Georgia Tech Small Satellite Real-Time Hardware-in-the-Loop Simulation Environment: SoftSim6D

Sean B. Chait

Advisor: Dr. David Spencer

AE 8900: Special Problems

Fall 2015

Georgia Institute of Technology

School of Aerospace Engineering

Space Systems Design Lab

THIS PAGE IS INTENTIONALY LEFT BLANK

"Well, actually, we have a lot better rockets than the coyote."
-Dan Truman, Armageddon

Acknowledgements

To my fiancé, Jessica, for her unwavering support while enduring

the many late nights, long hours, and extended trips in pursuit of rockets.

Abstract

The capabilities of small satellites produced by the university and small business community have seen a sharp rise in recent years. With this growth in capabilities has come an increase in mission complexity to encompass those architectures previously only found in well-funded government programs, including proximity operations. The inherent complexity of proximity operations-based missions introduces a great deal of risk to the mission's success. The low-budget nature of the small satellite community has limited the development of relevant testing infrastructure to match the pace of mission complexity increase to adequately mitigate risk. This research will leverage the standardization of CubeSat components to develop a highly adaptable hardware-in-the-loop testing capability for the verification and validation of small satellite avionics boards and flight software. MATLAB© Simulink Real-Time will be utilized to create a user friendly framework that can easily be adapted to support a wide range of small satellite mission architectures. This architecture, known as SoftSim6D, has been designed to thoroughly exercise the robustness of a satellite with the primary aim of minimizing mission risk to ensure full mission success. An examination of the overall framework, verified capabilities, and current variants will be discussed.

Acronyms

6DOF	Six degrees of freedom
ADACS	Analog to Digital Acquisition System
ADC	analog to digital converter
CML	Communication Management Layer
COTS	Commercial-off-the-Shelf
CPM	Communication Processing Module
DBM	Data Buffer Module
DDOS	Distributed Denial of Service
DPL	Data Preparation Layer
DTL	Data Transmission Layer
ECEF	Earth Centered Earth Fixed
ECI	Earth Centered Inertial
EGSE	Electrical ground support equipment
ENV	Environment data bus
FIFO	First In First Out
FOV	Field of view
GB	Gigabyte
GN&C	Guidance, navigation, and control
GPS	Global Positioning System
HWITL	Hardware-in-the-Loop
I/O	Input/output
IBLE	Integrated Base Level Environment
IMU	Inertial measurement unit
INIT	Initialization data bus
LEO	Low Earth Orbit
LOS	Line of sight
LSB	Least Significant Bit
MADS	Modular Attitude Determination System
MEO	Middle Earth Orbit
MSFC	Marshall Space Flight Center
PCI	Peripheral Component Interconnect
RAM	Random access memory
SSF	Sensor fixed frame
SICD	Software interface control document
SSIP	Spacecraft and Simulation Initialization File
STATE	State data bus
STK	Systems Tool Kit
TAB	Test avionics board
UART	Universal asynchronous receiver/transmitter
VDF	Variant Definition File

Table of Contents

Table of Figures

Table of Tables

1. Introduction

1.1. *The Growing Need of HWITL Testing*

Throughout the history of spaceflight, relative proximity operations and rendezvous have undergone a significant evolution from human-in-the-loop to ground-in-the-loop to varying levels of autonomy. Due to the inherent complexity of automated proximity operations, the development of such a system presents a high operational and cost risk to developing organizations. Errors in algorithms or flight coding that are not caught through testing have the potential to result in a mission failure. This reality is what makes it difficult for mission designers to truly remove the ground from on-orbit maneuver planning and allow the system complete autonomy. The only way to guarantee the system is robust enough to be able to operate on a completely autonomous basis is to have a comprehensive ground test program designed specifically to exercise the system in such a way that faults in the system (if any) will present themselves in a laboratory environment as opposed to during mission critical operations.

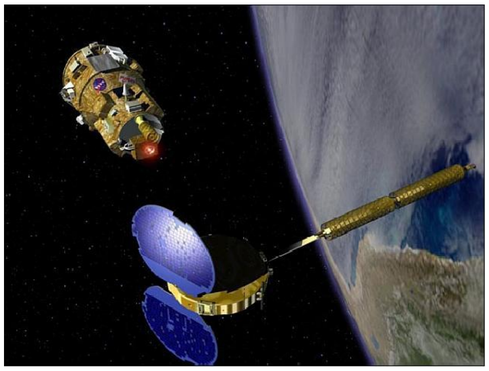


Figure 1: DART Concept Visualization *Invalid source specified.*

Autonomous proximity operations-based missions are by definition inherently risk prone as they involve at least one spacecraft maneuvering in close quarters to another space object. A slight miscalculation or incorrect reaction can create the potential for a collision resulting in a mission failure and possibly result in the loss of both space assets. This inherent risk further backs the needs for a system dedicated to the comprehensive check out of a satellite's guidance, navigation, and control (GN&C) system so as to verify the robustness of the system. Multi-million dollar class missions often undergo extensive testing regimes but without a system independently designed to reach these testing goals, mission failure is still a possibility. This was shown in the NASA DART mission (illustrated in Figure 1) where inadequate software requirements and software failures resulted in a collision with the MUBLCOM spacecraft and the loss of a $110 million mission [1]. The MUBLCOM spacecraft was not critically damaged by the collision, but loss of the DART mission shows the risk involved in autonomous relative operations even for well-funded programs. The high profile Orbital Express mission was also plagued with failures due to incorrect software implementation. Although the mission was recovered through on-orbit software updates, it further stresses the potential for these types of failures even in the best-funded situations [2]. The possibility of failure is not an option if such autonomous systems wish to be used to service high-value assets or in support of manned spaceflight.

With the rising popularity of CubeSats and small satellite platforms, the cost of producing these spacecraft has decreased drastically allowing complicated missions to be developed on the scale of hundreds of thousands of dollars as opposed to tens of millions. This has resulted in the addition of large numbers of universities and start-up companies to the small satellite community. As these organizations operate on very restricted budgets, they often do not have in-house test capabilities that are capable of fully testing complex GN&C systems. Although they do not have this capability, it has not deterred them from developing proximity operations-based missions such as Tyvak Nano-Satellite Systems CubeSat Proximity Operations Demonstration (CPOD) and Georgia Tech's Prox-1 mission [3] (see Figure 2), [4]. These missions, by design, are high-risk operations but these risks are further increased by limited capability and funding to perform extensive testing prior to operations. It is thus desirable for there to exist a highly adaptable test capability to reduce operational risk but also at minimal expense such that low-budget missions can still afford to adopt more in depth test programs.

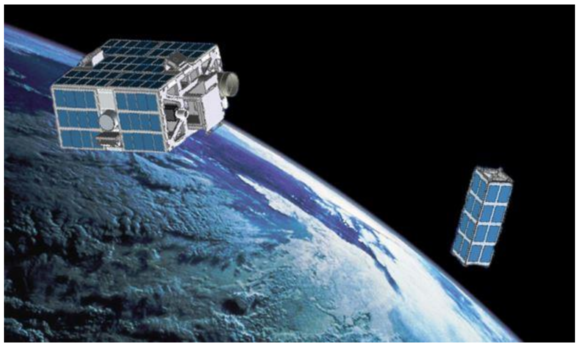


Figure 2: Prox-1 Mission Visualization

1.2. *Previous Approaches*

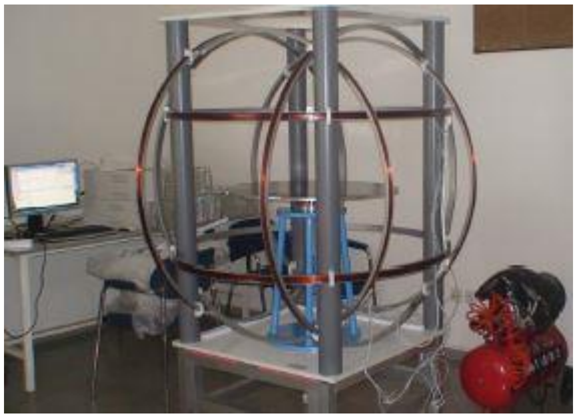


Figure 3: ITU-PSAT II Test Setup [5]

There are several universities and organizations that have developed various levels of such a capability but there are limitations to many of these approaches. One prevalent method is the physical augmentation of an integrated spacecraft to measure system performance. This involves the use of rate tables, Helmholtz cages, image simulators, as well as other devices to emulate the on-orbit spacecraft performance. A prime example of this is MIT's testing system designed for testing the attitude determination and control system used with the ITU-PSAT II mission which utilized an air bearing table and Helmholtz cage [5] (shown in Figure 3). Although it has been proven to verify certain portions of ITU-PSAT II's ACS system, it is limited in scope and not easily reconfigurable for additional scenarios. Similar approaches can be found at other institutions, but they all present the issue that they cannot be easily reconfigured for additional scenarios without extensive time and cost [6], [7], [8], [9].

Another approach removes hardware actuation from the loop and focusses on software emulation of sensors and actuators [10], [11], [12]. This approach has been successfully utilized in the development in Marshall Space Flight Center’s SPRITE tool for HWITL testing of CubeSats. Here the plant dynamics of a spacecraft are simulated and resulting sensor readings are generated to be fed into a spacecraft’s flight computer [11]. The benefit to this approach is that the primary adaptations required between different testing scenarios are software based, not hardware, and thus reduce the complexity and cost of a reconfiguration. In the small satellite community, this is desirable as cost and schedule are often limiting factors in the extent of testing which will be conducted. Although this approach does not verify individual sensor and actuator performance and rather focusses on the spacecraft’s avionics, it can be argued that this approach is highly valuable.

Multiple approaches have been used in the past in order to tackle the problem of thoroughly testing complicated mission architectures. However, capabilities that have been developed have either been limited in scope or lacked the capability to be easily adapted for other mission architectures. The need clearly exists for the development of a reconfigurable system that can test a wide range of mission profiles for different spacecraft while still remaining cost effective for the small satellite community.

2. Methodology

2.1. *Approach Selection*

After conducting a survey of pre-existing testing environments it was determined that a framework which would achieve the most utility throughout the entire lifecycle of the mission and provide the greatest platform for fully exercising a spacecraft's avionics system should focus on the avionics of the spacecraft. Physical actuation of an integrated spacecraft was considered as a possible option, however it was decided that this would only have limited utility in fully verifying the system. Physical actuation (rate tables, Helmholtz cage's, etc.) would provide the ability to partially test nominal mission performance but would not give testing engineers the ability to adequately introduce off-nominal scenarios to fully characterize the system. These systems also often require significant modification between spacecraft which introduce large cost and schedule implications making the system less desirable. However, a testing environment based on integrated avionics testing can both fully exercise the system's performance and be adaptable enough to make it a viable "generic" testing environment.

In the small satellite community, the vast majority of sensors and actuators used on-board satellites originate from commercial vendors as opposed to custom-designed, mission-specific solutions. As these Commercial-Off-The-Shelf (COTS) components gain more flight heritage, the validity of the performance specifications for these units has increased dramatically. It is for this reason that oftentimes it is safe to take these specifications at "face" value and not invest extensive resources in verifying individual sensor performance. This is especially the case for university-based and other low-budget missions where this sort of testing is not within the budgetary allowances of the program. Since these components have a high probability of meeting performance specification during operation, it is possible to remove them from the testing chain. This fact allows us to now formulate the main methodology behind the HWITL test bed.

As previously discussed, the largest risk to a successful mission lies with the successful implementation of the hardware/software interface as well as adequate definition and implementation of flight software requirements. The software interface with hardware can often prove to be a complicated interface to design for nominal conditions. Without proper requirements and extensive testing, this software interface may not be able to properly function should the hardware malfunction. In university-class missions, requirements concerning off-nominal hardware communication are often not adequately defined, thus requiring further testing to verify functionality. A major cause of mission failure also lies with the definition of flight software requirements rather than their implementation. Requirements may be developed, implemented, and successfully tested, but if those requirements do not adequately encompass the true needs of the mission, failure is still possible. For instance, one of the causes for the failure of the DART mission was a GPS velocity error bias of 0.6 m/s. The design requirements stated that the measured velocity error must be within ±2 m/s (meaning this error fell within requirements). However, as the failure of the mission proved, this requirement was not properly defined and thus contributed to the mission failure [1]. A key ability of the HWITL platform will be to rigorously test the spacecraft avionics in a system realistic enough such that potential in-flight errors caused by poor requirements definition will be revealed.

To replace the physical simulators and actuators that we have removed from the testing chain, the HWITL platform will instead mimic the low level output of each of these components. As opposed to many testing schemes where this data would be fed into the flight computer via an electrical ground support equipment (EGSE) connection, this approach would feed the emulated component I/O directly into the hardware connections on the avionics boards where the actual component would be connected. Introducing component signals at this level would therefore allow the complete verification of the entire avionics hardware and software chain while still allowing the test engineer a great deal of control over the system. This capability is especially important in distributed architectures where information passes through multiple levels of signal and data processing before reaching the primary flight processor. EGSE connections typically bypass all lower level hardware/software and pass data directly to the flight computer. Although this form of testing may verify performance of software on the primary flight computer, it does not provide any verification for the multiple lower levels of hardware/software that in reality sensor data would need to pass through before it reaches the flight computer. Therefore the EGSE methodology would effectively be ignoring a large number of potential fault locations and not truly validating the robustness of the system.

The resulting environment will therefore need to be a full avionics in the loop test bed, capable of exercising all of the avionics electronics of the spacecraft, from low-level serial communication and associated basic electronics such as logic level converters to subsystem level microprocessors to the fully integrated avionics system with the primary flight computer in the loop.

2.2. *Framework Requirements*

The scope of the HWITL framework has already been defined as an avionics-in-the-loop test bed capable of fully exercising a spacecraft's flight avionics system. From here three primary driving requirements have been defined. From this point forward, the framework will be known as SoftSim6D.

Requirement 1

The test bed shall be a robust Hardware-in-the-Loop avionics testing environment with primary emphasis on supporting the development, verification, and validation of autonomous proximity operations based mission systems

Requirement 2
The system shall be capable of supporting spacecraft projects throughout the mission lifetime, from initial development, to engineering and flight unit testing to flight anomaly mitigation during on-orbit operation.

Requirement 3
The system shall be highly adaptable such that it can be rapidly configured for a new mission with no to minimal simulation development required.

2.3. *System Architecture*

Initial requirements definition of the SoftSim6D framework determined that the framework was to be both highly adaptable for specific mission requirements and capable for use during all phases of a spacecraft design lifecycle. SoftSim6D was designed with the intention that it can be used for development of MATLAB/Simulink control algorithms and mission design, testing of flight C code, and Hardware-in-the-Loop (HWITL) testing of flight avionics boards. In support of this, a framework was developed with three distinct layers: simulation, emulation, and interface.

The simulation layer is the primary engine of the environment, consisting of a series of high fidelity environmental, perturbation, and dynamics models. Environmental models generate the Earth-centric ephemerides of the spacecraft, sun, and moon. Perturbations caused by atmospheric drag, solar radiation pressure, spherical gravity harmonics, and third body effects are modeled. Accelerations and moments caused by these phenomena are fed into translational and rotational dynamic plants along with physical characteristics of the spacecraft to create a high fidelity six degrees-of-freedom environment.

The emulation layer is responsible for the simulation of spacecraft components and consists of two parts: spacecraft sensors and actuators. The spacecraft sensor block takes the true spacecraft state as determined by the simulation layer and generates the corresponding sensor readings for a suite of generic spacecraft sensors. The actuator block acts upon commands received from the test article and generates the resulting forces and moments to be fed back into the simulation level for propagation of the spacecraft state.

For a given spacecraft or flight program, once components are selected, the only modifications that will be required to interface with a given test article will be the interface layer. For basic testing of MATLAB/Simulink algorithms, this layer will simply generate data buses to be fed directly into the provided MATLAB code. For testing of flight C code and HWITL testing, the interface layer will act as the interpreter between the simulation and test module, generating realistic low-level input/output (IO) to model flight hardware conditions as realistically as possible. The layered architecture is illustrated in Figure 4: High Level Architecture OverviewFigure 4.

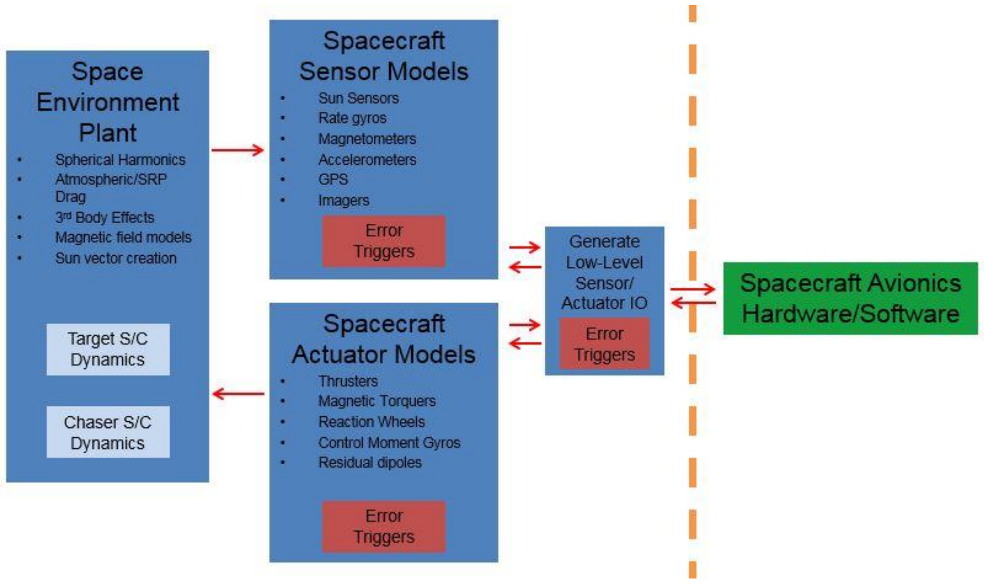


Figure 4: High Level Architecture Overview

A key attribute of this architecture is that it is highly adaptable and configurable such that it is able to accommodate a wide range of mission profiles, sensors, and testing requirements. As such, a standardized plant framework has been developed for all models to allow for new models to be "plugged" into the simulation, minimizing rework between each satellite. Generic models have also been developed for "standard" classes of COTS components such as reaction wheels, cold gas thrusters, inertial measurement units (IMU's), etc. with easily changeable configuration parameters to allow the plant models to be updated for different versions of hardware.

Different mission profiles can call for different fidelities in their environments models based on mission requirements. For example a spacecraft with a deployable boom operating in LEO would have greater concern about the effect of atmospheric drag on system performance than a communications satellite in MEO. It is for this reason that environment models such as atmospheric density, solar radiation pressure, and Earth's magnetic field will also be treated as interchangeable components within the overall framework. For example, the testing of a specific satellite may call for the use of a specific high fidelity magnetic field model not included in the standard HWITL framework libraries. To prevent the need for substantial code change to accommodate a new model, the specific model utilized by a simulation run will be another configuration parameter with a standard interface format.

3. Simulink Real-Time Implementation

3.1. *Overview*

This architecture has been implemented using MATLAB© Simulink Real-Time via a real-time target machine. Using this framework, the test environment will have three primary components; the host machine, the target machine, and the test article. The test article will either be the spacecraft avionics boards undergoing testing or MATLAB/Simulink algorithms. The architecture implementation is illustrated in Figure 5.

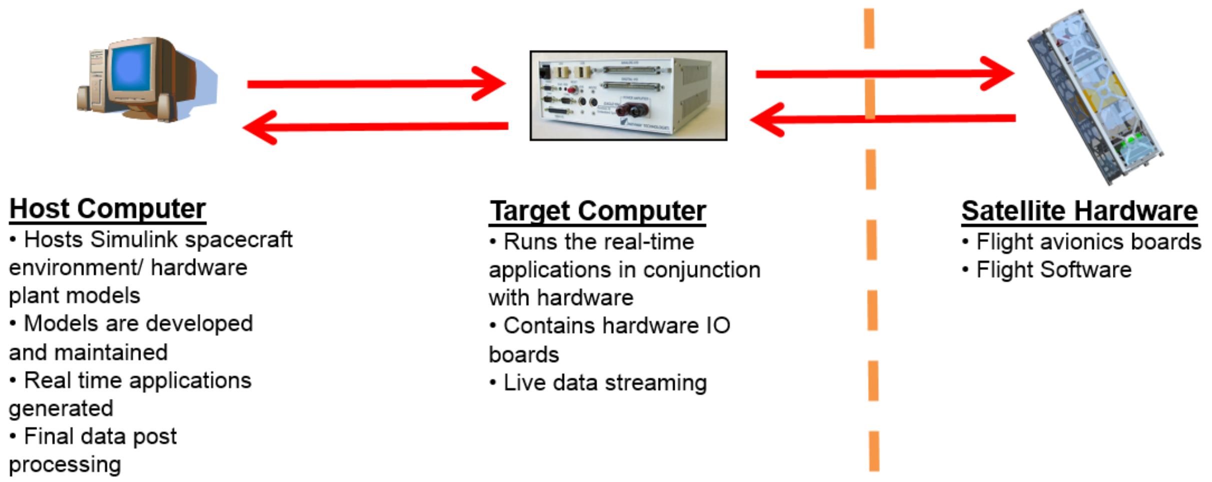

Host Computer
• Hosts Simulink spacecraft environment/ hardware plant models
• Models are developed and maintained
• Real time applications generated
• Final data post processing
Target Computer
• Runs the real-time applications in conjunction with hardware
• Contains hardware IO boards
• Live data streaming
Satellite Hardware
• Flight avionics boards
• Flight Software

Figure 5: Framework Hardware Implementation

The host machine is where the simulation is designed and configured for the specific test run via MATLAB© Simulink. This is where all spacecraft parameters are set, new models are defined, and simulation management occurs. When the simulation is completed it is compiled into a Simulink Real-Time C application and loaded onto the target machine for execution. Depending on the test scenario, the application can be compiled to run in real-time, for HWITL testing, or free-run mode, for algorithm testing and verification. Free-run mode is an accelerated mode which will execute the simulation as fast as the hardware capabilities of the target machine allow.

The target machine is where the execution of all testing occurs. The target machine is a modified PC that is booted into a MATLAB© kernel from an external USB drive. By using this kernel, the target computer does not load a traditional operating system which requires substantial processor overhead. Rather the purpose of the kernel is solely for communication with the host computer and management of the simulation. This allows the C application to utilize the complete power of the processor and drastically increase the speed of any real-time or free run simulations. Data monitoring during testing will occur via this machine while the final test data will be transferred to the host computer over crossover Ethernet for post processing. The MATLAB© kernel also allows for real-time communication between the target and host computers both before and during testing. As will be discussed later, this allows the test engineer to quickly configure the simulation from the host machine as well as change parameters or insert faults while a simulation is running.

The target machine will also host the low level IO interface cards which will be responsible for communication with the spacecraft avionics during HWITL testing. The wiring harness that interfaces the real-time target PC with the avionics board will be fabricated for each spacecraft being tested such that the connections are identical to those which the spacecraft would see from the real component.

For scenarios where MATLAB/Simulink algorithms are undergoing testing in lieu of hardware, during the configuration of the simulation on the host machine, the algorithms will be directly inserted into the simulation. These algorithms will then be compiled into the C application with the rest of the simulation and transferred to the target machine. Execution will still occur on the target machine so as to take full advantage of the increased simulation speeds allowed by the standalone MATLAB kernel.

3.2. *Data Bus Formulation*

To allow for easy configuration, expansion, and data management MATLAB data buses have been used to track all states, logic flags, and data products throughout the simulation. This was done intentionally to allow for easy management/access to all state information and to allow for the easy use of variable models that enable the simulation to be configured without substantial user input. As will be discussed, the specific utilization of data buses was designed to allow for development and implementation of new models in a plug and play fashion.

