire

SpaceWire is capable of full duplexing and uses a low latency point-to-point serial links. The structure at the Network layer is as follows:

The first 2 segments contain the address of the packet. The 3rd contains the protocol ID. Having this identifier allows for different protocols to act simultaneously without interfering with each other. Finally the last segment is the Frame Data which contains the actual message. There are some concerns with SpaceWire being the fragility of the cable and the fact that SpaceWire is asynchronous can be difficult to implement with an FPGA[8].

7.1.5 I2C

I2C Is a simple robust serial connection protocol. It only uses 2 pins to implement making it widely used across different fields. It allows a parent device to communicate with a child device.



1) SDA: The serial data line provides a 2 directional serial bus.

2) SCL: The serial clock provides a clock for the child device.

I2C has different speed capabilities ranging from 100KHz to 5MHz giving it a wide range of applications. There are some known issues with the reliability in space[2] for I2C because of these concerns I2C is not a good choice for our purposes.

8 Hardware and Hardware Specific Software

8.1 Overview

All software developed for the satellite will eventually have to be loaded and tested on the Onboard Computer (OBC). While the AEGIS team had originally planned on purchasing an OBC from Space Micro, the team has instead opted to build their own OBC. This will provide exceptional firsthand experience for the Command and Data Handling (C&DH) Team and allow for complete customization of what will make up the OBC. Because the OBC will be responsible for controlling the various electromechanical pieces of the satellite, it is an embedded system and many of the principles of embedded systems development, such as cross-compilation and a focus on reducing the memory footprint of our software, will apply.

8.2 Development Board

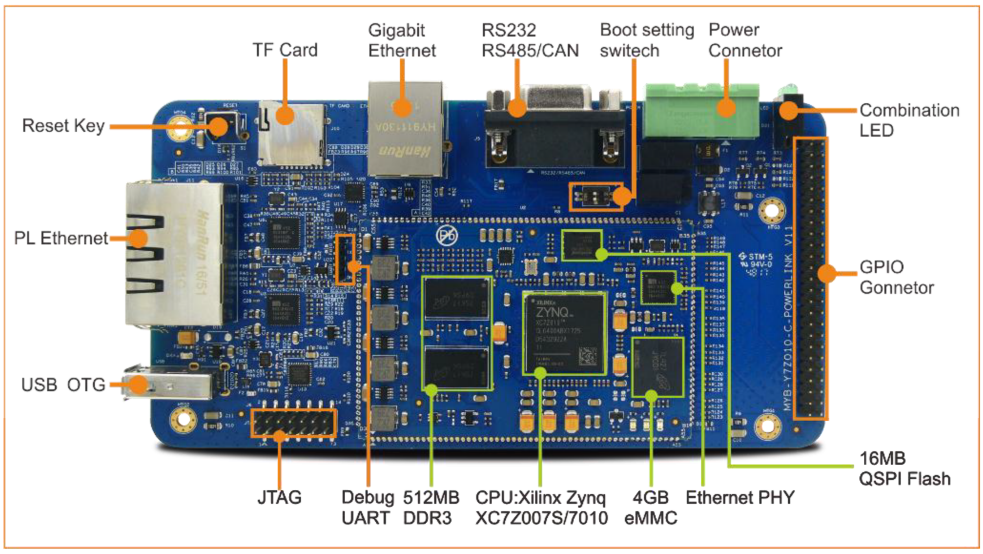
While we do not have the actual OBC, we can begin developing much of the low-level software for it since we know the target architecture. The OBC will be built around a Xilinx Zynq 7020 System on Chip Field Programmable Gate Array (SoC FPGA).

FPGAs are extremely useful in embedded systems, as they offer programmable hardware using Configurable Logic Blocks (CLBs). These CLBs can act as different arrangements of logic gates by reprogramming the associated Lookup Table (LUT) for that block. When programmed accordingly, these CLBs can work together to implement the functionality of dedicated hardware devices such as encoders, memory management units, or any other custom hardware desired by the user. Often, there is a desire to program a piece of hardware once, such as a memory management unit, and use it as a subsystem in multiple different FPGA configurations. To accomplish this, many FPGAs are built around the design of Intellectual Property (IP) Cores. An IP Core is a piece of code (either in Verilog or VHDL) that defines functionality for an FPGA and can be simply be added to an existing FPGA configuration. This allows for rapid development of FPGA configurations by combining well-tested IP Cores, and limiting the amount of one-time use code. Using an FPGA enables rapid prototyping and redevelopment and frees the C&DH team from either having to manually connect all desired hardware modules, or design and purchase an Application Specific Integrated Circuit (ASIC). Additionally, hardware functionality can be added as needed by utilizing an FPGA. Because of these advantages, FPGAs have a history of being used in Cube Satellites and are an ideal choice for AEGIS.

Another advantage of the Zynq 7020 chip is that it is an SoC FPGA. This chip combines the portability and integration offered by an SoC, in which all major components typically found on a motherboard are combined into one chip, with the flexibility of an FPGA. When working with an FPGA, any hardware that is not reprogrammable is referred to as a “hard” IP Core, while any unit of hardware that is programmed into the FPGA is a “soft” IP core. [14] Combining an SoC microprocessor and an FPGA onto a single chip yields significant benefit. When integrated, the SoC’s hard IP core provides dedicated hardware optimized for handling the majority of all computational work, while the FPGA’s resources are left free from having to configure a CPU from scratch with its resources (which while possible would consume a majority of the FPGA’s CLBs and run slower than dedicated circuitry), as well as preventing the slow down caused by using an external CPU. The Zynq 7020 chip contains a dual-core ARM Cortex-A9 SoC processor with a clock speed of up to 866MHz and a total of 85K LUTs (which are associated with a CLB) available for programming the hardware. [11, 14]

In the interest of speeding up development time, the AEGIS team has purchased a development board for testing. This allows us to generate and test boot procedures, our custom OS, and the F Prime generated software. This board is the MYIR MYD-Y7Z020 Development Board which we have colloquially referred to as the Buckler Board in keeping with the shield theme of AEGIS. The Buckler Board is made up of an MYC-Y7Z010 coreboard that is mounted on top of an MYB-Y7Z010 baseboard. The coreboard is built around the Zynq 7020 chip with external RAM and storage. The baseboard is responsible for connecting the coreboard to all the peripherals, such as Ethernet ports, USB connection, GPIO pins, (General Purpose Input Output), and the SD Card slot. [11] Shown below is a diagram of the entire board, as well as a diagram that shows if a component belongs to the baseboard or coreboard (represented by the inner box), and subdivides the coreboard

into the hard IP core components of the ARM processor, and the programmable components of the FPGA (the soft IP core).



Physical Layout of the Board

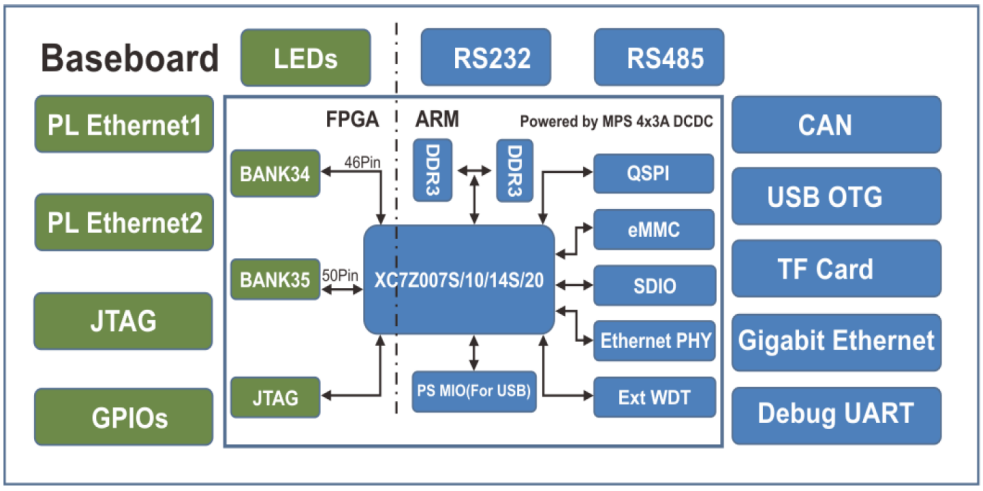


Diagram of Components of the Board

8.3 Design Tools

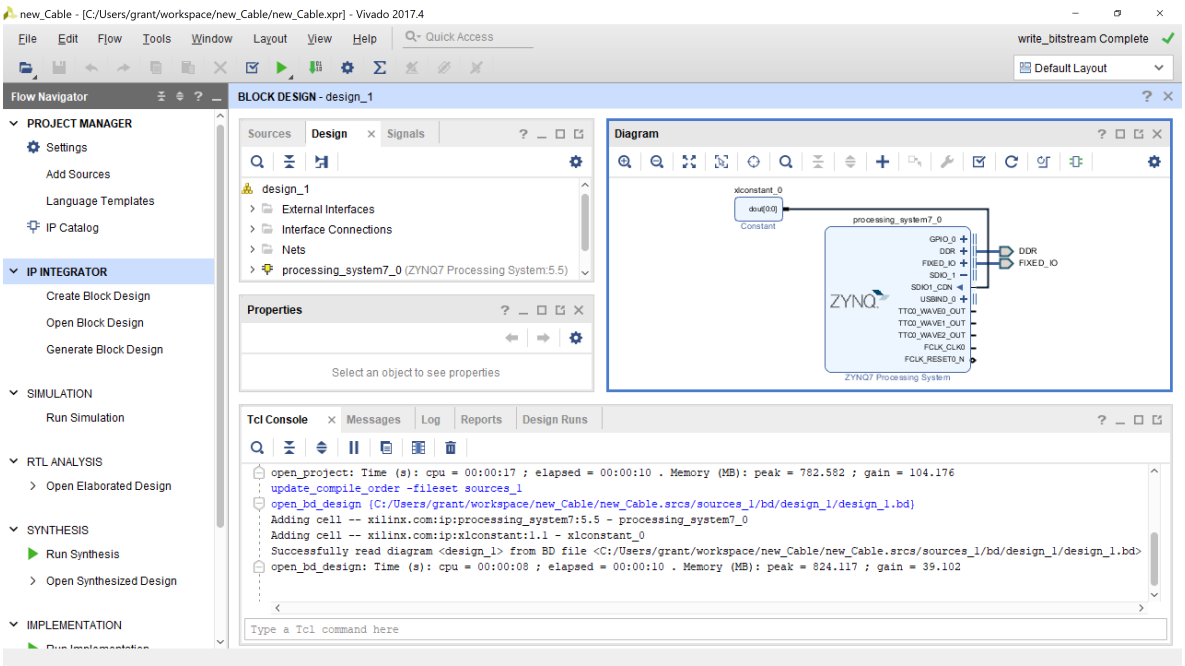
For an FPGA to be useful, it must be programmed. On the lowest level, this involves programming the individual lookup tables, however with 85k LUTs manually programming them is not practical. As such, automated tools are used to add entire hardware modules that connect to existing IP cores as specified by the user. Because this is a Xilinx chip, the recommended configuration tool for all Zynq FPGAs is the Vivado Design Suite.

The Vivado Design Suite is a set of tools useful for configuring and running programs on an FPGA. It is based around a GUI that greatly reduces or eliminates the need for writing Verilog or VHDL code to specify the hardware configuration.

The three primary pieces of the Vivado Design Suite are the Vivado Integrated Design Environment, and the Xilinx Software Development Kit (SDK; In the newest version of Vivado, this is referred to as Xilinx Vitis). Additionally, there is the Vivado High Level Synthesis (HLS) tool which can translate a C specification into an RTL design for an FPGA. This tool is optional on most Vivado installations but will likely be useful for creating complex IP cores. Much of Vivado is built around the concept of projects, which can include hardware configurations, custom IP Cores, user programs to run on the target device, etc. While Vivado can be used without projects, many of the below listed features will not work as described. As such, all future configurations of the development board should be modeled as a Vivado project. When creating a fresh project in Vivado, one should always start by opening the Vivado IDE, because this is system in which the hardware is specified and programmed.

Vivado IDE

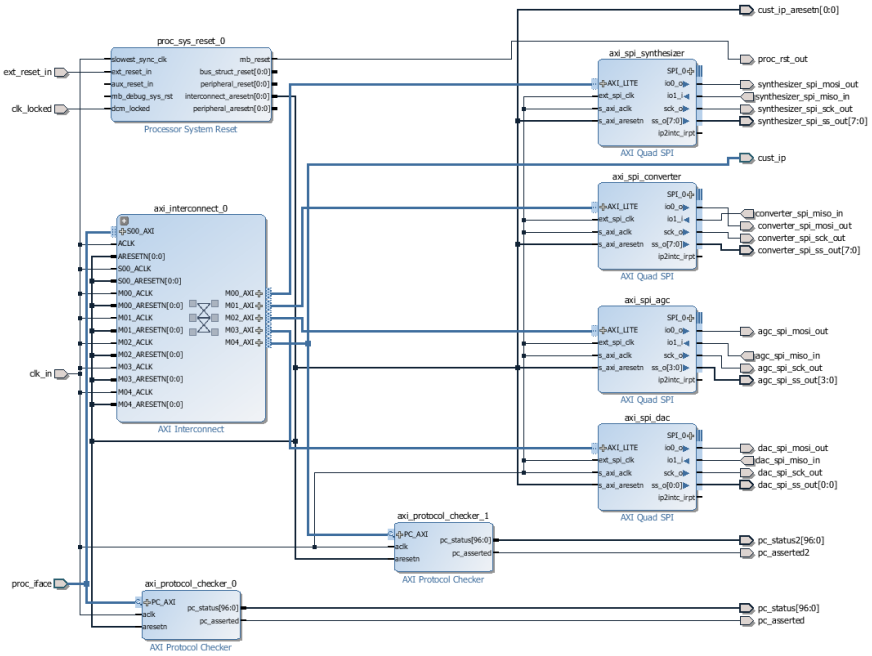
Pictured below is the home screen of the Vivado IDE. As mentioned previously, this IDE is used for specifying the desired layout of the FPGA, as well as performing analysis on the user created designs.



Vivado IDE Home Screen with an Example Project Loaded

Below are some of the capabilities of the Vivado IDE:

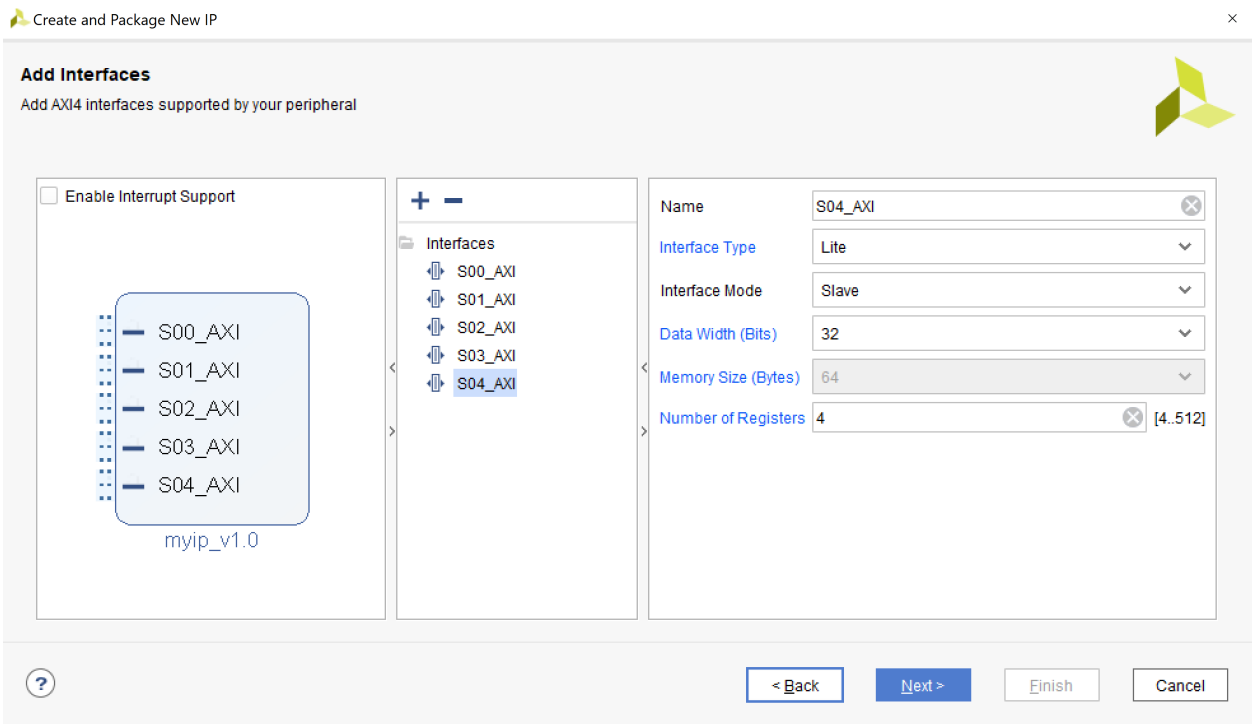
- Creating a Block Design – This is the single most important feature of the Vivado IDE, as it shows how the actual IP Cores are connected and is the basis for the generation of the bitstream file, which programs the FPGA on bootup. This is done by dragging and dropping blocks into the Block Design window, and then connecting them as desired. An example block design is shown below:



Example Block Configuration

- IP Catalog – The IP catalog is a collection of opensource IP's that can be used with Zynq and other Xilinx FPGA's, and greatly reduce development time by providing common functionalities such as Digital Signals Processing, Floating Point Units, and Reed-Solomon Encoders and Decoders. Additionally, some third part IP's from Xilinx partners can be downloaded and used by Vivado.

- Create a Custom IP – The create new IP tool is extremely useful. This tool allows you to specify the functionality of a new IP, and then add it into your local copy of the IP catalog for repeated use. The tool for making a custom IP core within the IDE is GUI-based, and as such it is not as powerful as the HLS tool, however it is still useful for quickly creating relatively simple designs that are not currently in the catalog. Below is a figure showing the IP generation wizard.