Three major data buses exist in all simulations, regardless of the configuration, test article, or models utilized. These are the environment (ENV), state (STATE), and initialization (INIT) buses. Each are required for the successful propagation of all dynamic and kinematic models as well as for the modelling of sensors and actuators. The environment bus is responsible for tracking all time conversions and any processes that exist external of the spacecraft. The state bus contains all information pertaining to the spacecraft inertial state, rotation matrices, and mass properties. The initialization bus was developed to allow for rapid configuration of the entire simulation with minimal modifications required. This bus contains all information on the spacecraft initial states as well as information pertaining to any other physical or performance characteristic (such as surface areas or sensor noise parameters). Updates to the default values of this bus allow the user to automatically configure many aspects the simulation at start-up. Data buses for sensor data

(SEN_DATA) and actuator response (SC_RESPONSE) are also defined within the baseline environment, however these will need to be updated to match the data sets of the spacecraft under examination.

3.3. *Variable Models*

A primary requirement of SoftSim6D was to have the ability to rapidly configure the simulation for different perturbation models, sensor models, etc. without substantial user effort. To meet this need, Simulink Model References and Variant Subsystems were utilized. Model References allow the generation of custom Simulink blocks for insertion into a high-level model. A model reference block is a standalone Simulink model that is configured in such a way that, it can be inserted into another model as a block as opposed to a subsystem. This enables easy configuration management as well as speeds up compile time of the overall simulation. Generic blank Simulink models configured for this purpose have been generated for each major variation subsystem within SoftSim6D to simplify the development of future modules.

Simulink Variant Subsystems are a powerful tool that enables much of SoftSim6D's rapid configuration abilities. A variant subsystem allows the definition of multiple instances of the same subsystem, however only one is active at a time, as determined by an external setting. This setting can be set at initialization and thus allows the multiple models to be rapidly interchanged without manual manipulation of the model. In SoftSim6D, each variant is an externally defined model reference block. To ensure proper functionality, all variants placed within a given variant subsystem are required to have identical inputs and outputs. Universally across the simulation, with few exceptions, all variants/model references contain the three standard buses (ENV, STATE, and INIT). These were designed to contain the necessary information for all derived calculations within SoftSim6D. The outputs of each variant model are dependent on the specific application and are typically simple vector outputs. New bus definition (such as SEN_DATA and SC_RESPONSE) occur at a higher level within the simulation.

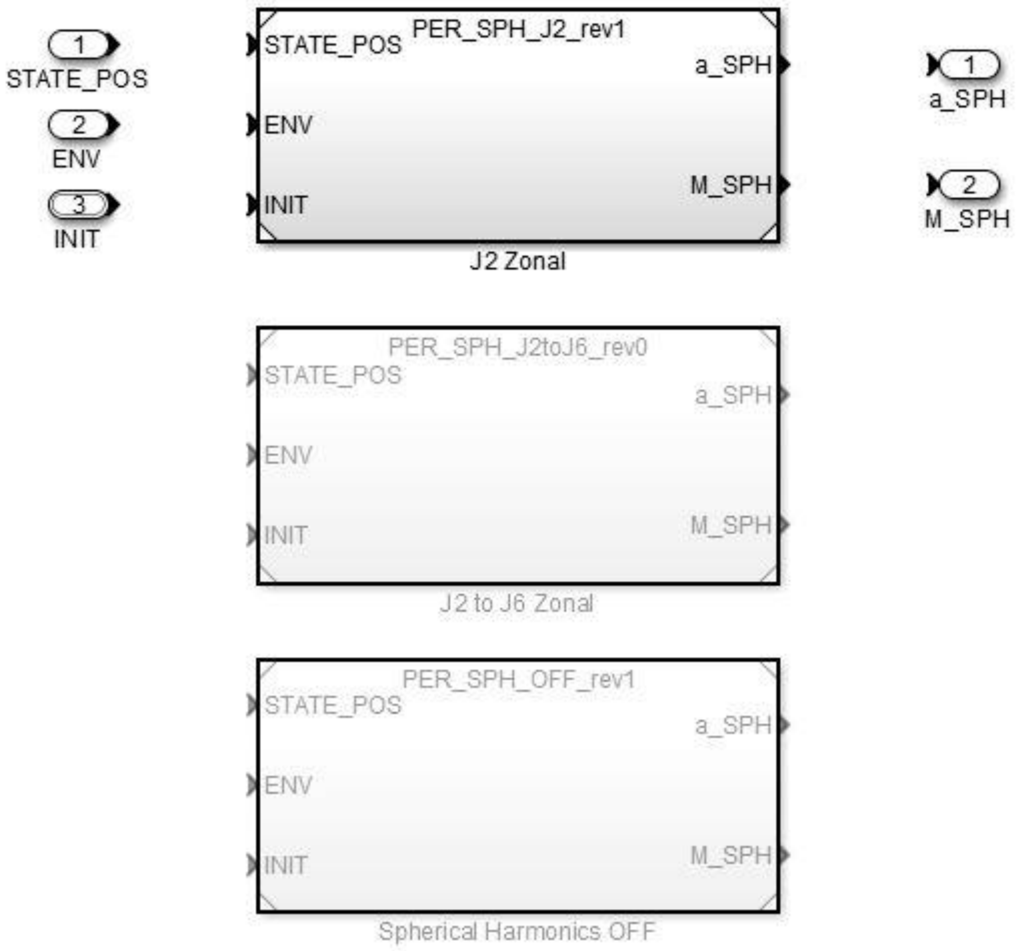


Figure 6: Variant Subsystem Example: Spherical Harmonics

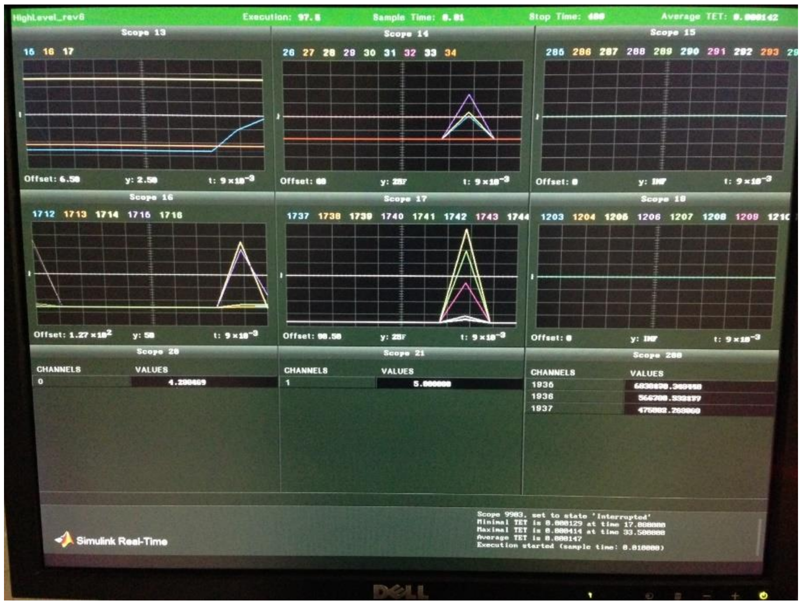
Figure 6 shows an example of a variant subsystem. The displayed subsystem is for the spherical harmonics perturbation model. Three variants currently exist within the model; no spherical harmonics, only J_2 zonal harmonics, and J_2 through J_6 harmonics. Currently the J_2 zonal harmonics model is active (this can be discerned by the fact that the other two models appear grayed out). To change which model is active, the user simply needs to alter the single parameter SPH_Option in the initialization file. No other action is required. To expand on previous discussion of standard model reference formats, it is important to note that these models produce both a translational acceleration and moment result. Although basic spherical harmonics do not produce torques on the spacecraft, configuration control requires that all perturbation models have both acceleration and moment outputs. In this case, the moment output of the models is of value zero.

Variant subsystems are used throughout SoftSim6D in any subsystem which is conducive to the desire to have numerous options available for rapid configuration. Expansion of a variant subsystem has also been designed to be straightforward. The user needs only to create their new model within the generic template and save it according to configuration standards. A new model reference corresponding to the new block is then inserted into the relevant variant subsystem. To be accessible as a rapid configuration option, a single variant control and condition must be added to the variant subsystem's block parameters. After this is completed, the new model is completely integrated and ready for testing.

3.4. *Data Visualization and Storage*

There are three primary means of data visualization and storage while executing a simulation on the target machine. Each methodology takes advantage of pre-supported Simulink Real-Time capabilities and their use customized to support the needs of SoftSim6D.

During all testing, a monitor is connected to the target machine. This monitor is primarily used for displaying real-time information and visual confirmation of simulation settings. Up to 9 plots of type Target Scope can be displayed on this screen during simulations. Target Scopes sole use is for the displaying of information while a test in underway and do not provide for a means of storing data for post processing. It is for this reason that no target scopes are permanently configured in SoftSim6D. The test engineer has the latitude to insert these where desired within the simulation to monitor any desired signals. For HWITL testing, these scopes are useful for verifying successfully communication with test hardware. Figure 7 shows a HWITL test where a non-zero signal in the top row scopes signify incoming data and the bottom row signifies outgoing communications. This proves a useful sanity check to confirm proper operation during a simulation.



```
Scope 9903, set to state 'Interrupted'
Minimal TET is 0.000129 at time 17.000000
Maximal TET is 0.000414 at time 33.500000
Average TET is 0.000147
Execution started (sample time: 0.010000)
```

Figure 7: File Scope Example: HWITL Testing

The second and most used means of data monitoring and storage is through the use of Host Scopes. Although they are called scopes, their primary purpose is to save data from the simulation and not for plotting of data during execution. Host Scopes take the given input signal and save it in a Simulink Real-Time Scope object. Once all requested data has been gathered, the information is then transferred to host computer. Host scopes have been implemented across SoftSim6D to store all signals found in data busses and are found within Data Logging subsystems. This includes default storage utilities for ENV and STATE as well as for SEN_DATA and SC_RESPONSE. All listed signals are automatically acquired during every simulation run, in both real-time and free run modes. Additional scopes can also be readily added to capture new signals.

In certain situations, a simulation may need to be run at a relatively high frequency but the user may only require data acquisition at a slower frequency. All Host Scopes are configured to allow slower storage rates if desired and it is a parameter that is configurable at simulation compilation time. Host Scopes can also be utilized to display select data at near real-time. A real-time plotting utility has been created to allow user real-time viewing of state data. This capability is separate from the standard Host Scope configuration described previously as a different implementation path is required. An example host scope implementation using both long term data storage and real-time display is shown in Figure 8.

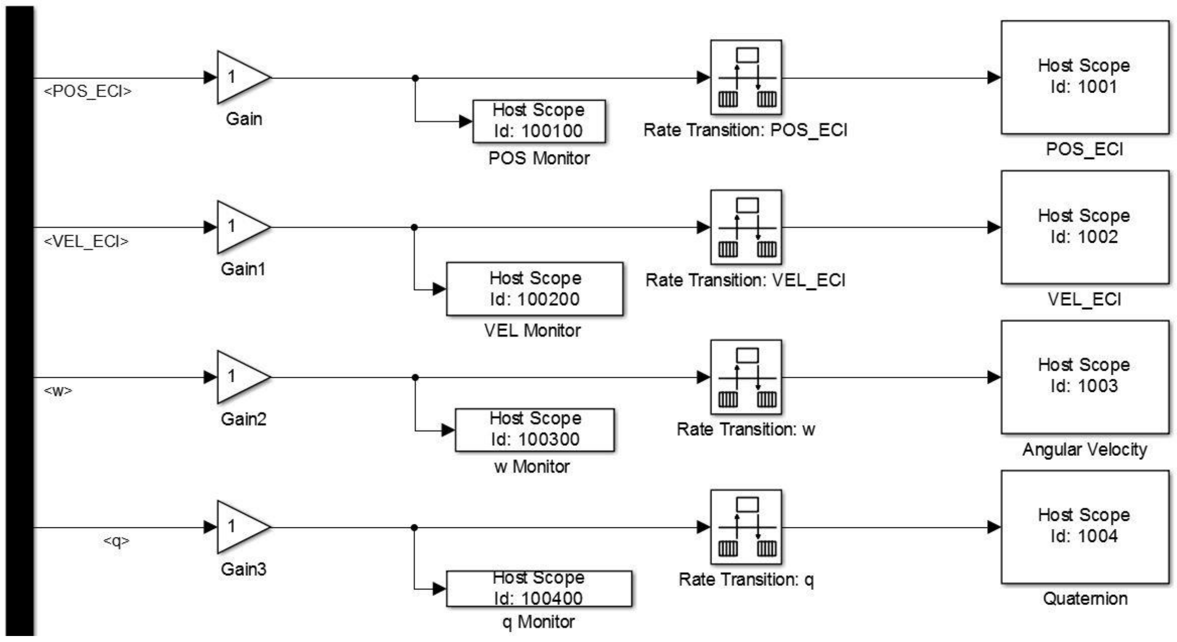
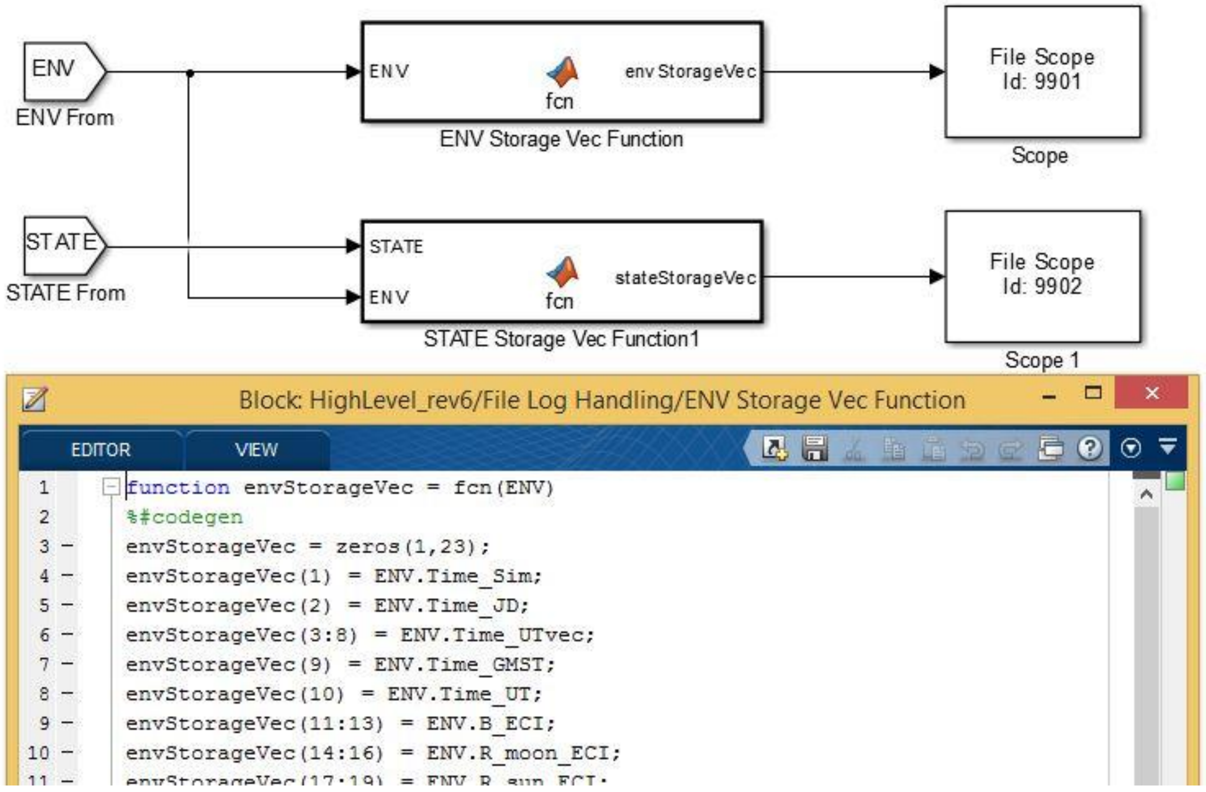


Figure 8: Host Scope Implementation Example

An unlimited number of Host Scopes are allowed by Simulink Real-Time, however limitations do exist on their use. Data found in Host Scopes are stored in the RAM of the target machine until the simulation is complete and the data is ported over to the host machine. Therefore if the number of signals saved, simulation duration, and sample frequency result in a data quantity that exceeds the 2 GB RAM limits of Simulink Real-Time, SoftSim6D will be unable to run. If this is the case, when the simulation is loaded onto the target machine prior to execution, an error will occur to inform the user to address the issue.

If Host Scopes do not provide sufficient data storage, the third method for data storage it to use File Scopes. These scopes save data directly to file system on the target machine and have no limit on size, however Simulink Real-Time is limited to 8 file scopes per simulation. Since there is no data limit on each of these scopes, signals are combined by bus in order to be stored concurrently in the same file. A significantly greater amount of custom configuration is required to both implement and extract data from File Scopes so it is recommended they are only used in long term HWITL testing where it is absolutely necessary to continuously store large amount of data for extended periods of time. Standard implementations have been generated for STATE and ENV data buses. An example of the ENV and STATE file scopes is shown in Figure 9.



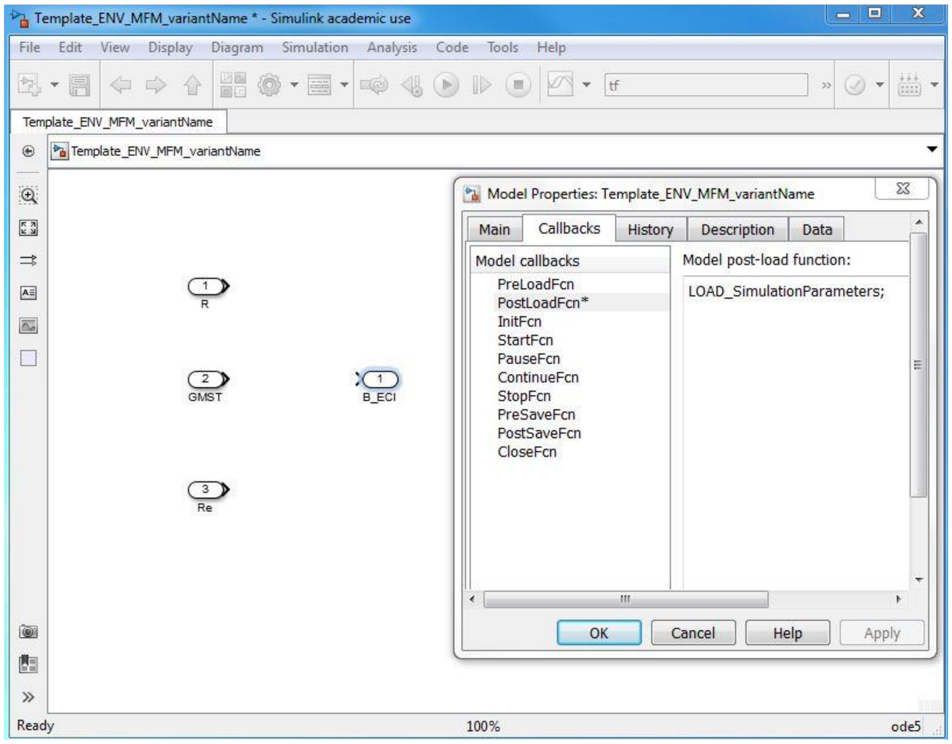
```
function envStorageVec = fcn(ENV)
%#codegen
envStorageVec = zeros(1,23);
envStorageVec(1) = ENV.Time_Sim;
envStorageVec(2) = ENV.Time_JD;
envStorageVec(3:8) = ENV.Time_UTvec;
envStorageVec(9) = ENV.Time_GMST;
envStorageVec(10) = ENV.Time_UT;
envStorageVec(11:13) = ENV.B_ECI;
envStorageVec(14:16) = ENV.R_moon_ECI;
envStorageVec(17:19) = ENV.R_sun_ECI;
```

Figure 9: File Scope Implementation Example

3.5. *Adaptability and Expansion*

The selection of Simulink Real-Time as the primary framework for SoftSim6D as well as the various methodologies already described for implementing models and storing data have all been done with the aim of making the simulation quickly adaptable and intuitive enough to allow new users to make additions. Model variants, spacecraft physical characteristics, initial state, component specifications, and simulation parameters have all been implemented in such a way that a test engineer can configure them at run-time without any modifications to the system. A detailed discussion of this is found in Chapter 5.

If a user requires a model not currently found in SoftSim6D libraries, generic Simulink models and code files have been created for the major elements of each layer. In each generic model, the IO with the corresponding higher level model has already been defined and all of the model parameters have been configured to automatically map to the settings of the higher level simulation. A user's guide is under development which further expedites this process. Figure 10 shows an example generic model.


Template_ENV_MFM_variantName * - Simulink academic use
File Edit View Display Diagram Simulation Analysis Code Tools Help
tf
Template_ENV_MFM_variantName
Template_ENV_MFM_variantName
1 R
2 GMST
3 Re
1 B_ECI
Model Properties: Template_ENV_MFM_variantName
Main Callbacks History Description Data
Model callbacks
PreLoadFcn
PostLoadFcn*
InitFcn
StartFcn
PauseFcn
ContinueFcn
StopFcn
PreSaveFcn
PostSaveFcn
CloseFcn
OK Cancel Help Apply
Ready
100%
ode5
Model post-load function:

LOAD_SimulationParameters;

Figure 10: Generic Model Variant: Magnetic Field Model Example

Detailed spreadsheets have been maintained for configuration management and revision control of existing models, tracking of data bus objects, and scope data ID trackers. Naming and numbering conventions have been developed and documented for all processes that are utilized across the simulation to ensure the plug and play capabilities of SoftSim6D.

3.6. *Target Machine and Hardware IO*

The target machine utilized in the Georgia Tech simulation environment is a standard Dell OPTIPLEX GX620 with a Pentium D processor. The machine has 4 GB of RAM (although only 2 GB are useable by Simulink Real-Time due to limitations in the MATLAB kernel). A 1 terabyte (TB) configured to a FAT-32 file system has also been added to allow for the use of File Scopes. A standard USB flash drive has been written as a boot-disk for the MATLAB kernel and the BIOS of the target computer is configured to automatically boot from this flash drive at start-up. An Intel PWLA8391GTL Ethernet Card has also been added to both the target machine and host computer. This specific type of Ethernet card allows for direct communication between the target and host computer via a cross-over CAT5 cable. A PCI expansion bus has also been added to the target machine to allow for a total of five PCI IO cards, used for HWITL testing. The current target machine is shown in Figure 11.

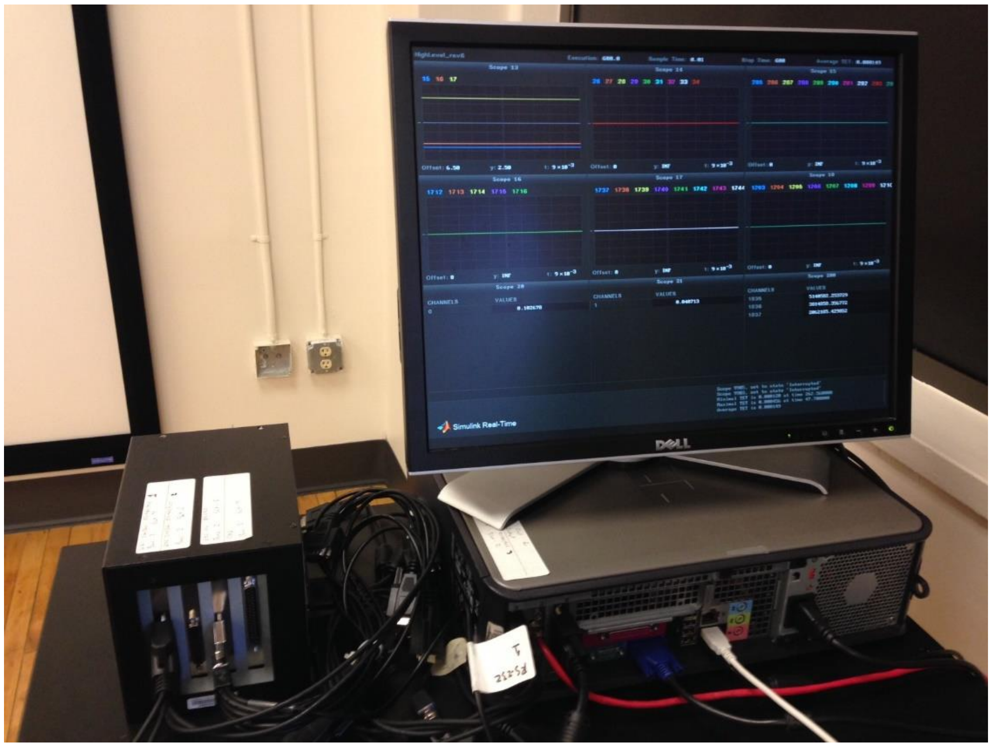


Figure 11: Georgia Tech Target Machine

To date IO cards have been added to the system to support serial communication over RS-232, RS-422, and RS-485 protocols as well as for the reading and generation of analog signals. All IO cards were selected based on their protocol, number of available IO lines, and pre-compatibility with Simulink Real-Time. A list of current IO cards and IO capabilities of the target machine is found in Table 1. The integrated IO cards in the PCI expansion bus attached to the target machine is shown in Figure 12.