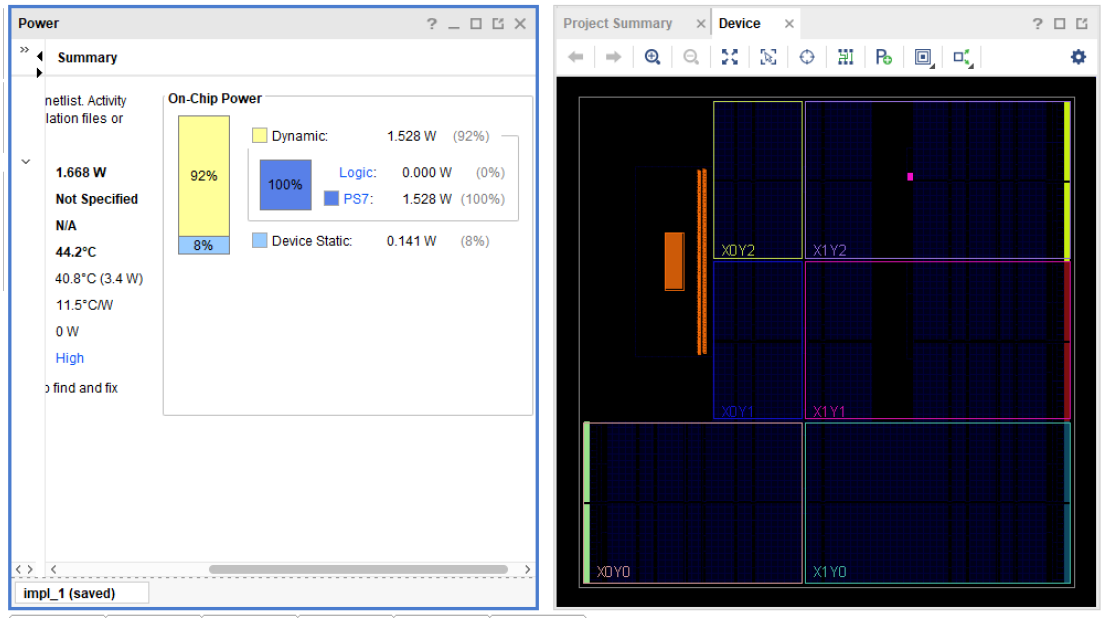


New IP Wizard with AXI4 Interface as an Example

- Validate Design – This tool allows you to ensure that your design is correctly connected in a functional way. This does not verify that the functionality is what you intended, only that it is a valid FPGA design. To increase the functionality of this tool, the user can specify constraints and even logic through the use of Design Rule Checks.

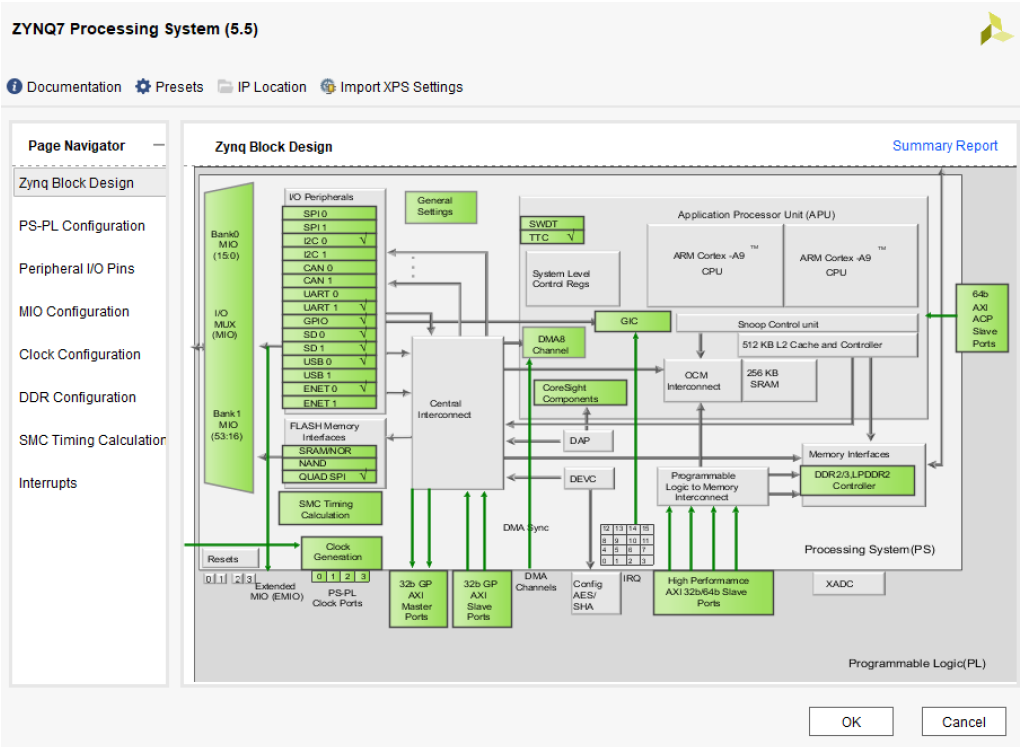
- Utilize TCL Scripts – All commands in Vivado are built around TCL (pronounced 'tickle') and as such, Vivado contains a TCL console window that is shown in the IDE home screen figure above. Beyond singular commands, a user has the ability to write and execute TCL scripts for a project. These can be used for automatically connecting given IPs within a block design, applying a specific configuration for a project, or any other string of Vivado commands that need to be performed repetitively. An example use of this would be to have a base configuration for the OBC saved as a TCL script, and then adding and testing new components or configurations that start from a common baseline.

- Analyze Physical Properties – If you are using a Xilinx chip, the Vivado IDE allows you to analyze many physical properties of the system, such as estimated power consumption, the flow of clock signals throughout the device, and I/O Schematics. Additionally, properties such as propagation delay can be estimated and viewed. All of these will be useful in testing potential FPGA configurations before they are flashed onto the OBC. An example figure showing power consumption statistics is shown below.



Power Consumption Statistics

- View schematics of a Xilinx Chip – Vivado also allows you to view a layout of Zynq and other Xilinx chips, for either reconfiguration or analysis. You can manage multiple properties of the chip, such as peripherals, clock signals, and memory. A figure of the ZYNQ7 processing system is shown below, with configuration options pictured on the side.



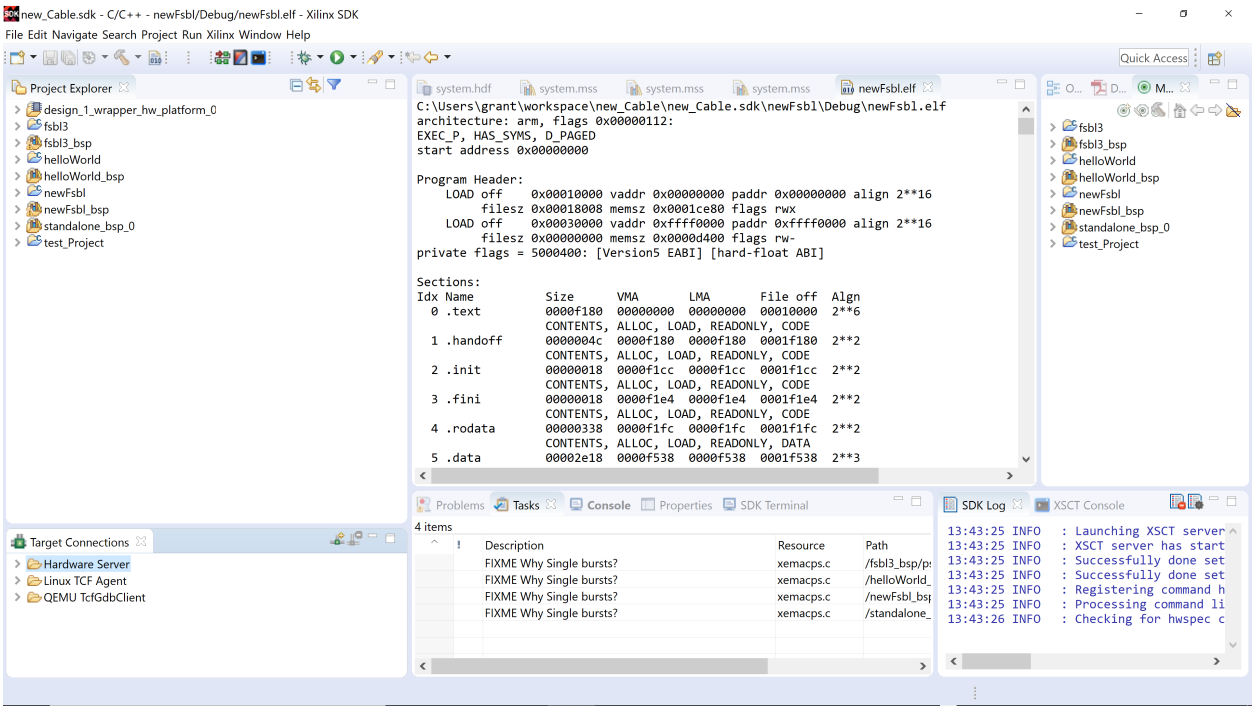
Example of Zynq Schematic and Configuration Options

- Generate Bitstream – This tool is simply a command that generates a bitstream file based on a given design. This file is what the FPGA looks for on bootup to program its CLBs.

These functionalities of Vivado are all useful as they ease the burden of generating and testing designs, as well as enable output products. Currently, only the Create Block Design and Generate Bitstream features have been used in configuring the development board.

Xilinx SDK (Later Vitis)

After the hardware specification has been developed, you can launch the SDK to begin the logical verification of these designs, as well as begin to develop programs that take advantage of the newly configured hardware. A picture of the SDK home screen is shown.