Table 1: Current IO Capabilities of Target Machine

	Communication Protocols				
IO Card	**RS-232**	**RS422/485**	**TTL**	**Analog to Digitial**	**Digitial to Analog**
Quatech ESCLP-100	8	0	0	0	0
Quatech QSCLP-200/300	0	4	0	0	0
Quatech QSCLP-200/300	0	4	0	0	0
Quatech QSCLP-200/300	0	4	0	0	0
PCIM-DAS1602/16	0	0	8	16	2
Target Computer Total	8	12	8	16	2

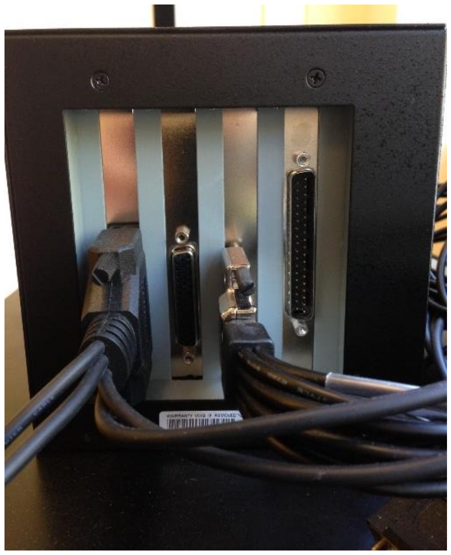


Figure 12: Target Machine PCI Expansion Bus

4. System Modelling

As previously discussed, SoftSim6D can be broken down into three distinct layers; simulation, emulations, and interface. Chapter 4 will discuss the internal workings of these layers and examples of functionality that has been implemented to do. Chapter 5 will discuss the integrated simulation environment that is created using these layers.

4.1. *Simulation Layer*

4.1.1. Overview

The simulation layer is the primary physics engine of the environment. This layer contains all models of the space environment, true spacecraft dynamics, and external perturbations. The states generated and monitored by this layer are fed into the emulation layer for use in generating sensor data. The simulation layer is also responsible for updating the true inertial state of the spacecraft based on commands interpreted by the interface layer and executed by the emulation layer.

4.1.2. Environment Models

Environments are defined as any processes of state that exist external of the spacecraft and are not dependent on the state of the spacecraft. Calculation of states of planetary bodies are calculated here, such as the positions of the Sun and moon in the Earth Centered Inertial (ECI) frame. Any quantities derived from these states, in conjunction with either time or the spacecraft state, are also determined. This includes the rotation from ECI to the Earth-Centered-Earth-Fixed (ECEF) frame, the eclipse state of the spacecraft, and the sun line-of-sight (LOS) vector from the spacecraft.

The framework also classifies any processes of the primary central body as environment models. The Earth's magnetic field is calculated with respect to the ECI frame at the current location of the spacecraft as well as atmospheric density.

Primary time keeping of the simulation time and current Epoch along with all other derived time measurements (GMST, Julian date, GPS Week and Second, etc.) are all monitored and propagated from the environment. Any use of time throughout the entire HWITL is calculated here. In sensitive GN&C missions, time tags on sampled data is of the utmost importance and even a variation in rounding within the simulation can have undesirable effects. It is for this reason that all time related calculation have been consolidated to the environment model to ensure 100% timing consistency. The implementation of this layer is shown in Figure 13.

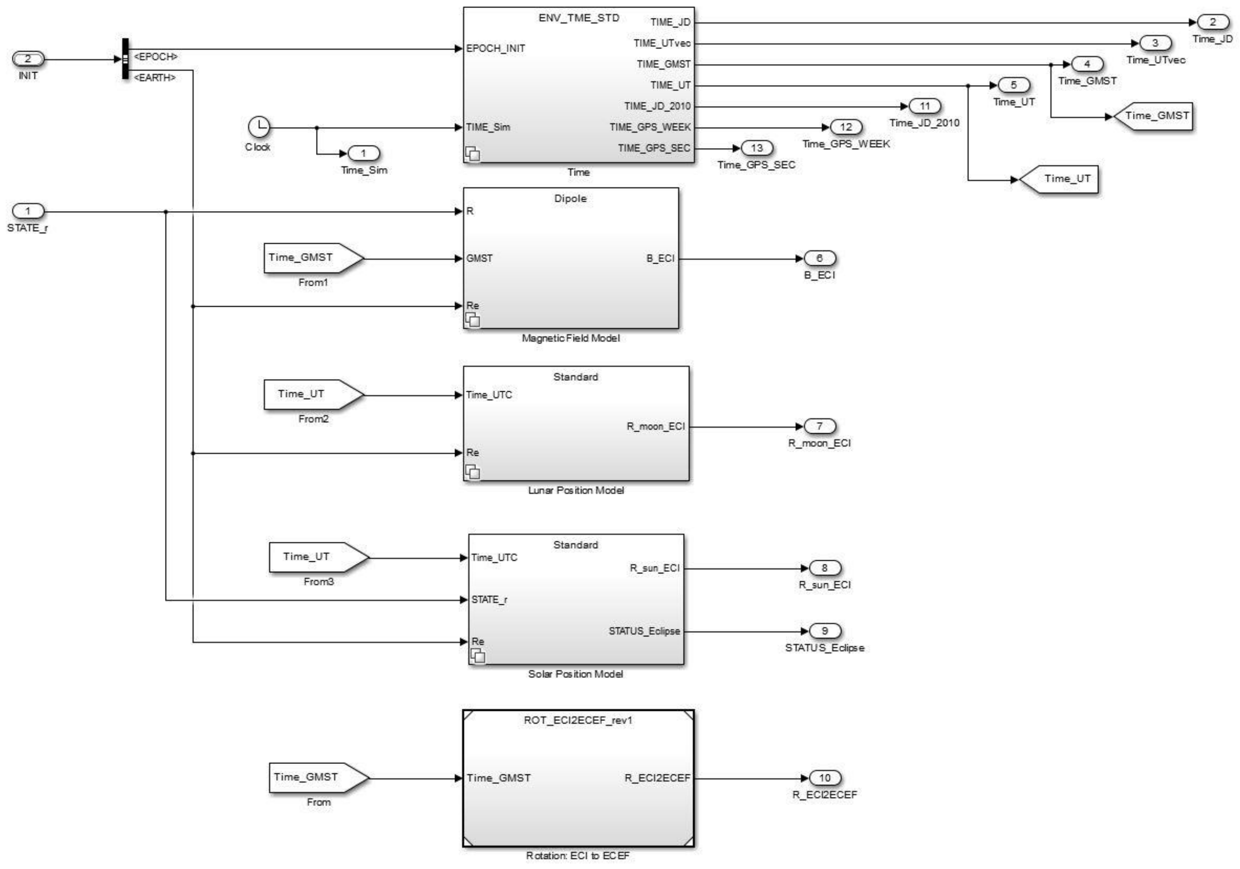


Figure 13: Environment Model Simulink Implementation

4.1.3. Dynamic Modelling

Dynamic modelling of the spacecraft is broken down into three categories; translational dynamics, rotational dynamics, and mass properties. The translational dynamics of the spacecraft are described by the generic two-body problem with the inclusion of imparted forces and perturbing accelerations.

$$\ddot{\boldsymbol{r}} = -\frac{\mu}{\|\boldsymbol{r}\|^3}\boldsymbol{r} + \boldsymbol{a}_{perturbations} + \frac{\boldsymbol{F}_{external}}{m} \tag{1}$$

As will be discussed later, all non-two-body gravitational forces are treated as external perturbations in order to allow for the greatest configurability of the 6DOF model.

Attitude is defined within the simulation via a quaternion of the form:

$$\boldsymbol{q} = \begin{bmatrix} \boldsymbol{\epsilon} \\ \eta \end{bmatrix} \tag{2}$$

The rotational state of the spacecraft is similarly modelled using the basic Euler's rotational equations with the inclusion of external moments.

$$\boldsymbol{J}\dot{\boldsymbol{w}} + \boldsymbol{w} \times \boldsymbol{J}\boldsymbol{w} = \boldsymbol{\tau_c} + \boldsymbol{\tau_d} \tag{3}$$

Where $\boldsymbol{\tau_c}$ are control torques and $\boldsymbol{\tau_d}$ are external disturbances. The kinematics of the orientation of the body frame with respect to the ECI frame is given by:

$$\dot{\boldsymbol{q}} = \frac{1}{2}\underline{\Xi}(\boldsymbol{q})\boldsymbol{\omega} \tag{4}$$

Where:

$$\underline{\Xi}(\boldsymbol{q}) = \begin{bmatrix} \eta\boldsymbol{I} + \boldsymbol{\varepsilon}^{x} \\ -\boldsymbol{\varepsilon}^{T} \end{bmatrix} \tag{5}$$

The attitude matrix defining the rotation from the spacecraft body-frame to the ECI frame is also calculated and stored here using Equation 6.

$$\boldsymbol{R}^{\boldsymbol{BFF2ECI}}(\boldsymbol{q}) = \boldsymbol{\Psi}^{\boldsymbol{T}}(\boldsymbol{q})\Xi(\boldsymbol{q}) \tag{6}$$

Where:

$$\boldsymbol{\Psi}(\boldsymbol{q}) = \begin{bmatrix} \eta\boldsymbol{I} - \boldsymbol{\epsilon}^{\times} \\ -\boldsymbol{\epsilon}^{\boldsymbol{T}} \end{bmatrix} \tag{7}$$

Mass properties are defined to be constant unless a propulsion system is present on the spacecraft, in which case a basic model for the change in mass and inertia for the spacecraft as a function of burn time is included. The implementation of the dynamics modelling subsystem is shown in Figure 14.

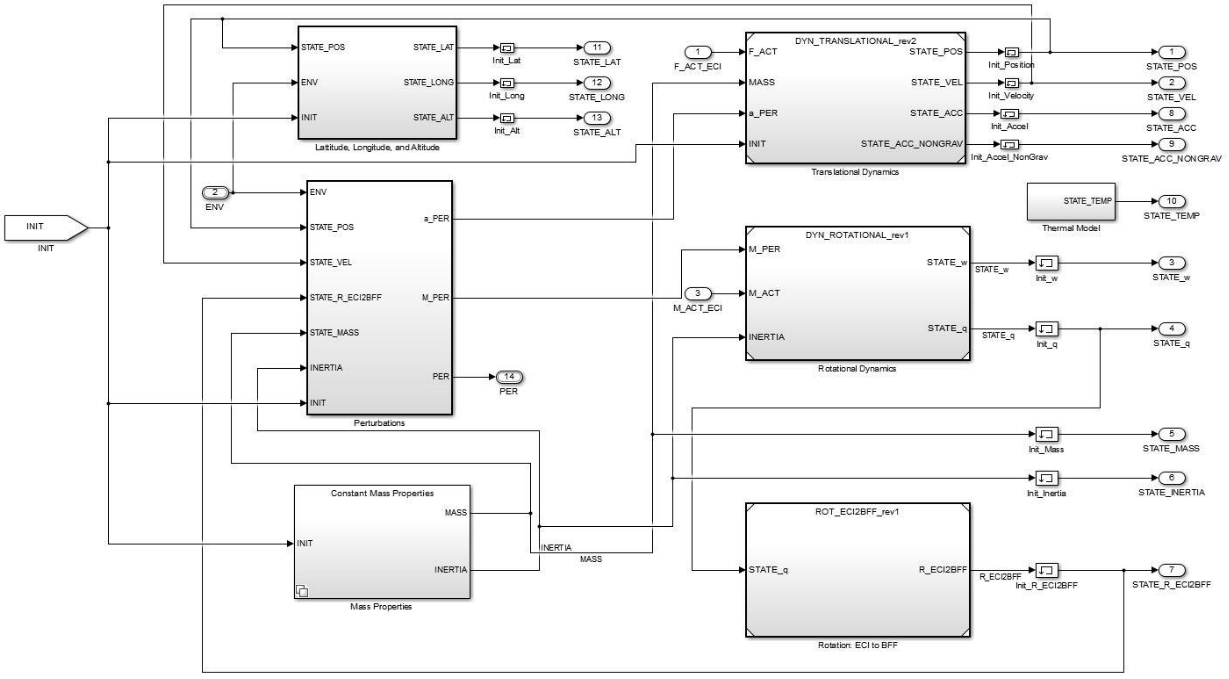


Figure 14: Dynamics Model Implementation

4.1.4. Perturbation Models

Perturbations are considered to be all external processes which create a force or moment on the spacecraft and are dependent only on the spacecraft's state and current environment. Moments and accelerations are calculated based on the user's current desired variant and then fed into the 6DOF dynamics models for integration. All perturbations can either be turned off at run-time by the user or configured to a specific fidelity or specification. To make the addition of new perturbations as simple as possible, it has also been defined that all perturbation models have a resultant acceleration and moment output. If a given perturbation does not require one of the outputs, within the model that output is simply set to zero. For instance, gravity gradient effects produce no translational accelerations on the spacecraft, therefore a non-zero moment would be produced along with a zero magnitude acceleration. The high level implementation of all perturbation models is shown in Figure 15.

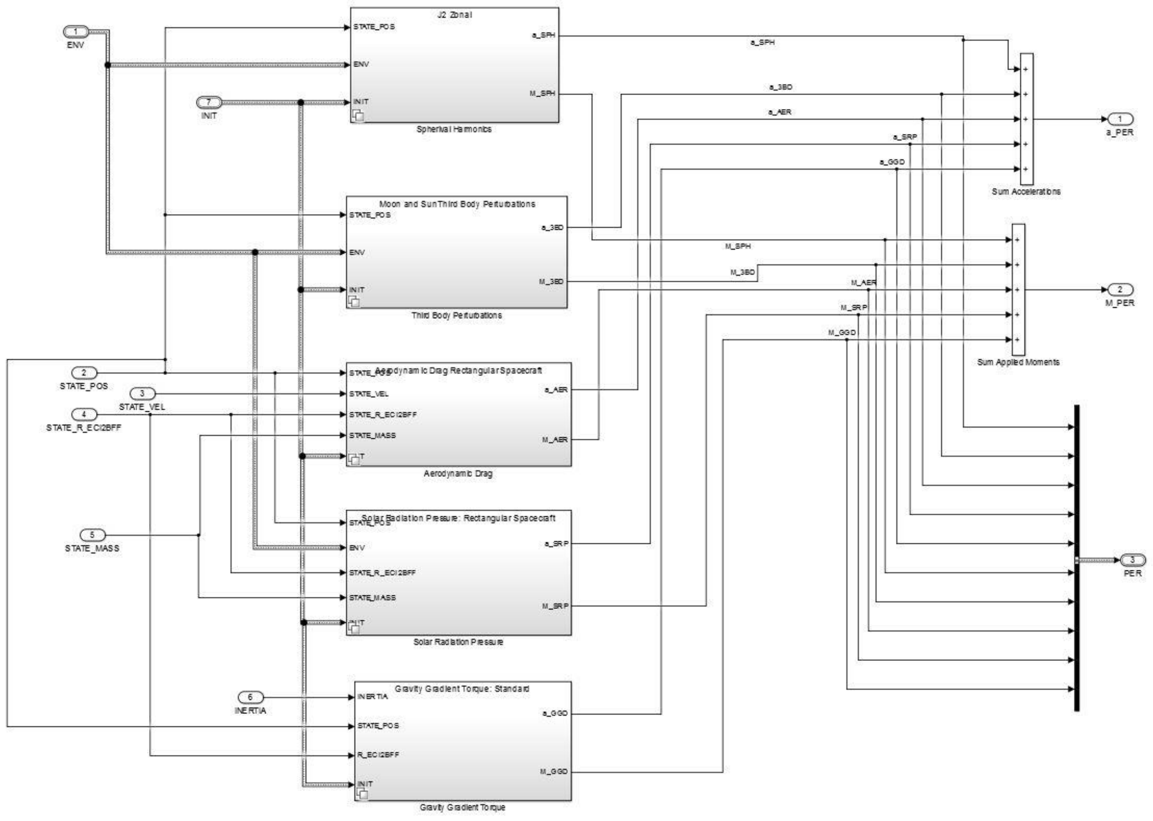


Figure 15: Perturbation Model Implementation

4.1.4.1. Third Body Effects

Currently, the ECI state of the moon and sun are calculated within the environments model. These states are then used in Equation to determine the resulting accelerations.

$$\boldsymbol{a}_{3BD} = \mu_{3BD}\left(\frac{\boldsymbol{R}_{3BD} - \boldsymbol{r}}{\|\boldsymbol{R}_{3BD} - \boldsymbol{r}\|^3} - \frac{\boldsymbol{R}_{3BD}}{\|\boldsymbol{R}_{3BD}\|^3}\right) \tag{8}$$

Where $\mathbf{R}_{3BD}$ is the position of the third body expressed in the ECI frame and μ_{3BD} is the gravitational parameter of the third body.

4.1.4.2. Atmospheric Drag

Two current models for atmospheric drag currently are defined within the pre-existing libraries. The first is a basic model which assumes a spherical spacecraft with a given frontal area A and drag coefficient C_D. Atmospheric density, as given by the environment model, is then used in Equation 9 to determine the translational acceleration on the spacecraft.

$$\boldsymbol{a}_{aero} = \frac{1}{2}\frac{A}{m}C_D\rho(h)(\boldsymbol{v}_{rel}\cdot\boldsymbol{v}_{rel})\hat{\boldsymbol{v}}_{rel} \tag{9}$$

Where

$$\boldsymbol{v}_{rel} = \boldsymbol{v}^{ECI} - \boldsymbol{v}_{atm}^{ECI} \tag{10}$$

A spherical spacecraft model assumes uniform mass distribution and no center of mass, center of pressure offset, therefore the resulting moment is:

$$\boldsymbol{M} = \boldsymbol{0} \tag{11}$$

The second atmospheric drag model assumes a spacecraft with discrete, non-overlapping panels of varying size, drag coefficients, and body-fixed orientation. Nominally, this model is defined with a basic six sided, rectangular spacecraft; however the simulation is configured such that n number of sides are possible. The current model does not account for self-shadowing effects, therefore any defined panels must not overlap. The resulting acceleration equations for a given side of the spacecraft are defined as:

$$\boldsymbol{a}_{aero,i} = -\left(\frac{1}{2}\frac{\rho(h)}{m}\|\boldsymbol{v}_{rel}\|^2\gamma_i A_i C_{D,i}\hat{\boldsymbol{n}}_i\cdot\hat{\boldsymbol{v}}_{rel}\right)\hat{\boldsymbol{v}}_{rel} \tag{12}$$

Where $\rho(h)$ is the atmospheric density at the given altitude, A_i is the surface area, $C_{D,i}$ is the surface drag coefficient, $\hat{\boldsymbol{n}}_i$ is the spacecraft face normal expressed in the ECI frame, and γ_i is defined below:

$$\gamma_i = \begin{cases} 1\ if\ \hat{\boldsymbol{\gamma}}_i\cdot\hat{\boldsymbol{v}}_{rel} > 0 \\ 0\ otherwise \end{cases} \tag{13}$$

Summing over the entire body, the total acceleration due to solar radiation pressure is:

$$\boldsymbol{a}_{aero} = \sum_{i=1}^{k}\boldsymbol{a}_{aero,i} \tag{14}$$

This model does account for imparted moments as a result of any offset of a given panels center of pressure, c_p, from the spacecraft's center of mass, c_m. The moments as a function of the spacecraft's current attitude are given by Equation 15.

$$\boldsymbol{M}_{aero,i} = -\frac{1}{2}\frac{\rho(h)}{m}\|\boldsymbol{v}_{rel}\|^2\gamma_i A_i C_{D,i}(\hat{\boldsymbol{n}}_i \cdot \hat{\boldsymbol{v}}_{rel})^2(\hat{\boldsymbol{v}}_{rel} \times \boldsymbol{r}_{cp,i}) \tag{15}$$

Where $\boldsymbol{r}_{cp,i}$ is the vector from the center of mass to the center of pressure of face *i*. Summing over all faces we find:

$$\boldsymbol{M}_{aero} = \sum_{i=1}^{k} \boldsymbol{M}_{aero,i} \tag{16}$$

4.1.4.3. Solar Radiation Pressure

Similarly to the atmospheric drag modelling, solar radiation pressure effects can be calculated for either a spherical or rectangular spacecraft. For all time, a constant solar radiation pressure of 4.57×10^{-6} W/m^2 is assumed. Accelerations on a spherical spacecraft are defined as:

$$\boldsymbol{a}_{solar} = -\left(\frac{SP}{m}\eta A C_r\right)\hat{\boldsymbol{r}}_{sun} \tag{17}$$

$$\boldsymbol{M} = \boldsymbol{0} \tag{18}$$

Accelerations on a rectangular spacecraft are defined by the equations:

$$\boldsymbol{a}_{solar,i} = -\left(\frac{SP}{m}\eta_i A_i C_{r,i}\hat{\boldsymbol{n}}_i \cdot \hat{\boldsymbol{r}}_{sun}\right)\hat{\boldsymbol{r}}_{sun} \tag{19}$$

Where SP is the mean solar pressure, $\hat{\boldsymbol{n}}_i$ is the spacecraft face normal expressed in the ECI frame, A_i is the surface area, $C_{r,i}$ is the coefficient reflectivity of face *i*, and η_i is defined below:

$$\eta_i = \begin{cases} 1 \; if \; \hat{\boldsymbol{n}}_i \cdot \hat{\boldsymbol{r}}_{sun} > 0 \\ 0 \; otherwise \end{cases} \tag{20}$$

Summing over the entire body, the total acceleration due to solar radiation pressure is:

$$\boldsymbol{a}_{solar} = \sum_{i=1}^{k} \boldsymbol{a}_{solar,i} \tag{21}$$

Torques due are given by:

$$\boldsymbol{M}_{solar,i} = -\left(\frac{SP}{m}\eta_i A_i C_{r,i}(\hat{\boldsymbol{n}}_i \cdot \hat{\boldsymbol{r}}_{sun})^2\right)(\hat{\boldsymbol{r}}_{sun} \times \boldsymbol{r}_{cp,i}) \tag{22}$$

$$\boldsymbol{M}_{solar} = \sum_{i=1}^{k} \boldsymbol{M}_{solar,i} \tag{23}$$

4.1.4.4. Gravity Gradient Torques

Gravity gradient torques are generated by non-uniformities in a spacecraft's inertia. The resulting moment produced by a non-uniform inertia tensor is defined as:

$$\boldsymbol{M}_{gg} = 3\frac{\mu}{\|\boldsymbol{r}\|^3}\hat{\boldsymbol{c}} \times (\boldsymbol{J} \cdot \hat{\boldsymbol{c}}) \tag{24}$$

Where $\hat{\boldsymbol{c}}$ is the unit vector in the nadir direction.

4.2. *Emulation Layer*

4.2.1. Overview

Unlike the simulation layer which is the same for every spacecraft and configured at run-time, the emulation layer is built for a specific spacecraft to reflect the sensors and actuators present on-board. The sensor block within the emulation layer is responsible for interpreting the current state of the spacecraft and producing the relevant sensor readings. The actuator block interprets the commands sent to an emulated sensor and generates the true forces and moments imparted on the spacecraft, which are then fed into the dynamics models.

Although the emulation layer is custom for each spacecraft, it is designed using a plug and play configuration. Libraries of Simulink blocks have been generated for generic sensor and actuator models. These models are all configurable at run-time for the hardware specifications of the desired components. For use they simply need be inserted into the relevant block and connected to the incoming data buses. All blocks have been configured to accept the generic data buses used for tracking environment and state information to make integration as simple as possible. If the user requires a sensor or actuator model not currently defined within the emulation libraries, a generic template and guidelines have been created to simplify the creation and implementation of the new model.

4.2.2. Sensor Models

Sensor models are responsible for taking the current environment and state information of the spacecraft and converting it into the corresponding data reading based on the physics of that sensor. Sensor models are responsible for transforming the data into the appropriate sensor fixed frame (SFF) as well as accounting for any imperfections inherent in the sensor. This includes sensor noise, bias, and misalignments. The corresponding sensor data is then transferred to the interface layer for transmission to the test article.

The high level implementation of several of the major sensor models are described below. Detailed information concerning each model can be seen within the comments of the actual sensor libraries and the indicated reference texts. A sample implementation of the sensor emulation layer can be found in Figure 16.

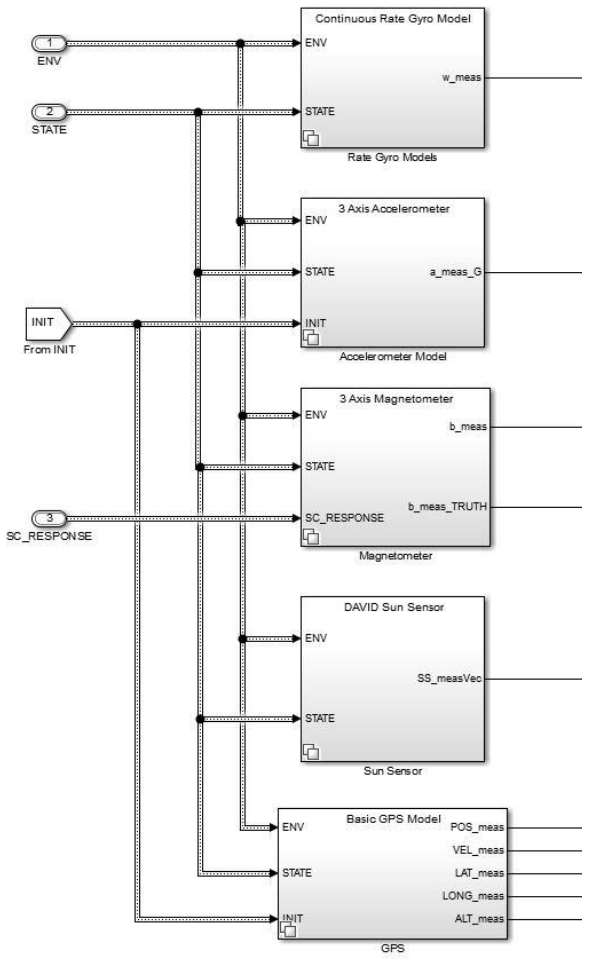


Figure 16: Example Sensor Model Implementation: DaVID CubeSat

4.2.2.1. Magnetometer Models

The primary magnetometer is a basic three-axis measurement configuration of the form:

$$\boldsymbol{b}_{meas}^{SFF} = \boldsymbol{R}^{BFF2SFF}\boldsymbol{R}^{ECI2BFF}\boldsymbol{b}_{true}^{ECI} + \boldsymbol{\eta}_{mag} + \boldsymbol{\beta}_{mag} \tag{25}$$

Where $\boldsymbol{\eta}_{mag} \sim N(0,\sigma^2)$ and $\boldsymbol{\beta}_{mag}$ is the constant measurement bias.

4.2.2.2. Gyro Models

Gyro models currently take two forms, either a single axis gyro measurement, or an integrated three-axis unit. The three axis measurement takes the form:

$$\boldsymbol{w}_{meas}^{SFF} = \boldsymbol{R}^{BFF2SFF}\boldsymbol{R}^{ECI2BFF}\boldsymbol{w}_{true} + \boldsymbol{\eta}_{gyro} + \boldsymbol{\beta}_{gyro} \tag{26}$$

4.2.2.3. Sun Sensor Models

Currently three forms of sun sensor models have been developed and tested. The first is a basic fine sun sensor model which produces a vector in the sensor frame when the sun is in the FOV of the sensor. When the sun is in the FOV and the spacecraft is not in eclipse, the measurement is calculated via:

$$\boldsymbol{s}_{meas} = \boldsymbol{R}^{BFF2SFF}\boldsymbol{R}^{ECI2BFF} * \left(\boldsymbol{r}_{sun}^{ECI} - \boldsymbol{r}_{spacecrat}^{ECI}\right) + \boldsymbol{\eta}_{sensor} \tag{27}$$

A second model which calculates the in plane angle of the sun with respect to two adjacent photo diodes is determined via basic geometry has also been developed. The last model produces a basic analog voltage reading assuming four adjacent photodiodes mounted on the same surface divided into four quadrants. Accurate voltage modeling of this configuration of sensors is highly dependent on test information and best-fit curves provided by the manufacturer. This model was specifically developed to emulate the Wallops Island CRUQS sun sensor. If a similar model is required for emulation for a different sensor, it is recommended to not directly use this model, and rather use it as a template for modification to produce the most accurate model as possible of the desired sensor.

4.2.2.4. Accelerometer Models

Current accelerometer models include either single or three axis variants and measure non-gravitational accelerations on the spacecraft. All existing models assume the units are placed near the center of mass of the spacecraft and do not measure centripetal accelerations caused by rapid rotation of the host craft. Since this framework has been designed for implementation on small spacecraft, it has been assumed that any rigid bodies simulated are not large enough for this effect to be significant. Minor modification would be required if an accelerometer needed to be emulated that was located far from the spacecraft c_m. Measured accelerations are determined by the equation:

$$\boldsymbol{a}_{meas} = \boldsymbol{R}^{BFF2SFF}\boldsymbol{R}^{ECI2BFF}\left(\boldsymbol{a}_{spacecraft} - \boldsymbol{a}_{gravitational}\right) + \boldsymbol{\eta}_{meas} \tag{28}$$

4.2.2.5. Inertial Measurement Units

Modern inertial measurement units (IMUs) for small spacecraft are available in many configurations that include three axis rate measurements, three axis acceleration measurements, and some units currently included integrated magnetometers as well. Since IMUs come in a wide range of configurations, it was decided to not generate standalone IMU models, rather when it is necessary to model an IMU, framework best-practice dictates that relevant gyroscope and accelerometer models are selected and independently inserted into the framework. The sensor data bus can then be used to track the measurements as integrated IMU data.

4.2.2.6. Global Positioning System

A basic model for a Global Positioning System (GPS) received has been developed and implemented. This model takes the current inertial translational state of the spacecraft, converts it into the ECEF coordinate frame and generates a GPS time stamped state vector. The model also has the ability to generate outputs in other coordinate frames or to generate other simulated data, depending on the specification of the emulated GPS.

$$\boldsymbol{r}_{meas}^{ECEF} = \boldsymbol{R}^{ECI2ECEF}\boldsymbol{r}_{spacecraft}^{ECI} + \boldsymbol{\eta}_{meas} \tag{29}$$

$$\boldsymbol{v}_{meas}^{ECEF} = \boldsymbol{R}^{ECI2ECEF}\left(\boldsymbol{v}_{spacecraft}^{ECI} - \boldsymbol{w} \times \boldsymbol{r}_{spacecraft}^{ECI}\right) + \boldsymbol{\upsilon}_{meas} \tag{30}$$

4.2.2.7. Star Trackers

A basic star tracker model has been implemented such that a noisy attitude quaternion measurement is generated; pending the view of the star tracker is not impeded by the Earth. The resulting quaternion represents the estimated attitude of the sensor frame with respect to the ECI frame.

$$\boldsymbol{q}_{meas} = \boldsymbol{q}_{spacecraft} + \boldsymbol{\eta}_{meas} \tag{31}$$

4.2.3. Actuator Models

Actuator models are responsible for taking commands generated by the test article and generating the resulting forces and moments on the spacecraft based on physics modeling of the given component. Since many actuators take on slightly different forms and accept varying command inputs, modification may be required to the pre-existing libraries to account for any variations. For instance, the current three axis reaction wheel model expects a torque input while others may require a wheel speed. If this is the case, the existing model will provide a foundation for any variants that need to be implemented.

4.2.3.1. Reaction Wheel

The current reaction wheel implementation assumes a three wheel configuration, orthogonally aligned with the primary spacecraft axis. A basic internal control law, accounting for viscous and coulomb friction is implemented to realize commanded torques. Also wheel saturation is tracked and accounted for in the output dynamics. Should a reaction wheel in a different configuration be required, the model must be implemented accordingly. However it can be derived from the same basic relationships as shown below.

$$\boldsymbol{H} = \boldsymbol{J\omega} + \boldsymbol{h} \tag{32}$$

Where h is the contribution of angular momentum from the reaction wheel. Assuming no external disturbances, taking the time derivative of the angular momentum results in:

$$\dot{\boldsymbol{H}} = \boldsymbol{J\dot{\omega}} + \dot{\boldsymbol{h}} + \boldsymbol{\omega} \times (\boldsymbol{J\omega} + \boldsymbol{h}) = \boldsymbol{0} \tag{33}$$

$$\boldsymbol{J\dot{\omega}} + \boldsymbol{\omega} \times \boldsymbol{J\omega} = -\dot{\boldsymbol{h}} - \boldsymbol{\omega} \times \boldsymbol{h} \tag{34}$$

Therefore, the control torque of a reaction wheel is given by:

$$\boldsymbol{\tau}_c = -\dot{\boldsymbol{h}} - \boldsymbol{\omega} \times \boldsymbol{h} \tag{35}$$

Where:

$$\boldsymbol{h} = \boldsymbol{A\Omega} \tag{36}$$

Where $\boldsymbol{A}$ is the inertia tensor of the wheel assembly expressed in the body frame and $\boldsymbol{\Omega}$ are the wheel rates, expressed in the body frame.

4.2.3.2. Magnetic Torque Rods

The magnetic torque rod emulator block simply takes in a commanded magnetic dipole in the body frame and direction to generate a resulting magnetic torque on the spacecraft. Misalignments of the torque rods are also taken into account.

$$\boldsymbol{\tau}_{TR} = \left(\boldsymbol{R}_{misalign}\boldsymbol{m}_{cmd}\right) \times \boldsymbol{b}_{ECI}^{BFF} \tag{37}$$

4.3. *Interface Layer*

4.3.1. Overview

The interface layer is the primary mechanism for SoftSim6D's successful integration with a test article. Careful consideration in development and maintenance of the interface layer will be the driving factor in simulating realistic hardware. In the emulation model, at every time-step of the simulation, sensor measurements are generated based upon the specifications of the desired sensor and stored as the corresponding double prevision vector, voltage, etc. When testing MATLAB or Simulink algorithms, this result is sufficient to be fed into the test article as at this level as primary emphasis will be on the design of the algorithms themselves and not on the hardware implementation. However, when dealing with HWITL testing, careful consideration must be made to ensure a realistic outcome. In order to prepare data for transmission to hardware, three additional layers are considered; data preparation, communication management, and data transmission.

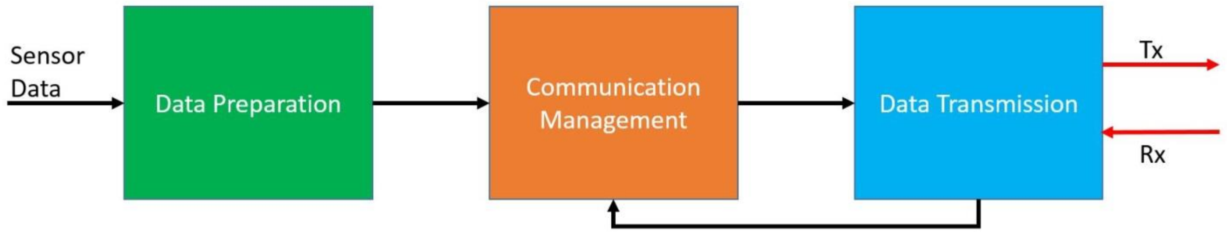


Figure 17: Sensor Interface Layer Work Flow

In many scenarios, unless the exact component has previously been modelled and verified for HWITL testing, the three layers will need to be developed for a given component. Generic frameworks for all three layers have been developed to ease this process and can be readily adapted to fit new hardware. The following sections will provide further explanation on the purposes of each layer and the sections of each layer will be custom for each component. Examples using primarily sensor models will be discussed, although implementation for actuator models undergo the exact same process, except in reverse order.

4.3.2. Data Preparation Layer

Preparing data for transmission is the simplest of the three required steps for HWITL testing, however a minor error at this phase can completely invalidate any transmitted data so care must be taken in defining this block for a given component. There are three primary processes handled by the Data Preparation Layer (DPL). The first is unit conversion. Each sensor model will output data in a given unit and base. For instance, the three axis magnetometer model gives a default output in Teslas (T). The component you are simulating may actually produce readings measured in Gauss (G) or perhaps nano-Teslas (nT). To minimize modification to emulation layer libraries, any necessary conversion is done in the DPL.

Many sensors fundamentally are built of analog electronics with analog to digital converters (ADC) used to capture the analog data before transmission over serial protocols. This process of converting data from an analog to digital format results in the discretization of the sensor data over finite intervals, based on the resolution of the ADC. The ADC will produce an integer digital number which requires an additional conversion to be a useful numeric measurement. Typically, for ease of implementation in serial communication, a sensor will transmit this integer number and leave conversion to the necessary decimal number and units to the user. This relationship is defined as a measure of units per Least Significant Bits (LSB). For instance, a rate gyro may have a conversion of 0.1 °/s/LSB. If a reading were to be used in an attitude control algorithm, an example reading of 22 would be converted as follows:

$$22 * \frac{0.1\,°/s}{LSB} = 2.2\,°/s \tag{38}$$

The DPL simulates the sensor, by reversing this process and discretizing the reading and converting it to the corresponding integer. This step is not necessary in all sensors as some may directly report floating or double precision numbers. This will be the case in more advance sensors that typically contain some form of internal microprocessor.

The last process in the DPL converts the measurement into the necessary precision. This involves converting the double precision number that has been used thus far in Simulink into the precision dictated by the hardware's Software Interface Control Document (SICD). An example Simulink implementation of the DPL can be shown in Figure 18.

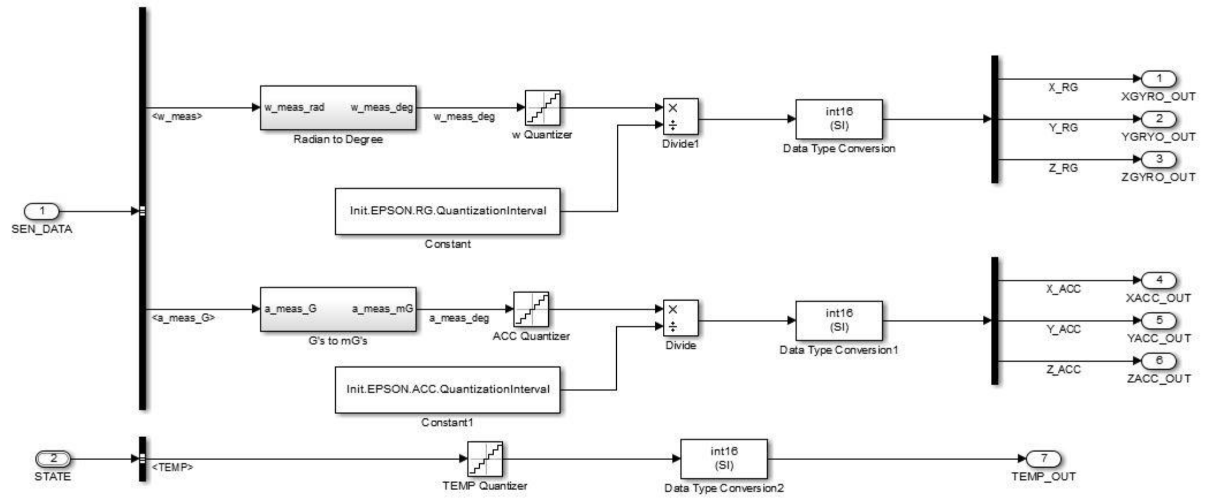


Figure 18: Data Preparation Layer Example: EPSON MG350 IMU

4.3.3. Communication Management Layer

The Communication Management Layer (CML) is the most complicated, yet essential, component of the interface layer. The CML is responsible for monitoring all incoming data test article for valid commands, retrieving the necessary data from the DPL, and then packaging the data into a valid format for transmission over the corresponding serial protocol. Two primary modules make up the CML; the Data Buffer Module (DBM) and the Command Processing Module (CPM). An example CML is shown in Figure 19.

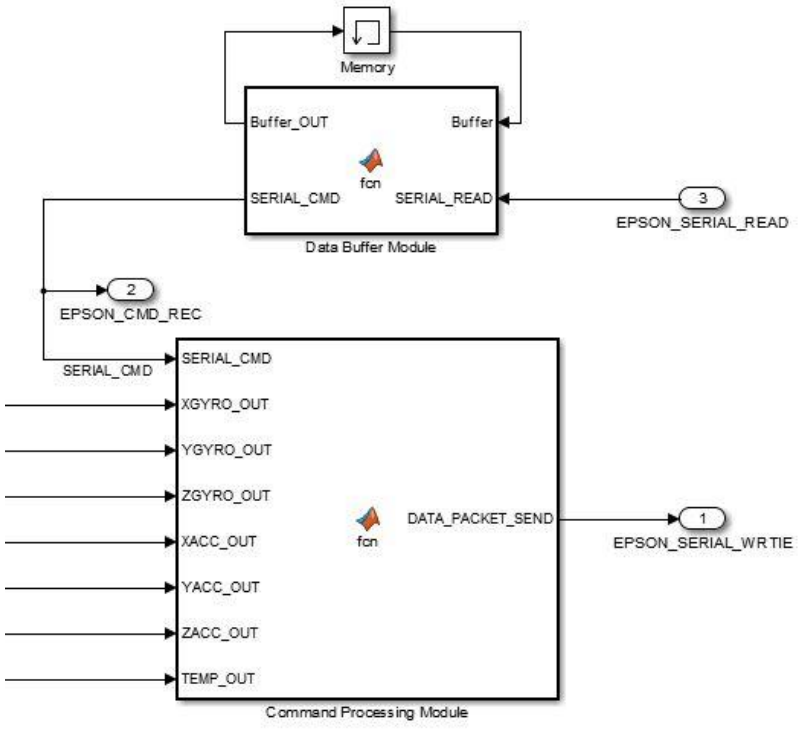
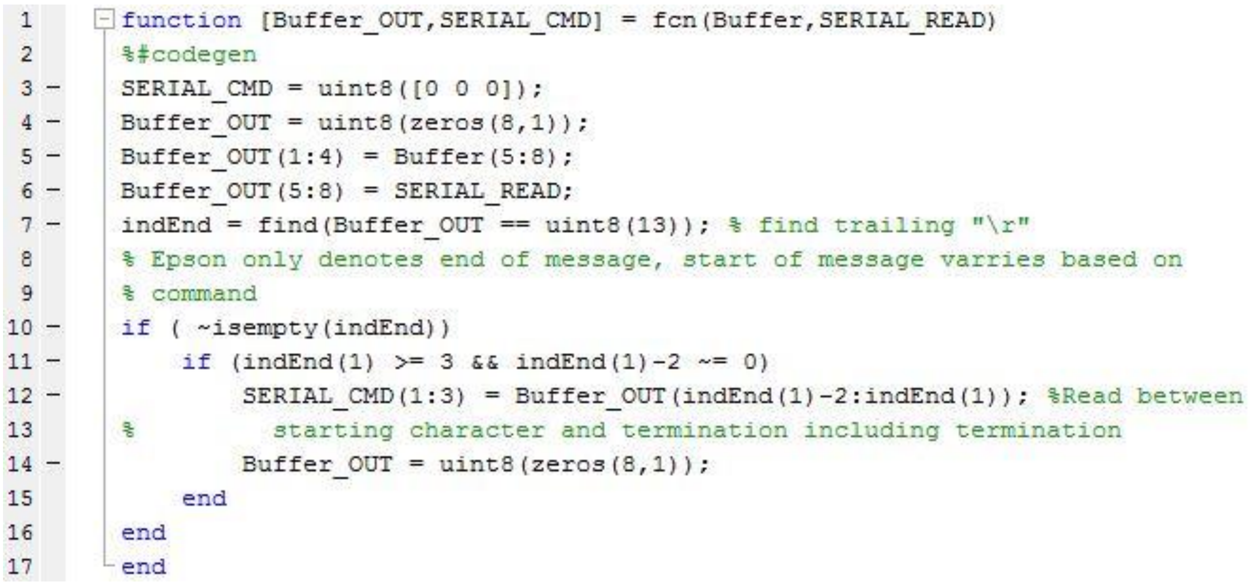


Figure 19: Communication Management Layer Example: EPSON MG350 IMU

4.3.3.1. Data Buffer Module

The IO cards used to communicate with the flight avionics board maintain FIFO buffers which hold all incoming data until they are read and emptied by the target computer. At each finite iteration of the simulation, the Data Buffer Module, reads and stores the information from this buffer. As the DBM operates in discrete time, the FIFO buffer ensures that no data is lost between iterations of the simulation. Since the majority of simulated components communicate asynchronously, there is no guarantee that a communication packet will be received during the exact window of a FIFO read operation by the DBM. It is for this reason that the DBM stores data from time adjacent FIFO read operations and combines them into a complete packet. Once the DBM has confirmed an entire packet has been formed, it will pass on the packet to the CPM, as shown in Figure 20.



```
function [Buffer_OUT,SERIAL_CMD] = fcn(Buffer,SERIAL_READ)
%#codegen
SERIAL_CMD = uint8([0 0 0]);
Buffer_OUT = uint8(zeros(8,1));
Buffer_OUT(1:4) = Buffer(5:8);
Buffer_OUT(5:8) = SERIAL_READ;
indEnd = find(Buffer_OUT == uint8(13)); % find trailing "\r"
% Epson only denotes end of message, start of message varries based on
% command
if ( ~isempty(indEnd))
    if (indEnd(1) >= 3 && indEnd(1)-2 ~= 0)
        SERIAL_CMD(1:3) = Buffer_OUT(indEnd(1)-2:indEnd(1)); %Read between
%          starting character and termination including termination
        Buffer_OUT = uint8(zeros(8,1));
    end
end
end
```

Figure 20: Data Buffer Module Example: EPSON MG350 IMU

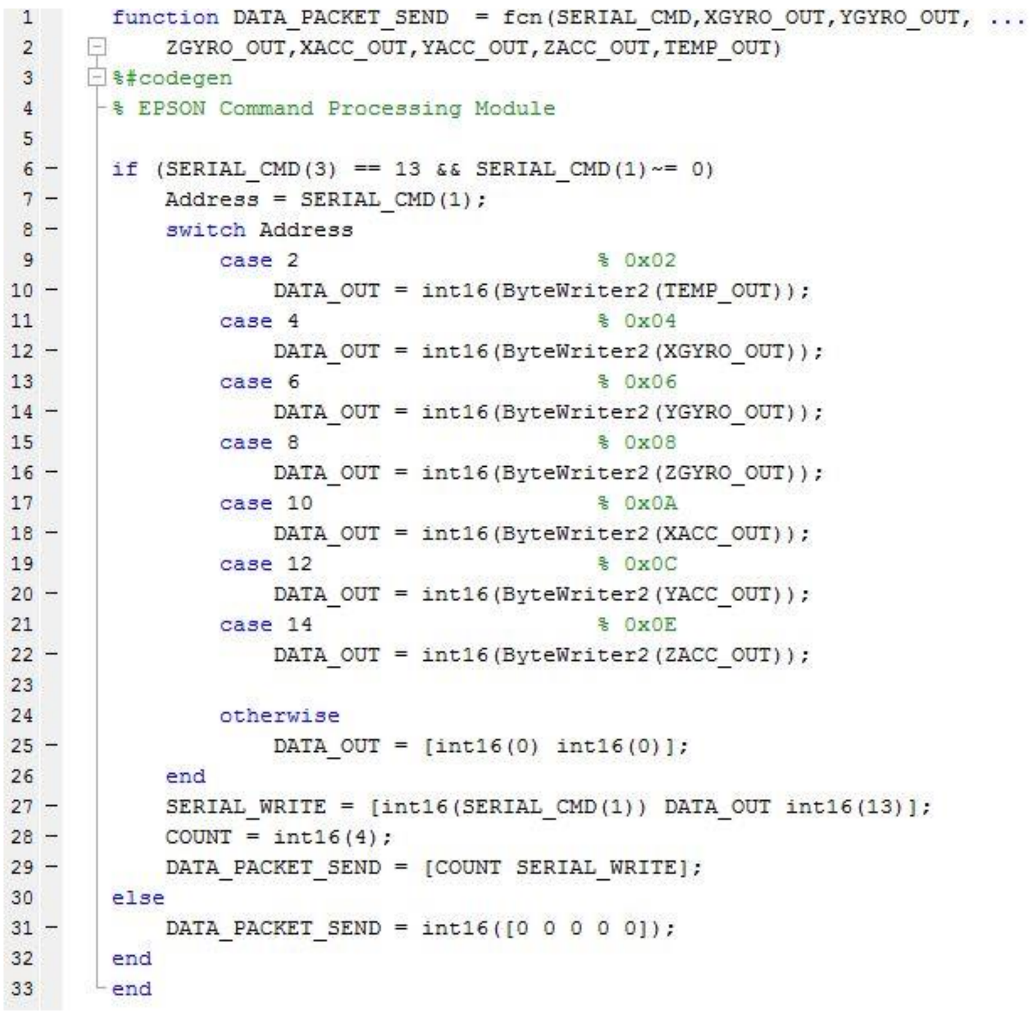
This process ensures no dropped transmissions, however it does present the possibility for inadvertently inserting latency into the system. For example, if an avionics board is designed to sample a sensor at 1 Hz, but the simulation is only running at 1 Hz, the data request packet has the potential to be read over two time steps by the target computer. This would result in a one second delay in response, which is undesirable. It is for this reason that all HWITL simulations be run at least 10 times faster than the maximum sampling rate of an avionics board. The minimum sample for any HWITL simulation is recommended to be at least 100 Hz.