Xilinx SDK Home Screen

Below is a summary of the SDK's capabilities:

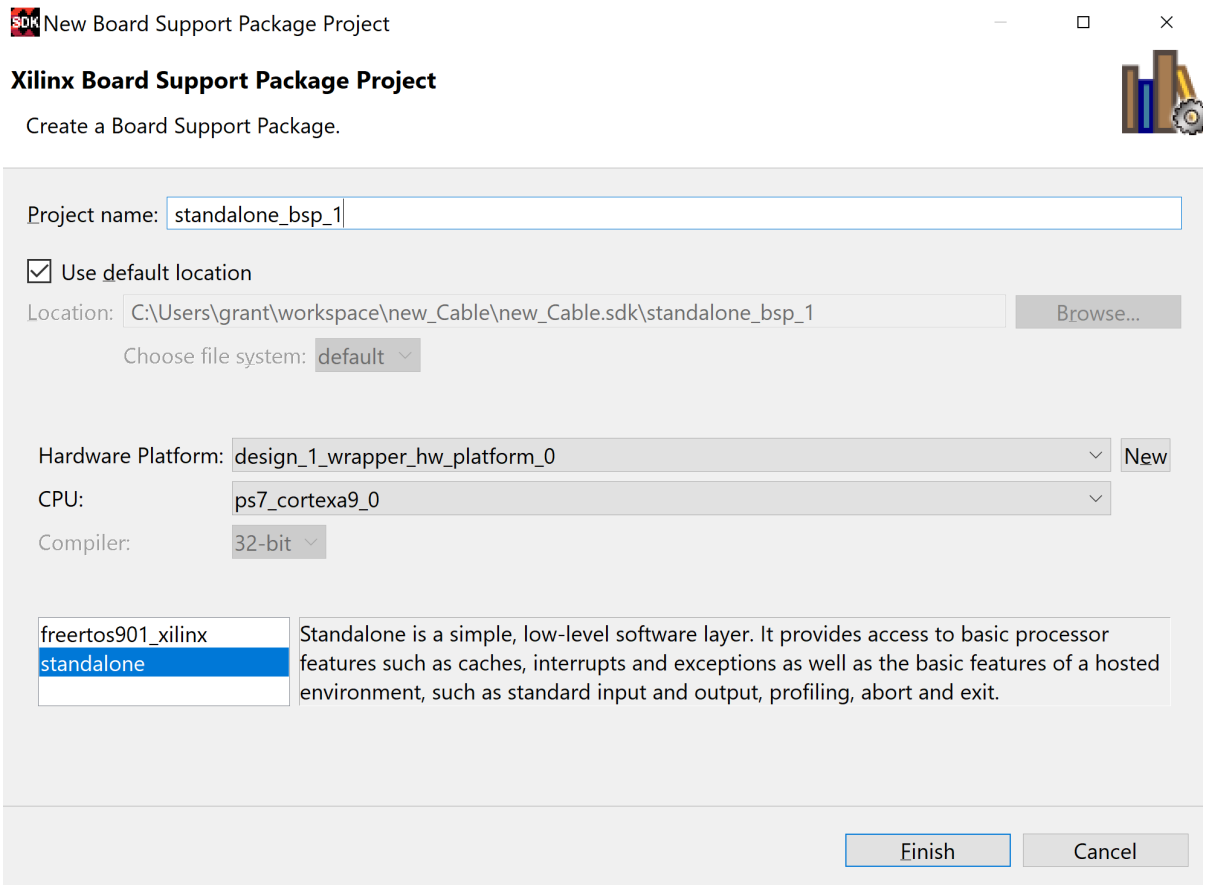
- Develop C Programs – The SDK provides an environment in which C, C++, and in Vitis versions Python can be developed for the target hardware. This is helpful because it removes the need for a separate environment for developing applications, as well as the fact that this environment is aware of the hardware specifications since you can import a Vivado IDE project into it.

- Debugging – Similar to other SDK's the Xilinx SDK comes with a debugger that can set breakpoints, step into and over functions, as well as visualize variables and data flows throughout a program.

- Run programs on the board without an OS – By connecting a Xilinx board to the host computer via JTAG, one can run programs on the board from the SDK without the need for an

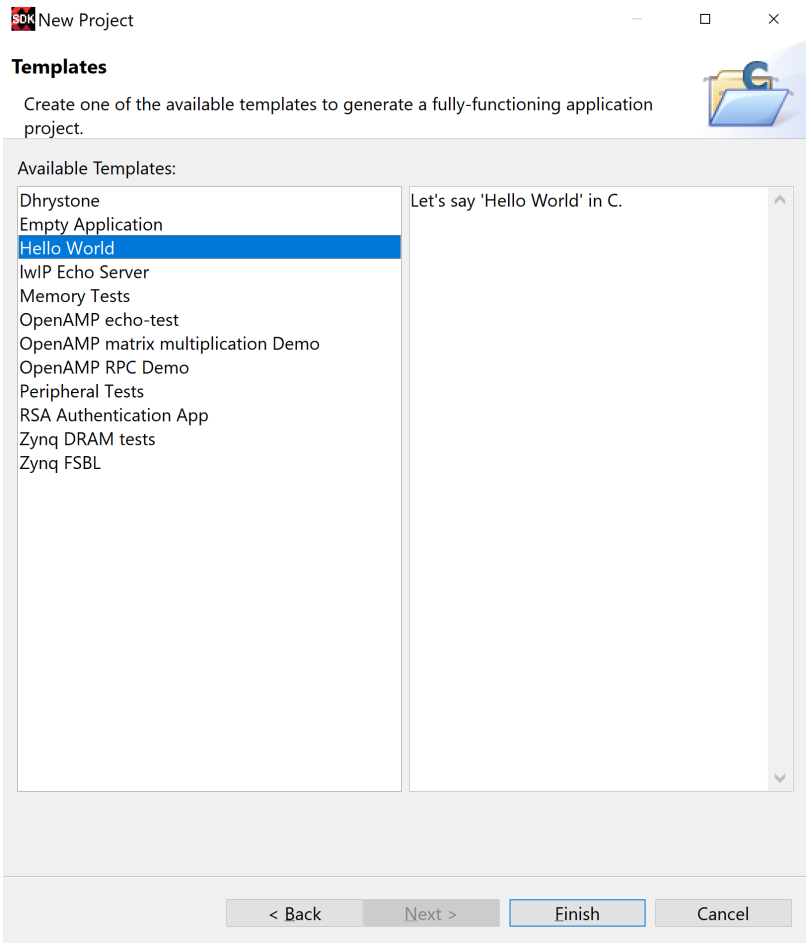
operating system on the board. This dependent boot configuration is useful for the rapid testing and reconfiguring of programs.

- Create Board Support Packages – The Xilinx SDK contains the ability to create your own Board Support Package (BSP), to include a first stage bootloader. A BSP is a collection of libraries and drivers that will form the lowest layer of your application software stack. Any user program that runs on the board must link to or run on top of the BSP software platform using its provided API. This functionality is mirrored by U-Boot however and U-Boot is typically easier to use for a standalone boot configuration, however this should be used for the dependent configuration. Below is a figure showing the BSP wizard.



BSP Wizard

- Run Benchmark Programs on Hardware – The Xilinx SDK offers a variety of standard applications such as hardware benchmarks, ethernet testing, a hello world program, and DRAM tests that can be target towards the custom hardware. These are extremely useful as they are well tested pieces of code that will provide standardized output, allowing a user to isolate hardware problems if output is different than expected. A list of the standard applications included with the free version of Vivado is shown below in the application manager.



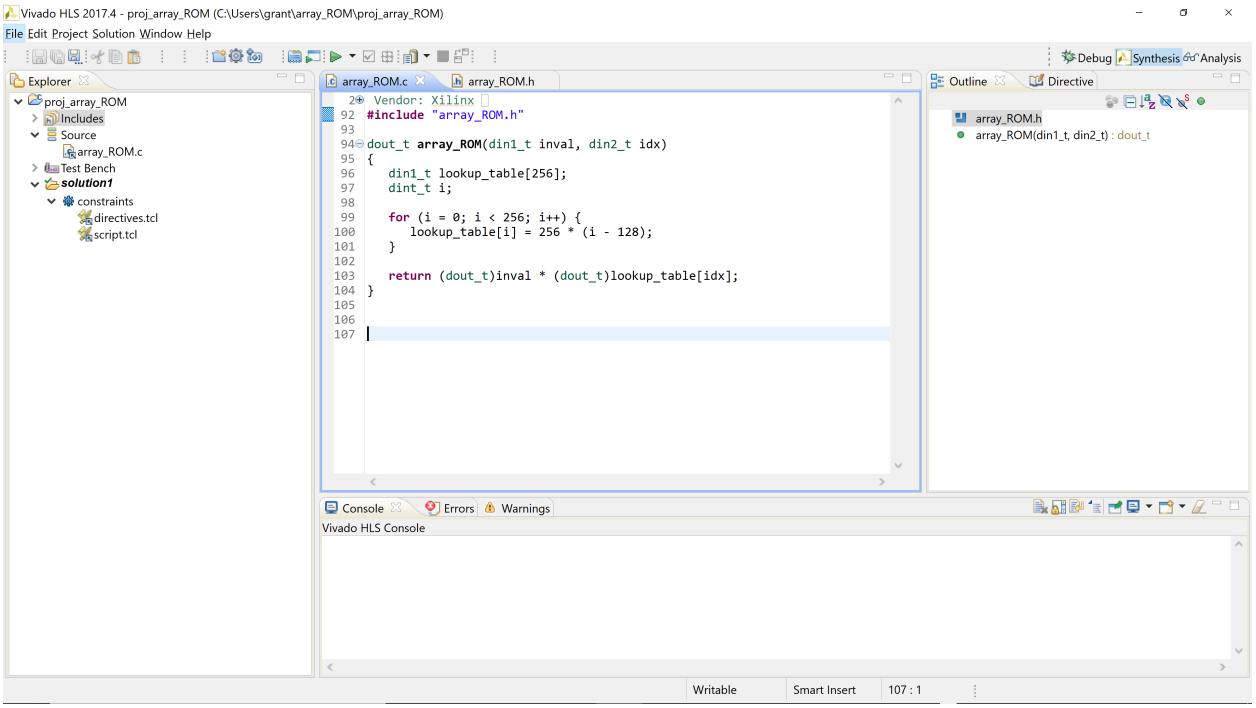
Wizard for Creating a Standard Project.