4.3.3.2. Communication Processing Module

Once a valid packet has been confirmed by the DBM, the Communication Processing Module (CPM) will decode the packet to determine the specific command that has been set. The CPM is able to simulate a wide range of activities for a given component. For instance, besides requesting data, a command packet received from the test article can request the change of a setting such as component ID number or gain. The CPM has the ability to recognize that, and change the resulting setting, pending the Interface Layer has been properly designed to account for said command. A common component command is to change the devices address or component ID. This is especially the case when multiple copies of a single component are connected to a single avionics board. The CPM has the ability to interpret that command, change the address, acknowledge the command, and have that change persist throughout the rest of the simulation.

Although a generic CPM model has been created for use in generating specific component models, the CPM requires a substantial knowledge of the software interface of the desired component in order to be adequately implemented. It is desirable that for any component being emulated a SICD be obtained from the manufacturer to ease the process of development. Typically SICD's are written to enable a user to write software capable of commanding and receiving/interpreting data from a device. The development of a component specific CPM will require the opposite of that. Generating a CPM will require the reverse engineering of a SICD such that the generated software accurately simulates the role of the component being commanded by an external processor. During the development process of the CPM it is highly encouraged to have a test article available that is designed to command the desired component to confirm proper serial communication. This can be as simple as a basic microprocessor such as an Arduino or Tiva C microcontroller configured to sample data at a set frequency. Having such an article available during the development and verification process of the simulation will drastically decrease set-up time required once the actual flight test article has arrived.

The intricacies of development for a specific CPM will vary by component. For example purposes, the design, implementation, and testing of the CPM for an EPSON IMU MG350 will be described in detail. Figure 21 shows an example of the CPM for this IMU. The EPSON CPM takes in the input SERIAL_CMD, which is the full command packet received by the target computer and verified by the DBM. The command packets to read from the EPSON unit are short three byte sequences containing a leading header byte, a command ID byte, and a termination byte. The CPM first checks for valid header and termination bytes. Once confirmed, the CPM will then determine which data the received command byte corresponds to. The desired information will then be passed into a custom MATLAB function known as ByteWriter2.m. This function internally converts the input value into a 16 bit signed binary number and then into a 2 byte hexadecimal number. The hexadecimal number is then converted into a MATLAB int16 class for use in data transmission.



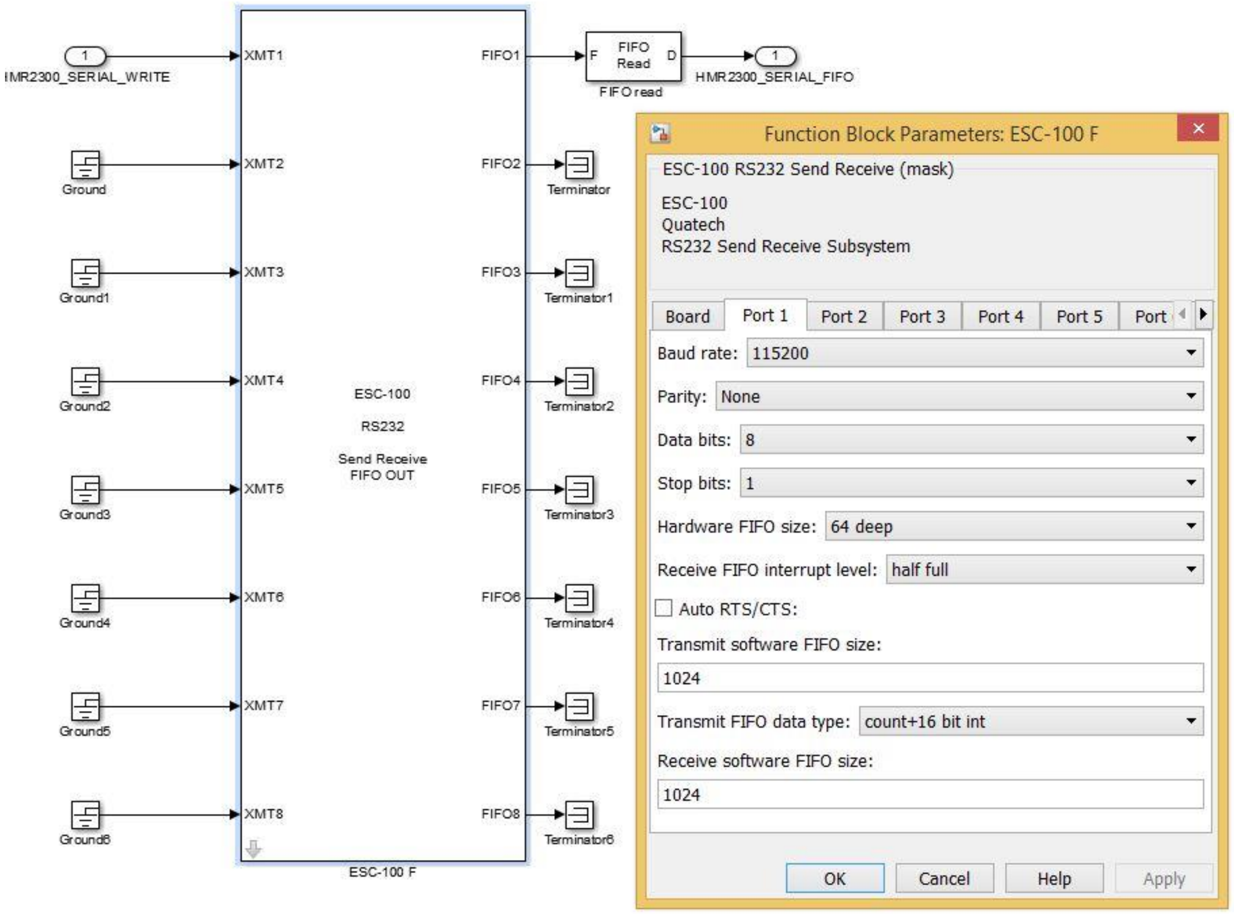
```
function DATA_PACKET_SEND  = fcn(SERIAL_CMD,XGYRO_OUT,YGYRO_OUT, ...
    ZGYRO_OUT,XACC_OUT,YACC_OUT,ZACC_OUT,TEMP_OUT)
%#codegen
% EPSON Command Processing Module

if (SERIAL_CMD(3) == 13 && SERIAL_CMD(1)~= 0)
    Address = SERIAL_CMD(1);
    switch Address
        case 2                          % 0x02
            DATA_OUT = int16(ByteWriter2(TEMP_OUT));
        case 4                          % 0x04
            DATA_OUT = int16(ByteWriter2(XGYRO_OUT));
        case 6                          % 0x06
            DATA_OUT = int16(ByteWriter2(YGYRO_OUT));
        case 8                          % 0x08
            DATA_OUT = int16(ByteWriter2(ZGYRO_OUT));
        case 10                         % 0x0A
            DATA_OUT = int16(ByteWriter2(XACC_OUT));
        case 12                         % 0x0C
            DATA_OUT = int16(ByteWriter2(YACC_OUT));
        case 14                         % 0x0E
            DATA_OUT = int16(ByteWriter2(ZACC_OUT));

        otherwise
            DATA_OUT = [int16(0) int16(0)];
    end
    SERIAL_WRITE = [int16(SERIAL_CMD(1)) DATA_OUT int16(13)];
    COUNT = int16(4);
    DATA_PACKET_SEND = [COUNT SERIAL_WRITE];
else
    DATA_PACKET_SEND = int16([0 0 0 0 0]);
end
end
```

Figure 21: Command Processing Module Example: EPSON MG350 IMU

It is important to note that although MATLAB int16 technically corresponds to a 16 bit signed integer, transmitting a single16 bit signed integer will not result in the transmission of the correct two byte number. This is due to both the design of MATLAB serial drivers as well as how MATLAB defines a 16 bit signed integer at the binary level (which is of primary concern for serial communication). In general, it is strongly recommended that for transmission of any data type using Simulink Real-Time, data integrity must be performed at the binary level in order to ensure accurate communication. This is also important because different implementations may use different byte orderings than considered convention. For example one sensor may send two bytes of data with the Most Significant Byte (MSB) first while others may send the Least Significant Byte (LSB). This is even more the case in communication requiring floating or double point precision.

4.3.4. Data Transmission Layer

The Data Transmission Layer (DTL) is responsible for all actual communication with the test hardware. The DTL reads the FIFO buffer of the corresponding data port and IO card in the target machine and then transforms it into a numeric format that can be utilized by the rest of the simulation. It primarily consists of what are known as Read Write Simulink blocks. These are custom Simulink IO blocks that come with supported IO cards. They contain all of the driver code necessary to operate the IO card in the target computer. Although the exact layout of the Block Parameters varies by card and manufacturer, all allow for configuration of major requirements for serial communication such as baud rate, parity, stop bits, FIFO size, etc. Analog to Digital Acquisition System (ADACS) cards provide the user options on voltage limits, sample times, and precision. An example DTL for the Honeywell HMR2300 magnetometer in conjunction with the Read Write block for the Quatech ESC-100 IO Card is shown in Figure 22.



ESC-100 RS232 Send Receive (mask)

ESC-100
Quatech
RS232 Send Receive Subsystem

Baud rate: 115200
Parity: None
Data bits: 8
Stop bits: 1
Hardware FIFO size: 64 deep
Receive FIFO interrupt level: half full
☐ Auto RTS/CTS:
Transmit software FIFO size: 1024
Transmit FIFO data type: count+16 bit int
Receive software FIFO size: 1024

Figure 22: Data Transmission Layer Example: Honeywell HMR2300

4.3.5. MATLAB and Simulink Algorithm Testing

In addition to real-time HWITL testing, SoftSim6D has the capability to be used for both real-time and accelerated testing of MATLAB/Simulink algorithms. For a given spacecraft, once the Simulation Layer and Emulation Layer have been defined, they do not need to be changed, regardless of algorithm or HWITL testing. The only alterations that are required are in the Interface Layer. For algorithm testing, the complexity of the Interface Layer is drastically reduced. For the highest fidelity testing, it is recommended that the user generate a DPL to account for potential issues caused by data discretization, although this is not required. For testing the user will then insert their MATLAB/Simulink algorithm directly into the Interface Layer via a Model Reference block and connect to the corresponding inputs. It is also important to note that in order to take advantage of SoftSim6D, the target algorithms need to be designed in accordance with requirements consistent with Simulink Coder. A sample work flow for both sensor and actuator IO using MATLAB/Simulink algorithms as the test article is shown in Figure 23.

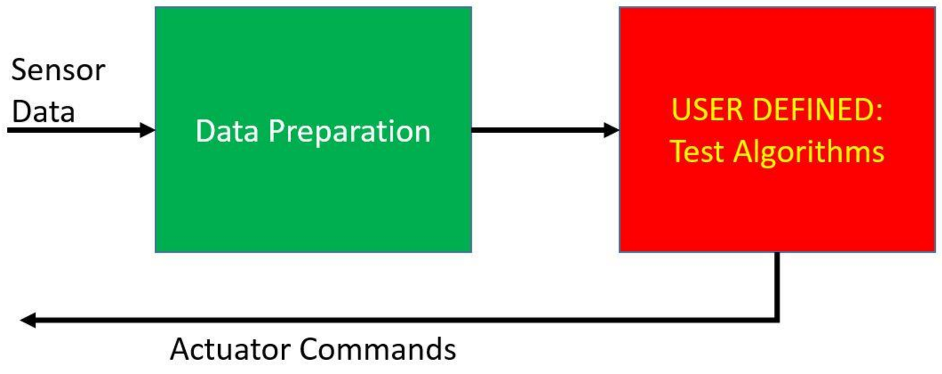


Figure 23: Algorithms Only Interface Layer Work Flow

5. Integrated Base Level Environment

5.1. *Overview*

With the development of the individual simulation layers in Chapter 4, the groundwork was successfully laid for a full simulation containing fully functioning integrated plant models, component emulation, and serial communication. When it is desired to configure a simulation for a new spacecraft or scenario, an Integrated Base Level Environment (IBLE) has been developed to act as a framework to build upon. Figure 24 shows a high level view of the IBLE prior to configuration by the user. The IBLE has two major components already implemented; the simulation layer, denoted by the red environment, dynamics, and INIT blocks as well as the emulation layer, denoted by the cyan sensor and actuator blocks. The simulation later within the IBLE is 100% configurable from initialization files by the user, given that no new functionality (such as new perturbation model) is required. The emulation layer will require manipulation by the user in order to insert relevant component models into the simulation, however extensive libraries for standard components have been developed to facilitate this process and ensure the plug and play usability of the framework. This chapter will discuss the configuration of a new simulation using the IBLE as well as the custom utilities that have been developed to support all forms of testing using SoftSim6D.

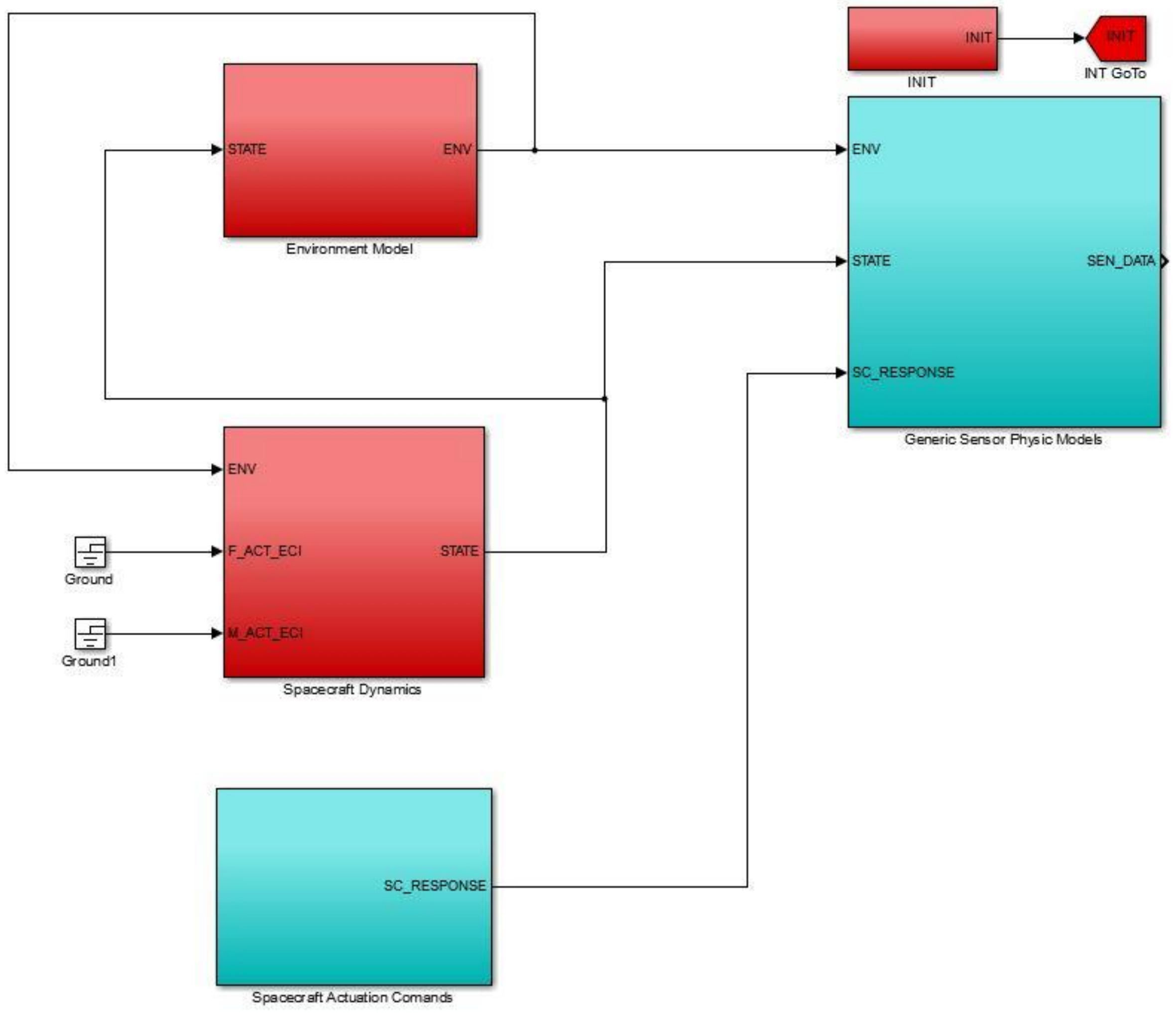


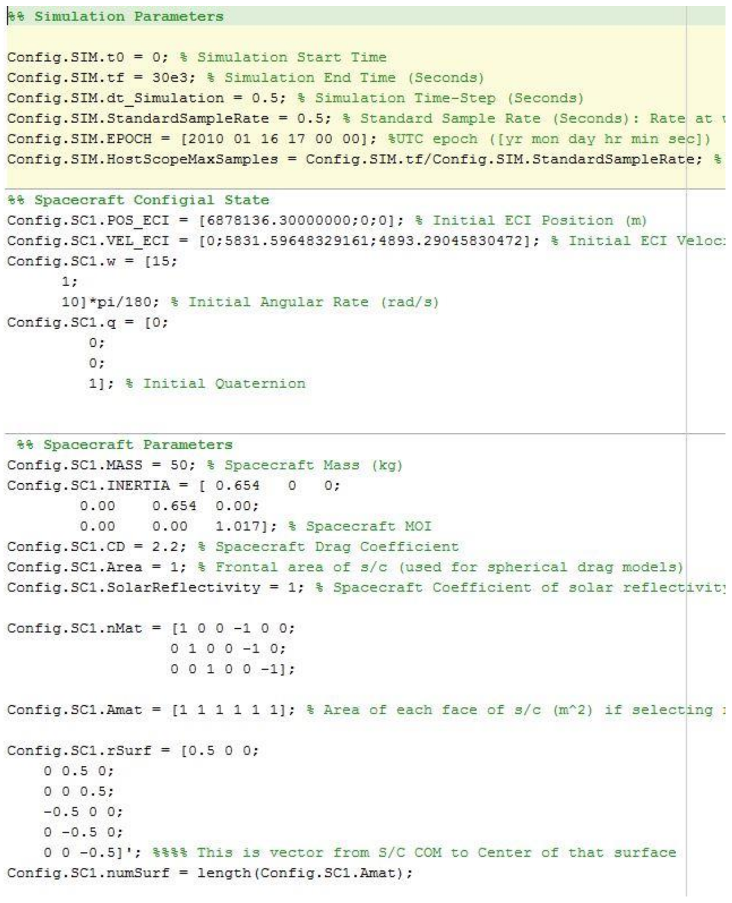
Figure 24: Standard IBLE for Single Spacecraft Mission Development

5.2. *Configuring a Simulation*

5.2.1. Simulation Layer

The Simulation Layer has been designed and implemented such that no manual user manipulation of the SoftSim6D Simulink model is required for configuration. As previously discussed, everything required to run a high fidelity 6DOF simulation is contained within this layer. To configure all parameters and select the desired model variants only two files are required to be edited; the spacecraft and simulation initialization file, LOAD_Init.m, and the variant definition file, LOAD_ModelVariantDefinition.m.

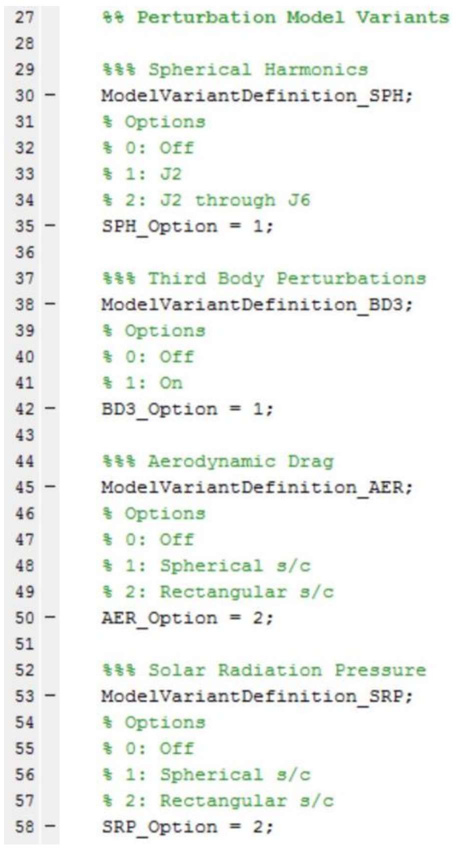
The spacecraft and simulation initialization file (SSIF) is the primary means of configuring the physical properties and state of the spacecraft as well as the parameters for the simulation. Spacecraft mass properties, areas, surface properties coefficients, etc are all defined here. The file contains detailed comments to define necessary units and what values may be ignored (if certain variants are not selected for use). The initial inertial translation and rotational states are also defined. Lastly basic simulation parameters are also tunable to the user's preference. This includes simulation length, Epoch, time step, and scope sample rate. This structure, shown in Figure 25, has been designed such any changes in a simulation's SSIF, at compilation and execution, will automatically be reflected in the test. As will be discussed in SoftSim6D's custom support utilities, anytime the simulation has already been compiled and loaded onto the target machine, many of the parameters found in the SSIF can be altered in real-time or between simulations without re-compiling.



```
%% Simulation Parameters

Config.SIM.t0 = 0; % Simulation Start Time
Config.SIM.tf = 30e3; % Simulation End Time (Seconds)
Config.SIM.dt_Simulation = 0.5; % Simulation Time-Step (Seconds)
Config.SIM.StandardSampleRate = 0.5; % Standard Sample Rate (Seconds): Rate at 
Config.SIM.EPOCH = [2010 01 16 17 00 00]; %UTC epoch ([yr mon day hr min sec])
Config.SIM.HostScopeMaxSamples = Config.SIM.tf/Config.SIM.StandardSampleRate; %

%% Spacecraft Configial State
Config.SC1.POS_ECI = [6878136.30000000;0;0]; % Initial ECI Position (m)
Config.SC1.VEL_ECI = [0;5831.59648329161;4893.29045830472]; % Initial ECI Veloc
Config.SC1.w = [15;
      1;
      10]*pi/180; % Initial Angular Rate (rad/s)
Config.SC1.q = [0;
         0;
         0;
         1]; % Initial Quaternion


 %% Spacecraft Parameters
Config.SC1.MASS = 50; % Spacecraft Mass (kg)
Config.SC1.INERTIA = [ 0.654   0   0;
        0.00    0.654  0.00;
        0.00    0.00   1.017]; % Spacecraft MOI
Config.SC1.CD = 2.2; % Spacecraft Drag Coefficient
Config.SC1.Area = 1; % Frontal area of s/c (used for spherical drag models)
Config.SC1.SolarReflectivity = 1; % Spacecraft Coefficient of solar reflectivit

Config.SC1.nMat = [1 0 0 -1 0 0;
                  0 1 0 0 -1 0;
                  0 0 1 0 0 -1];

Config.SC1.Amat = [1 1 1 1 1 1]; % Area of each face of s/c (m^2) if selecting 

Config.SC1.rSurf = [0.5 0 0;
    0 0.5 0;
    0 0 0.5;
    -0.5 0 0;
    0 -0.5 0;
    0 0 -0.5]'; %%%% This is vector from S/C COM to Center of that surface
Config.SC1.numSurf = length(Config.SC1.Amat);
```

Figure 25: Spacecraft and Simulation Initialization File

The Variant Definition File (VDF) provides the user with the capability to control all variant selections from a single location. Individual model variant definition files are defined for each variant subsystem in the corresponding directories where the models are stored. These files automatically create the necessary Simulink Model Variant objects to define a variant subsystem when the high level model is loaded. The VDF then calls these pre-defined objects and sets the integer logic flag associated with the user's desired model choice. The primary use of the VDF is to select change or turn off various perturbation models. The VDF can also be utilized to alter sensor variants if the scenario simulation is configured to do so. An example section of the VDF can be seen in Figure 26.



```
%% Perturbation Model Variants

%%% Spherical Harmonics
ModelVariantDefinition_SPH;
% Options
% 0: Off
% 1: J2
% 2: J2 through J6
SPH_Option = 1;

%%% Third Body Perturbations
ModelVariantDefinition_BD3;
% Options
% 0: Off
% 1: On
BD3_Option = 1;

%%% Aerodynamic Drag
ModelVariantDefinition_AER;
% Options
% 0: Off
% 1: Spherical s/c
% 2: Rectangular s/c
AER_Option = 2;

%%% Solar Radiation Pressure
ModelVariantDefinition_SRP;
% Options
% 0: Off
% 1: Spherical s/c
% 2: Rectangular s/c
SRP_Option = 2;
```

Figure 26: Variant Definition File

5.3. *Support Software*

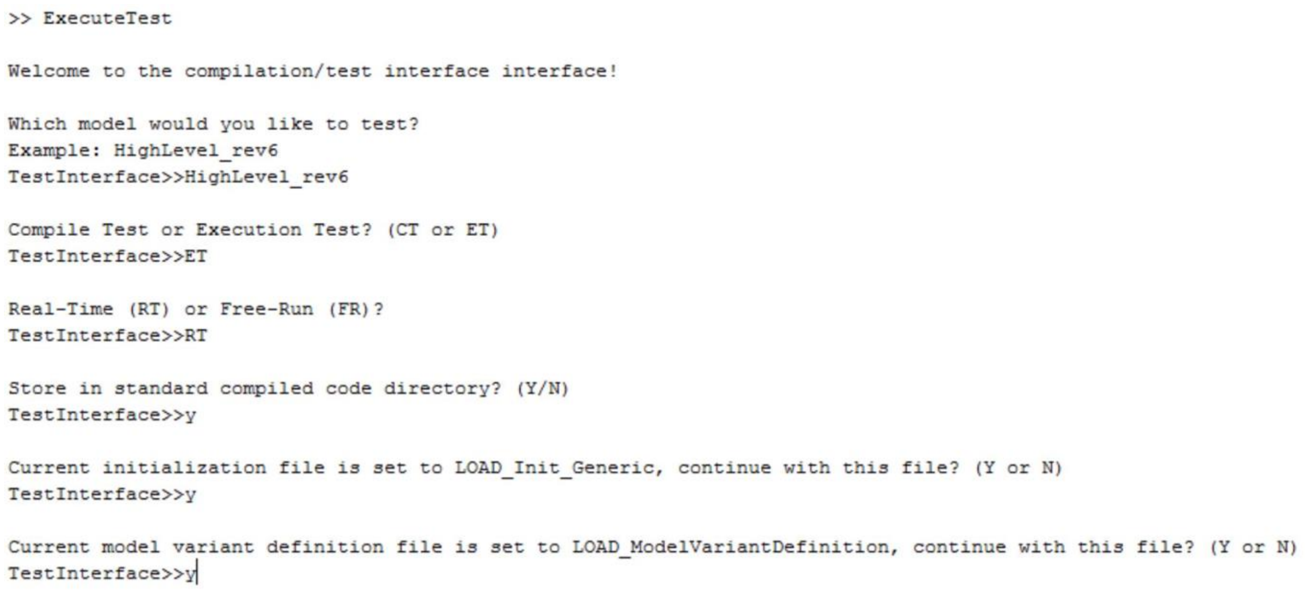
Numerous custom MATLAB utilities have been developed to aid in simulation verification, execution, real-time management, and data post-processing. Several of the major utilities that can support all forms of testing will be discussed.

5.3.1. ExecuteTest

To ensure proper compilation and execution of a simulation, multiple initialization files are required to ensure all updated parameters and settings are reflected as well as to prepare the necessary objects, scopes, and data structures that drastically simplify the process of real-time tuning and data post-processing. To aid in this process, a generic MATLAB utility called ExecuteTest.m has been created for SoftSim6D. This utility provides two essential functions; compilation of a new real-time C application and the loading of the application and parameters to the target machine.

The single most common error that occurs when modifying or generating a new model for use in SoftSim6D is a compilation error. This error does not necessarily occur because a Simulink model is not functional, rather because of either incorrect model parameters for integration with SoftSim6D or the use of MATLAB/Simulink features that are not auto-codable into C. A model may be developed and run correctly in Simulink, but if the provided generic blocks that automatically configure to SoftSim6D are not used or MATLAB C guidelines are not followed, the model may need to be updated. To verify that none of these errors will manifest themselves when it is attempted to generate and run a full simulation, the “compilation test” functionality of ExecuteTest was developed. When this test is run, if such an error exists, a red MATLAB error will appear which points to the problem of the source. If there are no errors, the test will simply display complete and exit the application. It is important to run this test frequently when making significant changes to any simulation. The error that is displayed can only identify a few errors at once and more may appear to exist when actually a single error occurs and is allowed to propagate through multiple levels. This can be avoided by frequently verifying the ability of the simulation to compile.

The second process provided by this utility compiles and executes the test. Before compilation, the application will ask the user for the desired SSIF and VDF files and then load these into the simulation. The user can also select whether a real-time or free run application will be generated. The simulation is then loaded onto the target machine and executed at the user’s command. All of these tasks can be conducted manually through MATLAB’s command line and the running/loading of various SoftSim6D configuration files, however ExecuteTest drastically expedites the process for the user. The user interface for ExecuteTest can be seen in Figure 27.



```
>> ExecuteTest

Welcome to the compilation/test interface interface!

Which model would you like to test?
Example: HighLevel_rev6
TestInterface>>HighLevel_rev6

Compile Test or Execution Test? (CT or ET)
TestInterface>>ET

Real-Time (RT) or Free-Run (FR)?
TestInterface>>RT

Store in standard compiled code directory? (Y/N)
TestInterface>>y

Current initialization file is set to LOAD_Init_Generic, continue with this file? (Y or N)
TestInterface>>y

Current model variant definition file is set to LOAD_ModelVariantDefinition, continue with this file? (Y or N)
TestInterface>>y
```

Figure 27: ExecuteTest User Interface

5.3.2. ConfigExec

As previously discussed, Simulink Real-Time has the capability that once a real-time application is loaded onto the target machine, certain parameters can be changed either before or during the execution of a simulation. In order for this to occur, the user must have a detailed knowledge of the file path, data type, block type, and setting type for a given parameter. The manual execution of this process is very time consuming and not practical for a large number of parameters. To expedite this process, ConfigExec was generated. After a simulation is generated and loaded onto the target machine, SoftSim6D has been designed such that nearly all parameters defined in the SSIF can be altered, preventing the need for recompilation which can take several minutes. Due to how data is stored and the simulation is compiled, model variants (defined in the VDF) and the sample time of host scopes cannot be altered without re-running ExecuteTest.

All initial conditions, environment parameters, and sensor/actuator parameters are purposely defined within the INIT block to enable the maximum utility of ConfigExec. Since they can be updated by a simple command, ConfigExec makes large scale Monte-Carlos analysis of nearly any parameter possible. No modifications are required to the simulation to enable this form of large scale testing, rather a basic script utilizing the functions described in this section can be created and implemented to remove the user completely from the process.

5.3.3. IO_Killer

While ConfigExec provides the functionality of updating parameters prior to testing, IO_Killer utilizes Simulink Real-Time's ability to update parameters during real-time test execution. IO_Killer provides the framework and is the first implementation of fault injection in SoftSim6D. IO_Killer is a custom Simulink Block that can be inserted between the CML and DTL. This block enables the user to simulate various data transmission errors that are fairly common in serial communication. When the command to implement a specific error is given, the block takes the incoming communication from the CML, corrupts it in the specified manner, and then sends it to the DTL for transmission.

Currently three types of fault injection are possible. These are all common errors that can occur in serial communication and it is therefore important to verify the test article's ability to identify and correct for them. The first is to simply disable outgoing communication with the test article. Commands from the test article are still processed and recorded in this mode, however, no data packet is transmitted back to the test article. This simulates the failure of a component. The second is to corrupt the outgoing data packet. When this mode is implemented, a packet of the correct size and with the correct termination byte is transmitted, however the contents are purposely corrupted to simulate a "bad" packet. This can be configured to either occur once (to simulate a temporary failure) or every time the test article requests information. Lastly, an error similar to a Distributed Denial of Service (DDOS) attack is possible. When this is activated, miscellaneous data is continuously streamed to the target device. This functionality, although unlikely to manifest itself in this manner is actuality, exercises the target device's ability to buffer, recognize, and dispose of undesirable data. The Simulink block and settings dialogue are shown in Figure 28 and Figure 29.

Although IO_Killer only deals specifically with serial communication faults, both the MATLAB real-time execution function and the corresponding Simulink block were written in a generic manner to provide the framework for any future fault injection developments.

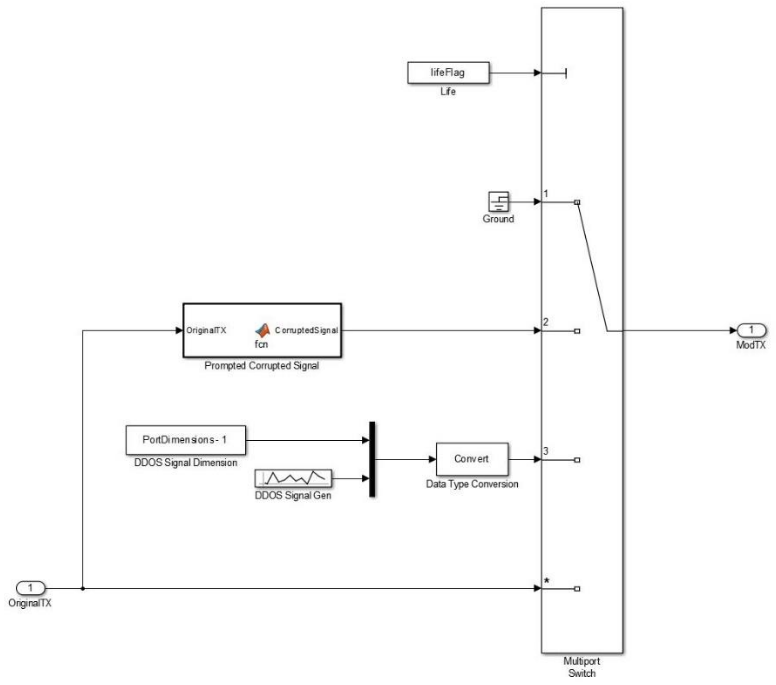
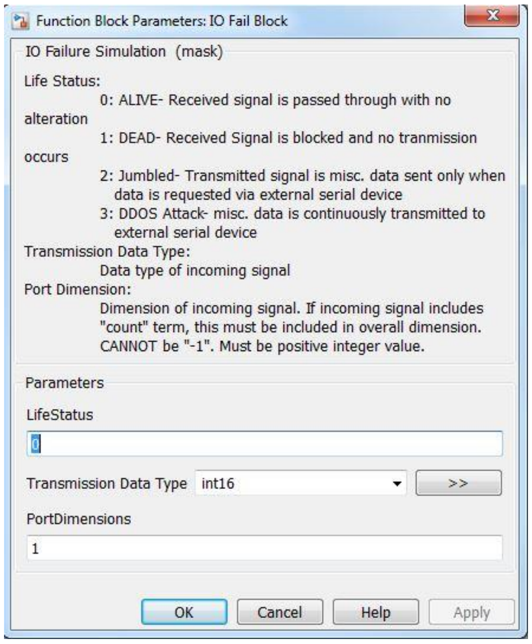


Figure 28: IO_Killer Simulink Implementation


Function Block Parameters: IO Fail Block

IO Failure Simulation (mask)

Life Status:
0: ALIVE- Received signal is passed through with no alteration
1: DEAD- Received Signal is blocked and no tranmission occurs
2: Jumbled- Transmitted signal is misc. data sent only when data is requested via external serial device
3: DDOS Attack- misc. data is continuously transmitted to external serial device
Transmission Data Type:
Data type of incoming signal
Port Dimension:
Dimension of incoming signal. If incoming signal includes "count" term, this must be included in overall dimension. CANNOT be "-1". Must be positive integer value.

Parameters

LifeStatus: 0

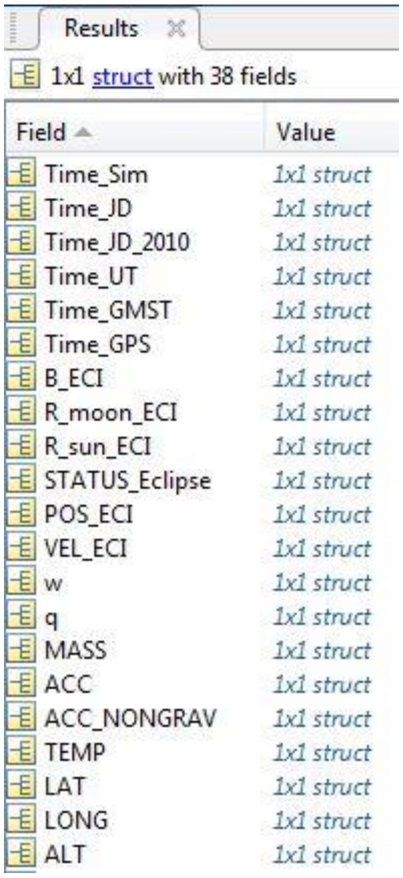
Transmission Data Type: int16

PortDimensions: 1

Figure 29: IO_Killer Settings Prompt

5.3.4. PlotRun

The high number of host scopes distributed across the standard simulation generates an extremely large quantity of data sets assigned to individual Simulink Host Scope handles. To allow the user to rapidly extract all of this data from the target computer, put it into a concise data structure and plot important qualities, PlotRun.m was created. After a simulation is completed, PlotRun is called and all data contained within the standard host scopes of the IBLE is saved in a structure called Results in the MATLAB workspace. A portion of this structure is shown in Figure 30.



1x1 struct with 38 fields

Field	Value
Time_Sim	1x1 struct
Time_JD	1x1 struct
Time_JD_2010	1x1 struct
Time_UT	1x1 struct
Time_GMST	1x1 struct
Time_GPS	1x1 struct
B_ECI	1x1 struct
R_moon_ECI	1x1 struct
R_sun_ECI	1x1 struct
STATUS_Eclipse	1x1 struct
POS_ECI	1x1 struct
VEL_ECI	1x1 struct
w	1x1 struct
q	1x1 struct
MASS	1x1 struct
ACC	1x1 struct
ACC_NONGRAV	1x1 struct
TEMP	1x1 struct
LAT	1x1 struct
LONG	1x1 struct
ALT	1x1 struct

Figure 30: Results Data Structure

PlotRun then provides the user the ability to plot certain common properties, specifically spacecraft state time-histories and sensor readings. The state plotting utility can be used universally across any simulation built from the IBLE, however the sensor plotting ability must be updated to reflect the specific configuration of the spacecraft under testing. Figure 31 shows the standard state plots generated by PlotRun during a validation simulation.

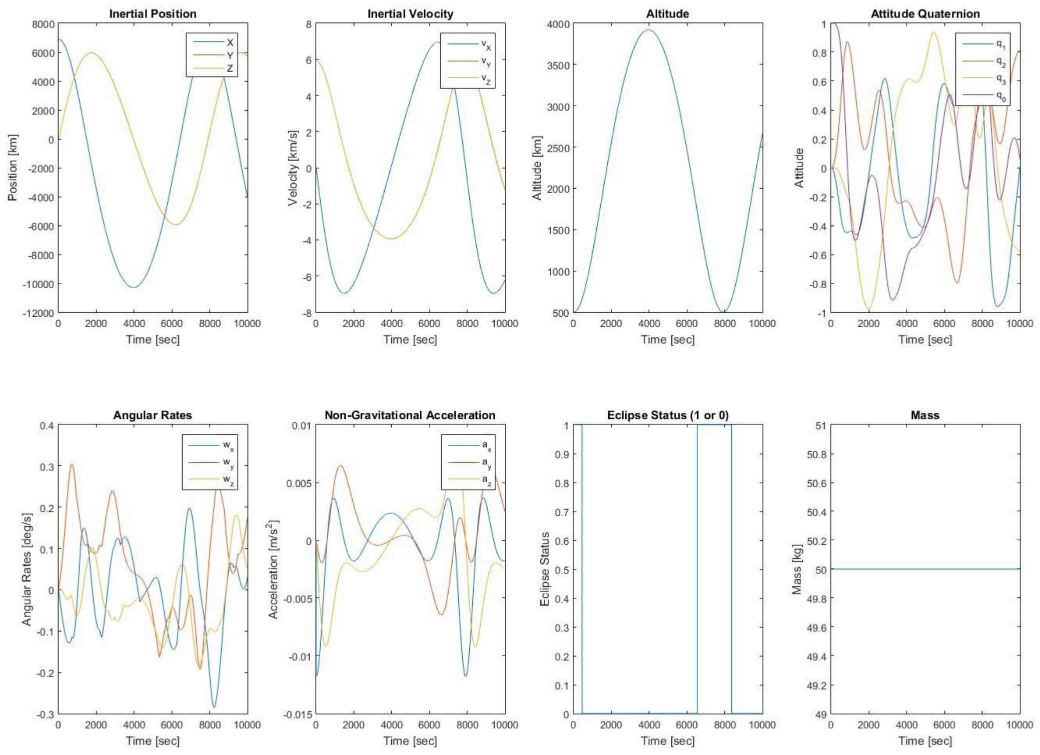


Figure 31: PlotRun Generated State Time-Histories

5.4. *Validation*

The dynamic and sensor simulation capabilities of SoftSim6D were verified through several means. Individual outputs of perturbation, environment, and sensor models were compared to various sources. Models that have counterparts in other already verified Georgia Tech simulation tools were run through various operational test cases and their outputs compared. This was the case for the attitude dynamics models as well as several sensor models. Subsystems or models that did not have another simulation verification source readily available were compared to test cases and other information found in relevant technical literature. Lastly, translational high fidelity dynamics results were compared to various simulations in Systems Tool Kit (STK). Since exact information on the various perturbation models and integration routines used in STK are not publically available, the parameters and relevant model variants were selected to be as close as possible to the inferred capabilities of STK's HPOP propagator. Table 2 and Table 3 show that SoftSim6D's dynamics simulation is within a kilometer of inertial position as predicted by STK and velocity is within a meter per second. Through this and extensive other means of verifying output data, the current release of SoftSim6D is considered to be fully validated.

Table 2: STK Verification Position Testing Results

Position Error	**Value [km]**
Average	0.93545
Standard Deviation	0.14701
Maximum	1.2343

Table 3: STK Verification Velocity Testing Results

Velocity Error	**Value [km/s]**
Average	0.00093758
Standard Deviation	0.00014316
Maximum	0.0012219

6. Hardware-in-the-Loop Testing

6.1. *Overview*

The most powerful capability of SoftSim6D is its ability to perform real-time hardware-in-the-loop testing. Common test set-ups only provide the ability to verify that avionics are able to properly communicate over designated serial protocols, not actively provide real-time simulated inputs that allow the user to "fly" the hardware. SoftSim6D is a compact, all-in-one solution to both verify hardware IO and exercise all on-board software through active simulation. Currently SoftSim6D has been verified to communicate over RS-232, RS-422, and RS-485 protocols as well as generate and read analog signals. This sections provides an overview of the verification of the Interface Layer components required to communicate over these protocols as well as a full HWITL test case of SoftSim6D.

6.2. *Verifying HWITL IO Capability*

The first step in implementing the interface layer for HWITL capabilities, specifically the DBM, CPM, and CTL, was the generation of a suitable test article. A Texas Instrument Tiva C Series microcontroller was programmed to act as a test avionics board (TAB) for the purposes of verifying serial communication. Initial testing focuses on the capabilities of the CTL and DBM to verify that data was successfully received and transformed into an actionable data type. The TAB was configured to send data packets of various lengths and data type over basic serial UART. A breadboard was then used in conjunction with the relevant low-level logic converters to transform the signal into either RS-232, RS-422, or RS-485 and then connected to the relevant IO port in the target machine. The output of the CTL and DBM were then recorded and analyzed to determine if the correct signal was received and properly interpreted.

To verify the data transmission capability of the CPM and CTL, specifically their ability to convert MATLAB data types into the necessary binary and then hexadecimal numbers, a reverse test set up was assembled. A test Simulink model containing only the CPM and CTL was created to generate and transmit data at a fixed interval through an IO card to the breadboard/TAB hardware. The TAB was then programmed to read and record the received data for verification. Initial testing focused on basis transmission of hexadecimal numbers, the foundation of most serial communication packets. Next, development and verification focused on transmission of numeric information of various integer and floating point data types. Testing discovered inconsistencies with Simulink Real-Time's stated binary conversion and communication transmission utilities. This led to the development of several custom functions (including ByteWriter2, discussed in Chapter 4) to ensure proper packing and transmission of all data types. Further testing using the TAB proved that the updated CPM and CTL properly transmitted data across all supported serial protocols and data types.

A final test configuration was set up using a standalone closed loop DBM, CPM, and CTL in conjunction with the TAB. The TAB was configured to send a basic set of commands to the target machine and the test simulation to respond with pre-set numeric responses. After this was successfully completed, the same test set up was configured to talk to over multiple IO ports using different command sequences simultaneously. A single TAB was set up to serve as the test article. The successful completion of this test, verified both the software and hardware capabilities of SoftSim6D to send, receive, and interpret data simultaneously over multiple IO ports.

The analog capabilities of the target machine were also verified. To verify reading analog inputs (analogous to analog commands from an avionics board), a power supply was used to simulate a voltage source. The power supply was manually stepped through different voltage settings while a standalone version of the analog CTL read the inputs. The generation of analog signals by the target machine (to simulate an analog sensor) was verified using a hand held digital multi-meter.

6.3. *Test Case: Modular Attitude Determination System*

6.3.1. Overview

The Modular Attitude Determination System (MADS) is a CubeSat avionics board currently under development by Marshall Space Flight Center (MSFC). The MADS board is designed to be highly configurable for various mission needs. It is based around an 80 MHz 32-bit ARM Cortex-M4F processor and can accommodate up to 8 UART devices, 6 I2C lines, 4 SPI lines and 4 A/D converters. The MADS board was designed such that it can be rapidly reproduced with connectors for new sensors while still utilizing the same hardware framework and software. Currently the board is configured for a Novatel GPS unit, EPSON MG350 IMU, Honeywell HMR2300 magnetometer, and the Wallops Island CRUQS sun sensor. The onboard software is configured to sample data and provide an inertial state and attitude solution at a rate of 1 Hz. In flight, this information will then be transmitted to the primary flight computer via one of its SPI lines.