- Generate ELF Files – The SDK also has the ability to generate executable binaries in the form of ELF files for loading onto the board. These are cross compiled for the target architecture and can be run in a standalone configuration as well as dependent.

All these tools will be relevant to the development of hardware-specific software for the OBC, such as device and kernel level drivers. It is my recommendation that these kinds of software be developed and tested within the Xilinx SDK. Then by dependent booting of the development board and finally standalone boot testing. As of this writing, all SDK tools mentioned above have been used with the development board with the exception of the debugger as only provided C programs have been used and modified.

Vivado HLS

Finally, the last system in the Vivado Design Suite is the Vivado High Level Synthesis Tool. This is another tool for creating custom IPs, however it is more powerful than the tool contained within the Vivado IDE. This tool has the ability to translate algorithmic specifications in C/C++ into RTL (Register Transfer Level) designs for IP cores. This enables the development of complex cores in which the manual specifying of each hardware interaction is not feasible. Below is an example of a ROM array specified in C.

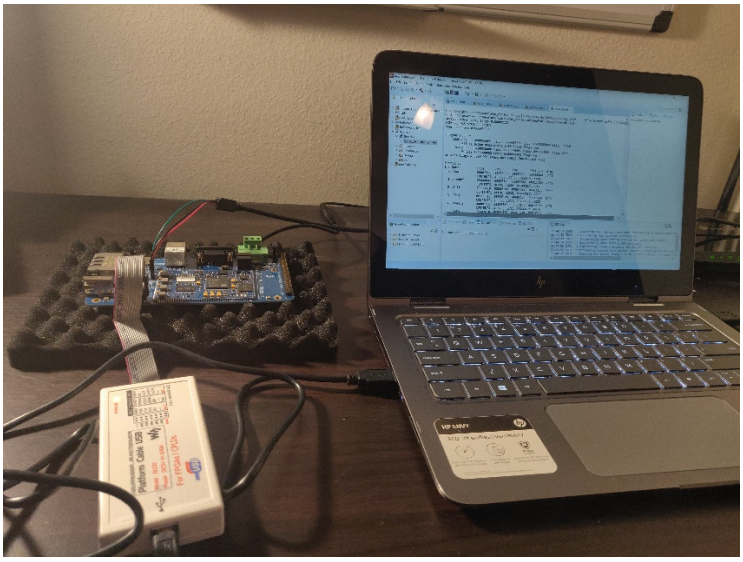


Example C Specification of a ROM Array

At this time, Vivado HLS has not been used for configuring the development board.

The Vivado Design Suite is incredibly powerful and complex. After the learning curve is overcome, it can be used to greatly reduce the development time from having a blank FPGA to a working custom system.

Once the desired configuration has been made within Vivado, you can then export the created bitstream file. This bitstream file is one of the first files loaded into the FPGA at bootup, as it is responsible for programming the various LUTs. For initial testing however, using the SDK is recommended as it is quicker to redesign and reprogram the board. Within the SDK, you can either generate and compile user programs for a standalone boot process or connect the Buckler Board to the computer via a JTAG cable and boot it in a dependent configuration. The dependent configuration allows for the rapid testing and debugging of your programs, while the standalone configuration more closely simulates the actual usage of the OBC, in which all files will be loaded onto the OBC prior to launch, and are expected to run independently. Shown below is a picture of the board and the Xilinx SDK in the dependent configuration. In the picture, the white box is a USB to JTAG converter, and it is over this connection that the board receives the bitstream and other boot files.



Board Being Booted with the Vivado SDK

Finally, for providing user program input and output to the board, you should use the software PuTTY to establish a serial connection via a UART to USB cable. The UART pins on the board will be the primary interface to user programs on the board until a GUI for F Prime is necessary. Once configured appropriately, PuTTY will provide a simple terminal from which the board can be used.

7.4 Implementing Hardware Redundancy

The massive amount of radiation in space can have disastrous effects on reliable computation. It is a common occurrence that a radiation strike can flip a bit, or even cause a CPU to be incapable of producing correct output. As such, redundancy is needed to ensure correct computations, as well as detect when erroneous output is present. One strategy for doing this is using Triple Modular Redundancy (TMR) in which the same process is performed by three different CPU's and then their output is majority voted on, and that is the final output for that process. This idea of a 2/3 majority enables one CPU to be faulty and still allow the error to be corrected. The Vivado IDE provides ample support for implementing TMR into the FPGA design, and all Xilinx created TMR IP Cores are included with the base version of Vivado. Below is a summary of the provided IP Cores relevant to TMR.

- TMR Voter – The primary TMR Core provided by Vivado is the TMR voter. This voter is used to output the value of the three processors as a single majority output. This output will then be fed into other systems as standard computation output. The voter can be configured for multiple different bus interfaces, such as UART, AXI4, and GPIO.

- TMR Comparator – The TMR Comparator differed from the voter by detecting a mismatch rather than performing the majority vote. This can be used to detect when a CPU has gone bad and allow the Flight Software to respond appropriately by sending a shutdown signal to the CPU.

- TMR Soft Error Mitigation Interface – The Soft Error Mitigation core allows Zynq processors to detect, correct, and classify soft errors in configuration memory. Soft errors are

unintended changes to the state of memory bits caused by radiation. This is useful for detecting a bad input to the CPU from memory, as opposed to an entirely compromised CPU.

- TMR Inject – The TMR Inject allows users to inject faults into various points of the computation flow (I.E. introduce soft errors or simulate a bad CPU). This is extremely useful for testing the effectiveness and correct configuration of the TMR system.

- TMR Manager – The Manager core is what provides overall control of the TMR system. It is responsible for interfacing with the Voter and Comparator.

When implementing TMR in Vivado, Xilinx highly recommends using the block automation feature to ensure proper connection and configuration of a TMR system.

7.5 Bootloader Information

To bring the Buckler Board online independently, a Bootloader is required. A Bootloader works as follows: once power has been established the first thing the CPU does is read a section of Read-Only Memory (ROM) that points to the memory address of the first stage bootloader. The address to which this section of ROM points is changed on the Buckler Board by setting the boot switches appropriately (this determines if the board looks for the appropriate boot files over the JTAG interface, from its internal memory, or an inserted SD card). Once the first stage bootloader has been loaded into memory, the first stage bootloader loads the second stage bootloader. This second-stage bootloader is responsible for setting up essential mechanisms such as the root filesystem and loading the kernel into memory. For the Buckler Board, we will use the Universal Bootloader, better known as Das U-Boot or simply U-Boot. U-Boot is a widely used tool for bringing up many kinds of hardware and can be configured for ARM. U-Boot is completely customizable for the end user's needs, and as such, it can be tailored for what applies to the Buckler Board, or the OBC in the future (the majority of work involved in porting the OS will likely be in reconfiguring U-Boot). Furthermore, U-Boot leaves a significantly smaller footprint in memory than alternatives such as UEFI; this is essential when memory space is at a premium, as it is on the OBC. A successful U-Boot configuration produces a "Boot.bin" file which is the precompiled binary responsible for booting the device and a "uEnv.txt" file which contains commands for U-Boot to execute. [1] U-Boot provides a simple shell that is loaded onto the board to provide debugging information to the user and then closes once the OS kernel is successfully loaded.

7.6 Overview of bringing Linux Online for the Development Board

As previously mentioned, the OBC is an Embedded System and will be running a Linux kernel. Embedded Linux systems are widely used as Linux is a free open-source distribution with a larger community and relatively easy to bring online. For an embedded system to work it needs a minimum of four major components: a Toolchain, a Bootloader, a Kernel (which in our case will be our customized Linux), and a Root File System. [12]

The toolchain is used to build the other components into binary executables that will be loaded onto the board. In nearly all embedded systems, the toolchain is kept on a more powerful development machine and builds the Bootloader, Kernel, Root File System, and any user programs on this development machine for later transfer to the target system. This more powerful machine is referred

to as the host machine, and the intended embedded system is referred to as the target system. Often, the host architecture (i.e. the actual type of processor) is not the same as the target architecture. As such, one must cross-compile all files for a known target architecture. This means using a toolchain that is aware of both the host and the target architectures and builds it for the target architecture. [12] The advantage of cross-compilation is that it enables a user to use whatever machine they have available for compilation, and then deploy it to the target system. One drawback is that the generated files cannot be tested on the host machine without a hardware emulator, though this is a minor inconvenience in the case of the Buckler Board since deployment to the target architecture is a matter of re-flashing an SD card.

The Bootloader, as discussed earlier, will be a U-Boot configuration that has been tailored to the Buckler Board or the OBC. Concerning the OS kernel, the goal for low-level development is the ability to easily swap kernels, and this is achieved by simply compiling an updated kernel and testing its performance on the board. Doing this will enable those working on the OS to rapidly test any changes in their designs on the actual Buckler Board. Furthermore, the ability to change kernels with relatively low overhead will be useful for rolling back to an earlier OS version for debugging purposes. Current testing has shown the compilation process for the kernel to take around 10 minutes, and it will produce a file name “uImage”.

Finally, the Root Filesystem is essential to many basic functionalities in Linux, and all other user filesystems are mounted as children of this filesystem. When the board is running, a Ramdisk is used for the Root File System and must also be built. The Ramdisk is used to package the root filesystem so that the root system is always accessible by the board in memory without having to refer to the SD card. This provides a significant boost in speed when booting from external memory sources such as an SD card.