6.3.2. Configured Emulation Layer

For testing of the MADS board four of the five components required in the emulation layer were already developed and verified as part of the initial library development efforts. These were the rate gyro, accelerometer, magnetometer, and GPS model blocks. The only custom block required was for the Wallops Island sun sensor as this had a custom measurement output. The Wallop Island block (denoted in Figure 32 as the DAVID Sun Sensor) was independently developed and verified then inserted into the simulation using the same process as any pre-existing blocks. All pre-existing blocks were simply inserted into the IBLE and their parameters updated in the associated initialization file. The custom emulation layer for the MADS board is shown in Figure 32.

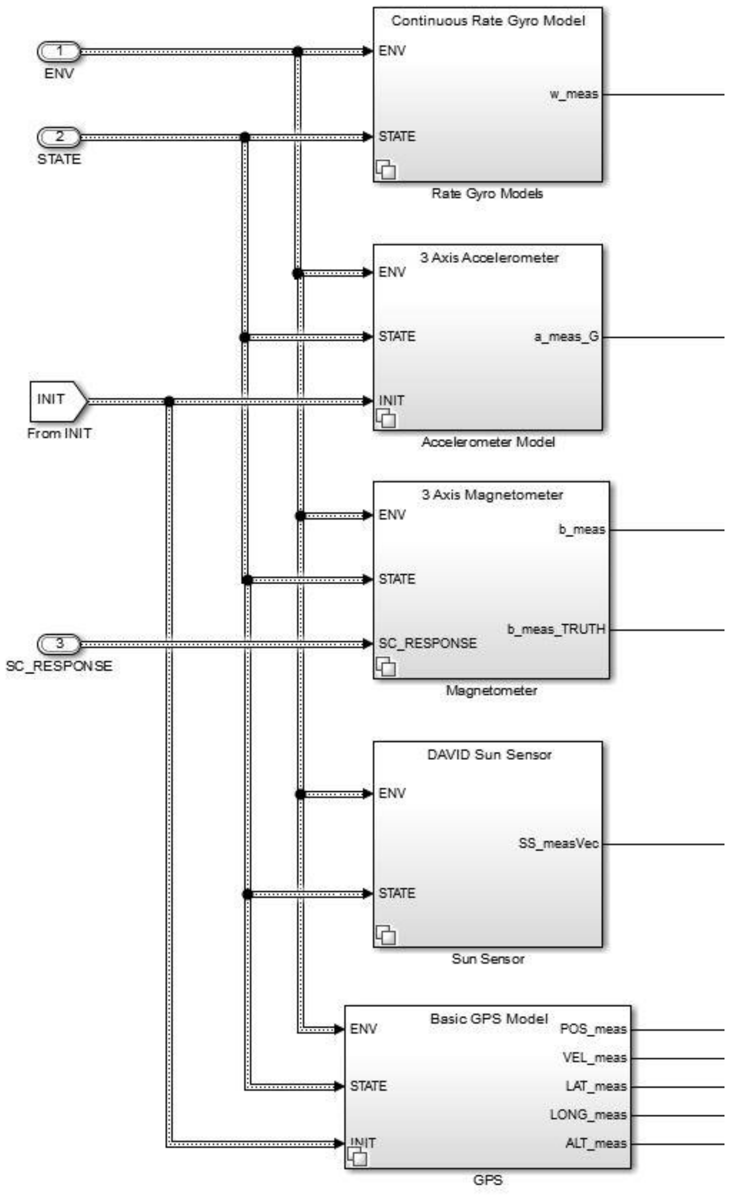


Figure 32: MADS Emulation Layer Implementation

6.3.3. Configured Interface Layer

The testing of the MADS board was the first complete implementation of closed loop communication between a target avionics board and SoftSim6D. The individual interface layer blocks for each component were configured to accept and respond accordingly to all possible commands that the MADS board was able to send. This included basic ping commands, re-setting of device ID commands, and querying of data. All command were individually verified with the MADS board in de-bug mode prior to testing. The MADS board has also successfully been able to communicate with all actual hardware, therefore we can say with confidence that SoftSim6D accurately emulated all sensors. This also was the first time fault injection was used in testing, by means of the IO_Killer Simulink libraries. The completed Data Preparation and Communication Management Layers are shown in Figure 33. The Data Transmission Layer is shown in Figure 34.

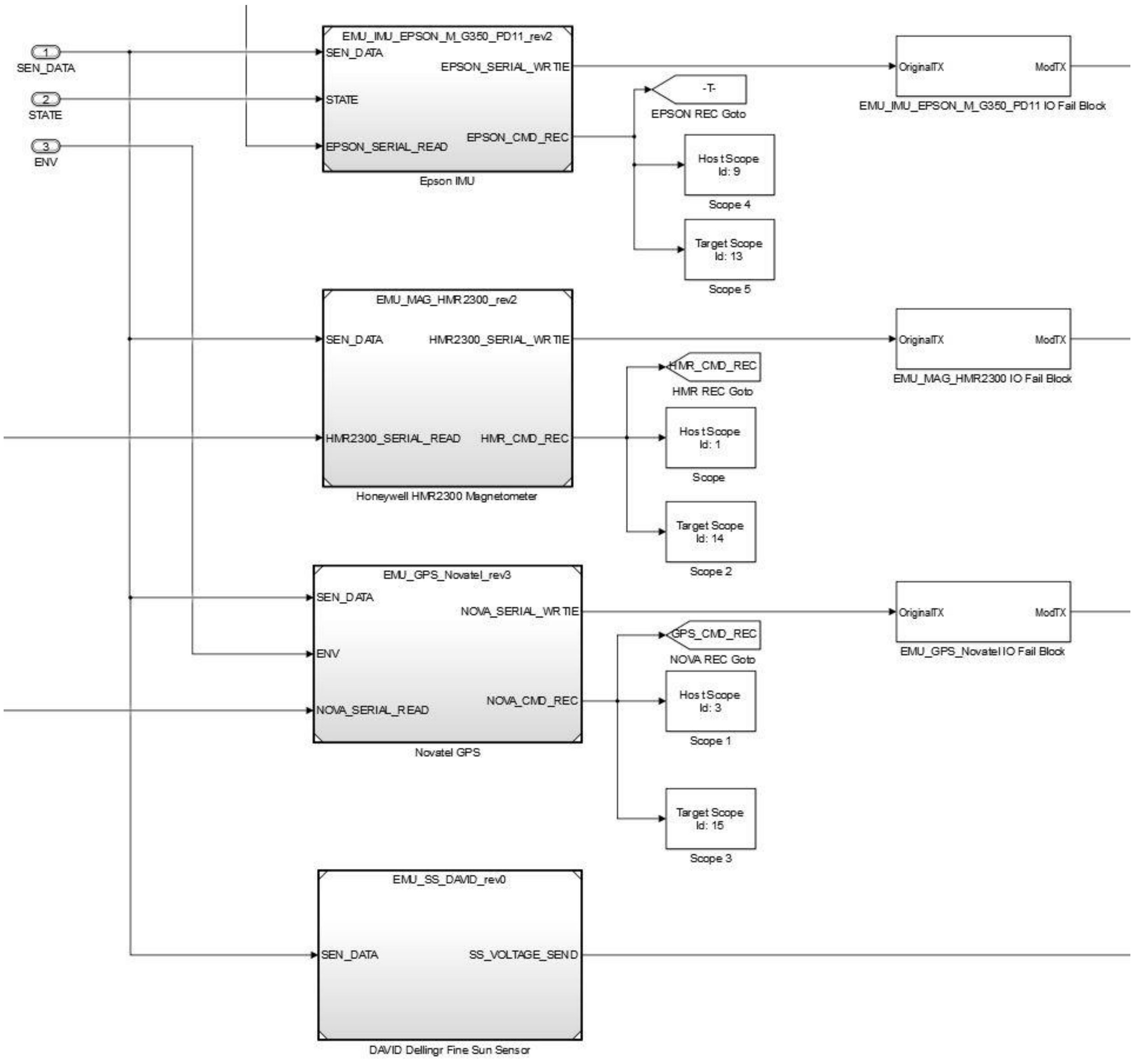


Figure 33: MADS Data Preparation and Communication Layer Implementation

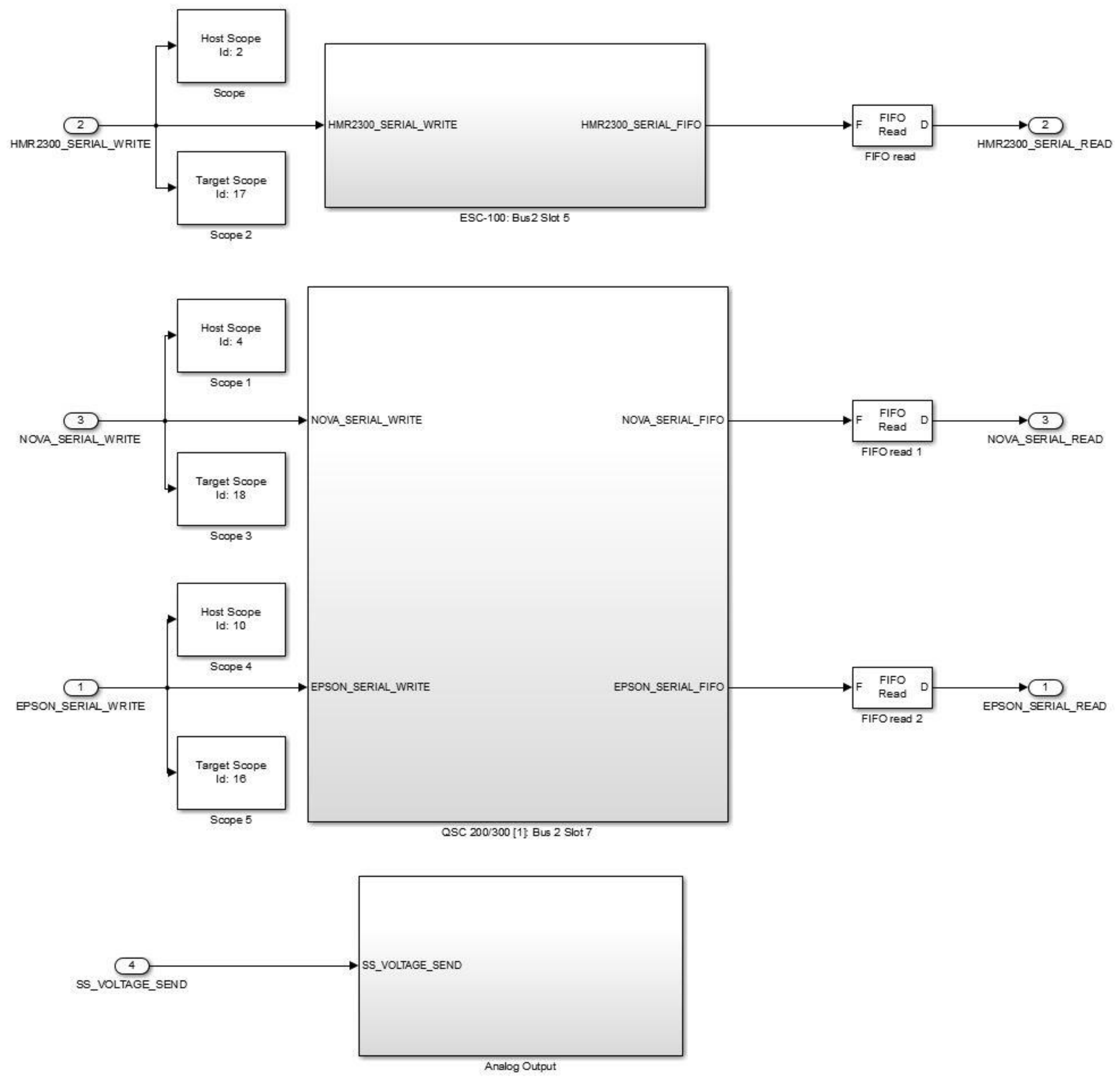


Figure 34: MADS Data Transmission Layer Implementation

6.3.4. Hardware Set-Up

To interface with the MADS board, a combination of custom harnessing and bread boarding were used. The HMR2300 magnetometer is a standard RS-232 signal so a basic harness converting SoftSim6D’s standard DB-9 connector to the board’s micro DB-9 connector was manufactured. The EPSON IMU and Novatel GPS are both designed to be board-mounted and communicate directly over UART. Therefore, to accommodate this, a breadboard was used in conjunction with DS8921N RS-485/422 to UART conversion chips to convert the RS-485 signal generated by

SoftSim6D to UART. Analog outputs from SoftSim6D utilized a custom harness which connected the ADACS card to the micro DB-25 connector on the board. For power, the MADS board requires a 5V and 3.3V power rail. A standard Arduino MEGA was used to regulate and supply this to the board. The hardware test set up of the MADS board is shown in Figure 35.

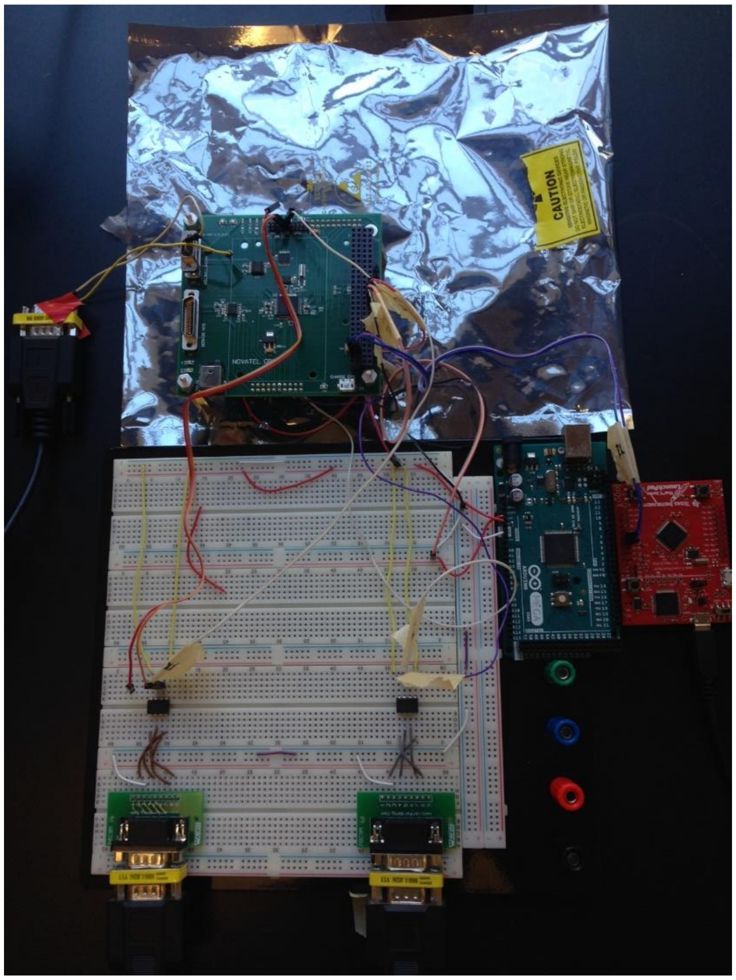
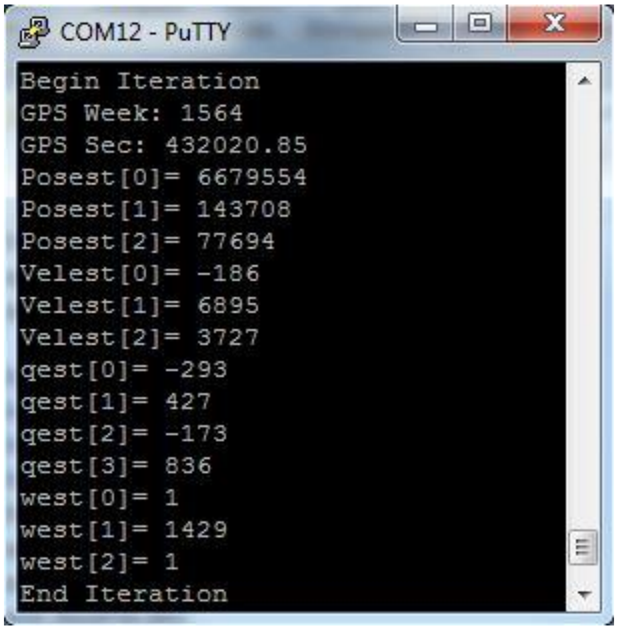


Figure 35: MADS HWITL Test Set Up

Lastly, for data monitoring, a Texas Instruments Tiva C Series microcontroller was connected to one of the MADS board's spare UART ports. This microcontroller was used to mimic the flight computer that MADS is sending its calculated state and attitude estimate to. This information was stored using the Tiva microcontroller for later post processing and comparison with the saved true states as determined by SoftSim6D. A sample of the computed information provided by the MADS board in real-time is shown in Figure 36.



COM12 - PuTTY

```
Begin Iteration
GPS Week: 1564
GPS Sec: 432020.85
Posest[0]= 6679554
Posest[1]= 143708
Posest[2]= 77694
Velest[0]= -186
Velest[1]= 6895
Velest[2]= 3727
qest[0]= -293
qest[1]= 427
qest[2]= -173
qest[3]= 836
west[0]= 1
west[1]= 1429
west[2]= 1
End Iteration
```

Figure 36: MADS Calculated Solution Output

6.3.5. Testing Results

Several rounds of testing were conducted to fully characterize the performance of the MADS board. As issues were found within the board, these bugs were communicated to MSFC engineers and updates were made before continuing on to the next round of testing. The first round of testing focused on short ten to sixty minute tests to verify the calculated state and attitude solution of the MADS board. The saved states from SoftSim6D were transferred from the target machine using PlotRun and then a custom data processing script was written to match the true inertial state of the spacecraft with the corresponding timestamped solution determined by the MADS board.

The translational state true and estimated error from a single ten minute test run are shown in Figure 37. It is important to note that an error of almost 150 meters in position knowledge occurs in this time frame. Such a deviation is significant as the noise of the GPS measurement is only on the order of tens of meters. This was the first error discovered through SoftSim6D testing. To characterize this issue, longer duration tests were conducted and revealed that the error was sinusoidal in nature. If a filter error on part of the MADS algorithms was the primary cause, it would be expected that this error would grow exponentially. This type of sinusoidal error points to a timing error between the true and estimated states. Ultimately it was confirmed that a timing error was the cause of the ~1 m/s velocity bias that induced the sinusoidal position error. The cause was a conversion from GPS Week and Second to fractional Julian days where a rounding error on the order of the seventh decimal place created a fraction of a second time bias. Since this rounding initially appears so insignificant to the user, this error would not have been found without SoftSim6D. The error was subsequently fixed and not witnessed in later testing.

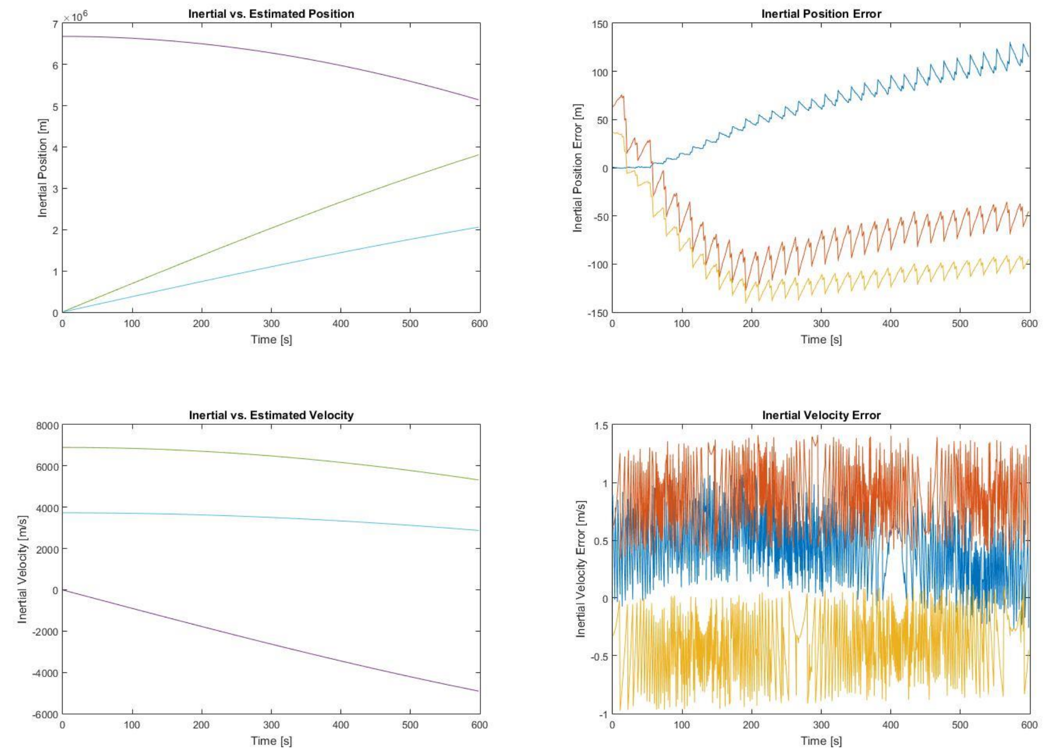


Figure 37: MADS Inertial Translational State Estimation Results

A sample angular velocity and attitude quaternion test result are shown in Figure 38 and Figure 39, respectively. Figure 38 demonstrates that the MADS algorithms are able to estimate spacecraft rate fairly well, estimating bias correctly and keeping overall noise within expected limits. The brief spikes in angular velocity error were determined to be caused be errors in spacecraft attitude estimation that resulted in momentary spikes in estimated rate bias. Figure 39 shows that the attitude solution provided by MADS is highly inaccurate, never converging to the correct solution. This, however, was expected. Although we are feeding sun sensor measurements to the MADS board over analog channels, at the time of this testing the MADS algorithms were not configured to directly use this information in attitude determination. Rather they were designed to use the digital output of the sun sensor provided over an SPI line. Since SoftSim6D currently does not have the hardware capability to communicate over this protocol, for this testing the MADS algorithms operated as if no sensor was active (the analog inputs were simply recorded but not used). The algorithms were not designed to function on magnetometers only for attitude determination, therefore no viable solution was found. To verify these results, tests were conducted on the standalone algorithms in MSFC's development framework assuming no sun sensor input and results were comparable. It was therefore accepted by MSFC engineers that the board was operating as expected under the conditions of a sun sensor failure.

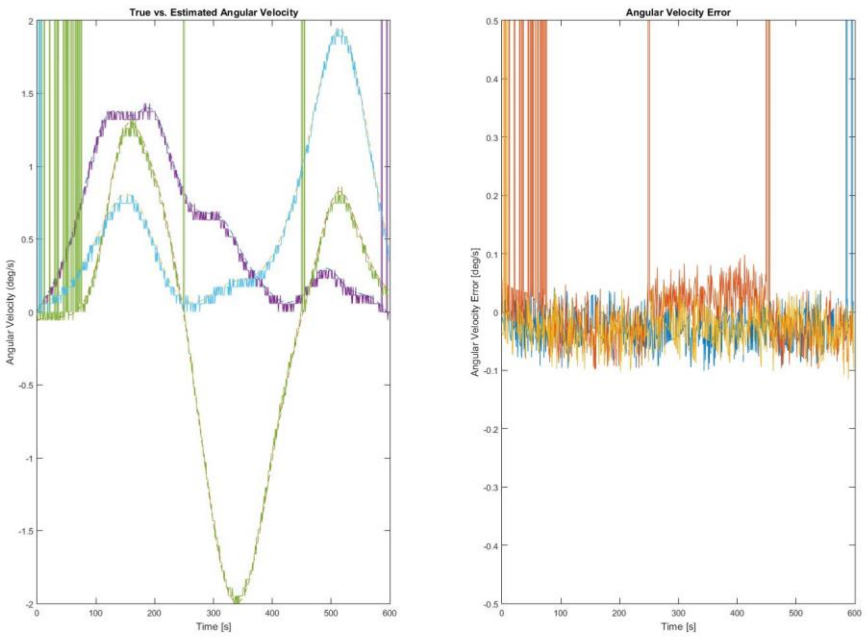


Figure 38: MADS Angular Velocity Estimation Test Results

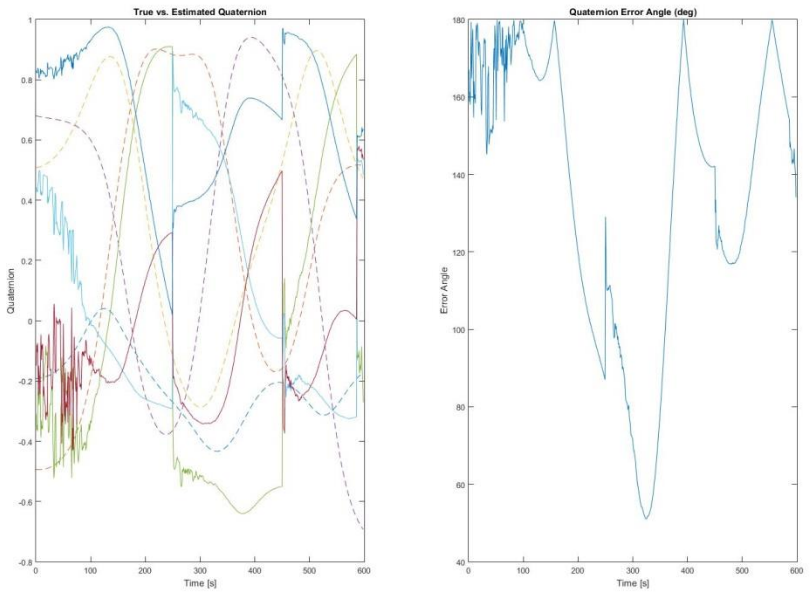


Figure 39: MADS Attitude Estimation Test Results

In addition to basic state and attitude estimation verification, the IO fault injection utilities of SoftSim6D were utilized to fully exercise the MADS board. Using the termination capability of IO_Killer, component boot-up order was evaluated. Testing revealed that if the MADS board was powered on before the GPS unit, the MADS algorithm were unable to find a solution for all time. The error source was determined to be the initialization of the state filter. When the filter initializes on boot-up, if there is no GPS information available, a divide by zero operation occurs resulting in an infinite solution. Once GPS data is available, the previous infinite states prevented any solution from being found. A patch was implemented to prevent this and was not witnessed again. No errors were induced by delayed boot-up of other components.

Both corrupted data injection and DDOS attacks were performed on the unit, with no errors detected. The MADS board performed as expected, rejecting corrupted data and re-requesting data until a valid packet was received. Intermittent component failures after board initialization were also tested. Components were temporarily failed for a period of seconds or minutes and then reactivated. The MADS algorithms performed as expected, simply propagating its last estimated solution until communication could be restored with the device. At the conclusion of IO testing and the subsequent software updates, no injected faults were able to trigger an unexpected error state.

Lastly, long duration testing was conducted to determine if any error states will manifest themselves in the MADS board over extended operations. Tests of 15, 18, and 24 hours were conducted. To accommodate this significant amount of information (the 15 hour test generated roughly 16 GB of data), file scopes were implemented to store data to the hard drive of the target machine. Substantial post-processing scripts were written to extract and process the file scope data as well as compare to the calculated states determined by the MADS board. Results showed that during long term testing, no variations from expected behavior occurred and there were no communication failures. In all, SoftSim6D testing was able to fully verify both the nominal and off-nominal capabilities of the MADS avionics board, within testing limitations.

7. Proximity Operations Scenario Mission Simulation

7.1. *Updated Capabilities*

Significant development efforts have been dedicated to developing the final expansion of the HWITL simulation environment; the capability of the environment to simulate multiple spacecraft simultaneously for support of testing proximity operations-based mission architectures (SoftSim6D-ProxOps). The primary challenge involved in expanding the environment to be capable of supporting multiple spacecraft was the modification of data structures and data handling routines to account for any number of primary spacecraft. Development of the initial simulation capability was focused on the implementation of "plug and play," rapidly adaptable libraries and data structures were intentionally designed to handle only a single spacecraft at a time to allow for easier testing and debugging. The framework required for a single spacecraft was leveraged as a baseline for the individual spacecraft within a constellation and then a wrapper structure to handle all information throughout the constellation was developed and tested. The new wrapper has been developed to handle any number of simulated spacecraft, with its only limitation being the hardware memory capacity of the target machine.

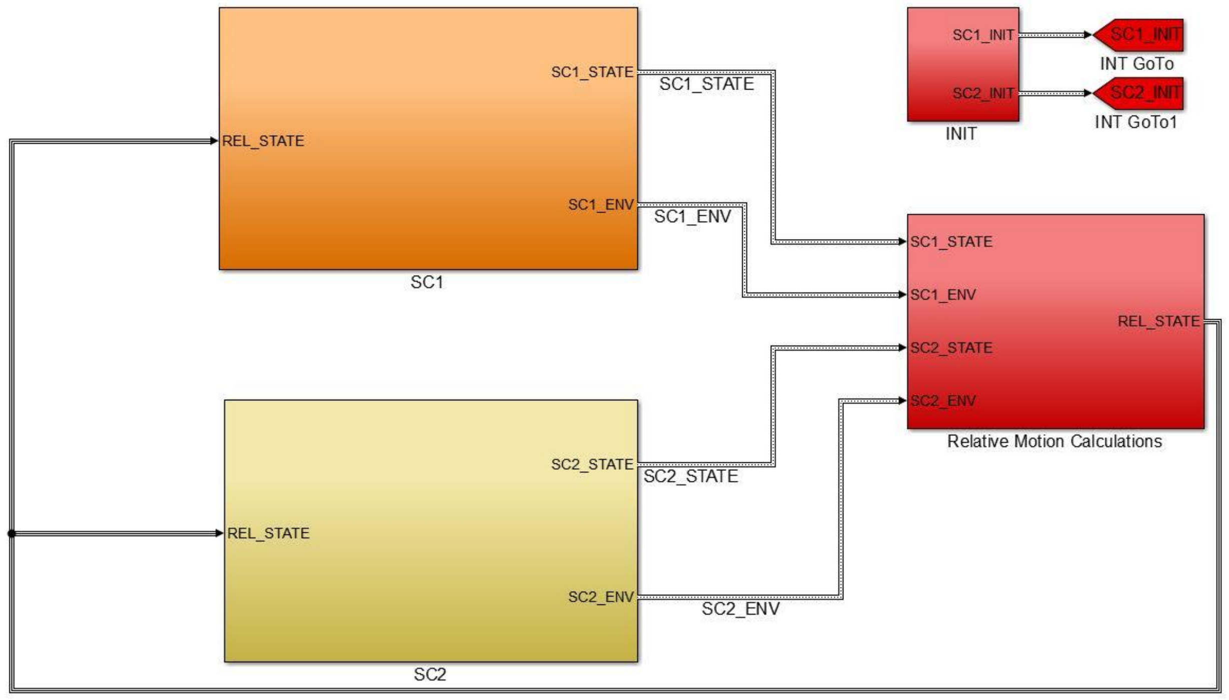


Figure 40: Proximity Operations SoftSim6D Implementation

After the data structure was completed and tested, a two spacecraft example scenario was implemented (shown in Figure 40). For the simulation of the spacecraft themselves, no new models had to be developed. The pre-existing dynamics models were used to implement both spacecraft and the new data structure wrapper allowed for individual configuration of each spacecraft. The existing libraries of sensors and actuators were also used for each spacecraft. Modifications were required to the sensor and actuator libraries to allow for multiple instances of the same component with different configuration parameters to be simulated simultaneously. For example, in the scenario where both spacecraft have the same general class of magnetometer except each have different performance characteristics. A general fix to the issue was determined and applied across libraries. This has also been added to the generic models used to develop future sensors/actuators to prevent the issue from reoccurring.

In the Proximity Operations environment, the components typically found in the highest level of the IBLE are now found within each spacecraft block. The only block from the original single spacecraft IBLE that is reserved for the high level simulation is now the INIT block. In addition to the two spacecraft blocks, shown in Figure 40, a new high level block was created for multiple spacecraft scenarios. This block, known as the Relative Motion Calculation subsystem, is responsible for calculating the relative inertial states and any desired relative reference frames between the two spacecraft. The information generated in this block is then fed into each individual spacecraft model. This data is made available to each spacecraft for use in the calculation of any relative sensor data, such as images, beacon readings, laser ranging readings, etc.

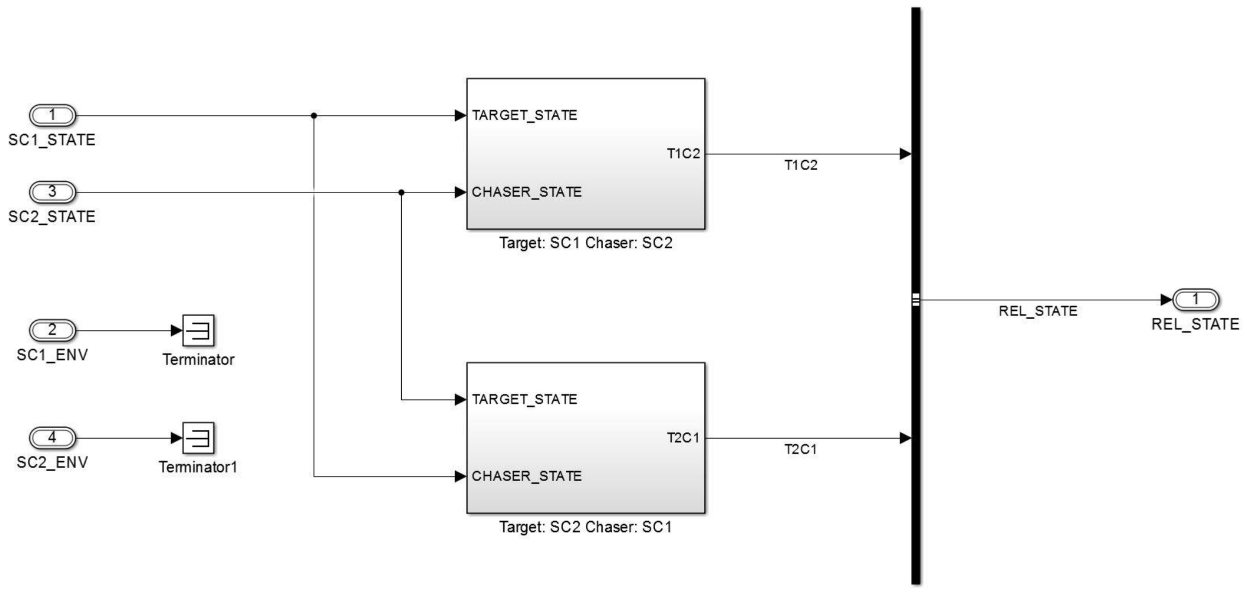


Figure 41: Proximity Operations Relative Motion Calculation Subsystem

To aid in testing and verification of the expanded environment, new utilities were constructed to manage, extract, and plot data for multi-spacecraft simulations. Updated versions of ExecuteTest (ExecuteTest_ProxOps), ConfigExec (ConfigExec_MultipleSC), and PlotRun (PlotRun_Multiple) were generated. These are all configured for the two spacecraft case, however they have been designed such that they can be expanded to any number of spacecraft without significant user development.

7.2. *Testing and Expandability Limitations*

To verify the integrated two spacecraft Proximity Operations example, two individual single spacecraft scenarios were run using the original IBLE. The outputs of these simulations were compared to the results of the Proximity Operations test case and were verified to be identical. Several scenarios were examined and all results verified as consistent with the single spacecraft equivalent simulation. An example of two spacecraft being simulated simultaneously is shown in Figure 42 and Figure 43.

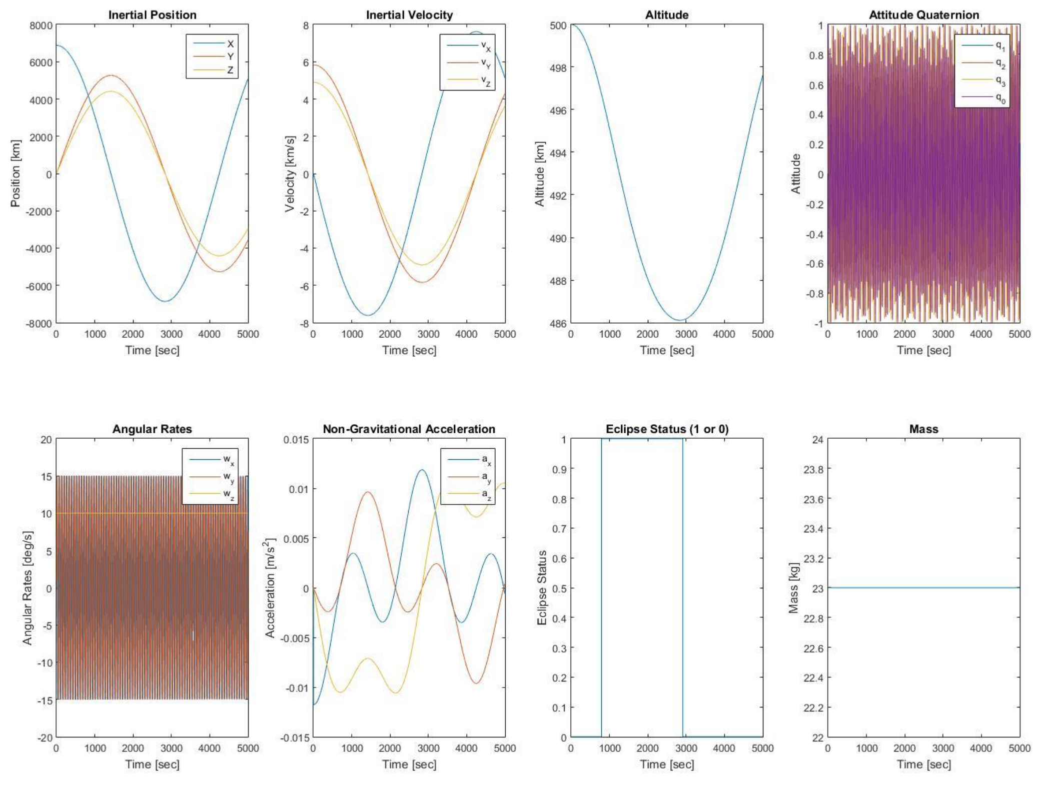


Figure 42: Proximity Operations Test Case: Spacecraft 1 Inertial State

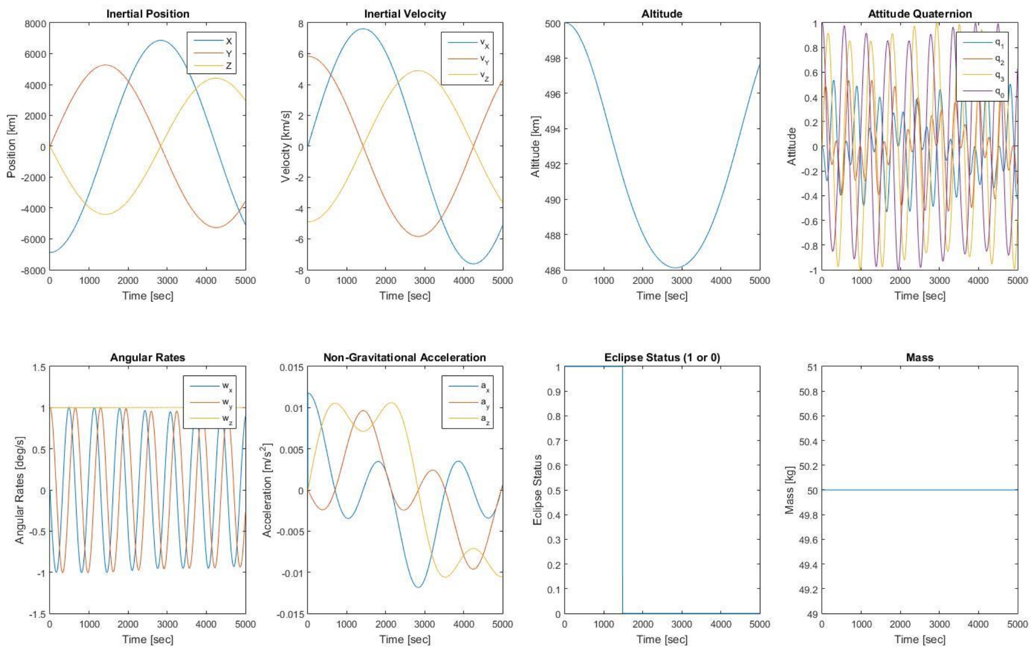


Figure 43: Proximity Operations Test Case: Spacecraft 2 Inertial State

Testing to date has verified the nominal performance of SoftSim6D-ProxOps as well as the associated support functions. The only limitations for the number of total spacecraft to be simulated arise from the hardware limitations of the target machine. This includes processor power, memory allowance, and the number of IO ports. The current configuration of the target machine should be capable of simulating several spacecraft simultaneously without issues, however if large constellations (on the orders of tens of spacecraft) are to be tested, hardware upgrades may be required.

Currently all updated SoftSim6D-ProxOps data structures and utilities have been configured for two spacecraft. Modification will be required for larger constellations, although such modifications are a simple expansion of the current configuration. A user's guide has been developed to walk the user through this process.

8. Current Development and Forward Work

8.1. *Martian System Simulation: SoftSim6D-Mars*

In support of development work currently underway at Georgia Tech for the development of a CubeSat constellation in Martian orbit, a new variant of SoftSim6D was developed, known as SoftSim6D-Mars. Softsim6D-Mars leverages the framework of the original IBLE and applies it to the Martian system. Environment, perturbation, and dynamics models have been updated to use Mars as the primary central body. Models within the 6DOF simulation capability include the Martian atmosphere, solar radiation pressure, spherical harmonics, gravity gradient torques, and third-body effects from the sun, Phobos, and Deimos. The same sensor and actuator libraries developed for the IBLE are utilized for the emulation layer. The data bus structures and storage routines are directly inherited from the IBLE. All original support utilities are also compatible with the SoftSim6D-Mars IBLE. Similar to original verification work, SoftSim6D-Mars was verified through comparison with STK simulations. Errors were comparable to those of the Earth centered IBLE.

Although initial functionality has been proven, updates to SoftSim6D-Mars are suggested to fully utilize its potential. Currently the initial states of third bodies (i.e. the sun, Phobos, and Deimos) are calculated via STK for the desired starting Epoch and fed into an initialization function. To speed up testing, the development of an initialization script to automatically determine the initial states of these bodies is desirable. Also only a basic exponential atmospheric density model is used for aerodynamic drag calculations. This should be sufficient for high altitude missions, however the development of a new density block may be desirable for certain scenarios. Lastly, the current configuration of SoftSim6D-Mars is a single spacecraft variant. Updates will be required to simulate a constellation of small spacecraft in Martian orbit. The steps laid out by the SoftSim6D user's guide for expanding the basic Earth simulation for additional spacecraft can be followed to perform such an operation.

8.2. *Generic Central Body Implementation: SoftSim6D-Universal*

In the future, it is highly probable that small satellite architectures will be utilized for either heliocentric missions or concepts based around other large bodies in the solar system. It is for this reason that it would be practical to update SoftSim6D to be generic for any central body. A library of environment and perturbation models would need to be generated for each specific central body, however a generic capability would at minimum allow for two body dynamics about any major planetary object in the solar system. Such a capability is oftentimes sufficient for initial trade studies and mission developments. Primary body specific models could then be developed for missions that will undergo further development.

When SoftSim6D-Mars was developed, initial strides were made towards this end. Planet specific parameters such as radius and gravitational parameters were made generic throughout the entire framework, configurable from the initialization file. The INIT data structure was also updated to account for varying central bodies. Further updates to initialization scripts and data structures are needed to make SoftSim6D-Mars rapidly configurable but no significant additional changes to the framework should be necessary.

8.3. *Additional IO Capability*

Currently, SoftSim6D can only communicate over the standard RS-232, RS-422, and RS-485 protocols and generate/read basic analog signals. To increase the number of devices SoftSim6D is able to emulate, it is desirable to purchase additional IO cards to expand the capabilities of the target machine. Specifically IO cards capable of I2C and SPI communication. Unlike UART protocols, I2C and SPI communication can vary slightly from device to device. Therefore, IO cards capable of adapting to a specific component's needs are rather complex. To work across a wide range of components, customized IO cards are most likely required and the associated MATLAB drivers. Mathworks has several licensed vendors who can provide these cards, but at a cost that typically runs into the thousands of dollars. Lower cost solutions do exist and can be adapted when absolutely necessary, however they are may not be adaptable to all configurations. It is therefore recommended that investment be made in the higher cost custom IO cards to provide a robust solution for SoftSim6D's needs.

9. Conclusion

To date, SoftSim6D has been proven to achieve all of its primary requirements with the potential to be further expanded to accommodate an even wider range of capabilities. The rapid adaptability and versatility of SoftSim6D should considerably reduce the amount of time required for future projects to develop a high fidelity simulation environment. Initial algorithm testing capability can be achieved on the order of days instead of weeks and HWITL testing capabilities can be ready within weeks of testing requirement definition. Once the baseline implementation of SoftSim6D was completed, the short timeline required to prepare for testing of the MSFC MADS avionics board serves as proof of the system's capabilities. Recent modifications now allow for the simultaneous testing of multiple spacecraft and missions in the Martian system. Well defined configuration management and coding procedures in addition to instruction manuals also make SoftSim6D a very user friendly environment.

Through extensive development planning, implementation, and verification efforts, an extremely robust and adaptable small spacecraft testing environment has been created. SoftSim6D has proven to be a versatile environment that will drastically increase the reliability of future Georgia Tech small satellite missions. With the conclusion of this master's project, initial operability of SoftSim6D has been achieved and the framework is ready for use in conjunction of full flight missions.

References

[1] NASA Engineering and Safety Center, "Demonstration of Autonomous Rendezvous Technology Mishap Review Board RP-06-119," NASA, 2006.

[2] R. B. Friend, "Orbital Express Program Summary and Mission Overview," in *SPIE Sensors and Systems for Space Applications II*, Orlando, FL, 2008.

[3] S. MacGillivray, "Proximity Operations Nano-Satellite Flight Demonstration (PONSFD) Overview," in *10th Annual CubeSat Developers Workshop 2013*, San Luis Obispo, CA, 2013.

[4] S. Chait and D. Spencer, "Prox-1: Automated Trajectory Control For On-Orbit Inspection," in *AAS Guidance, Navigation, and Control Conference*, Breckenridge, CO, 2013.

[5] N. K. Ure, Y. Kaya and G. Inalhan, "The Development of a Software and Hardware-in-The-Loop Test System for ITU-PSAT II Nano Satellite ADCS," in *IEEE Aerospace Conference*, Big Sky, MT, 2011.

[6] A. Ptak and K. Foundy, "Real-Time Spacecraft Simulation and Hardware-in-the-Loop Testing," in *IEEE Real-Time Technology and Applications Symposium*, Denver, CO, 1998.

[7] Y. Yang and X. Cao, "Design and Development of the Small Satellite Attitude Control System Simulator," in *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Keystone, CO, 2006.

[8] J. Debbler, J. J. Davis, J. Valasek and J. L. Junkins, "Mobile Robotic System for Ground Testing of Multi-Spacecraft Proximity Operations," in *AIAA Modelling and Simulation Technologies Conference and Exhibit*, Honolulu, HI, 2008.

[9] B. e. a. Kelm, "FREND: Pushing the Envelope of Space Robotics," Naval Research Laboratory, 2008.

[10] S. Montenegro, S. Janhichen and O. Maibaum, "Simulation-Based Testing of Embedded Software in Space Applications," *Embedded Systems Modelling - Technology, and Applications*, pp. 73-82, 2006.

[11] S. Tashakkor and J. Molina-Fraticelli, "SPRITE-HIL," NASA Marshall Space Flight Center, 2013, 2013.

[12] S. Zhaowei, X. Guodong, L. Xiaohui and C. Xibin, "The Integrated System for Design, Analysis, System Simulation, and Evaluation of the Small Satellite," *Advances in Engineering Software*, vol. 31, no. 7, pp. 437-443, 2000.

[13] "Satellite Missions Database," ESA, 2014. [Online]. Available: https://directory.eoportal.org/image/image_gallery?img_id=168595&t=1338044414980. [Accessed 10 November 2014].