Once these files have been successfully compiled, they can be loaded onto the board. For the Buckler Board, seven files are necessary on the SD card. [1]

These files are as follows:

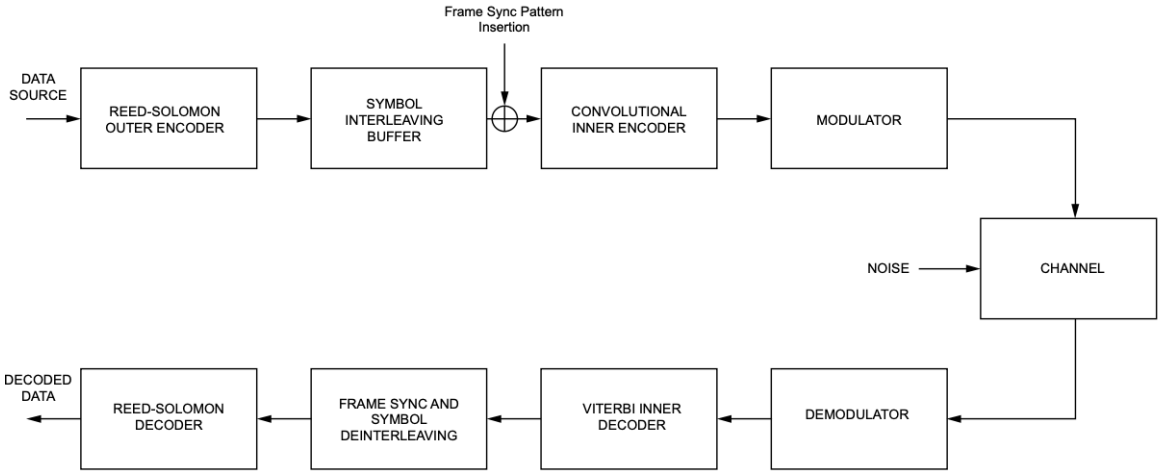
- Boot.bin – This is the output of a successfully compiled U-Boot configuration.
- uEnv.txt – This is a configuration text file that gives commands to U-Boot once power is established.
- Bitstream – This is the bitstream file generated by the Vivado design suite after configuring the FPGA and is responsible for programming the LUTs.
- Device Tree – This file is responsible for telling the Linux Kernel the available hardware resources and the layout of these resources. This file can also be generated using the Xilinx SDK. [3]
- Ramdisk Image – This is a tar file containing the Ramdisk.
- Root Filesystem – This is another tar file that contains everything that will build the Root Filesystem
- uImage – This is the result of successfully compiling the Linux kernel and is what U-Boot will look to load into memory.

Once a specific Linux kernel has been successfully loaded onto the board, the next step in the development process will is loading user programs onto the Buckler Board (either pre-compiled or loaded on the board or compiled locally if resources allow) and testing the specific kernel's efficiency or the F Prime program's correctness.

Reed-Solomon and Convolutional Concatenated Codes

8.4 Overview

Almost all spacecraft employ forward error correcting (FEC) codes to make more efficient use of the communications channel. Forward error correcting codes add additional symbols to the transmitted data stream that the decoder can use to improve its estimate of the encoded bit stream. The Reed-Solomon (RS) code provides excellent performance with minimum bandwidth expansion in a high signal-to-noise environment. It is most often used as an outer code in combination with a Convolutional inner code [1].



Channel Coding Block Diagram

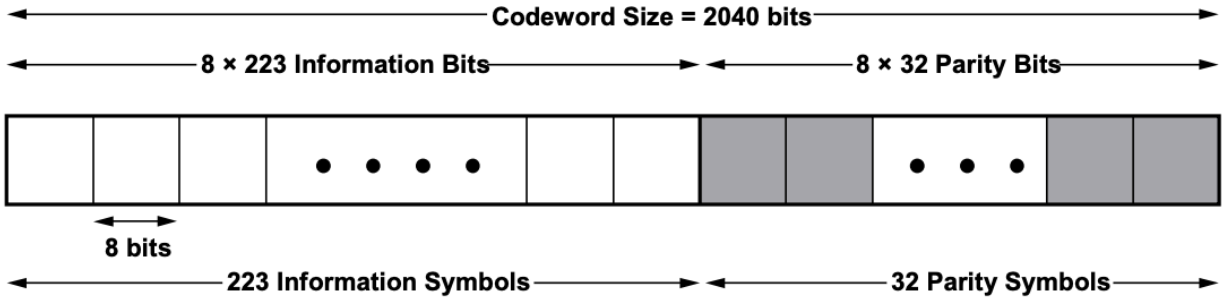
The Consultative Committee for Space Data Systems (CCSDS) recommends the concatenated coding system shown above which includes a (255, 223) Reed-Solomon code and a (7, 1/2) Convolutional code.

9 Reed-Solomon Codes

9.1.1 Introduction

RS codes are a particularly useful class of linear block codes denoted by (*n, k)* where *n* represents the block length and *k* represents the number of information symbols.

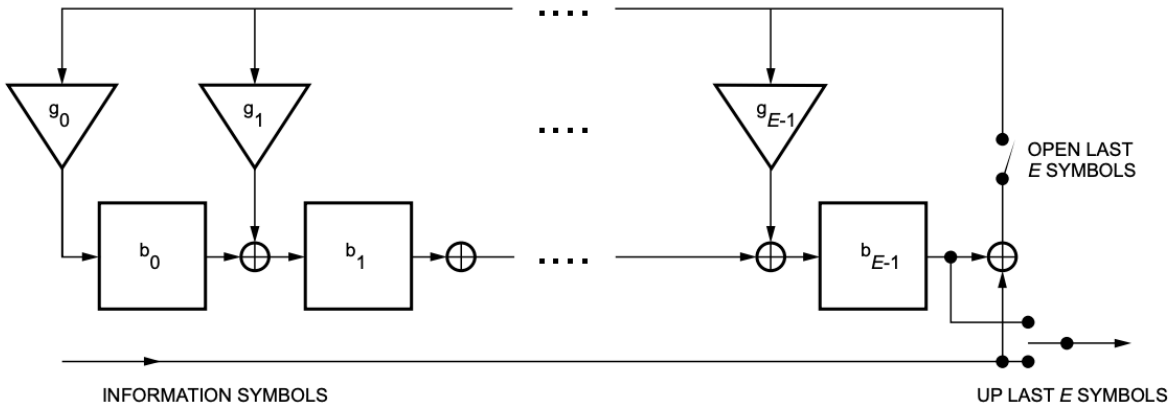
The RS code recommended by the CCSDS has codeword size $n = 255$ symbols, symbol size of 8 bits, and can correct $E = 16$ errors. The recommended RS code is non-binary and each member of the coding alphabet is one of 256 elements of a finite field rather than 'zero' or 'one'. A string of eight bits is used to represent each element in the field so that the output of the encoder still looks like binary data. An RS symbol size of eight bits was chosen primarily because telemetry Transfer Frames are octet-based. The recommended RS code was shown to have the best performance when concatenated with a (7, 1/2) Convolutional inner code. Since two check symbols are required for each symbol error to be corrected, there are a total of 32 check symbols and 223 information symbols per codeword [1, 2].



Structure of a (255, 223) Reed-Solomon Symbol

9.1.2 Encoder

RS codes are block codes. This means that a fixed block of input data is processed into a fixed block of output data. In the case of the (255, k) code, $k = 255–2E$ RS input symbols (each eight bits long) are encoded into 255 output symbols. The RS code in the Recommended Standard is systematic. This means that a portion of the codeword contains the input data in unaltered form. If desired, a 'quick look' at the information bits is possible.

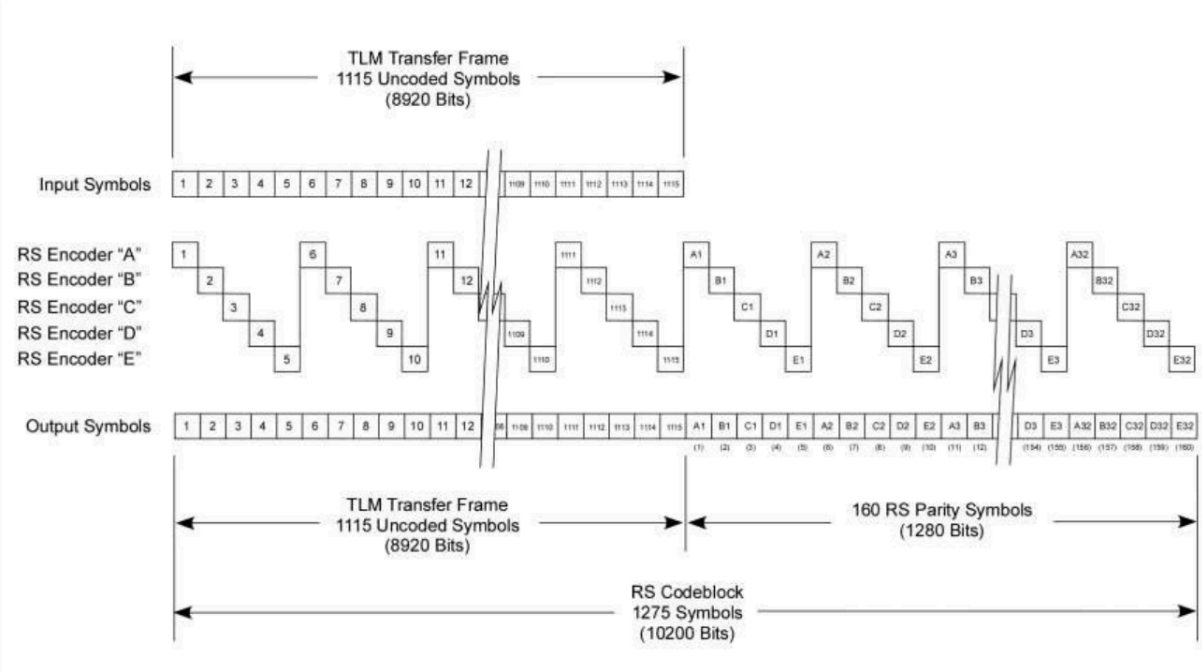


Reed-Solomon Encoder Logic Diagram

The RS code is defined by the code generator polynomial $F(x) = x^8 + x^7 + x^2 + x + 1$ over GF(2^8) and a field generator over GF(2) that contains '0' or '1'. Verifying a codeword is a matter of checking for divisibility by *F(x)*[7].

9.1.3 Interleaving

When concatenated coding is used, interleaving of the RS code symbols improves code performance. Without interleaving, burst error events would tend to occur within one RS codeword, and one codeword would have to correct all of these errors. Therefore, over a period of time there would be a tendency for some codewords to have 'too many' errors to correct (i.e., greater than E). The purpose of interleaving and de-interleaving is to make the RS symbol errors, at the input of the RS decoder, independent of each other and to distribute the RS symbol errors over several codewords. The entire package of *I* RS codewords constitutes one RS codeblock and interleaving of *I* RS codewords produces a codeblock of length *I* * [RS codeword length] [13].



Codeblock for Interleave Depth *I* = 5

Successive information symbols are written into a matrix column by column, where there is no need to store the entire array of 1,115 information symbols because each column of *I* newly written symbols can be immediately read out as the next *I* symbols of the RS codeword, as soon as the encoder computes the linear contribution of each of these *I* symbols to its corresponding set of RS parity symbols [9].

9.1.4 Decoding Interleaved Reed-Solomon Symbols

Encoding or decoding of a concatenated code is a matter of encoding or decoding the two codes in sequence. Unlike the 'soft' channel symbol values that are input to a Viterbi decoder for Convolutional codes, the symbols input to the RS decoder are 'hard', which means that the RS decoder operates on symbols drawn from exactly the same alphabet as that used in producing the encoded symbols. This generation of hard symbol inputs to the RS decoder happens automatically when these symbols are generated by a Viterbi decoder for an inner Convolutional code. In this case, the Viterbi decoder generates hard bit-by-bit decisions, and eight consecutive bits from the Viterbi decoder are grouped to form one symbol from the RS alphabet. The basic idea behind all RS decoding algorithms was developed by Berlekamp as described in reference [9] but there are dozens of variants of his basic algorithm in current use. A very detailed discussion on RS decoding algorithms can be found in reference [10].

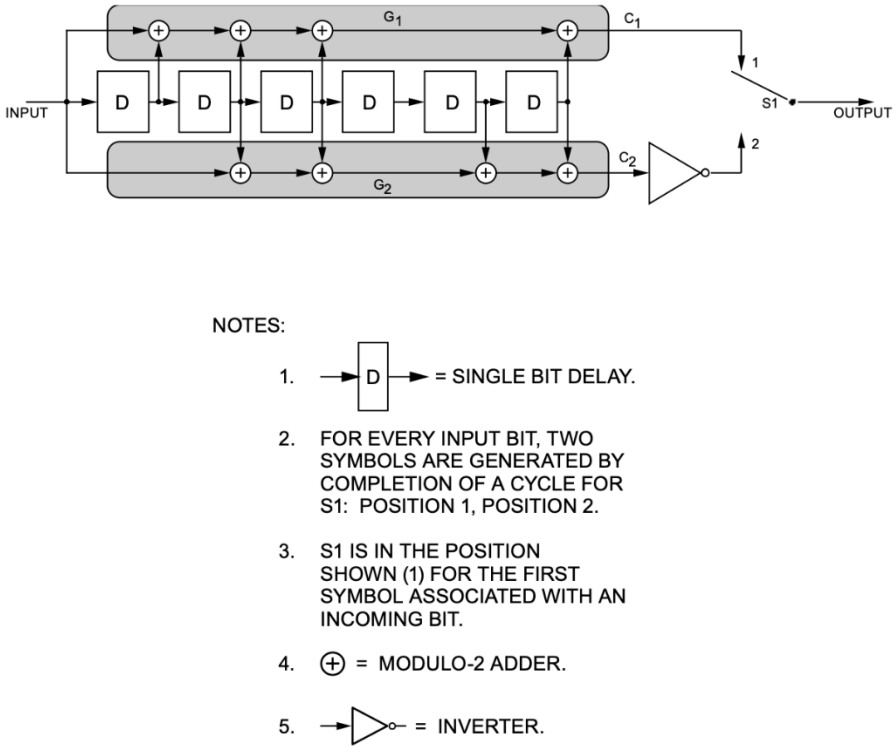
9.2 Convolutional Codes

9.2.1 Introduction

Convolutional codes are specified by their constraint length (K) and rate (r). Constraint length is the number of sequential input bits required to define the output symbols at any point in time. Rate is the number of data bits with respect to the number of coded symbols expressed as a fraction. In general, the performance of a Convolutional code increases directly with k and inversely with r, but codes must be selected carefully because the channel bandwidth also varies inversely with r and decoder complexity increases exponentially with k [3].

9.2.2 Encoder

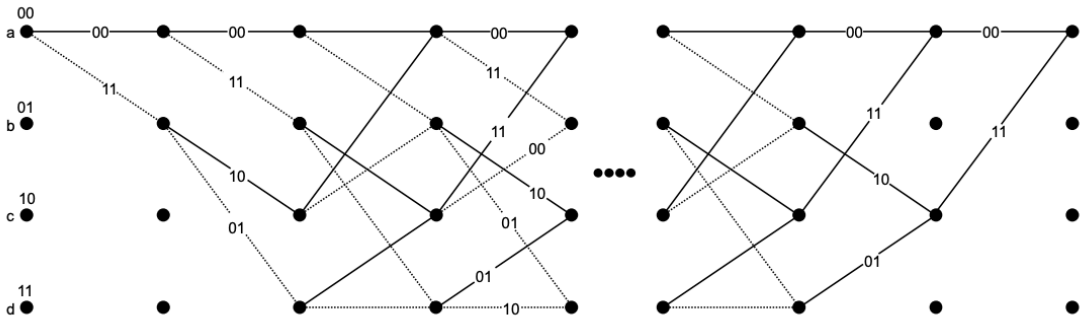
A rate $r = 1/n$ Convolutional encoder is a linear finite-state machine with one binary input, n outputs and an m-stage shift register, where m is the memory of the encoder. Such a finite state encoder has 2^m possible states. In comparison to block codes, Convolutional codes encode the input data bits continuously rather than in blocks. The encoder from the CCSDS Recommended Standard is shown in the figure below. It consists of a shift register and some exclusive OR gates that implement the two parity checks. The two checks are then multiplexed into one line. It is customary to invert one of the parity checks in the encoder to ensure that there are sufficient transitions in the channel stream for the symbol synchronizer to work in the case of a steady state (all 'zeros' or all 'ones') input [1, 2].



CCSDS Recommended Standard (7, 1/2) Convolutional Encoder

9.2.3 Viterbi Decoding

Soft maximum likelihood decoding of Convolutional codes can be accomplished by using the Viterbi algorithm. For this, it is helpful to understand the trellis representation of the Convolutional encoder. For a constraint length K, code rate $r = 1/n$, (K, r) Convolutional encoder, the state is defined by the $(K–1) = m$ most recent bits in the shift register. The figure below shows an encoder for a (3, 1/2) Convolutional code. This is just an illustrative example.



Trellis Representation of a (3, 1/2) Convolutional Code

The explanation that follows is from the CCSDS Green Book in reference [6]. The diagram starts in the all-'zero' state, node *a*, and makes transitions corresponding to the next data bit. These transitions are denoted by a solid line (branch) for a '0' and by a dotted line for a '1'. Node *a* proceeds to node *a* or *b* with outputs bits '00' or '11'. A branch weight is the number of '1's in the *n* code symbols in the branch.

The Viterbi algorithm implements maximum-likelihood decoding. An exhaustive search maximum-likelihood decoder would calculate the likelihood of the received data for code symbol sequences on all paths through the trellis. However, this is impractical since the number of paths for an L bit information sequence is 2^L. It is possible to reduce the effort by taking advantage of the special structure of the code trellis since the trellis assumes a fixed periodic structure after depth K is reached.

Paths are said to have diverged at some state if their information bits disagree at depths j and $(j + 1)$. Fortunately, paths can remerge after $(K – 1)$ consecutive identical information bits. Finding the shortest route through the trellis is identical to solving the maximum-likelihood decoding problem. Consider a rate $1/n$ convolutional code, with $u_0 \ldots u_{t-1} u_t u_{t-1} \ldots$ denoting the information bits input to the encoder. At time t, the encoder state is defined as $s_t = u_t \ldots u_{t-K+1}$.

Given a sequence of observations $y_0, y_1, \ldots, y_L$, where $y_i = (y_{i1} \ldots . y_{in})$, every path may be assigned a 'length' proportional to metric $-\log p(y|s)$, where $p(y|s)$ is the likelihood function and $s = (s_0, \ldots, s_L)$ is the state sequence associated with that path.

The Viterbi algorithm solves the problem of finding the path whose length $-\log p(y|s)$ is minimum. It should be noted that to every possible state sequence s there corresponds a unique path through the trellis, and vice versa. If the channel is memoryless, then

$$-\log p(y \mid s) = \sum_{t=1}^{L} \lambda\left(s_t, s_{t-1}\right),$$

where

$$\lambda(s_t, s_{t-1}) = -\log p(y_t|s_t, s_{t-1}) = -\log p(y_t|s_t)$$

is the branch 'length' or metric. $T_t(s_t, s_{t-1})$ denotes the transition from state s_{t-1} to s_t associated with branch symbols $x_t = (x_{t1} \ldots x_{tn})$, which correspond to the information sequence $u_t \ldots u_{t-K}$.

Therefore, the state transition can be defined as $T_t(s_t, s_{t-1}) = u_t \ldots u_{t-K}$. $\mathbf{s}(s_t)$ denotes a segment $(s_0, s_1, \ldots, s_t)$ consisting of the states up to time t of the state sequence $\mathbf{s}$. In the trellis, $\mathbf{s}(s_t)$ corresponds to a path segment starting at the state s_0 and terminating at state s_t. For any particular time t and state s_t, there will in general be several such path segments, each with some length

$$\lambda\left(\mathbf{s}\left(s_t\right)\right) = \sum_{i=1}^{t} \lambda\left(s_i, s_{i-1}\right).$$

The shortest such path segment is called the *survivor*, corresponding to the state s_t, and is denoted $\hat{\mathbf{s}}(\mathbf{s_t})$. For any time t > 0, there are 2^m survivors in all, one for each s_t.

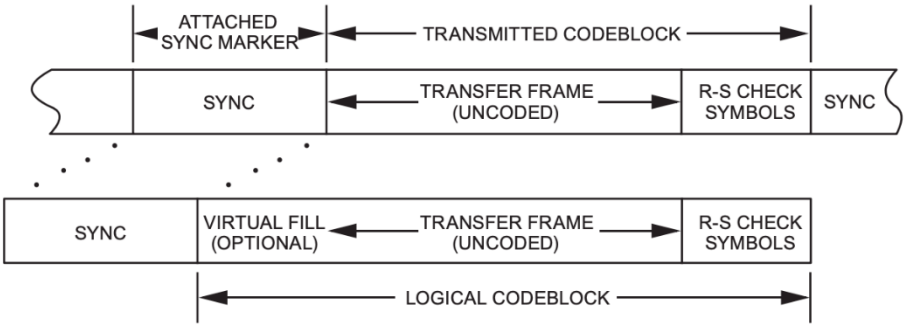
Thus at any time t, one need remember only the 2^m survivors $\hat{\mathbf{s}}(\mathbf{s_t})$ and their lengths $\Gamma(s_t) = \lambda(\mathbf{s}(s_t))$. To get to time $t+1$, one need only extend all time t survivors by one time unit, compute the lengths of the extended path segments, and, for each state s_{t+1}, select the shortest extended path segment terminating in s_{t+1} as the corresponding time $t+1$ survivor. Recursion proceeds indefinitely without the number of survivors ever exceeding 2^m [6].

The advantage here is that the number of decoder operations performed in decoding L bits is only $L2^m$, which is linear in L.

9.3 Frame Synchronization

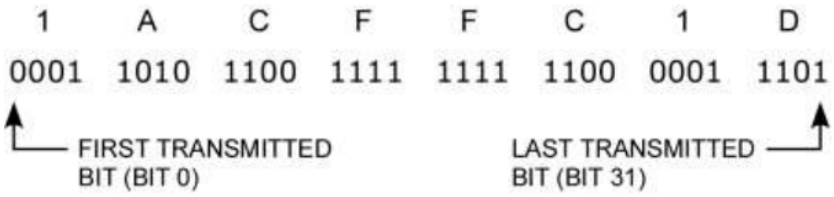
9.3.1 Attached Sync Markers

Frame synchronization must be established before processing any Reed-Solomon codes. Synchronization is accomplished by preceding each codeblock or transfer frame with a fixed-length Attached Synchronization Marker (ASM).



ASM Placement Relative to the Codeblock

This known bit pattern can be recognized to determine the start of the codeblocks or transfer frames. The CCSDS has adopted the 32-bit ASM shown below for synchronization in the bit domain.



CCSDS 32-bit Attached Synchronization Marker

10 Glossary

Block Encoding: A one-to-one transformation of sequences of length *k* of elements of a source alphabet to sequences of length *n* of elements of a code alphabet, $n > k$.

BSP: A BSP is a board support package.

Channel Symbol: The unit of output of the innermost encoder that is a serial representation of bits, or binary digits, which have been encoded to protect against transmission induced errors.

Codeblock: The aggregation of *I* codewords, where *I* is the interleaving depth. The term codeblock is used for RS coding, and when $I=1$, the terms codeblock and codeword are used interchangeably.

Codeword: In a block code, one of the sequences of length *n* in the range of the one-to-one transformation

Concatenation: The use of two or more codes to process data sequentially with the output of one encoder used as the input of the next.

Constraint Length: In convolutional coding, the number of consecutive input bits that are needed to determine the value of the output symbols at any time.

F Prime: A component-driven framework that enables rapid development and deployment of spaceflight and other embedded software applications.

FEC: Forward error correcting.

FPGA: Field programmable gate array.

HRM: Hold and release mechanism.

JPL: Jet Propulsion Laboratory.

OBC: OBC stands for on board computer.

Peripherals: A peripheral is a device that is used to collect information from the surrounding environment. The satellite's peripherals are: Reaction Wheels (RW), Inertial Measurement Unit (IMU), Sun Sensor (SS), and Star Tracker (ST).

PHM: The PHM is a system that synchronously runs in the satellite to ensure that the satellite is in good "health". Variables in the PHM (such as state of charge (SoC), temperature, etc) have both an upper and lower bound, and if either one of these are exceeded, the satellite will immediately break out of the state and go into Safety 1.

PHM flags are always set before the PHM system is instantiated. An important note to keep in mind is that when the satellite transitions to another state, PHM flags of the previous state are still in effect until the current state's PHM flags are set. For example, if the satellite were to transition from Charge to Comms, the PHM flags for Charge are carried over until the PHM flags for Comms are set.

The PHM flags could be the same between two states. However, this is unlikely to happen, so it's better to overwrite every PHM flag to maintain consistency between states.

Peer Watching: A peer watch timer is a timer runs on the same clock signal as processes are running. If this software timer is exceeded, then it will be logged in diagnostics appropriately and the software will continue to move through the state accordingly.

QEMU: A generic and open source machine emulator and virtualizer.

RF Inhibit: The RF Inhibit is a hardware constraint that determines if the satellite has the green light to communicate to another entity. This is a timer that cannot be bypassed by any means. Thus, the only way to bypass this timer is to wait for the timer to time out.

ROM: Read-only memory. This is a stable type of memory that cannot be written to.

RW: Reaction wheel.

RW Desaturation: Reaction Wheel (RW) Desaturation is a process/state that allows for the wheels to desaturate. When these wheels built up enough momentum to exceed the maximum speed of the wheel, it can be said that the wheel is "saturated". Thus, RW Desaturation has to happen to slow the momentum to get the wheel back under control.

Thermal Set Point (TSP)

The TSP is a temperature is set so the satellite has a nominal temperature to be. For example, if the TSP is set to 10°C, then the satellite should be around temperature is 10°C. The Thermal Control subsystem maintains the integrity of the TSP.

Watchdog timer: This timer is also a software timer but runs on a separate clock signal. If this timer misses a kick/nibble, then it will shut the satellite down, signifying a larger error has occurred.

Yocto Project: An an open source collaboration project that helps developers create custom Linux-based systems regardless of the hardware architecture.

f kid

11 References

[1] "Board Linux Development Manual." MYIR, 3 Jan. 2018.

[2] Bouwmeester, Jasper, Martin Langer, and Eberhard Gill. "Survey on the implementation and reliability of CubeSat electrical bus interfaces." *Ceas space journal* 9.2 (2017): 163-173.

[3] "Build Device Tree Blob." *Confluence*, Xilinx, 13 Aug. 2020, xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842279/Build Device Tree Blob#BuildDeviceTreeBlob-TaskOutputProducts.

[4] Canham, Timothy. "F' User's Guide."

github.com/nasa/fprime/blob/master/docs/UsersGuide/FprimeUserGuide.pdf.

[5] Canham, Timothy. “F’ Software Framework.” https://github.com/nasa/fprime/blob/master/docs/Architecture/FPrimeSoftwareArchitecture.pdf.

[6] CCSDS 130.1-G-3, Informational Report, TM Synchronization and Channel Coding-Summary of Concept and Rationale, June 2020

[7] CCSDS 131.0-B-3, Recommendation, TM Synchronization and Channel Coding, September 2017

[8] Cook, Barry M., and C. Paul H. Walker. " SpaceWire on FPGA–Challenges and Solutions." *DASIA Conference, Majorca, Spain*. 2008.

[9] Deep Space Network Telemetry Data Decoding. JPL Publication 810-005-208, Rev. B. California Institute of Technology: JPL, January 10, 2013

[10] M. Perlman and J. J. Lee. Reed-Solomon Encoders—Conventional vs. Berlekamp's Architecture. JPL Publication 82-71. Pasadena, California: JPL, December 1, 1982.

[11] “MYD Y7Z 010/007S Development Board Hardware User Guide.” MYIR, 3 Jan. 2018.

[12] Petazzoni, Thomas. “Introduction to Embedded Linux.” *Bootlin.com*, Bootlin, 2018, bootlin.com/pub/conferences/2011/limoges-clermont/presentation.pdf.

[13] R. J. McEliece. “The Decoding of Reed-Solomon Codes.” TDA Progress Report 42-95, July-September 1988 (November 15, 1988): 153–157.

[14] “What Is an SoC FPGA?” *Intel.com*, Altera, 2014, www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ab/ab1_soc_fpga.pdf.

[15] “Zynq-7000 SoC Data Sheet: Overview.” *Xilinx.com*, Xilinx, 2 July 2018, www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.